Svyatoslav Kulikov

Using MySQL, MS SQL Server, and Oracle

BY EXAMPLES

2nd EDITION

© EPAM Systems, RD Dep.

Using MySQL, MS SQL Server, and Oracle by Examples

Practical Guide
for Software Engineers and Testers

(2nd edition)

Book version 2.0.31 as of 2025-07-07

Table of Contents

FOREWORD **4**

CHAPTER 1: MODEL DESCRIPTION, EXPORT AND FILLING THE DATABASE **6**

1.1. GENERAL DESCRIPTION OF THE DATABASE MODEL **6**

1.2. DATABASE MODEL FOR MYSQL **8**

1.3. DATABASE MODEL FOR MS SQL SERVER **10**

1.4. DATABASE MODEL FOR ORACLE **12**

1.5. DATABASE EXPORT AND FILLING **14**

CHAPTER 2: QUERIES TO SELECT AND MODIFY DATA **19**

2.1. DATA SELECTION FROM A SINGLE TABLE **19**

2.1.1. EXAMPLE 1: ALL DATA SELECTION 19
2.1.2. EXAMPLE 2: DISTINCT DATA SELECTION 20
2.1.3. EXAMPLE 3: COUNT FUNCTION AND ITS PERFORMANCE 23
2.1.4. EXAMPLE 4: COUNT FUNCTION WITH CONDITIONS 31
2.1.5. EXAMPLE 5: SUM, MIN, MAX, AVG FUNCTIONS USAGE 33
2.1.6. EXAMPLE 6: ORDERING QUERY RESULTS 37
2.1.7. EXAMPLE 7: USING COMPOUND CONDITIONS 41
2.1.8. EXAMPLE 8: GETTING MINIMUM AND MAXIMUM VALUES 45
2.1.9. EXAMPLE 9: CALCULATING AVERAGE VALUES OF AGGREGATED DATA 58
2.1.10. EXAMPLE 10: DATA GROUPING 64

2.2. DATA SELECTION FROM MULTIPLE TABLES **69**

2.2.1. EXAMPLE 11: USING JOINS TO OBTAIN HUMAN-READABLE DATA 69
2.2.2. EXAMPLE 12: USING JOINS WITH COLUMNS-TO-ROWS TRANSFORMATION 74
2.2.3. EXAMPLE 13: JOINS AND SUBQUERIES WITH IN 86
2.2.4. EXAMPLE 14: NON-TRIVIAL CASES OF JOINS AND SUBQUERIES WITH IN 96
2.2.5. EXAMPLE 15: DOUBLE SUBQUERIES WITH IN 101
2.2.6. EXAMPLE 16: JOINS WITH COUNT 106
2.2.7. EXAMPLE 17: JOINS WITH COUNT AND AGGREGATION FUNCTIONS 119
2.2.8. EXAMPLE 18: MULTIPLE AND COMPOUND CONDITIONS 135
2.2.9. EXAMPLE 19: JOINS WITH MIN, MAX, AVG, RANGES 140
2.2.10. EXAMPLE 20: EACH AND EVERY JOIN VARIETY IN THREE DBMSES 159

2.3. DATA MODIFICATION **195**

2.3.1. EXAMPLE 21: DATA INSERTION 195
2.3.2. EXAMPLE 22: DATA UPDATE 203
2.3.3. EXAMPLE 23: DATA DELETION 206
2.3.4. EXAMPLE 24: DATA MERGING 208
2.3.5. EXAMPLE 25: CONDITIONAL DATA MODIFICATION 213

CHAPTER 3: USING VIEWS **223**

3.1. USING VIEWS TO SELECT DATA **223**

3.1.1. EXAMPLE 26: USING NON-CACHING VIEWS TO SELECT DATA 223
3.1.2. EXAMPLE 27: USING CACHING VIEWS AND TABLES TO SELECT DATA 229
3.1.3. EXAMPLE 28: USING VIEWS TO OBSCURE DATABASE STRUCTURES AND DATA VALUES 255

3.2. USING VIEWS TO MODIFY DATA **259**

3.2.1. EXAMPLE 29: USING UPDATABLE VIEWS TO MODIFY DATA 259
3.2.2. EXAMPLE 30: USING TRIGGERS ON VIEWS TO MODIFY DATA 271

CHAPTER 4: USING TRIGGERS 286

4.1. USING TRIGGERS TO AGGREGATE DATA 286

4.1.1. EXAMPLE 31: USING TRIGGERS TO UPDATE CACHING TABLES AND FIELDS 286
4.1.2. EXAMPLE 32: USING TRIGGERS TO ENSURE DATA CONSISTENCY 306

4.2. USING TRIGGERS TO CONTROL DATA OPERATIONS 329

4.2.1. EXAMPLE 33: USING TRIGGERS TO CONTROL DATA MODIFICATION 329
4.2.2. EXAMPLE 34: USING TRIGGERS TO CONTROL DATA FORMAT AND VALUES 352
4.2.3. EXAMPLE 35: USING TRIGGERS TO CORRECT DATA ON-THE-FLY 359

CHAPTER 5: USING STORED FUNCTIONS AND STORED PROCEDURES 367

5.1. USING STORED FUNCTIONS 367

5.1.1. EXAMPLE 36: USING STORED FUNCTIONS FOR DATA OPERATIONS 367
5.1.2. EXAMPLE 37: USING STORED FUNCTIONS FOR DATA CONTROL 385

5.2. USING STORED PROCEDURES 390

5.2.1. EXAMPLE 38: USING STORED PROCEDURES TO EXECUTE DYNAMIC QUERIES 390
5.2.2. EXAMPLE 39: USING STORED PROCEDURES FOR PERFORMANCE OPTIMIZATION 403
5.2.3. EXAMPLE 40: USING STORED PROCEDURES TO MANIPULATE DATABASE OBJECTS 415

CHAPTER 6: USING TRANSACTIONS 423

6.1. USING IMPLICIT AND EXPLICIT TRANSACTIONS 423

6.1.1. EXAMPLE 41: MANAGING IMPLICIT TRANSACTIONS 423
6.1.2. EXAMPLE 42: MANAGING EXPLICIT TRANSACTIONS 434

6.2. UNDERSTANDING TRANSACTIONS CONCURRENCY 443

6.2.1. EXAMPLE 43: MANAGING TRANSACTIONS ISOLATION LEVEL 443
6.2.2. EXAMPLE 44: CONCURRENT TRANSACTIONS INTERACTION 449
6.2.3. EXAMPLE 45: MANAGING TRANSACTIONS IN TRIGGERS, STORED FUNCTIONS AND PROCEDURES 480

CHAPTER 7: SOLVING TYPICAL TASKS AND PERFORMING TYPICAL OPERATIONS 489

7.1. OPERATIONS WITH HIERARCHICAL AND LINKED DATA 489

7.1.1. EXAMPLE 46: MANAGING HIERARCHICAL STRUCTURES 489
7.1.2. EXAMPLE 47: MANAGING GRAPH STRUCTURES 507

7.2. OPERATIONS WITH DATABASES 533

7.2.1. EXAMPLE 48: DATABASE BACKUP AND RESTORE 533

7.3. OPERATIONS WITH DBMS 539

7.3.1. EXAMPLE 49: USER MANAGEMENT, DBMS START AND STOP 539
7.3.2. EXAMPLE 50: CHARSET MANAGEMENT 542

7.4. USEFUL OPERATIONS WITH DATA 545

7.4.1. EXAMPLE 51: DATE OPERATIONS 545
7.4.2. EXAMPLE 52: ACQUIRING NON-REPEATABLE UNIQUE VALUES 554
7.4.3. EXAMPLE 53: WORKING WITH JSON 561
7.4.4. EXAMPLE 54: WORKING WITH PIVOT DATA 573
7.4.5. EXAMPLE 55: SUGGEST YOUR EXAMPLE 585

CHAPTER 8: BRIEF COMPARISON OF MYSQL, MS SQL SERVER, ORACLE 586

CHAPTER 9: LICENSE AND DISTRIBUTION 590

Foreword

Many thanks to colleagues from EPAM Systems for their valuable comments and recommendations during the preparation of the material.

My special thanks go to the thousands of readers who have sent in questions, suggestions, and comments — your input has made the book better.

Since the first edition was published, the book has undergone numerous revisions based on readers' feedback and the author's reconsideration of certain ideas and formulations. As a result of questions from readers and discussions at training sessions, it became possible to clarify and smooth out controversial points, refine definitions, and provide explanations where it has proven necessary. The ideal is unattainable, but it is good to believe that a great step has been made in its direction.

This book is dedicated to the practice of using SQL to solve typical problems. It does not cover the theory of relational databases (assuming that you are familiar with it or can find the missing information[1]), but it does cover over 500 SQL queries, from basic "SELECTs" to the use of views, triggers, stored procedures, and functions.

All examples are presented as a problem statement and its solution using MySQL, MS SQL Server, and Oracle, and are accompanied by explanations and analysis of typical mistakes.

This material will primarily be useful to those who:

- used to study databases, but has forgotten a lot;
- has experience with one DBMS, but wants to switch to another;
- wants to learn how to write typical SQL queries in a very short time.

All solutions are made on MySQL Community Server 8.x, Microsoft SQL Server Express 2019, Oracle 18c Express Edition and are likely to work successfully even on newer versions of these DBMS, **but not on older ones** (in those cases where alternative solutions exist for older versions, they are also given).

In most solutions with complex logic the algorithm is explained on the example of MySQL, and for the other two DBMSes only the code is given with some comments, so it is advisable to consider solutions for all three DBMSes, even if you are interested in only one of them.

Source materials (schemas, scripts, etc.) are available at this link:
https://svyatoslav.biz/database_book_download/src.zip

The conventions used in this book are as follows:

Problem statement. Let's note at once that almost every considered example contains several problems. Since in some cases the solution of the problem will be placed far from its formulation, a {reference} to the solution will be given next to the problem number in curly brackets.

[1] You can start with the "Relational Databases by Examples" book (by Svyatoslav Kulikov) [https://svyatoslav.biz/relational_databases_book/]

Expected result. First, we will see what we should get by solving the problem correctly, and then we will describe the solution.

Problem solution. The solution will always be given for MySQL, MS SQL Server, and Oracle. Sometimes all solutions will be similar, sometimes very different. Since the solution to the problem will in some cases be placed far from its initial problem formulation, a {reference} to the problem statement will be given next to the solution number in curly brackets.

Exploration. This is also a problem, but not to get some data, it is to check how the DBMS behaves in some conditions.

Caution. We will consider typical mistakes and give explanations of their causes and consequences.

Task for self-study. It is highly recommended that you do these tasks. Even if it seems very easy to you. If, on the contrary, it seems to you that everything is very complicated — first try to work through the examples already solved, referring to the ready solution only for self-checking.

The course material is structured in such a way that it can be studied both sequentially and as a quick reference book (all the necessary explanations in the text are referenced).

In addition to the text of this book, we recommend a free online course containing a series of video lessons, tests, and tasks for self-study.

Let's begin!

Chapter 1: Model Description, Export and Filling the Database

1.1. General Description of the Database Model

We will use three databases to solve problems and review examples: “Library”, “Big Library (for experiments)”, and “Exploration”. The “Exploration” database will consist of many different tables needed to demonstrate the features of DBMS behavior, and we will form it gradually as we conduct the experiments.

The “Library” and “Big Library (for experiments)” databases models are completely identical (the only difference is the number of records). There are a total of seven tables:

- **`genres`** — genres of literature:
  - **`g_id`** — genre’s identifier (number, primary key);
  - **`g_name`** — genre’s name (string);
- **`books`** — books in the library:
  - **`b_id`** — book’s identifier (number, primary key);
  - **`b_name`** — book’s name (string);
  - **`b_year`** — book’s publication year (number);
  - **`b_quantity`** — the number of copies of a book in the library (number);
- **`authors`** — authors of books:
  - **`a_id`** — author’s identifier (number, primary key);
  - **`a_name`** — author’s name (string);
- **`subscribers`** — readers (subscribers) of the library:
  - **`s_id`** — reader’s identifier (number, primary key);
  - **`s_name`** — reader's name (string);
- **`subscriptions`** — facts of the delivery and return of books (so-called “subscriptions”):
  - **`sb_id`** — subscription’s identifier (number, primary key);
  - **`sb_subscriber`** — reader’s (subscriber’s) identifier (number, foreign key);
  - **`sb_book`** — book’s identifier (number, foreign key);
  - **`sb_start`** — book’s delivery date (subscription start) (date);
  - **`sb_finish`** — scheduled book’s return date (date);
  - **`sb_is_active`** — subscription’s activity flag (contains the **`Y`** value if the book is still being held by the reader, and **`N`** value if the book has already been returned to the library);
- **`m2m_books_genres`** — a service table to organize the “many to many” relationship between the **`books`** and **`genres`** tables:
  - **`b_id`** — book’s identifier (number, foreign key, part of the composite primary key);
  - **`g_id`** — genre’s identifier (number, foreign key, part of the composite primary key);
- **`m2m_books_authors`** — a service table to organize the “many to many” relationship between the **`books`** and **`authors`** tables:
  - **`b_id`** — book’s identifier (number, foreign key, part of the composite primary key);
  - **`a_id`** — author’s identifier (number, foreign key, part of the composite primary key).

In `m2m_books_genres` and `m2m_books_authors` tables both foreign keys are included in the composite primary key to prevent the situation when, e.g., a book belongs to a certain genre more than once (such errors lead to incorrect operation of queries like “calculate how many genres the book belongs to”).

The general database schema is shown in figure 1.a. You can find the database generation scripts for each DBMS in the source material, the link to which is given in the foreword[{4}].

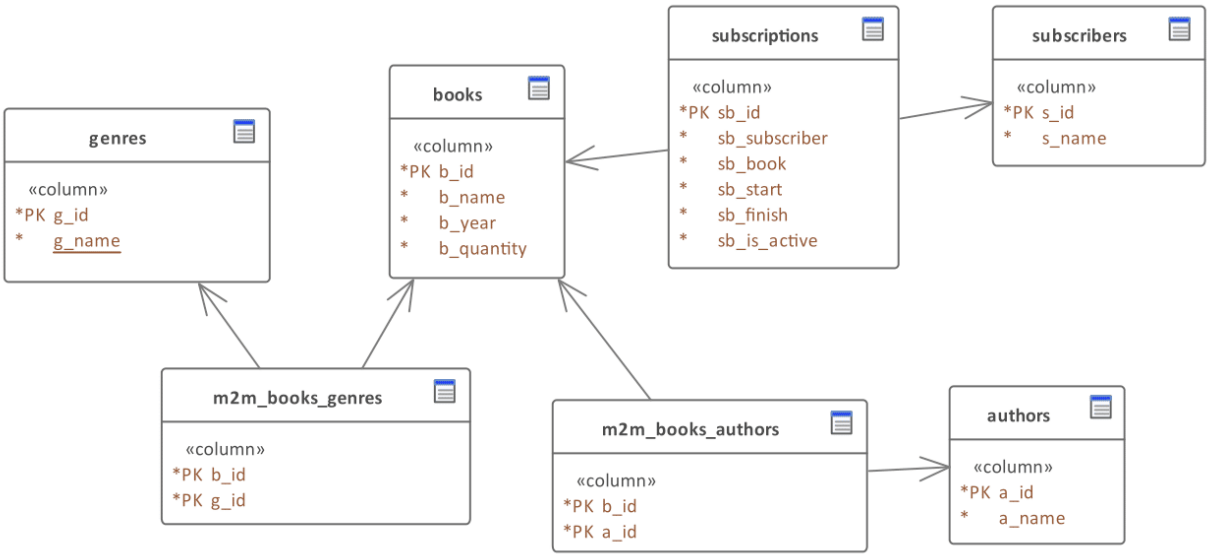


Figure 1.a — General Database Model

1.2. Database Model for MySQL

The database model for MySQL is shown in figures 1.b and 1.c. Note the following important points:

- The primary keys are represented by unsigned integers to extend the maximum range of values.
- Strings are of `VARCHAR` type up to 150 characters long (to ensure that the limit of 767 bytes per index length is met; 150*4 = 600, as MySQL in “worst case scenarios” aligns UTF string characters to a length of four bytes per character in comparison operations). This limitation can be bypassed in MySQL version 8 (and newer) using the InnoDB storage engine and `DYNAMIC` or `COMPRESSED` record formats, but for compatibility reasons we will not take the risk and will still use strings of 150 characters.
- The `sb_is_active` field is represented by the MySQL-specific `ENUM` data type (which allows selecting one of the specified values and is very convenient for storing a predefined set of values — `Y` and `N` in our case).
- The `g_name` field is marked as `UNIQUE` because the existence of genres with the same name is not allowed.
- The `sb_start` and `sb_finish` fields are represented by the `DATE` type (rather than the more complete `DATETIME` type), because we store the date of delivery and return of the book with a day accuracy.

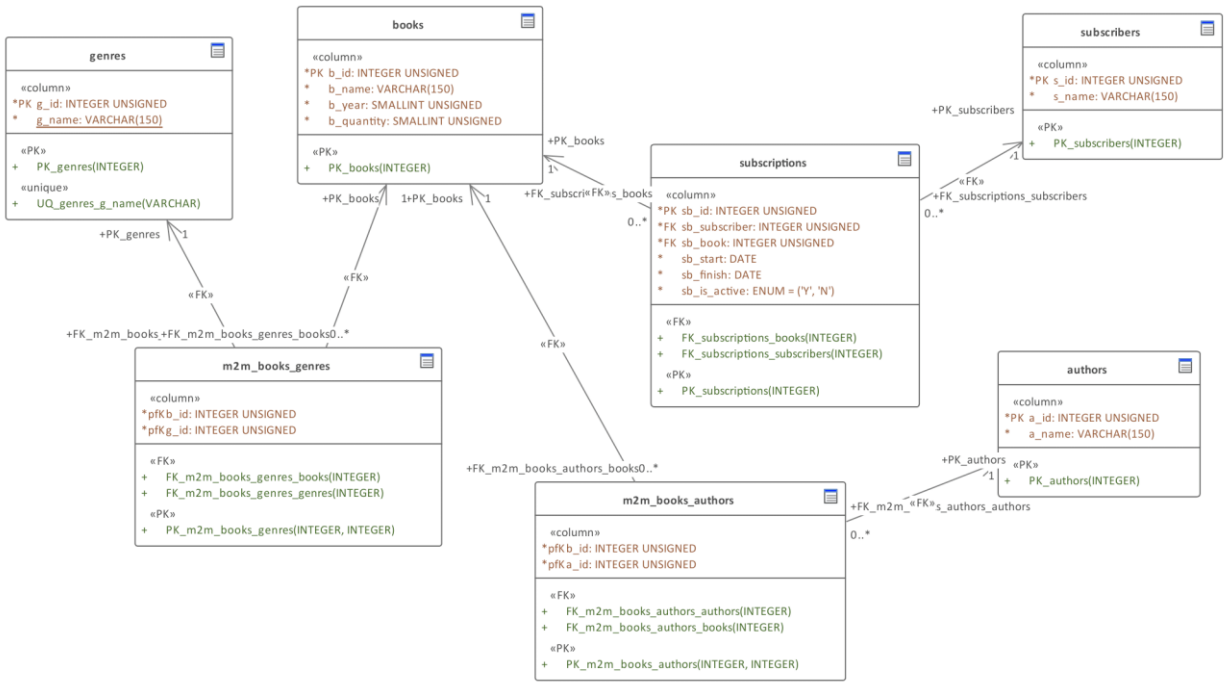


Figure 1.b — Database Model for MySQL (in Sparx EA)

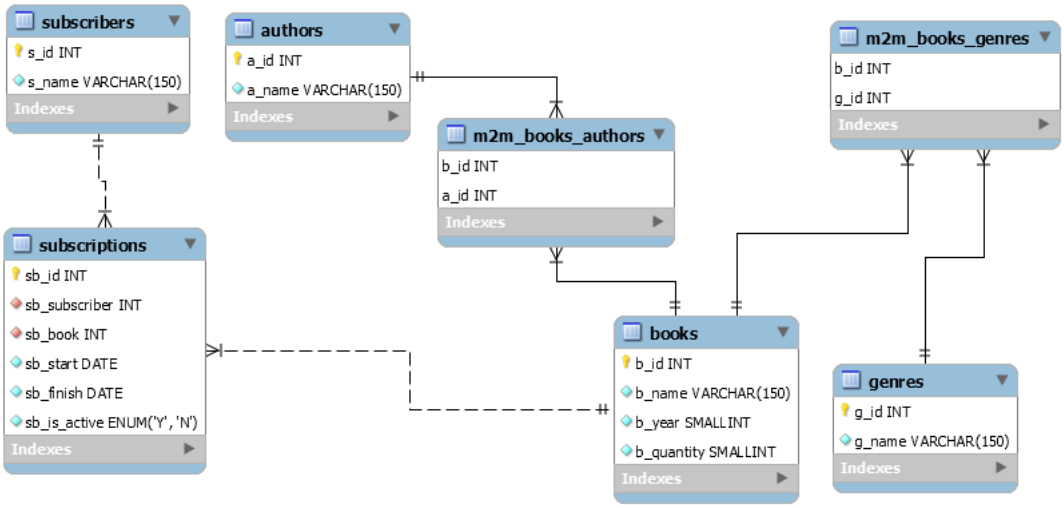


Figure 1.c — Database Model for MySQL (in MySQL Workbench)

1.3. Database Model for MS SQL Server

The database model for MS SQL Server is shown in figures 1.d and 1.e. Note the following important points:

- Primary keys are represented by signed integers, because MS SQL Server does not allow **`BIGINT`**, **`INT`** and **`SMALLINT`** to be **`UNSIGNED`** (yet **`TINYINT`**, on the contrary, is **`UNSIGNED`** only).
- Strings are represented by **`NVARCHAR`** type with length up to 150 characters (both for analogy with MySQL, and to guarantee the limit of 900 bytes per index length; 150 * 2 = 300, as MS SQL Server uses two bytes per character to store and compare characters in national encodings).
- The **`sb_is_active`** field is represented by **`CHAR`** data type with one character length (because MS SQL Server does not have **`ENUM`** data type), and to comply with MySQL analogy; this field is restricted by **`CHECK`** constraint with the condition "**`[sb_is_active] IN ('Y', 'N')`**".
- As in MySQL, the **`sb_start`** and **`sb_finish`** fields are represented by the **`DATE`** data type (rather than the more complete **`DATETIME`** type), because we store the date of delivery and return of the book with a day accuracy.

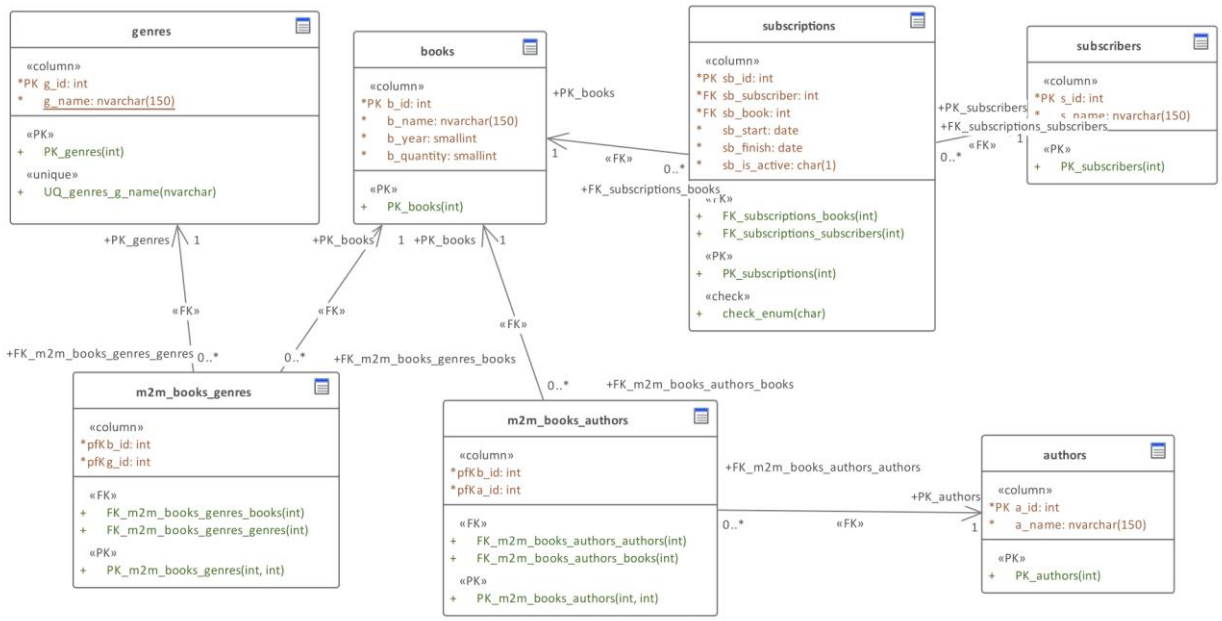


Figure 1.d — Database Model for MS SQL Server (in Sparx EA)

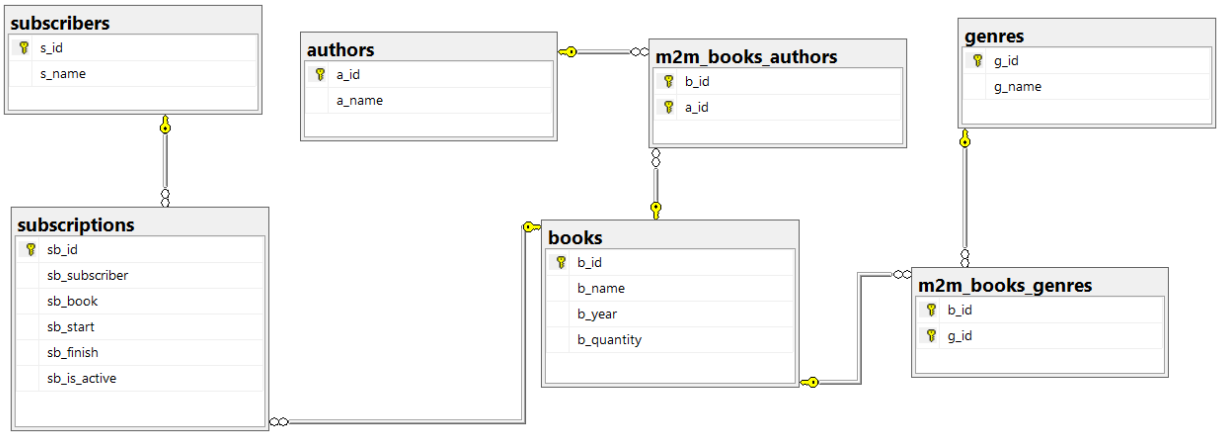


Figure 1.e — Database Model for MS SQL Server
(in MS SQL Server Management Studio)

1.4. Database Model for Oracle

The database model for Oracle is shown in figures 1.f and 1.g. Note the following important points:

- Integer fields are represented by the `NUMBER` data type with length specification, as this is the easiest way to emulate `INTEGER` data type from other DBMSes.
- Strings are represented by `NVARCHAR2` data type with the length up to 150 characters (both for analogy with MySQL and MS SQL Server, and to guarantee the limit of 758–6498 bytes per index length; 150*2 = 300, as Oracle uses two bytes per character to store and compare characters in national encodings, and the maximum index length can depend on different conditions, but the minimum is 758 bytes).
- The `sb_is_active` field is represented by `CHAR` data type with one character length (as Oracle does not have `ENUM` data type), and to comply with MySQL analogy, the same solution as in MS SQL Server is applied: a `CHECK` constraint with condition `"sb_is_active" IN ('Y', 'N')` is applied to this field.
- For the `sb_start` and `sb_finish` fields we chose `DATE` as the "simplest" date storage data type available in Oracle. Yes, it still stores hours, minutes, and seconds, but we can put zero values there.

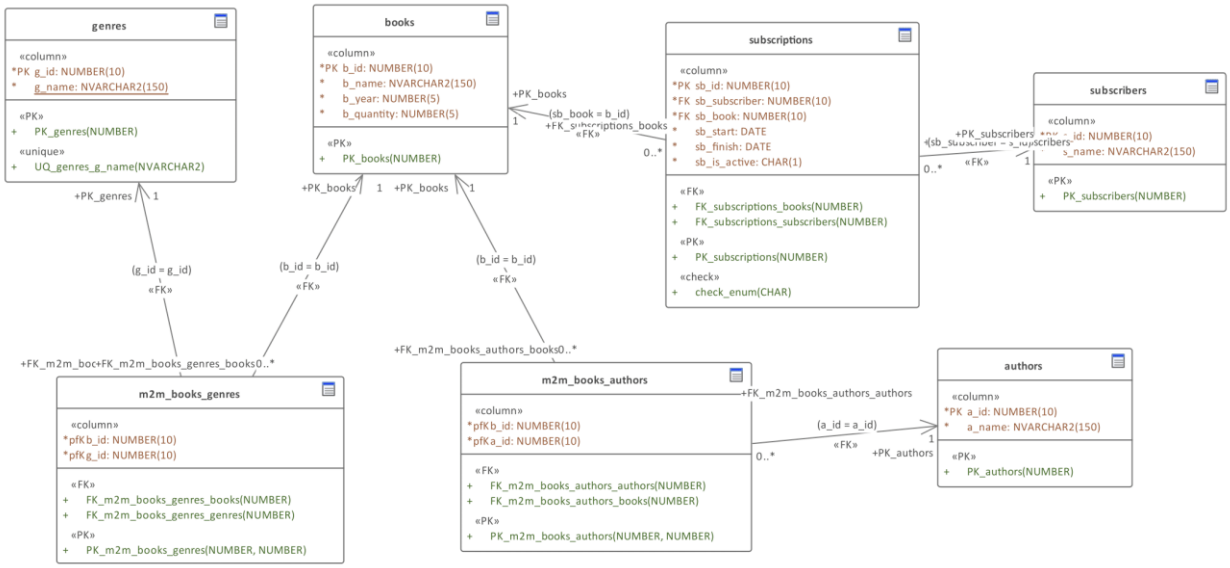


Figure 1.f — Database Model for Oracle (in Sparx EA)

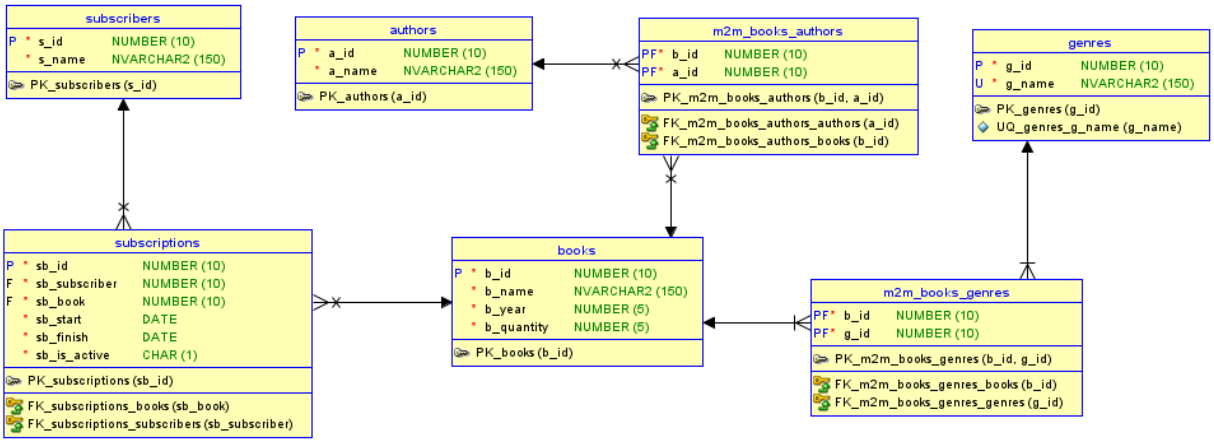


Figure 1.g — Database Model for Oracle
(in Oracle SQL Developer Data Modeler)

1.5. Database Export and Filling

Based on the models created in Sparx Enterprise Architect, let's get DDL scripts[4] for each database and execute them to create databases themselves (see the Sparx EA documentation for information about how to get a database export script based on a database model).

Let's fill the existing databases with the following data.

The `books` table:

b_id	b_name	b_year	b_quantity
1	Eugene Onegin	1985	2
2	The Fisherman and the Golden Fish	1990	3
3	Foundation and Empire	2000	5
4	Programming Psychology	1998	1
5	The C++ Programming Language	1996	3
6	Course of Theoretical Physics	1981	12
7	The Art of Computer Programming	1993	7

The `authors` table:

a_id	a_name
1	Donald Knuth
2	Isaac Asimov
3	Dale Carnegie
4	Lev Landau
5	Evgeny Lifshitz
6	Bjarne Stroustrup
7	Alexander Pushkin

The `genres` table:

g_id	g_name
1	Poetry
2	Programming
3	Psychology
4	Science
5	Classic
6	Science Fiction

The `subscribers` table:

s_id	s_name
1	Ivanov I.I.
2	Petrov P.P.
3	Sidorov S.S.
4	Sidorov S.S.

The presence of two subscribers with the name ("Sidorov S.S.") is not a mistake. We will need this variant of full namesakes to demonstrate a few typical mistakes in queries.

The `m2m_books_authors` table:

b_id	a_id
1	7
2	7
3	2
4	3
4	6
5	6
6	5
6	4
7	1

The `m2m_books_genres` table:

b_id	g_id
1	1
1	5
2	1
2	5
3	6
4	2
4	3
5	2
6	5
7	2
7	5

The `subscriptions` table:

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
100	1	3	2011-01-12	2011-02-12	N
2	1	1	2011-01-12	2011-02-12	N
3	3	3	2012-05-17	2012-07-17	Y
42	1	2	2012-06-11	2012-08-11	N
57	4	5	2012-06-11	2012-08-11	N
61	1	7	2014-08-03	2014-10-03	N
62	3	5	2014-08-03	2014-10-03	Y
86	3	1	2014-08-03	2014-09-03	Y
91	4	1	**2015-10-07**	**2015-03-07**	Y
95	1	4	2015-10-07	2015-11-07	N
99	4	4	**2015-10-08**	**2025-11-08**	Y

Strange combinations of book delivery and return dates (2015-10-07 / 2015-03-07 and 2015-10-08 / 2025-11-08) are added on purpose to further demonstrate the specifics of solving some problems.

Also note that among the readers who took the books, there is never a “Petrov P.P.” (with identifier 2), we will also need this in the future.

The disorder of entries by identifiers and “gaps” in the numbering of identifiers are also made deliberately to make it more realistic.

After inserting the data into the databases under all three DBMSes, it is worth checking to see if we have made a mistake somewhere. We perform the following queries, which will allow us to see data about the books and library operations in a human-readable form. A walk-through of these queries is presented in Example 11{69}, but for now let's just execute this code:

MySQL

```
SELECT `b_name`,
       `a_name`,
       `g_name`
FROM   `books`
       JOIN `m2m_books_authors` USING(`b_id`)
       JOIN `authors` USING(`a_id`)
       JOIN `m2m_books_genres` USING(`b_id`)
       JOIN `genres` USING(`g_id`)
```

MS SQL

```
SELECT [b_name],
       [a_name],
       [g_name]
FROM   [books]
       JOIN [m2m_books_authors]
         ON [books].[b_id] = [m2m_books_authors].[b_id]
       JOIN [authors]
         ON [m2m_books_authors].[a_id] = [authors].[a_id]
       JOIN [m2m_books_genres]
         ON [books].[b_id] = [m2m_books_genres].[b_id]
       JOIN [genres]
         ON [m2m_books_genres].[g_id] = [genres].[g_id]
```

Oracle

```
SELECT "b_name",
       "a_name",
       "g_name"
FROM   "books"
       JOIN "m2m_books_authors"
         ON "books"."b_id" = "m2m_books_authors"."b_id"
       JOIN "authors"
         ON "m2m_books_authors"."a_id" = "authors"."a_id"
       JOIN "m2m_books_genres"
         ON "books"."b_id" = "m2m_books_genres"."b_id"
       JOIN "genres"
         ON "m2m_books_genres"."g_id" = "genres"."g_id"
```

After running these queries, we will get a result showing how information from different tables is combined into meaningful sets:

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin	Classic
The Fisherman and the Golden Fish	Alexander Pushkin	Classic
Course of Theoretical Physics	Lev Landau	Classic
Course of Theoretical Physics	Evgeny Lifshitz	Classic
The Art of Computer Programming	Donald Knuth	Classic
Eugene Onegin	Alexander Pushkin	Poetry
The Fisherman and the Golden Fish	Alexander Pushkin	Poetry
Programming Psychology	Dale Carnegie	Programming
Programming Psychology	Bjarne Stroustrup	Programming
The C++ Programming Language	Bjarne Stroustrup	Programming
The Art of Computer Programming	Donald Knuth	Programming
Programming Psychology	Dale Carnegie	Psychology
Programming Psychology	Bjarne Stroustrup	Psychology
Foundation and Empire	Isaac Asimov	Science Fiction

Let's run another query to see human-readable information about who took what books from the library (see more about this query in Example 11{69}):

MySQL

```
SELECT `b_name`,
       `s_id`,
       `s_name`,
       `sb_start`,
       `sb_finish`
FROM   `books`
       JOIN `subscriptions`
         ON `b_id` = `sb_book`
       JOIN `subscribers`
         ON `sb_subscriber` = `s_id`
```

MS SQL

```
SELECT [b_name],
       [s_id],
       [s_name],
       [sb_start],
       [sb_finish]
FROM   [books]
       JOIN [subscriptions]
         ON [b_id] = [sb_book]
       JOIN [subscribers]
         ON [sb_subscriber] = [s_id]
```

Oracle

```
SELECT "b_name",
       "s_id",
       "s_name",
       "sb_start",
       "sb_finish"
FROM   "books"
       JOIN "subscriptions"
         ON "b_id" = "sb_book"
       JOIN "subscribers"
         ON "sb_subscriber" = "s_id"
```

The result of these queries is as follows:

b_name	s_id	s_name	sb_start	sb_finish
Eugene Onegin	1	Ivanov I.I.	2011-01-12	2011-02-12
The Fisherman and the Golden Fish	1	Ivanov I.I.	2012-06-11	2012-08-11
The Art of Computer Programming	1	Ivanov I.I.	2014-08-03	2014-10-03
Programming Psychology	1	Ivanov I.I.	2015-10-07	2015-11-07
Foundation and Empire	1	Ivanov I.I.	2011-01-12	2011-02-12
Foundation and Empire	3	Sidorov S.S.	2012-05-17	2012-07-17
The C++ Programming Language	3	Sidorov S.S.	2014-08-03	2014-10-03
Eugene Onegin	3	Sidorov S.S.	2014-08-03	2014-09-03
The C++ Programming Language	4	Sidorov S.S.	2012-06-11	2012-08-11
Eugene Onegin	4	Sidorov S.S.	2015-10-07	2015-03-07
Programming Psychology	4	Sidorov S.S.	2015-10-08	2025-11-08

Note that "Sidorov S.S." are actually two different people (with identifiers 3 and 4).

Let's also put the following number of automatically generated records into the "Big Library (for experiments)" database:

- in the **`authors`** table — 10'000 authors' names (duplicate names are allowed);
- in the **`books`** table — 100'000 books' titles (duplicate titles are allowed);
- in the **`genres`** table — 100'000 genres' names (all names are unique);
- in the **`subscribers`** table — 1'000'000 subscribers' names (duplicate names are allowed);
- in the **`m2m_books_authors`** table — 1'000'000 "author-book" pairs (co-authorship is allowed);
- in the **`m2m_books_genres`** table — 1'000'000 links "book-genre" pairs (multiple genres of books are allowed);
- in the **`subscriptions`** table — 10'000'000 records about deliveries and returns of books (the return always happens not before the delivery; the situations when the same reader took the same book many times are allowed — even without returning the previous copy).

During the experiments to evaluate query execution speed, all DBMS operations will be performed in unbuffered mode (when query results are not transferred to the application memory), and before each iteration (except for exploration 2.1.3.EXP.A[23]) the DBMS cache will be cleared as follows:

MySQL
```
1    RESET QUERY CACHE;
```

MS SQL
```
1    DBCC FREEPROCCACHE;
2    DBCC DROPCLEANBUFFERS;
```

Oracle
```
1    ALTER SYSTEM FLUSH BUFFER_CACHE;
2    ALTER SYSTEM FLUSH SHARED_POOL;
```

So, the database creation and filling is successfully completed and we can move on to solving the problems.

Chapter 2: Queries to Select and Modify Data

2.1. Data Selection from a Single Table

2.1.1. Example 1: All Data Selection

Problem 2.1.1.a{19}: show all information about all readers (subscribers).

Expected result 2.1.1.a.

s_id	s_name
1	Ivanov I.I.
2	Petrov P.P.
3	Sidorov S.S.
4	Sidorov S.S.

Solution 2.1.1.a{19}.

In this case, the simplest query — the classic `SELECT *` — is enough. For all three DBMSes, the queries are exactly the same.

MySQL Solution 2.1.1.a

```
1    SELECT *
2    FROM `subscribers`
```

MS SQL Solution 2.1.1.a

```
1    SELECT *
2    FROM [subscribers]
```

Oracle Solution 2.1.1.a

```
1    SELECT *
2    FROM "subscribers"
```

Task 2.1.1.TSK.A: show all information about all:

- authors;
- genres.

2.1.2. Example 2: Distinct Data Selection

Problem 2.1.2.a{20}: show without duplication the identifiers of readers who have ever borrowed books from the library.

Problem 2.1.2.b{21}: show a list of readers with the number of full namesakes for each name.

Expected result 2.1.2.a.

sb_subscriber
1
3
4

Expected result 2.1.2.b.

s_name	people_count
Ivanov I.I.	1
Petrov P.P.	1
Sidorov S.S.	2

Solution 2.1.2.a{20}.

To get a correct result, it is necessary to use the `DISTINCT` keyword which tells the DBMS to remove all duplicates from the result.

MySQL Solution 2.1.2.a

```
1    SELECT DISTINCT `sb_subscriber`
2    FROM `subscriptions`
```

MS SQL Solution 2.1.2.a

```
1    SELECT DISTINCT [sb_subscriber]
2    FROM [subscriptions]
```

Oracle Solution 2.1.2.a

```
1    SELECT DISTINCT "sb_subscriber"
2    FROM "subscriptions"
```

If the `DISTINCT` keyword is removed from the query, the result will be as follows (because the readers took the books several times each):

sb_subscriber
1
1
1
1
1
3
3
3
4
4
4

It is important to understand that the DBMS in the `DISTINCT` mode of query execution does not compare individual fields, but whole records, so if there is a difference between records at least by one field, they are not considered identical.

A typical mistake is trying to use `DISTINCT` "as a function". Suppose we want to see all the readers' names without duplications (remember, "Sidorov S.S." occurs twice), selecting not only the name but also the identifier. The following query is incorrect.

MySQL — Example of an incorrect query 2.1.2.ERR.A

```
1    SELECT DISTINCT(`s_name`),
2                   `s_id`
3    FROM   `subscribers`
```

MS SQL — Example of an incorrect query 2.1.2.ERR.A

```
1    SELECT DISTINCT([s_name]),
2                   [s_id]
3    FROM   [subscribers]
```

Oracle — Example of an incorrect query 2.1.2.ERR.A

```
1    SELECT DISTINCT("s_name"),
2                   "s_id"
3    FROM   "subscribers"
```

As a result of executing this query, we will get a result where "Sidorov S.S." is represented twice (the duplication is not eliminated) because these records have different identifiers (3 and 4), which does not allow the DBMS to consider the two corresponding result lines as matching. The result of executing query 2.1.2.ERR.A will be:

s_name	s_id
Ivanov I.I.	1
Petrov P.P.	2
Sidorov S.S.	3
Sidorov S.S.	4

A frequent question is: then how to solve this problem, how to show records without repeating rows that have fields with different values? In the general case, you can't, because the problem statement itself is wrong (we'll just lose some data).

If the problem is formulated somewhat differently (e.g., "show a list of readers with the number of full namesakes for each name"), then grouping and aggregation functions can help us. We will look into this question in detail later (see Example 10{64}), but for now let's solve the problem just mentioned, to show the difference in the way queries work.

Solution 2.1.2.b{20}.

Let's use the `GROUP BY` clause to group similar rows and the `COUNT` function to calculate the number of rows in each group.

MySQL — Solution 2.1.2.b

```
1    SELECT `s_name`,
2           COUNT(*) AS `people_count`
3    FROM   `subscribers`
4    GROUP  BY `s_name`
```

MS SQL Solution 2.1.2.b

```
1    SELECT [s_name],
2           COUNT(*) AS [people_count]
3    FROM   [subscribers]
4    GROUP  BY [s_name]
```

Oracle Solution 2.1.2.b

```
1    SELECT "s_name",
2           COUNT(*) AS "people_count"
3    FROM   "subscribers"
4    GROUP  BY "s_name"
```

The aggregating function `COUNT(*)` (presented in the second line of all queries 2.1.2.b) counts records grouped by matching `s_name` field values (see the fourth line in queries 2.1.2.b).

To quickly understand the logic of grouping, you can use the analogy of merging cells with the same values in a Word or Excel table, and then analyze (most often counting or adding up) the cells corresponding to each such “merged cell”. In example 2.1.2.b, just discussed, this analogy can be expressed as follows:

s_id	s_name
1	Ivanov I.I.
2	Petrov P.P.
3	Sidorov S.S.
4	

Now the DBMS counts the table rows in the context of the grouping performed, and gets that we have “one Ivanov”, and “one Petrov”, and “two Sidorovs”, which is reflected in the expected result 2.1.2.b.

Task 2.1.2.TSK.A: show without duplication the identifiers of the books that were ever taken by the readers.

Task 2.1.2.TSK.B: for each book readers ever borrowed, show the number of times a book was taken from the library.

2.1.3. Example 3: COUNT Function and its Performance

Problem 2.1.3.a[{23}]: show how many different books are registered in the library.

Expected result 2.1.3.a.

total_books
7

Solution 2.1.3.a[{23}].

If we reformulate the problem, it would be “show how many entries there are in the `books` table”, and the solution looks like this.

MySQL Solution 2.1.3.a

```
1    SELECT COUNT(*) AS `total_books`
2    FROM   `books`
```

MS SQL Solution 2.1.3.a

```
1    SELECT COUNT(*) AS [total_books]
2    FROM   [books]
```

Oracle Solution 2.1.3.a

```
1    SELECT COUNT(*) AS "total_books"
2    FROM   "books"
```

The `COUNT` function can be used in the following five formats:

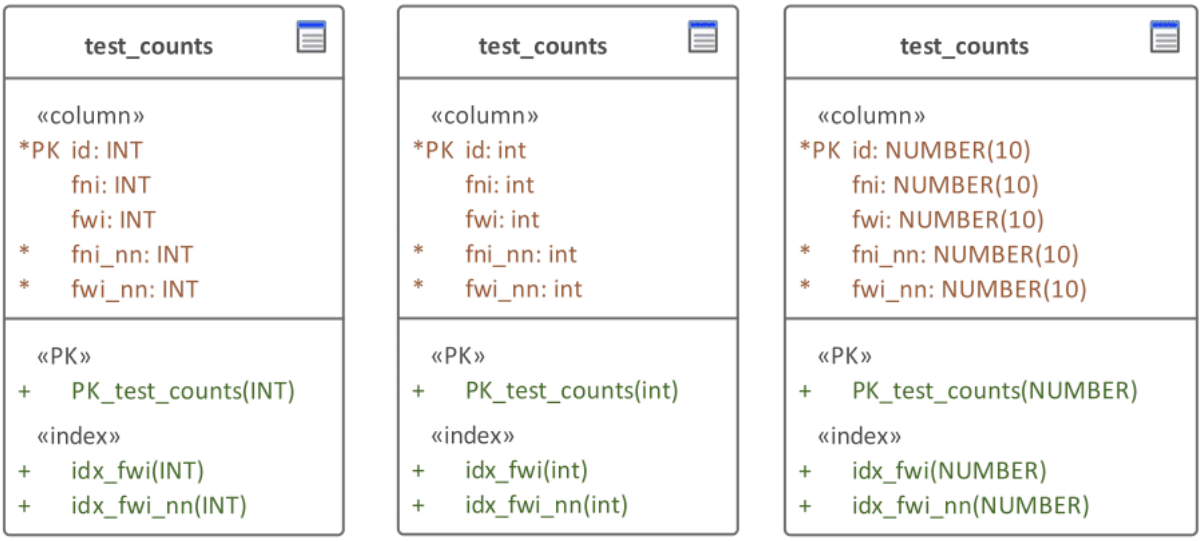
- `COUNT(*)` — the classic option used for counting the number of records;
- `COUNT(1)` — an alternative notation of the classic option;
- `COUNT(primary_key)` — an alternative notation of the classic option;
- `COUNT(field)` — counting the records in the specified field of which there are **no** `NULL` values;
- `COUNT(DISTINCT field)` — counting without duplications the records in the specified field of which there are **no** `NULL` values.

One of the most frequent questions about the different `COUNT` options is about performance: which option is faster?

Exploration 2.1.3.EXP.A: evaluating the speed of different `COUNT` options depending on the volume of processed data. This exploration is the only one in which the DBMS cache will not be reset before each subsequent query is executed.

In the “Exploration” database, let’s create a table `test_counts`, which contains the following fields:

- `id` — auto-incrementable primary key (number);
- `fni` — a field without an index (“field, no index”) (number or `NULL`);
- `fwi` — a field with an index (“field, with index”) (number or `NULL`);
- `fni_nn` — a field without an index and `NULL`s (“field, no index, no nulls”) (number);
- `fwi_nn` — a field with an index and without `NULL`s (“field, with index, no nulls”) (number).



MySQL MS SQL Server Oracle

Figure 2.1.a — The `test_counts` table in all three DBMSes

The essence of the exploration is to sequentially add a thousand records (from zero to ten million in increments of a thousand) to the table and run the following seven queries with **`COUNT`** after adding each ten thousand records:

MySQL Exploration 2.1.3.EXP.A

```
1    -- Option 1: COUNT(*)
2    SELECT COUNT(*)
3    FROM   `test_counts`

1    -- Option 2: COUNT(primary_key)
2    SELECT COUNT(`id`)
3    FROM   `test_counts`

1    -- Option 3: COUNT(1)
2    SELECT COUNT(1)
3    FROM   `test_counts`

1    -- Option 4: COUNT(field_without_index)
2    SELECT COUNT(`fni`)
3    FROM   `test_counts`

1    -- Option 5: COUNT(field_with_index)
2    SELECT COUNT(`fwi`)
3    FROM   `test_counts`

1    -- Option 6: COUNT(DISTINCT field_without_index)
2    SELECT COUNT(DISTINCT `fni`)
3    FROM   `test_counts`

1    -- Option 7: COUNT(DISTINCT field_with_index)
2    SELECT COUNT(DISTINCT `fwi`)
3    FROM   `test_counts`
```

MS SQL Exploration 2.1.3.EXP.A

```
1    -- Option 1: COUNT(*)
2    SELECT COUNT(*)
3    FROM   [test_counts]

1    -- Option 2: COUNT(primary_kary)
2    SELECT COUNT([id])
3    FROM   [test_counts]

1    -- Option 3: COUNT(1)
2    SELECT COUNT(1)
3    FROM   [test_counts]

1    -- Option 4: COUNT(field_without_index)
2    SELECT COUNT([fni])
3    FROM   [test_counts]

1    -- Option 5: COUNT(field_with_index)
2    SELECT COUNT([fwi])
3    FROM   [test_counts]

1    -- Option 6: COUNT(DISTINCT field_without_index)
2    SELECT COUNT(DISTINCT [fni])
3    FROM   [test_counts]

1    -- Option 7: COUNT(DISTINCT field_with_index)
2    SELECT COUNT(DISTINCT [fwi])
3    FROM   [test_counts]
```

Oracle Exploration 2.1.3.EXP.A

```
1    -- Option 1: COUNT(*)
2    SELECT COUNT(*)
3    FROM   "test_counts"

1    -- Option 2: COUNT(primary_key)
2    SELECT COUNT("id")
3    FROM   "test_counts"

1    -- Option 3: COUNT(1)
2    SELECT COUNT(1)
3    FROM   "test_counts"

1    -- Option 4: COUNT(field_without_index)
2    SELECT COUNT("fni")
3    FROM   "test_counts"

1    -- Option 5: COUNT(field_with_index)
2    SELECT COUNT("fwi")
3    FROM   "test_counts"

1    -- Option 6: COUNT(DISTINCT field_without_index)
2    SELECT COUNT(DISTINCT "fni")
3    FROM   "test_counts"

1    -- Option 7: COUNT(DISTINCT field_with_index)
2    SELECT COUNT(DISTINCT "fwi")
3    FROM   "test_counts"
```

First of all, let's consider the time it takes each DBMS to insert a thousand records, and the dependence of this time on the number of records already in the table. The corresponding data are shown in figure 2.1.b.

As can be seen from the chart, even with such a relatively small amount of data, all three DBMSes showed a drop in insertion operation performance closer to the end of the experiment. This effect was strongest for Oracle. MySQL showed the lowest execution time during the whole experiment (also, MySQL results were the most consistent).

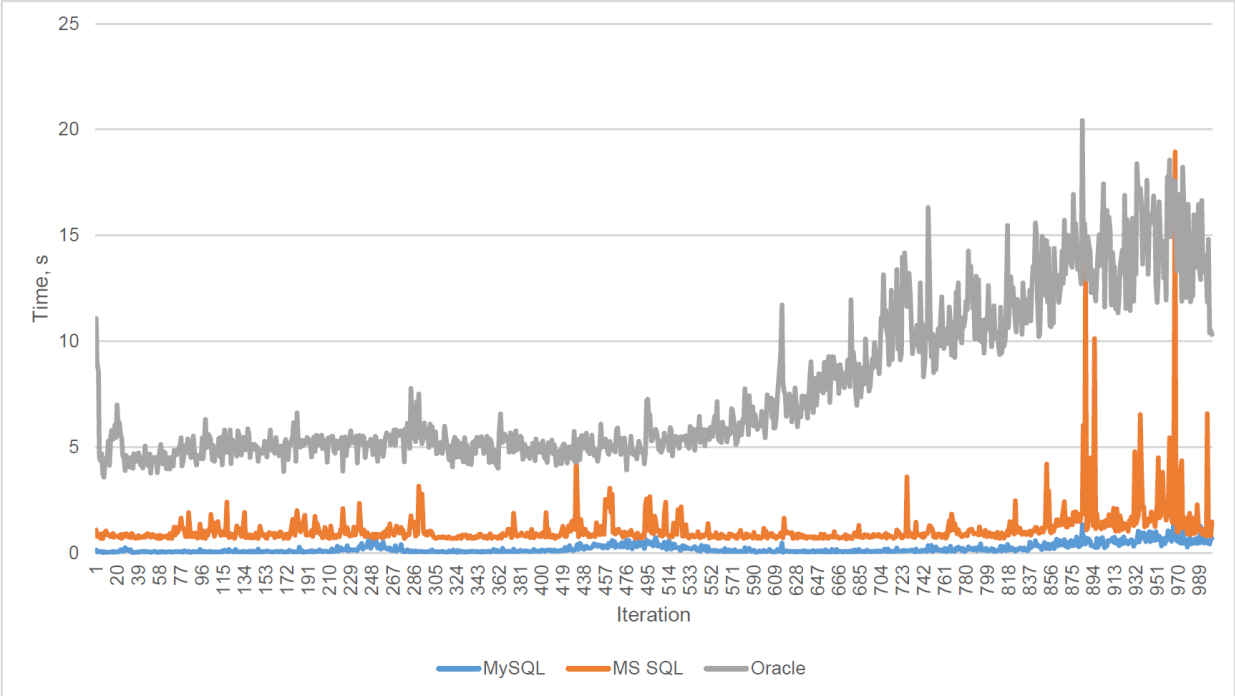


Figure 2.1.b — Time spent by each DBMS to insert a thousand records

Now let's look at the charts of the execution time for each of the seven queries 2.1.3.EXP.A discussed above, depending on the amount of data in the table. Figures 2.1.c, 2.1.d, 2.1.e show the charts for MySQL, MS SQL Server, and Oracle, respectively.

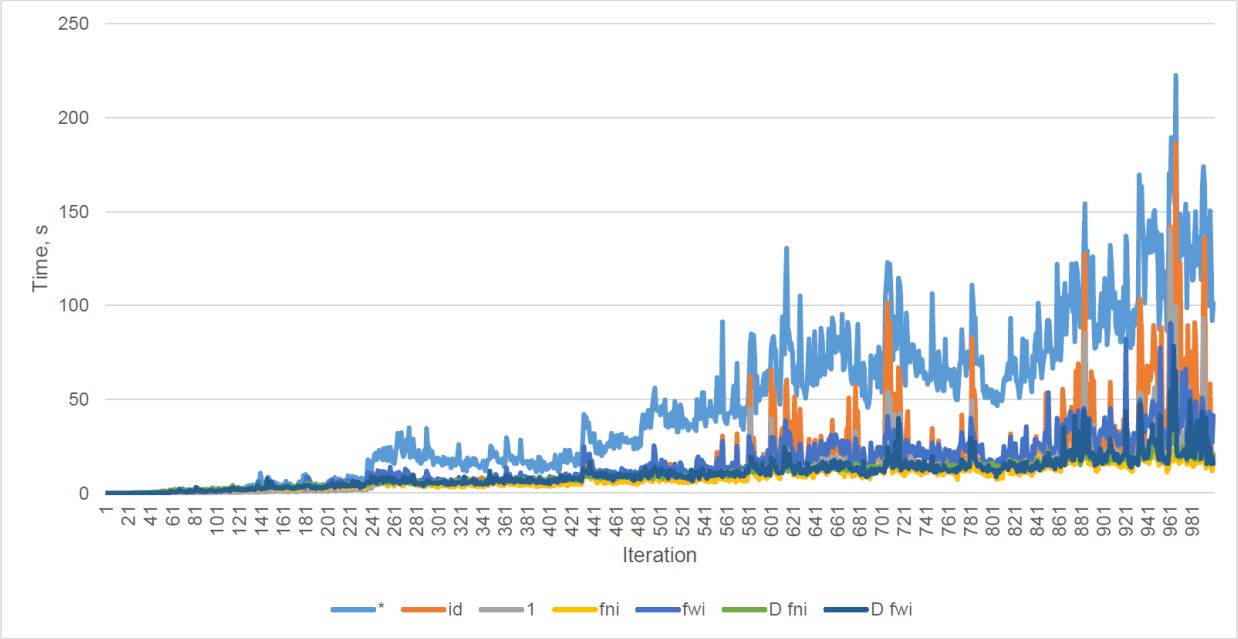


Figure 2.1.c — Time taken by MySQL to execute different `COUNT`s

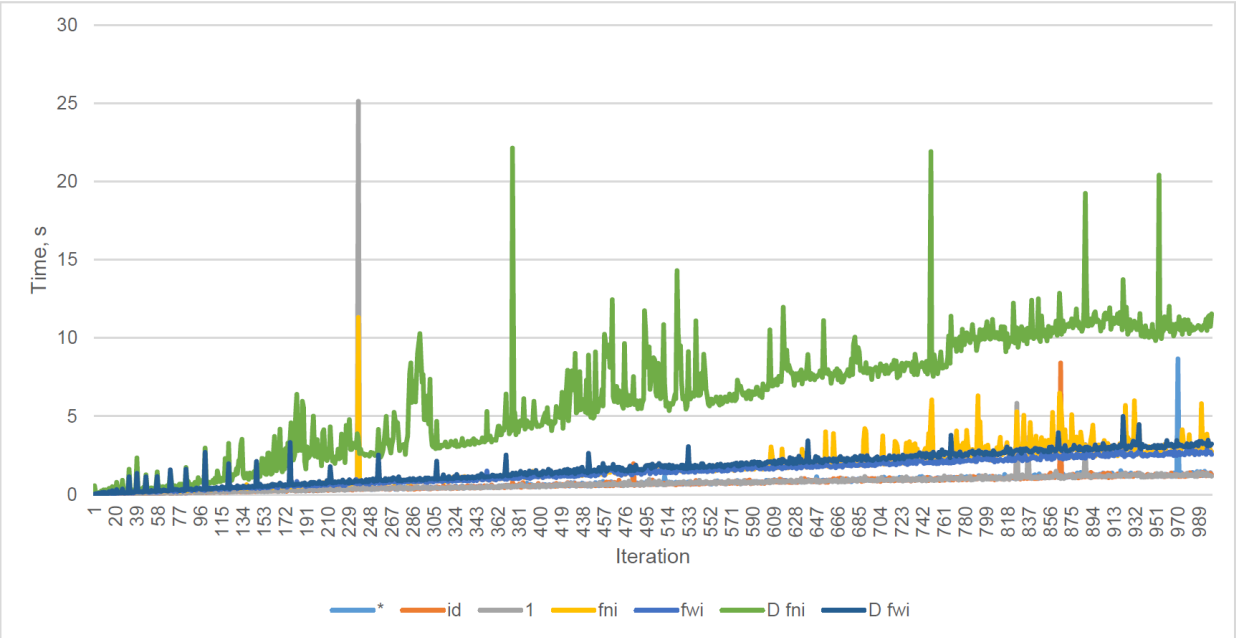


Figure 2.1.d — Time taken by MS SQL to execute different `COUNT`s

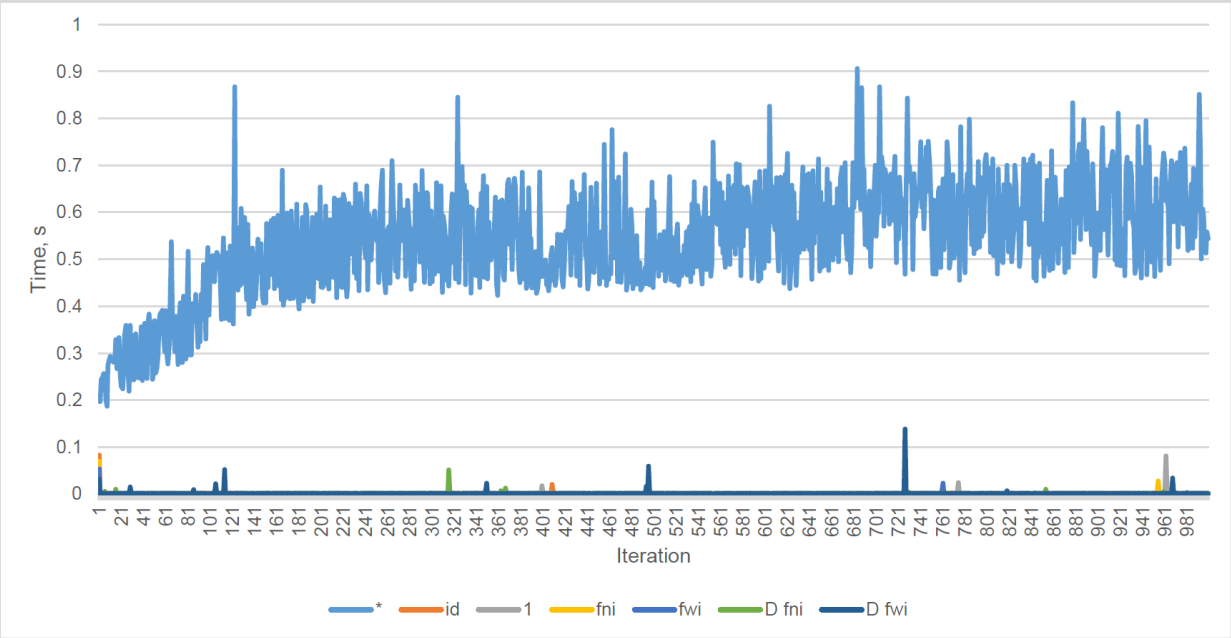


Figure 2.1.e — Time taken by Oracle to execute different `COUNT`s

To see the whole picture, let's combine into one table the median execution time values for each query of 2.1.3.EXP.A for each DBMS:

	MySQL	MS SQL Server	Oracle
COUNT(*)	**36.662**	0.702	**0.541**
COUNT(id)	11.120	**0.679**	**0.001**
COUNT(1)	9.995	0.681	**0.001**
COUNT(fni)	**6.999**	1.494	**0.001**
COUNT(fwi)	12.333	1.475	**0.001**
COUNT(DISTINCT fni)	9.252	**6.300**	**0.001**
COUNT(DISTINCT fwi)	9.626	1.743	**0.001**

Here you can see a somewhat strange picture.

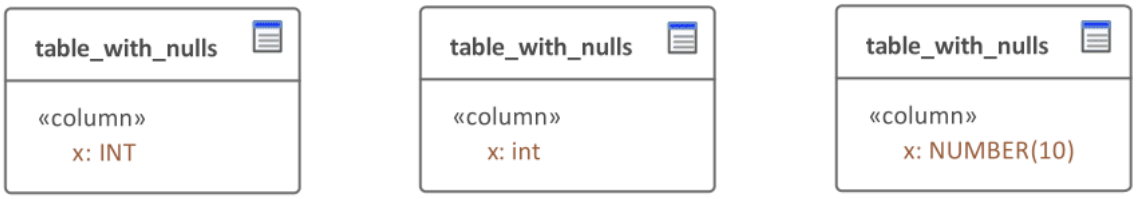
For MySQL, the popular belief that `COUNT(*)` is the slowest is confirmed, but unexpectedly the fastest option is `COUNT(field_without_index)`. However, it is worth noting that the situation is completely different for smaller data volumes (up to a million records) — `COUNT(*)` is the fastest.

MS SQL Server showed the expected results: `COUNT(primary_key)` was the fastest, `COUNT(DISTINCT field_without_index)` was the slowest. This result does not change on smaller amounts of data.

And now the most interesting — the results of Oracle: here again the rumors about slow `COUNT(*)` were confirmed, but all other queries showed amazing results, very high stability of the result and complete independence from the volume of analyzed data. (Refer to Oracle documentation to find out how this DBMS manages to execute some `COUNT` types so quickly).

Although this study does not claim to be scientific, there is a general recommendation: use `COUNT(1)` as one of the fastest options "in general" for different DBMSes and different volumes of data, because the other options are sometimes faster, and sometimes slower.

Exploration 2.1.3.EXP.B: let's see how the `COUNT` function reacts to `NULL` values and to duplicated values in DISTINCT mode. Let's add the `table_with_nulls` table to the "Exploration" database.



MySQL | MS SQL Server | Oracle

Figure 2.1.f — The `table_with_nulls` table in all three DBMSes

Let's put the following data into the created table (the same data for all three DBMSes):

x
1
1
2
NULL

Let's run the following queries to count the number of records in this table:

MySQL | Exploration 2.1.3.EXP.B

```
1    -- Option 1: COUNT(*)
2    SELECT COUNT(*) AS `total_records`
3    FROM   `table_with_nulls`

1    -- Option 2: COUNT(1)
2    SELECT COUNT(1) AS `total_records`
3    FROM   `table_with_nulls`
```

MS SQL Exploration 2.1.3.EXP.B

```
1     -- Option 1: COUNT(*)
2     SELECT COUNT(*) AS [total_records]
3     FROM   [table_with_nulls]

1     -- Option 2: COUNT(1)
2     SELECT COUNT(1) AS [total_records]
3     FROM   [table_with_nulls]
```

Oracle Exploration 2.1.3.EXP.B

```
1     -- Option 1: COUNT(*)
2     SELECT COUNT(*) AS "total_records"
3     FROM   "table_with_nulls"

1     -- Option 2: COUNT(1)
2     SELECT COUNT(1) AS "total_records"
3     FROM   "table_with_nulls"
```

All queries in all DBMSes will produce the same result:

total_records
4

Now let's run the query with **COUNT**, where the argument will be the name of the field:

MySQL Exploration 2.1.3.EXP.B

```
1     -- Option 3: COUNT(field)
2     SELECT COUNT(`x`) AS `total_records`
3     FROM   `table_with_nulls`
```

MS SQL Exploration 2.1.3.EXP.B

```
1     -- Option 3: COUNT(field)
2     SELECT COUNT([x]) AS [total_records]
3     FROM   [table_with_nulls]
```

Oracle Exploration 2.1.3.EXP.B

```
1     -- Option 3: COUNT(field)
2     SELECT COUNT("x") AS "total_records"
3     FROM   "table_with_nulls"
```

Since in this case the calculation does not take **NULL** values into account, in all three DBMSes the result will be as follows:

total_records
3

Finally, let's run the query with **COUNT** in **DISTINCT** mode:

MySQL Exploration 2.1.3.EXP.B

```
1     -- Option 4: COUNT(DISTINCT field)
2     SELECT COUNT(DISTINCT `x`) AS `total_records`
3     FROM   `table_with_nulls`
```

MS SQL Exploration 2.1.3.EXP.B

```
1     -- Option 4: COUNT(DISTINCT field)
2     SELECT COUNT(DISTINCT [x]) AS [total_records]
3     FROM   [table_with_nulls]
```

Oracle Exploration 2.1.3.EXP.B

```
1     -- Option 4: COUNT(DISTINCT field)
2     SELECT COUNT(DISTINCT "x") AS "total_records"
3     FROM   "table_with_nulls"
```

In this mode, **COUNT** does not consider both **NULL** values, and also all duplicate values (in our original data set the "1" value was represented twice). So, the result of query execution in all three DBMSes is:

total_records
2

It remains to check how **COUNT** works on the empty data. Let's change the query so that none of the rows are correspond to the **WHERE** condition:

MySQL Exploration 2.1.3.EXP.B

```
1    -- Option 5: COUNT(field_with_empty_data_set)
2    SELECT COUNT(`x`) AS `negative_records`
3    FROM   `table_with_nulls`
4    WHERE  `x` < 0
```

MS SQL Exploration 2.1.3.EXP.B

```
1    -- Option 5: COUNT(field_with_empty_data_set)
2    SELECT COUNT([x]) AS [negative_records]
3    FROM   [table_with_nulls]
4    WHERE  [x] < 0
```

Oracle Exploration 2.1.3.EXP.B

```
1    -- Option 5: COUNT(field_with_empty_data_set)
2    SELECT COUNT("x") AS "negative_records"
3    FROM   "table_with_nulls"
4    WHERE  "x" < 0
```

All three DBMSes produce the same easy-to-predict result:

negative_records
0

Task 2.1.3.TSK.A: show how many readers (subscribers) are registered in the library.

2.1.4. Example 4: COUNT Function with Conditions

Problem 2.1.4.a[31]: show how many copies of books were given out to readers.

Problem 2.1.4.b[32]: show how many different books were given out to readers.

Expected result 2.1.4.a.

in_use
5

Expected result 2.1.4.b.

in_use
4

Solution 2.1.4.a[31].

All the information about the books given out to the readers (the facts of giving and returning) is stored in the **subscriptions** table, the **sb_book** field of which contains the identifier of the given-out book. The **sb_is_active** field contains the **Y** value in case the book is now with the reader (not returned to the library).

Thus, we will be interested only in rows with the value of **sb_is_active** field equal to **Y** (which is reflected in the **WHERE** section, see the third line of all six queries 2.1.4.a-2.1.4.b), and the difference between the problems 2.1.4.a and 2.1.4. b is that in the first case we just count all cases “the book is with the reader”, and in the second case we do not count duplicates (when several copies of the same book are given to readers), and we achieve this by **COUNT(DISTINCT field)** expression.

MySQL Solution 2.1.4.a

```
1    SELECT COUNT(`sb_book`) AS `in_use`
2    FROM   `subscriptions`
3    WHERE  `sb_is_active` = 'Y'
```

MS SQL Solution 2.1.4.a

```
1    SELECT COUNT([sb_book]) AS [in_use]
2    FROM   [subscriptions]
3    WHERE  [sb_is_active] = 'Y'
```

Oracle Solution 2.1.4.a

```
1    SELECT COUNT("sb_book") AS "in_use"
2    FROM   "subscriptions"
3    WHERE  "sb_is_active" = 'Y'
```

Solution 2.1.4.b[31].

Here we extend the solution[31] of problem 2.1.4.a[31] by adding the `DISTINCT` keyword to the `COUNT` function parameter, which ensures that non-repeating values of the `sb_book` field are counted.

MySQL Solution 2.1.4.b

```
1    SELECT COUNT(DISTINCT `sb_book`) AS `in_use`
2    FROM   `subscriptions`
3    WHERE  `sb_is_active` = 'Y'
```

MS SQL Solution 2.1.4.b

```
1    SELECT COUNT(DISTINCT [sb_book]) AS [in_use]
2    FROM   [subscriptions]
3    WHERE  [sb_is_active] = 'Y'
```

Oracle Solution 2.1.4.b

```
1    SELECT COUNT(DISTINCT "sb_book") AS "in_use"
2    FROM   "subscriptions"
3    WHERE  "sb_is_active" = 'Y'
```

Task 2.1.4.TSK.A: show how many times readers were given books in total.

Task 2.1.4.TSK.B: show how many readers borrowed books from the library.

2.1.5. Example 5: SUM, MIN, MAX, AVG Functions Usage

Problem 2.1.5.a{33}: show the total (sum), minimum, maximum, and average value of the number of copies of books in the library.

Expected result 2.1.5.a.

sum	min	max	avg
33	1	12	4.7143

Solution 2.1.5.a{33}.

In such a simple formulation, this problem is solved by the query, in which it is enough to list the corresponding functions, passing the `b_quantity` field as a parameter (and only in MS SQL Server we have to make a slight modification).

MySQL Solution 2.1.5.a

```
1    SELECT SUM(`b_quantity`) AS `sum`,
2           MIN(`b_quantity`) AS `min`,
3           MAX(`b_quantity`) AS `max`,
4           AVG(`b_quantity`) AS `avg`
5    FROM   `books`
```

MS SQL Solution 2.1.5.a

```
1    SELECT SUM([b_quantity]) AS [sum],
2           MIN([b_quantity]) AS [min],
3           MAX([b_quantity]) AS [max],
4           AVG(CAST([b_quantity] AS FLOAT)) AS [avg]
5    FROM   [books]
```

Oracle Solution 2.1.5.a

```
1    SELECT SUM("b_quantity") AS "sum",
2           MIN("b_quantity") AS "min",
3           MAX("b_quantity") AS "max",
4           AVG("b_quantity") AS "avg"
5    FROM   "books"
```

Note the 4th line in the query 2.1.5.a for MS SQL Server: without the `CAST` function to convert the number of books to a `FLOAT` value, the final result of the `AVG` function call will be incorrect (it will be an integer), because MS SQL Server chooses the data type of the result based on the data type of the input parameter. Let's demonstrate this.

MS SQL Solution 2.1.5.a (a query with a mistake)

```
1    SELECT SUM([b_quantity]) AS [sum],
2           MIN([b_quantity]) AS [min],
3           MAX([b_quantity]) AS [max],
4           AVG([b_quantity]) AS [avg]
5    FROM   [books]
```

We get:

sum	min	max	Avg
33	1	12	4

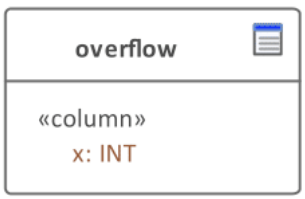
And how it was supposed to be:

sum	min	max	avg
33	1	12	4.7143

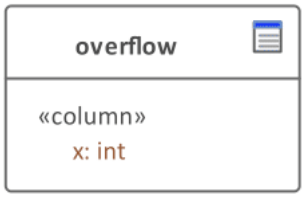
It is worth mentioning another dangerous mistake: it is very easy to forget that when calculating the sum and the average value (which is defined as the sum divided by the number of values), an arithmetic overflow can occur.

Exploration 2.1.5.EXP.A: let's analyze the behavior of different DBMSes in the situation of arithmetic overflow.

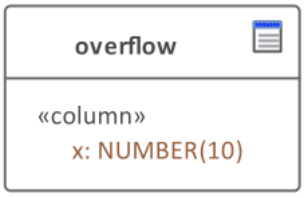
Let's create the **`overflow`** table with one **`INTEGER`** field in the "Exploration" database:



MySQL



MS SQL Server



Oracle

Figure 2.1.g — The **`overflow`** table in all three DBMSes

Now let's put into the created table three maximal values for its **`x`** field:

x
2147483647
2147483647
2147483647

MySQL

x
2147483647
2147483647
2147483647

MS SQL Server

x
9999999999
9999999999
9999999999

Oracle

Now let's run queries for each DBMS to get the sum of the values from the **`x`** field and the average value in the **`x`** field:

MySQL Exploration 2.1.5.EXP.A

```
1    -- Query 1: SUM
2    SELECT SUM(`x`) AS `sum`
3    FROM   `overflow`

1    -- Query 2: AVG
2    SELECT AVG(`x`) AS `avg`
3    FROM   `overflow`
```

MS SQL Exploration 2.1.5.EXP.A

```
1    -- Query 1: SUM
2    SELECT SUM([x]) AS [sum]
3    FROM   [overflow]

1    -- Query 2: AVG
2    SELECT AVG([x]) AS [avg]
3    FROM   [overflow]
```

Oracle Exploration 2.1.5.EXP.A

```
1    -- Query 1: SUM
2    SELECT SUM("x") AS "sum"
3    FROM   "overflow"

1    -- Query 2: AVG
2    SELECT AVG("x") AS "avg"
3    FROM   "overflow"
```

While MySQL and Oracle will execute queries 2.1.5.EXP.A and return the correct data, in MS SQL Server we will get the following error message:

```
Msg 8115, Level 16, State 2, Line 1. Arithmetic overflow error converting ex-
pression to data type int.
```

This is quite logical, because when we try to place a triple maximum value of **INTEGER** type in a variable of **INTEGER** type, the arithmetic overflow occurs.

MySQL and Oracle are less prone to this effect because they have **DECIMAL** and **NUMBER** data types "in stock", in the format of which the calculation takes place. But if the amount of data is large enough, arithmetic overflow also occurs there.

A special danger of this error is that at the stage of development and quick testing of the database it does not manifest itself, and only in time, when the real users have accumulated a large amount of data, at some point the previously perfectly working queries stop working.

Exploration 2.1.5.EXP.B. To avoid going back to the peculiarities of aggregation functions, let's consider their behavior in the case of **NULL** values in the analyzed field, as well as in the case of an empty set of input values.

We'll use the previously created **table_with_nulls**[28] table. It still contains the following data:

x
1
1
2
NULL

Let's execute queries 2.1.5.EXP.B:

MySQL Exploration 2.1.5.EXP.B

```
1    SELECT SUM(`x`) AS `sum`,
2           MIN(`x`) AS `min`,
3           MAX(`x`) AS `max`,
4           AVG(`x`) AS `avg`
5    FROM   `table_with_nulls`
```

MS SQL Exploration 2.1.5.EXP.B

```
1    SELECT SUM([x]) AS [sum],
2           MIN([x]) AS [min],
3           MAX([x]) AS [max],
4           AVG(CAST([x] AS FLOAT)) AS [avg]
5    FROM   [table_with_nulls]
```

Oracle Exploration 2.1.5.EXP.B

```
1    SELECT SUM("x") AS "sum",
2           MIN("x") AS "min",
3           MAX("x") AS "max",
4           AVG("x") AS "avg"
5    FROM   "table_with_nulls"
```

All queries 2.1.5.EXP.B results are almost the same (the only difference is the number of decimal places in the **AVG** function result: MySQL has 4 decimal places, while MS SQL Server and Oracle have 14 and 38 decimal places, respectively):

sum	min	max	avg
4	1	2	1.3333

As we can easily see from the output data, none of the functions consider **NULL** values.

Exploration 2.1.5.EXP.C. The last experiment will be about applying such a condition to which no row corresponds: thus, the data set will contain an empty set of rows.

MySQL Exploration 2.1.5.EXP.C

```
SELECT SUM(`x`) AS `sum`,
       MIN(`x`) AS `min`,
       MAX(`x`) AS `max`,
       AVG(`x`) AS `avg`
FROM   `table_with_nulls`
WHERE  `x` < 0
```

MS SQL Exploration 2.1.5.EXP.C

```
SELECT SUM([x]) AS [sum],
       MIN([x]) AS [min],
       MAX([x]) AS [max],
       AVG(CAST([x] AS FLOAT)) AS [avg]
FROM   [table_with_nulls]
WHERE  [x] < 0
```

Oracle Exploration 2.1.5.EXP.C

```
SELECT SUM("x") AS "sum",
       MIN("x") AS "min",
       MAX("x") AS "max",
       AVG("x") AS "avg"
FROM   "table_with_nulls"
WHERE  "x" < 0
```

Here all three DBMSes also work the same way, clearly demonstrating that on an empty data set the **SUM**, **MIN**, **MAX**, and **AVG** functions return **NULL**:

sum	min	max	avg
NULL	NULL	NULL	NULL

Also note that when calculating the average value there was no “division by zero” error.

The logic of **COUNT** function working on an empty dat set has already been discussed earlier (see exploration 2.1.3.EXP.B{30}).

Task 2.1.5.TSK.A: show the first and last dates of giving a book to a reader.

2.1.6. Example 6: Ordering Query Results

Problem 2.1.6.a{37}: show all books in the library in order of year of publication.

Problem 2.1.6.b{38}: show all the books in the library in descending order of year of publication.

Expected result 2.1.6.a.

b_name	b_year
Course of Theoretical Physics	1981
Eugene Onegin	1985
The Fisherman and the Golden Fish	1990
The Art of Computer Programming	1993
The C++ Programming Language	1996
Programming Psychology	1998
Foundation and Empire	2000

Expected result 2.1.6.b.

b_name	b_year
Foundation and Empire	2000
Programming Psychology	1998
The C++ Programming Language	1996
The Art of Computer Programming	1993
The Fisherman and the Golden Fish	1990
Eugene Onegin	1985
Course of Theoretical Physics	1981

Solution 2.1.6.a{37}.

To order[2] the results of the query it is necessary to apply the `ORDER BY` clause (line 4 of each query) in which we specify:

- the field by which the sorting is performed (`b_year`);
- sorting direction (`ASC`).

MySQL Solution 2.1.6.a

```
1    SELECT `b_name`,
2           `b_year`
3    FROM   `books`
4    ORDER  BY `b_year` ASC
```

MS SQL Solution 2.1.6.a

```
1    SELECT [b_name],
2           [b_year]
3    FROM   [books]
4    ORDER  BY [b_year] ASC
```

[2] In everyday life, most people use the term “sorting” instead of “ordering” anyway, but they are not exactly identical concepts. “Sorting” refers more to the process of reordering, while “ordering” refers to the finished result.

Oracle Solution 2.1.6.a

```
SELECT "b_name",
       "b_year"
FROM   "books"
ORDER  BY "b_year" ASC
```

Solution 2.1.6.b[{37}].

Here (in comparison to the solution[{37}] of problem 2.1.6.a[{37}]) it is only necessary to change the ordering direction from “ascending” (**`ASC`**) to “descending” (**`DESC`**).

MySQL Solution 2.1.6.b

```
SELECT `b_name`,
       `b_year`
FROM   `books`
ORDER  BY `b_year` DESC
```

MS SQL Solution 2.1.6.b

```
SELECT [b_name],
       [b_year]
FROM   [books]
ORDER  BY [b_year] DESC
```

Oracle Solution 2.1.6.b

```
SELECT "b_name",
       "b_year"
FROM   "books"
ORDER  BY "b_year" DESC
```

An alternative solution can be obtained by adding a “minus” sign before the name of the field for which the ordering is implemented in ascending order, i.e., **`ORDER BY numeric_field DESC`** is equivalent to **`ORDER BY -numeric_field ASC`**.

MySQL Solution 2.1.6.b (alternate solution)

```
SELECT `b_name`,
       `b_year`
FROM   `books`
ORDER  BY -`b_year` ASC
```

MS SQL Solution 2.1.6.b (alternate solution)

```
SELECT [b_name],
       [b_year]
FROM   [books]
ORDER  BY -[b_year] ASC
```

Oracle Solution 2.1.6.b (alternate solution)

```
SELECT "b_name",
       "b_year"
FROM   "books"
ORDER  BY -"b_year" ASC
```

Exploration 2.1.6.EXP.A. Another unexpected ordering problem is related to where different DBMSes place `NULL` values by default, either at the beginning of the result, or at the end. Let's check.

Again, we use the existing `table_with_nulls`[28] table. It still contains the following data:

x
1
1
2
NULL

Let's execute queries 2.1.6.EXP.A:

MySQL Exploration 2.1.6.EXP.A

```
1    SELECT `x`
2    FROM   `table_with_nulls`
3    ORDER  BY `x` DESC
```

MS SQL Exploration 2.1.6.EXP.A

```
1    SELECT [x]
2    FROM   [table_with_nulls]
3    ORDER  BY [x] DESC
```

Oracle Exploration 2.1.6.EXP.A

```
1    SELECT "x"
2    FROM   "table_with_nulls"
3    ORDER  BY "x" DESC
```

The results will be as follows (note where the `NULL` value is located):

x
2
1
1
NULL

MySQL

x
2
1
1
NULL

MS SQL Server

x
NULL
2
1
1

Oracle

We can get the behavior similar to Oracle (and vice versa) in MySQL and MS SQL Server using the following queries:

MySQL Exploration 2.1.6.EXP.A

```
1    SELECT `x`
2    FROM   `table_with_nulls`
3    ORDER  BY `x` IS NULL DESC,
4              `x` DESC
```

MS SQL Exploration 2.1.6.EXP.A

```
1    SELECT [x]
2    FROM   [table_with_nulls]
3    ORDER  BY ( CASE
4                  WHEN [x] IS NULL THEN 0
5                  ELSE 1
6                END ) ASC,
7              [x] DESC
```

Oracle Exploration 2.1.6.EXP.A

```
1    SELECT "x"
2    FROM   "table_with_nulls"
3    ORDER  BY "x" DESC NULLS LAST
```

In the case of Oracle, we explicitly specify whether to put `NULL` values at the beginning of the result (`NULLS FIRST`) or at the end of the result (`NULLS LAST`). Since MySQL and MS SQL Server does not support this syntax, the data has to be ordered by two levels: at the first level (line 3 for MySQL, lines 3-6 for MS SQL Server) on the basis of "whether the value of the field is `NULL`", and the second level by the value of the field.

Task 2.1.6.TSK.A: show the list of authors in reverse alphabetical order (i.e., "Z → A").

2.1.7. Example 7: Using Compound Conditions

Problem 2.1.7.a[41]: show books published between 1990 and 2000, available in the library in quantities of three or more copies.

Problem 2.1.7.b[42]: show the identifiers and dates of subscriptions for the summer of 2012.

Expected result 2.1.7.a.

b_name	b_year	b_quantity
The Fisherman and the Golden Fish	1990	3
Foundation and Empire	2000	5
The C++ Programming Language	1996	3
The Art of Computer Programming	1993	7

Expected result 2.1.7.b.

sb_id	sb_start
42	2012-06-11
57	2012-06-11

Solution 2.1.7.a[41].

There are two query options for each DBMS: with the **`BETWEEN`** keyword (often used just to specify date range) and without it, as a double inequality (which looks more familiar to those who have programming experience).

If **`BETWEEN`** is used, the boundaries are included in the range of values searched for.

In the solutions below, the individual conditions are deliberately left out of the brackets. Syntactically, this solution is correct and works fine, but it is more difficult to read the more components are included in the complex condition. That is why it is advisable to bracket each individual part.

MySQL Solution 2.1.7.a

```
1    -- Option 1: BETWEEN
2    SELECT `b_name`,
3           `b_year`,
4           `b_quantity`
5    FROM   `books`
6    WHERE  `b_year` BETWEEN 1990 AND 2000
7           AND `b_quantity` >= 3

1    -- Option 2: double inequality
2    SELECT `b_name`,
3           `b_year`,
4           `b_quantity`
5    FROM   `books`
6    WHERE  `b_year` >= 1990
7           AND `b_year` <= 2000
8           AND `b_quantity` >= 3
```

MS SQL Solution 2.1.7.a

```
-- Option 1: BETWEEN
SELECT [b_name],
       [b_year],
       [b_quantity]
FROM   [books]
WHERE  [b_year] BETWEEN 1990 AND 2000
       AND [b_quantity] >= 3

-- Option 2: double inequality
SELECT [b_name],
       [b_year],
       [b_quantity]
FROM   [books]
WHERE  [b_year] >= 1990
       AND [b_year] <= 2000
       AND [b_quantity] >= 3
```

Oracle Solution 2.1.7.a

```
-- Option 1: BETWEEN
SELECT "b_name",
       "b_year",
       "b_quantity"
FROM   "books"
WHERE  "b_year" BETWEEN 1990 AND 2000
       AND "b_quantity" >= 3

-- Option 2: double inequality
SELECT "b_name",
       "b_year",
       "b_quantity"
FROM   "books"
WHERE  "b_year" >= 1990
       AND "b_year" <= 2000
       AND "b_quantity" >= 3
```

Solution 2.1.7.b[41].

Let's first look at the correct and wrong solutions, and then explain what the problem is with the wrong one.

Correct solution:

MySQL Solution 2.1.7.b

```
SELECT `sb_id`,
       `sb_start`
FROM   `subscriptions`
WHERE  `sb_start` >= '2012-06-01'
       AND `sb_start` < '2012-09-01'
```

MS SQL Solution 2.1.7.b

```
SELECT [sb_id],
       [sb_start]
FROM   [subscriptions]
WHERE  [sb_start] >= '2012-06-01'
       AND [sb_start] < '2012-09-01'
```

Oracle Solution 2.1.7.b

```
SELECT "sb_id",
       "sb_start"
FROM   "subscriptions"
WHERE  "sb_start" >= TO_DATE('2012-06-01', 'yyyy-mm-dd')
       AND "sb_start" < TO_DATE('2012-09-01', 'yyyy-mm-dd')
```

Specifying the right boundary of the date range as a strict inequality is convenient because we don't have to remember or calculate the last day of a month (especially true for February) or write constructs like 23:59:59.9999 (if we have to consider time too). Whichever part of the date we operate with (year, month, day, hour, minute, second, fraction of a second), we can always form the next value outside the range we are looking for and use a strict inequality.

Also note lines 4-5 of query 2.1.7.b: MySQL and MS SQL Server allow strings for dates specification (and automatically perform the necessary conversions), while Oracle requires explicit conversion of string date representation to an appropriate data type.

It is time to consider a very common but **wrong** solution that results correctly but is fatal for performance. Instead of getting two constants (the beginning and the end of date range) and directly comparing the values of the analyzed table column with them (using an index), the DBMS has to perform two transformations **for each record** in the table and then compare the results with the specified constants (directly, without using an index, because the table has no indexes for the "year" and "month" extraction results).

MySQL — Solution 2.1.7.b (incorrect option in terms of performance)

```
1    SELECT `sb_id`,
2           `sb_start`
3    FROM   `subscriptions`
4    WHERE  YEAR(`sb_start`) = 2012
5           AND MONTH(`sb_start`) BETWEEN 6 AND 8
```

MS SQL — Solution 2.1.7.b (incorrect option in terms of performance)

```
1    SELECT [sb_id],
2           [sb_start]
3    FROM   [subscriptions]
4    WHERE  YEAR([sb_start]) = 2012
5           AND MONTH([sb_start]) BETWEEN 6 AND 8
```

Oracle — Solution 2.1.7.b (incorrect option in terms of performance)

```
1    SELECT "sb_id",
2           "sb_start"
3    FROM   "subscriptions"
4    WHERE  EXTRACT(year FROM "sb_start") = 2012
5           AND EXTRACT(month FROM "sb_start") BETWEEN 6 AND 8
```

Exploration 2.1.7.EXP.A. Let's demonstrate the difference in DBMS performance when executing queries from the correct and incorrect solution.

Let's create another table in the "Exploration" database with one field to store some date, over which the index will be built.

MySQL MS SQL Server Oracle

Figure 2.1.h — The `dates` table in all three DBMSes

Let's fill the table with a million records and run a hundred times each variation of the query 2.1.7.b, both correct and incorrect.

The median execution time values are as follows:

	MySQL	MS SQL Server	Oracle
Correct solution	0.003	0.059	0.674
Incorrect solution	0.436	0.086	0.966
Difference (times)	145.3	1.4	1.5

Even this trivial study shows that a query that requires extraction of a part of a field results in a performance drop of about 1.5 to about 150 times.

Unfortunately, sometimes we don't have the luxury to specify a date range (for example, we need to show information about books issued from the 3rd to the 7th day of each month of each year, or about books issued on any Sunday, see problem 2.3.3.b{206}). If there are many such queries, we should either store the date as separate fields (year, month, day, weekday) or create so-called "calculated fields" for year, month, day, week-day, and also build an index over these fields.

Task 2.1.7.TSK.A: show books with fewer copies than the average quantity of books in the library.

Task 2.1.7.TSK.B: show the identifiers and dates of subscriptions for the first year of library service (the first year of library service is "all dates from the first subscription through December 31st (inclusive) of the year in which the library began its service").

2.1.8. Example 8: Getting Minimum and Maximum Values

Problem 2.1.8.a{no}: show the book represented in the library by the maximum number of copies.

The very formulation of this problem hides a trap: in this formulation, problem 2.1.8.a has no correct solution (hence it will not be presented). In fact, there is not one problem, but three.

Problem 2.1.8.b{46}: show simply any one book whose number of copies is the maximum (is equal to the maximum among all books).

Problem 2.1.8.c{48}: show all books whose number of copies is maximal (and is the same for all these shown books).

Problem 2.1.8.d{50}: show the book (if any) that has more copies than any other book.

The expected results for these three problems are as follows (for the first one, the result may vary, since we do not specify which book from among those that fit the condition to show).

Expected result 2.1.8.b.

b_name	b_quantity
Course of Theoretical Physics	12

Expected result 2.1.8.c.

b_name	b_quantity
Course of Theoretical Physics	12

Expected result 2.1.8.d.

b_name	b_quantity
Course of Theoretical Physics	12

Seems strange, doesn't it? Three different problems, but three identical results. Yes, with the set of data that is now in the database, all three solutions give the same result, but as soon as we bring ten more copies of "Eugene Onegin" to the library (to make them 12 as well, like with the "Course of Theoretical Physics" book), the result is as follows (see for yourself, making the appropriate edit in the database).

Possible expected result 2.1.8.b.

b_name	b_quantity
Eugene Onegin	12

Possible expected result 2.1.8.c.

b_name	b_quantity
Eugene Onegin	12
Course of Theoretical Physics	12

Possible expected result 2.1.8.d.

b_name	b_quantity
{empty set, the query returned zero rows}	

Solution 2.1.8.b[45].

This task is the easiest: we need to order the result in descending order of the `b_quantity` field and take the first row of the resulting data set.

MySQL Solution 2.1.8.b

```
1    SELECT `b_name`,
2           `b_quantity`
3    FROM   `books`
4    ORDER  BY `b_quantity` DESC
5    LIMIT  1
```

MS SQL Solution 2.1.8.b

```
1    -- Option 1: using TOP
2    SELECT TOP 1 [b_name],
3                 [b_quantity]
4    FROM   [books]
5    ORDER  BY [b_quantity] DESC

1    -- Option 2: using FETCH NEXT
2    SELECT   [b_name],
3             [b_quantity]
4    FROM     [books]
5    ORDER BY [b_quantity] DESC
6    OFFSET   0 ROWS
7    FETCH    NEXT 1 ROWS ONLY
```

Oracle Solution 2.1.8.b-1 (for versions prior to 12c)

```
1    SELECT "b_name",
2           "b_quantity"
3    FROM   (SELECT "b_name",
4                  "b_quantity",
5                  ROW_NUMBER() OVER(ORDER BY "b_quantity" DESC) AS "rn"
6            FROM   "books")
7    WHERE  "rn" = 1
```

Oracle Solution 2.1.8.b-2 (for version 12c and newer)

```
1    SELECT   "b_name",
2             "b_quantity"
3    FROM     "books"
4    ORDER BY "b_quantity" DESC
5    OFFSET   0 ROWS
6    FETCH    NEXT 1 ROWS ONLY
```

In MySQL it is simple: we just specify how many rows (`LIMIT 1`) to return from the resulting data set.

In MS SQL Server, the `TOP 1` option is quite similar to the solution for MySQL: we tell the DBMS that we are only interested in the first ("top") row. The second query 2.1.8.b for MS SQL Server (lines 5-6) tells the DBMS to skip zero rows and return one next row.

The solution for Oracle is the most non-trivial. Oracle 12c (and newer versions) already supports syntax similar to MS SQL Server (see query 2.1.8.b-2), but if we work with older versions of this DBMS, we can implement the so-called "classic option" (see query 2.1.8.b-1).

In line 5 of Oracle query 2.1.8.b-1, we use a special analytic function `ROW_NUMBER` that allows us to assign a row number based on an expression. In our case, the expression used in a simplified mode: it does not specify the principle of dividing rows into groups and restarting numbering, we only ask the DBMS to number rows in the order they appear in the ordered data set.

Oracle does not allow both numbering rows and imposing a condition on the selection based on this numbering in the same query, so we are forced to use a subquery (rows 3-6). An alternative to subquery can be so-called CTE (Common Table Expression), but we will talk about CTEs later{76}.

A frequent question about the "classic solution" for Oracle is the applicability of not the `ROW_NUMBER` function here, but the "pseudo-field" `ROWNUM`. It cannot be used, because numbering with `ROWNUM` occurs before `ORDER BY` sorts the resulting data set.

Exploration 2.1.8.EXP.A. Let's demonstrate the result of incorrectly using the `ROWNUM` "pseudo-field" instead of the `ROW_NUMBER` function.

Oracle — Exploration 2.1.8.EXP.A, incorrect query sample

```
1    SELECT "b_name",
2           "b_quantity"
3    FROM   (SELECT "b_name",
4                   "b_quantity",
5                   ROWNUM AS "rn"
6            FROM   "books"
7            ORDER  BY "b_quantity" DESC)
8    WHERE  "rn" = 1
```

The result of this query is:

b_name	b_quantity
Eugene Onegin	2

We get this result because the subquery (lines 3-7) returns the following data:

b_name	b_quantity	rn
Course of Theoretical Physics	12	6
The Art of Computer Programming	7	7
Foundation and Empire	5	3
The Fisherman and the Golden Fish	3	2
The C++ Programming Language	3	5
Eugene Onegin	**2**	**1**
Programming Psychology	1	4

The line numbering here happened before the ordering, and the first number was assigned to the "Eugene Onegin" book, while the `ROW_NUMBER` function works correctly.

Solution 2.1.8.c[45].

If we want to show all books represented by an equal maximum number of copies, we need to find this maximum number and use the resulting value as a condition.

MySQL Solution 2.1.8.c

```
1    -- Option 1: using MAX
2    SELECT `b_name`,
3           `b_quantity`
4    FROM   `books`
5    WHERE  `b_quantity` = (SELECT MAX(`b_quantity`)
6                           FROM   `books`)

1    -- Option 2: using RANK (starting with MySQL 8)
2    SELECT `b_name`,
3           `b_quantity`
4    FROM   (SELECT `b_name`,
5                  `b_quantity`,
6                  RANK() OVER (ORDER BY `b_quantity` DESC) AS `rn`
7            FROM   `books`) AS `temporary_data`
8    WHERE  `rn` = 1
```

MS SQL Solution 2.1.8.c

```
1    -- Option 1: using MAX
2    SELECT [b_name],
3           [b_quantity]
4    FROM   [books]
5    WHERE  [b_quantity] = (SELECT MAX([b_quantity])
6                           FROM   [books])

1    -- Option 2: using RANK
2    SELECT [b_name],
3           [b_quantity]
4    FROM   (SELECT [b_name],
5                  [b_quantity],
6                  RANK() OVER (ORDER BY [b_quantity] DESC) AS [rn]
7            FROM   [books]) AS [temporary_data]
8    WHERE  [rn] = 1
```

Oracle Solution 2.1.8.c

```
1    -- Option 1: using MAX
2    SELECT "b_name",
3           "b_quantity"
4    FROM   "books"
5    WHERE  "b_quantity" = (SELECT MAX("b_quantity")
6                           FROM   "books")

1    -- Option 2: using RANK
2    SELECT "b_name",
3           "b_quantity"
4    FROM   (SELECT "b_name",
5                  "b_quantity",
6                  RANK() OVER (ORDER BY "b_quantity" DESC) AS "rn"
7            FROM   "books")
8    WHERE  "rn" = 1
```

In the case of MySQL before version 8, only the first option is available[3]: we use a subquery (lines 5-6) to find the maximum number of copies of books and then we use this number as a **`WHERE`** condition. The same solution works perfectly in MS SQL Server and Oracle.

MS SQL Server, Oracle (and MySQL starting with version 8) support so-called "window (ranking) functions" that allow to implement the second option. Note line 6 of this option: MySQL and MS SQL Server require explicit naming of subquery used as the data source, while Oracle does not (naming of subquery is available, but not required in this case).

The **`RANK`** function allows us to rank the resulting rows (i.e., "place" them 1st, 2nd, 3rd, 4th, and so on) by a specified condition (in our case, by decreasing number of copies of books). The books with the same number of copies will occupy the same places, and the first place will be taken by the books with the maximum number of copies. All that remains is to show the books that took first place.

To illustrate, let's consider the subquery represented by lines 4-7 of the second option of the solution:

MySQL Solution 2.1.8.c (query fragment)

```
1    SELECT `b_name`,
2           `b_quantity`,
3           RANK() OVER (ORDER BY `b_quantity` DESC) AS `rn`
4    FROM   `books`
```

MS SQL Solution 2.1.8.c (query fragment)

```
1    SELECT [b_name],
2           [b_quantity],
3           RANK() OVER (ORDER BY [b_quantity] DESC) AS [rn]
4    FROM   [books]
```

Oracle Solution 2.1.8.c (query fragment)

```
1    SELECT "b_name",
2           "b_quantity",
3           RANK() OVER (ORDER BY "b_quantity" DESC) AS "rn"
4    FROM   "books"
```

b_name	b_quantity	rn
Course of Theoretical Physics	12	1
The Art of Computer Programming	7	2
Foundation and Empire	5	3
The Fisherman and the Golden Fish	3	4
The C++ Programming Language	3	4
Eugene Onegin	2	6
Programming Psychology	1	7

[3] In fact, it is possible to emulate ranking (window) functions in MySQL. Examples of such emulation are presented in the exploration 2.1.8.EXP.D{48} and solutions 2.2.7.d{119}, 2.2.9.d{139}.

Exploration 2.1.8.EXP.B. what is faster, MAX or RANK? Let's use the previously created and filled with data (ten million records) **`test_counts`**{23} table and check.

The median time values after a hundred query executions are as follows:

	MySQL	MS QL Server	Oracle
MAX	0.012	0.009	0.341
RANK	5.621	6.940	0.818

The **`MAX`** option is faster in this case. However, exploration 2.2.7.EXP.A{125} will show the opposite situation, where the option with ranking will be much faster than the option with **`MAX`** function. Thus, the idea that a performance study is worth performing in a specific situation on a specific data set is confirmed once again.

Solution 2.1.8.d{45}.

To find the "absolute winner" for the number of copies, we use a set function **`ALL`** that allows us to compare some value to each element of a set.

MySQL Solution 2.1.8.d

```
1     -- Option 1: using ALL with subquery
2     SELECT `b_name`,
3            `b_quantity`
4     FROM   `books` AS `ext`
5     WHERE  `b_quantity` > ALL (SELECT `b_quantity`
6                                FROM   `books` AS `int`
7                                WHERE  `ext`.`b_id` != `int`.`b_id`)

1     -- Option 2: using Common Table Expression with RANK (MySQL 8 and newer)
2     WITH `ranked`
3          AS (SELECT `b_name`,
4                     `b_quantity`,
5                     RANK()
6                       OVER (
7                         ORDER BY `b_quantity` DESC) AS `rank`
8              FROM   `books`),
9          `counted`
10         AS (SELECT `rank`,
11                    COUNT(*) AS `competitors`
12             FROM   `ranked`
13             GROUP  BY `rank`)
14    SELECT `b_name`,
15           `b_quantity`
16    FROM   `ranked`
17          JOIN `counted`
18           USING (`rank`)
19    WHERE  `counted`.`rank` = 1
20       AND `counted`.`competitors` = 1
```

MS SQL Solution 2.1.8.d

```
1     -- Option 1: using ALL with subquery
2     SELECT [b_name],
3            [b_quantity]
4     FROM   [books] AS [ext]
5     WHERE  [b_quantity] > ALL (SELECT [b_quantity]
6                                FROM   [books] AS [int]
7                                WHERE  [ext].[b_id] != [int].[b_id])
```

```
1     -- Option 2: using Common Table Expression with RANK
2     WITH [ranked]
3          AS (SELECT [b_name],
4                     [b_quantity],
5                     RANK()
6                       OVER (
7                         ORDER BY [b_quantity] DESC) AS [rank]
8              FROM   [books]),
9          [counted]
10         AS (SELECT [rank],
11                    COUNT(*) AS [competitors]
12             FROM   [ranked]
13             GROUP  BY [rank])
14    SELECT [b_name],
15           [b_quantity]
16    FROM   [ranked]
17           JOIN [counted]
18             ON [ranked].[rank] = [counted].[rank]
19    WHERE  [counted].[rank] = 1
20           AND [counted].[competitors] = 1
```

Oracle Solution 2.1.8.d

```
1     -- Option 1: using ALL with subquery
2     SELECT "b_name",
3            "b_quantity"
4     FROM   "books" "ext"
5     WHERE  "b_quantity" > ALL (SELECT "b_quantity"
6                                FROM   "books" "int"
7                                WHERE  "ext"."b_id" != "int"."b_id")
```

```
1     -- Option 2: using Common Table Expression with RANK
2     WITH "ranked"
3          AS (SELECT "b_name",
4                     "b_quantity",
5                     RANK()
6                       OVER (
7                         ORDER BY "b_quantity" DESC) AS "rank"
8              FROM   "books"),
9          "counted"
10         AS (SELECT "rank",
11                    COUNT(*) AS "competitors"
12             FROM   "ranked"
13             GROUP  BY "rank")
14    SELECT "b_name",
15           "b_quantity"
16    FROM   "ranked"
17           JOIN "counted"
18             ON "ranked"."rank" = "counted"."rank"
19    WHERE  "counted"."rank" = 1
20           AND "counted"."competitors" = 1
```

The first options for all three DBMSes are identical (only in Oracle here, for naming a table, there must be no **AS** keyword between the original name and the alias). Now let's explain how it works.

The same table **books** appears twice in query 2.1.8.d: as **ext** (for the external part of the query) and **int** (for the internal part of the query). This is necessary to allow the DBMS to apply the condition presented in line 7, i.e.: "for each row of table **ext** select the value of the **b_quantity** field from all rows of **int** table except for the row currently considered in the **ext** table".

This is a fundamental principle of building the so-called "correlated subqueries", so let's show the logic of the DBMS graphically. So, we have seven books with identifiers from 1 to 7:

The row from the **ext** table	Which rows are analyzed in the **int** table
1	2, 3, 4, 5, 6, 7 {i.e., all but the 1st}
2	1, 3, 4, 5, 6, 7 {i.e., all but the 2nd}
3	1, 2, 4, 5, 6, 7 {i.e., all but the 3rd}
4	1, 2, 3, 5, 6, 7 {i.e., all but the 4th}
5	1, 2, 3, 4, 6, 7 {i.e., all but the 5th}
6	1, 2, 3, 4, 5, 7 {i.e., all but the 6th}
7	1, 2, 3, 4, 5, 6 {i.e., all but the 7th}

When selecting the values of the **b_quantity** field, the DBMS checks that the value selected from the **ext** table is greater than each of the values selected from the **int** table:

ext.b_quantity value	**int.b_quantity** set of values
2	3, 5, 1, 3, 12, 7
3	2, 5, 1, 3, 12, 7
5	2, 3, 1, 3, 12, 7
1	2, 3, 5, 3, 12, 7
3	2, 3, 5, 1, 12, 7
12	**2, 3, 5, 1, 3, 7**
7	2, 3, 5, 1, 3, 12

As we can see, only the book with 12 copies meets the condition. If at least one other book had the same number of copies, the condition would not be met for any row, and the query would return an empty result (which is shown at the beginning of this example, see "Possible expected result 2.1.8.d").

The second solution options are also identical for all three DBMSes (just note that MySQL supports Common Table Expressions only starting from version 8).

The idea of the second option is to get rid of correlated subqueries. For this purpose in lines 1-7 of this option in the **ranked** Common Table Expression data are prepared by ranking books by the number of their copies, then in lines 8-12 in the **counted** Common Table Expression the number of books ranked the same is counted, and in the main part of the query in lines 13-19 the obtained data are combined with imposing filter "must be first place and only one book must be in first place".

Exploration 2.1.8.EXP.C. What works faster, the correlated subquery option or the Common Table Expression followed by a `JOIN`? Let's run the corresponding 2.1.8.d queries a hundred times each on the "Big Library (for Experiments)" database.

The median time values after a hundred query executions are as follows:

	MySQL	MS QL Server	Oracle
Correlated subquery	0.299	0.198	0.402
Common Table Expression followed by a `JOIN`	0.265	0.185	0.378

The option with a Common Table Expression is just a little bit faster, but still, it is faster.

Exploration 2.1.8.EXP.D. Once again, let's demonstrate the difference in the performance of solutions based on correlated subqueries, aggregating functions, and ranking. Let's imagine that for each reader we need to show exactly one (any, if there are several, yet just one) record from subscriptions table corresponding to the reader's first visit to the library.

As a result, we expect to see:

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
2	1	1	2011-01-12	2011-02-12	N
3	3	3	2012-05-17	2012-07-17	Y
57	4	5	2012-06-11	2012-08-11	N

In each DBMS, there are three possible ways of getting this result (yes, there is also an option with a Common Table Expression, but we will not consider it deliberately, limiting ourselves to its analogue with subqueries):

- based on the correlated subqueries (note that in Oracle before version 12c we have to emulate the limitation on the number of selected records in a very non-trivial way in the subquery; the same approach is implemented through `LIMIT 1` and `TOP 1` in MySQL and MS SQL Server; a similar mechanism in Oracle is available only since version 12c);
- based on aggregating functions;
- based on the ranking (note that in MySQL before version 8 we have to emulate the behavior of the `ROW_NUMBER` function that is available in MS SQL Server and Oracle).

MySQL Exploration 2.1.8.EXP.D

```
-- Option 1: correlated subqueries
SELECT `sb_id`,
       `sb_subscriber`,
       `sb_book`,
       `sb_start`,
       `sb_finish`,
       `sb_is_active`
FROM   `subscriptions` AS `outer`
WHERE  `sb_id` = (SELECT `sb_id`
                  FROM   `subscriptions` AS `inner`
                  WHERE  `outer`.`sb_subscriber` = `inner`.`sb_subscriber`
                  ORDER  BY `sb_start` ASC
                  LIMIT  1)
```

```
-- Option 2: aggregating functions
SELECT `sb_id`,
       `subscriptions`.`sb_subscriber`,
       `sb_book`,
       `sb_start`,
       `sb_finish`,
       `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` IN (SELECT MIN(`sb_id`)
                   FROM   `subscriptions`
                          JOIN (SELECT `sb_subscriber`,
                                       MIN(`sb_start`) AS `min_date`
                                FROM   `subscriptions`
                                GROUP  BY `sb_subscriber`) AS `prepared`
                            ON `subscriptions`.`sb_subscriber` =
                               `prepared`.`sb_subscriber`
                               AND `subscriptions`.`sb_start` =
                                   `prepared`.`min_date`
                   GROUP  BY `prepared`.`sb_subscriber`,
                             `prepared`.`min_date`)
```

```
-- Option 3: ranking (before MySQL 8)
SELECT `subscriptions`.`sb_id`,
       `sb_subscriber`,
       `sb_book`,
       `sb_start`,
       `sb_finish`,
       `sb_is_active`
FROM   `subscriptions`
       JOIN (SELECT `sb_id`,
                    @row_num := IF(@prev_value = `sb_subscriber`,
                                   @row_num + 1,
                                   1) AS `visit`,
                    @prev_value := `sb_subscriber`
             FROM   `subscriptions`,
                    (SELECT @row_num := 1) AS `x`,
                    (SELECT @prev_value := '') AS `y`
             ORDER  BY `sb_subscriber` ASC,
                       `sb_start` ASC) AS `prepared`
         ON `subscriptions`.`sb_id` = `prepared`.`sb_id`
WHERE  `visit` = 1
```

MySQL Exploration 2.1.8.EXP.D (continued)

```mysql
-- Option 3: ranking (MySQL 8 and newer)
SELECT `subscriptions`.`sb_id`,
       `sb_subscriber`,
       `sb_book`,
       `sb_start`,
       `sb_finish`,
       `sb_is_active`
FROM   `subscriptions`
       JOIN (SELECT `sb_id`,
                    ROW_NUMBER()
                      OVER (
                        PARTITION BY `sb_subscriber`
                        ORDER BY `sb_start` ASC) AS `visit`
             FROM   `subscriptions`) AS `prepared`
         ON `subscriptions`.`sb_id` = `prepared`.`sb_id`
WHERE  `visit` = 1
```

MS SQL Exploration 2.1.8.EXP.D

```mssql
-- Option 1: correlated subqueries
SELECT [sb_id],
       [sb_subscriber],
       [sb_book],
       [sb_start],
       [sb_finish],
       [sb_is_active]
FROM   [subscriptions] AS [outer]
WHERE  [sb_id] = (SELECT TOP 1 [sb_id]
                  FROM   [subscriptions] AS [inner]
                  WHERE  [outer].[sb_subscriber] = [inner].[sb_subscriber]
                  ORDER  BY [sb_start] ASC)

-- Option 2: aggregating functions
SELECT [sb_id],
       [subscriptions].[sb_subscriber],
       [sb_book],
       [sb_start],
       [sb_finish],
       [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] IN (SELECT MIN([sb_id])
                   FROM   [subscriptions]
                          JOIN (SELECT [sb_subscriber],
                                       MIN([sb_start]) AS [min_date]
                                FROM   [subscriptions]
                                GROUP  BY [sb_subscriber]) AS [prepared]
                            ON [subscriptions].[sb_subscriber] =
                               [prepared].[sb_subscriber]
                               AND [subscriptions].[sb_start] =
                                   [prepared].[min_date]
                   GROUP  BY [prepared].[sb_subscriber],
                             [prepared].[min_date])
```

MS SQL Exploration 2.1.8.EXP.D (continued)

```
1    -- Option 3: ranking
2    SELECT [subscriptions].[sb_id],
3           [sb_subscriber],
4           [sb_book],
5           [sb_start],
6           [sb_finish],
7           [sb_is_active]
8    FROM   [subscriptions]
9           JOIN (SELECT [sb_id],
10                       ROW_NUMBER()
11                         OVER (
12                           PARTITION BY [sb_subscriber]
13                           ORDER BY [sb_start] ASC) AS [visit]
14                FROM   [subscriptions]) AS [prepared]
15             ON [subscriptions].[sb_id] = [prepared].[sb_id]
16   WHERE  [visit] = 1
```

Oracle Exploration 2.1.8.EXP.D

```
1    -- Option 1: correlated subqueries (before Oracle 12c)
2    SELECT "sb_id",
3           "sb_subscriber",
4           "sb_book",
5           "sb_start",
6           "sb_finish",
7           "sb_is_active"
8    FROM   "subscriptions" "outer"
9    WHERE  "sb_id" = (SELECT DISTINCT FIRST_VALUE("inner"."sb_id")
10                                     OVER (ORDER BY "inner"."sb_start" ASC)
11                     FROM   "subscriptions" "inner"
12                     WHERE  "outer"."sb_subscriber" = "inner"."sb_subscriber")
```

```
1    -- Option 1: correlated subqueries (Oracle 12c and newer)
2    SELECT "sb_id",
3           "sb_subscriber",
4           "sb_book",
5           "sb_start",
6           "sb_finish",
7           "sb_is_active"
8    FROM   "subscriptions" "outer"
9    WHERE  "sb_id" = (SELECT "sb_id"
10                     FROM   "subscriptions" "inner"
11                     WHERE  "outer"."sb_subscriber" = "inner"."sb_subscriber"
12                     ORDER  BY "sb_start" ASC
13                     FETCH FIRST 1 ROWS ONLY)
```

```
1    -- Option 2: aggregating functions
2    SELECT "sb_id",
3           "subscriptions"."sb_subscriber",
4           "sb_book",
5           "sb_start",
6           "sb_finish",
7           "sb_is_active"
8    FROM   "subscriptions"
9    WHERE  "sb_id" IN (SELECT MIN("sb_id")
10                      FROM   "subscriptions"
11                             JOIN (SELECT "sb_subscriber",
12                                          MIN("sb_start") AS "min_date"
13                                   FROM   "subscriptions"
14                                   GROUP  BY "sb_subscriber") "prepared"
15                               ON "subscriptions"."sb_subscriber" =
16                                  "prepared"."sb_subscriber"
17                                  AND "subscriptions"."sb_start" =
18                                      "prepared"."min_date"
19                      GROUP  BY "prepared"."sb_subscriber",
20                                "prepared"."min_date")
```

Oracle Exploration 2.1.8.EXP.D (continued)

```
1    -- Option 3: ranking
2    SELECT "subscriptions"."sb_id",
3           "sb_subscriber",
4           "sb_book",
5           "sb_start",
6           "sb_finish",
7           "sb_is_active"
8    FROM   "subscriptions"
9           JOIN (SELECT "sb_id",
10                       ROW_NUMBER()
11                         OVER (
12                           partition BY "sb_subscriber"
13                           ORDER BY "sb_start" ASC) AS "visit"
14                FROM   "subscriptions") "prepared"
15             ON "subscriptions"."sb_id" = "prepared"."sb_id"
16   WHERE  "visit" = 1
```

After performing each of the queries once (you can easily understand why only once when you see the results) on the “Big Library (for Experiments)” database, the following time values were obtained:

	MySQL	MS SQL Server	Oracle
Correlated subqueries	43:12:17.674	247:53:21.645	763:32:22.878
Aggregating functions	44:17:12.736	688:43:58.244	041:19:43.344
Ranking	00:18:48.828	000:00:41.511	000:02:32.274

Each of the DBMSes was fastest in one type of query, but in all three, the ranking-based solution was the undisputed leader (compare, for example, the best and the worst results for MS SQL Server: 41.5 seconds instead of almost a month).

Task 2.1.8.TSK.A: show the identifier of the one (any) reader who borrowed the most books from the library.

Task 2.1.8.TSK.B: show the identifiers of all “most-readers” who borrowed the most books from the library.

Task 2.1.8.TSK.C: show the identifier of the “absolute winning” reader who borrowed more books from the library than any other reader.

Task 2.1.8.TSK.D: write a second option of problem 2.1.8.d solution (based on Common Table Expressions) for MySQL, emulating Common Table Expression through subqueries.

2.1.9. Example 9: Calculating Average Values of Aggregated Data

Problem 2.1.9.a{59}: show how many copies of books each reader now has on average.

Problem 2.1.9.b{59}: show how many books each reader now has on average.

Problem 2.1.9.c{60}: show how for how many days, on average, readers borrow books (consider only cases where books were returned).

Problem 2.1.9.d{60}: show how many days, on average, readers read a book (take into account both cases when the book was returned and when it was not returned).

The difference between the problems 2.1.9.a and 2.1.9.b is that the first takes into account the cases “the reader has several copies of the same book”, while the second will consider any number of such duplicates as one book.

The difference between problems 2.1.9.c and 2.1.9.d is that to solve problem 2.1.9.c the data from the table is enough, and to solve problem 2.1.9.d we have to determine the current date.

Expected result 2.1.9.a.

avg_books
2.5

Expected result 2.1.9.b.

avg_books
2.5

The expected results of 2.1.9.a and 2.1.9.b are the same on the available data set, since no reader currently has two or more copies of the same book, but you can change the data in the database and see how the query results change.

Expected result 2.1.9.c.

avg_days
46

Expected result 2.1.9.d.

avg_days
560.6364

Note: the expected result of 2.1.9.d depends on the date on which the query was run. Therefore it is bound to be different for you!

Solution 2.1.9.a[58].

See explanation below after the solution 2.1.9.b[59].

MySQL Solution 2.1.9.a

```
1    SELECT AVG(`books_per_subscriber`) AS `avg_books`
2    FROM   (SELECT COUNT(`sb_book`) AS `books_per_subscriber`
3            FROM   `subscriptions`
4            WHERE  `sb_is_active` = 'Y'
5            GROUP  BY `sb_subscriber`) AS `count_subquery`
```

MS SQL Solution 2.1.9.a

```
1    SELECT AVG(CAST([books_per_subscriber] AS FLOAT)) AS [avg_books]
2    FROM   (SELECT COUNT([sb_book]) AS [books_per_subscriber]
3            FROM   [subscriptions]
4            WHERE  [sb_is_active] = 'Y'
5            GROUP  BY [sb_subscriber]) AS [count_subquery]
```

Oracle Solution 2.1.9.a

```
1    SELECT AVG("books_per_subscriber") AS "avg_books"
2    FROM   (SELECT COUNT("sb_book") AS "books_per_subscriber"
3            FROM   "subscriptions"
4            WHERE  "sb_is_active" = 'Y'
5            GROUP  BY "sb_subscriber")
```

Solution 2.1.9.b[58].

MySQL Solution 2.1.9.b

```
1    SELECT AVG(`books_per_subscriber`) AS `avg_books`
2    FROM   (SELECT COUNT(DISTINCT `sb_book`) AS `books_per_subscriber`
3            FROM   `subscriptions`
4            WHERE  `sb_is_active` = 'Y'
5            GROUP  BY `sb_subscriber`) AS `count_subquery`
```

MS SQL Solution 2.1.9.b

```
1    SELECT AVG(CAST([books_per_subscriber] AS FLOAT)) AS [avg_books]
2    FROM   (SELECT COUNT(DISTINCT [sb_book]) AS [books_per_subscriber]
3            FROM   [subscriptions]
4            WHERE  [sb_is_active] = 'Y'
5            GROUP  BY [sb_subscriber]) AS [count_subquery]
```

Oracle Solution 2.1.9.b

```
1    SELECT AVG("books_per_subscriber") AS "avg_books"
2    FROM   (SELECT COUNT(DISTINCT "sb_book") AS "books_per_subscriber"
3            FROM   "subscriptions"
4            WHERE  "sb_is_active" = 'Y'
5            GROUP  BY "sb_subscriber")
```

The essence of solutions 2.1.9.a[59] and 2.1.9.b[59] is to first prepare aggregated data (subquery on lines 2-5 of all six queries 2.1.9.a-2.1.9.b presented above) and then calculate the average of these pre-prepared values.

The difference in solving problems 2.1.9.a[58] and 2.1.9.b[58] is the use of the **`DISTINCT`** keyword (line 2 of all six queries 2.1.9.a-2.1.9.b) in the second case, which allows us to ignore duplicate books.

The difference in the solutions for the three different DBMSes is that the **`AVG`** function argument must be first converted to **`FLOAT`** in MS SQL Server and that it is not necessary to name the subquery in Oracle. Otherwise, solutions 2.1.9.a[59] and 2.1.9.b[59] are identical for all three DBMSes.

For clarity, let's show what data were retrieved by the subqueries (lines 2-5 of all six queries 2.1.9.a-2.1.9.b):

books_per_subscriber
3
2

Solution 2.1.9.c[58].

MySQL Solution 2.1.9.c

```
1    SELECT AVG(DATEDIFF(`sb_finish`, `sb_start`)) AS `avg_days`
2    FROM   `subscriptions`
3    WHERE  `sb_is_active` = 'N'
```

MS SQL Solution 2.1.9.c

```
1    SELECT AVG(CAST (DATEDIFF(day, [sb_start], [sb_finish]) AS FLOAT))
2           AS [avg_days]
3    FROM   [subscriptions]
4    WHERE  [sb_is_active] = 'N'
```

Oracle Solution 2.1.9.c

```
1    SELECT AVG("sb_finish" - "sb_start") AS "avg_days"
2    FROM   "subscriptions"
3    WHERE  "sb_is_active" = 'N'
```

The solution to problem 2.1.9.c[58] is the same for all three DBMSes, except for the syntax for calculating the difference in days between two dates (line 1 in each of the three queries 2.1.9.c). The results of calculating the date difference are as follows (this data goes to the input of the **`AVG`** function):

data
31
61
61
61
31
31

Solution 2.1.9.d[60].

The biggest question here is how to determine the time range to be considered. Its beginning is stored in the **`sb_start`** field, but the end is not so simple. If the book has been returned, the end date of the reading period can be the value of the **`sb_finish`** field, if it is in the past. If the book has not been returned and the **`sb_finish`** value is in the past, the current date can be taken as the end date of the reading period

But what if the value of **`sb_finish`** is in the future? The right answer to this question in real life can only be obtained from the customer of the application being developed. For training purposes, we will use the current date if the book has already been returned, and **`sb_finish`** if it has not yet been returned.

So, we have four options for calculating book reading time:

- **sb_finish** is in the past, book is returned: **sb_finish - sb_start**.
- **sb_finish** is in the past, the book is not returned: **CURRENT_DATE - sb_start**.
- **sb_finish** is in the future, the book is returned: **CURRENT_DATE - sb_start**.
- **sb_finish** is in the future, the book is not returned: **sb_finish - sb_start**.

It is easy to see that there are only two calculation algorithms, but each of them is activated by two independent conditions. The easiest way to solve this is to combine the results of the two queries using the **UNION** operator. It is important to remember that **UNION** by default works in **DISTINCT** mode, i.e., we should explicitly write **UNION ALL**.

MySQL | Solution 2.1.9.d

```
1   SELECT AVG(`diff`) AS `avg_days`
2   FROM   (
3            SELECT DATEDIFF(`sb_finish`, `sb_start`) AS `diff`
4            FROM   `subscriptions`
5            WHERE     ( `sb_finish` <= CURDATE() AND `sb_is_active` = 'N' )
6                   OR ( `sb_finish` > CURDATE() AND `sb_is_active` = 'Y' )
7            UNION ALL
8            SELECT DATEDIFF(CURDATE(), `sb_start`) AS `diff`
9            FROM   `subscriptions`
10           WHERE     ( `sb_finish` <= CURDATE() AND `sb_is_active` = 'Y' )
11                  OR ( `sb_finish` > CURDATE() AND `sb_is_active` = 'N' )
12         ) AS `diffs`
```

Although this query is cumbersome, it is simple. Line 1 solves the main task of calculating the average value, while lines 2-13 only prepare the necessary data. Line 7 uses the just-mentioned **UNION ALL** operator, which combines the results of two separate queries in lines 3-6 and 8-11, respectively. The volumetric **WHERE** constructs in lines 5-6 and 10-11 define the conditions that activate one of the two algorithms for calculating the book reading time

These are the data sets returned by the queries in lines 3-6 and 8-11.

The first query (lines 3-6) returns:

diff
31
61
61
61
3684
31

The second query (lines 8-11) returns.

diff
1266
458
458
28
28

Solutions for MS SQL Server and Oracle follow the same logic and differ only in the syntax for getting the difference in days between dates (and the necessity to convert the argument of the **AVG** function to **FLOAT** for MS SQL Server).

MS SQL Solution 2.1.9.d

```
1     SELECT AVG(CAST([diff] AS FLOAT)) AS [avg_days]
2     FROM   (
3            SELECT DATEDIFF(day, [sb_start], [sb_finish]) AS [diff]
4            FROM   [subscriptions]
5            WHERE     ( [sb_finish] <= CONVERT(date, GETDATE())
6                         AND [sb_is_active] = 'N' )
7                  OR ( [sb_finish] > CONVERT(date, GETDATE())
8                         AND [sb_is_active] = 'Y' )
9            UNION ALL
10           SELECT DATEDIFF(day, [sb_start], CONVERT(date, GETDATE())) AS [diff]
11           FROM   [subscriptions]
12           WHERE     ( [sb_finish] <= CONVERT(date, GETDATE())
13                        AND [sb_is_active] = 'Y' )
14                 OR ( [sb_finish] > CONVERT(date, GETDATE())
15                        AND [sb_is_active] = 'N' )
16           ) AS [diffs]
```

Oracle Solution 2.1.9.d

```
1     SELECT AVG("diff") AS "avg_days"
2     FROM   (
3            SELECT ("sb_finish" - "sb_start") AS "diff"
4            FROM   "subscriptions"
5            WHERE     ( "sb_finish" <= TRUNC(SYSDATE) AND "sb_is_active" = 'N' )
6                  OR ( "sb_finish" > TRUNC(SYSDATE) AND "sb_is_active" = 'Y' )
7            UNION ALL
8            SELECT (TRUNC(SYSDATE) - "sb_start") AS "diff"
9            FROM   "subscriptions"
10           WHERE     ( "sb_finish" <= TRUNC(SYSDATE) AND "sb_is_active" = 'Y' )
11                 OR ( "sb_finish" > TRUNC(SYSDATE) AND "sb_is_active" = 'N' )
12           )
```

In the solution for Oracle, it is also worth noting that there is no such thing as “just date without time”, so we have to use `TRUNC(SYSDATE)` construct to “cut off” time from date. Otherwise, the result of subtraction in line 8 will return not an integer but a fractional number of days (which is different from how MySQL and MS SQL Server behave), and also all conditions involving a date may work unexpectedly

And one more obvious but noteworthy fact: in all queries for all DBMSes, we used `<= current_date` and `> current_date` in conditions related to current date, i.e., we included “today” in one of the ranges. If we use two strict inequalities or two non-strict ones, we risk either “losing” records with `sb_finish` value coinciding with current date or considering such cases twice.

If you have carefully studied the solution we just discussed, you must have wondered how the correctness of the calculations is affected by the record from the `subscriptions` table with the identifier 91 (in which the return date is in the past in relation to the date of the book delivery):

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
91	4	1	**2015-10-07**	**2015-03-07**	Y

The answer is simple and disappointing: yes, because of this mistake the result is distorted. What to do? Nothing. We do not have the luxury of considering the possibility of such errors in every query (e.g., the date of the book delivery may have been in the future). Control over such situations should be assigned to `INSERT` and `UPDATE` operations. A corresponding example (see problem 4.2.1.a{329}) will be discussed in the section{286} about triggers.

There is another problem with date differences that is worth keeping in mind: in everyday life, we are used to rounding these values, while the DBMS does not perform this operation. Check yourself: how many years have passed between 2011-01-01 and 2012-01-01-01? One year, right? And between 2011-01-01 and 2012-12-31?

Exploration 2.1.9.EXP.A. Checking DBMS's behavior when calculating the distance between two dates in years.

MySQL Exploration 2.1.9.EXP.A

```
1    SELECT YEAR('2012-01-01') - YEAR('2011-01-01') - (
2           DATE_FORMAT('2012-01-01', '%m%d') <
3           DATE_FORMAT('2011-01-01', '%m%d'))

1    SELECT YEAR('2012-12-31') - YEAR('2011-01-01') - (
2           DATE_FORMAT('2012-12-31', '%m%d') <
3           DATE_FORMAT('2011-01-01', '%m%d'))
```

MS SQL Exploration 2.1.9.EXP.A

```
1    SELECT DATEDIFF(year, '2011-01-01', '2012-01-01')

1    SELECT DATEDIFF(year, '2011-01-01', '2012-12-31')
```

Oracle Exploration 2.1.9.EXP.A

```
1    SELECT FLOOR(MONTHS_BETWEEN(DATE '2012-01-01', DATE '2011-01-01') / 12)
2    FROM dual;

1    SELECT FLOOR(MONTHS_BETWEEN(DATE '2012-12-31', DATE '2011-01-01') / 12)
2    FROM dual;
```

All six queries 2.1.6.EXP.A will return the same result: 1, i.e., one year has passed between the specified dates from the DBMS point of view. And it is true: "one full year has passed".

Also note how differently the problem of calculating the difference between dates in years is solved in different DBMSes. Especially interesting are lines 2-3 in queries for MySQL: they allow us to get the correct result when the year values are different, but in fact the year has not yet passed (for example, 2011-05-01 and 2012-04-01).

Task 2.1.9.TSK.A: show how many copies of books there are in the library on average.

Task 2.1.9.TSK.B: show in days how long on average readers have been registered in the library (the range from the first date the reader received a book to the current date).

2.1.10. Example 10: Data Grouping

Problem 2.1.10.a{64}: show for each year how many times readers borrowed books that year.

Problem 2.1.10.b{65}: show for each year how many readers (per that year) visited the library.

Problem 2.1.10.c{66}: show how many books were returned and not returned to the library.

Expected result 2.1.10.a.

year	books_taken
2011	2
2012	3
2014	3
2015	3

Expected result 2.1.10.b.

year	subscribers
2011	1
2012	3
2014	2
2015	2

Expected result 2.1.10.c.

status	books
Returned	6
Not returned	5

As the title of the section implies, all three tasks will be solved by data grouping, i.e., by using the `GROUP BY` clause.

Solution 2.1.10.a{64}.

MySQL Solution 2.1.10.a

```
1    SELECT YEAR(`sb_start`) AS `year`,
2           COUNT(`sb_id`)   AS `books_taken`
3    FROM   `subscriptions`
4    GROUP  BY `year`
5    ORDER  BY `year`
```

MS SQL Solution 2.1.10.a

```
1    SELECT YEAR([sb_start]) AS [year],
2           COUNT([sb_id])   AS [books_taken]
3    FROM   [subscriptions]
4    GROUP  BY YEAR([sb_start])
5    ORDER  BY [year]
```

Oracle Solution 2.1.10.a

```
1    SELECT EXTRACT(year FROM "sb_start") AS "year",
2           COUNT("sb_id")                AS "books_taken"
3    FROM   "subscriptions"
4    GROUP  BY EXTRACT(year FROM "sb_start")
5    ORDER  BY "year"
```

Note the difference in line 4 in queries 2.1.10.a: MySQL allows us to refer in **GROUP BY** to the name of the expression just calculated, while MS SQL Server and Oracle do not allow this.

Earlier we have already considered the logic of grouping{22}, but let's emphasize it once again. After extracting the value of the year and "combining cells according to the equality of values" the result obtained by the DBMS can be conventionally represented as follows:

year	grouping result	how many cells are merged
2011	2011	2
2011		
2012	2012	3
2012		
2012		
2014	2014	3
2014		
2014		
2015	2015	3
2015		
2015		

Solution 2.1.10.b{64}.

MySQL Solution 2.1.10.b

```
1    SELECT YEAR(`sb_start`)                 AS `year`,
2           COUNT(DISTINCT `sb_subscriber`)  AS `subscribers`
3    FROM   `subscriptions`
4    GROUP  BY `year`
5    ORDER  BY `year`
```

MS SQL Solution 2.1.10.b

```
1    SELECT YEAR([sb_start])                 AS [year],
2           COUNT(DISTINCT [sb_subscriber])  AS [subscribers]
3    FROM   [subscriptions]
4    GROUP  BY YEAR([sb_start])
5    ORDER  BY [year]
```

Oracle Solution 2.1.10.b

```
1    SELECT EXTRACT(year FROM "sb_start")    AS "year",
2           COUNT(DISTINCT "sb_subscriber")  AS "subscribers"
3    FROM   "subscriptions"
4    GROUP  BY EXTRACT(year FROM "sb_start")
5    ORDER  BY "year"
```

Solution 2.1.10.b{65} is very similar to solution 2.1.10.a{64}: the only difference is that in the first case we are interested in all records for each year, and in the second we are interested in the number of readers' identifiers without duplication for each year. Let's show this graphically:

year	sb_subscriber	grouping by year	number of unique readers' identifiers
2011	1	2011	1
2011	1		
2012	1	2012	3
2012	3		
2012	4		
2014	1	2014	2
2014	3		
2014	3		
2015	1	2015	2
2015	4		
2015	4		

Solution 2.1.10.c{64}.

MySQL Solution 2.1.10.c

```
1    SELECT IF(`sb_is_active` = 'Y', 'Not returned', 'Returned') AS `status`,
2           COUNT(`sb_id`)                                       AS `books`
3    FROM   `subscriptions`
4    GROUP  BY `status`
5    ORDER  BY `status` DESC
```

MS SQL Solution 2.1.10.c

```
1    SELECT (CASE
2              WHEN [sb_is_active] = 'Y'
3              THEN 'Not returned'
4              ELSE 'Returned'
5             END)           AS [status],
6           COUNT([sb_id]) AS [books]
7    FROM   [subscriptions]
8    GROUP  BY (CASE
9                 WHEN [sb_is_active] = 'Y'
10                THEN 'Not returned'
11                ELSE 'Returned'
12               END)
13   ORDER  BY [status] DESC
```

Oracle Solution 2.1.10.c

```
1    SELECT (CASE
2              WHEN "sb_is_active" = 'Y'
3              THEN 'Not returned'
4              ELSE 'Returned'
5             END)           AS "status",
6           COUNT("sb_id") AS "books"
7    FROM   "subscriptions"
8    GROUP  BY (CASE
9                 WHEN "sb_is_active" = 'Y'
10                THEN 'Not returned'
11                ELSE 'Returned'
12               END)
13   ORDER  BY "status" DESC
```

Solution 2.1.10.c has one difficulty: we need to get human-readable “Not returned” and “Returned” labels based on the values of the `sb_is_active` field `Y` (subscription is active, i.e., the book is not returned) and `N` (subscription is inactive, i.e., the book is returned).

In MySQL, all the necessary actions are placed in one line (line 1 of the query), and because of MySQL’s ability to refer to the name of the calculated expression in `GROUP BY`, there is no need to repeat this entire construction in line 4.

MS SQL Server and Oracle support the same, but slightly more cumbersome syntax: lines 1-5 of the queries for these DBMSes contain the necessary conversions, and the duplication of the same code in lines 8-12 is caused by the fact that these DBMSes do not allow us to refer in `GROUP BY` to the calculated expression by its name.

Again, let’s show graphically what kind of data the DBMS has to work with:

sb_is_active (initial value)	status (conversion result)	grouping	calculation
N	Returned	Returned	6
N	Returned		
N	Returned		
N	Returned		
N	Returned		
N	Returned		
Y	Not returned	Not returned	5
Y	Not returned		
Y	Not returned		
Y	Not returned		
Y	Not returned		

Exploration 2.1.10.EXP.A. How fast, in principle, does grouping work on large amounts of data? Let’s use the “Big Library (for Experiments)” database and calculate how many books are taken be each reader (subscriber).

MySQL Exploration 2.1.10.EXP.A

```
1    SELECT `sb_subscriber`,
2           COUNT(`sb_id`) AS `books_taken`
3    FROM   `subscriptions`
4    WHERE  `sb_is_active` = 'Y'
5    GROUP  BY `sb_subscriber`
```

MS SQL Exploration 2.1.10.EXP.A

```
1    SELECT [sb_subscriber],
2           COUNT([sb_id]) AS [books_taken]
3    FROM   [subscriptions]
4    WHERE  [sb_is_active] = 'Y'
5    GROUP  BY [sb_subscriber]
```

Oracle Exploration 2.1.10.EXP.A

```
1    SELECT "sb_subscriber",
2           COUNT("sb_id") AS "books_taken"
3    FROM   "subscriptions"
4    WHERE  "sb_is_active" = 'Y'
5    GROUP  BY "sb_subscriber"
```

Median time values after executing 100 times each of the queries 2.1.10.EXP.A:

MySQL	MS SQL Server	Oracle
34.333	8.189	2.306

On the one hand, the results do not look daunting, but if we increase the amount of data by a factor of 10, 100, 1000, etc., the execution time will already be measured in hours or even days.

Task 2.1.10.TSK.A: rewrite solution 2.1.10.c, so that when counting returned and unreturned books, DBMS operates with original values of `sb_is_active` field (i.e., `Y` and N), and conversion to “Returned” and “Not returned” takes place after counting.

2.2. Data Selection from Multiple Tables

2.2.1. Example 11: Using JOINs to Obtain Human-readable Data

The following two problems have already been mentioned before[16], now we will consider them in detail.

Problem 2.2.1.a[70]: show all human-readable information about all books (i.e., title, author, genre).

Problem 2.2.1.b[72]: show all human-readable information about all library visits (i.e., reader's name and title of a book taken).

Expected result 2.2.1.a.

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin	Classic
The Fisherman and the Golden Fish	Alexander Pushkin	Classic
Course of Theoretical Physics	Lev Landau	Classic
Course of Theoretical Physics	Evgeny Lifshitz	Classic
The Art of Computer Programming	Donald Knuth	Classic
Eugene Onegin	Alexander Pushkin	Poetry
The Fisherman and the Golden Fish	Alexander Pushkin	Poetry
Programming Psychology	Dale Carnegie	Programming
Programming Psychology	Bjarne Stroustrup	Programming
The C++ Programming Language	Bjarne Stroustrup	Programming
The Art of Computer Programming	Donald Knuth	Programming
Programming Psychology	Dale Carnegie	Psychology
Programming Psychology	Bjarne Stroustrup	Psychology
Foundation and Empire	Isaac Asimov	Science Fiction

Expected result 2.2.1.b.

b_name	s_id	s_name	sb_start	sb_finish
Eugene Onegin	1	Ivanov I.I.	2011-01-12	2011-02-12
The Fisherman and the Golden Fish	1	Ivanov I.I.	2012-06-11	2012-08-11
The Art of Computer Programming	1	Ivanov I.I.	2014-08-03	2014-10-03
Programming Psychology	1	Ivanov I.I.	2015-10-07	2015-11-07
Foundation and Empire	1	Ivanov I.I.	2011-01-12	2011-02-12
Foundation and Empire	3	Sidorov S.S.	2012-05-17	2012-07-17
The C++ Programming Language	3	Sidorov S.S.	2014-08-03	2014-10-03
Eugene Onegin	3	Sidorov S.S.	2014-08-03	2014-09-03
The C++ Programming Language	4	Sidorov S.S.	2012-06-11	2012-08-11
Eugene Onegin	4	Sidorov S.S.	2015-10-07	2015-03-07
Programming Psychology	4	Sidorov S.S.	2015-10-08	2025-11-08

Solution 2.2.1.a[69].

MySQL Solution 2.2.1.a

```
SELECT `b_name`,
       `a_name`,
       `g_name`
FROM   `books`
       JOIN `m2m_books_authors` USING(`b_id`)
       JOIN `authors` USING(`a_id`)
       JOIN `m2m_books_genres` USING(`b_id`)
       JOIN `genres` USING(`g_id`)
```

MS SQL Solution 2.2.1.a

```
SELECT [b_name],
       [a_name],
       [g_name]
FROM   [books]
       JOIN [m2m_books_authors]
         ON [books].[b_id] = [m2m_books_authors].[b_id]
       JOIN [authors]
         ON [m2m_books_authors].[a_id] = [authors].[a_id]
       JOIN [m2m_books_genres]
         ON [books].[b_id] = [m2m_books_genres].[b_id]
       JOIN [genres]
         ON [m2m_books_genres].[g_id] = [genres].[g_id]
```

Oracle Solution 2.2.1.a

```
SELECT "b_name",
       "a_name",
       "g_name"
FROM   "books"
       JOIN "m2m_books_authors" USING("b_id")
       JOIN "authors" USING("a_id")
       JOIN "m2m_books_genres" USING("b_id")
       JOIN "genres" USING("g_id")
```

The key difference between solutions for MySQL and Oracle and solutions for MS SQL Server is that these two DBMSes support a special syntax for specifying fields to be joined: if such fields have the same name in the tables being joined, we can use **`USING(field)`** instead of the **`ON first_table.field = second_table.field`**, which often greatly improves readability of the query.

Despite the fact that problem 2.2.1.a is probably the simplest case of using **`JOIN`**, in it we are joining five tables in one query, so let's show graphically on the example of one row how the final result is formed.

There are other fields in the tables in question, but only those that are of interest in the context of this task are shown here.

First step:

```
[books] JOIN [m2m_books_authors]
ON [books].[b_id] = [m2m_books_authors].[b_id]
```

The **books** table:

b_id	b_name
1	Eugene Onegin

The **m2m_books_authors** table:

b_id	a_id
1	7

Intermediate result:

b_id	b_name	a_id
1	Eugene Onegin	7

Second step:

```
{first step result} JOIN [authors]
ON [m2m_books_authors].[a_id] = [authors].[a_id]
```

b_id	b_name	a_id
1	Eugene Onegin	7

The **authors** table:

a_id	a_name
7	Alexander Pushkin

Intermediate result:

b_id	b_name	a_name
1	Eugene Onegin	Alexander Pushkin

Third step:

```
{second step result} JOIN [m2m_books_genres]
ON [books].[b_id] = [m2m_books_genres].[b_id]
```

b_id	b_name	a_name
1	Eugene Onegin	Alexander Pushkin

The **m2m_books_genres** table:

b_id	g_id
1	1
1	5

Intermediate result:

b_name	a_name	g_id
Eugene Onegin	Alexander Pushkin	1
Eugene Onegin	Alexander Pushkin	5

Fourth step:

```
{third step result} JOIN [genres]
ON [m2m_books_genres].[g_id] = [genres].[g_id]
```

b_name	a_name	g_id
Eugene Onegin	Alexander Pushkin	1
Eugene Onegin	Alexander Pushkin	5

The `genres` table:

g_id	g_name
1	Poetry
5	Classic

Final result:

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin	Poetry
Eugene Onegin	Alexander Pushkin	Classic

The same sequence of actions is repeated for each row in the `books` table, which leads to the expected result 2.2.1.a.

Solution 2.2.1.b[69].

MySQL Solution 2.2.1.b

```
SELECT `b_name`,
       `s_id`,
       `s_name`,
       `sb_start`,
       `sb_finish`
FROM   `books`
       JOIN `subscriptions`
         ON `b_id` = `sb_book`
       JOIN `subscribers`
         ON `sb_subscriber` = `s_id`
```

MS SQL Solution 2.2.1.b

```
SELECT [b_name],
       [s_id],
       [s_name],
       [sb_start],
       [sb_finish]
FROM   [books]
       JOIN [subscriptions]
         ON [b_id] = [sb_book]
       JOIN [subscribers]
         ON [sb_subscriber] = [s_id]
```

Oracle | Solution 2.2.1.b

```
SELECT "b_name",
       "s_id",
       "s_name",
       "sb_start",
       "sb_finish"
FROM   "books"
       JOIN "subscriptions"
         ON "b_id" = "sb_book"
       JOIN "subscribers"
         ON "sb_subscriber" = "s_id"
```

The logic of solution 2.2.1.b{72} is completely equivalent to that of solution 2.2.1.a{70} (here, even fewer tables have to be joined, just three instead of five). Pay attention to the fact that if the **`JOIN`** occurs not by identically named fields, in MySQL and Oracle we also have to use the **`ON`** syntax instead of **`USING`**.

Task 2.2.1.TSK.A: show a list of books that have more than one author.

Task 2.2.1.TSK.B: show a list of books belonging to exactly one genre.

2.2.2. Example 12: Using JOINs with Columns-to-Rows Transformation

You may have noticed that there is an inconvenience in the solution{70} of problem 2.2.1.a{69}; if a book has several authors and/or genres, the information starts to be duplicated (let's order the result by the `b_name` field for clarity):

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin	Classic
Eugene Onegin	Alexander Pushkin	Poetry
The Art of Computer Programming	Donald Knuth	Classic
The Art of Computer Programming	Donald Knuth	Programming
Course of Theoretical Physics	Lev Landau	Classic
Course of Theoretical Physics	Evgeny Lifshitz	Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Dale Carnegie	Programming
Programming Psychology	Bjarne Stroustrup	Programming
Programming Psychology	Dale Carnegie	Psychology
Programming Psychology	Bjarne Stroustrup	Psychology
The Fisherman and the Golden Fish	Alexander Pushkin	Classic
The Fisherman and the Golden Fish	Alexander Pushkin	Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

Users, however, are much more accustomed to the following representation of data:

Book	Author(s)	Genre(s)
Eugene Onegin	Alexander Pushkin	Classic, Poetry
The Art of Computer Programming	Donald Knuth	Classic, Programming
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau	Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup, Dale Carnegie	Programming, Psychology
The Fisherman and the Golden Fish	Alexander Pushkin	Classic, Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

Problem 2.2.2.a{75}: show all books with their authors (duplication of books' titles is not allowed).

Problem 2.2.2.b{79}: show all books with their authors' and genres' (duplication of books' titles and authors' names is not allowed).

Expected result 2.2.2.a.

book	author(s)
Eugene Onegin	Alexander Pushkin
The Art of Computer Programming	Donald Knuth
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau
Foundation and Empire	Isaac Asimov
Programming Psychology	Bjarne Stroustrup, Dale Carnegie
The Fisherman and the Golden Fish	Alexander Pushkin
The C++ Programming Language	Bjarne Stroustrup

Expected result 2.2.2.b.

book	author(s)	genre(s)
Eugene Onegin	Alexander Pushkin	Classic, Poetry
The Art of Computer Programming	Donald Knuth	Classic, Programming
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau	Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup, Dale Carnegie	Programming, Psychology
The Fisherman and the Golden Fish	Alexander Pushkin	Classic, Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

Solution 2.2.2.a[{74}].

MySQL | Solution 2.2.2.a

```
1    SELECT `b_name`
2           AS `book`,
3           GROUP_CONCAT(`a_name` ORDER BY `a_name` SEPARATOR ', ')
4           AS `author(s)`
5    FROM   `books`
6           JOIN `m2m_books_authors` USING(`b_id`)
7           JOIN `authors` USING(`a_id`)
8    GROUP  BY `b_id`
9    ORDER  BY `b_name`
```

The solution for MySQL is very simple and elegant because this DBMS supports the **`GROUP_CONCAT`** function, which does all the basic work. This function has a very advanced syntax (even in our case we use ordering and delimiter specification), which you should definitely read in the official documentation.

Pay special attention to the line 8 of the query: never perform grouping by book title! Such a mistake leads to the fact that the DBMS considers several different books with the same title to be the same book (which can happen very often in real life). Grouping by a table primary key ensures that no different records are mixed into one group.

And one more peculiarity of MySQL deserves attention: in line 1, we extract `b_name` field, which is not mentioned in the `GROUP BY` clause in line 8 and is not an aggregation function. MS SQL Server and Oracle do not allow us to do this and strictly require that any field from `SELECT` clause that is not an aggregation function must be explicitly mentioned in the `GROUP BY` statement.

So, what does `GROUP_CONCAT` do? In fact, it "unrolls part of a column into a line" (while also arranging the authors alphabetically and separating them with ", "):

b_name	a_name		author(s)
Eugene Onegin	Alexander Pushkin	→	Alexander Pushkin
The Art of Computer Programming	Donald Knuth	→	Donald Knuth
Course of Theoretical Physics	Lev Landau	→	Evgeny Lifshitz, Lev Landau
	Evgeny Lifshitz		
Foundation and Empire	Isaac Asimov	→	Isaac Asimov
Programming Psychology	Dale Carnegie	→	Bjarne Stroustrup, Dale Carnegie
	Bjarne Stroustrup		
The Fisherman and the Golden Fish	Alexander Pushkin	→	Alexander Pushkin
The C++ Programming Language	Bjarne Stroustrup	→	Bjarne Stroustrup

MS SQL Server before version 2017 does not support any analogues of `GROUP_CONCAT` function, so the solution for it is very nontrivial:

MS SQL Solution 2.2.2.a (before version 2017)

```
1    WITH [prepared_data]
2         AS (SELECT [books].[b_id],
3                    [b_name],
4                    [a_name]
5               FROM [books]
6                    JOIN [m2m_books_authors]
7                      ON [books].[b_id] = [m2m_books_authors].[b_id]
8                    JOIN [authors]
9                      ON [m2m_books_authors].[a_id] = [authors].[a_id]
10            )
11   SELECT [outer].[b_name]
12          AS [book],
13          STUFF ((SELECT ', ' + [inner].[a_name]
14                  FROM   [prepared_data] AS [inner]
15                  WHERE  [outer].[b_id] = [inner].[b_id]
16                  ORDER  BY [inner].[a_name]
17                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
18                  1, 2, '')
19          AS [author(s)]
20   FROM   [prepared_data] AS [outer]
21   GROUP  BY [outer].[b_id],
22             [outer].[b_name]
```

Let's start with lines 1-10. They contain so called CTE (Common Table Expression). Very simplistically, Common Table Expression can be considered as a separate named query, the result of which can be accessed as a table. This is especially useful when several such queries are later used (without a Common Table Expression using a classical subquery, the body of such a subquery would have to be written wherever you need to refer to it).

In our case, the Common Table Expression results in the following data:

b_id	b_name	a_name
1	Eugene Onegin	Alexander Pushkin
2	The Fisherman and the Golden Fish	Alexander Pushkin
3	Foundation and Empire	Isaac Asimov
4	Programming Psychology	Dale Carnegie
4	Programming Psychology	Bjarne Stroustrup
5	The C++ Programming Language	Bjarne Stroustrup
6	Course of Theoretical Physics	Lev Landau
6	Course of Theoretical Physics	Evgeny Lifshitz
7	The Art of Computer Programming	Donald Knuth

This is almost a complete solution, the only thing left is to “transform into line” parts of the **a_name** column with authors of the same book. This task will be done by lines 13-19 of the query.

The main working part of the process is the code of the correlated subquery:

```
SELECT ', ' + [inner].[a_name]
FROM   [prepared_data] AS [inner]
WHERE  [outer].[b_id] = [inner].[b_id]
ORDER  BY [inner].[a_name]
```

For each row of the result obtained from the Common Table Expression, a subquery is executed that returns data from the same result related to the row in question on the external level. We have already considered a simpler version of the correlated subquery{52}, and now let’s show how the DBMS works in this particular case:

The row from the external part of the query (b_id)	Which rows will be processed by the internal part of the query (b_id)	What data will be collected
1	1	, Alexander Pushkin
2	2	, Alexander Pushkin
3	3	, Isaac Asimov
4	4 and 4	, Bjarne Stroustrup, Dale Carnegie
4	4 and 4	, Bjarne Stroustrup, Dale Carnegie
5	5	, Bjarne Stroustrup
6	6 and 6	, Evgeny Lifshitz, Lev Landau
6	6 and 6	, Evgeny Lifshitz, Lev Landau
7	7	, Donald Knuth

The “, ” (comma and space) at the beginning of each value in the “What data will be collected” column is not a typo. We are explicitly telling the DBMS to retrieve exactly the text as **", " + author_name**. To avoid these characters in the final result, we use the **STUFF** function.

To make it easier to explain, let's rewrite this part of the query in simplified form:

```
STUFF ((SELECT {data from the correlated subquery}
FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'), 1, 2, '')
```

The `FOR XML PATH(''), TYPE` part refers to `SELECT` and tells the DBMS to represent the result of the selection as part of the XML document for the `''` (empty string) path, i.e., just as a plain string, even though it looks like "some data in XML format" to the DBMS so far.

The intermediate result already looks simpler:

```
STUFF (({XML-string}).value('.', 'nvarchar(max)'), 1, 2, '')
```

The `value(path, data_type)` method is used in MS SQL Server to extract a string representation from XML. Without going into details, let's say that the first path parameter is equal to `.` (dot) says to take the current XML-document element (our "current element" is our string), and the `data_type` parameter is set to `nvarchar(max)` as one of the most convenient types in MS SQL Server for storing long strings

The result is almost ready:

```
STUFF ({string with ", " at the beginning}, 1, 2, '')
```

All that remains is to get rid of the `", "` characters at the beginning of each list of authors. This is what the `STUFF` function does by replacing the first through second characters (see the second and third function parameters `1` and `2`, respectively) with an empty string (see the fourth function parameter, equal to, `''`) in the `{string with ", " at the beginning}` string.

Starting from version 2017, MS SQL Server supports the `STRING_AGG` function (analog of the `GROUP_CONCAT` function in MySQL). The solution using this function looks much simpler, more compact and, in fact, similar to the solution for MySQL.

The only difference is the need to use the `ON` syntax instead of `USING` when executing the joining (lines 5-9 of the query) and the need to explicitly specify all fields from the `SELECT` clause, which are not the results of the aggregation functions (line 10 of the query), as parameters of the `GROUP BY` clause.

MS SQL | Solution 2.2.2.a (version 2017 and newer)

```
1    SELECT [b_name]
2           AS [book],
3           STRING_AGG([a_name], ', ') WITHIN GROUP (ORDER BY [a_name] ASC)
4           AS [author(s)]
5    FROM   [books]
6           JOIN [m2m_books_authors]
7            ON [books].[b_id] = [m2m_books_authors].[b_id]
8           JOIN [authors]
9            ON [m2m_books_authors].[a_id] = [authors].[a_id]
10   GROUP  BY [books].[b_id], [books].[b_name]
11   ORDER  BY [books].[b_name]
```

And that's it with MS SQL Server, let's move on to Oracle. Here we implement the third approach to the solution:

Oracle | Solution 2.2.2.a

```
1    SELECT "b_name" AS "book",
2           UTL_RAW.CAST_TO_NVARCHAR2
3           (
4              LISTAGG
5              (
6                  UTL_RAW.CAST_TO_RAW("a_name"),
7                  UTL_RAW.CAST_TO_RAW(N', ')
8              )
9              WITHIN GROUP (ORDER BY "a_name")
10          )
11          AS "author(s)"
12   FROM   "books"
13          JOIN "m2m_books_authors" USING ("b_id")
14          JOIN "authors" USING("a_id")
15   GROUP  BY "b_id",
16             "b_name"
```

In Oracle, the solution is simpler than in MS SQL Server, because here (starting from version 11gR2) the **`LISTAGG`** function is supported, which does almost the same thing as **`GROUP_CONCAT`** in MySQL. Thus, lines 4-8 of the query are responsible for "unrolling part of a column into a line".

The **`UTL_RAW.CAST_TO_RAW`** method calls in lines 6-7 and the **`UTL_RAW.CAST_TO_NVARCHAR2`** method in line 2 are needed to first represent text data in a format that is processed without interpreting byte values and then return the processed data from this format to text form. Without this conversion, the information about author becomes an unreadable set of special characters.

In other respects, Oracle's behavior in solving this problem is quite equivalent to that of MySQL.

Solution 2.2.2.b[{74}].

MySQL | Solution 2.2.2.b

```
1    SELECT `b_name`
2           AS `book`,
3           GROUP_CONCAT(DISTINCT `a_name` ORDER BY `a_name` SEPARATOR ', ')
4           AS `author(s)`,
5           GROUP_CONCAT(DISTINCT `g_name` ORDER BY `g_name` SEPARATOR ', ')
6           AS `genre(s)`
7    FROM   `books`
8           JOIN `m2m_books_authors` USING(`b_id`)
9           JOIN `authors` USING(`a_id`)
10          JOIN `m2m_books_genres` USING(`b_id`)
11          JOIN `genres` USING(`g_id`)
12   GROUP  BY `b_id`
13   ORDER  BY `b_name`
```

Solution 2.2.2.b for MySQL is only slightly more complicated than solution 2.2.2.a: here we've added another **`GROUP_CONCAT`** call (lines 5-6), and in both **`GROUP_CONCAT`** calls we've added the **`DISTINCT`** keyword to avoid duplicate information about authors and genres, which appears objectively in the process of joining:

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin	Classic
	Alexander Pushkin	Poetry
The Art of Computer Programming	Donald Knuth	Classic
	Donald Knuth	Programming
Course of Theoretical Physics	Evgeny Lifshitz	Classic
	Lev Landau	Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup	Programming
	Bjarne Stroustrup	Psychology
	Dale Carnegie	Programming
	Dale Carnegie	Psychology
The Fisherman and the Golden Fish	Alexander Pushkin	Classic
	Alexander Pushkin	Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

MS SQL | Solution 2.2.2.b (before version 2017)

```
1    WITH [prepared_data]
2         AS (SELECT [books].[b_id],
3                    [b_name],
4                    [a_name],
5                             [g_name]
6               FROM   [books]
7                    JOIN [m2m_books_authors]
8                      ON [books].[b_id] = [m2m_books_authors].[b_id]
9                    JOIN [authors]
10                     ON [m2m_books_authors].[a_id] = [authors].[a_id]
11                   JOIN [m2m_books_genres]
12                     ON [books].[b_id] = [m2m_books_genres].[b_id]
13                   JOIN [genres]
14                     ON [m2m_books_genres].[g_id] = [genres].[g_id]
15            )
16   SELECT [outer].[b_name]
17          AS [book],
18          STUFF ((SELECT DISTINCT ', ' + [inner].[a_name]
19                  FROM   [prepared_data] AS [inner]
20                  WHERE  [outer].[b_id] = [inner].[b_id]
21                  ORDER  BY ', ' + [inner].[a_name]
22                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
23                  1, 2, '')
24          AS [author(s)],
25          STUFF ((SELECT DISTINCT ', ' + [inner].[g_name]
26                  FROM   [prepared_data] AS [inner]
27                  WHERE  [outer].[b_id] = [inner].[b_id]
28                  ORDER  BY ', ' + [inner].[g_name]
29                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
30                  1, 2, '')
31          AS [genre(s)]
32   FROM   [prepared_data] AS [outer]
33   GROUP  BY [outer].[b_id],
34             [outer].[b_name]
```

This solution for MS SQL Server (before version 2017) is also based on solution 2.2.2.a{75} and differs by a little more **`JOIN`**s in the Common Table Expression (lines 11-14 were added), and (for the same reasons as in MySQL) by adding **`DISTINCT`** in lines 18 and 25. Blocks of lines 18-24 and 25-31 differ only in the name of the selected field (in the first case it is **`a_name`**, in the second it is **`g_name`**).

Starting with version 2017, the solution for MS SQL Server can be simplified using the **`STRING_AGG`** function.

MS SQL | Solution 2.2.2.b (version 2017 and newer)

```
1    WITH [books_names_and_authors] AS
2    (
3     SELECT [books].[b_id],
4            [books].[b_name]
5            AS [book],
6            STRING_AGG([a_name], ', ') WITHIN GROUP (ORDER BY [a_name] ASC)
7            AS [author(s)]
8     FROM   [books]
9            JOIN [m2m_books_authors]
10            ON [books].[b_id] = [m2m_books_authors].[b_id]
11           JOIN [authors]
12            ON [m2m_books_authors].[a_id] = [authors].[a_id]
13    GROUP  BY [books].[b_id], [books].[b_name]
14   ),
15   [books_genres] AS
16   (
17   SELECT [books].[b_id],
18          STRING_AGG([g_name], ', ') WITHIN GROUP (ORDER BY [g_name] ASC)
19          AS [genre(s)]
20   FROM   [books]
21          JOIN [m2m_books_genres]
22           ON [books].[b_id] = [m2m_books_genres].[b_id]
23          JOIN [genres]
24           ON [m2m_books_genres].[g_id] = [genres].[g_id]
25   GROUP  BY [books].[b_id]
26   )
27   SELECT [books_names_and_authors].[book],
28             [books_names_and_authors].[author(s)],
29             [books_genres].[genre(s)]
30     FROM [books_names_and_authors]
31          JOIN [books_genres]
32              ON [books_names_and_authors].[b_id] = [books_genres].[b_id]
```

Unfortunately, we cannot use here the same simple syntax as in lines 1-6 of the MySQL's solution, because MS SQL Server not only does not allow the **`DISTINCT`** keyword to be used as a "parameter" of the **`STRING_AGG`** function, but also forbids using two or more **`STRING_AGG`** calls with different field ordering in the same expression.

To get around this limitation, in lines 1-14 and 15-26 we prepare two intermediate data sets, i.e., a list of books with their names and authors (lines 1-14 of the query) and a list of books with their genres (lines 15-26 of the query), then we form the final data set by combining the two intermediate results (lines 27-32 of the query).

Here, the solution for MS SQL Server is finished, let's move on to Oracle. Of course, here too we can use an approach similar to the just discussed solution for MS SQL Server (for version 17 and newer), but for the sake of completeness we'll apply the approach without CTE (Common Table Expressions).

Oracle Solution 2.2.2.b

```
1    SELECT "book", "author(s)",
2           UTL_RAW.CAST_TO_NVARCHAR2
3           (
4              LISTAGG
5              (
6                  UTL_RAW.CAST_TO_RAW("g_name"),
7                  UTL_RAW.CAST_TO_RAW(N', ')
8              )
9              WITHIN GROUP (ORDER BY "g_name")
10          )
11          AS "genre(s)"
12   FROM
13   (
14          SELECT "b_id", "b_name" AS "book",
15                 UTL_RAW.CAST_TO_NVARCHAR2
16                 (
17                    LISTAGG
18                    (
19                       UTL_RAW.CAST_TO_RAW("a_name"),
20                       UTL_RAW.CAST_TO_RAW(N', ')
21                    )
22                    WITHIN GROUP (ORDER BY "a_name")
23                 )
24                 AS "author(s)"
25          FROM   "books"
26          JOIN "m2m_books_authors" USING ("b_id")
27          JOIN "authors" USING("a_id")
28          GROUP  BY "b_id",
29                 "b_name"
30   ) "first_level"
31   JOIN "m2m_books_genres" USING ("b_id")
32   JOIN "genres" USING("g_id")
33   GROUP  BY "b_id",
34             "book",
35             "author(s)"
```

Here the subquery in lines 13-30 is the solution 2.2.2.a[75], in which the `b_id` field is added to the `SELECT` clause to make it available for further `JOIN` and `GROUP BY` operations. If we rewrite the query with this information in mind, we get:

Oracle Solution 2.2.2.b

```
1    SELECT "book", "author(s)",
2           UTL_RAW.CAST_TO_NVARCHAR2
3           (
4              LISTAGG
5              (
6                  UTL_RAW.CAST_TO_RAW("g_name"),
7                  UTL_RAW.CAST_TO_RAW(N', ')
8              )
9              WITHIN GROUP (ORDER BY "g_name")
10          )
11          AS "genre(s)"
12   FROM {data_from_sulution_2_2_2_a + b_id field}
13   JOIN "m2m_books_genres" USING ("b_id")
14   JOIN "genres" USING("g_id")
15   GROUP  BY "b_id",
16             "book",
17             "author(s)"
```

We use this two-level approach here because there is no simple and productive way to eliminate duplication of data in the **LISTAGG** function. There is nowhere to apply **DISTINCT**, and there are no special built-in mechanisms for deduplication. Alternative solutions with regular expressions or preparation of filtered data are even more complicated and slower.

But nothing prevents us from splitting this task into two steps, putting the first part into a subquery (which can now be viewed as a temporary table), and making the second part completely identical to the first except for the name of the aggregated field (was **a_name**, became **g_name**).

Why doesn't the one-level solution work (when we try to get a set of authors and a set of genres at once in one **SELECT**)? So, we have the data:

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin	Classic
	Alexander Pushkin	Poetry
The Art of Computer Programming	Donald Knuth	Classic
	Donald Knuth	Programming
Course of Theoretical Physics	Evgeny Lifshitz	Classic
	Lev Landau	Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup	Programming
	Bjarne Stroustrup	Psychology
	Dale Carnegie	Programming
	Dale Carnegie	Psychology
The Fisherman and the Golden Fish	Alexander Pushkin	Classic
	Alexander Pushkin	Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

Trying to get a list of authors and genres immediately as one row looks like this:

b_name	a_name	g_name
Eugene Onegin	Alexander Pushkin, Alexander Pushkin	Classic, Poetry
The Art of Computer Programming	Donald Knuth, Donald Knuth	Classic, Programming
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau	Classic, Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup, Bjarne Stroustrup, Dale Carnegie, Dale Carnegie	Programming, Psychology, Programming, Psychology
The Fisherman and the Golden Fish	Alexander Pushkin, Alexander Pushkin	Classic, Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

Here's how the two-step solution works. First, we have this data set (rows which when grouped will turn into a single line and give us a list of more than one author are highlighted with a gray background):

b_name	a_name
Eugene Onegin	Alexander Pushkin
The Fisherman and the Golden Fish	Alexander Pushkin
Foundation and Empire	Isaac Asimov
Programming Psychology	Dale Carnegie
Programming Psychology	Bjarne Stroustrup
The C++ Programming Language	Bjarne Stroustrup
Course of Theoretical Physics	Lev Landau
Course of Theoretical Physics	Evgeny Lifshitz
The Art of Computer Programming	Donald Knuth

The result of the first grouping and line-by-line joining:

book	author(s)
Eugene Onegin	Alexander Pushkin
The Fisherman and the Golden Fish	Alexander Pushkin
Foundation and Empire	Isaac Asimov
Programming Psychology	Bjarne Stroustrup, Dale Carnegie
The C++ Programming Language	Bjarne Stroustrup
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau
The Art of Computer Programming	Donald Knuth

In the second step, the data set about each book and its authors already allows us to group by two fields at once, i.e., `book` and `author(s)`, because the lists of authors are already prepared, and no information about them will be lost:

book	author(s)	g_name
Eugene Onegin	Alexander Pushkin	Classic
Eugene Onegin	Alexander Pushkin	Poetry
The Fisherman and the Golden Fish	Alexander Pushkin	Classic
The Fisherman and the Golden Fish	Alexander Pushkin	Poetry
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup, Dale Carnegie	Programming
Programming Psychology	Bjarne Stroustrup, Dale Carnegie	Psychology
The C++ Programming Language	Bjarne Stroustrup	Programming
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau	Classic
The Art of Computer Programming	Donald Knuth	Classic
The Art of Computer Programming	Donald Knuth	Programming

And the final result is:

book	author(s)	genre(s)
Eugene Onegin	Alexander Pushkin	Classic, Poetry
The Art of Computer Programming	Donald Knuth	Classic, Programming
Course of Theoretical Physics	Evgeny Lifshitz, Lev Landau	Classic
Foundation and Empire	Isaac Asimov	Science Fiction
Programming Psychology	Bjarne Stroustrup, Dale Carnegie	Programming, Psychology
The Fisherman and the Golden Fish	Alexander Pushkin	Classic, Poetry
The C++ Programming Language	Bjarne Stroustrup	Programming

Task 2.2.2.TSK.A: show all books with their genres (duplication of books' titles is not allowed).

Task 2.2.2.TSK.B: show all the authors with all the books they have written and all the genres they have worked in (no duplication of authors' names, books' titles, or genres' names is allowed).

2.2.3. Example 13: JOINs and Subqueries with IN

Very often, `JOIN` queries can be converted into queries with a subquery and the `IN` keyword (the reverse conversion is also possible). Let's look at some typical examples:

Problem 2.2.3.a[87]: show the list of readers who have ever borrowed books from the library (use `JOIN`).

Problem 2.2.3.b[88]: show the list of readers who have ever borrowed books from the library (do not use `JOIN`).

Problem 2.2.3.c[90]: show the list of readers who have never borrowed books from the library (use `JOIN`).

Problem 2.2.3.d[91]: show the list of readers who have never borrowed books from the library (do not use `JOIN`).

It is easy to see that the problem pairs 2.2.3.a-2.2.3.b and 2.2.3.c-2.2.3.d are just a prerequisite for using the `JOIN` to `IN` conversion.

Expected result 2.2.3.a.

s_id	s_name
1	Ivanov I.I.
3	Sidorov S.S.
4	Sidorov S.S.

Expected result 2.2.3.b.

s_id	s_name
1	Ivanov I.I.
3	Sidorov S.S.
4	Sidorov S.S.

Expected result 2.2.3.c.

s_id	s_name
2	Petrov P.P.

Expected result 2.2.3.d.

s_id	s_name
2	Petrov P.P.

Solution 2.2.3.a[87].

MySQL Solution 2.2.3.a

```
SELECT DISTINCT `s_id`,
                `s_name`
FROM   `subscribers`
       JOIN `subscriptions`
         ON `s_id` = `sb_subscriber`
```

MS SQL Solution 2.2.3.a

```
SELECT DISTINCT [s_id],
                [s_name]
FROM   [subscribers]
       JOIN [subscriptions]
         ON [s_id] = [sb_subscriber]
```

Oracle Solution 2.2.3.a

```
SELECT DISTINCT "s_id",
                "s_name"
FROM   "subscribers"
       JOIN "subscriptions"
         ON "s_id" = "sb_subscriber"
```

For all three DBMSes this solution is completely equivalent. The `DISTINCT` keyword plays an important role here, because without it the result would be like this (due to the fact that `JOIN` will find all cases of giving books to each of the readers, and we need just the fact that a person took a book at least once):

s_id	s_name
1	Ivanov I.I.
1	Ivanov I.I.
1	Ivanov I.I.
1	Ivanov I.I.
1	Ivanov I.I.
3	Sidorov S.S.
3	Sidorov S.S.
3	Sidorov S.S.
4	Sidorov S.S.
4	Sidorov S.S.
4	Sidorov S.S.

But the `DISTINCT` also brings danger. Imagine that we decided not to retrieve the readers' identifiers, limiting ourselves to the readers' names (we will give an example only for MySQL, because in MS SQL Server and Oracle the situation is absolutely identical):

MySQL Solution 2.2.3.a (demonstrating a potential problem)

```
SELECT DISTINCT `s_name`
FROM   `subscribers`
       JOIN `subscriptions`
         ON `s_id` = `sb_subscriber`
```

The query resulted in the following data:

s_name
Ivanov I.I.
Sidorov S.S.

In the correct expected result, there were three records (including two Sidorovs with different identifiers). Now this information is lost.

Solution 2.2.3.b[86].

MySQL Solution 2.2.3.b

```
1    SELECT `s_id`,
2           `s_name`
3    FROM   `subscribers`
4    WHERE  `s_id` IN (SELECT DISTINCT `sb_subscriber`
5                      FROM   `subscriptions`)
```

MS SQL Solution 2.2.3.b

```
1    SELECT [s_id],
2           [s_name]
3    FROM   [subscribers]
4    WHERE  [s_id] IN (SELECT DISTINCT [sb_subscriber]
5                      FROM   [subscriptions])
```

Oracle Solution 2.2.3.b

```
1    SELECT "s_id",
2           "s_name"
3    FROM   "subscribers"
4    WHERE  "s_id" IN (SELECT DISTINCT "sb_subscriber"
5                      FROM   "subscriptions")
```

Again, the solution for all three DBMSes is exactly the same. The **DISTINCT** keyword in the subquery (in line 4 of all three queries) is used so that the DBMS has to analyze a smaller data set. Whether the difference in performance is significant, we will see immediately after solving the following two problems.

For now, let's graphically show how these two queries work. Let's start with the **JOIN** option. The DBMS searches for **s_id** values from the **subscribers** table and checks if there are corresponding records in the **subscriptions** table, whose **sb_subscriber** field contains the same value:

The **subscribers** table

s_id	s_name
1	Ivanov I.I.
2	Petrov P.P.
3	Sidorov S.S.
4	Sidorov S.S.

The **subscriptions** table

sb_subscriber
1
1
3
1
4
1
3
3
4
1
4

The search for a match results in the following:

s_id	sb_subscriber	s_name
1	1	Ivanov I.I.
1	1	Ivanov I.I.
3	3	Sidorov S.S.
1	1	Ivanov I.I.
4	4	Sidorov S.S.
1	1	Ivanov I.I.
3	3	Sidorov S.S.
3	3	Sidorov S.S.
4	4	Sidorov S.S.
1	1	Ivanov I.I.
4	4	Sidorov S.S.

We do not specify the `sb_subscriber` field in the `SELECT` clause, so that only two columns are left:

s_id	s_name
1	Ivanov I.I.
1	Ivanov I.I.
3	Sidorov S.S.
1	Ivanov I.I.
4	Sidorov S.S.
1	Ivanov I.I.
3	Sidorov S.S.
3	Sidorov S.S.
4	Sidorov S.S.
1	Ivanov I.I.
4	Sidorov S.S.

Finally, by using the `DISTINCT` keyword, we tell the DBMS to eliminate duplicates:

s_id	s_name
1	Ivanov I.I.
3	Sidorov S.S.
4	Sidorov S.S.

In the case of subquery and `IN` keyword the situation looks different. First, the DBMS selects all values of the `sb_subscriber` field:

sb_subscriber
1
1
3
1
4
1
3
3
4
1
4

Then there is the elimination of duplicates:

sb_subscriber
1
3
4

Now the DBMS analyzes each row of the `subscribers` table to find out whether its `s_id` value is in this set:

s_id	s_name	Set of sb_subscriber values	Whether to put the record in the final result
1	Ivanov I.I.	***1***, 3, 4	Yes
2	Petrov P.P.	~~1, 3, 4~~	No
3	Sidorov S.S.	1, ***3***, 4	Yes
4	Sidorov S.S.	1, 3, ***4***	Yes

Thus it turns out:

s_id	s_name
1	Ivanov I.I.
3	Sidorov S.S.
4	Sidorov S.S.

Note: here, describing the DBMS logic step by step, we assume for simplicity that the elimination of duplicates occurs at the very end. In fact, this is not the case. The internal data processing algorithms allow the DBMS to perform deduplication operations both after the result is formed and right in the process of its formation. The solution of using one or another variant will depend on the specific DBMS, the storage engine used, the query execution plan and other factors.

Solution 2.2.3.c[86].

MySQL Solution 2.2.3.c

```
SELECT `s_id`,
       `s_name`
FROM   `subscribers`
       LEFT JOIN `subscriptions`
              ON `s_id` = `sb_subscriber`
WHERE  `sb_subscriber` IS NULL
```

MS SQL Solution 2.2.3.c

```
SELECT [s_id],
       [s_name]
FROM   [subscribers]
       LEFT JOIN [subscriptions]
              ON [s_id] = [sb_subscriber]
WHERE  [sb_subscriber] IS NULL
```

Oracle Solution 2.2.3.c

```
SELECT "s_id",
       "s_name"
FROM   "subscribers"
       LEFT JOIN "subscriptions"
              ON "s_id" = "sb_subscriber"
WHERE  "sb_subscriber" IS NULL
```

When searching for readers who have never borrowed a book, the DBMS logic is as follows. First, the so called “left (outer) union” is performed, i.e., the DBMS retrieves all records from the **subscribers** table and tries to find a “pair” for them from the **subscriptions** table. If no “pair” is found (it does not exist), the **NULL** value will be substituted for the **sb_subscriber** field value:

s_id	s_name	sb_subscriber
1	Ivanov I.I.	1
1	Ivanov I.I.	1
1	Ivanov I.I.	1
1	Ivanov I.I.	1
1	Ivanov I.I.	1
2	**Petrov P.P.**	**NULL**
3	Sidorov S.S.	3
3	Sidorov S.S.	3
3	Sidorov S.S.	3
4	Sidorov S.S.	4
4	Sidorov S.S.	4
4	Sidorov S.S.	4

Thanks to the **WHERE "sb_subscriber" IS NULL** condition, only information about Petrov will get into the final result:

s_id	s_name
2	Petrov P.P.

Solution 2.2.3.d{86}.

MySQL Solution 2.2.3.d

```
SELECT `s_id`,
       `s_name`
FROM   `subscribers`
WHERE  `s_id` NOT IN (SELECT DISTINCT `sb_subscriber`
                      FROM   `subscriptions`)
```

MS SQL Solution 2.2.3.d

```
SELECT [s_id],
       [s_name]
FROM   [subscribers]
WHERE  [s_id] NOT IN (SELECT DISTINCT [sb_subscriber]
                      FROM   [subscriptions])
```

Oracle Solution 2.2.3.d

```
SELECT "s_id",
       "s_name"
FROM   "subscribers"
WHERE  "s_id" NOT IN (SELECT DISTINCT "sb_subscriber"
                      FROM   "subscriptions")
```

Here the DBMS behavior is similar to the solution 2.2.3.b except that when searching for `sb_subscriber` values it is necessary **not** to find `s_id` value among them:

s_id	s_name	Set of sb_subscriber values	Whether to put the record in the final result
1	Ivanov I.I.	***1***, 3, 4	No
2	Petrov P.P.	~~1, 3, 4~~	Yes
3	Sidorov S.S.	1, ***3***, 4	No
4	Sidorov S.S.	1, 3, ***4***	No

We get:

s_id	s_name
2	Petrov P.P.

Exploration 2.2.3.EXP.A: Let's analyze the effect of `DISTINCT` in a subquery on operation performance.

It's high time we talk about performance. We will be interested in two questions:

- Does the presence of `DISTINCT` in a subquery (for `IN` solutions) affect performance?
- Which works faster, `JOIN` or `IN` (in both cases: when we are looking for both readers who have taken books and readers who have not taken)?

We will conduct our exploration using the "Big Library (for Experiments)" database. Let's run the following queries a hundred times:

MySQL | Exploration 2.2.3.EXP.A

```
1    -- Query 1: JOIN
2    SELECT DISTINCT `s_id`,
3                    `s_name`
4    FROM   `subscribers`
5           JOIN `subscriptions`
6             ON `s_id` = `sb_subscriber`

1    -- Query 2: IN (... DISTINCT ...)
2    SELECT `s_id`,
3           `s_name`
4    FROM   `subscribers`
5    WHERE  `s_id` IN (SELECT DISTINCT `sb_subscriber`
6                      FROM   `subscriptions`)

1    -- Query 3: IN
2    SELECT `s_id`,
3           `s_name`
4    FROM   `subscribers`
5    WHERE  `s_id` IN (SELECT `sb_subscriber`
6                      FROM   `subscriptions`)

1    -- Query 4: LEFT JOIN
2    SELECT `s_id`,
3           `s_name`
4    FROM   `subscribers`
5           LEFT JOIN `subscriptions`
6                  ON `s_id` = `sb_subscriber`
7    WHERE  `sb_subscriber` IS NULL
```

MySQL Exploration 2.2.3.EXP.A (continued)

```
1     -- Query 5: NOT IN (... DISTINCT ...)
2     SELECT `s_id`,
3            `s_name`
4     FROM   `subscribers`
5     WHERE  `s_id` NOT IN (SELECT DISTINCT `sb_subscriber`
6                           FROM   `subscriptions`)

1     -- Query 6: NOT IN
2     SELECT `s_id`,
3            `s_name`
4     FROM   `subscribers`
5     WHERE  `s_id` NOT IN (SELECT `sb_subscriber`
6                           FROM   `subscriptions`)
```

MS SQL Exploration 2.2.3.EXP.A

```
1     -- Query 1: JOIN
2     SELECT DISTINCT [s_id],
3                     [s_name]
4     FROM   [subscribers]
5            JOIN [subscriptions]
6              ON [s_id] = [sb_subscriber]

1     -- Query 2: IN (... DISTINCT ...)
2     SELECT [s_id],
3            [s_name]
4     FROM   [subscribers]
5     WHERE  [s_id] IN (SELECT DISTINCT [sb_subscriber]
6                       FROM   [subscriptions])

1     -- Query 3: IN
2     SELECT [s_id],
3            [s_name]
4     FROM   [subscribers]
5     WHERE  [s_id] IN (SELECT [sb_subscriber]
6                       FROM   [subscriptions])

1     -- Query 4: LEFT JOIN
2     SELECT [s_id],
3            [s_name]
4     FROM   [subscribers]
5            LEFT JOIN [subscriptions]
6                   ON [s_id] = [sb_subscriber]
7     WHERE  [sb_subscriber] IS NULL

1     -- Query 5: NOT IN (... DISTINCT ...)
2     SELECT [s_id],
3            [s_name]
4     FROM   [subscribers]
5     WHERE  [s_id] NOT IN (SELECT DISTINCT [sb_subscriber]
6                           FROM   [subscriptions])

1     -- Query 6: NOT IN
2     SELECT [s_id],
3            [s_name]
4     FROM   [subscribers]
5     WHERE  [s_id] NOT IN (SELECT [sb_subscriber]
6                           FROM   [subscriptions])
```

Oracle | Exploration 2.2.3.EXP.A

```
1     -- Query 1: JOIN
2     SELECT DISTINCT "s_id",
3                     "s_name"
4     FROM   "subscribers"
5            JOIN "subscriptions"
6              ON "s_id" = "sb_subscriber"

1     -- Query 2: IN (... DISTINCT ...)
2     SELECT "s_id",
3            "s_name"
4     FROM   "subscribers"
5     WHERE  "s_id" IN (SELECT DISTINCT "sb_subscriber"
6                       FROM   "subscriptions")

1     -- Query 3: IN
2     SELECT "s_id",
3            "s_name"
4     FROM   "subscribers"
5     WHERE  "s_id" IN (SELECT "sb_subscriber"
6                       FROM   "subscriptions")

1     -- Query 4: LEFT JOIN
2     SELECT "s_id",
3            "s_name"
4     FROM   "subscribers"
5            LEFT JOIN "subscriptions"
6                   ON "s_id" = "sb_subscriber"
7     WHERE  "sb_subscriber" IS NULL

1     -- Query 5: NOT IN (... DISTINCT ...)
2     SELECT "s_id",
3            "s_name"
4     FROM   "subscribers"
5     WHERE  "s_id" NOT IN (SELECT DISTINCT "sb_subscriber"
6                           FROM   "subscriptions")

1     -- Query 6: NOT IN
2     SELECT "s_id",
3            "s_name"
4     FROM   "subscribers"
5     WHERE  "s_id" NOT IN (SELECT "sb_subscriber"
6                           FROM   "subscriptions")
```

Medians of time taken to complete each query:

	MySQL	MS SQL Server	Oracle
JOIN	91.065	8.574	1.136
IN (… DISTINCT …)	0.068	6.711	0.298
IN	0.039	6.723	0.309
LEFT JOIN	45.788	8.695	0.284
NOT IN (… DISTINCT …)	45.564	7.437	0.329
NOT IN	46.020	7.384	0.311

We posed two questions before conducting the exploration, and now we have the answers:

- Does the presence of `DISTINCT` in a subquery (for `IN` solutions) affect performance?
  - In the case of `IN`, the presence of `DISTINCT` noticeably slows down MySQL and slightly speeds up MS SQL Server and Oracle.
  - In the case of `NOT IN`, the presence of `DISTINCT` speeds up MySQL a bit and slows down MS SQL Server and Oracle a bit.
- Which works faster, `JOIN` or `IN` (in both cases: when we are looking for both readers who have taken books and readers who have not taken)?
  - `JOIN` works slower than `IN`.
  - `LEFT JOIN` works a little slower than `NOT IN` in MS SQL Server and a little faster than `NOT IN` in MySQL and Oracle.

Since most of the results are extremely close in their values, there is only one unambiguous conclusion: `IN` works faster than `JOIN`, in other cases it is worth conducting additional research.

Task 2.2.3.TSK.A: show the list of books that have ever been borrowed by readers.

Task 2.2.3.TSK.B: show the list of books that no reader has ever borrowed.

2.2.4. Example 14: Non-trivial Cases of JOINs and Subqueries with IN

There are problems that at first glance are very simple to solve. However, it turns out that the simple and obvious solution is wrong.

Problem 2.2.4.a{96}: show the list of readers who don't have books in their hands right now (use `JOIN`).

Problem 2.2.4.b{99}: show the list of readers who don't have books in their hands right now (do not use JOIN).

In problems 2.2.3.* of example 13{86} it was simple: if the `subscriptions` table had information about the reader, then they borrowed books from the library; if not, they did not. Now we will be interested both in readers who have never borrowed books (they objectively do not have books in their hands) and in readers who have borrowed books (but some have returned everything they borrowed, and some are still reading something).

Expected result 2.2.4.a.

s_id	s_name
1	Ivanov I.I.
2	Petrov P.P.

Expected result 2.2.4.b.

s_id	s_name
1	Ivanov I.I.
2	Petrov P.P.

Solution 2.2.4.a{96}.

MySQL Solution 2.2.4.a

```
SELECT `s_id`,
       `s_name`
FROM   `subscribers`
       LEFT OUTER JOIN `subscriptions`
                    ON `s_id` = `sb_subscriber`
GROUP  BY `s_id`
HAVING COUNT(IF(`sb_is_active` = 'Y', `sb_is_active`, NULL)) = 0
```

MS SQL Solution 2.2.4.a

```
SELECT [s_id],
       [s_name]
FROM   [subscribers]
       LEFT OUTER JOIN [subscriptions]
                    ON [s_id] = [sb_subscriber]
GROUP  BY [s_id],
          [s_name]
HAVING COUNT(CASE
                WHEN [sb_is_active] = 'Y' THEN [sb_is_active]
                ELSE NULL
             END) = 0
```

Oracle | Solution 2.2.4.a

```
1     SELECT "s_id",
2            "s_name"
3     FROM   "subscribers"
4            LEFT OUTER JOIN "subscriptions"
5                         ON "s_id" = "sb_subscriber"
6     GROUP  BY "s_id",
7               "s_name"
8     HAVING COUNT(CASE
9                    WHEN "sb_is_active" = 'Y' THEN "sb_is_active"
10                   ELSE NULL
11                END) = 0
```

Why is such a non-trivial solution? Lines 1-5 of query 2.2.4.a result in the following data (let's add the `sb_is_active` field for clarity):

s_id	s_name	sb_is_active
1	Ivanov I.I.	N
1	Ivanov I.I.	N
1	Ivanov I.I.	N
1	Ivanov I.I.	N
1	Ivanov I.I.	N
2	Petrov P.P.	NULL
3	Sidorov S.S.	Y
3	Sidorov S.S.	Y
3	Sidorov S.S.	Y
4	Sidorov S.S.	N
4	Sidorov S.S.	Y
4	Sidorov S.S.	Y

An indication that a reader returned all books is that they has no (`COUNT(…) = 0`) records with the value `sb_is_active = 'Y'`. The problem is that we can't "force" `COUNT` to count only `Y` values, it will consider any values not equal to `NULL`. Hence it is easy to conclude that we just have to turn into `NULL` any values of `sb_is_active` field which are not equal to `Y`, i.e., to get this result:

s_id	s_name	sb_is_active
1	Ivanov I.I.	NULL
1	Ivanov I.I.	NULL
1	Ivanov I.I.	NULL
1	Ivanov I.I.	NULL
1	Ivanov I.I.	NULL
2	Petrov P.P.	NULL
3	Sidorov S.S.	Y
3	Sidorov S.S.	Y
3	Sidorov S.S.	Y
4	Sidorov S.S.	NULL
4	Sidorov S.S.	Y
4	Sidorov S.S.	Y

That is what the `IF` statements in line 7 of the MySQL query and `CASE` statements in lines 8-11 of the MS SQL Server and Oracle queries are responsible for: they turn any value of the `sb_is_active` field, other than `Y`, into `NULL`.

Now it remains to check the result of `COUNT`: if it is zero, the reader has no books on hand.

A typical mistake when solving problem 2.2.4.a is trying to get the desired result with the following query:

MySQL Solution 2.2.4.a (**wrong** query)

```
SELECT `s_id`,
       `s_name`
FROM   `subscribers`
       LEFT OUTER JOIN `subscriptions`
                    ON `s_id` = `sb_subscriber`
WHERE  `sb_is_active` = 'Y'
        OR `sb_is_active` IS NULL
GROUP  BY `s_id`
HAVING COUNT(`sb_is_active`) = 0
```

MS SQL Solution 2.2.4.a (**wrong** query)

```
SELECT [s_id],
       [s_name]
FROM   [subscribers]
       LEFT OUTER JOIN [subscriptions]
                    ON [s_id] = [sb_subscriber]
WHERE  [sb_is_active] = 'Y'
        OR [sb_is_active] IS NULL
GROUP  BY [s_id], [s_name], [sb_is_active]
HAVING COUNT([sb_is_active]) = 0
```

Oracle Solution 2.2.4.a (**wrong** query)

```
SELECT "s_id",
       "s_name"
FROM   "subscribers"
       LEFT OUTER JOIN "subscriptions"
                    ON "s_id" = "sb_subscriber"
WHERE  "sb_is_active" = 'Y'
        OR "sb_is_active" IS NULL
GROUP  BY "s_id", "s_name", "sb_is_active"
HAVING COUNT("sb_is_active") = 0
```

Here we get the wrong result:

s_id	s_name
2	Petrov P.P.

This solution considers readers who have never borrowed books (`sb_is_active IS NULL`) and those who have at least one book in hand (`sb_is_active = 'Y'`), but those who have returned all books do not fit this condition:

s_id	s_name	sb_is_active	
1	Ivanov I.I.	N	These entries do not satisfy the **WHERE** condition and will be skipped.
1	Ivanov I.I.	N	
1	Ivanov I.I.	N	
1	Ivanov I.I.	N	
1	Ivanov I.I.	N	
2	Petrov P.P.	NULL	
3	Sidorov S.S.	Y	
3	Sidorov S.S.	Y	
3	Sidorov S.S.	Y	
4	Sidorov S.S.	N	
4	Sidorov S.S.	Y	
4	Sidorov S.S.	Y	

Thus, Ivanov I.I. turns out to be missing. Therefore, we should use the solution presented at the beginning of problem 2.2.4.a (i.e., the solution with transformation of the value of the `sb_is_active` field).

Solution 2.2.4.b[96].

MySQL Solution 2.2.4.b

```
SELECT `s_id`,
       `s_name`
FROM   `subscribers`
WHERE  `s_id` NOT IN (SELECT DISTINCT `sb_subscriber`
                      FROM   `subscriptions`
                      WHERE  `sb_is_active` = 'Y')
```

MS SQL Solution 2.2.4.b

```
SELECT [s_id],
       [s_name]
FROM   [subscribers]
WHERE  [s_id] NOT IN (SELECT DISTINCT [sb_subscriber]
                      FROM   [subscriptions]
                      WHERE  [sb_is_active] = 'Y')
```

Oracle Solution 2.2.4.b

```
SELECT "s_id",
       "s_name"
FROM   "subscribers"
WHERE  "s_id" NOT IN (SELECT DISTINCT "sb_subscriber"
                      FROM   "subscriptions"
                      WHERE  "sb_is_active" = 'Y')
```

A typical mistake when solving problem 2.2.4.b is trying to get the desired result with the following query:

MySQL Solution 2.2.4.b (**wrong** query)

```
SELECT `s_id`,
       `s_name`
FROM   `subscribers`
WHERE  `s_id` IN (SELECT DISTINCT `sb_subscriber`
                  FROM   `subscriptions`
                  WHERE  `sb_is_active` = 'N')
```

MS SQL Solution 2.2.4.b (**wrong** query)

```
SELECT [s_id],
       [s_name]
FROM   [subscribers]
WHERE  [s_id] IN (SELECT DISTINCT [sb_subscriber]
                  FROM   [subscriptions]
                  WHERE  [sb_is_active] = 'N')
```

Oracle Solution 2.2.4.b (**wrong** query)

```
SELECT "s_id",
       "s_name"
FROM   "subscribers"
WHERE  "s_id" IN (SELECT DISTINCT "sb_subscriber"
                  FROM   "subscriptions"
                  WHERE  "sb_is_active" = 'N')
```

We get the following result:

s_id	s_name
1	Ivanov I.I.
4	Sidorov S.S.

But this is a solution to the problem of “showing readers who returned a book at least once”. This problem (characteristic of many such situations) lies not so much in the area of correct querying as in the area of understanding the meaning (semantics) of the database model.

If for some fact of subscription, it is marked that “a book was returned” (the `sb_is_active` field has the `N` value), it does not mean that there is no other fact of subscriptions for same reader with a book that they has not yet returned. Consider this on the example of the reader with id 4 (“Sidorov S.S.”):

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
57	4	5	2012-06-11	2012-08-11	N
91	4	1	2015-10-07	2015-03-07	Y
99	4	4	2015-10-08	2025-11-08	Y

He has returned one book, so the subquery will return his identifier, but he did not return two more books: so, on the condition of the original problem he should not be in the final list.

It is also obvious that readers who have never borrowed a book will not be on the list of people who have no books in their hands (the identifiers of such readers never appear in the `subscriptions` table at all).

Task 2.2.4.TSK.A: show the list of books, not a single copy of which is currently in the hands of readers.

2.2.5. Example 15: Double Subqueries with IN

Problem 2.2.5.a[{102}]: show books from the “Programming” and/or “Classic” genres (do not use JOIN; genres’ identifiers are known).

Problem 2.2.5.b[{103}]: show books from the “Programming” and/or “Classic” genres (do not use JOIN; genres’ identifiers are unknown).

Problem 2.2.5.c[{104}]: show books from the “Programming” and/or “Classic” genres (use JOIN; genres’ identifiers are known).

Problem 2.2.5.d[{105}]: show books from the “Programming” and/or “Classic” genres (use JOIN; genre identifiers are unknown).

A variation of this problem, where instead of “and/or” there is a strict “and” is shown in example 18[{135}].

Expected result 2.2.5.a.

b_id	b_name
1	Eugene Onegin
7	The Art of Computer Programming
6	Course of Theoretical Physics
4	Programming Psychology
2	The Fisherman and the Golden Fish
5	The C++ Programming Language

Expected result 2.2.5.b.

b_id	b_name
1	Eugene Onegin
7	The Art of Computer Programming
6	Course of Theoretical Physics
4	Programming Psychology
2	The Fisherman and the Golden Fish
5	The C++ Programming Language

Expected result 2.2.5.c.

b_id	b_name
1	Eugene Onegin
7	The Art of Computer Programming
6	Course of Theoretical Physics
4	Programming Psychology
2	The Fisherman and the Golden Fish
5	The C++ Programming Language

Expected result 2.2.5.d.

b_id	b_name
1	Eugene Onegin
7	The Art of Computer Programming
6	Course of Theoretical Physics
4	Programming Psychology
2	The Fisherman and the Golden Fish
5	The C++ Programming Language

Solution 2.2.5.a{101}.

Using the **`m2m_books_genres`** table we can find the list of identifiers of books belonging to the desired genres, and then based on this list of identifiers we can select the titles of books from the **`books`** table.

MySQL Solution 2.2.5.a

```
1    SELECT `b_id`,
2           `b_name`
3    FROM   `books`
4    WHERE  `b_id` IN (SELECT DISTINCT `b_id`
5                      FROM   `m2m_books_genres`
6                      WHERE  `g_id` IN ( 2, 5 ))
7    ORDER  BY `b_name` ASC
```

MS SQL Solution 2.2.5.a

```
1    SELECT [b_id],
2           [b_name]
3    FROM   [books]
4    WHERE  [b_id] IN (SELECT DISTINCT [b_id]
5                      FROM   [m2m_books_genres]
6                      WHERE  [g_id] IN ( 2, 5 ))
7    ORDER  BY [b_name] ASC
```

Oracle Solution 2.2.5.a

```
1    SELECT "b_id",
2           "b_name"
3    FROM   "books"
4    WHERE  "b_id" IN (SELECT DISTINCT "b_id"
5                      FROM   "m2m_books_genres"
6                      WHERE  "g_id" IN ( 2, 5 ))
7    ORDER  BY "b_name" ASC
```

The subqueries in lines 4-6 result in the following values:

b_id
4
5
7
1
2
6

Solution 2.2.5.b[101].

The result is achieved in three steps, the first of which uses the genre name to determine its identifier using the **genres** table, after which the second and third steps are equivalent to the solution 2.2.5.a[102].

MySQL Solution 2.2.5.b

```
1     SELECT `b_id`,
2            `b_name`
3     FROM   `books`
4     WHERE  `b_id` IN (SELECT DISTINCT `b_id`
5                       FROM   `m2m_books_genres`
6                       WHERE  `g_id` IN (SELECT `g_id`
7                                         FROM   `genres`
8                                         WHERE
9                                               `g_name` IN ( 'Programming',
10                                                            'Classic' )
11                                        ))
12    ORDER  BY `b_name` ASC
```

MS SQL Solution 2.2.5.b

```
1     SELECT [b_id],
2            [b_name]
3     FROM   [books]
4     WHERE  [b_id] IN (SELECT DISTINCT [b_id]
5                       FROM   [m2m_books_genres]
6                       WHERE  [g_id] IN (SELECT [g_id]
7                                         FROM   [genres]
8                                         WHERE
9                                               [g_name] IN ( N'Programming',
10                                                            N'Classic' )
11                                        ))
12    ORDER  BY [b_name] ASC
```

Oracle Solution 2.2.5.b

```
1     SELECT "b_id",
2            "b_name"
3     FROM   "books"
4     WHERE  "b_id" IN (SELECT DISTINCT "b_id"
5                       FROM   "m2m_books_genres"
6                       WHERE  "g_id" IN (SELECT "g_id"
7                                         FROM   "genres"
8                                         WHERE
9                                               "g_name" IN ( N'Programming',
10                                                            N'Classic' )
11                                        ))
12    ORDER  BY "b_name" ASC
```

Subqueries 2.2.5.b on lines 6-11 result in the following data:

g_id
5
2

Subqueries 2.2.5.b in lines 4-10 result in the same data as subqueries in lines 3-5 of the solution 2.2.5.a.

Solution 2.2.5.c[101].

Here it is still convenient to use **IN** to specify the filtering condition, but the second use of **IN** by the problem statement should be replaced by **JOIN**.

MySQL Solution 2.2.5.c

```
1    SELECT DISTINCT `b_id`,
2                    `b_name`
3    FROM    `books`
4            JOIN `m2m_books_genres` USING ( `b_id` )
5    WHERE   `g_id` IN ( 2, 5 )
6    ORDER   BY `b_name` ASC
```

MS SQL Solution 2.2.5.c

```
1    SELECT DISTINCT [books].[b_id],
2                    [b_name]
3    FROM    [books]
4            JOIN [m2m_books_genres]
5              ON [books].[b_id] = [m2m_books_genres].[b_id]
6    WHERE   [g_id] IN ( 2, 5 )
7    ORDER   BY [b_name] ASC
```

Oracle Solution 2.2.5.c

```
1    SELECT DISTINCT "b_id",
2                    "b_name"
3    FROM    "books"
4            JOIN "m2m_books_genres" USING ( "b_id" )
5    WHERE   "g_id" IN ( 2, 5 )
6    ORDER   BY "b_name" ASC
```

We cannot avoid using the **DISTINCT** keyword, because otherwise we will get duplicate results for books belonging to both required genres at the same time:

b_id	b_name
1	Eugene Onegin
7	The Art of Computer Programming
7	The Art of Computer Programming
6	Course of Theoretical Physics
4	Programming Psychology
2	The Fisherman and the Golden Fish
5	The C++ Programming Language

But because we put a book identifier in the **SELECT**, we don't risk combining several books with the same title into a single line with **DISTINCT**.

Note the syntax differences of this solution for different DBMSes: MySQL and Oracle do not require specifying from which table to select the **b_id** field value (line 1 of queries 2.2.5.c for MySQL and Oracle), and they also support a reduced form of specifying the **JOIN** condition (line 4 of queries 2.2.5.c for MySQL and Oracle).

MS SQL Server requires to specify the table from which the value of the **b_id** field will be extracted (line 1 of query 2.2.5.c for MS SQL Server), and also does not support the abbreviated form of specifying the **JOIN** condition (line 5 of query 2.2.5.c for MS SQL Server).

Solution 2.2.5.d[101]:

Since the first use of `IN` by the problem statement had to be replaced by `JOIN`, there is one less `IN` level than in the solution 2.2.5.b[103].

MySQL Solution 2.2.5.d

```
SELECT DISTINCT `b_id`,
                `b_name`
FROM    `books`
        JOIN `m2m_books_genres` USING ( `b_id` )
WHERE   `g_id` IN (SELECT `g_id`
                   FROM   `genres`
                   WHERE  `g_name` IN ( N'Programming',
                                        N'Classic' )
                  )
ORDER  BY `b_name` ASC
```

MS SQL Solution 2.2.5.d

```
SELECT DISTINCT [books].[b_id],
                [b_name]
FROM    [books]
        JOIN [m2m_books_genres]
          ON [books].[b_id] = [m2m_books_genres].[b_id]
WHERE   [g_id] IN (SELECT [g_id]
                   FROM   [genres]
                   WHERE  [g_name] IN ( N'Programming',
                                        N'Classic' )
                  )
ORDER  BY [b_name] ASC
```

Oracle Solution 2.2.5.d

```
SELECT DISTINCT "b_id",
                "b_name"
FROM    "books"
        JOIN "m2m_books_genres" USING ( "b_id" )
WHERE   "g_id" IN (SELECT "g_id"
                   FROM   "genres"
                   WHERE  "g_name" IN ( N'Programming',
                                        N'Classic' )
                  )
ORDER  BY "b_name" ASC
```

The logic of the solution 2.2.5.d is a combination of 2.2.5.b (by deepest nested `IN`) and 2.2.5.c (by using `JOIN`).

Task 2.2.5.TSK.A: show books written by Pushkin and/or Asimov (individually or in co-authorship, it doesn't matter).

Task 2.2.5.TSK.B: show books **co-authored** by Carnegie and Stroustrup.

2.2.6. Example 16: JOINs with COUNT

Problem 2.2.6.a{106}: show books with more than one authors.

Problem 2.2.6.b{107}: show how many copies of each book are actually in the library right now.

Expected result 2.2.6.a.

b_id	b_name	authors_count
4	Programming Psychology	2
6	Course of Theoretical Physics	2

Expected result 2.2.6.b.

b_id	b_name	real_count
6	Course of Theoretical Physics	12
7	The Art of Computer Programming	7
3	Foundation and Empire	4
2	The Fisherman and the Golden Fish	3
5	The C++ Programming Language	2
1	Eugene Onegin	0
4	Programming Psychology	0

Solution 2.2.6.a{106}.

MySQL Solution 2.2.6.a

```
SELECT `b_id`,
       `b_name`,
       COUNT(`a_id`) AS `authors_count`
FROM   `books`
       JOIN `m2m_books_authors` USING (`b_id`)
GROUP  BY `b_id`
HAVING `authors_count` > 1
```

MS SQL Solution 2.2.6.a

```
SELECT [books].[b_id],
       [books].[b_name],
       COUNT([m2m_books_authors].[a_id]) AS [authors_count]
FROM   [books]
       JOIN [m2m_books_authors]
         ON [books].[b_id] = [m2m_books_authors].[b_id]
GROUP  BY [books].[b_id],
          [books].[b_name]
HAVING COUNT([m2m_books_authors].[a_id]) > 1
```

Oracle Solution 2.2.6.a

```
SELECT "b_id",
       "b_name",
       COUNT("a_id") AS "authors_count"
FROM   "books"
       JOIN "m2m_books_authors" USING ("b_id")
GROUP  BY "b_id", "b_name"
HAVING COUNT("a_id") > 1
```

Note how much shorter and easier the solution for MySQL is than for MS SQL Server and Oracle, due to the fact that in MySQL there is no need to specify the table name to resolve the ambiguity of the **b_id** field belongingness, and there is no need to group the result by the **b_name** field.

Solution 2.2.6.b{106}.

We will consider several solutions similar in essence, but fundamentally different in syntax, to show the breadth of SQL language and used DBMSes capabilities, as well as to compare the speed of different solutions.

In the MySQL solution, options 2 and 3 are presented only to show how subqueries can be used to emulate Common Table Expressions (support for which has only been available in MySQL since version 8).

MySQL Solution 2.2.6.b

```
1    -- Option 1: correlated subquery
2    SELECT DISTINCT `b_id`,
3                    `b_name`,
4                    ( `b_quantity` - (SELECT COUNT(`int`.`sb_book`)
5                                      FROM   `subscriptions` AS `int`
6                                      WHERE  `int`.`sb_book` = `ext`.`sb_book`
7                                             AND `int`.`sb_is_active` = 'Y')
8                                    ) AS `real_count`
9    FROM    `books`
10           LEFT OUTER JOIN `subscriptions` AS `ext`
11                       ON `books`.`b_id` = `ext`.`sb_book`
12   ORDER  BY `real_count` DESC

1    -- Option 2: correlated subquery and another subquery
2    -- (as emulation of CTE for MySQL prior to version 8)
3    SELECT `b_id`,
4           `b_name`,
5           ( `b_quantity` - IFNULL((SELECT `taken`
6                                    FROM   (SELECT `sb_book`         AS `b_id`,
7                                                   COUNT(`sb_book`) AS `taken`
8                                            FROM   `subscriptions`
9                                            WHERE  `sb_is_active` = 'Y'
10                                           GROUP  BY `sb_book`
11                                           ) AS `books_taken`
12                                   WHERE  `books`.`b_id` =
13                                          `books_taken`.`b_id`), 0
14                           ) ) AS
15          `real_count`
16   FROM    `books`
17   ORDER  BY `real_count` DESC
```

MySQL Solution 2.2.6.b (continued)

```
1     -- Option 3: step-by-step subqueries (MySQL prior to version 8)
2     SELECT `b_id`,
3            `b_name`,
4            ( `b_quantity` - (SELECT `taken`
5                              FROM   (SELECT `b_id`,
6                                             COUNT(`sb_book`) AS `taken`
7                                      FROM   `books`
8                                             LEFT OUTER JOIN
9                                             (SELECT `sb_book`
10                                             FROM   `subscriptions`
11                                             WHERE `sb_is_active` = 'Y'
12                                            ) AS `books_taken`
13                                            ON `b_id` = `sb_book`
14                                     GROUP  BY `b_id`) AS `real_taken`
15                             WHERE  `books`.`b_id` = `real_taken`.`b_id`)
16           ) AS `real_count`
17    FROM   `books`
18    ORDER  BY `real_count` DESC

1     -- Option 4: Common Table Expression emulation with
2     -- subquery (MySQL prior to version 8)
3     SELECT `b_id`,
4            `b_name`,
5            ( `b_quantity` - IFNULL(`taken`, 0) ) AS `real_count`
6     FROM   `books`
7            LEFT OUTER JOIN (SELECT `sb_book`,
8                                    COUNT(`sb_book`) AS `taken`
9                             FROM   `subscriptions`
10                            WHERE  `sb_is_active` = 'Y'
11                            GROUP  BY `sb_book`) AS `books_taken`
12                    ON `b_id` = `sb_book`
13    ORDER  BY `real_count` DESC
```

Let's analyze the logic of these queries.

Option 1 starts with a `JOIN` execution. If we temporarily replace the subquery on lines 4-8 with a constant, we get the following code:

MySQL Solution 2.2.6.b (modified query)

```
1     -- Option 1: correlated subquery
2     SELECT DISTINCT `b_id`,
3                     `b_name`,
4                     'X' AS `real_count`
5     FROM   `books`
6            LEFT OUTER JOIN `subscriptions` AS `ext`
7                         ON `books`.`b_id` = `ext`.`sb_book`
8     ORDER  BY `real_count` DESC
```

As a result of this query, we get:

b_id	b_name	real_count
1	Eugene Onegin	X
2	The Fisherman and the Golden Fish	X
3	Foundation and Empire	X
4	Programming Psychology	X
5	The C++ Programming Language	X
6	Course of Theoretical Physics	X
7	The Art of Computer Programming	X

Then the result of the correlated subquery (lines 4-7 in the original query) is subtracted from the value of the **`b_quantity`** field for each book):

MySQL Solution 2.2.6.b (code fragment with a subquery)

```
    -- Option 1: correlated subquery
    -- ...
4   SELECT COUNT(`int`.`sb_book`)
5   FROM   `subscriptions` AS `int`
6   WHERE  `int`.`sb_book` = `ext`.`sb_book`
7          AND `int`.`sb_is_active` = 'Y'
    -- ...
```

If in such a query, instead of the **`` `ext`.`sb_book` ``** expression we use the identifier value, then this piece of code can be executed as a separate query and will produce a specific number (e.g., for a book with identifier **`1`** it will be the number **`2`**, i.e., two copies of the book is currently in the hands of readers.

Finally, if we slightly modify the original query to see all data about each book (total number of its copies in the library, number of copies given to readers, number of copies remaining in the library), we get the following non-optimal, but illustrative query (subqueries in lines 5-8 and 10-13 have to be duplicated, because MySQL cannot immediately calculate the number of books given to readers and then use this value in the expression that calculates the remaining books in the library):

MySQL Solution 2.2.6.b (query reworked for clarity)

```
1   -- Option 1: correlated subquery
2   SELECT DISTINCT `b_id`,
3                   `b_name`,
4                   `b_quantity`,
5                   (SELECT COUNT(`int`.`sb_book`)
6                    FROM   `subscriptions` AS `int`
7                    WHERE  `int`.`sb_book` = `ext`.`sb_book`
8                           AND `int`.`sb_is_active` = 'Y')
9                    AS `taken`,
10                  ( `b_quantity` - (SELECT COUNT(`int`.`sb_book`)
11                                    FROM   `subscriptions` AS `int`
12                                    WHERE  `int`.`sb_book` = `ext`.`sb_book`
13                                           AND `int`.`sb_is_active` = 'Y') )
14                   AS `real_count`
15  FROM   `books`
16         LEFT OUTER JOIN `subscriptions` AS `ext`
17                      ON `books`.`b_id` = `ext`.`sb_book`
18  ORDER  BY `real_count` DESC
```

The result of executing such a query is:

b_id	b_name	b_quantity	taken	real_count
6	Course of Theoretical Physics	12	0	12
7	The Art of Computer Programming	7	0	7
3	Foundation and Empire	5	1	4
2	The Fisherman and the Golden Fish	3	0	3
5	The C++ Programming Language	3	1	2
1	Eugene Onegin	2	2	0
4	Programming Psychology	1	1	0

Option 2 is built on the idea of emulating Common Table Expressions, which are only supported by MySQL since version 8.

As further research on query execution speed shows, this will turn out to be a very bad idea, but as an experiment, let's consider the logic of such a query.

The work starts with the deepest nested subquery (lines 6-10 in the original query):

MySQL Solution 2.2.6.b (code fragment with a subquery)

```
    -- Option 2: correlated subquery and another subquery
    -- (as emulation of CTE for MySQL prior to version 8)
    -- ...
6   SELECT `sb_book`         AS `b_id`,
7          COUNT(`sb_book`) AS `taken`
8   FROM   `subscriptions`
9   WHERE  `sb_is_active` = 'Y'
10  GROUP  BY `sb_book`
    -- ...
```

The result of this subquery is as follows (there is no data here for books, none of which are currently in the hands of readers):

b_id	taken
1	2
3	1
4	1
5	1

The correlated subquery (lines 5-15) handles this data, where the `IFNULL` function returns `0` instead of `NULL` for books that have not been given out to readers. For books of which at least one copy has been given out to readers, a ready (other than zero) value is returned. The value obtained from the subquery is subtracted from the `b_quantity` field, and so we get the final result.

Option 3 is based on the idea of sequential execution of several subqueries. The first subquery is the most deeply nested one, which returns the identifiers of books in the hands of readers (lines 10-13 of the original query):

MySQL Solution 2.2.6.b (code fragment with a subquery)

```
    -- Option 3: step-by-step subqueries (MySQL prior to version 8)
    -- ...
9   (SELECT `sb_book`
10   FROM   `subscriptions`
11   WHERE `sb_is_active` = 'Y'
12  ) AS `books_taken`
    -- ...
```

The result of this subquery is as follows:

sb_book
3
5
1
1
4

Next, a subquery is performed to determine the number of copies of each book in the hands of readers (lines 6-15 of the original query):

MySQL Solution 2.2.6.b (code fragment with a subquery)

```
    -- Option 3: step-by-step subqueries (MySQL prior to version 8)
    -- ...
5   SELECT          `b_id`,
6                   count(`sb_book`) AS `taken`
7   FROM            `books`
8   LEFT OUTER JOIN
9                   (
10                          SELECT `sb_book`
11                          FROM   `subscriptions`
12                          WHERE  `sb_is_active` = 'Y' ) AS `books_taken`
13  ON              `b_id` = `sb_book`
14  GROUP BY        `b_id`) AS `real_taken`
    -- ...
```

The result of this subquery is as follows:

b_id	taken
1	2
2	0
3	1
4	1
5	1
6	0
7	0

The resulting data are used in the correlated subquery (lines 5-17 of the original query) to determine the final result (the number of copies of books in the library):

MySQL Solution 2.2.6.b (code fragment with a subquery)

```
    -- Option 3: step-by-step subqueries (MySQL prior to version 8)
    -- ...
4           ( `b_quantity` - (SELECT `taken`
5                             FROM   (SELECT `b_id`,
6                                            COUNT(`sb_book`) AS `taken`
7                                     FROM   `books`
8                                            LEFT OUTER JOIN
9                                            (SELECT `sb_book`
10                                            FROM   `subscriptions`
11                                            WHERE `sb_is_active` = 'Y'
12                                           ) AS `books_taken`
13                                           ON `b_id` = `sb_book`
14                                    GROUP  BY `b_id`) AS `real_taken`
15                            WHERE  `books`.`b_id` = `real_taken`.`b_id`)
16          ) AS `real_count`
    -- ...
```

Executing this subquery produces the final desired value, which appears in the result of the main query.

Option 4 is similar to option 2 in that it also implements emulation of Common Table Expressions through subqueries, but there is a significant difference from option 2, as there are no correlated subqueries.

The subquery emulating a Common Table Expression (lines 7-11 of the original query) prepares all the necessary data about how many copies of each book are in the hands of readers:

MySQL Solution 2.2.6.b (code fragment with a subquery)

```
    -- Option 4: Common Table Expression emulation with
    -- subquery (MySQL prior to version 8)
    -- ...
7   SELECT `sb_book`,
8          COUNT(`sb_book`) AS `taken`
9   FROM   `subscriptions`
10  WHERE  `sb_is_active` = 'Y'
11  GROUP  BY `sb_book`
    -- ...
```

The result of this subquery is as follows:

sb_book	taken
1	2
3	1
4	1
5	1

This result is used in the **JOIN** operator (lines 7-12 of the original query), and the **IFNULL** function in line 5 of the original query allows us to get a numeric representation of the number of books given to readers in the case when no copies were given. To demonstrate this, let's rewrite option 4 by adding to the **SELECT** the **b_quantity** field and the original **taken** value obtained by combining it with the data from the subquery:

MySQL Solution 2.2.6.b (modified query)

```
1   -- Option 4: Common Table Expression emulation with
2   -- subquery (MySQL prior to version 8)
3   SELECT `b_id`,
4          `b_name`,
5          `b_quantity`,
6          `taken`,
7          ( `b_quantity` - IFNULL(`taken`, 0) ) AS `real_count`
8   FROM   `books`
9          LEFT OUTER JOIN (SELECT `sb_book`,
10                                 COUNT(`sb_book`) AS `taken`
11                          FROM   `subscriptions`
12                          WHERE  `sb_is_active` = 'Y'
13                          GROUP  BY `sb_book`) AS `books_taken`
14                      ON `b_id` = `sb_book`
15  ORDER  BY `real_count` DESC
```

The result of executing this modified query is:

b_id	b_name	b_quantity	taken	real_count
6	Course of Theoretical Physics	12	NULL	12
7	The Art of Computer Programming	7	NULL	7
3	Foundation and Empire	5	1	4
2	The Fisherman and the Golden Fish	3	NULL	3
5	The C++ Programming Language	3	1	2
1	Eugene Onegin	2	2	0
4	Programming Psychology	1	1	0

In the original query, the **NULL** values of the **taken** field are converted to **0**, then the **taken** field value is subtracted from the **b_quantity** field value, and thus the final values of the **real_count** field are obtained.

Starting from version 8, MySQL also supports Common Table Expressions, that is why options 2-4 can be rewritten using them. The obtained result will be completely similar to the solution for MS SQL Server and Oracle (except for the syntax of processing **NULL** values), therefore, here we will only give the query code itself and show its explanation further on the examples of MS SQL Server and Oracle.

MySQL Solution 2.2.6.b (MySQL 8 and newer)

```
-- Option 2: correlated subquery and Common Table Expression
-- (MySQL 8 and newer)
WITH `books_taken`
     AS (SELECT `sb_book`         AS `b_id`,
                COUNT(`sb_book`) AS `taken`
         FROM   `subscriptions`
         WHERE  `sb_is_active` = 'Y'
         GROUP  BY `sb_book`)
SELECT `b_id`,
       `b_name`,
       ( `b_quantity` - IFNULL((SELECT `taken`
                                FROM   `books_taken`
                                WHERE  `books`.`b_id` =
                                       `books_taken`.`b_id`), 0
                         ) ) AS
       `real_count`
FROM   `books`
ORDER  BY `real_count` DESC
```

```
-- Option 3: step-by-step subqueries and Common Table Expression
-- (MySQL 8 and newer)
WITH `books_taken`
     AS (SELECT `sb_book`
         FROM   `subscriptions`
         WHERE  `sb_is_active` = 'Y'),
     `real_taken`
     AS (SELECT `b_id`,
                COUNT(`sb_book`) AS `taken`
         FROM   `books`
                LEFT OUTER JOIN `books_taken`
                             ON `b_id` = `sb_book`
         GROUP  BY `b_id`)
SELECT `b_id`,
       `b_name`,
       ( `b_quantity` - (SELECT `taken`
                         FROM   `real_taken`
                         WHERE  `books`.`b_id` = `real_taken`.`b_id`) ) AS
       `real_count`
FROM   `books`
ORDER  BY `real_count` DESC
```

```
-- Option 4: without subqueries (MySQL 8 and newer)
WITH `books_taken`
     AS (SELECT `sb_book`,
                COUNT(`sb_book`) AS `taken`
         FROM   `subscriptions`
         WHERE  `sb_is_active` = 'Y'
         GROUP  BY `sb_book`)
SELECT `b_id`,
       `b_name`,
       ( `b_quantity` - IFNULL(`taken`, 0) ) AS `real_count`
FROM   `books`
       LEFT OUTER JOIN `books_taken`
                    ON `b_id` = `sb_book`
ORDER  BY `real_count` DESC
```

Let's consider the solution of problem 2.2.6.b for MS SQL Server and Oracle. Option 1 of these solutions is completely identical to option 1 of the MySQL solution, and options 2-4 are completely identical for MS SQL Server and Oracle, so we will consider them only once on the example of MS SQL Server.

MS SQL Solution 2.2.6.b

```
1    -- Option 1: correlated subquery
2    SELECT DISTINCT [b_id],
3                    [b_name],
4                    ( [b_quantity] - (SELECT COUNT([int].[sb_book])
5                                      FROM   [subscriptions] AS [int]
6                                      WHERE  [int].[sb_book] = [ext].[sb_book]
7                                             AND [int].[sb_is_active] = 'Y') )
8    AS
9                    [real_count]
10   FROM   [books]
11          LEFT OUTER JOIN [subscriptions] AS [ext]
12                       ON [books].[b_id] = [ext].[sb_book]
13   ORDER  BY [real_count] DESC
```

For a description of **option 1** logic, see above as this query is identical in all three DBMSes.

MS SQL Solution 2.2.6.b

```
1    -- Option 2: Common Table Expression
2    -- and correlated subquery
3    WITH [books_taken]
4         AS (SELECT [sb_book]         AS [b_id],
5                    COUNT([sb_book]) AS [taken]
6             FROM   [subscriptions]
7             WHERE  [sb_is_active] = 'Y'
8             GROUP  BY [sb_book])
9    SELECT [b_id],
10          [b_name],
11          ( [b_quantity] - ISNULL((SELECT [taken]
12                                   FROM   [books_taken]
13                                   WHERE  [books].[b_id] =
14                                          [books_taken].[b_id]), 0
15                           ) ) AS
16          [real_count]
17   FROM   [books]
18   ORDER  BY [real_count] DESC

1    -- Option 3: step-by-step Common Table Expression and subquery
2    WITH [books_taken]
3         AS (SELECT [sb_book]
4             FROM   [subscriptions]
5             WHERE  [sb_is_active] = 'Y'),
6         [real_taken]
7         AS (SELECT [b_id],
8                    COUNT([sb_book]) AS [taken]
9             FROM   [books]
10                   LEFT OUTER JOIN [books_taken]
11                                ON [b_id] = [sb_book]
12            GROUP  BY [b_id])
13   SELECT [b_id],
14          [b_name],
15          ( [b_quantity] - (SELECT [taken]
16                            FROM   [real_taken]
17                            WHERE  [books].[b_id] = [real_taken].[b_id]) ) AS
18          [real_count]
19   FROM   [books]
20   ORDER  BY [real_count] DESC
```

MS SQL Solution 2.2.6.b (continues)

```
1    -- Option 4: without subqueries
2    WITH [books_taken]
3         AS (SELECT [sb_book],
4                    COUNT([sb_book]) AS [taken]
5               FROM   [subscriptions]
6               WHERE  [sb_is_active] = 'Y'
7               GROUP  BY [sb_book])
8    SELECT [b_id],
9           [b_name],
10          ( [b_quantity] - ISNULL([taken], 0) ) AS [real_count]
11   FROM   [books]
12          LEFT OUTER JOIN [books_taken]
13                       ON [b_id] = [sb_book]
14   ORDER  BY [real_count] DESC
```

Option 2 is based on the fact that the Common Table Expression in rows 3-8 prepares information about the number of copies of books in the hands of readers. Let's execute separately the corresponding fragment of the query:

MS SQL Solution 2.2.6.b (modified query fragment)

```
1    -- Option 2: Common Table Expression
2    -- and correlated subquery
3    WITH [books_taken]
4         AS (SELECT [sb_book]         AS [b_id],
5                    COUNT([sb_book]) AS [taken]
6               FROM   [subscriptions]
7               WHERE  [sb_is_active] = 'Y'
8               GROUP  BY [sb_book])
9    SELECT * FROM [books_taken]
```

The result of executing this query fragment is as follows:

b_id	taken
1	2
3	1
4	1
5	1

Then in lines 11-16 of the original query a correlated subquery is executed which results in the number of copies given out to readers for each book or `NULL` if no copies are given out. To be able to correctly use such a result in the arithmetic expression, in line 11 of the original query we use the `ISNULL` function to convert `NULL` values to `0`.

Option 3, based on the step-by-step application of two Common Table Expressions, prepares a complete data set for the correlated subquery.

The first Common Table Expression (lines 2-5) returns the following data:

sb_book
3
5
1
1
4

The second Common Table Expression (lines 6-12) returns the following data:

b_id	taken
1	2
2	0
3	1
4	1
5	1
6	0
7	0

Based on the data obtained, the correlated subquery on lines 15-18 calculates the actual number of copies of books in the library. Since the data come from the second Common Table Expression with “ready zeros” for books with no copies given out to readers, there is no need to use the `ISNULL` function here.

Option 4 is based on the preliminary preparation in the Common Table Expression of information about how many books are given out to readers, and then subtracting this number from the number of books registered in the library. The Common Table Expression returns the following data:

sb_book	taken
1	2
3	1
4	1
5	1

Since the grouping for books, no copies of which are given to readers, the `taken` value will be `NULL`, we apply the `ISNULL` function, which converts `NULL`s to `0` in the 10th line of the query.

Let’s consider the solution of problem 2.2.6.b for Oracle. The only noticeable difference between this solution and the solution for MS SQL Server is that Oracle uses the `NVL` function to obtain behavior similar to the `ISNULL` function in MS SQL Server (substituting the value `0` instead of `NULL`).

Oracle | Solution 2.2.6.b

```
1    -- Option 1: correlated subquery
2    SELECT DISTINCT "b_id",
3                    "b_name",
4                    ( "b_quantity" - (SELECT COUNT("int"."sb_book")
5                                      FROM   "subscriptions" "int"
6                                      WHERE  "int"."sb_book" = "ext"."sb_book"
7                                             AND "int"."sb_is_active" = 'Y') )
8    AS
9                    "real_count"
10   FROM    "books"
11           LEFT OUTER JOIN "subscriptions" "ext"
12                        ON "books"."b_id" = "ext"."sb_book"
13   ORDER  BY "real_count" DESC
```

Oracle | Solution 2.2.6.b (continued)

```
-- Option 2: Common Table Expression
-- and correlated subquery
WITH "books_taken"
     AS (SELECT "sb_book"         AS "b_id",
                COUNT("sb_book") AS "taken"
         FROM   "subscriptions"
         WHERE  "sb_is_active" = 'Y'
         GROUP  BY "sb_book")
SELECT "b_id",
       "b_name",
       ( "b_quantity" - NVL((SELECT "taken"
                             FROM   "books_taken"
                             WHERE  "books"."b_id" =
                                    "books_taken"."b_id"), 0
                           ) ) AS
       "real_count"
FROM   "books"
ORDER  BY "real_count" DESC
```

```
-- Option 3: step-by-step Common Table Expression and subquery
WITH "books_taken"
     AS (SELECT "sb_book"
         FROM   "subscriptions"
         WHERE  "sb_is_active" = 'Y'),
     "real_taken"
     AS (SELECT "b_id",
                COUNT("sb_book") AS "taken"
         FROM   "books"
                LEFT OUTER JOIN "books_taken"
                             ON "b_id" = "sb_book"
         GROUP  BY "b_id")
SELECT "b_id",
       "b_name",
       ( "b_quantity" - (SELECT "taken"
                         FROM   "real_taken"
                         WHERE  "books"."b_id" = "real_taken"."b_id") ) AS
       "real_count"
FROM   "books"
ORDER  BY "real_count" DESC
```

```
-- Option 4: without subqueries
WITH "books_taken"
     AS (SELECT "sb_book",
                COUNT("sb_book") AS "taken"
         FROM   "subscriptions"
         WHERE  "sb_is_active" = 'Y'
         GROUP  BY "sb_book")
SELECT "b_id",
       "b_name",
       ( "b_quantity" - NVL("taken", 0) ) AS "real_count"
FROM   "books"
       LEFT OUTER JOIN "books_taken"
                    ON "b_id" = "sb_book"
ORDER  BY "real_count" DESC
```

Exploration 2.2.6.EXP.A: let's compare the speed of each of the four options of the 2.2.6.b queries for all three DBMSes on the "Big Library (for Experiments)" database.

By running each query a hundred times for each DBMS, we get the following median time values:

	MySQL (prior to version 8)	MySQL (version 8 and newer)	MS SQL Server	Oracle
Option 1	0.545	0.317	50.575	1.393
Option 2	16240.234	9970.088	5.814	0.430
Option 3	220.918	9891.257	5.733	0.342
Option 4	19061.824	9756.696	5.309	0.383

Note how differently all DBMSes behave. For MS SQL Server and Oracle, the correlated subquery option (option 1) was expectedly the slowest, but in MySQL it was much faster than the "emulation of a Common Table Expression", and even when using the "real Common Table Expression" in MySQL version 8, the correlated subquery option was much faster.

Task 2.2.6.TSK.A: show authors who have written more than one book.

Task 2.2.6.TSK.B: show books belonging to more than one genre.

Task 2.2.6.TSK.C: show readers who now have more than one book in their hands.

Task 2.2.6.TSK.D: show how many copies of each book have now been given out to readers.

Task 2.2.6.TSK.E: show all authors and the number of copies of books by each author.

Task 2.2.6.TSK.F: show all authors and the number of books (not copies of books, but "books as titles") for each author.

Task 2.2.6.TSK.G: show all readers who have not returned at least one book and the number of unreturned books for each such reader.

2.2.7. Example 17: JOINs with COUNT and Aggregation Functions

Problem 2.2.7.a[120]: show the readability of the authors, i.e., all authors and the number of times books by these authors have been borrowed by readers.

Problem 2.2.7.b[121]: show the most-read author, i.e., the author (or authors, if there are several) whose books readers borrowed the most.

Problem 2.2.7.c[125]: show the average readability of the authors, i.e., the average of how many times readers borrowed each author's book.

Problem 2.2.7.d[126]: show the median readability of the authors, i.e., the median value from how many times readers borrowed each author's book.

Problem 2.2.7.e[131]: write a query to check if there is an error in the database, in which it appears that more copies of a certain book are now on hand than there were registered in the library; return 1 if there is an error and 0 if there is no error.

Expected result 2.2.7.a.

a_id	a_name	books
7	Alexander Pushkin	4
6	Bjarne Stroustrup	4
3	Dale Carnegie	2
2	Isaac Asimov	2
1	Donald Knuth	1
5	Evgeny Lifshitz	0
4	Lev Landau	0

Expected result 2.2.7.b.

a_id	a_name	books
6	Bjarne Stroustrup	4
7	Alexander Pushkin	4

Expected result 2.2.7.c: the default number of decimal places is different in MySQL, MS SQL Server, and Oracle.

MySQL

avg_reading
1.8571

MS SQL Server

avg_reading
1.85714285714286

Oracle

avg_reading
1.857142857142857142857142857142857142857

Expected result 2.2.7.d: the default number of decimal places is different in MySQL, MS SQL Server, and Oracle.

MySQL

med_reading
2.0000

MS SQL Server

med_reading
2

Oracle

med_reading
2

Expected result 2.2.7.e.

error_exists
0

Solution 2.2.7.a[{119}].

MySQL Solution 2.2.7.a

```
1     SELECT `a_id`,
2            `a_name`,
3            COUNT(`sb_book`) AS `books`
4     FROM   `authors`
5            JOIN `m2m_books_authors` USING ( `a_id` )
6            LEFT OUTER JOIN `subscriptions`
7                         ON `m2m_books_authors`.`b_id` = `sb_book`
8     GROUP  BY `a_id`
9     ORDER  BY `books` DESC
```

MS SQL Solution 2.2.7.a

```
1     SELECT [authors].[a_id],
2            [authors].[a_name],
3            COUNT([sb_book]) AS [books]
4     FROM   [authors]
5            JOIN [m2m_books_authors]
6              ON [authors].[a_id] = [m2m_books_authors].[a_id]
7            LEFT OUTER JOIN [subscriptions]
8                         ON [m2m_books_authors].[b_id] = [sb_book]
9     GROUP  BY [authors].[a_id],
10              [authors].[a_name]
11    ORDER  BY COUNT([sb_book]) DESC
```

Oracle Solution 2.2.7.a

```
1     SELECT "a_id",
2            "a_name",
3            COUNT("sb_book") AS "books"
4     FROM   "authors"
5            JOIN "m2m_books_authors" USING ( "a_id" )
6            LEFT OUTER JOIN "subscriptions"
7                         ON "m2m_books_authors"."b_id" = "sb_book"
8     GROUP  BY "a_id",
9               "a_name"
10    ORDER  BY "books" DESC
```

Solution for all three DBMSes differs only in the nuances of syntax, but in essence it is trivial: we need to gather information about the authors, books, and subscriptions (this is achieved through two `JOIN` operators), then calculate the number of subscriptions, grouping the results of the count by authors' ids.

Solution 2.2.7.b[119].

MySQL Solution 2.2.7.b (before version 8)

```
1    -- Option 1: using MAX function (before MySQL 8)
2    SELECT `a_id`,
3           `a_name`,
4           COUNT(`sb_book`) AS `books`
5    FROM   `authors`
6           JOIN `m2m_books_authors` USING (`a_id`)
7           LEFT OUTER JOIN `subscriptions`
8                        ON `m2m_books_authors`.`b_id` = `sb_book`
9    GROUP  BY `a_id`
10   HAVING `books` = (SELECT MAX(`books`)
11                     FROM
12                     (SELECT COUNT(`sb_book`) AS `books`
13                      FROM   `authors`
14                             JOIN `m2m_books_authors` USING ( `a_id` )
15                             LEFT OUTER JOIN `subscriptions`
16                             ON `m2m_books_authors`.`b_id` = `sb_book`
17                             GROUP  BY `a_id`
18                     ) AS `books_per_author`)
```

Since MySQL before version 8 does not support either ranking (window) functions or MS SQL Server specific syntax `TOP ... WITH TIES`, the only option is to calculate the number of times readers took books by each author (lines 2-9 of the query), which is completely similar to the solution of problem 2.2.7.a[120], and then leave in the final result only those authors for whom this number coincides with the maximum value for all authors (this maximum is calculated in lines 10-18 of the query).

However, since MySQL 8 supports Common Table Expressions and ranking (window) functions, and here we can do without redefining the number of taken books by each author (because the subquery on lines 12-17 differs from the main part of the query in lines 2-9 only by omitting of `a_id` and `a_name` fields, otherwise it is a complete duplication of code).

Since the modification of this solution using Common Table Expressions in MySQL is identical to the solution for MS SQL Server and Oracle, it will be described further on the example of MS SQL Server, and here we will only give the code for MySQL (version 8 and newer).

MySQL Solution 2.2.7.b (version 8 and newer)

```
1    -- Option 1: using MAX function (MySQL 8 and newer)
2    WITH `prepared_data`
3         AS (SELECT `authors`.`a_id`,
4                    `authors`.`a_name`,
5                    COUNT(`sb_book`) AS `books`
6             FROM   `authors`
7                    JOIN `m2m_books_authors`
8                      ON `authors`.`a_id` = `m2m_books_authors`.`a_id`
9                    LEFT OUTER JOIN `subscriptions`
10                                   ON `m2m_books_authors`.`b_id` = `sb_book`
11            GROUP  BY `authors`.`a_id`,
12                      `authors`.`a_name`)
13   SELECT `a_id`,
14          `a_name`,
15          `books`
16   FROM   `prepared_data`
17   WHERE  `books` = (SELECT MAX(`books`)
18                     FROM   `prepared_data`)
```

MySQL Solution 2.2.7.b (version 8 and newer)

```
1    -- Option 2: using ranking (MySQL 8 and newer)
2    WITH `prepared_data`
3         AS (SELECT `authors`.`a_id`,
4                    `authors`.`a_name`,
5                    COUNT(`sb_book`)                      AS `books`,
6                    RANK()
7                      OVER (
8                        ORDER BY COUNT(`sb_book`) DESC) AS `rank`
9             FROM   `authors`
10                   JOIN `m2m_books_authors`
11                     ON `authors`.`a_id` = `m2m_books_authors`.`a_id`
12                   LEFT OUTER JOIN `subscriptions`
13                                ON `m2m_books_authors`.`b_id` = `sb_book`
14            GROUP  BY `authors`.`a_id`,
15                      `authors`.`a_name`)
16   SELECT `a_id`,
17          `a_name`,
18          `books`
19   FROM   `prepared_data`
20   WHERE  `rank` = 1
```

Moving on to MS SQL Server.

MS SQL Solution 2.2.7.b

```
1    -- Option 1: using MAX function
2    WITH [prepared_data]
3         AS (SELECT [authors].[a_id],
4                    [authors].[a_name],
5                    COUNT([sb_book]) AS [books]
6             FROM   [authors]
7                    JOIN [m2m_books_authors]
8                      ON [authors].[a_id] = [m2m_books_authors].[a_id]
9                    LEFT OUTER JOIN [subscriptions]
10                                ON [m2m_books_authors].[b_id] = [sb_book]
11            GROUP  BY [authors].[a_id],
12                      [authors].[a_name])
13   SELECT [a_id],
14          [a_name],
15          [books]
16   FROM   [prepared_data]
17   WHERE  [books] = (SELECT MAX([books])
18                     FROM   [prepared_data])

1    -- Option 2: using ranking
2    WITH [prepared_data]
3         AS (SELECT [authors].[a_id],
4                    [authors].[a_name],
5                    COUNT([sb_book])                      AS [books],
6                    RANK()
7                      OVER (
8                        ORDER BY COUNT([sb_book]) DESC) AS [rank]
9             FROM   [authors]
10                   JOIN [m2m_books_authors]
11                     ON [authors].[a_id] = [m2m_books_authors].[a_id]
12                   LEFT OUTER JOIN [subscriptions]
13                                ON [m2m_books_authors].[b_id] = [sb_book]
14            GROUP  BY [authors].[a_id],
15                      [authors].[a_name])
16   SELECT [a_id],
17          [a_name],
18          [books]
19   FROM   [prepared_data]
20   WHERE  [rank] = 1
```

MS SQL Solution 2.2.7.b (continued)

```
1    -- Option 3: using TOP ... WITH TIES
2    WITH [prepared_data]
3         AS (SELECT [authors].[a_id],
4                    [authors].[a_name],
5                    COUNT([sb_book]) AS [books]
6             FROM   [authors]
7                    JOIN [m2m_books_authors]
8                      ON [authors].[a_id] = [m2m_books_authors].[a_id]
9                    LEFT OUTER JOIN [subscriptions]
10                                   ON [m2m_books_authors].[b_id] = [sb_book]
11            GROUP  BY [authors].[a_id],
12                      [authors].[a_name])
13   SELECT TOP 1 WITH TIES [a_id],
14                          [a_name],
15                          [books]
16   FROM   [prepared_data]
17   ORDER  BY [books] DESC
```

Option 1 of solution for MS SQL Server differs from option 1 for MySQL (before version 8) in that due to the possibility of using a Common Table Expression, we can avoid double calculation of the number of subscriptions of each author's books: this information is once defined in the Common Table Expression (lines 2-12), and on the same data the maximum number of subscriptions (lines 17-18) is calculated. A similar solution was implemented for MySQL (version 8 and newer).

Option 2 is based on ranking authors by the number of times their books are taken by readers and then selecting the authors who are ranked first. The Common Table Expression in rows 2-15 returns the following data:

a_id	a_name	books	rank
6	Bjarne Stroustrup	4	1
7	Alexander Pushkin	4	1
2	Isaac Asimov	2	3
3	Dale Carnegie	2	3
1	Donald Knuth	1	5
4	Lev Landau	0	6
5	Evgeny Lifshitz	0	6

On line 20, a restriction is imposed on this set, putting only authors with a rank value of 1 into the final result.

Option 3 is based on using MS SQL Server specific syntax `TOP ... WITH TIES` to select a limited number of first records and add to this set a few more records, matching the first ones by the value of ordering field.

First, we get the following data set:

a_id	a_name	Books
6	Bjarne Stroustrup	4
7	Alexander Pushkin	4
2	Isaac Asimov	2
3	Dale Carnegie	2
1	Donald Knuth	1
4	Lev Landau	0
5	Evgeny Lifshitz	0

Then, thanks to `SELECT TOP 1 WITH TIES ... FROM [prepared_data] ORDER BY [books] DESC` MS SQL Server leaves only the first record (`TOP 1`) and adds to it (`WITH TIES`) all other records with the same value in the `books` field, because the `ORDER BY [books]` is used for ordering. In our case this value is 4. This is how we get the final result:

a_id	a_name	books
6	Bjarne Stroustrup	4
7	Alexander Pushkin	4

The following solution of problem 2.2.7.b for Oracle is fully equivalent to the solution for MS SQL Server except for the absence of the third option, because Oracle does not support the `TOP ... WITH TIES` syntax.

Oracle | Solution 2.2.7.b

```
1    -- Option 1: using MAX function
2    WITH "prepared_data"
3         AS (SELECT "a_id",
4                    "a_name",
5                    COUNT("sb_book") AS "books"
6               FROM   "authors"
7                    JOIN "m2m_books_authors" USING("a_id")
8                    LEFT OUTER JOIN "subscriptions"
9                                 ON "m2m_books_authors"."b_id" = "sb_book"
10              GROUP  BY "a_id",
11                        "a_name")
12   SELECT "a_id",
13          "a_name",
14          "books"
15   FROM   "prepared_data"
16   WHERE  "books" = (SELECT MAX("books")
17                     FROM   "prepared_data")

1    -- Option 2: using ranking
2    WITH "prepared_data"
3         AS (SELECT "a_id",
4                    "a_name",
5                    COUNT("sb_book")                    AS "books",
6                    RANK()
7                      OVER (
8                        ORDER BY COUNT("sb_book") DESC) AS "rank"
9               FROM   "authors"
10                   JOIN "m2m_books_authors" USING("a_id")
11                   LEFT OUTER JOIN "subscriptions"
12                                ON "m2m_books_authors"."b_id" = "sb_book"
13              GROUP  BY "a_id",
14                        "a_name")
15   SELECT "a_id",
16          "a_name",
17          "books"
18   FROM   "prepared_data"
19   WHERE  "rank" = 1
```

Exploration 2.2.7.EXP.A: let's compare the performance of the solutions to this problem by running all queries 2.2.7.b on the "Big Library (for Experiments)" database.

Median time values after a hundred executions of each query:

	MySQL (before version 8)	MySQL (version 8 and newer)	MS SQL Server	Oracle
Option 1 (`MAX()`)	4787.841	4443.391	16.106	155.918
Option 2 (`RANK()`)	-	4434.870	8.138	1.407
Option 3 (`TOP ...`)	-	-	7.312	-

Here we observe a situation opposite to the one obtained in exploration 2.1.8.EXP.B{50}, where the option with `MAX` function turned out to be faster than the option with `RANK`. And this is quite normal, since the experiments were conducted on different data sets and in the context of different queries. I.e., the speed of query execution is affected not only by using this or that function, but also by many other parameters.

Also note the fact that (unlike MS SQL Server and Oracle) in MySQL queries based on the `MAX` function and based on the ranking are almost the same in terms of performance.

Solution 2.2.7.c{119}.

MySQL Solution 2.2.7.c

```
1    SELECT AVG(`books`) AS `avg_reading`
2    FROM   (SELECT COUNT(`sb_book`) AS `books`
3            FROM   `authors`
4                   JOIN `m2m_books_authors` USING (`a_id`)
5                   LEFT OUTER JOIN `subscriptions`
6                                ON `m2m_books_authors`.`b_id` = `sb_book`
7            GROUP  BY `a_id`) AS `prepared_data`
```

MS SQL Solution 2.2.7.c

```
1    SELECT AVG(CAST([books] AS FLOAT)) AS [avg_reading]
2    FROM   (SELECT COUNT([sb_book]) AS [books]
3            FROM   [authors]
4                   JOIN [m2m_books_authors]
5                     ON [authors].[a_id] = [m2m_books_authors].[a_id]
6                   LEFT OUTER JOIN [subscriptions]
7                                ON [m2m_books_authors].[b_id] = [sb_book]
8            GROUP  BY [authors].[a_id]) AS [prepared_data]
```

Oracle Solution 2.2.7.c

```
1    SELECT AVG("books") AS "avg_reading"
2    FROM   (SELECT COUNT("sb_book") AS "books"
3            FROM   "authors"
4                   JOIN "m2m_books_authors" USING ("a_id")
5                   LEFT OUTER JOIN "subscriptions"
6                                ON "m2m_books_authors"."b_id" = "sb_book"
7            GROUP  BY "a_id") "prepared_data"
```

The solution of this problem can also be presented using a Common Table Expression (see self-study task 2.2.7.TSK.E{134}).

The subquery in the `FROM` section plays the role of data source and counts the number of subscriptions for each author. Then in the main query section (line 1 for all three DBMS) the desired average value is extracted from the prepared set.

Solution 2.2.7.d{119}.

Note how easy it is to solve this problem in Oracle (which supports `MEDIAN` function), and how non-trivial the solutions for MySQL and MS SQL Server (up to version 2017) are.

The solution logic for MySQL and MS SQL Server (before version 2017) is based on mathematical determination of the median value: the data set is sorted, and then for sets with odd number of elements the median value is the value of the central element, and for sets with an even number of elements the median value is the average value of the two central elements. Let's explain by an example.

Suppose we have the following data set with an odd number of elements:

Element position	Element value
1	40
2	65
3	90

← median = 65

The central element is element number 2, and its value 65 is the median for this set.

If we have a data set with an even number of elements:

Element position	Element value
1	40
2	65
3	90
4	95

← median = (65 + 90) / 2 = 77.5

The central elements are elements with numbers 2 and 3, and the average of their values (65 + 90) / 2 is the median for this set.

So, we need to:

- Get a sorted data set.
- Determine the number of elements in the set.
- Determine the central elements of the set.
- Get the average value of those central elements.

We may consider cases where there is one central element and cases where there are two as the same, and calculate the average value in both cases, since the average value of any one number is the number itself.

Let's look at the solution for MySQL (before version 8, without support for Common Table Expressions). The explanation will be long, but it also touches upon the mathematical logic, so don't skip it even if you never intended to work with MySQL before version 8.

MySQL | Solution 2.2.7.d (before version 8)

```
1    SELECT AVG(`books`) AS `med_reading`
2    FROM   (SELECT @rownum := @rownum + 1 AS `row_number`,
3                   `books`
4            FROM   (SELECT COUNT(`sb_book`) AS `books`
5                    FROM   `authors`
6                           JOIN `m2m_books_authors` USING (`a_id`)
7                           LEFT OUTER JOIN `subscriptions`
8                           ON `m2m_books_authors`.`b_id` = `sb_book`
9                    GROUP  BY `a_id`) AS `inner_data`,
10                  (SELECT @rownum := 0) AS `rownum_initialisation`
11           ORDER  BY `books`) AS `popularity`,
12          (SELECT COUNT(*) AS `row_count`
13           FROM   (SELECT COUNT(`sb_book`) AS `books`
14                   FROM   `authors`
15                          JOIN `m2m_books_authors` USING (`a_id`)
16                          LEFT OUTER JOIN `subscriptions`
17                          ON `m2m_books_authors`.`b_id` = `sb_book`
18                   GROUP  BY `a_id`) AS `inner_data`) AS `total_rows`
19   WHERE  `row_number` IN ( FLOOR(( `row_count` + 1 ) / 2),
20                            FLOOR(( `row_count` + 2 ) / 2) )
```

Let's start with the most deeply nested subqueries in lines 4-9 and 13-18: it is easy to see that they are completely duplicated (alas, we cannot prepare this data once and use it repeatedly in MySQL before version 8). Both subqueries return the following data:

books
1
2
2
0
0
4
4

The subquery on lines 12-18 determines the number of rows in this data set and returns one number: 7.

The subquery in rows 2-11 arranges this data set and numbers its rows. Since MySQL before version 8 has no built-in functions for numbering rows, we have to get the desired effect in several steps:

- The `SELECT @rownum := 0` clause on line 10 initializes the `@rownum` variable with 0.
- The `SELECT @rownum := @rownum + 1 AS `row_number`` clause on line 2 increments the `@rownum` by 1 for each subsequent row of the selected data. The column where the row numbers will be located will be called `row_number`.

The result of the subquery on lines 2-11 is as follows:

row_number	books
1	0
2	0
3	1
4	2
5	2
6	4
7	4

Let's step up a level and see what the subquery on lines 2-18 returns in its entirety:

row_number	books	row_count
1	0	7
2	0	7
3	1	7
4	2	7
5	2	7
6	4	7
7	4	7

The repeating value of `row_count` looks a bit annoying, but for this solution such approach is the simplest.

The `WHERE` clause in lines 19 and 20 indicates the necessity to take for the final analysis only those rows whose `row_number` values satisfy the following conditions:

- Condition 1: `FLOOR(( `row_count` + 1 ) / 2)`.
- Condition 2: `FLOOR(( `row_count` + 2 ) / 2)`.

In our particular case RowCount = 7, so, we get:

- Condition 1: `FLOOR(( 7 + 1 ) / 2) = FLOOR (8 / 2) = 4`.
- Condition 2: `FLOOR(( 7 + 2 ) / 2) = FLOOR (9 / 2) = 4`.

Both conditions point to the same row number 4. The value 2 of the `books` field from row 4 is passed to the `AVG` function (the first line of the query), and since `AVG(2) = 2`, we get the final result: the median is 2.

If the number of rows were even (e.g., 8), the conditions in lines 19 and 20 would take the following values:

- Condition 1: `FLOOR(( 8 + 1 ) / 2) = FLOOR (9 / 2) = 4`.
- Condition 2: `FLOOR(( 8 + 2 ) / 2) = FLOOR (10 / 2) = 5`.

Starting with MySQL version 8, which already has support for Common Table Expressions and ranking (window) functions, the solution can be greatly simplified.

MySQL Solution 2.2.7.d (version 8 and newer)

```
1   WITH `authors_popularity` AS
2   (
3    SELECT COUNT(`sb_book`) AS `books`,
4           ROW_NUMBER() OVER (ORDER BY COUNT(`sb_book`) ASC) AS `row_number`
5      FROM `authors`
6      JOIN `m2m_books_authors` USING (`a_id`)
7      LEFT OUTER JOIN `subscriptions`
8        ON `m2m_books_authors`.`b_id` = `sb_book`
9    GROUP  BY `a_id`
10  )
11  SELECT AVG(`books`) AS `med_reading`
12  FROM `authors_popularity`,
13       (SELECT COUNT(*) AS `row_count`
14          FROM `authors_popularity`) AS `total_rows`
15  WHERE  `row_number` IN ( FLOOR(( `row_count` + 1 ) / 2),
16                           FLOOR(( `row_count` + 2 ) / 2) )
```

Here, the Common Table Expression (lines 1-10 of the query) gives the following result:

books	row_number
0	1
0	2
1	3
2	4
2	5
4	6
4	7

In the first column (`books`) we get information about how many times readers have borrowed this or that book by this or that author (we are not interested either in books' ids or authors' ids, only the number is important), and in the second column (`row_number`) we see the serial number of the table row.

Note that because of the `ORDER BY COUNT(`sb_book`) ASC` in the 4th line of the query, the table rows are already ordered by ascending values of the `books` column.

Now we just need to find out the total number of records (we get it by subquery on lines 13-14) and use it to determine the numbers of the records we're interested in (which we do in lines 15-16 of the query).

Mathematically, this solution is no different from the version for MySQL before version 8 (which is described in detail above), here we only greatly simplified the query syntax thanks to new features of the DBMS.

Let's move on to a solution for MS SQL Server.

Due to the presence of Common Table Expressions and row numbering functions, the solution for MS SQL Server is much easier. However, until version 2017, this DBMS does not support special functions that would simplify this task, so the code is still quite cumbersome.

MS SQL | Solution 2.2.7.d (before version 2017)

```
1    WITH [popularity]
2         AS (SELECT COUNT([sb_book]) AS [books]
3             FROM   [authors]
4                    JOIN [m2m_books_authors]
5                      ON [authors].[a_id] = [m2m_books_authors].[a_id]
6                    LEFT OUTER JOIN [subscriptions]
7                                 ON [m2m_books_authors].[b_id] = [sb_book]
8             GROUP  BY [authors].[a_id]),
9         [meadian_preparation]
10        AS (SELECT CAST([books] AS FLOAT) AS [books],
11                   ROW_NUMBER()
12                     OVER (
13                       ORDER BY [books])  AS [row_number],
14                   COUNT(*)
15                     OVER (
16                       PARTITION BY NULL) AS [row_count]
17            FROM   [popularity])
18   SELECT AVG([books]) AS [med_reading]
19   FROM   [meadian_preparation]
20   WHERE  [row_number] IN ( ( [row_count] + 1 ) / 2, ( [row_count] + 2 ) / 2 )
```

The first Common Table Expression in lines 1-8 produces the following data:

books
1
2
2
0
0
4
4

The second Common Table Expression refines this data set, resulting in:

books	row_number	row_count
0	1	7
0	2	7
1	3	7
2	4	7
2	5	7
4	6	7
4	7	7

The main part of the query in lines 18-20 works just like the main part of the MySQL solution before version 8 (lines 1 and 19-20): the serial numbers of the central rows are determined and the average value of the **`books`** field values of these rows is calculated, which is the required value of the median.

As of version 2017, MS SQL Server supports the **`PERCENTILE_CONT`** function, the use of which greatly simplifies the solution.

MS SQL | Solution 2.2.7.d (version 2017 and newer)

```
1    WITH [popularity]
2         AS (SELECT COUNT([sb_book]) AS [books]
3             FROM   [authors]
4                    JOIN [m2m_books_authors]
5                      ON [authors].[a_id] = [m2m_books_authors].[a_id]
6                    LEFT OUTER JOIN [subscriptions]
7                                 ON [m2m_books_authors].[b_id] = [sb_book]
8             GROUP  BY [authors].[a_id])
9    SELECT DISTINCT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY [books])
10                                        OVER () AS [Median]
11   FROM   [popularity]
```

Here the Common Table Expression (lines 1-8 of the query) returns the following result (the popularity of each book by each author, i.e., how many times readers borrowed this book):

books
1
2
2
0
0
4
4

Then using the **PERCENTILE_CONT** function (lines 9-10 of the query) we immediately get the final result. Pay special attention to the keyword **DISTINCT** in line 9 of the query: without it, we would get an entire column of values instead of a single median value (the result would have the same number of rows as the original set of records to be analyzed).

Let's move on to looking at the solution for Oracle.

Oracle Solution 2.2.7.d

```
1    WITH "popularity"
2         AS (SELECT COUNT("sb_book") AS "books"
3             FROM   "authors"
4                    JOIN "m2m_books_authors" USING ("a_id")
5                    LEFT OUTER JOIN "subscriptions"
6                                 ON "m2m_books_authors"."b_id" = "sb_book"
7             GROUP  BY "a_id")
8    SELECT MEDIAN("books") AS "med_reading" FROM "popularity"
```

Thanks to the **MEDIAN** function in Oracle, the solution comes down to preparing the set of values who's median we are looking for, and... calling the **MEDIAN** function. The Common Table Expression in lines 1-7 prepares a data set that we already know very well from the solutions for the other two DBMSes:

books
1
2
2
0
0
4
4

Solution 2.2.7.e{119}.

To solve this problem, it is necessary to:

- Determine for each book the number of copies given to readers.
- Subtract this value from the number of copies of the book registered in the library.
- Check if there are books for which the result of this subtraction is negative, and return 0 if there are no such books, and 1 if there are such books.

MySQL Solution 2.2.7.e

```
1    SELECT EXISTS (SELECT `b_id`
2                   FROM   `books`
3                          LEFT OUTER JOIN (SELECT `sb_book`,
4                                                  COUNT(`sb_book`) AS `taken`
5                                           FROM   `subscriptions`
6                                           WHERE  `sb_is_active` = 'Y'
7                                           GROUP  BY `sb_book`
8                                           ) AS `books_taken`
9                                        ON `b_id` = `sb_book`
10                  WHERE  ( `b_quantity` - IFNULL(`taken`, 0) ) < 0
11                  LIMIT 1)
12          AS `error_exists`
```

MySQL interprets `TRUE` and `FALSE` as `1` and `0` respectively, so on the top level of the query (line 1) we can simply return the value of the `EXISTS` function.

The subquery on lines 3-9 returns the following data (the number of copies given to readers, for each book of which at least one copy was given):

sb_book	taken
1	2
3	1
4	1
5	1

The subquery on lines 1-11 returns no more than one row containing books' identifiers with a negative balance ("no more than one" since we are simply interested in the presence or absence of such books). If this subquery returns zero rows, `EXISTS` function will return `FALSE` (`0`) on an empty set. If the subquery returns one row, the set is already nonempty, and the `EXISTS` function returns `TRUE` (`1`).

Since in MS SQL Server and Oracle we cannot directly get 1 and 0 from the result of the `EXISTS` function, the solutions for these DBMSes will be a bit more complicated.

MS SQL Solution 2.2.7.e

```
1    WITH [books_taken]
2         AS (SELECT [sb_book],
3                    COUNT([sb_book]) AS [taken]
4             FROM   [subscriptions]
5             WHERE  [sb_is_active] = 'Y'
6             GROUP  BY [sb_book])
7    SELECT TOP 1 CASE
8                   WHEN EXISTS (SELECT TOP 1 [b_id]
9                                FROM   [books]
10                                      LEFT OUTER JOIN [books_taken]
11                                                   ON [b_id] = [sb_book]
12                               WHERE  ([b_quantity] - ISNULL([taken], 0)) < 0)
13                  THEN 1
14                  ELSE 0
15                END AS [error_exists]
16   FROM   [books_taken]
```

The Common Table Expression in lines 1-6 returns information about how many copies of each book were taken by readers:

sb_book	taken
1	2
3	1
4	1
5	1

The subquery on lines 8-12 returns no more than one book identifier with a "negative balance" and passes this information to the `EXISTS` function, which returns `TRUE` or `FALSE` depending on whether such books are found or not.

The clause of `CASE ... WHEN ... THEN ... ELSE ... END` in lines 7-15 converts a logical result of the `EXISTS` function into the `1` and `0` required by the problem.

Specifying `TOP 1` in line 7 is necessary to ensure that the final result contains one row, and not as many rows as the Common Table Expression will return.

Oracle | Solution 2.2.7.e

```
1    WITH "books_taken"
2         AS (SELECT "sb_book",
3                    COUNT("sb_book") AS "taken"
4             FROM   "subscriptions"
5             WHERE  "sb_is_active" = 'Y'
6             GROUP  BY "sb_book")
7    SELECT CASE
8                   WHEN EXISTS (SELECT "b_id"
9                                FROM   "books"
10                                      LEFT OUTER JOIN "books_taken"
11                                                   ON "b_id" = "sb_book"
12                               WHERE  ("b_quantity" - NVL("taken", 0)) < 0
13                               AND ROWNUM = 1)
14                  THEN 1
15                  ELSE 0
16               END AS "error_exists"
17   FROM   "books_taken"
18   WHERE ROWNUM = 1
```

Solution for Oracle is identical to the solution for MS SQL Server, except for using the `WHERE ROWNUM = 1` clause instead of `TOP 1`, which allows us to return no more than one row.

In Oracle (as well as in MS SQL Server) there is one more option of the solution, it is not worse and not better, it is simply built on a different logic and, perhaps, will seem simpler to someone.

Oracle | Solution 2.2.7.e (alternative solution)

```
1    SELECT
2        CASE
3            WHEN COUNT("b_id") > 0
4            THEN 1
5            ELSE 0
6        END AS "error_exists"
7    FROM
8    (
9        SELECT "b_id"
10       FROM   "books"
11       LEFT OUTER JOIN
12       (
13           SELECT "sb_book",
14                  COUNT("sb_book") AS "taken"
15           FROM   "subscriptions"
16           WHERE  "sb_is_active" = 'Y'
17           GROUP BY "sb_book"
18       ) "books_taken"
19       ON     "b_id" = "books_taken"."sb_book"
20       WHERE "b_quantity" < NVL("books_taken"."taken", 0)
21       FETCH FIRST 1 ROWS ONLY
22   ) "final_data"
```

Here we “at once” get information about whether the library has the error we are looking for (lines 8-22 of the query), and then interpret (lines 2-6) the result of the query as it is prescribed by the problem condition.

Task 2.2.7.TSK.A: show the readability of genres, i.e., all genres and the number of times books of these genres have been borrowed by readers.

Task 2.2.7.TSK.B: show the most-read genre, i.e., the genre (or genres, if there are several) of books that readers borrowed most often.

Task 2.2.7.TSK.C: show the average readability of the genres, i.e., the average of how many times readers borrowed books of each genre.

Task 2.2.7.TSK.D: show the median readability of genres, i.e., the median value from how many times readers borrowed books from each genre.

Task 2.2.7.TSK.E: rewrite solution 2.2.7.c{125} for MySQL, MS SQL Server, and Oracle using Common Table Expressions.

Task 2.2.7.TSK.F: implement the alternative version of Oracle's solution 2.2.7.e{131} for MS SQL Server.

2.2.8. Example 18: Multiple and Compound Conditions

Problem 2.2.8.a[135]: show authors who have worked **simultaneously** in two or more genres (i.e., at least one book by an author must simultaneously belong to two or more genres).

Problem 2.2.8.b[137]: show authors who have worked in two or more genres (even if each individual author's book belongs to only one genre).

Expected result 2.2.8.a.

a_id	a_name	genres_count
1	Donald Knuth	2
3	Dale Carnegie	2
6	Bjarne Stroustrup	2
7	Alexander Pushkin	2

Expected result 2.2.8.b.

a_id	a_name	genres_count
1	Donald Knuth	2
3	Dale Carnegie	2
6	Bjarne Stroustrup	2
7	Alexander Pushkin	2

On the existing data set, the expected results 2.2.8.a and 2.2.8.b are the same, but it by no means follows that they must always be the same.

Solution 2.2.8.a[135].

MySQL Solution 2.2.8.a

```
1    SELECT `a_id`,
2           `a_name`,
3           MAX(`genres_count`) AS `genres_count`
4    FROM   (SELECT `a_id`,
5                   `a_name`,
6                   COUNT(`g_id`) AS `genres_count`
7            FROM   `authors`
8                   JOIN `m2m_books_authors` USING (`a_id`)
9                   JOIN `m2m_books_genres` USING (`b_id`)
10           GROUP  BY `a_id`,
11                     `b_id`
12           HAVING `genres_count` > 1) AS `prepared_data`
13   GROUP  BY `a_id`
```

There are two steps to obtain the result.

The first step is represented by the subquery on lines 4-12: here we join three tables (information about authors is taken from `authors` table, information about books written by each author is taken from `m2m_books_authors` table, and information about genres each book belongs to is taken from `m2m_books_genres` table) and only the authors who have books belonging to two or more genres get into the result.

The second step (represented by the main part of the query in lines 1-3 and 13) is needed to correctly handle the situation when some author has several books belonging to two or more genres, but each belonging to a different number of genres. The data from the subquery would look like this (please note that `DISTINCT` will not help here, because the 4th and 5th entries have different values of the `genres_count` field):

a_id	a_name	genres_count
1	Donald Knuth	2
3	Dale Carnegie	2
6	Bjarne Stroustrup	2
7	Alexander Pushkin	**2**
7	Alexander Pushkin	**3**

This value (3) is given here to illustrate the idea. On the existing data set we will have 2 here.

But due to re-grouping by authors' identifiers with a search for as many genres as possible, this data would take on this final appearance:

a_id	a_name	genres_count
1	Donald Knuth	2
3	Dale Carnegie	2
6	Bjarne Stroustrup	2
7	Alexander Pushkin	3

This value (3) is given here to illustrate the idea. On the existing data set we will have 2 here.

Since problems 2.2.8.a and 2.2.8.b are similar in many ways, to better understand their key differences we will now look at the internal data returned by the subquery on lines 4-12. Let's rewrite this part by adding book information to the result and removing the "two or more genres" condition:

MySQL Solution 2.2.8.a (modified subquery code)

```
1    SELECT `a_id`,
2           `a_name`,
3           `b_id`,
4           `b_name`,
5           COUNT(`g_id`) AS `genres_count`
6    FROM   `authors`
7           JOIN `m2m_books_authors` USING (`a_id`)
8           JOIN `m2m_books_genres` USING (`b_id`)
9           JOIN `books` USING (`b_id`)
10   GROUP  BY `a_id`,
11             `b_id`
```

By double-grouping by author and book identifier, we get information about the number of genres of each individual book by each author:

a_id	a_name	b_id	b_name	gen-res_count
1	Donald Knuth	7	The Art of Computer Programming	2
2	Isaac Asimov	3	Foundation and Empire	1
3	Dale Carnegie	4	Programming Psychology	2
4	Lev Landau	6	Course of Theoretical Physics	1
5	Evgeny Lifshitz	6	Course of Theoretical Physics	1
6	Bjarne Stroustrup	4	Programming Psychology	2
6	Bjarne Stroustrup	5	The C++ Programming Language	1
7	Alexander Pushkin	1	Eugene Onegin	2
7	Alexander Pushkin	2	The Fisherman and the Golden Fish	2

In other words, here we count "the number of genres of each book," but in problem 2.2.8.b we will count "the number of genres of the author".

The solution of this problem for MS SQL Server and Oracle differs from the solution for MySQL only in nuances of syntax.

MS SQL Solution 2.2.8.a

```
SELECT [a_id],
       [a_name],
       MAX([genres_count]) AS [genres_count]
FROM   (SELECT [authors].[a_id],
               [authors].[a_name],
               COUNT([m2m_books_genres].[g_id]) AS [genres_count]
        FROM   [authors]
               JOIN [m2m_books_authors]
                 ON [authors].[a_id] = [m2m_books_authors].[a_id]
               JOIN [m2m_books_genres]
                 ON [m2m_books_authors].[b_id] = [m2m_books_genres].[b_id]
        GROUP  BY [authors].[a_id],
                  [a_name],
                  [m2m_books_authors].[b_id]
        HAVING COUNT([m2m_books_genres].[g_id]) > 1) AS [prepared_data]
GROUP  BY [a_id],
          [a_name]
```

Oracle Solution 2.2.8.a

```
SELECT "a_id",
       "a_name",
       MAX("genres_count") AS "genres_count"
FROM   (SELECT "a_id",
               "a_name",
               COUNT("g_id") AS "genres_count"
        FROM   "authors"
               JOIN "m2m_books_authors" USING ("a_id")
               JOIN "m2m_books_genres" USING ("b_id")
        GROUP  BY "a_id",
                  "a_name",
                  "b_id"
        HAVING COUNT("g_id") > 1) "prepared_data"
GROUP  BY "a_id",
          "a_name"
```

Solution 2.2.8.b[135].

As just emphasized in the explanation of the solution[135] to problem 2.2.8.a[135], here we have to count the “number of genres of the author”.

MySQL Solution 2.2.8.b

```
SELECT `a_id`,
       `a_name`,
       COUNT(`g_id`) AS `genres_count`
FROM   (SELECT DISTINCT `a_id`,
                        `g_id`
        FROM   `m2m_books_genres`
               JOIN `m2m_books_authors` USING (`b_id`)
       ) AS `prepared_data`
       JOIN `authors` USING (`a_id`)
GROUP  BY `a_id`
HAVING `genres_count` > 1
```

Let's modify the subquery again (lines 4-8) and see what data it returns:

MySQL Solution 2.2.8.b (modified subquery code)

```
1     SELECT DISTINCT `a_id`,
2                     `a_name`,
3                     `g_id`,
4                     `g_name`
5     FROM    `m2m_books_genres`
6             JOIN `m2m_books_authors` USING (`b_id`)
7             JOIN `authors` USING (`a_id`)
8             JOIN `genres` USING (`g_id`)
9     ORDER  BY `a_id`,
10               `g_id`
```

The data comes out like this (a list without duplication of all the genres in which the author worked):

a_id	a_name	g_id	g_name
1	Donald Knuth	2	Programming
1	Donald Knuth	5	Classic
2	Isaac Asimov	6	Science Fiction
3	Dale Carnegie	2	Programming
3	Dale Carnegie	3	Psychology
4	Lev Landau	5	Classic
5	Evgeny Lifshitz	5	Classic
6	Bjarne Stroustrup	2	Programming
6	Bjarne Stroustrup	3	Psychology
7	Alexander Pushkin	1	Poetry
7	Alexander Pushkin	5	Classic

In the main part of the query (lines 1-3 and lines 9-11) we only need to calculate the number of items in the genre list for each author, and then leave those authors whose number is greater than one. This is how we get the final result.

The solution of this problem for MS SQL Server and Oracle differs from the solution for MySQL only in nuances of syntax.

MS SQL Solution 2.2.8.b

```
1     SELECT [prepared_data].[a_id],
2            [a_name],
3            COUNT([g_id]) AS [genres_count]
4     FROM   (SELECT DISTINCT [m2m_books_authors].[a_id],
5                             [m2m_books_genres].[g_id]
6             FROM   [m2m_books_genres]
7                    JOIN [m2m_books_authors]
8                      ON [m2m_books_genres].[b_id] = [m2m_books_authors].[b_id])
9     AS
10           [prepared_data]
11           JOIN [authors]
12             ON [prepared_data].[a_id] = [authors].[a_id]
13    GROUP  BY [prepared_data].[a_id],
14              [a_name]
15    HAVING COUNT([g_id]) > 1
```

Oracle | Solution 2.2.8.b

```
SELECT "a_id",
       "a_name",
       COUNT("g_id") AS "genres_count"
FROM   (SELECT DISTINCT "a_id",
                        "g_id"
        FROM   "m2m_books_genres"
               JOIN "m2m_books_authors" USING ("b_id")
       ) "prepared_data"
       JOIN "authors" USING ("a_id")
GROUP  BY "a_id",
          "a_name"
HAVING COUNT("g_id") > 1
```

Task 2.2.8.TSK.A: rewrite the solutions to problems 2.2.8.a{135} and 2.2.8.b{137} for MS SQL Server and Oracle using Common Table Expressions.

Task 2.2.8.TSK.B: show readers taking the most diverse genre books (i.e., books that belong to the maximum number of genres at the same time).

Task 2.2.8.TSK.C: show readers of the greatest number of genres (whether they borrowed books that each belonged to many genres at once, or simply many books from different genres, each of which belonged to a small number of genres).

2.2.9. Example 19: JOINs with MIN, MAX, AVG, ranges

Problem 2.2.9.a{140}: show the reader who was the first to borrow a book from the library.

Problem 2.2.9.b{142}: show the reader (or readers, if there are several) who read the book the fastest (only when the book is returned).

Problem 2.2.9.c{146}: show which book (or books, if more than one) each reader borrowed on their first visit to the library.

Problem 2.2.9.d{149}: show the first book each reader borrowed from the library.

Expected result 2.2.9.a.

s_name
Ivanov I.I.

Expected result 2.2.9.b.

s_id	s_name	days
1	Ivanov I.I.	31

Expected result 2.2.9.c.

s_id	s_name	books_list
1	Ivanov I.I.	Eugene Onegin, Foundation and Empire
3	Sidorov S.S.	Foundation and Empire
4	Sidorov S.S.	The C++ Programming Language

These are different Sidorovs!

Expected result 2.2.9.d.

s_id	s_name	b_name
1	Ivanov I.I.	Eugene Onegin
3	Sidorov S.S.	Foundation and Empire
4	Sidorov S.S.	The C++ Programming Language

These are different Sidorovs!

Solution 2.2.9.a{140}.

In this case, we will make the assumption that the primary keys in the `subscriptions` table never change, i.e., the minimum value of the primary key really corresponds to the first fact of giving a book to a reader. Then the solution is very simple (what happens if we do not make such an assumption is shown in solution 2.2.9.c{146}).

For all three DBMSes, let's consider two solutions that differ in the logic of obtaining the minimum value of the primary key of the **`subscriptions`** table: using the **`MIN`** function and using ascending ordering followed by the use of the first row only.

MySQL Solution 2.2.9.a

```
1     -- Option 1: using MIN function
2     SELECT `s_name`
3     FROM   `subscribers`
4     WHERE  `s_id` = (SELECT `sb_subscriber`
5                      FROM   `subscriptions`
6                      WHERE  `sb_id` = (SELECT MIN(`sb_id`)
7                                        FROM   `subscriptions`))

1     -- Option 2: using ordering
2     SELECT `s_name`
3     FROM   `subscribers`
4     WHERE  `s_id` = (SELECT `sb_subscriber`
5                      FROM   `subscriptions`
6                      ORDER  BY `sb_id` ASC
7                      LIMIT 1)
```

MS SQL Solution 2.2.9.a

```
1     -- Option 1: using MIN function
2     SELECT [s_name]
3     FROM   [subscribers]
4     WHERE  [s_id] = (SELECT [sb_subscriber]
5                      FROM   [subscriptions]
6                      WHERE  [sb_id] = (SELECT MIN([sb_id])
7                                        FROM   [subscriptions]))

1     -- Option 2: using ordering
2     SELECT [s_name]
3     FROM   [subscribers]
4     WHERE  [s_id] = (SELECT TOP 1 [sb_subscriber]
5                      FROM   [subscriptions]
6                      ORDER  BY [sb_id] ASC)
```

Oracle Solution 2.2.9.a

```
1     -- Option 1: using MIN function
2     SELECT "s_name"
3     FROM   "subscribers"
4     WHERE  "s_id" = (SELECT "sb_subscriber"
5                      FROM   "subscriptions"
6                      WHERE  "sb_id" = (SELECT MIN("sb_id")
7                                        FROM   "subscriptions"))

1     -- Option 2: using ordering
2     SELECT "s_name"
3     FROM   "subscribers"
4     WHERE  "s_id" = (SELECT "sb_subscriber"
5                      FROM   (SELECT "sb_subscriber",
6                                     ROW_NUMBER()
7                                       OVER(
8                                         ORDER BY "sb_id" ASC) AS "rn"
9                              FROM   "subscriptions")
10                     WHERE  "rn" = 1)
```

Exploration 2.2.9.EXP.A: let's compare the performance of the solutions to this problem by running queries 2.2.9.a on the "Big Library (for Experiments)" database.

Median time values after one hundred executions of each query:

	MySQL	MS SQL Server	Oracle
Using MIN function	0.001	0.020	2.876
Using ordering	0.001	0.018	10.330

In MySQL and MS SQL Server both queries work with approximately comparable performance, but in Oracle emulation of `LIMIT/TOP` clause through numbering rows leads to a very noticeable performance drop.

Solution 2.2.9.b[{140}].

MySQL before version 8 does not support Common Table Expressions and ranking (window) functions, so here is the solution.

MySQL Solution 2.2.9.b (before version 8)

```
1   -- Option 1: using subquery and ordering (before version 8)
2   SELECT DISTINCT `s_id`,
3                   `s_name`,
4                   DATEDIFF(`sb_finish`, `sb_start`) AS `days`
5   FROM    `subscribers`
6           JOIN `subscriptions`
7             ON `s_id` = `sb_subscriber`
8   WHERE   `sb_is_active` = 'N'
9           AND DATEDIFF(`sb_finish`, `sb_start`) =
10              (SELECT    DATEDIFF(`sb_finish`, `sb_start`) AS `days`
11               FROM      `subscriptions`
12               WHERE     `sb_is_active` = 'N'
13               ORDER  BY `days` ASC
14               LIMIT  1)
```

The subquery on lines 10-14 returns information about the minimum number of days in which the book was returned:

days
31

The main part of the query in lines 2-8 gets information about all readers and the number of days each reader has been reading each book:

s_id	s_name	days
1	Ivanov I.I.	31
1	Ivanov I.I.	61
4	Sidorov S.S.	61

The condition in line 9 leaves out of this set only those records in which the number of days is equal to the number defined by the subquery on lines 10-14. So, we get the final data set.

Starting from version 8, MySQL already supports Common Table Expressions and ranking (window) functions, so the following two solutions can be implemented similarly to MS SQL Server and Oracle (on the example of which these solutions will be explained, here we will simply give the code for MySQL).

MySQL Solution 2.2.9.b (version 8 and newer)

```
1     -- Option 2: using Common Table Expression and MIN function
2     -- (version 8 and newer)
3     WITH `prepared_data`
4          AS (SELECT DISTINCT `s_id`,
5                              `s_name`,
6                              DATEDIFF(`sb_finish`, `sb_start`) AS `days`
7              FROM    `subscribers`
8                      JOIN `subscriptions`
9                        ON `s_id` = `sb_subscriber`
10             WHERE   `sb_is_active` = 'N')
11    SELECT `s_id`,
12           `s_name`,
13           `days`
14    FROM    `prepared_data`
15    WHERE   `days` = (SELECT MIN(`days`)
16                      FROM    `prepared_data`)
```

```
1     -- Option 3: using Common Table Expression and ranking
2     -- (version 8 and newer)
3     WITH `prepared_data`
4          AS (SELECT DISTINCT `s_id`,
5                              `s_name`,
6                              DATEDIFF(`sb_finish`, `sb_start`) AS `days`,
7                              RANK()
8                                OVER (
9                                      ORDER BY
10                                     DATEDIFF(`sb_finish`, `sb_start`) ASC
11                               ) AS `rank`
12             FROM    `subscribers`
13                     JOIN `subscriptions`
14                       ON `s_id` = `sb_subscriber`
15             WHERE   `sb_is_active` = 'N')
16    SELECT `s_id`,
17           `s_name`,
18           `days`
19    FROM    `prepared_data`
20    WHERE   `rank` = 1
```

Option 1 of solutions for MS SQL Server and Oracle are built completely similarly to the solution for MySQL (and discussed above).

MS SQL Solution 2.2.9.b

```
1     -- Option 1: using subquery and ordering
2     SELECT DISTINCT [s_id],
3                     [s_name],
4                     DATEDIFF(day, [sb_start], [sb_finish]) AS [days]
5     FROM    [subscribers]
6             JOIN [subscriptions]
7               ON [s_id] = [sb_subscriber]
8     WHERE   [sb_is_active] = 'N'
9             AND DATEDIFF(day, [sb_start], [sb_finish]) =
10            (SELECT TOP 1 DATEDIFF(day, [sb_start], [sb_finish]) AS [days]
11             FROM          [subscriptions]
12             WHERE         [sb_is_active] = 'N'
13             ORDER  BY     [days] ASC)
```

MS SQL Solution 2.2.9.b (continued)

```
1    -- Option 2: using Common Table Expression and MIN function
2    WITH [prepared_data]
3         AS (SELECT DISTINCT [s_id],
4                             [s_name],
5                             DATEDIFF(day, [sb_start], [sb_finish]) AS [days]
6             FROM    [subscribers]
7                     JOIN [subscriptions]
8                       ON [s_id] = [sb_subscriber]
9             WHERE   [sb_is_active] = 'N')
10   SELECT [s_id],
11          [s_name],
12          [days]
13   FROM    [prepared_data]
14   WHERE   [days] = (SELECT MIN([days])
15                     FROM    [prepared_data])

1    -- Option 3: using Common Table Expression and ranking
2    WITH [prepared_data]
3         AS (SELECT DISTINCT [s_id],
4                             [s_name],
5                             DATEDIFF(day, [sb_start], [sb_finish]) AS [days],
6                             RANK()
7                               OVER (
8                                     ORDER BY
9                                     DATEDIFF(day, [sb_start], [sb_finish]) ASC
10                                  ) AS [rank]
11            FROM    [subscribers]
12                    JOIN [subscriptions]
13                      ON [s_id] = [sb_subscriber]
14            WHERE   [sb_is_active] = 'N')
15   SELECT [s_id],
16          [s_name],
17          [days]
18   FROM    [prepared_data]
19   WHERE   [rank] = 1
```

In **option 2**, the Common Table Expression (lines 2-9) returns the same set of data as the main part of the query in option 1 (lines 2-8):

s_id	s_name	days
1	Ivanov I.I.	31
1	Ivanov I.I.	61
4	Sidorov S.S.	61

The subquery on lines 14-15 determines the minimum value of the `days` column, which is then used as a condition in the main part of the query (lines 10-14).

In **option 3**, the Common Table Expression in rows 2-14 prepares the already familiar data set, but also ranked by the value of the `days` column:

s_id	s_name	days	rank
1	Ivanov I.I.	31	1
1	Ivanov I.I.	61	4
4	Sidorov S.S.	61	4

In the main part of the query (lines 15-19), all that remains is to extract the “first place” rows from this prepared result (`rank = 1`).

The solution for Oracle is similar to the solution for MS SQL Server.

Oracle | Solution 2.2.9.b

```
-- Option 1: using subquery and ordering
SELECT DISTINCT "s_id",
                "s_name",
                ( "sb_finish" - "sb_start" ) AS "days"
FROM   "subscribers"
       JOIN "subscriptions"
         ON "s_id" = "sb_subscriber"
WHERE  "sb_is_active" = 'N'
       AND ( "sb_finish" - "sb_start" ) =
       (SELECT "min_days"
        FROM   (SELECT ("sb_finish" - "sb_start" )          AS "min_days",
                       ROW_NUMBER()
                        OVER(
                        ORDER BY ( "sb_finish" - "sb_start") ASC) AS "rn"
                FROM   "subscriptions"
                WHERE  "sb_is_active" = 'N')
         WHERE  "rn" = 1)
```

```
-- Option 2: using Common Table Expression and MIN function
WITH "prepared_data"
     AS (SELECT DISTINCT "s_id",
                         "s_name",
                         ( "sb_finish" - "sb_start" ) AS "days"
           FROM   "subscribers"
                  JOIN "subscriptions"
                    ON "s_id" = "sb_subscriber"
           WHERE  "sb_is_active" = 'N')
SELECT "s_id",
       "s_name",
       "days"
FROM   "prepared_data"
WHERE  "days" = (SELECT MIN("days")
                 FROM   "prepared_data")
```

```
-- Option 3: using Common Table Expression and ranking
WITH "prepared_data"
     AS (SELECT DISTINCT "s_id",
                         "s_name",
                         ("sb_finish" - "sb_start")               AS "days",
                         RANK()
                         OVER (
                         ORDER BY ("sb_finish" - "sb_start") ASC ) AS "rank"
           FROM   "subscribers"
                  JOIN "subscriptions"
                    ON "s_id" = "sb_subscriber"
           WHERE  "sb_is_active" = 'N')
SELECT "s_id",
       "s_name",
       "days"
FROM   "prepared_data"
WHERE  "rank" = 1
```

Solution 2.2.9.c{140}.

Here we will not make assumptions like those made in the solution of Problem 2.2.9.a{140}, and we will build the query solely on information related to the subject area.

So, for each reader, we need to:

- find out the date of the first visit to the library;
- get a set of books taken by the reader on that day;
- present the set of these books as a string.

MySQL | Solution 2.2.9.c

```
1    SELECT `s_id`,
2           `s_name`,
3           GROUP_CONCAT(`b_name` ORDER BY `b_name` SEPARATOR ', ')
4           AS `books_list`
5    FROM   (SELECT `s_id`,
6                  `s_name`,
7                  `b_name`
8           FROM   `subscribers`
9                  JOIN (SELECT `subscriptions`.`sb_subscriber`,
10                              `subscriptions`.`sb_book`
11                     FROM   `subscriptions`
12                            JOIN (SELECT `sb_subscriber`,
13                                         MIN(`sb_start`) AS `min_date`
14                                  FROM   `subscriptions`
15                                  GROUP  BY `sb_subscriber`)
16                                  AS `first_visit`
17                              ON `subscriptions`.`sb_subscriber` =
18                                 `first_visit`.`sb_subscriber`
19                                 AND `subscriptions`.`sb_start` =
20                                     `first_visit`.`min_date`)
21                     AS `books_list`
22                   ON `s_id` = `sb_subscriber`
23                 JOIN `books`
24                   ON `sb_book` = `b_id`) AS `prepared_data`
25   GROUP  BY `s_id`
```

The most deeply nested subquery (lines 12-16) defines the date of each reader's first visit to the library:

sb_subscriber	min_date
1	2011-01-12
3	2012-05-17
4	2012-06-11

The next nesting level subquery (lines 9-20) defines the list of books borrowed by each reader on the date of their first visit to the library:

sb_subscriber	sb_book
1	1
1	3
3	3
4	5

The next nesting level subquery (lines 5-24) prepares data with readers' names and books' titles:

s_id	s_name	b_name
1	Ivanov I.I.	Eugene Onegin
1	Ivanov I.I.	Foundation and Empire
3	Sidorov S.S.	Foundation and Empire
4	Sidorov S.S.	The C++ Programming Language

Finally, the main part of the query (lines 1-5 and 25) turns separate rows with information about the readers' books (marked with a gray background above) into a single string with a list of books. This is how we get the final result.

MS SQL Solution 2.2.9.c

```
1    WITH [step_1]
2         AS (SELECT [sb_subscriber],
3                    MIN([sb_start]) AS [min_date]
4               FROM   [subscriptions]
5               GROUP  BY [sb_subscriber]),
6         [step_2]
7         AS (SELECT [subscriptions].[sb_subscriber],
8                    [subscriptions].[sb_book]
9               FROM   [subscriptions]
10                   JOIN [step_1]
11                     ON [subscriptions].[sb_subscriber] =
12                                [step_1].[sb_subscriber]
13                        AND [subscriptions].[sb_start] = [step_1].[min_date]),
14        [step_3]
15        AS (SELECT [s_id],
16                   [s_name],
17                   [b_name]
18              FROM   [subscribers]
19                   JOIN [step_2]
20                     ON [s_id] = [sb_subscriber]
21                   JOIN [books]
22                     ON [sb_book] = [b_id])
23   SELECT [s_id],
24          [s_name],
25          STUFF
26          ((SELECT    ', ' + [int].[b_name]
27            FROM      [step_3] AS [int]
28            WHERE     [ext].[s_id] = [int].[s_id]
29            ORDER  BY [int].[b_name]
30            FOR xml path(''), type).value('.', 'nvarchar(max)'),
31            1, 2, '') AS [books_list]
32   FROM   [step_3] AS [ext]
33   GROUP BY [s_id], [s_name]
```

In the solution for MS SQL Server, Common Table Expressions return the same data as the subqueries in the solution for MySQL (let's leave changing the MySQL's solution using Common Table Expressions to a self-study task, see 2.2.9.TSK.E{158}).

The Common Table Expression `step_1` (lines 1-5) defines the date of the first visit to the library for each of the readers:

sb_subscriber	min_date
1	2011-01-12
3	2012-05-17
4	2012-06-11

The Common Table Expression `step_2` (lines 6-13) defines the list of books borrowed by each reader on the date of their first visit to the library:

sb_subscriber	sb_book
1	1
1	3
3	3
4	5

The Common Table Expression step_3 (lines 14-22) prepares data with readers' names and books' titles:

s_id	s_name	b_name
1	Ivanov I.I.	Eugene Onegin
1	Ivanov I.I.	Foundation and Empire
3	Sidorov S.S.	Foundation and Empire
4	Sidorov S.S.	The C++ Programming Language

Finally, the main part of the query (lines 23-33) turns separate rows with information about the readers' books (marked with a gray background above) into a single string with a list of books. This is how we get the final result.

The solution for Oracle is similar to the solution for MS SQL Server.

Oracle | Solution 2.2.9.c

```
1    WITH "step_1"
2         AS (SELECT "sb_subscriber",
3                    MIN("sb_start") AS "min_date"
4             FROM   "subscriptions"
5             GROUP  BY "sb_subscriber"),
6         "step_2"
7         AS (SELECT "subscriptions"."sb_subscriber",
8                    "subscriptions"."sb_book"
9             FROM   "subscriptions"
10                   join "step_1"
11                     ON "subscriptions"."sb_subscriber" =
12                                "step_1"."sb_subscriber"
13                        AND "subscriptions"."sb_start" = "step_1"."min_date"),
14        "step_3"
15        AS (SELECT "s_id",
16                   "s_name",
17                   "b_id",
18                   "b_name"
19            FROM   "subscribers"
20                   JOIN "step_2"
21                     ON "s_id" = "sb_subscriber"
22                   JOIN "books"
23                     ON "sb_book" = "b_id")
24   SELECT "s_id",
25          "s_name",
26          UTL_RAW.CAST_TO_NVARCHAR2
27          (
28           LISTAGG
29           (
30            UTL_RAW.CAST_TO_RAW("b_name"),
31            UTL_RAW.CAST_TO_RAW(N', ')
32           )
33           WITHIN GROUP (ORDER BY "b_name")
34          )
35          AS "books_list"
36   FROM   "step_3"
37   GROUP  BY "s_id",
38             "s_name"
```

It is worth noting that the last step (turning several rows with books' titles into a single string) is implemented quite differently in each DBMS due to very serious differences in syntax and in the set of supported functions. This operation is discussed in detail in problem 2.2.2.a{74}.

Starting with version 2017, MS SQL Server already supports a more compact syntax for grouping string values from multiple series (see task 2.2.2.a{74}), and therefore a simpler solution can be implemented here, which you are invited to do by performing the self-study task 2.2.9.TSK.F{158}.

Exploration 2.2.9.EXP.B: let's compare the performance of the three DBMSes when executing queries 2.2.9.c on the "Big Library (for Experiments)" database.

Median time values after a hundred executions of each query:

MySQL	MS SQL Server	Oracle
91789.850	203.083	5.167

The results are self-explanatory. In the real life situations for MySQL and MS SQL Server, we will obviously have to look for a different, albeit technically more complex, but more productive solution.

Solution 2.2.9.d{140}.

This problem is a logical continuation of problem 2.2.9.c{140}, but now we need to determine which book of those borrowed during a first visit to the library was given out first. Since the date of subscription is stored accurate to the day, we have no choice but to assume that the first book to be delivered was the one with the minimum value of the primary key.

The first solution is based on sequentially defining the first visit for each reader, the first subscription within that visit, the identifier of the book in that subscription, and the joining with the `subscribers` and `books` tables, from which we retrieve readers' names and books' titles.

The query below represents a general approach that will work fine in MySQL before version 8 as well as in newer versions of this DBMS.

Since version 8, each of the subqueries can be turned into a Common Table Expression, which you are invited to do by performing the self-study task 2.2.9.TSK.G{158}.

MySQL Solution 2.2.9.d

```
1    -- Option 1: four-steps solution without ranking
2    SELECT `s_id`,
3           `s_name`,
4           `b_name`
5    FROM   (SELECT `subscriptions`.`sb_subscriber`,
6                   `sb_book`
7            FROM   `subscriptions`
8                   JOIN (SELECT `subscriptions`.`sb_subscriber`,
9                                MIN(`sb_id`) AS `min_sb_id`
10                        FROM   `subscriptions`
11                               JOIN (SELECT `sb_subscriber`,
12                                            MIN(`sb_start`) AS `min_sb_start`
13                                     FROM   `subscriptions`
14                                     GROUP  BY `sb_subscriber`)
15                                     AS `step_1`
16                                 ON `subscriptions`.`sb_subscriber` =
17                                    `step_1`.`sb_subscriber`
18                                    AND `subscriptions`.`sb_start` =
19                                        `step_1`.`min_sb_start`
20                        GROUP  BY `subscriptions`.`sb_subscriber`,
21                                  `min_sb_start`)
22                        AS `step_2`
23                     ON `subscriptions`.`sb_id` = `step_2`.`min_sb_id`)
24           AS `step_3`
25          JOIN `subscribers`
26            ON `sb_subscriber` = `s_id`
27          JOIN `books`
28            ON `sb_book` = `b_id`
```

The most deeply nested subquery (**`step_1`**) in lines 11-15 returns information about the date of each reader's first visit to the library:

sb_subscriber	min_sb_start
1	2011-01-12
3	2012-05-17
4	2012-06-11

The next subquery (**`step_2`**) in lines 8-22, based on the information just received, determines the minimum value of the primary key of the **`subscribers`** table, corresponding to each reader and the date of their first visit to the library:

sb_subscriber	min_sb_id
1	2
3	3
4	57

The next subquery (**`step_3`**) in lines 5-24 uses this information to determine reader's and book's identifiers corresponding to each subscription with the **`min_sb_id`** identifier:

sb_subscriber	sb_book
1	1
3	3
4	5

The main part of the query in lines 2-4 and 25-28 is the fourth step, which determines the name of the reader and the book title from the available identifiers. This is how we get the final result.

MySQL Solution 2.2.9.d (before version 8)

```
1    -- Option 2: using ranking (before version 8)
2    SELECT `s_id`,
3           `s_name`,
4           `b_name`
5    FROM   (SELECT `sb_subscriber`,
6                   `sb_start`,
7                   `sb_id`,
8                   `sb_book`,
9                   ( CASE
10                      WHEN ( @sb_subscriber_value = `sb_subscriber` )
11                       THEN @i := @i + 1
12                      ELSE ( @i := 1 )
13                       AND ( @sb_subscriber_value := `sb_subscriber` )
14                    END ) AS `rank_by_subscriber`,
15                  ( CASE
16                      WHEN ( @sb_subscriber_value = `sb_subscriber` )
17                           AND ( @sb_start_value = `sb_start` )
18                       THEN @j := @j + 1
19                      ELSE ( @j := 1 )
20                           AND ( @sb_subscriber_value := `sb_subscriber` )
21                           AND ( @sb_start_value := `sb_start` )
22                    END ) AS `rank_by_date`
23            FROM   `subscriptions`,
24                   (SELECT @i := 0,
25                           @j := 0,
26                           @sb_subscriber_value := '',
27                           @sb_start_value := ''
28                   ) AS `initialisation`
29            ORDER  BY `sb_subscriber`,
30                      `sb_start`,
31                      `sb_id`) AS `ranked`
32          JOIN `subscribers`
33            ON `sb_subscriber` = `s_id`
34          JOIN `books`
35            ON `sb_book` = `b_id`
36   WHERE  `rank_by_subscriber` = 1
37          AND `rank_by_date` = 1
```

The second option of the solution is based on the idea of ranking the dates of each reader's visit and ranking the subscriptions within each visit of each reader.

Since MySQL before version 8 does not support any ranking (window) functions, their behavior will have to be emulated.

The subquery on lines 24-28 is responsible for the initial initialization of variables:

- the `@i` variable stores the serial number of each reader's visit;
- the `@j` variable stores the serial number of the book within each visit of each reader;
- the `@sb_subscriber_value` variable is used to determine the fact that the reader's identifier has changed during data processing;
- the `@sb_start_value` variable is used to determine the fact that the date value of the visit has changed during data processing.

The expression in lines 9-14 determines whether the reader's identifier has changed and either increments the ordinal number `@i` (if the identifier has not changed) or initializes it with a value of 1 (if the identifier has changed).

The expression in lines 15-22 determines whether the reader's identifier and date have changed, and either increments the ordinal number `@j` (if the identifier and date have not changed) or initializes it with a value of 1 (if the identifier or date have changed).

The result of the subquery on lines 5-22 is the following data set (rows that are of interest to us in the context of solving the problem are marked in gray):

sb_subscriber	sb_start	sb_id	sb_book	rank_by_subscriber	rank_by_date
1	2011-01-12	2	1	1	1
1	2011-01-12	100	3	2	2
1	2012-06-11	42	2	3	1
1	2014-08-03	61	7	4	1
1	2015-10-07	95	4	5	1
3	2012-05-17	3	3	1	1
3	2014-08-03	62	5	2	1
3	2014-08-03	86	1	3	2
4	2012-06-11	57	5	1	1
4	2015-10-07	91	1	2	1
4	2015-10-08	99	4	3	1

Selecting from this set the rows corresponding to the first visit of a reader and the first subscription within this visit (lines 36-37 of the query), we get:

sb_subscriber	sb_start	sb_id	sb_book	rank_by_subscriber	rank_by_date
1	2011-01-12	2	1	1	1
3	2012-05-17	3	3	1	1
4	2012-06-11	57	5	1	1

Now it remains to get the names and titles of readers and books (lines 2-4 and 32-35 of the query) based on their identifiers. This is how we get the final result.

This query can be optimized by selecting fewer fields and performing fewer checks, but then it becomes more difficult to understand.

If we use MySQL version 8 or newer, this solution can be greatly simplified by using the ranking (window) functions.

The MySQL query below differs from the MS SQL Server query only in the limiting characters of the structure names (`` ` `` instead of `[ ]`), so we will consider it in detail on the MS SQL Server example, and here we will just provide the code itself.

MySQL Solution 2.2.9.d (version 8 and newer)

```
-- Option 2: using ranking (version 8 and newer)
WITH `step_1`
     AS (SELECT `sb_subscriber`,
                `sb_start`,
                `sb_id`,
                `sb_book`,
                ROW_NUMBER()
                  OVER (
                    PARTITION BY `sb_subscriber`
                    ORDER BY `sb_start` ASC)
                AS `rank_by_subscriber`,
                ROW_NUMBER()
                  OVER (
                    PARTITION BY `sb_subscriber`, `sb_start`
                    ORDER BY `sb_subscriber`, `sb_start` ASC)
                AS `rank_by_date`
         FROM   `subscriptions`)
SELECT `s_id`,
       `s_name`,
       `b_name`
FROM   `step_1`
       JOIN `subscribers`
         ON `sb_subscriber` = `s_id`
       JOIN `books`
         ON `sb_book` = `b_id`
WHERE `rank_by_subscriber` = 1 AND `rank_by_date` = 1
```

In solutions for MS SQL Server and Oracle the first option will also be implemented without ranking.

MS SQL Solution 2.2.9.d

```
-- Option 1: four-steps solution without ranking
WITH [step_1]
     AS (SELECT [sb_subscriber],
                MIN([sb_start]) AS [min_sb_start]
         FROM   [subscriptions]
         GROUP  BY [sb_subscriber]),
     [step_2]
     AS (SELECT [subscriptions].[sb_subscriber],
                MIN([sb_id]) AS [min_sb_id]
         FROM   [subscriptions]
                JOIN [step_1]
                  ON [subscriptions].[sb_subscriber] =
                             [step_1].[sb_subscriber]
                     AND [subscriptions].[sb_start] =
                             [step_1].[min_sb_start]
         GROUP  BY [subscriptions].[sb_subscriber],
                   [min_sb_start]),
     [step_3]
     AS (SELECT [subscriptions].[sb_subscriber],
                [sb_book]
         FROM   [subscriptions]
                JOIN [step_2]
                  ON [subscriptions].[sb_id] = [step_2].[min_sb_id])
SELECT [s_id],
       [s_name],
       [b_name]
FROM   [step_3]
       JOIN [subscribers]
         ON [sb_subscriber] = [s_id]
       JOIN [books]
         ON [sb_book] = [b_id]
```

Here, Common Table Expressions return the same data sets as the subqueries with the corresponding names in the MySQL solution before version 8.

The Common Table Expression `step_1` (lines 2-6 of the query) returns information about the date of the first visit to the library for each reader:

sb_subscriber	min_sb_start
1	2011-01-12
3	2012-05-17
4	2012-06-11

The Common Table Expression `step_2` (lines 7-17 of the query) returns the minimum value of the primary key of the `subscribers` table, corresponding to each reader and the date of their first visit to the library:

sb_subscriber	min_sb_id
1	2
3	3
4	57

The Common Table Expression `step_3` (lines 18-23 of the query) returns the book identifier corresponding to each subscription with the `min_sb_id` identifier:

sb_subscriber	sb_book
1	1
3	3
4	5

The main part of the query in lines 24-31 is the fourth step, in which the readers' name and the books' titles are determined from the available identifiers. This is how we get the final result.

MS SQL Solution 2.2.9.d

```
1    -- Option 2: two-steps solution with ranking
2    WITH [step_1]
3         AS (SELECT [sb_subscriber],
4                    [sb_start],
5                    [sb_id],
6                   [sb_book],
7                    ROW_NUMBER()
8                      OVER (
9                        PARTITION BY [sb_subscriber]
10                       ORDER BY [sb_start] ASC)
11                   AS [rank_by_subscriber],
12                   ROW_NUMBER()
13                     OVER (
14                       PARTITION BY [sb_subscriber], [sb_start]
15                       ORDER BY [sb_subscriber], [sb_start] ASC)
16                   AS [rank_by_date]
17             FROM    [subscriptions])
18   SELECT [s_id],
19          [s_name],
20          [b_name]
21   FROM   [step_1]
22          JOIN [subscribers]
23            ON [sb_subscriber] = [s_id]
24          JOIN [books]
25            ON [sb_book] = [b_id]
26   WHERE [rank_by_subscriber] = 1 AND [rank_by_date] = 1
```

The Common Table Expression on lines 2-17 returns the same data as the subquery in the MySQL solution before version 8, i.e., the information on subscriptions, ranked by the number of readers' visit to the library and the number of a subscription within each visit:

sb_subscriber	sb_start	sb_id	sb_book	rank_by_subscriber	rank_by_date
1	2011-01-12	2	1	1	1
1	2011-01-12	100	3	2	2
1	2012-06-11	42	2	3	1
1	2014-08-03	61	7	4	1
1	2015-10-07	95	4	5	1
3	2012-05-17	3	3	1	1
3	2014-08-03	62	5	2	1
3	2014-08-03	86	1	3	2
4	2012-06-11	57	5	1	1
4	2015-10-07	91	1	2	1
4	2015-10-08	99	4	3	1

The main part of the query in lines 18-26 leaves from this set only each first subscription in each first visit and gets readers' names and books' titles by their identifiers. This is how we get the final result.

In MySQL before version 8, it made no sense to implement the third option, because grouping there would violate the logic of row numbering. MS SQL Server and Oracle do not have this restriction, so another solution is possible, the third option (you are invited to implement the same solution for MySQL version 8 and newer on your own within the task 2.2.9.TSK.H{158}).

MS SQL Solution 2.2.9.d

```
1    -- Option 3: three-steps solution with ranking
2    -- and grouping
3    WITH [step_1]
4         AS (SELECT [sb_subscriber],
5                    [sb_start],
6                    MIN([sb_id])                  AS [min_sb_id],
7                    RANK()
8                      OVER (
9                        PARTITION BY [sb_subscriber]
10                       ORDER BY [sb_start] ASC) AS [rank]
11             FROM   [subscriptions]
12             GROUP  BY [sb_subscriber],
13                       [sb_start]),
14        [step_2]
15        AS (SELECT [subscriptions].[sb_subscriber],
16                   [subscriptions].[sb_book]
17             FROM   [subscriptions]
18                    JOIN [step_1]
19                      ON [subscriptions].[sb_id] = [step_1].[min_sb_id]
20             WHERE  [rank] = 1)
21   SELECT [s_id],
22          [s_name],
23          [b_name]
24   FROM   [step_2]
25          JOIN [subscribers]
26            ON [sb_subscriber] = [s_id]
27          JOIN [books]
28            ON [sb_book] = [b_id]
```

The first Common Table Expression (lines 3-13 of the query) immediately finds out the smallest value of the primary key of the **subscriptions** table and ranks the resulting data by the number of the reader's visit to the library:

sb_subscriber	sb_start	min_sb_id	rank
1	2011-01-12	2	1
1	2012-06-11	42	2
1	2014-08-03	61	3
1	2015-10-07	95	4
3	2012-05-17	3	1
3	2014-08-03	62	2
4	2012-06-11	57	1
4	2015-10-07	91	2
4	2015-10-08	99	3

The second Common Table Expression removes unnecessary data and, having information about the record identifier from the **subscriptions** table, extracts from there the value of the **sb_book** field, i.e., the book identifier. Thus, the result is:

sb_subscriber	sb_book
1	1
3	3
4	5

The main part of the query in lines 21-28 is only needed to replace the readers' and books' identifiers with names and titles. This is how we get the final result.

The solution for Oracle is similar to the solution for MS SQL Server and differs only in minor details of syntax.

Oracle | Solution 2.2.9.d

```
1    -- Option 1: four-steps solution without ranking
2    WITH "step_1"
3         AS (SELECT "sb_subscriber",
4                    MIN("sb_start") AS "min_sb_start"
5             FROM   "subscriptions"
6             GROUP  BY "sb_subscriber"),
7         "step_2"
8         AS (SELECT "subscriptions"."sb_subscriber",
9                    MIN("sb_id") AS "min_sb_id"
10            FROM   "subscriptions"
11                   JOIN "step_1"
12                     ON "subscriptions"."sb_subscriber" =
13                                "step_1"."sb_subscriber"
14                        AND "subscriptions"."sb_start" =
15                                "step_1"."min_sb_start"
16            GROUP  BY "subscriptions"."sb_subscriber",
17                      "min_sb_start"),
18        "step_3"
19        AS (SELECT "subscriptions"."sb_subscriber",
20                   "sb_book"
21            FROM   "subscriptions"
22                   JOIN "step_2"
23                     ON "subscriptions"."sb_id" = "step_2"."min_sb_id")
24   SELECT "s_id",
25          "s_name",
26          "b_name"
27   FROM   "step_3"
28          JOIN "subscribers"
29            ON "sb_subscriber" = "s_id"
30          JOIN "books"
31            ON "sb_book" = "b_id"
```

Oracle | Solution 2.2.9.d (continued)

```
-- Option 2: two-steps solution with ranking
WITH "step_1"
     AS (SELECT "sb_subscriber",
                "sb_start",
                "sb_id",
                        "sb_book",
                ROW_NUMBER()
                  OVER (
                    PARTITION BY "sb_subscriber"
                    ORDER BY "sb_start" ASC)
                AS "rank_by_subscriber",
                ROW_NUMBER()
                  OVER (
                    PARTITION BY "sb_subscriber", "sb_start"
                    ORDER BY "sb_subscriber", "sb_start" ASC)
                AS "rank_by_date"
         FROM   "subscriptions")
SELECT "s_id",
       "s_name",
       "b_name"
FROM   "step_1"
       JOIN "subscribers"
         ON "sb_subscriber" = "s_id"
       JOIN "books"
         ON "sb_book" = "b_id"
WHERE "rank_by_subscriber" = 1 AND "rank_by_date" = 1
```

```
-- Option 3: three-steps solution with ranking
-- and grouping
WITH "step_1"
     AS (SELECT "sb_subscriber",
                "sb_start",
                MIN("sb_id")                  AS "min_sb_id",
                RANK()
                  OVER (
                    PARTITION BY "sb_subscriber"
                    ORDER BY "sb_start" ASC) AS "rank"
         FROM   "subscriptions"
         GROUP  BY "sb_subscriber",
                   "sb_start"),
     "step_2"
     AS (SELECT "subscriptions"."sb_subscriber",
                "subscriptions"."sb_book"
         FROM   "subscriptions"
                JOIN "step_1"
                  ON "subscriptions"."sb_id" = "step_1"."min_sb_id"
         WHERE  "rank" = 1)
SELECT "s_id",
       "s_name",
       "b_name"
FROM   "step_2"
       JOIN "subscribers"
         ON "sb_subscriber" = "s_id"
       JOIN "books"
         ON "sb_book" = "b_id"
```

Exploration 2.2.9.EXP.C: let's compare the performance of the solutions to this problem by running queries 2.2.9.d on the "Big Library (for Experiments)" database.

Median time values after a hundred executions of each query:

	MySQL	MS SQL Server	Oracle
Option 1	87135.510	31.158	2.281
Option 2	62.524	35.226	1.040
Option 3	*See 2.2.9.TSK.H*{158}	39.339	1.250

Note the difference in performance between the first and second options for MySQL, and how Oracle is superior to the competitors in solving problems of this class.

Task 2.2.9.TSK.A: show the reader who was the last to borrow a book from the library.

Task 2.2.9.TSK.B: show the reader (or readers, if there are several) who has held the book the longest (only consider cases where the book is not returned).

Task 2.2.9.TSK.C: show which book (or books, if more than one) each reader took on their most recent visit to the library.

Task 2.2.9.TSK.D: show the latest book that each reader borrowed from the library.

Task 2.2.9.TSK.E: rewrite the solution to problem 2.2.9.c{140} for MySQL using Common Table Expressions.

Task 2.2.9.TSK.F: rewrite the solution to problem 2.2.9.c{140} for MS SQL Server using a more compact syntax for grouping string values from multiple rows (i.e., the `STRING_AGG` function).

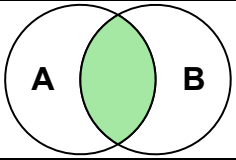
Task 2.2.9.TSK.G: rewrite the first option to solve problem 2.2.9.d{140} for MySQL using Common Table Expressions.

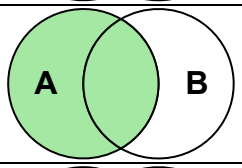
Task 2.2.9.TSK.H: write a third option of the solution of problem 2.2.9.d{140} for MySQL, using ranking and grouping (similar to how it is done in solutions for MS SQL Server and Oracle).

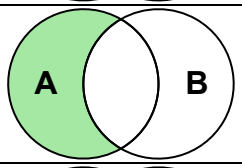
2.2.10. Example 20: Each and Every JOIN Variety in Three DBMSes

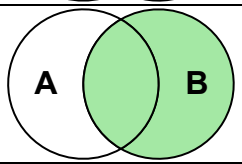
First, let's recap what results can be obtained by joining data from two tables using the classic versions of `JOIN`[4]:

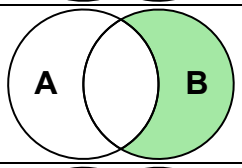
Type of join	Graphic representation	Query pseudocode

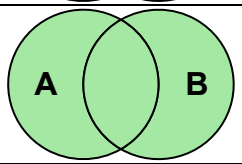
Inner join		`SELECT fields`
		`FROM A INNER JOIN B`
		`ON A.field = B.field`

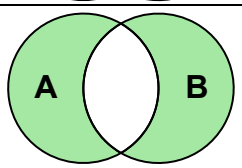
Left outer join		`SELECT fields`
		`FROM A LEFT OUTER JOIN B`
		`ON A.field = B.field`

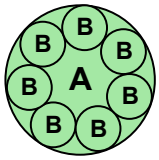
Exclusive left outer join		`SELECT fields`
		`FROM A LEFT OUTER JOIN B`
		`ON A.field = B.field`
		`WHERE B.field IS NULL`

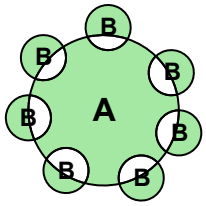
Right outer join		`SELECT fields`
		`FROM A RIGHT OUTER JOIN B`
		`ON A.field = B.field`

Exclusive right outer join		`SELECT fields`
		`FROM A RIGHT OUTER JOIN B`
		`ON A.field = B.field`
		`WHERE A.field IS NULL`

Full outer join		`SELECT fields`
		`FROM A FULL OUTER JOIN B`
		`ON A.field = B.field`

Exclusive full outer join		`SELECT fields`
		`FROM A FULL OUTER JOIN B`
		`ON A.field = B.field`
		`WHERE A.field IS NULL`
		`OR B.field IS NULL`

Cross join		`SELECT fields`
		`FROM A CROSS JOIN B`
		or
		`SELECT fields`
		`FROM A, B`

Exclusive cross join		`SELECT fields`
		`FROM A CROSS JOIN B`
		`WHERE A.field != B.field`
		or
		`SELECT fields`
		`FROM A, B`
		`WHERE A.field != B.field`

[4] The original image: http://www.codeproject.com/Articles/33052/Visual-Representation-of-SQL-Joins

To demonstrate specific examples, let's create in the "Exploration" database `rooms` and `computers` tables connected by "one to many" relationship:

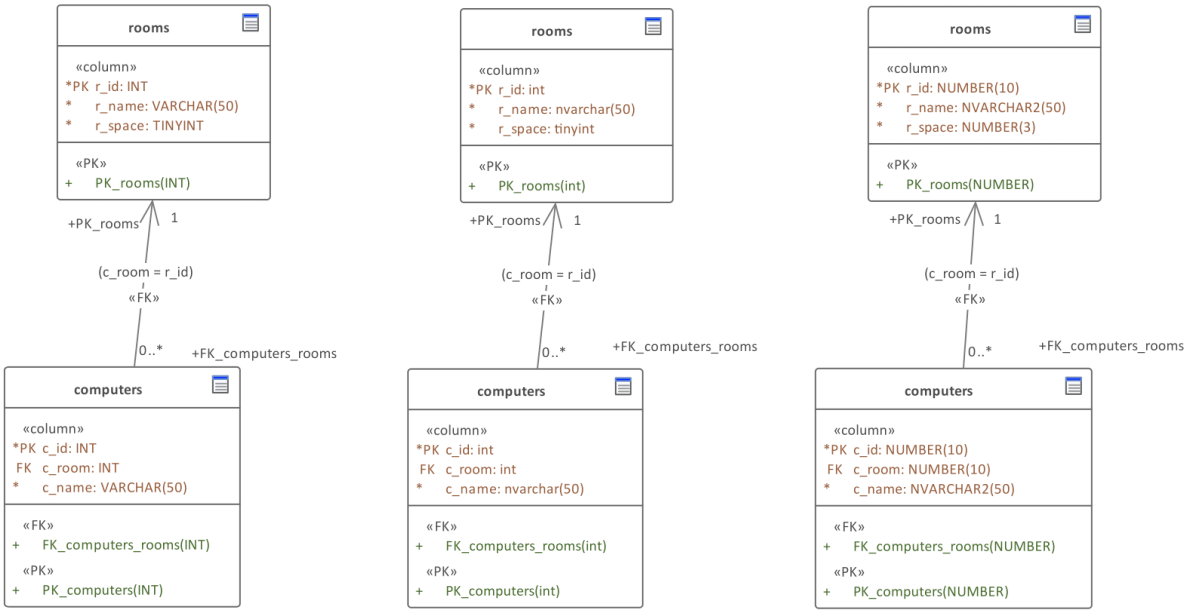


Figure 2.2.a — The `rooms` and `computers` tables in all three DBMSes

Let's put the following data into the `rooms` table:

r_id	r_name	r_space
1	A room with two computers	5
2	A room with three computers	5
3	An empty room 1	2
4	An empty room 2	2
5	An empty room 3	2

Let's put the following data into the `computers` table:

c_id	c_room	c_name
1	1	The computer A in the room 1
2	1	The computer B in the room 1
3	2	The computer A in the room 2
4	2	The computer B in the room 2
5	2	The computer C in the room 2
6	NULL	Unassigned computer A
7	NULL	Unassigned computer B
8	NULL	Unassigned computer C

We should note right away that the words `INNER` and `OUTER` in the vast majority of cases are so-called "syntactic sugar" (i.e., added for human convenience, while in no way affecting the logic of the query) and therefore are optional: "Just `JOIN`" is always *inner*, and `LEFT JOIN`, `RIGHT JOIN` and `FULL JOIN` are always *outer*.

There are so many problems in this example that it's easy to get confused, so we'll first list them all and then put the formulations, expected results, and solutions together.

Problems on “classic joins”:

- 2.2.10.a{162}: show information about how computers are assigned to rooms;
- 2.2.10.b{163}: show all the rooms with computers in them;
- 2.2.10.c{164}: show all empty rooms;
- 2.2.10.d{165}: show all the computers with information about which rooms they are located in;
- 2.2.10.e{166}: show all unassigned computers;
- 2.2.10.f{168}: show all the information about how computers are arranged by rooms (including empty rooms and unassigned computers);
- 2.2.10.g{170}: show information on all empty rooms and unassigned computers;
- 2.2.10.h{172}: show possible options for arranging the computers in the rooms (do not consider the capacity of the rooms);
- 2.2.10.i{174}: show possible options for rearranging computers by rooms (the computer should not be placed in the room in which it is currently located, do not consider the capacity of the rooms).

Problems on “non-classic joins”:

- 2.2.10.j{176}: show possible options for arranging the computers in the rooms (consider the capacity of the rooms);
- 2.2.10.k{179}: show possible options for arranging unassigned computers in empty rooms (do not consider the capacity of the rooms);
- 2.2.10.l{180}: show possible options for arranging unassigned computers in empty rooms (consider the capacity of the rooms);
- 2.2.10.m{184}: show possible options for arranging unassigned computers in the rooms (take into account the residual capacity of the rooms);
- 2.2.10.n{187}: show the arrangement of computers in non-empty rooms so that the result does not include more computers than can fit in the room;
- 2.2.10.o{191}: show the arrangement of computers in all rooms so that the result does not include more computers than can fit in the room.

Problems for “classic joins” involve solution based on direct use of the syntax provided by the DBMS (without subqueries and other tricks).

Problems for “non-classic joins” require a solution based on either DBMS-specific syntax or additional actions (as a rule, subqueries).

Problem 2.2.10.a: show information about how computers are assigned to rooms.

Expected result 2.2.10.a.

r_id	r_name	c_id	c_room	c_name
1	A room with two computers	1	1	The computer A in the room 1
1	A room with two computers	2	1	The computer B in the room 1
2	A room with three computers	3	2	The computer A in the room 2
2	A room with three computers	4	2	The computer B in the room 2
2	A room with three computers	5	2	The computer C in the room 2

Solution 2.2.10.a: we'll use the inner join.

MySQL Solution 2.2.10.a

```
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       JOIN `computers`
         ON `r_id` = `c_room`
```

MS SQL Solution 2.2.10.a

```
SELECT [r_id],
       [r_name],
       [c_id],
       [c_room],
       [c_name]
FROM   [rooms]
       JOIN [computers]
         ON [r_id] = [c_room]
```

Oracle Solution 2.2.10.a

```
SELECT "r_id",
       "r_name",
       "c_id",
       "c_room",
       "c_name"
FROM   "rooms"
       JOIN "computers"
         ON "r_id" = "c_room"
```

The logic of inner join is to pick pairs of records from two tables that have the same value of the fields that is being used for join operation. To make it clear, let's look at the expected result again and put the fields that are joined next to each other:

r_name	r_id	c_room	c_id	c_name
A room with two computers	1	1	1	The computer A in the room 1
A room with two computers	1	1	2	The computer B in the room 1
A room with three computers	2	2	3	The computer A in the room 2
A room with three computers	2	2	4	The computer B in the room 2
A room with three computers	2	2	5	The computer C in the room 2

Problem 2.2.10.b: show all the rooms with computers in them.

Expected result 2.2.10.b.

r_id	r_name	c_id	c_room	c_name
1	A room with two computers	1	1	The computer A in the room 1
1	A room with two computers	2	1	The computer B in the room 1
2	A room with three computers	3	2	The computer A in the room 2
2	A room with three computers	4	2	The computer B in the room 2
2	A room with three computers	5	2	The computer C in the room 2
3	An empty room 1	NULL	NULL	NULL
4	An empty room 2	NULL	NULL	NULL
5	An empty room 3	NULL	NULL	NULL

Solution 2.2.10.b: we'll use the left outer join. In this solution we need to show all records from the **`rooms`** table, both those for which there is a match in the **`computers`** table, and those for which there is no such match.

MySQL Solution 2.2.10.b

```
1    SELECT `r_id`,
2           `r_name`,
3           `c_id`,
4           `c_room`,
5           `c_name`
6    FROM   `rooms`
7           LEFT JOIN `computers`
8                  ON `r_id` = `c_room`
```

MS SQL Solution 2.2.10.b

```
1    SELECT [r_id],
2           [r_name],
3           [c_id],
4           [c_room],
5           [c_name]
6    FROM   [rooms]
7           LEFT JOIN [computers]
8                  ON [r_id] = [c_room]
```

Oracle Solution 2.2.10.b

```
1    SELECT "r_id",
2           "r_name",
3           "c_id",
4           "c_room",
5           "c_name"
6    FROM   "rooms"
7           LEFT JOIN "computers"
8                  ON "r_id" = "c_room"
```

In case of a left outer join, the DBMS fetches all the records from the left table and tries to find a pair for them from the right table. If no pair is found, the corresponding record part in the resulting table is filled with **`NULL`** values.

Let's modify the expected result so that this idea is more obvious:

r_name	r_id	c_room	c_id	c_name
A room with two computers	1	1	1	The computer A in the room 1
A room with two computers	1	1	2	The computer B in the room 1
A room with three computers	2	2	3	The computer A in the room 2
A room with three computers	2	2	4	The computer B in the room 2
A room with three computers	2	2	5	The computer C in the room 2
An empty room 1	3	NULL	NULL	NULL
An empty room 2	4	NULL	NULL	NULL
An empty room 3	5	NULL	NULL	NULL

Problem 2.2.10.c: show all empty rooms.

Expected result 2.2.10.c.

r_id	r_name	c_id	c_room	c_name
3	An empty room 1	NULL	NULL	NULL
4	An empty room 2	NULL	NULL	NULL
5	An empty room 3	NULL	NULL	NULL

Solution 2.2.10.c: we'll use the exclusive left outer join, i.e., we'll select only those records from the **`rooms`** table, for which there is no match in the **`computers`** table.

MySQL Solution 2.2.10.c

```
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       LEFT JOIN `computers`
              ON `r_id` = `c_room`
WHERE  `c_room` IS NULL
```

MS SQL Solution 2.2.10.c

```
SELECT [r_id],
       [r_name],
       [c_id],
       [c_room],
       [c_name]
FROM   [rooms]
       LEFT JOIN [computers]
              ON [r_id] = [c_room]
WHERE  [c_room] IS NULL
```

Oracle Solution 2.2.10.c

```
SELECT "r_id",
       "r_name",
       "c_id",
       "c_room",
       "c_name"
FROM   "rooms"
       LEFT JOIN "computers"
              ON "r_id" = "c_room"
WHERE  "c_room" IS NULL
```

Due to the condition in line 9 of each query only lines with `NULL` values in the `c_room` field get into the final result from a data set equivalent to the one obtained in the previous problem (2.2.10.b):

r_name	r_id	c_room	c_id	c_name
A room with two computers	1	1	1	The computer A in the room 1
A room with two computers	1	1	2	The computer B in the room 1
A room with three computers	2	2	3	The computer A in the room 2
A room with three computers	2	2	4	The computer B in the room 2
A room with three computers	2	2	5	The computer C in the room 2
An empty room 1	3	NULL	NULL	NULL
An empty room 2	4	NULL	NULL	NULL
An empty room 3	5	NULL	NULL	NULL

Problem 2.2.10.d: show all the computers with information about which rooms they are located in.

Expected result 2.2.10.d.

r_id	r_name	c_id	c_room	c_name
1	A room with two computers	1	1	The computer A in the room 1
1	A room with two computers	2	1	The computer B in the room 1
2	A room with three computers	3	2	The computer A in the room 2
2	A room with three computers	4	2	The computer B in the room 2
2	A room with three computers	5	2	The computer C in the room 2
NULL	NULL	6	NULL	Unassigned computer A
NULL	NULL	7	NULL	Unassigned computer B
NULL	NULL	8	NULL	Unassigned computer C

Solution 2.2.10.d: we'll use the right outer join. This problem is the inverse of problem 2.2.10.b{163}, here we need to show all records from the `computers` table, regardless of whether they have matches from the `rooms` table.

MySQL Solution 2.2.10.d

```
1    SELECT `r_id`,
2           `r_name`,
3           `c_id`,
4           `c_room`,
5           `c_name`
6    FROM   `rooms`
7           RIGHT JOIN `computers`
8                   ON `r_id` = `c_room`
```

MS SQL Solution 2.2.10.d

```
1    SELECT [r_id],
2           [r_name],
3           [c_id],
4           [c_room],
5           [c_name]
6    FROM   [rooms]
7           RIGHT JOIN [computers]
8                   ON [r_id] = [c_room]
```

Oracle Solution 2.2.10.d

```
SELECT "r_id",
       "r_name",
       "c_id",
       "c_room",
       "c_name"
FROM   "rooms"
       RIGHT JOIN "computers"
               ON "r_id" = "c_room"
```

In the case of right outer join, the DBMS fetches all the records from the right table and tries to find a pair for them from the left table. If no pair is found, the corresponding record part in the resulting table is filled with `NULL` values.

Let's modify the expected result so that this idea is more obvious:

r_name	r_id	c_room	c_id	c_name
A room with two computers	1	1	1	The computer A in the room 1
A room with two computers	1	1	2	The computer B in the room 1
A room with three computers	2	2	3	The computer A in the room 2
A room with three computers	2	2	4	The computer B in the room 2
A room with three computers	2	2	5	The computer C in the room 2
NULL	NULL	NULL	6	Unassigned computer A
NULL	NULL	NULL	7	Unassigned computer B
NULL	NULL	NULL	8	Unassigned computer C

Problem 2.2.10.e: show all unassigned computers.

Expected result 2.2.10.e.

r_id	r_name	c_id	c_room	c_name
NULL	NULL	6	NULL	Unassigned computer A
NULL	NULL	7	NULL	Unassigned computer B
NULL	NULL	8	NULL	Unassigned computer C

Solution 2.2.10.e: we'll use the exclusive right outer join. This problem is the inverse of problem 2.2.10.c{164}, here we select only those entries from the `computers` table for which there is no match in the `rooms` table.

MySQL Solution 2.2.10.e

```
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       RIGHT JOIN `computers`
               ON `r_id` = `c_room`
WHERE  `r_id` IS NULL
```

MS SQL Solution 2.2.10.e

```
SELECT [r_id],
       [r_name],
       [c_id],
       [c_room],
       [c_name]
FROM   [rooms]
       RIGHT JOIN [computers]
               ON [r_id] = [c_room]
WHERE  [r_id] IS NULL
```

Oracle Solution 2.2.10.e

```
SELECT "r_id",
       "r_name",
       "c_id",
       "c_room",
       "c_name"
FROM   "rooms"
       RIGHT JOIN "computers"
               ON "r_id" = "c_room"
WHERE  "r_id" IS NULL
```

The same result (as a rule, in such problems we are not interested in fields from parent table, because they by definition will be **NULL**) can be obtained without **JOIN**. This method works when the source of information is a child table, but problem 2.2.10.c{164} could not be solved in such a trivial way (there we needed to run a subquery with the **NOT IN** clause):

c_id	c_room	c_name
6	NULL	Unassigned computer A
7	NULL	Unassigned computer B
8	NULL	Unassigned computer C

MySQL Solution 2.2.10.e (simplified option)

```
SELECT `c_id`,
       `c_room`,
       `c_name`
FROM   `computers`
WHERE  `c_room` IS NULL
```

MS SQL Solution 2.2.10.e (simplified option)

```
SELECT [c_id],
       [c_room],
       [c_name]
FROM   [computers]
WHERE  [c_room] IS NULL
```

Oracle Solution 2.2.10.e (simplified option)

```
SELECT "c_id",
       "c_room",
       "c_name"
FROM   "computers"
WHERE  "c_room" IS NULL
```

Problem 2.2.10.f: show all the information about how computers are arranged by rooms (including empty rooms and unassigned computers).

Expected result 2.2.10.f.

r_id	r_name	c_id	c_room	c_name
1	A room with two computers	1	1	The computer A in the room 1
1	A room with two computers	2	1	The computer B in the room 1
2	A room with three computers	3	2	The computer A in the room 2
2	A room with three computers	4	2	The computer B in the room 2
2	A room with three computers	5	2	The computer C in the room 2
3	An empty room 1	NULL	NULL	NULL
4	An empty room 2	NULL	NULL	NULL
5	An empty room 3	NULL	NULL	NULL
NULL	NULL	6	NULL	Unassigned computer A
NULL	NULL	7	NULL	Unassigned computer B
NULL	NULL	8	NULL	Unassigned computer C

Solution 2.2.10.f: we'll use full outer join. This problem is a combination of problems 2.2.10.b{163} and 2.2.10.d{165}, here we need to show all records from the `rooms` table, regardless of whether there is a match in the `computers` table, and all records from the `computers` table, regardless of whether there is a match in the `rooms` table.

Important! MySQL does not support full outer join (yes, even in version 8), so the use of `FULL JOIN` there will not give the correct result.

MySQL — Solution 2.2.10.f (wrong query)

```
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       FULL JOIN `computers`
              ON `r_id` = `c_room`
```

This query results in the same data set as in the problem 2.2.10.a{162}:

r_id	r_name	c_id	c_room	c_name
1	A room with two computers	1	1	The computer A in the room 1
1	A room with two computers	2	1	The computer B in the room 1
2	A room with three computers	3	2	The computer A in the room 2
2	A room with three computers	4	2	The computer B in the room 2
2	A room with three computers	5	2	The computer C in the room 2

The easiest[5] solution of this problem in MySQL is to combine the solutions of problems 2.2.10.b{163} and 2.2.10.d{165} using the `UNION` construct.

[5] Several alternatives are discussed in this article: http://www.xaprb.com/blog/2006/05/26/how-to-write-full-outer-join-in-mysql/

MySQL Solution 2.2.10.f

```
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       LEFT JOIN `computers`
              ON `r_id` = `c_room`
UNION
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       RIGHT JOIN `computers`
               ON `r_id` = `c_room`
```

MS SQL Server and Oracle support full outer join, and this task is much easier there:

MS SQL Solution 2.2.10.f

```
SELECT [r_id],
       [r_name],
       [c_id],
       [c_room],
       [c_name]
FROM   [rooms]
       FULL JOIN [computers]
              ON [r_id] = [c_room]
```

Oracle Solution 2.2.10.f

```
SELECT "r_id",
       "r_name",
       "c_id",
       "c_room",
       "c_name"
FROM   "rooms"
       FULL JOIN "computers"
              ON "r_id" = "c_room"
```

When performing the full outer join, the DBMS retrieves **all** records from **both** tables and searches for pairs. Where pairs are found, the final result gets a row with data from both tables. Where there is no pair, the missing data is filled with `NULL` values. Let's show this graphically:

r_name	r_id	c_room	c_id	c_name
A room with two computers	1	1	1	The computer A in the room 1
A room with two computers	1	1	2	The computer B in the room 1
A room with three computers	2	2	3	The computer A in the room 2
A room with three computers	2	2	4	The computer B in the room 2
A room with three computers	2	2	5	The computer C in the room 2
An empty room 1	3	NULL	NULL	NULL
An empty room 2	4	NULL	NULL	NULL
An empty room 3	5	NULL	NULL	NULL
NULL	NULL	NULL	6	Unassigned computer A
NULL	NULL	NULL	7	Unassigned computer B
NULL	NULL	NULL	8	Unassigned computer C

Problem 2.2.10.g: show information on all empty rooms and unassigned computers.

Expected result 2.2.10.g.

r_id	r_name	c_id	c_room	c_name
3	An empty room 1	NULL	NULL	NULL
4	An empty room 2	NULL	NULL	NULL
5	An empty room 3	NULL	NULL	NULL
NULL	NULL	6	NULL	Unassigned computer A
NULL	NULL	7	NULL	Unassigned computer B
NULL	NULL	8	NULL	Unassigned computer C

Solution 2.2.10.g: we'll use exclusive full outer join. This problem is a combination of problems 2.2.10.c[164] and 2.2.10.e[166], here we need to show all records from the **`rooms`** table, for which there is no match in the **`computers`** table, and all records from the **`computers`** table, for which there is no match in the **`rooms`** table.

Here is the same issue with MySQL as in the previous problem, the lack of support for full outer join, forcing us again to use two separate queries, the results of which are combined using **`UNION`**:

MySQL Solution 2.2.10.g

```
1    SELECT `r_id`,
2           `r_name`,
3           `c_id`,
4           `c_room`,
5           `c_name`
6    FROM   `rooms`
7           LEFT JOIN `computers`
8                  ON `r_id` = `c_room`
9    WHERE  `c_id` IS NULL
10   UNION
11   SELECT `r_id`,
12          `r_name`,
13          `c_id`,
14          `c_room`,
15          `c_name`
16   FROM   `rooms`
17          RIGHT JOIN `computers`
18                  ON `r_id` = `c_room`
19   WHERE  `r_id` IS NULL
```

In MS SQL Server and Oracle everything is easier: it is enough to specify in which fields we expect **`NULL`**.

MS SQL Solution 2.2.10.g

```
1    SELECT [r_id],
2           [r_name],
3           [c_id],
4           [c_room],
5           [c_name]
6    FROM   [rooms]
7           FULL JOIN [computers]
8                  ON [r_id] = [c_room]
9    WHERE     [r_id] IS NULL
10          OR [c_id] IS NULL
```

Oracle | Solution 2.2.10.g

```
1     SELECT "r_id",
2            "r_name",
3            "c_id",
4            "c_room",
5            "c_name"
6     FROM   "rooms"
7            FULL JOIN "computers"
8                   ON "r_id" = "c_room"
9     WHERE       "r_id" IS NULL
10           OR "c_id" IS NULL
```

The conditions in lines 9-10 of MS SQL Server and Oracle queries do not allow in the final result of rows, other than having the **`NULL`** value in the fields **`r_id`** or **`c_id`**. These fields are not chosen by chance: they are the primary keys of their tables, and therefore the appearance of "natural" **`NULL`** values (initially placed in the table) is extremely unlikely in contrast to the "normal fields", where **`NULL`** may well be stored as a sign of "empty value".

Graphically, the set of records falling into the final result looks like this (marked with a gray background):

r_name	r_id	c_room	c_id	c_name
A room with two computers	1	1	1	The computer A in the room 1
A room with two computers	1	1	2	The computer B in the room 1
A room with three computers	2	2	3	The computer A in the room 2
A room with three computers	2	2	4	The computer B in the room 2
A room with three computers	2	2	5	The computer C in the room 2
An empty room 1	3	NULL	**NULL**	NULL
An empty room 2	4	NULL	**NULL**	NULL
An empty room 3	5	NULL	**NULL**	NULL
NULL	**NULL**	NULL	6	Unassigned computer A
NULL	**NULL**	NULL	7	Unassigned computer B
NULL	**NULL**	NULL	8	Unassigned computer C

Problem 2.2.10.h: show possible options for arranging the computers in the rooms (do not consider the capacity of the rooms).

Expected result 2.2.10.h.

r_id	r_name	c_id	c_room	c_name
1	A room with two computers	1	1	The computer A in the room 1
2	A room with three computers	1	1	The computer A in the room 1
3	An empty room 1	1	1	The computer A in the room 1
4	An empty room 2	1	1	The computer A in the room 1
5	An empty room 3	1	1	The computer A in the room 1
1	A room with two computers	2	1	The computer B in the room 1
2	A room with three computers	2	1	The computer B in the room 1
3	An empty room 1	2	1	The computer B in the room 1
4	An empty room 2	2	1	The computer B in the room 1
5	An empty room 3	2	1	The computer B in the room 1
1	A room with two computers	3	2	The computer A in the room 2
2	A room with three computers	3	2	The computer A in the room 2
3	An empty room 1	3	2	The computer A in the room 2
4	An empty room 2	3	2	The computer A in the room 2
5	An empty room 3	3	2	The computer A in the room 2
1	A room with two computers	4	2	The computer B in the room 2
2	A room with three computers	4	2	The computer B in the room 2
3	An empty room 1	4	2	The computer B in the room 2
4	An empty room 2	4	2	The computer B in the room 2
5	An empty room 3	4	2	The computer B in the room 2
1	A room with two computers	5	2	The computer C in the room 2
2	A room with three computers	5	2	The computer C in the room 2
3	An empty room 1	5	2	The computer C in the room 2
4	An empty room 2	5	2	The computer C in the room 2
5	An empty room 3	5	2	The computer C in the room 2
1	A room with two computers	6	NULL	Unassigned computer A
2	A room with three computers	6	NULL	Unassigned computer A
3	An empty room 1	6	NULL	Unassigned computer A
4	An empty room 2	6	NULL	Unassigned computer A
5	An empty room 3	6	NULL	Unassigned computer A
1	A room with two computers	7	NULL	Unassigned computer B
2	A room with three computers	7	NULL	Unassigned computer B
3	An empty room 1	7	NULL	Unassigned computer B
4	An empty room 2	7	NULL	Unassigned computer B
5	An empty room 3	7	NULL	Unassigned computer B
1	A room with two computers	8	NULL	Unassigned computer C
2	A room with three computers	8	NULL	Unassigned computer C
3	An empty room 1	8	NULL	Unassigned computer C
4	An empty room 2	8	NULL	Unassigned computer C
5	An empty room 3	8	NULL	Unassigned computer C

Solution 2.2.10.h: we'll use the cross join (i.e., the Cartesian product).

MySQL Solution 2.2.10.h

```
1    -- Option 1: without JOIN keyword
2    SELECT `r_id`,
3           `r_name`,
4           `c_id`,
5           `c_room`,
6           `c_name`
7    FROM   `rooms`,
8           `computers`

1    -- Option 2: with JOIN keyword
2    SELECT `r_id`,
3           `r_name`,
4           `c_id`,
5           `c_room`,
6           `c_name`
7    FROM   `rooms`
8           CROSS JOIN `computers`
```

MS SQL Solution 2.2.10.h

```
1    -- Option 1: without JOIN keyword
2    SELECT [r_id],
3           [r_name],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   [rooms],
8           [computers]

1    -- Option 2: with JOIN keyword
2    SELECT [r_id],
3           [r_name],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   [rooms]
8           CROSS JOIN [computers]
```

Oracle Solution 2.2.10.h

```
1    -- Option 1: without JOIN keyword
2    SELECT "r_id",
3           "r_name",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms",
8           "computers"

1    -- Option 2: with JOIN keyword
2    SELECT "r_id",
3           "r_name",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms"
8           CROSS JOIN "computers"
```

When performing the cross join (producing the Cartesian product), the DBMS matches every record in the left table with every record in the right table. In other words, the DBMS finds all possible pairwise combinations of records from both tables.

Problem 2.2.10.i: show possible options for rearranging computers by rooms (the computer should not be placed in the room in which it is currently located, do not consider the capacity of the rooms).

Expected result 2.2.10.i.

r_id	r_name	c_id	c_room	c_name
2	A room with three computers	1	1	The computer A in the room 1
3	An empty room 1	1	1	The computer A in the room 1
4	An empty room 2	1	1	The computer A in the room 1
5	An empty room 3	1	1	The computer A in the room 1
2	A room with three computers	2	1	The computer B in the room 1
3	An empty room 1	2	1	The computer B in the room 1
4	An empty room 2	2	1	The computer B in the room 1
5	An empty room 3	2	1	The computer B in the room 1
1	A room with two computers	3	2	The computer A in the room 2
3	An empty room 1	3	2	The computer A in the room 2
4	An empty room 2	3	2	The computer A in the room 2
5	An empty room 3	3	2	The computer A in the room 2
1	A room with two computers	4	2	The computer B in the room 2
3	An empty room 1	4	2	The computer B in the room 2
4	An empty room 2	4	2	The computer B in the room 2
5	An empty room 3	4	2	The computer B in the room 2
1	A room with two computers	5	2	The computer C in the room 2
3	An empty room 1	5	2	The computer C in the room 2
4	An empty room 2	5	2	The computer C in the room 2
5	An empty room 3	5	2	The computer C in the room 2

Solution 2.2.10.i: we'll use the exclusive cross join.

MySQL Solution 2.2.10.i

```
1    -- Option 1: without JOIN keyword
2    SELECT `r_id`,
3           `r_name`,
4           `c_id`,
5           `c_room`,
6           `c_name`
7    FROM   `rooms`,
8           `computers`
9    WHERE  `r_id` != `c_room`

1    -- Option 2: with JOIN keyword
2    SELECT `r_id`,
3           `r_name`,
4           `c_id`,
5           `c_room`,
6           `c_name`
7    FROM   `rooms`
8          CROSS JOIN `computers`
9    WHERE  `r_id` != `c_room`
```

MS SQL Solution 2.2.10.i

```
1    -- Option 1: without JOIN keyword
2    SELECT [r_id],
3           [r_name],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   [rooms],
8           [computers]
9    WHERE  [r_id] != [c_room]

1    -- Option 2: with JOIN keyword
2    SELECT [r_id],
3           [r_name],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   [rooms]
8           CROSS JOIN [computers]
9    WHERE  [r_id] != [c_room]
```

Oracle Solution 2.2.10.i

```
1    -- Option 1: without JOIN keyword
2    SELECT "r_id",
3           "r_name",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms",
8           "computers"
9    WHERE  "r_id" != "c_room"

1    -- Option 2: with JOIN keyword
2    SELECT "r_id",
3           "r_name",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms"
8           CROSS JOIN "computers"
9    WHERE  "r_id" != "c_room"
```

When producing the exclusive Cartesian product (exclusive cross join), the DBMS does not allow the result to contain the real pairs of records from both tables, i.e., it gets all possible pairwise combinations except for those that actually exist.

That's all for the classic joins.

Problems for “non-classic joins” assume solution on the basis of either DBMS-specific syntax or additional actions (as a rule, subqueries).

Many tasks in this subsection owe their emergence to the existence in MS SQL Server and Oracle (since version 12c) of such operators as **`CROSS APPLY`** and **`OUTER APPLY`**. Therefore, hereinafter, the solution for MS SQL Server will be primary, and the solutions for MySQL and Oracle will be built through emulation of the corresponding behavior.

Problem 2.2.10.j: show possible options for arranging the computers in the rooms (consider the capacity of the rooms).

Expected result 2.2.10.j.

r_id	r_name	r_space	c_id	c_room	c_name
1	A room with two computers	5	1	1	The computer A in the room 1
1	A room with two computers	5	2	1	The computer B in the room 1
1	A room with two computers	5	3	2	The computer A in the room 2
1	A room with two computers	5	4	2	The computer B in the room 2
1	A room with two computers	5	5	2	The computer C in the room 2
2	A room with three computers	5	1	1	The computer A in the room 1
2	A room with three computers	5	2	1	The computer B in the room 1
2	A room with three computers	5	3	2	The computer A in the room 2
2	A room with three computers	5	4	2	The computer B in the room 2
2	A room with three computers	5	5	2	The computer C in the room 2
3	An empty room 1	2	1	1	The computer A in the room 1
3	An empty room 1	2	3	2	The computer A in the room 2
4	An empty room 2	2	1	1	The computer A in the room 1
4	An empty room 2	2	3	2	The computer A in the room 2
5	An empty room 3	2	1	1	The computer A in the room 1
5	An empty room 3	2	3	2	The computer A in the room 2

Note that no room was assigned more computers than the value in the `r_space` field.

Solution 2.2.10.j: we'll use `CROSS APPLY` in MS SQL Server and emulate the same behavior in MySQL as well as in Oracle up to version 12c (for version 12c and newer we'll also use `CROSS APPLY`).

MySQL | Solution 2.2.10.j

```
SELECT `r_id`,
       `r_name`,
       `r_space`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       CROSS JOIN (SELECT `c_id`,
                          `c_room`,
                          `c_name`,
                          @row_num := @row_num + 1 AS `position`
                   FROM   `computers`,
                          (SELECT @row_num := 0) AS `x`
                   ORDER  BY `c_name` ASC) AS `cross_apply_data`
WHERE  `position` <= `r_space`
ORDER  BY `r_id`,
          `c_id`
```

The subquery on lines 8-14 returns a numbered list of computers:

c_id	c_room	c_name	position
1	1	The computer A in the room 1	1
3	2	The computer A in the room 2	2
2	1	The computer B in the room 1	3
4	2	The computer B in the room 2	4
5	2	The computer C in the room 2	5
6	NULL	Unassigned computer A	6
7	NULL	Unassigned computer B	7
8	NULL	Unassigned computer C	8

The condition in line 15 allows us to exclude from the final result all computers with numbers exceeding the room capacity. Thus, we get the final result.

MS SQL Solution 2.2.10.j

```
1     SELECT [r_id],
2            [r_name],
3            [r_space],
4            [c_id],
5            [c_room],
6            [c_name]
7     FROM   [rooms]
8            CROSS APPLY (SELECT TOP ([r_space])
9                                   [c_id],
10                                  [c_room],
11                                  [c_name]
12                           FROM   [computers]
13                           ORDER  BY [c_name] ASC) AS [cross_apply_data]
14    ORDER  BY [r_id],
15              [c_id]
```

In MS SQL Server, the `CROSS APPLY` operator allows us to get access data "provided by the left part of the query" from the "right part of the query" without any additional actions. Due to this, the `SELECT TOP ([r_space]) ...` clause leads to selecting a number of records from the `computer` table not exceeding the value of `r_space` field in the corresponding row of the `rooms` table. Let's explain it graphically:

r_id	r_name	r_space	What is substituted in the TOP x
1	A room with two computers	5	`SELECT TOP 5 ...`
2	A room with three computers	5	`SELECT TOP 5 ...`
3	An empty room 1	2	`SELECT TOP 2 ...`
4	An empty room 2	2	`SELECT TOP 2 ...`
5	An empty room 3	2	`SELECT TOP 2 ...`

Due to this behavior, each row from the `rooms` table is substituted with a certain number of records from the `computers` table, and so the final result is obtained.

Solution for Oracle (before version 12c) is equivalent to the solution for MySQL and differs only in the way of numbering computers: here we can use the ready-made function `ROW_NUMBER`.

Oracle Solution 2.2.10.j (before 12c)

```
1     SELECT "r_id",
2            "r_name",
3            "r_space",
4            "c_id",
5            "c_room",
6            "c_name"
7     FROM   "rooms"
8            CROSS JOIN (SELECT "c_id",
9                               "c_room",
10                              "c_name",
11                              ROW_NUMBER() OVER (ORDER BY "c_name" ASC)
12                              AS "position"
13                       FROM   "computers"
14                       ORDER  BY "c_name" ASC) "cross_apply_data"
15    WHERE  "position" <= "r_space"
16    ORDER  BY "r_id",
17              "c_id"
```

Since version 12c Oracle already supports **`CROSS APPLY`** operator, which allows us to write a solution different from MS SQL Server only by the way to extract the necessary number of rows in subquery (in MS SQL Server we do it with **`TOP ([r_space])`** syntax (see line 8 of query) and in Oracle we do it with **`FETCH NEXT "r_space" ROWS ONLY`** syntax (see line 14 of query)).

Oracle Solution 2.2.10.j (12c and newer)

```
1     SELECT "r_id",
2            "r_name",
3            "r_space",
4            "c_id",
5            "c_room",
6            "c_name"
7     FROM   "rooms"
8            CROSS APPLY (SELECT "c_id",
9                                "c_room",
10                               "c_name"
11                       FROM   "computers"
12                       ORDER  BY "c_name" ASC
13                       FETCH NEXT "r_space" ROWS ONLY) "cross_apply_data"
14    ORDER  BY "r_id",
15              "c_id"
```

Problem 2.2.10.k: show possible options for arranging unassigned computers in empty rooms (do not consider the capacity of the rooms).

Expected result 2.2.10.k.

r_id	r_name	c_id	c_room	c_name
3	An empty room 1	6	NULL	Unassigned computer A
3	An empty room 1	7	NULL	Unassigned computer B
3	An empty room 1	8	NULL	Unassigned computer C
4	An empty room 2	6	NULL	Unassigned computer A
4	An empty room 2	7	NULL	Unassigned computer B
4	An empty room 2	8	NULL	Unassigned computer C
5	An empty room 3	6	NULL	Unassigned computer A
5	An empty room 3	7	NULL	Unassigned computer B
5	An empty room 3	8	NULL	Unassigned computer C

Solution 2.2.10.k: we'll use cross join with some prior preparation.

The only difficulty of this task is getting a list of empty rooms (since we elementarily determine unassigned computers by the **`NULL`** value in the **`c_room`** field). This task is also perfect to demonstrate one typical mistake.

MySQL Solution 2.2.10.k

```
SELECT `r_id`,
       `r_name`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   (SELECT `r_id`,
               `r_name`
        FROM   `rooms`
        WHERE  `r_id` NOT IN (SELECT DISTINCT `c_room`
                              FROM   `computers`
                              WHERE  `c_room` IS NOT NULL))
       AS `empty_rooms`
       CROSS JOIN `computers`
WHERE  `c_room` IS NULL
```

MS SQL Solution 2.2.10.k

```
SELECT [r_id],
       [r_name],
       [c_id],
       [c_room],
       [c_name]
FROM   (SELECT [r_id],
               [r_name]
        FROM   [rooms]
        WHERE  [r_id] NOT IN (SELECT DISTINCT [c_room]
                              FROM   [computers]
                              WHERE  [c_room] IS NOT NULL))
       AS [empty_rooms]
       CROSS JOIN [computers]
WHERE  [c_room] IS NULL
```

Oracle | Solution 2.2.10.k

```
1    SELECT "r_id",
2           "r_name",
3           "c_id",
4           "c_room",
5           "c_name"
6    FROM   (SELECT "r_id",
7                   "r_name"
8            FROM   "rooms"
9            WHERE  "r_id" NOT IN (SELECT DISTINCT "c_room"
10                                 FROM   "computers"
11                                 WHERE  "c_room" IS NOT NULL))
12          "empty_rooms"
13          CROSS JOIN "computers"
14   WHERE  "c_room" IS NULL
```

In this solution, line 14 in all three queries is responsible for getting only unassigned computers. The free rooms are determined by the subquery on lines 6-12 (it returns a list of rooms whose identifiers do not occur in the `computers` table):

r_id	r_name
3	An empty room 1
4	An empty room 2
5	An empty room 3

A very common typical mistake is the absence of `WHERE c_room IS NOT NULL` condition in the internal subquery on lines 9-11. Because of that, its results contain `NULL` value, which, when processed, returns `FALSE` for any value of `r_id`, and as a result subquery on lines 6-12 returns empty result. Combining with an empty result also yields an empty result. So, because of one unobvious condition the whole query stops returning any data at all.

This is exactly how `NOT IN` behaves. Simply `IN` works as expected, i.e., it returns `TRUE` for values included in the analyzed set and `FALSE` for values not included.

Problem 2.2.10.l: show possible options for arranging unassigned computers in empty rooms (consider the capacity of the rooms).

Expected result 2.2.10.l.

r_id	r_name	r_space	c_id	c_room	c_name
3	An empty room 1	2	6	NULL	Unassigned computer A
3	An empty room 1	2	7	NULL	Unassigned computer B
4	An empty room 2	2	6	NULL	Unassigned computer A
4	An empty room 2	2	7	NULL	Unassigned computer B
5	An empty room 3	2	6	NULL	Unassigned computer A
5	An empty room 3	2	7	NULL	Unassigned computer B

Solution 2.2.10.l: we'll use `CROSS APPLY` in MS SQL Server and emulate the same behavior in MySQL as well as in Oracle before version 12c (for version 12c and newer we'll also use `CROSS APPLY`).

The solution of this problem is a combination of the solutions of problems 2.2.10.j{176} and 2.2.10.k{179}; we will take the logic of `CROSS APPLY` from the first, and the logic of getting a list of empty rooms from the second.

MySQL Solution 2.2.10.l

```
1    SELECT `r_id`,
2           `r_name`,
3           `r_space`,
4           `c_id`,
5           `c_room`,
6           `c_name`
7    FROM   (SELECT `r_id`,
8                  `r_name`,
9                  `r_space`
10           FROM   `rooms`
11           WHERE  `r_id` NOT IN (SELECT `c_room`
12                                 FROM   `computers`
13                                 WHERE  `c_room` IS NOT NULL))
14          AS `empty_rooms`
15          CROSS JOIN (SELECT `c_id`,
16                             `c_room`,
17                             `c_name`,
18                             @row_num := @row_num + 1 AS `position`
19                      FROM   `computers`,
20                             (SELECT @row_num := 0) AS `x`
21                      WHERE  `c_room` IS NULL
22                      ORDER  BY `c_name` ASC)
23                      AS `cross_apply_data`
24   WHERE  `position` <= `r_space`
25   ORDER  BY `r_id`,
26             `c_id`
```

The subquery on lines 7-14 returns a list of empty rooms:

r_id	r_name	r_space
3	An empty room 1	2
4	An empty room 2	2
5	An empty room 3	2

The subquery on lines 15-23 returns a numbered list of unassigned computers:

c_id	c_room	c_name	position
6	NULL	Unassigned computer A	1
7	NULL	Unassigned computer B	2
8	NULL	Unassigned computer C	3

`CROSS JOIN` of these two results gives the following Cartesian product (the `position` field has been added for clarity:

r_id	r_name	r_space	c_id	c_room	c_name	position
3	An empty room 1	2	6	NULL	Unassigned computer A	1
3	An empty room 1	2	7	NULL	Unassigned computer B	2
3	An empty room 1	2	8	NULL	Unassigned computer C	3
4	An empty room 2	2	6	NULL	Unassigned computer A	1
4	An empty room 2	2	7	NULL	Unassigned computer B	2
4	An empty room 2	2	8	NULL	Unassigned computer C	3
5	An empty room 3	2	6	NULL	Unassigned computer A	1
5	An empty room 3	2	7	NULL	Unassigned computer B	2
5	An empty room 3	2	8	NULL	Unassigned computer C	3

The condition in line 24 does not allow to select records (marked with a gray background), in which the value of the `position` field is greater than the value of the `r_space` field. Thus, we get the final result.

In MS SQL Server everything is much simpler again.

MS SQL Solution 2.2.10.l

```
1    SELECT [r_id],
2           [r_name],
3           [r_space],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   (SELECT [r_id],
8                  [r_name],
9                  [r_space]
10          FROM   [rooms]
11          WHERE  [r_id] NOT IN (SELECT [c_room]
12                                FROM   [computers]
13                                WHERE  [c_room] IS NOT NULL))
14         AS [empty_rooms]
15         CROSS APPLY (SELECT TOP ([r_space]) [c_id],
16                                             [c_room],
17                                             [c_name]
18                      FROM   [computers]
19                      WHERE  [c_room] IS NULL
20                      ORDER  BY [c_name] ASC)
21                     AS [cross_apply_data]
22   ORDER  BY [r_id],
23             [c_id]
```

The subquery on lines 7-14 returns a list of empty rooms:

r_id	r_name	r_space
3	An empty room 1	2
4	An empty room 2	2
5	An empty room 3	2

Then, thanks to `CROSS APPLY`, the value of the `r_space` field is used as the `TOP` argument:

r_id	r_name	r_space	Substituted to TOP x
3	An empty room 1	2	`SELECT TOP 2 ...`
4	An empty room 2	2	`SELECT TOP 2 ...`
5	An empty room 3	2	`SELECT TOP 2 ...`

That's how the final result comes out.

Solution for Oracle (before version 12c) is equivalent to the solution for MySQL and differs only in the way of numbering computers: here we can use the ready-made function **`ROW_NUMBER`**.

Oracle Solution 2.2.10.l (before 12c)

```
SELECT "r_id",
       "r_name",
       "r_space",
       "c_id",
       "c_room",
       "c_name"
FROM   (SELECT "r_id",
               "r_name",
               "r_space"
        FROM   "rooms"
        WHERE  "r_id" NOT IN (SELECT "c_room"
                              FROM   "computers"
                              WHERE  "c_room" IS NOT NULL))
       "empty_rooms"
       CROSS JOIN (SELECT "c_id",
                          "c_room",
                          "c_name",
                          ROW_NUMBER()
                            OVER (
                              ORDER BY "c_name" ASC) AS "position"
                   FROM   "computers"
                   WHERE  "c_room" IS NULL
                   ORDER  BY "c_name" ASC)
       "cross_apply_data"
WHERE  "position" <= "r_space"
ORDER  BY "r_id",
          "c_id"
```

Starting with version 12c, Oracle already supports **`CROSS APPLY`** statement, which allows us to write a solution different from MS SQL Server solution only by the way of extraction of necessary number of rows in subquery.

Oracle Solution 2.2.10.l (12c and newer)

```
SELECT "r_id",
       "r_name",
       "r_space",
       "c_id",
       "c_room",
       "c_name"
FROM   (SELECT "r_id",
               "r_name",
               "r_space"
        FROM   "rooms"
        WHERE  "r_id" NOT IN (SELECT "c_room"
                              FROM   "computers"
                              WHERE  "c_room" IS NOT NULL))
       "empty_rooms"
       CROSS APPLY (SELECT "c_id",
                           "c_room",
                           "c_name"
                    FROM   "computers"
                    WHERE  "c_room" IS NULL
                    ORDER  BY "c_name" ASC
                    FETCH NEXT "r_space" ROWS ONLY)
                   "cross_apply_data"
ORDER  BY "r_id",
          "c_id"
```

Problem 2.2.10.m: show possible options for arranging unassigned computers in the rooms (take into account the residual capacity of the rooms).

Expected result 2.2.10.m.

r_id	r_name	r_space	r_space_left	c_id	c_name
1	A room with two computers	5	3	6	Unassigned computer A
1	A room with two computers	5	3	7	Unassigned computer B
1	A room with two computers	5	3	8	Unassigned computer C
2	A room with three computers	5	2	6	Unassigned computer A
2	A room with three computers	5	2	7	Unassigned computer B
3	An empty room 1	2	2	6	Unassigned computer A
3	An empty room 1	2	2	7	Unassigned computer B
4	An empty room 2	2	2	6	Unassigned computer A
4	An empty room 2	2	2	7	Unassigned computer B
5	An empty room 3	2	2	6	Unassigned computer A
5	An empty room 3	2	2	7	Unassigned computer B

Solution 2.2.10.m: we'll use `CROSS APPLY` in MS SQL Server and emulate the same behavior in MySQL as well as in Oracle before version 12c (for version 12c and newer we'll also use `CROSS APPLY`).

This problem is similar to problem 2.2.10.j{176} with the exception that here we do not consider the total room capacity, but the residual capacity, i.e., the difference between the room capacity and the number of computers already located in it.

MySQL | Solution 2.2.10.m

```
1    SELECT `r_id`,
2           `r_name`,
3           `r_space`,
4           ( `r_space` - IFNULL(`r_used`, 0) ) AS `r_space_left`,
5           `c_id`,
6           `c_name`
7    FROM   `rooms`
8           LEFT JOIN (SELECT `c_room`         AS `c_room_inner`,
9                             COUNT(`c_room`) AS `r_used`
10                    FROM   `computers`
11                    GROUP  BY `c_room`) AS `computers_in_room`
12                ON `r_id` = `c_room_inner`
13          CROSS JOIN (SELECT `c_id`,
14                             `c_room`,
15                             `c_name`,
16                             @row_num := @row_num + 1 AS `position`
17                      FROM   `computers`,
18                             (SELECT @row_num := 0) AS `x`
19                      WHERE  `c_room` IS NULL
20                      ORDER  BY `c_name` ASC) AS `cross_apply_data`
21   WHERE `position` <= (`r_space` - IFNULL(`r_used`, 0))
22   ORDER  BY `r_id`,
23             `c_id`
```

The subquery on lines 13-20 returns a numbered list of unassigned computers:

c_id	c_room	c_name	position
6	NULL	Unassigned computer A	1
7	NULL	Unassigned computer B	2
8	NULL	Unassigned computer C	3

The subquery on lines 8-11 returns information about the number of computers in each room:

c_room_inner	r_used
NULL	0
1	2
2	3

After executing `LEFT JOIN` on line 8, the information about the number of computers in the room is merged with the list of rooms:

r_id	r_name	r_space	r_used
1	A room with two computers	5	2
2	A room with three computers	5	3
3	An empty room 1	2	NULL
4	An empty room 2	2	NULL
5	An empty room 3	2	NULL

Lines 4 and 21 use the information about the number of computers in the room to calculate the remaining number of empty slots (this is how the value of the `r_space_left` field is obtained). Note the need to use the `IFNULL` function to convert the `NULL` value of `r_used` field for empty rooms to 0.

The condition in line 21 does not allow to select unassigned computers with a sequence number greater than the number of free slots left in the room. This is how we get the final result.

In MS SQL Server the solution using `CROSS APPLY` is simpler and more compact.

MS SQL Solution 2.2.10.m

```
1    SELECT [r_id],
2           [r_name],
3           [r_space],
4           [r_space_left],
5           [c_id],
6           [c_name]
7    FROM   [rooms]
8           CROSS APPLY (SELECT TOP ([r_space] - (SELECT COUNT([c_room]) FROM
9                                    [computers] WHERE [c_room] =  [r_id]))
10                                  [c_id],
11                                  ( [r_space] - (SELECT COUNT([c_room])
12                                                 FROM   [computers]
13                                                 WHERE  [c_room] = [r_id]) )
14                                  AS [r_space_left],
15                                  [c_room],
16                                  [c_name]
17                          FROM   [computers]
18                          WHERE  [c_room] IS NULL
19                          ORDER  BY [c_name] ASC) AS [cross_apply_data]
20   ORDER BY [r_id],
21            [c_id]
```

Thanks to access to the fields of the analyzed record from the left table, subqueries in rows 8-9 and 11-14 can immediately get all the necessary information, and as a result, the subquery in rows 8-19 returns for each record from the “left part of the CROSS APPLY” no more unassigned computers than the number of free slots left in the corresponding room:

r_id	r_name	r_space	TOP x ...	r_space_left
1	A room with two computers	5	TOP 2	2
2	A room with three computers	5	TOP 3	3
3	An empty room 1	2	TOP 2	2
4	An empty room 2	2	TOP 2	2
5	An empty room 3	2	TOP 2	2

Solution for Oracle before version 12c is equivalent to the solution for MySQL except using the `NVL` function instead of the `IFNULL` function and the way to number unassigned computers using the `ROW_NUMBER` function instead of using an incrementable variable.

Oracle Solution 2.2.10.m (before 12c)

```
1    SELECT "r_id",
2           "r_name",
3           "r_space",
4           ( "r_space" - NVL("r_used", 0) ) AS "r_space_left",
5           "c_id",
6           "c_name"
7    FROM   "rooms"
8           LEFT JOIN (SELECT "c_room"         AS "c_room_inner",
9                             COUNT("c_room") AS "r_used"
10                    FROM   "computers"
11                    GROUP  BY "c_room") "computers_in_room"
12                ON "r_id" = "c_room_inner"
13          CROSS JOIN (SELECT "c_id",
14                             "c_room",
15                             "c_name",
16                             ROW_NUMBER() OVER (ORDER BY "c_name" ASC)
17                             AS "position"
18                      FROM   "computers" WHERE "c_room" IS NULL
19                      ORDER  BY "c_name" ASC) "cross_apply_data"
20   WHERE "position" <= ("r_space" - NVL("r_used", 0))
21   ORDER  BY "r_id",
22             "c_id"
```

In Oracle 12c (and newer versions) we can already use a solution similar to MS SQL Server.

Oracle Solution 2.2.10.m (12c and newer)

```
1    SELECT "r_id",
2           "r_name",
3           "r_space",
4           "r_space_left",
5           "c_id",
6           "c_name"
7    FROM   "rooms"
8           CROSS APPLY (SELECT ( "r_space" - (SELECT COUNT("c_room")
9                                              FROM   "computers"
10                                             WHERE  "c_room" = "r_id") )
11                               AS "r_space_left",
12                               "c_id",
13                               "c_room",
14                               "c_name"
15                       FROM   "computers"
16                       WHERE  "c_room" IS NULL
17                       ORDER  BY "c_name" ASC
18                  FETCH NEXT ("r_space" - (SELECT COUNT("c_room") FROM
19                                 "computers" WHERE "c_room" =  "r_id")) ROWS ONLY)
20                                 "cross_apply_data"
21   ORDER BY "r_id",
22             "c_id"
```

Problem 2.2.10.n: show the arrangement of computers in non-empty rooms so that the result does not include more computers than can fit in the room.

Expected result 2.2.10.n.

r_id	r_name	r_space	c_id	c_room	c_name
1	A room with two computers	5	1	1	The computer A in the room 1
1	A room with two computers	5	2	1	The computer B in the room 1
2	A room with three computers	5	3	2	The computer A in the room 2
2	A room with three computers	5	4	2	The computer B in the room 2
2	A room with three computers	5	5	2	The computer C in the room 2

Solution 2.2.10.n: we'll use `CROSS APPLY` in MS SQL Server and emulate the same behavior in MySQL and also in Oracle before version 12c (for version 12c and newer we'll also use `OUTER APPLY`).

This and the following (2.2.10.o[191]) problems are the most classical cases of using `CROSS APPLY` and `OUTER APPLY`: the tables are combined according to some condition, and an additional condition is taken into account, the data for which are taken from the left table.

In our case, the join condition is a match of the `r_id` and `c_room` fields, and the additional condition limiting selection from the right table is the room capacity represented in the `r_space` field.

Note that the `CROSS APPLY` emulation for MySQL and Oracle before version 12c in this case uses an inner join (`JOIN`), not a Cartesian product (`CROSS JOIN`).

Solution for MySQL turns out to be somewhat cumbersome in syntax, but very simple in essence.

MySQL Solution 2.2.10.n

```
SELECT `r_id`,
       `r_name`,
       `r_space`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       JOIN (SELECT `c_id`,
                    `c_room`,
                    `c_name`,
                    @row_num := IF(@prev_value = `c_room`, @row_num + 1, 1)
                    AS `position`,
                    @prev_value := `c_room`
             FROM   `computers`,
                    (SELECT @row_num := 1) AS `x`,
                    (SELECT @prev_value := '') AS `y`
             ORDER  BY `c_room`,
                       `c_name` ASC) AS `cross_apply_data`
         ON `r_id` = `c_room`
WHERE  `position` <= `r_space`
ORDER  BY `r_id`,
          `c_id`
```

The subquery on lines 8-18 returns a list of all computers with their numbering in the context of the room.

c_id	c_room	c_name	position
6	NULL	Unassigned computer A	1
7	NULL	Unassigned computer B	1
8	NULL	Unassigned computer C	1
1	1	The computer A in the room 1	1
2	1	The computer B in the room 1	2
3	2	The computer A in the room 2	1
4	2	The computer B in the room 2	2
5	2	The computer C in the room 2	3

In problems 2.2.10.j{176}, 2.2.10.l{180} and 2.210.m{184} we numbered the computers continuously without considering their room arrangement. The Cartesian product (`CROSS JOIN`) needs just such a continuous number because `CROSS JOIN` handles **all** records from the right table. But `JOIN` needs just the opposite, i.e., the computer number in the context of the room it is located in, because `JOIN` will look for a correspondence between the rooms and the computers in them.

The result of `JOIN` (line 8) looks like this:

r_id	r_name	r_space	c_id	c_room	c_name	position
1	A room with two computers	5	1	1	The computer A in the room 1	1
1	A room with two computers	5	2	1	The computer B in the room 1	2
2	A room with three computers	5	3	2	The computer A in the room 2	1
2	A room with three computers	5	4	2	The computer B in the room 2	2
2	A room with three computers	5	5	2	The computer C in the room 2	3

The `WHERE `position` <= `r_space`` condition in line 20 ensures that no computer whose sequence number (in the context of the room in which it is located) is greater than the room capacity will be selected. This is how the final result is obtained.

MS SQL Solution 2.2.10.n

```
1    SELECT [r_id],
2           [r_name],
3           [r_space],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   [rooms]
8           CROSS APPLY (SELECT TOP ([r_space])
9                               [c_id],
10                              [c_room],
11                              [c_name]
12                      FROM   [computers]
13                      WHERE [c_room] = [r_id]
14                      ORDER  BY [c_name] ASC) AS [cross_apply_data]
15   ORDER BY [r_id],
16            [c_id]
```

In MS SQL Server the join condition is specified in line 13 (`WHERE [c_room] = [r_id]`), and the additional condition (select no more computers than the room can handle) is specified in line 8 (`SELECT TOP ([r_space]`).

Thus, the right part of `CROSS APPLY` receives in one step a completely ready data set, for each room we get a list of its computers, with no more positions than the capacity of the room:

c_id	c_room	c_name	
1	1	The computer A in the room 1	Data set for the room 1
2	1	The computer B in the room 1	
3	2	The computer A in the room 2	Data set for the room 2
4	2	The computer B in the room 2	
5	2	The computer C in the room 2	

It only remains to add to each line of this set the information about the corresponding room (which is what `CROSS APPLY` does), and thus the final result is obtained.

Solution for Oracle before version 12c is similar to the solution for MySQL, except for one feature of computer numbering.

Oracle | Solution 2.2.10.n (before 12c)

```
SELECT "r_id",
       "r_name",
       "r_space",
       "c_id",
       "c_room",
       "c_name"
FROM   "rooms"
       JOIN (SELECT "c_id",
                    "c_room",
                    "c_name",
                    ( CASE
                        WHEN "c_room" IS NULL THEN 1
                        ELSE ROW_NUMBER()
                               OVER (
                                 PARTITION BY "c_room"
                                 ORDER BY "c_name" ASC)
                      END ) AS "position"
             FROM   "computers") "cross_apply_data"
         ON "r_id" = "c_room"
WHERE  "position" <= "r_space"
ORDER  BY "r_id",
          "c_id"
```

The numbering peculiarity is in the way MySQL and Oracle number the unassigned computers. For each unassigned computer, MySQL considers its “number in the room” to be 1 (which makes sense). This effect results from the logic of the `@row_num := IF(@prev_value = `c_room`, @row_num + 1, 1)` condition: if the previous value was `NULL`, and the next one is also `NULL`, they are not equal (`NULL` is not equal to itself). Finally, MySQL gets:

c_id	c_room	c_name	position
6	NULL	Unassigned computer A	1
7	NULL	Unassigned computer B	1
8	NULL	Unassigned computer C	1
1	1	The computer A in the room 1	1
2	1	The computer B in the room 1	2
3	2	The computer A in the room 2	1
4	2	The computer B in the room 2	2
5	2	The computer C in the room 2	3

Oracle is considering this situation in the behavior of the **ROW_NUMBER** function and treats all **NULL** values as equal to each other:

c_id	c_room	c_name	position
1	1	The computer A in the room 1	1
2	1	The computer B in the room 1	2
3	2	The computer A in the room 2	1
4	2	The computer B in the room 2	2
5	2	The computer C in the room 2	3
6	NULL	Unassigned computer A	1
7	NULL	Unassigned computer B	2
8	NULL	Unassigned computer C	3

To get Oracle to behave like MySQL, we use the **CASE** statement (lines 11-17).

This feature is not important for this particular problem (unassigned computers are not counted in any way, so their sequence number is not important), but it is useful to know and remember about this difference in the behavior of the two DBMSes.

Starting with version 12c, we can use an approach similar to the one used in MS SQL Server in Oracle. But there will be one key difference.

Oracle | Solution 2.2.10.n (12c and newer)

```
1    SELECT "r_id",
2           "r_name",
3           "r_space",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms"
8           OUTER APPLY (SELECT "c_id",
9                               "c_room",
10                              "c_name"
11                      FROM   "computers"
12                      WHERE "r_id" = "c_room"
13                      ORDER BY "c_name" ASC
14                      FETCH NEXT "r_space" ROWS ONLY)
15                      "cross_apply_data"
16   WHERE "c_room" IS NOT NULL
17   ORDER BY "r_id",
18            "c_id"
```

Because of peculiarities of row numbering logic and peculiarities of access to “external data” from subquery in the “right part” of **APPLY** in Oracle we cannot use **CROSS APPLY** (you can see for yourself that in case of **CROSS APPLY** such query will return empty result).

Therefore, in line 8 of this query we use **OUTER APPLY**, and the condition in line 16 of this query is needed to exclude empty rooms from the final result (which would contradict the condition of the problem).

Problem 2.2.10.o: show the arrangement of computers in all rooms so that the result does not include more computers than can fit in the room.

Expected result 2.2.10.o.

r_id	r_name	r_space	c_id	c_room	c_name
1	A room with two computers	5	1	1	The computer A in the room 1
1	A room with two computers	5	2	1	The computer B in the room 1
2	A room with three computers	5	3	2	The computer A in the room 2
2	A room with three computers	5	4	2	The computer B in the room 2
2	A room with three computers	5	5	2	The computer C in the room 2
3	An empty room 1	2	NULL	NULL	NULL
4	An empty room 2	2	NULL	NULL	NULL
5	An empty room 3	2	NULL	NULL	NULL

Solution 2.2.10.o: we'll use `OUTER APPLY` in MS SQL Server and emulate the same behavior in MySQL as well as in Oracle before version 12c (for version 12c and newer we'll also use `OUTER APPLY`).

The only difference between this problem and problem 2.2.10.n{187} is that we need to add empty rooms to the result. For MySQL and Oracle, this is done by replacing `JOIN` with `LEFT JOIN`, and for MS SQL Server, by replacing `CROSS APPLY` with `OUTER APPLY`. Also, in MySQL and Oracle we need to change the condition responsible for comparing computer number and room capacity a bit.

MySQL | Solution 2.2.10.o

```
SELECT `r_id`,
       `r_name`,
       `r_space`,
       `c_id`,
       `c_room`,
       `c_name`
FROM   `rooms`
       LEFT JOIN (SELECT `c_id`,
                         `c_room`,
                         `c_name`,
                         @row_num := IF(@prev_value = `c_room`, @row_num + 1, 1)
                         AS `position`,
                         @prev_value := `c_room`
                  FROM   `computers`,
                         (SELECT @row_num := 1) AS `x`,
                         (SELECT @prev_value := '') AS `y`
                  ORDER  BY `c_room`,
                            `c_name` ASC) AS `cross_apply_data`
        ON `r_id` = `c_room`
WHERE  `position` <= `r_space` OR `position` IS NULL
ORDER  BY `r_id`,
          `c_id`
```

MS SQL Solution 2.2.10.o

```
1    SELECT [r_id],
2           [r_name],
3           [r_space],
4           [c_id],
5           [c_room],
6           [c_name]
7    FROM   [rooms]
8           OUTER APPLY (SELECT TOP ([r_space])
9                               [c_id],
10                              [c_room],
11                              [c_name]
12                      FROM   [computers]
13                      WHERE [c_room] = [r_id]
14                      ORDER  BY [c_name] ASC) AS [cross_apply_data]
15   ORDER BY [r_id],
16             [c_id]
```

Oracle Solution 2.2.10.o (before 12c)

```
1    SELECT "r_id",
2           "r_name",
3           "r_space",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms"
8           LEFT JOIN (SELECT "c_id",
9                             "c_room",
10                            "c_name",
11                            ( CASE
12                                WHEN "c_room" IS NULL THEN 1
13                                ELSE ROW_NUMBER()
14                                       OVER (
15                                         PARTITION BY "c_room"
16                                         ORDER BY "c_name" ASC)
17                              END ) AS "position"
18                 FROM   "computers") "cross_apply_data"
19             ON "r_id" = "c_room"
20   WHERE  "position" <= "r_space" OR "position" IS NULL
21   ORDER  BY "r_id",
22             "c_id"
```

If we had not added the second part of the condition (`OR position IS NULL`) to the lines 20 queries for MySQL and Oracle, the empty rooms would not be included in the final sample, because they do not match computers, that is, the "number" of any computer for them is `NULL`, and comparing `NULL` with the `r_space` field value results `FALSE`.

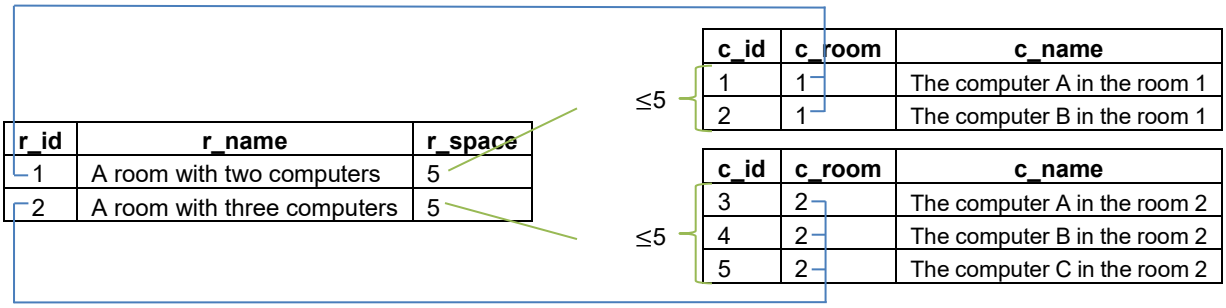
Solution for Oracle 12c (and newer) looks like the solution for MS SQL Server.

Oracle Solution 2.2.10.o (12c and newer)

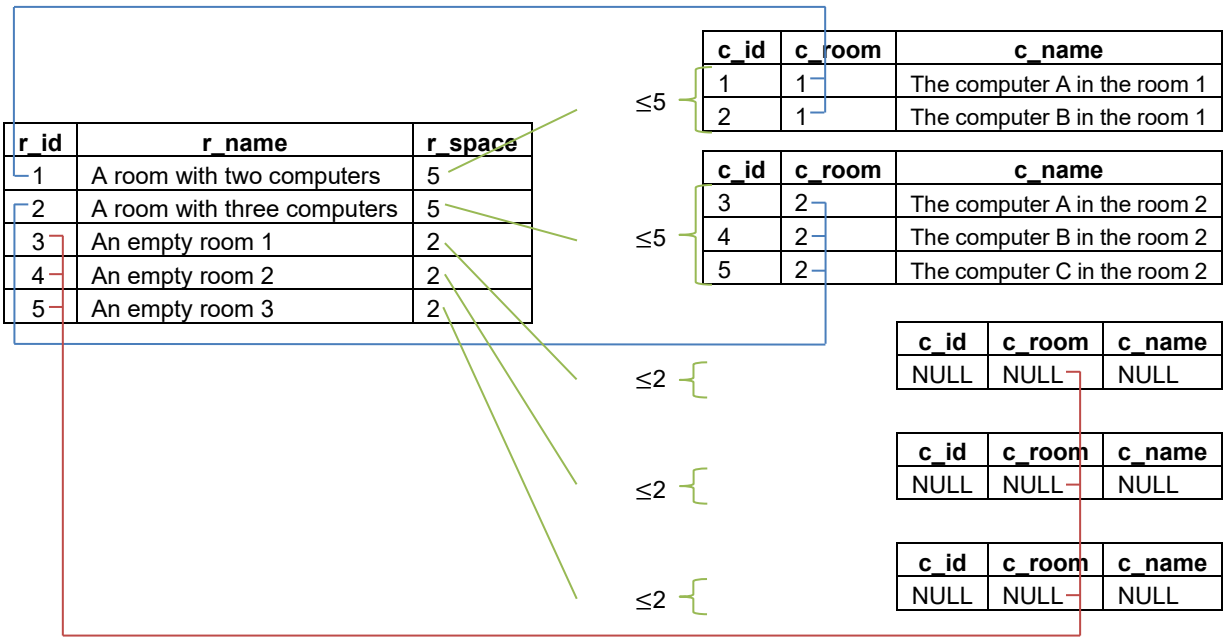
```
1    SELECT "r_id",
2           "r_name",
3           "r_space",
4           "c_id",
5           "c_room",
6           "c_name"
7    FROM   "rooms"
8           OUTER APPLY (SELECT "c_id",
9                               "c_room",
10                              "c_name"
11                      FROM   "computers"
12                      ORDER BY "c_name" ASC
13                      FETCH NEXT "r_space" ROWS ONLY)
14                      "cross_apply_data"
15   ORDER BY "r_id",
16             "c_id"
```

Let's explain again with a graphical example the logic and difference between **`CROSS APPLY`** and **`OUTER APPLY`**.

In problem 2.2.10.n{187} (**`CROSS APPLY`**):



In problem 2.2.10.o{191} (**`OUTER APPLY`**):



Task 2.2.10.TSK.A: show information about which readers have borrowed books from the library and when.

Task 2.2.10.TSK.B: show information about all readers and the dates of their borrowing of books from the library.

Task 2.2.10.TSK.C: show information about readers who have never borrowed a book from the library.

Task 2.2.10.TSK.D: show books that have never been borrowed by any of the readers.

Task 2.2.10.TSK.E: show information about what books each reader can borrow from the library in principle.

Task 2.2.10.TSK.F: show information about which books (assuming they haven't already borrowed them) each reader can borrow from the library.

Task 2.2.10.TSK.G: show information about what books published before 2010 can be borrowed from the library by each of the readers.

Task 2.2.10.TSK.H: show information about what books published before 2010 can each reader borrow from the library (assuming they haven't already borrowed them).

2.3. Data Modification

2.3.1. Example 21: Data Insertion

A very useful source of good examples of performing data insertion for any DBMS is studying database dumps (there you will also see many ready-made examples of SQL constructs for creating tables, links, triggers, etc.)

All further considerations about the method of transferring string data are based on the assumption that the databases are created in the following encodings:

- MySQL: utf8 / utf8_general_ci;
- MS SQL Server: UNICODE / Cyrillic_General_CI_AS;
- Oracle: AL32UTF8 / AL16UTF16.

The topic of encodings and working with them is huge, very specific for each DBMS and will hardly be covered in this book (except for problems 7.3.2.a{542} and 7.3.2.b{542}) — see the documentation for the corresponding DBMS.

Problem 2.3.1.a{195}: add to the database the information that the reader with identifier 4 borrowed a book with identifier 3 from the library on January 15, 2016, and promised to return it on January 30, 2016.

Problem 2.3.1.b{198}: add to the database the information that the reader with identifier 2 borrowed books with identifiers 1, 3, 5 from the library on January 25, 2016, and promised to return them on April 30, 2016.

Expected result 2.3.1.a: this new entry should appear in the `subscriptions` table.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
101	4	3	2016-01-15	2016-01-30	N

Expected result 2.3.1.b: these three new entries should appear in the `subscriptions` table.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
102	2	1	2016-01-25	2016-04-30	N
103	2	3	2016-01-25	2016-04-30	N
104	2	5	2016-01-25	2016-04-30	N

Solution 2.3.1.a{195}.

In all three DBMSes, we have created auto-incrementable primary keys in the `subscriptions` table, so we don't need to specify their values. It remains only to pass the data we know.

MySQL Solution 2.3.1.a

```
1    INSERT INTO `subscriptions`
2                (`sb_id`,
3                 `sb_subscriber`,
4                 `sb_book`,
5                 `sb_start`,
6                 `sb_finish`,
7                 `sb_is_active`)
8    VALUES      (NULL,
9                 4,
10                3,
11                '2016-01-15',
12                '2016-01-30',
13                'N')
```

If the insertion is performed in all fields of the table, the part of the query on lines 2-7 may be omitted, otherwise this part is mandatory.

If we do not want to explicitly specify a value of auto-incrementable primary key, in MySQL, we can pass **`NULL`** instead of its value or exclude this field from the list of fields and do not pass any data (i.e., remove field name in line 2 and **`NULL`** value in line 8).

MS SQL Solution 2.3.1.a

```
1    INSERT INTO [subscriptions]
2                ([sb_subscriber],
3                 [sb_book],
4                 [sb_start],
5                 [sb_finish],
6                 [sb_is_active])
7    VALUES      (4,
8                 3,
9                 CAST(N'2016-01-15' AS DATE),
10                CAST(N'2016-01-30' AS DATE),
11                N'N')
```

In MS SQL Server, auto incrementation of a primary key is performed by marking the corresponding field as **`IDENTITY`**. Inserting **`NULL`** and **`DEFAULT`** values into such a field is prohibited, so we must exclude it from the list of fields and not pass any values into it.

Even if we knew the exact value of the primary key of the record to be inserted, we would still have to perform two additional operations to insert it:

- before inserting the data, run the command **`SET IDENTITY_INSERT [subscriptions] ON`**;
- after inserting the data, run the command **`SET IDENTITY_INSERT [subscriptions] OFF`**.

These commands respectively allow and deny the insertion of explicitly passed values into the **`IDENTITY`** field.

In lines 9-10 of the query the explicit conversion of the string containing the date to the **`DATE`** data type is performed. MS SQL Server allows us not to do this (the conversion is done automatically), but for safety reasons it is recommended to use the explicit conversion.

The same safety considerations are behind the need to put the letter **`N`** before string constants (lines 9-11 of the query) to explicitly indicate to the DBMS that the strings are represented in the so-called “national encoding” (and Unicode as a form of character representation). For English and date strings, this rule can be neglected, but as soon as at least one character appears in the string that is represented differently in different encodings, we risk corrupting the data.

Interesting fact is that even in our case (format of `sb_is_active` field is `CHAR(1)`, not `NCHAR(1)`) in the database dump created by MS SQL Server Management Studio tools the letter `N` is present before the values of `sb_is_active` field. Short conclusion: if we have doubts whether to use the letter `N` before string constants or not, we'd better use it.

Oracle | Solution 2.3.1.a

```
1    INSERT INTO "subscriptions"
2                ("sb_id",
3                 "sb_subscriber",
4                 "sb_book",
5                 "sb_start",
6                 "sb_finish",
7                 "sb_is_active")
8    VALUES      (NULL,
9                 4,
10                3,
11                TO_DATE('2016-01-15', 'YYYY-MM-DD'),
12                TO_DATE('2016-01-30', 'YYYY-MM-DD'),
13                'N')
```

In Oracle it is obligatory to convert string representation of dates to an appropriate type.

Unlike MS SQL Server there is no need to specify the letter `N` before string constants with date values and `sb_is_active` field (we can do it, it will not cause an error). But for most other data (for example, text in Russian) it is better to specify the letter `N`.

Of particular interest is the passing of the value of an auto-incrementable primary key. In Oracle, such a key is implemented in a non-trivial way by creating a sequence (`SEQUENCE`) as a "source of numbers" and a trigger (`TRIGGER`), which receives the next number from the sequence and uses it as the value of the primary key of the inserted record.

It follows that we can pass… anything as a value of the autoincrementable key: `NULL`, `DEFAULT`, a number, anything; and the trigger will anyway replace the value we pass with the next number from the sequence.

If we need to explicitly specify the value of the `sb_id` field, we need to perform two additional operations:

- before inserting the data, run the command `ALTER TRIGGER "TRG_subscriptions_sb_id" DISABLE`;
- after inserting the data, run the command `ALTER TRIGGER "TRG_subscriptions_sb_id" ENABLE`.

These commands respectively disable and re-enable the trigger that sets the value of the auto-incrementable primary key.

Solution 2.3.1.b{198}.

Technically, we can put this problem down to performing three separate queries similar to the ones presented in the solution of problem 2.3.1.a{195}, but there is a more efficient way of performing the insertion of a data set, which involves three steps:

- temporary disabling indexes;
- inserting data as one big block;
- enabling indexes.

The presence of indexes may significantly decrease the data modification speed, because the DBMS will have to spend a lot of resources on constantly updating the indexes to keep them up-to-date. Also, executing multiple separate queries instead of a single one (even if large) leads to additional overhead on transaction management and other DBMS internal mechanisms.

Disabling and re-enabling indexes makes sense only when modifying a really large data set (tens of thousands of records or more), but we will still consider the corresponding commands.

Be sure to read the relevant sections of your DBMS documentation. The recommendations for using certain commands (and even the applicability of the commands themselves) can vary greatly depending on a variety of factors.

MySQL has a special command to disable and re-enable indexes (`ALTER TABLE ... DISABLE KEYS` and `ALTER TABLE ... ENABLE KEYS`), but it only works on tables that work with the MyISAM storage engine. It does not work for the now ubiquitous InnoDB storage engine.

What can we do with InnoDB storage engine?

- Disable and then enable uniqueness control (actually, disable and enable unique indexes).
- Disable and then enable foreign key control.
- Disable and then re-enable automatic transactions commit (if we are going to perform several separate queries).
- We can also “play tricks” with auto-incrementable primary keys[6], but there is no general recommendation.

Disabling unique indexes and foreign key control is an extremely dangerous operation that can lead to catastrophic data corruption. Perform it only as a last resort if nothing else helps improve performance and you are 101 % sure that the data you are sending fully meets the requirements that unique indexes and foreign keys are intended to control.

Now leaving aside all the nuances and non-obvious recommendations, we are left with one main conclusion: do the insertion in one query (insert several lines at once), it is a faster option compared to several separate queries, each of which inserts one line.

[6] https://dev.mysql.com/doc/refman/8.0/en/innodb-auto-increment-handling.html#innodb-auto-increment-lock-mode-usage-implications

MySQL Solution 2.3.1.b

```
1    -- Relevant for MyISAM, not relevant for InnoDB:
2    ALTER TABLE `subscriptions` DISABLE KEYS; -- Disabling indexes.
3
4    -- The next two commands are VERY dangerous!
5    SET foreign_key_checks = 0; -- Disabling foreign key check.
6    SET unique_checks = 0; -- Disabling unique indexes.
7
8    SET autocommit = 0; -- Disabling automatic transactions commit.
9
10   -- The insertion query itself:
11   INSERT INTO `subscriptions`
12               (`sb_subscriber`,
13                `sb_book`,
14                `sb_start`,
15                `sb_finish`,
16                `sb_is_active`)
17   VALUES       ( 2,
18                  1,
19                  '2016-01-25',
20                  '2016-04-30',
21                  'N' ),
22                ( 2,
23                  3,
24                  '2016-01-25',
25                  '2016-04-30',
26                  'N' ),
27                ( 2,
28                  5,
29                  '2016-01-25',
30                  '2016-04-30',
31                  'N' );
32
33   COMMIT; -- Transaction commit.
34
35   SET autocommit = 1; -- Enabling automatic transactions commit.
36
37   SET unique_checks = 1; -- Enabling unique indexes.
38   SET foreign_key_checks = 1; -- Enabling foreign key check.
39
40   -- Relevant for MyISAM, not relevant for InnoDB:
41   ALTER TABLE `subscriptions` ENABLE KEYS; -- Enabling indexes.
```

Once again, note that in the general case you can and should be limited to the query shown in lines 10-16. The rest of the ideas are given as background information.

In MS SQL Server we can also temporarily disable and then re-enable indexes with `ALTER INDEX ALL ON ... DISABLE` and `ALTER INDEX ALL ON ... REBUILD` commands.

Be sure to read at least two[7, 8] of the relevant sections of the documentation, as these operations can have very unexpected and dangerous side effects. They may seriously compromise the DBMS' ability to control data consistency.

[7] https://msdn.microsoft.com/en-us/library/ms177456.aspx
[8] https://msdn.microsoft.com/en-us/library/ms190645.aspx

MS SQL Solution 2.3.1.b

```
1    -- A VERY dangerous action!
2    ALTER INDEX ALL ON [subscriptions] DISABLE;
3
4    -- Be sure to enable the clustered index
5    -- (in our case, the primary key) before further
6    -- operations, otherwise queries will end up with an error.
7    ALTER INDEX [PK_subscriptions] ON [subscriptions] REBUILD;
8
9    -- The insertion query itself:
10   INSERT INTO [subscriptions]
11               ([sb_subscriber],
12                [sb_book],
13                [sb_start],
14                [sb_finish],
15                [sb_is_active])
16   VALUES       (2,
17                 1,
18                 CAST(N'2016-01-25' AS DATE),
19                 CAST(N'2016-04-30' AS DATE),
20                 N'N'),
21                (2,
22                 3,
23                 CAST(N'2016-01-25' AS DATE),
24                 CAST(N'2016-04-30' AS DATE),
25                 N'N'),
26                (2,
27                 5,
28                 CAST(N'2016-01-25' AS DATE),
29                 CAST(N'2016-04-30' AS DATE),
30                 N'N');
31
32   -- Enabling the indexes after disabling them:
33   ALTER INDEX ALL ON [subscriptions] REBUILD;
```

Unlike MySQL and MS SQL Server, Oracle has no ready-made solution for turning all indexes off and on. There are many algorithmic solutions[9] for this task, but the vast majority of authors quite rightly warn of many potential problems, recommend against it, and even suggest that removing and recreating an index may be more advantageous than turning it off and on.

The only index on the `subscriptions` table is its primary key, so in the example below we will enable and disable it.

Again and again, this is a very dangerous operation; you risk fatally damaging your database if something goes wrong in the way you planned or expected. Never try to perform such experiments on real databases, but only on copies of them, which you do not regret to lose.

Another important aspect of the Oracle solution is that this DBMS does not support the classical syntax for inserting multiple records in a single query, so we have to use the alternative syntax.

[9] http://johnlevandowski.com/oracle-disable-constraints-and-make-indexes-unusable/

Oracle | Solution 2.3.1.b

```
1    -- A VERY dangerous action!
2    ALTER TABLE "subscriptions" MODIFY CONSTRAINT "PK_subscriptions" DISABLE;
3
4    -- The insertion query itself:
5    INSERT ALL
6     INTO   "subscriptions"
7            ("sb_subscriber",
8             "sb_book",
9             "sb_start",
10            "sb_finish",
11            "sb_is_active")
12    VALUES (2,
13            1,
14            TO_DATE('2016-01-25', 'YYYY-MM-DD'),
15            TO_DATE('2016-04-30', 'YYYY-MM-DD'),
16            'N')
17    INTO   "subscriptions"
18           ("sb_subscriber",
19            "sb_book",
20            "sb_start",
21            "sb_finish",
22            "sb_is_active")
23    VALUES (2,
24            3,
25            TO_DATE('2016-01-25', 'YYYY-MM-DD'),
26            TO_DATE('2016-04-30', 'YYYY-MM-DD'),
27            'N')
28    INTO   "subscriptions"
29           ("sb_subscriber",
30            "sb_book",
31            "sb_start",
32            "sb_finish",
33            "sb_is_active")
34    VALUES (2,
35            5,
36            TO_DATE('2016-01-25', 'YYYY-MM-DD'),
37            TO_DATE('2016-04-30', 'YYYY-MM-DD'),
38            'N')
39   SELECT 1 FROM "DUAL";
40
41   -- Enabling the index after disabling it:
42   ALTER TABLE "subscriptions" MODIFY CONSTRAINT "PK_subscriptions" ENABLE;
```

Exploration 2.3.1.EXP.A: let's compare the data insertion performance based on cyclic repetition (1000 iterations) of a query inserting one record and executing one query inserting 1000 records at a time. Let's execute each option a hundred times in turn.

The median values of the time taken to insert a thousand records by each of the options are as follows:

	MySQL	MS SQL Server	Oracle
Cycle	3.000	2.000	6.000
One query	0.112	0.937	5.807

The numbers for the first (cyclic) option have such neat values because the execution time for each individual query was counted to the nearest thousandth of a second, then the resulting median value was multiplied by a thousand.

Each DBMS has a different ratio of running times of the two solutions presented, but the option of inserting a large number of records in one query wins everywhere in terms of performance.

In Oracle, such a small difference in the running time of the two presented options is caused by the fact that, actually, it is the same option, only written in different ways. In this DBMS it is also possible to achieve very high insertion performance, but it is achieved by more complex ways[10].

To increase the reliability of insertion operations, it is recommended to get information on the number of rows affected by the operation after their execution at the application level. If the received number does not coincide with the number of rows passed to the insertion, there was an error somewhere.

Task 2.3.1.TSK.A: add information about three new readers to the database, e.g., “Orlov O.O.”, “Sokolov S.S.”, “Berkutov B.B.”.

Task 2.3.1.TSK.B: reflect in the database that each of the readers added in task 2.3.1.TSK.A borrowed on January 20, 2016 the “Course of Theoretical Physics” book for a month.

Task 2.3.1.TSK.C: add to the database five any authors and ten books by these authors (two for each); if necessary, add the corresponding genres to the database. Reflect the authorship of the added books and their belonging to the corresponding genres.

[10] http://www.oracle.com/technetwork/issue-archive/2012/12-sep/o52plsql-1709862.html

2.3.2. Example 22: Data Update

Problem 2.3.2.a{203}: for subscription with identifier 99 change the return date to the current date and indicate that the book is returned.

Problem 2.3.2.b{204}: change the expected return date for all books that the reader with identifier 2 borrowed on January 25, 2016, to "plus two months" (i.e., the reader will read them two months longer than they planned).

Expected result 2.3.2.a: a row with a primary key equal to 99 will look like this (your date in the `sb_finish` field will be different).

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
99	4	4	2015-10-08	2016-01-06	N

Expected result 2.3.2.b: the following rows will look like this.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
102	2	1	2016-01-25	2016-06-30	N
103	2	3	2016-01-25	2016-06-30	N
104	2	5	2016-01-25	2016-06-30	N

Solution 2.3.2.a{203}:

The current date value can be obtained on the application side and transferred to DBMS, then the solution will be very simple (see option 1), but the current date can also be obtained on the DBMS side (see option 2), which does not complicate the query much.

MySQL Solution 2.3.2.a

```
1    -- Option 1: direct substitution of the current date into the query.
2    UPDATE `subscriptions`
3    SET    `sb_finish` = '2016-01-06',
4           `sb_is_active` = 'N'
5    WHERE  `sb_id` = 99
6
7    -- Option 2: getting the current date on the DBMS side.
8    UPDATE `subscriptions`
9    SET    `sb_finish` = CURDATE(),
10          `sb_is_active` = 'N'
11   WHERE  `sb_id` = 99
```

MS SQL Solution 2.3.2.a

```
1    -- Option 1: direct substitution of the current date into the query.
2    UPDATE [subscriptions]
3    SET    [sb_finish] = CAST(N'2016-01-06' AS DATE),
4           [sb_is_active] = N'N'
5    WHERE  [sb_id] = 99
6
7    -- Option 2: getting the current date on the DBMS side.
8    UPDATE [subscriptions]
9    SET    [sb_finish] = CONVERT(date, GETDATE()),
10          [sb_is_active] = N'N'
11   WHERE  [sb_id] = 99
```

Oracle | Solution 2.3.2.a

```
1    -- Option 1: direct substitution of the current date into the query.
2    UPDATE "subscriptions"
3    SET    "sb_finish" = TO_DATE('2016-01-06', 'YYYY-MM-DD'),
4           "sb_is_active" = 'N'
5    WHERE  "sb_id" = 99
6
7    -- Option 2: getting the current date on the DBMS side.
8    UPDATE "subscriptions"
9    SET    "sb_finish" = TRUNC(SYSDATE),
10          "sb_is_active" = 'N'
11   WHERE  "sb_id" = 99
```

Applying the **TRUNC** function in line 9 of the query is necessary to get only the date from the full date-time representation format.

Be careful to make sure that your update query contains a condition (in this problem, lines 5 and 11 of all three queries). **UPDATE** without a condition will update **all** records in the table, so you will "cripple" the data.

There are two other very common mistakes in any task related to the concept of "current date":

1) If the user and the server with the DBMS are in different time zones, every 24 hours there is a period of time when the "current date" is different for the user and the server. This should be considered by making appropriate adjustments either at the level of displaying data to the user or at the level of data storage.

2) If a date contains not only information about a year, month and day, but also hours, minutes, seconds (fractions of seconds), comparison operations can give unexpected results (e.g., that from 01.01.2011 to 01.01.2012 a year has not passed, if in the database the first date is represented as "2011-01-01 15:28:27" and "today" is represented as "2012-01-01 11:23:17", there are just over four hours to pass a year)

There is no one-size-fits-all solution to these situations. A holistic approach is required, starting with the selection of the optimal storage format, and ending with the implementation and in-depth testing of data processing algorithms.

Solution 2.3.2.b{203}.

In contrast to the previous problem, here it is extremely unreasonable to get the new date from the application and put the ready value into the query (we don't know how many records we have to process, and different records may well have different initial values of the date to be updated). Thus, we have to add two months to date using DBMS tools.

Typical mistake: decompose the date into string fragments, get a numeric representation of the month, add the desired "offset" to it, convert it back to a string, and "assemble" the new date. Imagine, we need to add six months to December 15, 2011: 2011-12-15 → 2011-**18**-15. We get 18th month, which is somewhat illogical. With negative offsets we get an even more "funny" picture.

Conclusion: operations with addition or subtraction of time intervals should be performed with the help of special tools that can consider years, months, days, hours, minutes, seconds, etc.

MySQL Solution 2.3.2.b

```
1    UPDATE `subscriptions`
2    SET    `sb_finish` = DATE_ADD(`sb_finish`, INTERVAL 2 MONTH)
3    WHERE  `sb_subscriber` = 2
4           AND `sb_start` = '2016-01-25';
```

MS SQL Solution 2.3.2.b

```
1    UPDATE [subscriptions]
2    SET    [sb_finish] = DATEADD(month, 2, [sb_finish])
3    WHERE  [sb_subscriber] = 2
4           AND [sb_start] = CONVERT(date, '2016-01-25');
```

Oracle Solution 2.3.2.b

```
1    UPDATE "subscriptions"
2    SET    "sb_finish" = ADD_MONTHS("sb_finish", 2)
3    WHERE  "sb_subscriber" = 2
4           AND "sb_start" = TO_DATE('2016-01-25', 'YYYY-MM-DD');
```

In all three DBMSes solutions are quite simple and implemented in an identical way, except for the specifics of date increment functions.

To increase the reliability of update operations, it is recommended to receive information after their execution on the application level about how many rows were affected by the operation. If the received number does not coincide with the expected number, there was an error somewhere.

Yes, we cannot always know in advance how many records will be affected by the update, but there are still cases where it is known (e.g., the librarian checked a certain three items and clicked "Book Returned", so three records should be affected by the update).

Task 2.3.2.TSK.A: mark all subscriptions with identifiers ≤50 as returned.

Task 2.3.2.TSK.B: for all subscriptions made before January 1, 2012, decrease the value of the day of delivery by 3.

Task 2.3.2.TSK.C: mark all subscriptions for the reader with identifier 2 as unreturned.

2.3.3. Example 23: Data Deletion

Problem 2.3.3.a{206}: delete the information that the reader with identifier 4 borrowed a book with identifier 3 from the library on January 15, 2016.

Problem 2.3.3.b{207}: delete information about all library visits by the reader with identifier 3 on Sundays.

Expected result 2.3.3.a: the following entry must be deleted.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
101	4	3	2016-01-15	2016-01-30	N

Expected result 2.3.3.b: the following entries must be deleted.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
62	3	5	2014-08-03	2014-10-03	Y
86	3	1	2014-08-03	2014-09-03	Y

Solution 2.3.3.a{206}.

In this case the solution is trivial and identical for all three DBMS, except for syntax of `sb_start` field value formation.

MySQL Solution 2.3.3.a

```
1    DELETE FROM `subscriptions`
2    WHERE  `sb_subscriber` = 4
3           AND `sb_start` = '2016-01-15'
4           AND `sb_book` = 3
```

MS SQL Solution 2.3.3.a

```
1    DELETE FROM [subscriptions]
2    WHERE  [sb_subscriber] = 4
3           AND [sb_start] = CONVERT(date, '2016-01-15')
4           AND [sb_book] = 3
```

Oracle Solution 2.3.3.a

```
1    DELETE FROM "subscriptions"
2    WHERE  "sb_subscriber" = 4
3           AND "sb_start" = TO_DATE('2016-01-15', 'YYYY-MM-DD')
4           AND "sb_book" = 3
```

Be careful to make sure that your deletion queries contain a condition (in this problem, lines 2-4 of all three queries). `DELETE` without a condition will delete **all** records in the table, i.e., you will lose all data.

Solution 2.3.3.b{206}.

In the solution{41} of problem 2.1.7.b{41} we've discussed in detail why it is better to use ranges rather than extract date fragments, and stipulated that there may be exceptions where ranges would not be a sufficient solution. This task is just such an exception: we have to get information about the number of the day of the week based on the original date value.

MySQL Solution 2.3.3.b

```
1    DELETE FROM `subscriptions`
2    WHERE  `sb_subscriber` = 3
3           AND DAYOFWEEK(`sb_start`) = 1
```

Note: the **DAYOFWEEK** function in MySQL returns the number of the day, starting on Sunday (1) and ending on Saturday (7).

MS SQL Solution 2.3.3.b

```
1    DELETE FROM [subscriptions]
2    WHERE  [sb_subscriber] = 3
3           AND DATEPART(weekday, [sb_start]) = 1
```

In MS SQL Server the **DATEPART** function also numbers the days of the week starting from Sunday.

Oracle Solution 2.3.3.b

```
1    DELETE FROM "subscriptions"
2    WHERE  "sb_subscriber" = 3
3           AND TO_CHAR("sb_start", 'D') = 1
```

Oracle behaves similarly: getting the number of the day of the week, the **TO_CHAR** function starts numbering from Sunday.

The logic behind the behavior of functions that determine the day of the week number may vary from DBMS to DBMS, depending on DBMS settings, operating system, and other factors. Do not assume that such functions will always and everywhere work the way you are used to.

To increase the reliability of deletion operations, it is recommended to receive information on the application level about how many rows were affected by the operation. If the received number does not coincide with the expected one, there was an error somewhere.

Yes, we cannot always know in advance how many records will be affected by deletion, but there are still cases where it is known (for example, the librarian checked the checkboxes for some three items and clicked "Delete", so three records must be affected by deletion).

Task 2.3.3.TSK.A: delete information about all subscriptions to the book with identifier 1.

Task 2.3.3.TSK.B: delete all books belonging to the "Classic" genre.

Task 2.3.3.TSK.C: delete the information about all subscriptions made after the 20th day of any month of any year.

2.3.4. Example 24: Data Merging

Problem 2.3.4.a{208}: add the “Philosophy”, “Detective”, and “Classic” genres to the database.

Problem 2.3.4.b{210}: copy (without duplication) the contents of the `genres` table from the “Big Library (for Experiments)” database into the “Library” database; if the primary keys match, add the “ [OLD]” suffix to the existing genre name.

Expected result 2.3.4.a: the contents of the genres table should look like this (note: the genre “Classic” was already in this table and must not be duplicated).

g_id	g_name
1	Poetry
2	Programming
3	Psychology
4	Science
5	Classic
6	Science Fiction
7	Philosophy
8	Detective

Expected result 2.3.4.b: this result is obtained on “reference data”; if you solve problem 2.3.4.a first, your expected result will contain all eight genres from the “Library” and some more genres from the “Big Library”.

g_id	g_name
1	Poetry [OLD]
2	Programming [OLD]
3	Psychology [OLD]
4	Science [OLD]
5	Classic [OLD]
6	Science Fiction [OLD]
And more rows with from the “Big Library”	

Solution 2.3.4.a{208}.

In all three DBMSes, a unique index is built on the `g_name` field of the `genres` table to eliminate the possibility of genres having the same name.

This problem can be solved by sequentially executing three `INSERT` queries and keeping track of their results (a query for inserting the “Classic” genre will result in an error). But it is possible to implement a more elegant solution.

MySQL supports the `REPLACE` operator, which works like `INSERT`, but considers the values of the primary key and unique indexes, adding a new row if there is no match, and updating an existing one if there is a match.

MySQL Solution 2.3.4.a

```
1    REPLACE INTO `genres`
2                (`g_id`,
3                 `g_name`)
4    VALUES      (NULL,
5                 'Philosophy'),
6                (NULL,
7                 'Detective'),
8                (NULL,
9                 'Classic')
```

With this approach, we can pass a lot of new data in one query (which is executed without errors), without fear of problems associated with primary keys and unique indexes duplication.

Unfortunately, MS SQL Server and Oracle do not support the **REPLACE** operator, but a similar behavior can be achieved using the **MERGE** operator.

MS SQL Solution 2.3.4.a

```
1    MERGE INTO [genres]
2    USING ( VALUES (N'Philosophy'),
3                   (N'Detective'),
4                   (N'Classic') ) AS [new_genres]([g_name])
5    ON [genres].[g_name] = [new_genres].[g_name]
6    WHEN NOT MATCHED BY TARGET THEN
7      INSERT ([g_name])
8      VALUES ([new_genres].[g_name]);
```

In this query, lines 2-4 are a non-trivial way to dynamically create a table, because the **MERGE** operator cannot get “for input” anything but tables and their merge condition.

Line 5 describes the condition to check (whether the name of the new genre matches the name of an existing one), and lines 6-8 tell DBMS to insert the data into the target table only if the condition is not met (i.e., there is no duplication).

Note the presence of the `;` character at the end of line 8. MS SQL Server requires it at the end of the **MERGE** operator.

Oracle Solution 2.3.4.a

```
1    MERGE INTO "genres"
2    USING (SELECT ( N'Philosophy' ) AS "g_name"
3           FROM   dual
4           UNION
5           SELECT ( N'Detective' ) AS "g_name"
6           FROM   dual
7           UNION
8           SELECT ( N'Classic' ) AS "g_name"
9           FROM   dual) "new_genres"
10   ON ( "genres"."g_name" = "new_genres"."g_name" )
11   WHEN NOT MATCHED THEN
12     INSERT ("g_name")
13     VALUES ("new_genres"."g_name")
```

Solution for Oracle is built using the same logic as the solution for MS SQL Server. The only difference is in the way the table (rows 2-9) is dynamically generated from the new data.

Solution 2.3.4.b[208].

To solve this problem with MySQL, we need to set up a connection to the DBMS as a user who has permissions to work with both databases (let's assume the "Library" is called `library` and the "Big Library" is called `huge_library`).

Then it remains to use the quite classical `INSERT ... SELECT` (it allows us to insert into the table the data obtained by executing the `SELECT` query; lines 1-8), and the problem condition about adding the " [OLD]" suffix is implemented through a special syntax `ON DUPLICATE KEY UPDATE` (lines 9-11), which instructs MySQL to update the target table record if a primary key or unique index overlap occurs between already existing and added data.

MySQL Solution 2.3.4.b

```
1    INSERT INTO `library`.`genres`
2                (
3                 `g_id`,
4                 `g_name`
5                )
6    SELECT `g_id`,
7           `g_name`
8    FROM   `huge_library`.`genres`
9    ON DUPLICATE KEY
10   UPDATE `library`.`genres`.`g_name` =
11          CONCAT(`library`.`genres`.`g_name`, ' [OLD]')
```

To solve this problem with MS SQL Server (as in the case of MySQL) we need to establish a connection to the database as a user who has permissions to work with both databases (let's assume the "Library" is called `[library]` and the "Big Library" is called `[huge_library]`).

MS SQL Server does not support `ON DUPLICATE KEY UPDATE`, but the same behavior can be achieved using `MERGE`.

MS SQL Solution 2.3.4.b

```
1    -- Allowing insertion of explicitly passed values in the IDENTITY field:
2    SET IDENTITY_INSERT [genres] ON;
3
4    -- Data merging:
5    MERGE [library].[dbo].[genres]      AS [destination]
6    USING [huge_library].[dbo].[genres] AS [source]
7    ON [destination].[g_id] = [source].[g_id]
8    WHEN MATCHED THEN
9      UPDATE SET [destination].[g_name] =
10               CONCAT([destination].[g_name], N' [OLD]')
11   WHEN NOT MATCHED THEN
12     INSERT ([g_id],
13             [g_name])
14     VALUES ([source].[g_id],
15             [source].[g_name]);
16
17   -- Prohibiting insertion of explicitly passed values in the IDENTITY field:
18   SET IDENTITY_INSERT [genres] OFF;
```

Since the `g_id` field is an `IDENTITY` field, we must explicitly allow data to be inserted there (line 2) before doing the insertion, and then prohibit it (line 18).

Lines 5-7 of the query describe the source table, the destination table, and the value match condition to be checked.

Lines 8-10 and 11-15 describe, respectively, the reaction to a match (perform an update) and a mismatch (perform an insert) of the primary keys of the source and destination tables.

To solve this problem with Oracle (as with MySQL and MS SQL Server), we need to set up a connection to the DBMS as a user who has permissions to work with both databases (let's assume the "Library" is called `"library"` and the "Big Library" is called `"huge_library"`).

Oracle as well as MS SQL Server does not support `ON DUPLICATE KEY UPDATE`, but the same behavior can be achieved using `MERGE`.

Solution for Oracle is similar to MS SQL Server, except for the necessity to turn off (line 2) the trigger responsible for autoincrementing of the primary key before inserting data and turn it on again (line 18) after the insertion.

Oracle | Solution 2.3.4.b

```
1   -- Disabling the trigger that provides autoincrementing of the primary key:
2   ALTER TRIGGER "library"."TRG_genres_g_id" DISABLE;
3
4   -- Data merging:
5   MERGE INTO "library"."genres" "destination"
6   USING "huge_library"."genres" "source"
7   ON ("destination"."g_id" = "source"."g_id")
8   WHEN MATCHED THEN
9     UPDATE SET "destination"."g_name" =
10           CONCAT("destination"."g_name", N' [OLD]')
11  WHEN NOT MATCHED THEN
12    INSERT ("g_id",
13            "g_name")
14    VALUES ("source"."g_id",
15            "source"."g_name");
16
17  -- Enabling the trigger that provides autoincrementing of the primary key:
18  ALTER TRIGGER "library"."TRG_genres_g_id" ENABLE;
```

If for some reason we cannot access the DBMS as a user with access to both schemas we are interested in, we can take the following route (such options were not considered for MySQL and MS SQL Server, since these DBMSes do not support some features, and the complexity of workarounds is beyond the scope of this book, it is orders of magnitude easier to create a user with access permission to both databases (schemas)).

Lines 6-23 of the large set of queries below are similar to the solution just discussed, and all the rest of the code (lines 1-4, 25-37) is needed to allow simultaneous access to data in two different schemas.

Lines 2-4 are responsible for creating and opening a connection to the source schema.

The `COMMIT` command in line 26 is needed because without it we risk losing some data when closing the connection.

Lines 29 and 32-34 present two options for closing the connection. As a rule, the first option works, but if it does not work, there is the second option.

In line 37 a previously created connection is dropped.

Oracle — Solution 2.3.4.b (working on behalf of two users)

```
-- Creating a connection to the source schema:
CREATE DATABASE LINK "huge"
CONNECT TO "login" IDENTIFIED BY "password"
USING 'localhost:1521/xe';

-- Disabling the trigger that provides autoincrement of the primary key:
ALTER TRIGGER "TRG_genres_g_id" DISABLE;

-- Data merging:
MERGE INTO "genres" "destination"
USING "genres"@"huge" "source"
ON ("destination"."g_id" = "source"."g_id")
WHEN MATCHED THEN
  UPDATE SET "destination"."g_name" =
             CONCAT("destination"."g_name", N' [OLD]')
WHEN NOT MATCHED THEN
  INSERT ("g_id",
          "g_name")
  VALUES ("source"."g_id",
          "source"."g_name");

-- Enabling the trigger that provides autoincrement of the primary key:
ALTER TRIGGER "TRG_genres_g_id" ENABLE;

-- Explicit confirmation of saving all changes:
COMMIT;

-- Closing the connection to the source schema:
ALTER SESSION CLOSE DATABASE LINK "huge";

-- If the ALTER SESSION CLOSE does not work...
BEGIN
DBMS_SESSION.CLOSE_DATABASE_LINK('huge');
END;

-- Dropping the connection to the source schema:
DROP DATABASE LINK "huge";
```

Task 2.3.4.TSK.A: add the “Politics”, “Psychology”, and “History” genres to the database.

Task 2.3.4.TSK.B: copy (without duplication) the contents of the **`subscribers`** table from the “Big Library (for Experiments)” database into the “Library” database; if the primary keys match, add the “ [OLD]” suffix to the existing subscriber name.

2.3.5. Example 25: Conditional Data Modification

Problem 2.3.5.a[{214}]: add to the database the information that the reader with identifier 4 borrowed books with identifiers 2 and 3 from the library on February 1, 2015, and planned to return them no later than July 20, 2015; if the current date is greater than July 20, 2015, mark the subscription as unreturned, if the current date less than (or equal to) July 20, 2015, mark the subscriptions as returned.

Problem 2.3.5.b[{216}]: change the return dates for all subscriptions to “two months from the present day” if the book is not returned and the corresponding reader now has more than two books on hand, and to “one month from the present day” otherwise (if the book is returned or the corresponding reader now has no more than two books on hand).

Problem 2.3.5.c[{218}]: update all reader names by adding to the end in square brackets the number of unreturned books (e.g., “ [3]”) and the words “ [RED]”, “ [YELLOW]”, “ [GREEN]”, respectively, if the reader now has more than five books on hand, three to five, less than three.

The expected results are presented under the assumption that you are using a “clean” copy of the “Library” database, unchanged by previous data modification examples, but each problem in this example works with the data modified by the previous problem.

Expected result 2.3.5.a: the following two entries should appear in the `subscriptions` table.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
101	4	2	2015-02-01	2015-07-20	Y
102	4	3	2015-02-01	2015-07-20	Y

Expected result 2.3.5.b.

				Before	After	
sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_finish	sb_is_active
2	1	1	2011-01-12	2011-02-12	2011-03-12	N
3	3	3	2012-05-17	2012-07-17	2012-09-17	Y
42	1	2	2012-06-11	2012-08-11	2012-09-11	N
57	4	5	2012-06-11	2012-08-11	2012-09-11	N
61	1	7	2014-08-03	2014-10-03	2014-11-03	N
62	3	5	2014-08-03	2014-10-03	2014-12-03	Y
86	3	1	2014-08-03	2014-09-03	2014-11-03	Y
91	4	1	2015-10-07	2015-03-07	2015-05-07	Y
95	1	4	2015-10-07	2015-11-07	2015-12-07	N
99	4	4	2015-10-08	2025-11-08	2026-01-08	Y
100	1	3	2011-01-12	2011-02-12	2011-03-12	N
101	4	2	2015-02-01	2015-07-20	2015-09-20	Y
102	4	3	2015-02-01	2015-07-20	2015-09-20	Y

Expected result 2.3.5.c.

s_id	s_name
1	Ivanov I.I. [0] [GREEN]
2	Petrov P.P. [0] [GREEN]
3	Sidorov S.S. [3] [YELLOW]
4	Sidorov S.S. [4] [YELLOW]

The solutions to all problems of this example are built on the use of a **`CASE`** expression, which allows us to consider several data options or several options of DBMS behavior within one query.

Solution 2.3.5.a[{213}].

MySQL | Solution 2.3.5.a

```
1    INSERT INTO `subscriptions`
2                (
3                 `sb_id`,
4                 `sb_subscriber`,
5                 `sb_book`,
6                 `sb_start`,
7                 `sb_finish`,
8                 `sb_is_active`
9                )
10               VALUES
11               (
12                NULL,
13                4,
14                2,
15                '2015-02-01',
16                '2015-07-20',
17                CASE
18                 WHEN CURDATE() > '2015-07-20'
19                 THEN (SELECT 'Y')
20                 ELSE (SELECT 'N')
21                END
22               ),
23               (
24                NULL,
25                4,
26                3,
27                '2015-02-01',
28                '2015-07-20',
29                CASE
30                 WHEN CURDATE() > '2015-07-20'
31                 THEN (SELECT 'Y')
32                 ELSE (SELECT 'N')
33                END
34               )
```

In lines 17-21 and 29-33 the **`CASE`** clause allows us to determine the current date at the DBMS level, compare it to the given date, and conclude which value (**`Y`** or **`N`**) the **`sb_is_active`** field should take.

In MySQL, simplified syntax of the **`THEN 'N'`** instead of **`THEN (SELECT 'N')`** also works. This is not a mistake, but such code is much harder to read and understand, because in the SQL context, the usual way to get some value is using **`SELECT`**.

Solution for MS SQL Server is implemented in a similar way:

MS SQL Solution 2.3.5.a

```
INSERT INTO [subscriptions]
            (
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active]
            )
            VALUES
            (
             4,
             2,
             CONVERT(date, '2015-02-01'),
             CONVERT(date, '2015-07-20'),
             CASE
              WHEN CONVERT(date, GETDATE()) > CONVERT(date, '2015-07-20')
              THEN (SELECT N'Y')
              ELSE (SELECT N'N')
             END
            ),
            (
             4,
             3,
             CONVERT(date, '2015-02-01'),
             CONVERT(date, '2015-07-20'),
             CASE
              WHEN CONVERT(date, GETDATE()) > CONVERT(date, '2015-07-20')
              THEN (SELECT N'Y')
              ELSE (SELECT N'N')
             END
            )
```

Like in MySQL, in MS SQL Server we can use the syntax of `THEN 'N'` instead of `THEN (SELECT 'N')`.

Solution for Oracle is implemented in a similar way.
Like in MySQL and MS SQL Server, we can use the syntax `THEN 'N'` instead of `THEN (SELECT 'N' FROM "DUAL")` in Oracle.

In this particular problem, there is no fundamental difference in using the syntax `THEN 'N'` or the `THEN (SELECT 'N')`: the DBMS query optimizer will still “understand” that a constant is chosen and replace the whole `SELECT` clause with its value.

But much more often we will not need to select a constant value, but to perform a full-fledged query, that’s why in this simple case, too, we recommend writing `THEN (SELECT 'N')` to maintain a unified style.

Oracle | Solution 2.3.5.a

```
1    INSERT ALL
2           INTO "subscriptions"
3                (
4                 "sb_subscriber",
5                 "sb_book",
6                 "sb_start",
7                 "sb_finish",
8                 "sb_is_active"
9                )
10               VALUES
11               (
12                4,
13                2,
14                TO_DATE('2015-02-01', 'YYYY-MM-DD'),
15                TO_DATE('2015-07-20', 'YYYY-MM-DD'),
16                CASE
17                 WHEN TRUNC(SYSDATE) > TO_DATE('2015-07-20', 'YYYY-MM-DD')
18                 THEN (SELECT 'Y' FROM "DUAL")
19                 ELSE (SELECT 'N' FROM "DUAL")
20                END
21               )
22          INTO "subscriptions"
23               (
24                "sb_subscriber",
25                "sb_book",
26                "sb_start",
27                "sb_finish",
28                "sb_is_active"
29               )
30               VALUES
31               (
32                4,
33                3,
34                TO_DATE('2015-02-01', 'YYYY-MM-DD'),
35                TO_DATE('2015-07-20', 'YYYY-MM-DD'),
36                CASE
37                 WHEN TRUNC(SYSDATE) > TO_DATE('2015-07-20', 'YYYY-MM-DD')
38                 THEN (SELECT 'Y' FROM "DUAL")
39                 ELSE (SELECT 'N' FROM "DUAL")
40                END
41               )
42   SELECT 1 FROM "DUAL";
```

Solution 2.3.5.b[{213}].

Unlike the previous problem, where the condition was extremely trivial and illustrative, here we have to solve two problems:

- Create a fairly complex compound condition based on a correlated subquery.
- “Convince” MySQL to allow the same table (`subscriptions`) to be used in the same query for both updating and reading, while MySQL prohibits such situations.

Lines 5-8 of the query are responsible for solving the second problem: we “wrap” the reading operation from the updated table into a subquery, which allows us to bypass the MySQL restriction. This is quite common, but at the same time dangerous solution, because it not only reduces performance, but also can potentially cause some queries to work incorrectly; and there is no general answer to the question “will it work or not”: you need to check the specific situation.

MySQL Solution 2.3.5.b

```
1    UPDATE `subscriptions` AS `ext`
2    SET    `sb_finish` = CASE
3                           WHEN `sb_is_active` = 'Y'
4                           AND EXISTS (SELECT `int`.`sb_subscriber`
5                                       FROM   (SELECT `sb_subscriber`,
6                                                      `sb_book`,
7                                                      `sb_is_active`
8                                               FROM   `subscriptions`) AS `int`
9                                       WHERE  `int`.`sb_is_active` = 'Y'
10                                             AND `int`.`sb_subscriber` =
11                                                 `ext`.`sb_subscriber`
12                                      GROUP  BY `int`.`sb_subscriber`
13                                      HAVING COUNT(`int`.`sb_book`) > 2)
14                          THEN (SELECT DATE_ADD(`sb_finish`, INTERVAL 2 MONTH))
15                          ELSE (SELECT DATE_ADD(`sb_finish`, INTERVAL 1 MONTH))
16                        END
```

The composite condition, which is the core of the solution of the problem, is presented in lines 3-13.

Its first part (line 3) is simple: we define the value of the field.

Its second part (lines 4-13) is a correlated subquery, the results of which are passed to the **`EXISTS`** function, i.e., we are interested in the fact whether the subquery returned at least one row or not.

The need for another nested query on lines 5-8 has just been discussed, that way we bypass the MySQL's restriction on reading from the table being updated.

For the rest it is a classic correlated subquery. If we execute it separately, manually substituting the reader identifier value, we get the following picture:

MySQL Solution 2.3.5.b (modified fragment)

```
1    SELECT `int`.`sb_subscriber`
2    FROM   (SELECT `sb_subscriber`,
3                   `sb_book`,
4                   `sb_is_active`
5            FROM   `subscriptions`) AS `int`
6    WHERE  `int`.`sb_is_active` = 'Y'
7           AND `int`.`sb_subscriber` = {reader id from the main query part}
8    GROUP  BY `int`.`sb_subscriber`
9    HAVING COUNT(`int`.`sb_book`) > 2
```

For readers with identifiers 1 and 2 this subquery will return zero lines, and for readers with identifiers 3 and 4 the result will be nonempty.

Lines 14 and 15 describe the desired behavior of the DBMS if the composite condition is met and not met, respectively.

MS SQL Solution 2.3.5.b

```
1    UPDATE [subscriptions]
2    SET    [sb_finish] = CASE
3                           WHEN [sb_is_active] = 'Y'
4                           AND EXISTS (SELECT [int].[sb_subscriber]
5                                       FROM   [subscriptions] AS [int]
6                                       WHERE [int].[sb_is_active] = 'Y'
7                                         AND [int].[sb_subscriber] =
8                                             [subscriptions].[sb_subscriber]
9                                       GROUP  BY [int].[sb_subscriber]
10                                      HAVING COUNT([int].[sb_book]) > 2)
11                          THEN (SELECT DATEADD(month, 2, [sb_finish]))
12                          ELSE (SELECT DATEADD(month, 1, [sb_finish]))
13                        END
```

The solution for MS SQL server is based on the same logic. It is even slightly easier to implement, because MS SQL Server does not forbid updating data in a table and reading data from the same table in the same query, so there is no need to “wrap” the action in line 5 into a subquery.

The Solution for Oracle differs from the solution for MS SQL Server only in the syntax of increasing the date by the desired number of months (lines 11-12). Like MS SQL Server, Oracle does not forbid to update data in a table and read data from the same table in one query.

Oracle | Solution 2.3.5.b

```
1    UPDATE "subscriptions"
2    SET    "sb_finish" = CASE
3                           WHEN "sb_is_active" = 'Y'
4                           AND EXISTS (SELECT "int"."sb_subscriber"
5                                       FROM   "subscriptions" "int"
6                                       WHERE "int"."sb_is_active" = 'Y'
7                                         AND "int"."sb_subscriber" =
8                                             "subscriptions"."sb_subscriber"
9                                       GROUP  BY "int"."sb_subscriber"
10                                      HAVING COUNT("int"."sb_book") > 2)
11                         THEN (SELECT ADD_MONTHS("sb_finish", 2) FROM "DUAL")
12                         ELSE (SELECT ADD_MONTHS("sb_finish", 1) FROM "DUAL")
13                        END
```

Solution 2.3.5.c[213].

In contrast to the previous problems of this example, here the solutions for each DBMS will be fundamentally different. The most general solution is implemented for Oracle (it can also be implemented in both MySQL and MS SQL Server with minimal syntax modifications).

MySQL | Solution 2.3.5.c

```
1    UPDATE `subscribers`
2    SET    `s_name` = CONCAT(`s_name`,
3           (
4            SELECT `postfix`
5            FROM   (SELECT `s_id`,
6                           @x := IFNULL
7                                 (
8                                  (SELECT COUNT(`sb_book`)
9                                   FROM   `subscriptions` AS `int`
10                                  WHERE `int`.`sb_is_active` = 'Y'
11                                        AND `int`.`sb_subscriber` =
12                                            `ext`.`sb_subscriber`
13                                  GROUP  BY `int`.`sb_subscriber`), 0
14                                ),
15                          CASE
16                           WHEN @x > 5 THEN
17                            (SELECT CONCAT(' [', @x, '] [RED]'))
18                           WHEN @x >= 3 AND @x <= 5 THEN
19                            (SELECT CONCAT(' [', @x, '] [YELLOW]'))
20                           ELSE (SELECT CONCAT(' [', @x, '] [GREEN]'))
21                          END AS `postfix`
22                   FROM   `subscribers`
23                   LEFT JOIN `subscriptions` AS `ext`
24                        ON `s_id` = `sb_subscriber`
25                   GROUP  BY `sb_subscriber`) AS `data`
26           WHERE  `data`.`s_id` = `subscribers`.`s_id`)
27          )
```

Let's start with the most deeply nested subquery (lines 6-14). This is a correlated subquery that counts the books in the hands of the reader currently being analyzed by the subquery in lines 5-25. The result of the subquery is placed in the variable `@x`:

s_id	@x
2	0
1	0
3	3
4	4

The `CASE` expression on lines 15-21 uses the value of the `@x` variable to determine the suffix to add to the reader's name:

s_id	@x	postfix
2	0	[0] [GREEN]
1	0	[0] [GREEN]
3	3	[3] [YELLOW]
4	4	[4] [YELLOW]

The correlated subquery on lines 3-27 passes the suffix corresponding to the identifier of the reader whose name is being updated to the `CONCAT` function (a corresponding comparison is made on line 26). The reader's name is replaced by the result of the `CONCAT` function and thus the final result is.

For a better understanding of this solution, here is a modified query that does not update anything, but shows all the necessary information:

MySQL Solution 2.3.5.c (modified query)

```
1    SELECT `s_id`,
2           `s_name`,
3           @x := IFNULL
4                 (
5                  (SELECT COUNT(`sb_book`)
6                   FROM  `subscriptions` AS `int`
7                   WHERE `int`.`sb_is_active` = 'Y'
8                     AND `int`.`sb_subscriber` =
9                         `ext`.`sb_subscriber`
10                  GROUP  BY `int`.`sb_subscriber`), 0
11                 ),
12          CASE
13           WHEN @x > 5 THEN
14            (SELECT CONCAT(' [', @x, '] [RED]'))
15           WHEN @x >= 3 AND @x <= 5 THEN
16            (SELECT CONCAT(' [', @x, '] [YELLOW]'))
17           ELSE (SELECT CONCAT(' [', @x, '] [GREEN]'))
18          END AS `postfix`
19   FROM    `subscribers`
20   LEFT JOIN `subscriptions` AS `ext`
21          ON `s_id` = `sb_subscriber`
22   GROUP  BY `sb_subscriber`
```

The result of running this modified query:

s_id	s_name	@x	postfix
2	Petrov P.P.	0	[0] [GREEN]
1	Ivanov I.I.	0	[0] [GREEN]
3	Sidorov S.S.	3	[3] [YELLOW]
4	Sidorov S.S.	4	[4] [YELLOW]

MS SQL Server does not allow to use variables as flexibly as MySQL, there are restrictions on simultaneous data fetching and variable value changing, as well as requirements on preliminary variable declaration.

However, MS SQL Server supports Common Table Expressions, which is where we transfer the logic to determine how many books are currently in the hands of each reader (lines 1-10). Unlike the MySQL solution, here instead of a correlated subquery we use a subquery as a data source (lines 4-8), which returns information only about books in the hands of readers.

MS SQL | Solution 2.3.5.c

```
1    WITH [prepared_data]
2         AS (SELECT [s_id], COUNT([sb_subscriber]) AS [x]
3             FROM   [subscribers]
4                    LEFT JOIN (
5                              SELECT [sb_subscriber]
6                              FROM [subscriptions]
7                              WHERE [sb_is_active] = 'Y'
8                              ) AS [active_only]
9                           ON [s_id] = [sb_subscriber]
10            GROUP  BY [s_id])
11   UPDATE [subscribers]
12   SET    [s_name] =
13                  (SELECT
14                    CASE
15                     WHEN [x] > 5
16                      THEN (SELECT CONCAT([s_name], ' [', [x], '] [RED]'))
17                     WHEN [x] >= 3 AND [x] <= 5
18                      THEN (SELECT CONCAT([s_name], ' [', [x], '] [YELLOW]'))
19                     ELSE (SELECT CONCAT([s_name], ' [', [x], '] [GREEN]'))
20                    END
21                  FROM   [prepared_data]
22                  WHERE  [subscribers].[s_id] = [prepared_data].[s_id])
```

The result of the Common Table Expression is as follows:

s_id	x
1	0
2	0
3	3
4	4

The correlated subquery on lines 13-22 uses the value of column **`x`** to generate the suffix value of the name of the corresponding reader. This is how the final result is obtained.

If there is a need to get rid of the correlated subquery in lines 13-22, we can rewrite the main part of the solution (the Common Table Expression remains the same) using **`JOIN`**.

MS SQL Solution 2.3.5.c (with JOIN instead of the subquery)

```
1    WITH [prepared_data]
2         AS (SELECT [s_id], COUNT([sb_subscriber]) AS [x]
3             FROM   [subscribers]
4                    LEFT JOIN (
5                              SELECT [sb_subscriber]
6                              FROM [subscriptions]
7                              WHERE [sb_is_active] = 'Y'
8                              ) AS [active_only]
9                           ON [s_id] = [sb_subscriber]
10            GROUP  BY [s_id])
11   UPDATE [subscribers]
12   SET    [s_name] =
13                  (CASE
14                     WHEN [x] > 5
15                      THEN (SELECT CONCAT([s_name], ' [', [x], '] [RED]'))
16                     WHEN [x] >= 3 AND [x] <= 5
17                      THEN (SELECT CONCAT([s_name], ' [', [x], '] [YELLOW]'))
18                     ELSE (SELECT CONCAT([s_name], ' [', [x], '] [GREEN]'))
19                    END)
20   FROM [subscribers]
21   JOIN [prepared_data]
22     ON [subscribers].[s_id] = [prepared_data].[s_id]
```

This solution (using **JOIN** in the context of **UPDATE**) is the most optimal, but also the least accustomed (especially for beginners).

Oracle does not support the just discussed use of **JOIN** in the context of **UPDATE**, so we cannot get rid of the correlated subquery in lines 2-20.

Also, Oracle does not support the use of Common Table Expressions in the **UPDATE** context, which leads to the need to move the data preparation to a subquery (lines 11-19).

Oracle Solution 2.3.5.c

```
1    UPDATE "subscribers"
2    SET    "s_name" =
3     (SELECT
4          CASE
5           WHEN "x" > 5
6            THEN (SELECT "s_name"  || ' [' || "x" || '] [RED]' FROM "DUAL")
7           WHEN "x" >= 3 AND "x" <= 5
8            THEN (SELECT "s_name" || ' [' || "x" || '] [YELLOW]' FROM "DUAL")
9           ELSE (SELECT "s_name" || ' [' || "x" || '] [GREEN]' FROM "DUAL")
10         END
11    FROM   (SELECT "s_id",
12                   COUNT("sb_subscriber") AS "x"
13            FROM   "subscribers"
14            LEFT   JOIN
15                       (SELECT "sb_subscriber"
16                        FROM "subscriptions"
17                        WHERE "sb_is_active" = 'Y')
18                   ON "s_id" = "sb_subscriber"
19            GROUP  BY "s_id") "prepared_data"
20    WHERE  "subscribers"."s_id" = "prepared_data"."s_id")
```

Another limitation of Oracle is that here the **CONCAT** function can only accept two parameters (yet we need to pass four), but we can get around this limitation by using the string concatenation operator **||**.

It was noted earlier that the Oracle solution is the most versatile and can easily be ported to other DBMSes. Let's demonstrate this.

MySQL Solution 2.3.5.c (based on the Oracle solution)

```
UPDATE `subscribers`
SET    `s_name` =
 (SELECT
   CASE
    WHEN `x` > 5 THEN
     (SELECT CONCAT(`s_name`, ' [', `x`, '] [RED]'))
    WHEN `x` >= 3 AND `x` <= 5 THEN
     (SELECT CONCAT(`s_name`, ' [', `x`, '] [YELLOW]'))
   ELSE (SELECT CONCAT(`s_name`, ' [', `x`, '] [GREEN]'))
  END
  FROM   (SELECT `s_id`,
                 COUNT(`sb_subscriber`) AS `x`
          FROM   `subscribers`
          LEFT   JOIN
                    (SELECT `sb_subscriber`
                     FROM `subscriptions`
                     WHERE `sb_is_active` = 'Y') AS `active_only`
                 ON `s_id` = `sb_subscriber`
          GROUP  BY `s_id`) AS `prepared_data`
  WHERE  `subscribers`.`s_id` = `prepared_data`.`s_id`)
```

MS SQL Solution 2.3.5.c (based on the Oracle solution)

```
UPDATE [subscribers]
SET    [s_name] =
 (SELECT
   CASE
    WHEN [x] > 5
     THEN (SELECT CONCAT([s_name], ' [', [x], '] [RED]'))
    WHEN [x] >= 3 AND [x] <= 5
   THEN (SELECT CONCAT([s_name], ' [', [x], '] [YELLOW]'))
    ELSE (SELECT CONCAT([s_name], ' [', [x], '] [GREEN]'))
  END
  FROM   (SELECT [s_id],
                 COUNT([sb_subscriber]) AS [x]
          FROM   [subscribers]
          LEFT   JOIN
                    (SELECT [sb_subscriber]
                     FROM [subscriptions]
                     WHERE [sb_is_active] = 'Y') AS [active_only]
                 ON [s_id] = [sb_subscriber]
          GROUP  BY [s_id]) AS [prepared_data]
  WHERE  [subscribers].[s_id] = [prepared_data].[s_id])
```

Task 2.3.5.TSK.A: add readers with the names “Sidorov S.S.”, “Ivanov I.I.”, “Orlov O.O.” to the database; if a reader with this name already exists, add a sequence number in square brackets to the end of the new reader name (e.g., if adding reader “Sidorov S.S.” finds that the database already contains four such readers, the added reader name should turn into “Sidorov S.S. [5]”)).

Task 2.3.5.TSK.B: update all authors’ names by adding “ [+]” to the end of the name if there are more than three books by that author in the library, or by adding “ [-]” to the end of the name otherwise.

Chapter 3: Using Views

3.1. Using Views to Select Data

3.1.1. Example 26: Using Non-Caching Views to Select Data

Classic views contain no data; they are just a way to access real database tables. An alternative is so-called caching (materialized, indexed) views, which will be discussed in the next section{229}.

Problem 3.1.1.a{223}: simplify the use of the solution of problem 2.2.9.d{140} so that we do not have to use the bulky queries presented in the solution{149} to get the necessary data.

Problem 3.1.1.b{227}: create a view to get a list of authors and the number of books available in the library for each author, but displays only those authors for whom there is more than one book.

Expected result 3.1.1.a.

Executing a `SELECT * FROM {view}` query gives the expected result for problem 2.2.9.d{140}, i.e.:

s_id	s_name	b_name
1	Ivanov I.I.	Eugene Onegin
3	Sidorov S.S.	Foundation and Empire
4	Sidorov S.S.	The C++ Programming Language

Expected result 3.1.1.b.

Executing a `SELECT * FROM {view}` query will produce the following result (under no circumstances should authors with less than two books registered in the library be displayed here):

a_id	a_name	books_in_library
6	Bjarne Stroustrup	2
7	Alexander Pushkin	2

Solution 3.1.1.a{223}.

Let's create a solution of this problem for MySQL based on the following query already written before (see solution 2.2.9.d{149}):

MySQL Solution 3.1.1.a (the original query, which must be “hidden” in the view)

```
SELECT `s_id`,
       `s_name`,
       `b_name`
FROM   (SELECT `subscriptions`.`sb_subscriber`,
               `sb_book`
        FROM   `subscriptions`
               JOIN (SELECT `subscriptions`.`sb_subscriber`,
                            MIN(`sb_id`) AS `min_sb_id`
                     FROM   `subscriptions`
                            JOIN (SELECT `sb_subscriber`,
                                         MIN(`sb_start`) AS `min_sb_start`
                                  FROM   `subscriptions`
                                  GROUP  BY `sb_subscriber`)
                                  AS `step_1`
                              ON `subscriptions`.`sb_subscriber` =
                                 `step_1`.`sb_subscriber`
                                 AND `subscriptions`.`sb_start` =
                                     `step_1`.`min_sb_start`
                     GROUP  BY `subscriptions`.`sb_subscriber`,
                               `min_sb_start`)
                     AS `step_2`
                 ON `subscriptions`.`sb_id` = `step_2`.`min_sb_id`)
        AS `step_3`
       JOIN `subscribers`
         ON `sb_subscriber` = `s_id`
       JOIN `books`
         ON `sb_book` = `b_id`
```

Ideally, we would just like to build a view on this query. But the ability to do this depends on the version of MySQL we are working with.

MySQL versions before 5.7.7 do not allow us to create views that rely on queries with subqueries in the `FROM` section. Unfortunately, we have as many as three such subqueries — `` `step_1` ``, `` `step_2` ``, `` `step_3` ``.

To work around this limitation, there is a not very elegant but very simple solution: build a separate view for each of these subqueries. Then `FROM` section will not refer to the subquery (which is forbidden), but to the view (which is allowed).

MySQL Solution 3.1.1.a (before version 5.7.7)

```
-- Replacing the first subquery with a view:
CREATE OR REPLACE VIEW `first_book_step_1`
AS
  SELECT `sb_subscriber`,
         MIN(`sb_start`) AS `min_sb_start`
  FROM   `subscriptions`
  GROUP  BY `sb_subscriber`

-- Replacing the second subquery with a view:
CREATE OR REPLACE VIEW `first_book_step_2`
AS
  SELECT `subscriptions`.`sb_subscriber`,
         MIN(`sb_id`) AS `min_sb_id`
  FROM   `subscriptions`
         JOIN `first_book_step_1`
           ON `subscriptions`.`sb_subscriber` =
              `first_book_step_1`.`sb_subscriber`
              AND `subscriptions`.`sb_start` =
                  `first_book_step_1`.`min_sb_start`
  GROUP  BY `subscriptions`.`sb_subscriber`,
            `min_sb_start`
```

MySQL Solution 3.1.1.a (before version 5.7.7)

```
22  -- Replacing the third subquery with a view:
23  CREATE OR REPLACE VIEW `first_book_step_3`
24  AS
25    SELECT `subscriptions`.`sb_subscriber`,
26           `sb_book`
27    FROM   `subscriptions`
28           JOIN `first_book_step_2`
29             ON `subscriptions`.`sb_id` = `first_book_step_2`.`min_sb_id`
30
31  -- Creating the main view:
32  CREATE OR REPLACE VIEW `first_book`
33  AS
34    SELECT `s_id`,
35           `s_name`,
36           `b_name`
37    FROM   `subscribers`
38           JOIN `first_book_step_3`
39             ON `sb_subscriber` = `s_id`
40           JOIN `books`
41             ON `sb_book` = `b_id`
```

Note that each successive view in this set builds on the previous one.

If we use MySQL version 5.7.7 or newer, we can immediately “directly” build a view on the original query.

MySQL Solution 3.1.1.a (version 5.7.7 and newer)

```
1   CREATE OR REPLACE VIEW `first_book`
2   AS
3    SELECT `s_id`,
4           `s_name`,
5           `b_name`
6    FROM   (SELECT `subscriptions`.`sb_subscriber`,
7                   `sb_book`
8            FROM   `subscriptions`
9                   JOIN (SELECT `subscriptions`.`sb_subscriber`,
10                               MIN(`sb_id`) AS `min_sb_id`
11                       FROM   `subscriptions`
12                              JOIN (SELECT `sb_subscriber`,
13                                           MIN(`sb_start`) AS `min_sb_start`
14                                    FROM   `subscriptions`
15                                    GROUP  BY `sb_subscriber`)
16                                    AS `step_1`
17                                ON `subscriptions`.`sb_subscriber` =
18                                   `step_1`.`sb_subscriber`
19                                   AND `subscriptions`.`sb_start` =
20                                       `step_1`.`min_sb_start`
21                       GROUP  BY `subscriptions`.`sb_subscriber`,
22                                 `min_sb_start`)
23                       AS `step_2`
24                    ON `subscriptions`.`sb_id` = `step_2`.`min_sb_id`)
25           AS `step_3`
26          JOIN `subscribers`
27            ON `sb_subscriber` = `s_id`
28          JOIN `books`
29            ON `sb_book` = `b_id`
```

Now to get the data it is enough to execute `SELECT * FROM `first_book`` query, which was required by the problem condition.

Solution for MS SQL Server is also based on the previously written query (see solution 2.2.9.d{149}):

MS SQL — Solution 3.1.1.a (the original query, which must be "hidden" in the view)

```
WITH [step_1]
     AS (SELECT [sb_subscriber],
                MIN([sb_start]) AS [min_sb_start]
         FROM   [subscriptions]
         GROUP  BY [sb_subscriber]),
     [step_2]
     AS (SELECT [subscriptions].[sb_subscriber],
                MIN([sb_id]) AS [min_sb_id]
         FROM   [subscriptions]
                JOIN [step_1]
                  ON [subscriptions].[sb_subscriber] =
                             [step_1].[sb_subscriber]
                     AND [subscriptions].[sb_start] =
                             [step_1].[min_sb_start]
         GROUP  BY [subscriptions].[sb_subscriber],
                   [min_sb_start]),
     [step_3]
     AS (SELECT [subscriptions].[sb_subscriber],
                [sb_book]
         FROM   [subscriptions]
                JOIN [step_2]
                  ON [subscriptions].[sb_id] = [step_2].[min_sb_id])
SELECT [s_id],
       [s_name],
       [b_name]
FROM   [step_3]
       JOIN [subscribers]
         ON [sb_subscriber] = [s_id]
       JOIN [books]
         ON [sb_book] = [b_id]
```

Since even quite "old" versions of MS SQL Server do not have MySQL-specific restrictions on queries on which the view is based, the final solution to the problem is obtained by adding one line to the beginning of the query:

MS SQL — Solution 3.1.1.a

```
CREATE OR ALTER VIEW [first_book] AS
{the text of the original query, which we "hide" in the view}
```

Now to get the data it is enough to execute `SELECT * FROM [first_book]` query, which was required by the problem condition.

Solution for Oracle also builds on the previously written query (see solution 2.2.9.d{149}):

Oracle Solution 3.1.1.a (the original query, which must be “hidden” in the view)

```
1    WITH "step_1"
2         AS (SELECT "sb_subscriber",
3                    MIN("sb_start") AS "min_sb_start"
4             FROM   "subscriptions"
5             GROUP  BY "sb_subscriber"),
6         "step_2"
7         AS (SELECT "subscriptions"."sb_subscriber",
8                    MIN("sb_id") AS "min_sb_id"
9             FROM   "subscriptions"
10                   JOIN "step_1"
11                     ON "subscriptions"."sb_subscriber" =
12                                "step_1"."sb_subscriber"
13                        AND "subscriptions"."sb_start" =
14                                "step_1"."min_sb_start"
15            GROUP  BY "subscriptions"."sb_subscriber",
16                      "min_sb_start"),
17        "step_3"
18        AS (SELECT "subscriptions"."sb_subscriber",
19                   "sb_book"
20            FROM   "subscriptions"
21                   JOIN "step_2"
22                     ON "subscriptions"."sb_id" = "step_2"."min_sb_id")
23   SELECT "s_id",
24          "s_name",
25          "b_name"
26   FROM   "step_3"
27          JOIN "subscribers"
28            ON "sb_subscriber" = "s_id"
29          JOIN "books"
30            ON "sb_book" = "b_id"
```

Since Oracle, like MS SQL Server, has no MySQL-specific restrictions on queries that are used to build a view, the final solution of the problem is obtained by adding one line to the beginning of the query:

Oracle Solution 3.1.1.a

```
1    CREATE OR REPLACE VIEW "first_book" AS
2    {the text of the original query, which we “hide” in the view}
```

Now to get the data it is enough to execute `SELECT * FROM "first_book"` query, which was required by the problem condition.

Solution 3.1.1.b[{223}].

The solution of this problem is identical in all three DBMS and boils down to writing a query that displays the data about the authors and the number of their books in the library, taking into account the condition specified in the problem: there must be more than one such book. The resulting query is then used to build a view.

The view name in Oracle cannot exceed 30 characters, so the word `with` in the view name had to be shortened to a single letter `w`.

MySQL Solution 3.1.1.b

```
1     CREATE OR REPLACE VIEW `authors_with_more_than_one_book`
2     AS
3       SELECT `a_id`,
4              `a_name`,
5              COUNT(`b_id`) AS `books_in_library`
6       FROM   `authors`
7              JOIN `m2m_books_authors` USING (`a_id`)
8       GROUP  BY `a_id`
9       HAVING `books_in_library` > 1
```

MS SQL Solution 3.1.1.b

```
1     CREATE OR ALTER VIEW [authors_with_more_than_one_book]
2     AS
3       SELECT [authors].[a_id],
4              [a_name],
5              COUNT([b_id]) AS [books_in_library]
6       FROM   [authors]
7              JOIN [m2m_books_authors]
8                ON [authors].[a_id] = [m2m_books_authors].[a_id]
9       GROUP  BY [authors].[a_id],
10                [a_name]
11      HAVING COUNT([b_id]) > 1
```

Oracle Solution 3.1.1.b

```
1     CREATE OR REPLACE VIEW "authors_w_more_than_one_book"
2     AS
3       SELECT "a_id",
4              "a_name",
5              COUNT("b_id") AS "books_in_library"
6       FROM   "authors"
7              JOIN "m2m_books_authors" USING ("a_id")
8       GROUP  BY "a_id", "a_name"
9       HAVING COUNT("b_id") > 1
```

Task 3.1.1.TSK.A: simplify the use of the solution of problem 2.2.8.b{135} so that we do not have to use the bulky queries presented in the solution{137} to get the necessary data.

Task 3.1.1.TSK.B: create a view to get a list of readers with the number of books in the hands of each reader, but displays only those readers for whom there are debts, i.e., the reader has at least one book in their hands, which they must have returned before the current date.

Task 3.1.1.TSK.C: rewrite solution 3.1.1.a{223} for MySQL version 8 or newer (using ranking and Common Table Expressions).

3.1.2. Example 27: Using Caching Views and Tables to Select Data

Caching (materialized, indexed) views, unlike their classical counterparts{223} form and store a separate prepared data set. Since not all DBMSes support such views and/or impose some serious restrictions on them, their counterpart can be implemented with the help of triggers{286} and caching or aggregating tables.

The use of solutions such as those presented in this example, in real life can have a completely unpredictable effect on productivity, both to increase it dramatically and to reduce it very much. Each case requires its own separate study. Therefore, the problems and their solutions presented below should be taken only as a demonstration of DBMS capabilities, not as a direct guide to action.

Problem 3.1.2.a{230}: create a view that speeds up obtaining information about the number of copies of books: available in the library, taken by readers, remaining in the library.

Problem 3.1.2.b{246}: create a view that speeds up obtaining all the information from the `subscriptions` table in human-readable form (where readers' and books' identifiers are replaced by names and titles respectively).

Expected result 3.1.2.a.

Executing a `SELECT * FROM {view}` query gives the following result:

total	given	rest
33	5	28

Expected result 3.1.2.b.

Executing a `SELECT * FROM {view}` query gives the following result:

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
2	Ivanov I.I.	Eugene Onegin	2011-01-12	2011-02-12	N
42	Ivanov I.I.	The Fisherman and the Golden Fish	2012-06-11	2012-08-11	N
61	Ivanov I.I.	The Art of Computer Programming	2014-08-03	2014-10-03	N
95	Ivanov I.I.	Programming Psychology	2015-10-07	2015-11-07	N
100	Ivanov I.I.	Foundation and Empire	2011-01-12	2011-02-12	N
3	Sidorov S.S.	Foundation and Empire	2012-05-17	2012-07-17	Y
62	Sidorov S.S.	The C++ Programming Language	2014-08-03	2014-10-03	Y
86	Sidorov S.S.	Eugene Onegin	2014-08-03	2014-09-03	Y
57	Sidorov S.S.	The C++ Programming Language	2012-06-11	2012-08-11	N
91	Sidorov S.S.	Eugene Onegin	2015-10-07	2015-03-07	Y
99	Sidorov S.S.	Programming Psychology	2015-10-08	2025-11-08	Y

Solution 3.1.2.a{229}.

Traditionally, we will start with the first solution for MySQL, and there is a problem: MySQL (even version 8) does not support so-called “caching views”. The only way to achieve the desired result in this DBMS is to create a real table that stores the data we need.

This data will have to be updated, and different approaches can be used for this:

- single filling (for the case when the original data does not change);
- periodic updating, e.g., by means of a stored procedure (suitable for the case when we can afford periodically getting information that is not the most up-to-date);
- automatic updating with the help of triggers (allows to get actual information at any moment of time).

We will implement the latter option, i.e., working through triggers (see details about triggers in the corresponding section{286}). This approach also decomposes into two possible solutions: triggers can update all data every time or react only to incoming changes (which is much faster but requires initial initialization of data in the aggregating / caching table).

So, we will implement the most productive (albeit the most complicated) option: create an aggregating table, write a query to initialize its data and create triggers that react to changes in the aggregated data.

Create an aggregating table:

MySQL Solution 3.1.2.a (creating an aggregating table)

```
CREATE TABLE `books_statistics`
(
        `total` INTEGER UNSIGNED NOT NULL,
        `given` INTEGER UNSIGNED NOT NULL,
        `rest`  INTEGER UNSIGNED NOT NULL
)
```

It is easy to see that this table has no primary key. It is not needed, because it is supposed to store exactly one row.

In real-world applications, there must be a mechanism to react when either zero rows or more than one row is found in such tables. Both of these situations can potentially cause the application to crash or malfunction.

Initialize the data in the created table:

MySQL Solution 3.1.2.a (cleaning the table and data initialization)

```
-- Cleaning the table:
TRUNCATE TABLE `books_statistics`;

-- Data initialization:
INSERT INTO `books_statistics`
            (`total`,
             `given`,
             `rest`)
SELECT IFNULL(`total`, 0),
       IFNULL(`given`, 0),
       IFNULL(`total` - `given`, 0) AS `rest`
FROM   (SELECT (SELECT SUM(`b_quantity`)
                FROM   `books`)              AS `total`,
               (SELECT COUNT(`sb_book`)
                FROM   `subscriptions`
                WHERE  `sb_is_active` = 'Y') AS `given`)
       AS `prepared_data`;
```

Let's write triggers that modify the data in the aggregating table. The aggregating is based on the information presented in the **`books`** and **`subscriptions`** tables, so we have to create triggers for both of these tables.

Changes in the **`books`** table affect the **`total`** and **`rest`** fields, and changes in the **`subscriptions`** table affect the **`given`** and **`rest`** fields. Data can change as a result of all three data modification operations (insert, delete, update), so, we have to create triggers for all three operations.

Important: in MySQL (even in version 8) triggers are not activated by cascade operations, so changes in **`subscriptions`** table, caused by deleting subscribers, will remain "invisible" for triggers on this table. In task 3.1.2.TSK.D{254} you are invited to improve this solution, eliminating this problem.

MySQL Solution 3.1.2.a (triggers for the **`books`** table)

```
-- Deleting old versions of triggers
-- (convenient during development and debugging):
DROP TRIGGER `upd_bks_sts_on_books_ins`;
DROP TRIGGER `upd_bks_sts_on_books_del`;
DROP TRIGGER `upd_bks_sts_on_books_upd`;

-- Switching the query completion delimiter,
-- because now the query will create a trigger,
-- inside which there are their own, classic queries:
DELIMITER $$

-- Creating a trigger that reacts to adding books:
CREATE TRIGGER `upd_bks_sts_on_books_ins`
BEFORE INSERT
ON `books`
 FOR EACH ROW
  BEGIN
   UPDATE `books_statistics` SET
          `total` = `total` + NEW.`b_quantity`,
          `rest` = `total` - `given`;
  END;
$$
```

MySQL Solution 3.1.2.a (triggers for the `books` table) (continued)

```
23  -- Creating a trigger that reacts to the deletion of books:
24  CREATE TRIGGER `upd_bks_sts_on_books_del`
25  BEFORE DELETE
26  ON `books`
27   FOR EACH ROW
28    BEGIN
29     UPDATE `books_statistics` SET
30            `total` = `total` - OLD.`b_quantity`,
31            `given` = `given` - (SELECT COUNT(`sb_book`)
32                                 FROM   `subscriptions`
33                                 WHERE  `sb_book`=OLD.`b_id`
34                                    AND `sb_is_active` = 'Y'),
35            `rest` = `total` - `given`;
36    END;
37  $$
38
39  -- Creating a trigger that reacts to
40  -- the change in the number of books:
42  CREATE TRIGGER `upd_bks_sts_on_books_upd`
42  BEFORE UPDATE
43  ON `books`
44   FOR EACH ROW
45    BEGIN
46     UPDATE `books_statistics` SET
47            `total` = `total` - OLD.`b_quantity` + NEW.`b_quantity`,
48            `rest` = `total` - `given`;
49    END;
50  $$
51
52  -- Restoring the query completion delimiter:
53  DELIMITER ;
```

Switching the delimiter of query completion (lines 7-10) is necessary to prevent MySQL from treating the `;` characters occurring at the end of queries within a trigger as the end of the trigger creation query itself. After all trigger creation operations completed, this delimiter returns to the initial state (line 53).

The **`FOR EACH ROW`** statement (lines 16, 27, 44) means that the body of the trigger will be executed for every record (triggers do not work any other way in MySQL) affected by the table operation (several records can be added, changed and deleted at a time).

The keywords **`NEW`** and **`OLD`** allow us to refer:

- when inserting, with **`NEW`** to the new (added) data;
- when updating, with **`OLD`** to the old data values and with **`NEW`** to the new data values;
- when deleting, with **`OLD`** to the values of the data to be deleted.

MySQL (unlike MS SQL Server) calculates new table field values “on the spot”, which allows us to perform all the necessary actions in all three triggers with a single query and use the expression **`` `rest` = `total` - `given` ``**, because **`` `total` ``** and/or **`` `given` ``** values are already updated by previously encountered commands in queries. In MS SQL Server, however, we have to run a separate query to calculate the **`` `rest` ``** value, because the **`` `total` ``** and/or **`` `given` ``** values do not change until the first query is finished.

In the triggers on the **books** table, the queries themselves (lines 18-20, 29-35, 46-48) are perfectly trivial, and their only unfamiliarity lies in the use of the **OLD** and **NEW** keywords, which we have just reviewed.

MySQL Solution 3.1.2.a (triggers for the **subscriptions** table)

```
1    -- Deleting old versions of triggers
2    -- (convenient during development and debugging):
3    DROP TRIGGER `upd_bks_sts_on_subscriptions_ins`;
4    DROP TRIGGER `upd_bks_sts_on_subscriptions_del`;
5    DROP TRIGGER `upd_bks_sts_on_subscriptions_upd`;
6
7
8    -- Switching the query completion delimiter,
9    -- because now the query will create a trigger,
10   -- inside which there are their own, classic queries:
11   DELIMITER $$
12
13   -- Creating a trigger that reacts to the addition of a subscription:
14   CREATE TRIGGER `upd_bks_sts_on_subscriptions_ins`
15   BEFORE INSERT
16   ON `subscriptions`
17    FOR EACH ROW
18     BEGIN
19
20      SET @delta = 0;
21
22      IF (NEW.`sb_is_active` = 'Y') THEN
23       SET @delta = 1;
24      END IF;
25
26      UPDATE `books_statistics` SET
27             `rest` = `rest` - @delta,
28             `given` = `given` + @delta;
29     END;
30   $$
31
32   -- Creating a trigger that reacts to the deletion of a subscription:
33   CREATE TRIGGER `upd_bks_sts_on_subscriptions_del`
34   BEFORE DELETE
35   ON `subscriptions`
36    FOR EACH ROW
37     BEGIN
38
39      SET @delta = 0;
40
41      IF (OLD.`sb_is_active` = 'Y') THEN
42       SET @delta = 1;
43      END IF;
44
45      UPDATE `books_statistics` SET
46             `rest` = `rest` + @delta,
47             `given` = `given` - @delta;
48     END;
49   $$
```

MySQL Solution 3.1.2.a (triggers for the `subscriptions` table) (continued)

```
50  -- Creating a trigger that reacts to the update of subscription:
51  CREATE TRIGGER `upd_bks_sts_on_subscriptions_upd`
52  BEFORE UPDATE
53  ON `subscriptions`
54   FOR EACH ROW
55    BEGIN
56     SET @delta = 0;
57
58     IF ((NEW.`sb_is_active` = 'Y') AND (OLD.`sb_is_active` = 'N')) THEN
59      SET @delta = -1;
60     END IF;
61
62     IF ((NEW.`sb_is_active` = 'N') AND (OLD.`sb_is_active` = 'Y')) THEN
63      SET @delta = 1;
64     END IF;
65
66     UPDATE `books_statistics` SET
67            `rest` = `rest` + @delta,
68            `given` = `given` - @delta;
69     END;
70  $$
71
72  -- Restoring the query completion delimiter:
73  DELIMITER ;
```

The triggers on the `subscriptions` table are a bit more complicated than those on the `books` table: here we have to analyze what is happening and take actions depending on the situation.

In the trigger that reacts to the addition of a subscription (lines 13-30), we need to change the values of `` `rest` `` and `` `given` `` only if the book in the added subscription is marked as being in the reader's hand. At first, we assume that this is not the case, so we initialize `@delta` variable (line 20) to `0`. If it turns out that the book is in hand, we change it to `1` (lines 22-24). So, in the query on lines 26-28, the values in the aggregating table will change to `0` (i.e., not changed) or `1`, depending on whether the book has been given to the reader.

We use exactly the same logic in the trigger that reacts to the deletion of a subscription (lines 32-49).

In the trigger that reacts to an update to the subscription, we need to consider four cases (of which we are really only interested in the last two):

- the book was in the reader's hands and remains there (the value of `` `sb_is_active` `` was `Y` and remains the same);
- the book was not in the reader's hands and remains the same (the value of `` `sb_is_active` `` was `N` and remains the same);
- the book was in the reader's hands, and they returned it (the value of `` `sb_is_active` `` was `Y`, but changed to `N`, see lines 58-60 of the query);
- the book was not in the reader's hands, but they took it (the value of `` `sb_is_active` `` was `N`, but changed to `Y`, see lines 62-64 of the query).

Obviously, the number of books delivered and remaining in the library changes only in the latter two cases, which are accounted for in the conditions presented in lines 58-64. The query in lines 66-68 uses the value of the variable `@delta`, modified by these conditions, to modify the aggregate data.

Let's check how what we have created works. We will modify data in the `books` and `subscriptions` tables and select data from the `books_statistics` table.

Let's add two books with the number of copies of 5 and 10:

MySQL Solution 3.1.2.a (checking the reaction to adding books)

```
INSERT INTO `books`
            (`b_id`,
             `b_name`,
             `b_quantity`,
             `b_year`)
VALUES      (NULL,
             'New book 1',
             5,
             2001),
            (NULL,
             'New book 2',
             10,
             2002)
```

	total	given	rest
Before	33	5	28
After	48	5	43

Let's increase by five the number of copies of the book, which is now registered in the library of 10 copies (we have one such book):

MySQL Solution 3.1.2.a (checking the reaction to changes in the book quantity)

```
UPDATE `books`
SET    `b_quantity` = `b_quantity` + 5
WHERE  `b_quantity` = 10
```

	total	given	rest
Before	48	5	43
After	53	5	48

Let's delete the book, both copies of which are now in the hands of readers (book with identifier 1).

MySQL Solution 3.1.2.a (checking the reaction to book deletion)

```
DELETE FROM `books`
WHERE  `b_id` = 1
```

	total	given	rest
Before	53	5	48
After	51	3	48

Let's mark that on subscription with the identifier 3 the book was returned:

MySQL Solution 3.1.2.a (checking the reaction to a book return)

```
UPDATE `subscriptions`
SET    `sb_is_active` = 'N'
WHERE  `sb_id` = 3
```

	total	given	rest
Before	51	3	48
After	51	2	49

Let's cancel this operation (mark the book as unreturned again):

MySQL Solution 3.1.2.a (checking the reaction to cancellation of the book return)

```
UPDATE `subscriptions`
SET    `sb_is_active` = 'Y'
WHERE  `sb_id` = 3
```

	total	given	rest
Before	51	2	49
After	51	3	48

Let's add to the database the information that the reader with identifier 2 took books with identifiers 5 and 6 from the library:

MySQL Solution 3.1.2.a (checking the reaction to the delivery of books)

```
INSERT INTO `subscriptions`
                (`sb_id`,
                 `sb_subscriber`,
                 `sb_book`,
                 `sb_start`,
                 `sb_finish`,
                 `sb_is_active`)
VALUES          (NULL,
                 2,
                 5,
                 '2016-01-10',
                 '2016-02-10',
                 'Y'),
                (NULL,
                 2,
                 6,
                 '2016-01-10',
                 '2016-02-10',
                 'Y')
```

	total	given	rest
Before	51	3	48
After	51	5	46

Let's delete the information about the subscription with identifier 42 (the book on this subscription has already been returned):

MySQL Solution 3.1.2.a (checking the reaction to the deletion of a subscription with a returned book)

```
DELETE FROM `subscriptions`
WHERE  `sb_id` = 42
```

	total	given	rest
Before	51	5	46
After	51	5	46

Let's delete the information about the subscription with identifier 62 (the book on this subscription has not been returned yet):

MySQL Solution 3.1.2.a (checking the reaction to the deletion of a subscription with an unreturned book)

```
DELETE FROM `subscriptions`
WHERE  `sb_id` = 62
```

	total	given	rest
Before	51	5	46
After	51	4	47

Finally, let's delete all books (which will also cascade delete all subscriptions):

MySQL Solution 3.1.2.a (check the reaction to the deletion of all books)

```
1    DELETE FROM `books`
```

	total	given	rest
Before	51	4	47
After	0	0	0

So, all data modification operations in the **books** and **subscriptions** tables cause corresponding changes in the aggregating **books_statistics** table, which in MySQL acts as a caching view.

Let's move on to MS SQL Server. Theoretically, everything should be fine here, since this DBMS supports so-called indexed views, but if we carefully study the list of limitations[11], we will come to a disappointing conclusion: we have to go the way of MySQL and create an aggregating table and triggers.

Let's create an aggregating table:

MS SQL Solution 3.1.2.a (creating an aggregating table)

```
1    CREATE TABLE [books_statistics]
2    (
3           [total] INTEGER NOT NULL,
4           [given] INTEGER NOT NULL,
5           [rest]  INTEGER NOT NULL
6    )
```

Now, we have to initialize the data in the created table:

MS SQL Solution 3.1.2.a (cleaning the table and data initialization)

```
1    -- Cleaning the table:
2    TRUNCATE TABLE [books_statistics];
3
4    -- Data initialization:
5    INSERT INTO [books_statistics]
6                ([total],
7                 [given],
8                 [rest])
9    SELECT ISNULL([total], 0) AS [total],
10          ISNULL([given], 0) AS [given],
11          ISNULL([total] - [given], 0) AS [rest]
12   FROM   (SELECT (SELECT SUM([b_quantity])
13                   FROM   [books])              AS [total],
14                  (SELECT COUNT([sb_book])
15                   FROM   [subscriptions]
16                   WHERE  [sb_is_active] = 'Y') AS [given])
17          AS [prepared_data];
```

So far, everything has been completely identical to MySQL, but the internal logic of MS SQL Server triggers is completely different, although we still have to create triggers on all three operations (insert, update, delete) for both tables (**books** and **subscriptions**).

[11] https://docs.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views

MS SQL Solution 3.1.2.a (triggers for the `books` table)

```
1    -- Deleting old versions of triggers
2    -- (convenient during development and debugging):
3    DROP TRIGGER [upd_bks_sts_on_books_ins];
4    DROP TRIGGER [upd_bks_sts_on_books_del];
5    DROP TRIGGER [upd_bks_sts_on_books_upd];
6    GO
7
8    -- Creating a trigger that reacts to adding books:
9    CREATE TRIGGER [upd_bks_sts_on_books_ins]
10   ON [books]
11   AFTER INSERT
12   AS
13     UPDATE [books_statistics] SET
14            [total] = [total]  + (SELECT SUM([b_quantity])
15                                  FROM   [inserted]);
16     UPDATE [books_statistics] SET
17            [rest] = [total] - [given];
18   GO
19
20   -- Creating a trigger that reacts to deleting books:
21   CREATE TRIGGER [upd_bks_sts_on_books_del]
22   ON [books]
23   AFTER DELETE
24   AS
25     UPDATE [books_statistics] SET
26            [total] = [total] - (SELECT SUM([b_quantity])
27                                 FROM   [deleted]),
28            [given] = [given] - (SELECT COUNT([sb_book])
29                                 FROM   [subscriptions]
30                                 WHERE  [sb_book] IN (SELECT [b_id]
31                                                      FROM   [deleted])
32                                    AND [sb_is_active] = 'Y');
33     UPDATE [books_statistics] SET
34            [rest] = [total] - [given];
35   GO
36
37   -- Creating a trigger that reacts to
38   -- changing the number of books:
39   CREATE TRIGGER [upd_bks_sts_on_books_upd]
40   ON [books]
42   AFTER UPDATE
42   AS
43     UPDATE [books_statistics] SET
44            [total] = [total] - (SELECT SUM([b_quantity])
45                                 FROM   [deleted]) + (SELECT SUM([b_quantity])
46                                                      FROM   [inserted]);
47     UPDATE [books_statistics] SET
48            [rest] = [total] - [given];
49   GO
```

There are two main differences from the MySQL solution:

- Trigger body is not executed for each row of modified data (as it is in MySQL), but once for the whole data set; hence, it is not an appeal to an individual field with `OLD` and `NEW` keywords, but rather to work with "pseudo-tables" `[deleted]` (contains information about rows to be deleted and old data about rows to be updated) and `[inserted]` (contains information about rows to be added and new data about rows to be updated)
- MS SQL Server does not allow us to modify field values in one query and use their new values immediately, that's why all three triggers use a separate query to calculate the `[rest]` field value.

In other respects, the behavior of triggers in MS SQL Server on the **`books`** table is identical to the behavior of the corresponding triggers in MySQL. But the triggers on the **`subscriptions`** table have significant differences.

MS SQL Solution 3.1.2.a (triggers for the **`subscriptions`** table)

```
-- Deleting old versions of triggers
-- (convenient during development and debugging):
DROP TRIGGER [upd_bks_sts_on_subscriptions_ins];
DROP TRIGGER [upd_bks_sts_on_subscriptions_del];
DROP TRIGGER [upd_bks_sts_on_subscriptions_upd];
GO

-- Creating a trigger that reacts to the addition of a subscription:
CREATE TRIGGER [upd_bks_sts_on_subscriptions_ins]
ON [subscriptions]
AFTER INSERT
AS
 DECLARE @delta INT = (SELECT COUNT(*)
                       FROM   [inserted]
                       WHERE  [sb_is_active] = 'Y');
 UPDATE [books_statistics] SET
        [rest] = [rest] - @delta,
        [given] = [given] + @delta;
GO

-- Creating a trigger that reacts to the deletion of a subscription:
CREATE TRIGGER [upd_bks_sts_on_subscriptions_del]
ON [subscriptions]
AFTER DELETE
AS
 DECLARE @delta INT = (SELECT COUNT(*)
                       FROM   [deleted]
                       WHERE  [sb_is_active] = 'Y');
 UPDATE [books_statistics] SET
        [rest] = [rest] + @delta,
        [given] = [given] - @delta;
GO

-- Creating a trigger that reacts to the update of a subscription:
CREATE TRIGGER [upd_bks_sts_on_subscriptions_upd]
ON [subscriptions]
AFTER UPDATE
AS
 DECLARE @taken INT = (
  SELECT COUNT(*)
  FROM   [inserted]
         JOIN [deleted]
           ON [inserted].[sb_id] = [deleted].[sb_id]
  WHERE  [inserted].[sb_is_active] = 'Y'
     AND [deleted].[sb_is_active] = 'N');

 DECLARE @returned INT = (
  SELECT COUNT(*)
  FROM   [inserted]
         JOIN [deleted]
           ON [inserted].[sb_id] = [deleted].[sb_id]
  WHERE  [inserted].[sb_is_active] = 'N'
     AND [deleted].[sb_is_active] = 'Y');

 DECLARE @delta INT = @taken - @returned;

 UPDATE [books_statistics] SET
        [rest] = [rest] - @delta,
        [given] = [given] + @delta;
GO
```

In MySQL, we calculated the value of `@delta` variable, which could become `0`, `1`, or `-1`, depending on how the change in the analyzed row should affect the data in the aggregating table.

In MS SQL Server, we need to implement a reaction to change of not a single line, but the entire set of modified lines. That's why in lines 13-15 and 26-28 the value of `@delta` variable is defined as the number of records satisfying the trigger condition.

In lines 39-55 this approach becomes even more complicated: we have to determine the number of books delivered (lines 39-45) and books returned (lines 47-53), and then in line 55 we can determine the difference between the numbers obtained and use its value (lines 57-59) to change the data in the aggregating table.

Again (as with MySQL), let's check how what we have created works. We'll modify the data in the `books` and `subscriptions` tables and select data from the `books_statistics` table.

Let's add two books with the number of copies of 5 and 10:

MS SQL Solution 3.1.2.a (checking the reaction to adding books)

```
1     INSERT INTO [books]
2                 ([b_name],
3                  [b_quantity],
4                  [b_year])
5     VALUES      (N'New book 1',
6                  5,
7                  2001),
8                 (N'New book 2',
9                  10,
10                 2002)
```

	total	given	rest
Before	33	5	28
After	48	5	43

Let's increase by five the number of copies of the book, which is now registered in the library of 10 copies (we have one such book):

MS SQL Solution 3.1.2.a (checking the reaction to changes in the book quantity)

```
1     UPDATE [books]
2     SET    [b_quantity] = [b_quantity] + 5
3     WHERE  [b_quantity] = 10
```

	total	given	rest
Before	48	5	43
After	53	5	48

Let's delete the book, both copies of which are now in the hands of readers (book with identifier 1).

MS SQL Solution 3.1.2.a (checking the reaction to book deletion)

```
1     DELETE FROM [books]
2     WHERE  [b_id] = 1
```

	total	given	rest
Before	53	5	48
After	51	3	48

Let's mark that on subscription with the identifier 3 the book was returned:

MS SQL Solution 3.1.2.a (checking the reaction to a book return)

```
1    UPDATE [subscriptions]
2    SET    [sb_is_active] = 'N'
3    WHERE  [sb_id] = 3
```

	total	given	rest
Before	51	3	48
After	51	2	49

Let's cancel this operation (mark the book as unreturned again):

MS SQL Solution 3.1.2.a (checking the reaction to cancellation of the book return)

```
1    UPDATE [subscriptions]
2    SET    [sb_is_active] = 'Y'
3    WHERE  [sb_id] = 3
```

	total	given	rest
Before	51	2	49
After	51	3	48

Let's add to the database the information that the reader with identifier 2 took books with identifiers 5 and 6 from the library:

MS SQL Solution 3.1.2.a (checking the reaction to the delivery of books)

```
1    INSERT INTO [subscriptions]
2                ([sb_subscriber],
3                 [sb_book],
4                 [sb_start],
5                 [sb_finish],
6                 [sb_is_active])
7    VALUES      (2,
8                 5,
9                 CAST(N'2016-01-10' AS DATE),
10                CAST(N'2016-02-10' AS DATE),
11                'Y'),
12               (2,
13                6,
14                CAST(N'2016-01-10' AS DATE),
15                CAST(N'2016-02-10' AS DATE),
16                'Y')
```

	total	given	rest
Before	51	3	48
After	51	5	46

Let's delete the information about the subscription with identifier 42 (the book on this subscription has already been returned):

MS SQL Solution 3.1.2.a (checking the reaction to the deletion of a subscription with a returned book)

```
1    DELETE FROM [subscriptions]
2    WHERE  [sb_id] = 42
```

	total	given	rest
Before	51	5	46
After	51	5	46

Let's delete the information about the subscription with identifier 62 (the book on this subscription has not been returned yet):

MS SQL Solution 3.1.2.a (checking the reaction to the deletion of a subscription with an unreturned book)

```
1    DELETE FROM [subscriptions]
2    WHERE  [sb_id] = 62
```

	total	given	rest
Before	51	5	46
After	51	4	47

Finally, let's delete all books (which will also cascade delete all subscriptions):

MS SQL Solution 3.1.2.a (checking the reaction to the deletion of all books)

```
1    DELETE FROM [books]
```

	total	given	rest
Before	51	4	47
After	0	0	0

So, all data modification operations in **books** and **subscriptions** tables cause corresponding changes in the aggregating **books_statistics** table, which in MS SQL Server acts as a caching view.

Let's move on to the Oracle solution.

Oracle is the only DBMS in which this problem is fully solved using materialized views. If our materialized view were "simpler" (not using aggregating functions and relying on data containing primary keys), we could implement the most elegant option (**REFRESH FAST ON COMMIT**), which tells the DBMS to optimally update data in the materialized view every time another transaction affecting the tables from which data are collected is completed

But due to the peculiarities of the query used in the view, this option is not available to us, so we will use **REFRESH FORCE START WITH (SYSDATE) NEXT (SYSDATE + 1/1440)** instead, which means that we will force the view to update data once every minute.

Oracle Solution 3.1.2.a

```
1    -- Deleting the old version of the materialized view
2    -- (handy for development and debugging):
3    DROP MATERIALIZED VIEW "books_statistics";
4
5    -- Creating a materialized view:
6    CREATE MATERIALIZED VIEW "books_statistics"
7    BUILD IMMEDIATE
8    REFRESH FORCE
9    START WITH (SYSDATE) NEXT (SYSDATE + 1/1440)
10   AS
11      SELECT "total",
12             "given",
13             "total" - "given" AS "rest"
14      FROM   (SELECT SUM("b_quantity") AS "total"
15              FROM   "books")
16             JOIN (SELECT COUNT("sb_book") AS "given"
17                   FROM   "subscriptions"
18                   WHERE  "sb_is_active" = 'Y')
19               ON 1 = 1
```

The `BUILD IMMEDIATE` clause in line 7 instructs the DBMS to initialize the materialized view with data immediately.

The expressions in lines 8-9 describe the way of updating the data in the materialized view (`REFRESH FORCE` — the DBMS itself chooses the best available way, either a quick update or a full update) and the frequency of this operation (`START WITH (SYSDATE) NEXT (SYSDATE + 1/1440)`, i.e., start immediately and repeat every 1/1440th of a day, i.e., every minute).

The query in lines 11-19 should logically be identical to the queries we used in MySQL and MS SQL Server to initialize data in the aggregating table. But Oracle does not allow materialized views to use queries with subqueries in the `FROM` section, yet it does allow combining data from two subqueries.

So, we formed two subqueries (that calculate the total number of books in lines 14-15 and that calculate the number of books issued to readers in lines 16-18), and then combined their results (each subquery returns just one number) using the guaranteed condition `1 = 1`.

The result of the SQL code in lines 14-15 is as follows:

total	given
33	5

It only remains to select the values of the `"total"` and `"given"` fields from it and use them to calculate the value of the `"rest"` field, which happens in lines 11-13.

Again (as with MySQL and MS SQL Server) let's check how what we have created works. We will modify the data in the `books` and `subscriptions` tables and select data from the `books_statistics` materialized view. Note that after each query a `COMMIT` transaction confirmation is explicitly performed to prevent Oracle from waiting for this event and not updating the materialized view.

Let's add two books with the number of copies of 5 and 10:

Oracle | Solution 3.1.2.a (checking the reaction to adding books)

```
1    INSERT ALL
2      INTO "books" ("b_name",
3                    "b_quantity",
4                    "b_year")
5            VALUES (N'New book 1',
6                    5,
7                    2001)
8      INTO "books" ("b_name",
9                    "b_quantity",
10                   "b_year")
11           VALUES (N'New book 2',
12                   10,
13                   2002)
14   SELECT 1 FROM "DUAL";
15   COMMIT; -- And wait one minute.
```

	total	given	rest
Before	33	5	28
After	48	5	43

Let's increase by five the number of copies of the book, which is now registered in the library of 10 copies (we have one such book):

Oracle Solution 3.1.2.a (checking the reaction to changes in the book quantity)

```
1    UPDATE "books"
2    SET    "b_quantity" = "b_quantity" + 5
3    WHERE  "b_quantity" = 10;
4    COMMIT; -- And wait one minute.
```

	total	given	rest
Before	48	5	43
After	53	5	48

Let's delete the book, both copies of which are now in the hands of readers (book with identifier 1).

Oracle Solution 3.1.2.a (checking the reaction to book deletion)

```
1    DELETE FROM "books"
2    WHERE  "b_id" = 1;
3    COMMIT; -- And wait one minute.
```

	total	given	rest
Before	53	5	48
After	51	3	48

Let's mark that on subscription with the identifier 3 the book was returned:

Oracle Solution 3.1.2.a (checking the reaction to a book return)

```
1    UPDATE "subscriptions"
2    SET    "sb_is_active" = 'N'
3    WHERE  "sb_id" = 3;
4    COMMIT; -- And wait one minute.
```

	total	given	rest
Before	51	3	48
After	51	2	49

Let's cancel this operation (mark the book as unreturned again):

Oracle Solution 3.1.2.a (checking the reaction to cancellation of the book return)

```
1    UPDATE "subscriptions"
2    SET    "sb_is_active" = 'Y'
3    WHERE  "sb_id" = 3;
4    COMMIT; -- And wait one minute.
```

	total	given	rest
Before	51	2	49
After	51	3	48

Let's add to the database the information that the reader with identifier 2 took books with identifiers 5 and 6 from the library:

Oracle Solution 3.1.2.a (checking the reaction to the delivery of books)

```
INSERT ALL
 INTO "subscriptions" ("sb_subscriber",
                       "sb_book",
                       "sb_start",
                       "sb_finish",
                       "sb_is_active")
              VALUES (2,
                      5,
                      TO_DATE('2016-01-10', 'YYYY-MM-DD'),
                      TO_DATE('2016-02-10', 'YYYY-MM-DD'),
                      'Y')
 INTO "subscriptions" ("sb_subscriber",
                       "sb_book",
                       "sb_start",
                       "sb_finish",
                       "sb_is_active")
              VALUES (2,
                      6,
                      TO_DATE('2016-01-10', 'YYYY-MM-DD'),
                      TO_DATE('2016-02-10', 'YYYY-MM-DD'),
                      'Y')
SELECT 1 FROM "DUAL";
COMMIT; -- And wait one minute.
```

	total	given	rest
Before	51	3	48
After	51	5	46

Let's delete the information about the subscription with identifier 42 (the book on this subscription has already been returned):

Oracle Solution 3.1.2.a (checking the reaction to the deletion of a subscription with a returned book)

```
DELETE FROM "subscriptions"
WHERE "sb_id" = 42;
COMMIT; -- And wait one minute.
```

	total	given	rest
Before	51	5	46
After	51	5	46

Let's delete the information about the subscription with identifier 62 (the book on this subscription has not been returned yet):

Oracle Solution 3.1.2.a (checking the reaction to the deletion of a subscription with an unreturned book)

```
DELETE FROM "subscriptions"
WHERE "sb_id" = 62;
COMMIT; -- And wait one minute.
```

	total	given	rest
Before	51	5	46
After	51	4	47

Finally, let's delete all books (which will also cascade delete all subscriptions):

Oracle — Solution 3.1.2.a (checking the reaction to the deletion of all books)

```
1    DELETE FROM "books";
2    COMMIT; -- And wait one minute.
```

	total	given	rest
Before	51	4	47
After	0	0	0

So, all data modification operations in the **books** and **subscriptions** tables cause corresponding changes in the **books_statistics** materialized view.

Solution 3.1.2.b{229}.

If you missed the solution{230} of problem 3.1.2.a{229}, it is highly recommended that you review it before continuing reading, since many of the non-obvious points you will encounter in this solution have been covered previously.

Compared to the previous problem{229} everything will be much easier here, since the query that forms the required data set does not contain expressions that fall under the restrictions of indexed views in MS SQL Server and materialized views in Oracle.

The only problem will be with MySQL because it has no such views as a concept, and we have to create a caching table and triggers again.

Let's create a caching table (it is a caching table, not an aggregating table, as in 3.1.2.a{229}, because here we do not aggregate anything, but only save the final result). The code of its creation can be almost completely taken from the code of **subscriptions** table creation, and for **sb_subscriber** and **sb_book** fields we can take their definitions from **subscribers** and **books** tables respectively.

MySQL — Solution 3.1.2.b (creating a caching table)

```
1    CREATE TABLE `subscriptions_ready`
2    (
3            `sb_id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
4            `sb_subscriber` VARCHAR(150) NOT NULL,
5            `sb_book` VARCHAR(150) NOT NULL,
6            `sb_start` DATE NOT NULL,
7            `sb_finish` DATE NOT NULL,
8            `sb_is_active` ENUM ('Y', 'N') NOT NULL,
9           CONSTRAINT `PK_subscriptions` PRIMARY KEY (`sb_id`)
10   )
```

Let's initialize the data in the created table:

MySQL Solution 3.1.2.b (cleaning the table and data initialization)

```
-- Cleaning the table:
TRUNCATE TABLE `subscriptions_ready`;

-- Data initialization:
INSERT INTO `subscriptions_ready`
            (`sb_id`,
             `sb_subscriber`,
             `sb_book`,
             `sb_start`,
             `sb_finish`,
             `sb_is_active`)
SELECT `sb_id`,
       `s_name` AS `sb_subscriber`,
       `b_name` AS `sb_book`,
       `sb_start`,
       `sb_finish`,
       `sb_is_active`
FROM   `books`
       JOIN `subscriptions`
         ON `b_id` = `sb_book`
       JOIN `subscribers`
         ON `sb_subscriber` = `s_id`;
```

Let's create triggers that modify the data in the caching table. The data source is the **books**, **subscribers**, and **subscriptions** tables, so we have to create triggers for all three tables.

If you are using MySQL versions older than 5.7.2 (such versions do not allow creating multiple triggers of the same type on the same table), you will have to remove triggers created in task 3.2.1.a{229} from **books** and **subscriptions** tables before running the following code. As of version 5.7.2, this restriction no longer exists.

The registration of new books and readers in the library has no effect on the contents of the **subscriptions** table, so there is no need to create **INSERT** triggers on the **books** and **subscribers** tables, the **DELETE** and **UPDATE** triggers are enough.

Note a few important points in the code below:

- inside the trigger in MySQL we cannot explicitly or implicitly commit a transaction, so we cannot use the **TRUNCATE** statement to clean up the table (it is "non-transactional" and therefore leads to an implicit confirmation of the transaction);
- we do not store book identifiers in our caching table, so we cannot find and delete individual records (we cannot search and delete by book title either, as several different books may have the same title), so we have to clear the whole table every time and refill it with data again;
- in problem 3.2.1.a{229} we used **BEFORE** triggers, though we could have used **AFTER** triggers too, it did not matter there, but here we must use only **AFTER** triggers, because otherwise MySQL behavior is different than expected due to transaction logic, and the information in the caching table may not get updated.

Except for the nuances just discussed, the code for the body of both triggers is perfectly trivial and represents two queries: to delete all data from the table and to fill the table with data based on the resulting selection. Both of these queries can be executed completely separately from the triggers as independent SQL constructs.

The trigger that reacts to an update of book information checks (line 45) if the book name has changed. If not, there is no need to update the cached data.

MySQL — Solution 3.1.2.b (triggers for the `books` table)

```
1    -- Deleting old versions of triggers
2    -- (handy during development and debugging):
3    DROP TRIGGER `upd_sbs_rdy_on_books_del`;
4    DROP TRIGGER `upd_sbs_rdy_on_books_upd`;
5
6    -- Switching the query completion delimiter,
7    -- because now the query will create a trigger,
8    -- inside which there are their own, classic queries:
9    DELIMITER $$
10
11   -- Creating a trigger that reacts to the deletion of books:
12   CREATE TRIGGER `upd_sbs_rdy_on_books_del`
13   AFTER DELETE
14   ON `books`
15    FOR EACH ROW
16     BEGIN
17      DELETE FROM `subscriptions_ready`;
18      INSERT INTO `subscriptions_ready`
19                  (`sb_id`,
20                   `sb_subscriber`,
21                   `sb_book`,
22                   `sb_start`,
23                   `sb_finish`,
24                   `sb_is_active`)
25           SELECT `sb_id`,
26                  `s_name`,
27                  `b_name`,
28                  `sb_start`,
29                  `sb_finish`,
30                  `sb_is_active`
31           FROM   `books`
32                  JOIN `subscriptions`
33                    ON `b_id` = `sb_book`
34                  JOIN `subscribers`
35                    ON `sb_subscriber` = `s_id`;
36     END;
37   $$
```

MySQL Solution 3.1.2.b (triggers for the `books` table) (continued)

```
38  -- Creating a trigger reacting to
39  -- change in book data:
40  CREATE TRIGGER `upd_sbs_rdy_on_books_upd`
41  AFTER UPDATE
42  ON `books`
43   FOR EACH ROW
44    BEGIN
45     IF (OLD.`b_name` != NEW.`b_name`)
46      THEN
47       DELETE FROM `subscriptions_ready`;
48       INSERT INTO `subscriptions_ready`
49                  (`sb_id`,
50                   `sb_subscriber`,
51                   `sb_book`,
52                   `sb_start`,
53                   `sb_finish`,
54                   `sb_is_active`)
55            SELECT `sb_id`,
56                   `s_name`,
57                   `b_name`,
58                   `sb_start`,
59                   `sb_finish`,
60                   `sb_is_active`
61            FROM   `books`
62                   JOIN `subscriptions`
63                     ON `b_id` = `sb_book`
64                   JOIN `subscribers`
65                     ON `sb_subscriber` = `s_id`;
66     END IF;
67    END;
68  $$
69
70  -- Restoring the query completion delimiter:
71  DELIMITER ;
```

The entire logic of testing such triggers is shown in detail in the solution {230} of problem 3.1.2.a{229}. You can perform an experiment yourself, changing the data in the **`books`** table randomly and tracking the corresponding changes in the **`subscriptions_ready`** table.

Triggers on the **`subscribers`** table differ from triggers on the **`books`** table only by their names, the names of their tables and the name of the field whose value changes are checked to determine whether the cached data should be updated (in the code above the **`b_name`** field is checked, in the code below the **`s_name`** field is checked).

MySQL Solution 3.1.2.b (triggers for the `subscribers` table)

```
1  -- Deleting old versions of triggers
2  -- (handy during development and debugging):
3  DROP TRIGGER `upd_sbs_rdy_on_subscribers_del`;
4  DROP TRIGGER `upd_sbs_rdy_on_subscribers_upd`;
5
6  -- Switching the query completion delimiter,
7  -- because now the query will create a trigger,
8  -- inside which there are their own, classic queries:
9  DELIMITER $$
```

MySQL — Solution 3.1.2.b (triggers for the `subscribers` table) (continued)

```
-- Creating a trigger reacting to the deletion of readers:
CREATE TRIGGER `upd_sbs_rdy_on_subscribers_del`
AFTER DELETE
ON `subscribers`
 FOR EACH ROW
  BEGIN
   DELETE FROM `subscriptions_ready`;
   INSERT INTO `subscriptions_ready`
               (`sb_id`,
                `sb_subscriber`,
                `sb_book`,
                `sb_start`,
                `sb_finish`,
                `sb_is_active`)
          SELECT `sb_id`,
                 `s_name`,
                 `b_name`,
                 `sb_start`,
                 `sb_finish`,
                 `sb_is_active`
          FROM   `books`
                 JOIN `subscriptions`
                   ON `b_id` = `sb_book`
                 JOIN `subscribers`
                   ON `sb_subscriber` = `s_id`;
  END;
$$

-- Creating a trigger reacting to the
-- changing the data on readers:
CREATE TRIGGER `upd_sbs_rdy_on_subscribers_upd`
AFTER UPDATE
ON `subscribers`
 FOR EACH ROW
  BEGIN
   IF (OLD.`s_name` != NEW.`s_name`)
    THEN
     DELETE FROM `subscriptions_ready`;
     INSERT INTO `subscriptions_ready`
                 (`sb_id`,
                  `sb_subscriber`,
                  `sb_book`,
                  `sb_start`,
                  `sb_finish`,
                  `sb_is_active`)
            SELECT `sb_id`,
                   `s_name`,
                   `b_name`,
                   `sb_start`,
                   `sb_finish`,
                   `sb_is_active`
            FROM   `books`
                   JOIN `subscriptions`
                     ON `b_id` = `sb_book`
                   JOIN `subscribers`
                     ON `sb_subscriber` = `s_id`;
   END IF;
  END;
$$

-- Restoring the query completion delimiter:
DELIMITER ;
```

All three triggers (**`INSERT`**, **`UPDATE`** and **`DELETE`**) have to be created on the **`subscriptions`** table, because each of these operations can affect the contents of the caching **`subscriptions_ready`** table. And the code of all three triggers will be very different.

MySQL Solution 3.1.2.b (triggers for the **`subscriptions`** table)

```
1    -- Deleting old versions of triggers
2    -- (handy during development and debugging):
3    DROP TRIGGER `upd_sbs_rdy_on_subscriptions_ins`;
4    DROP TRIGGER `upd_sbs_rdy_on_subscriptions_del`;
5    DROP TRIGGER `upd_sbs_rdy_on_subscriptions_upd`;
6
7    -- Switching the query completion delimiter,
8    -- because now the query will create a trigger,
9    -- inside which there are their own, classic queries:
10   DELIMITER $$
11
12   -- Creating a trigger reacting to the addition of a subscription:
13   CREATE TRIGGER `upd_sbs_rdy_on_subscriptions_ins`
14   AFTER INSERT
15   ON `subscriptions`
16    FOR EACH ROW
17     BEGIN
18      INSERT INTO `subscriptions_ready`
19                  (`sb_id`,
20                   `sb_subscriber`,
21                   `sb_book`,
22                   `sb_start`,
23                   `sb_finish`,
24                   `sb_is_active`)
25           SELECT `sb_id`,
26                  `s_name`,
27                  `b_name`,
28                  `sb_start`,
29                  `sb_finish`,
30                  `sb_is_active`
31           FROM   `books`
32                  JOIN `subscriptions`
33                    ON `b_id` = `sb_book`
34                  JOIN `subscribers`
35                    ON `sb_subscriber` = `s_id`
36           WHERE `s_id` = NEW.`sb_subscriber`
37             AND `b_id` = NEW.`sb_book`;
38     END;
39   $$
40
41   -- Creating a trigger reacting to the deletion of a subscription:
42   CREATE TRIGGER `upd_sbs_rdy_on_subscriptions_del`
43   AFTER DELETE
44   ON `subscriptions`
45    FOR EACH ROW
46     BEGIN
47      DELETE FROM `subscriptions_ready`
48      WHERE `subscriptions_ready`.`sb_id` = OLD.`sb_id`;
49     END;
50   $$
```

MySQL Solution 3.1.2.b (triggers for the `subscriptions` table) (continued)

```
51  -- Creating a trigger reacting to an update of a subscription:
52  CREATE TRIGGER `upd_sbs_rdy_on_subscriptions_upd`
53  AFTER UPDATE
54  ON `subscriptions`
55   FOR EACH ROW
56    BEGIN
57     UPDATE `subscriptions_ready`
58            JOIN (SELECT `sb_id`,
59                         `s_name`,
60                         `b_name`
61                  FROM   `books`
62                         JOIN `subscriptions`
63                           ON `b_id` = `sb_book`
64                         JOIN `subscribers`
65                           ON `sb_subscriber` = `s_id`
66                  WHERE  `s_id` = NEW.`sb_subscriber`
67                     AND `b_id` = NEW.`sb_book`
68                     AND `sb_id` = NEW.`sb_id`) AS `new_data`
69     SET    `subscriptions_ready`.`sb_id` = NEW.`sb_id`,
70            `subscriptions_ready`.`sb_subscriber` = `new_data`.`s_name`,
71            `subscriptions_ready`.`sb_book` = `new_data`.`b_name`,
72            `subscriptions_ready`.`sb_start` = NEW.`sb_start`,
73            `subscriptions_ready`.`sb_finish` = NEW.`sb_finish`,
74            `subscriptions_ready`.`sb_is_active` = NEW.`sb_is_active`
75     WHERE  `subscriptions_ready`.`sb_id` = OLD.`sb_id`;
76    END;
77  $$
78
79  -- Restoring the query completion delimiter:
80  DELIMITER ;
```

The code for the **`INSERT`** trigger (lines 12-39) is very similar to the same trigger on the **`books`** table. The difference is that in this case we know the reader's and book's identifiers, which saves us from having to generate a full data set.

The code of the **`DELETE`** trigger (lines 41-50) is the most compact: we just need to remove from the **`subscriptions_ready`** table the records whose identifiers coincide with the identifiers of the records to be removed from the **`subscriptions`** table.

The code for the **`UPDATE`** trigger (lines 51-77) is the most nontrivial. The **`SELECT`** based **`UPDATE`** syntax itself looks unusual (we have to join results from the table being updated and the source selection in lines 57-68).

The conditions in lines 66-67 reduce the number of rows to be selected, and the condition in line 68 ensures that we get data about the new record of the **`subscriptions`** table, even if its primary key has changed. Following the same logic, we do not specify the condition of **`` `subscriptions_ready` JOIN ... `new_data` ``**, because the only sensible join condition here might be a match of **`sb_id`** values, but if the primary key of a record in the **`subscriptions`** table has changed, there will be no such match, because the **`subscriptions_ready`** table still stores the old primary key of the record being updated.

In lines 69-74 we take the new data for updating the fields from two sources: from explicitly passed data via the **`NEW`** keyword (all values that we can get directly) and from the results of selection **`new_data`** (reader's name and book's title, since they are not and cannot be in the explicitly passed data available via the **`NEW`** keyword).

The condition in line 75 ensures that we update the right record: here we need to compare the identifier of the record to be updated exactly with **`OLD.`sb_id``** and not with **`NEW.`sb_id``**, because the primary key in the **`subscriptions`** table may have changed, and we need to find its old value in the **`subscriptions_ready`** table (the expression in line 69 will also update this value, if it has changed).

The entire logic of testing such triggers is shown in detail in the solution{230} of problem 3.1.2.a{229}. You can experiment by changing the data in the **`books`**, **`subscribers`**, and **`subscriptions`** table randomly and monitoring the corresponding changes in the **`subscriptions_ready`** table.

Compared to the MySQL solution, the MS SQL Server and Oracle solutions are extremely simple. They are based on a simple **`SELECT`** query, which can be executed on its own.

MS SQL Solution 3.1.2.b

```
1    -- Deleting the old version of the indexed view
2    -- (handy for developing and debugging):
3    DROP VIEW [subscriptions_ready];
4
5    -- Creating a view:
6    CREATE OR ALTER VIEW [subscriptions_ready]
7    WITH SCHEMABINDING
8    AS
9    SELECT [sb_id],
10          [s_name] AS [sb_subscriber],
11          [b_name] AS [sb_book],
12          [sb_start],
13          [sb_finish],
14          [sb_is_active]
15   FROM   [dbo].[books]
16          JOIN [dbo].[subscriptions]
17            ON [b_id] = [sb_book]
18          JOIN [dbo].[subscribers]
19            ON [sb_subscriber] = [s_id];
20
21   -- Creating a unique clustered index on the view.
22   -- This is the operation that "enables" the automatic update
23   -- of the view when the data in the tables on which it is built
24   -- are changed:
25   CREATE UNIQUE CLUSTERED INDEX [idx_subscriptions_ready]
26     ON [subscriptions_ready] ([sb_id]);
```

For the data in **`subscriptions_ready`** view to be updated automatically, it must be indexed, i.e., a unique clustered index must be created on it (lines 25-26).

A prerequisite for creating such an index on a view is linking the view to the database schema (line 7), which requires the DBMS to establish and monitor the correspondence between the use of database objects in the view code and the actual state of such objects (their existence, accessibility, etc.) not only when the view is created, but also at the moment of any modification to the database objects to which the view is referring.

The solution of this problem for Oracle is even simpler: it takes a **`SELECT`** query that retrieves the desired data and uses it as the body of a materialized view. The DBMS will automatically update the data in this view once a minute (the logic of this behavior and other features of creating materialized views in Oracle were discussed in solution{230} of problem 3.1.2.a{229}.).

Oracle | Solution 3.1.2.b

```
1    -- Deleting the old version of the materialized view
2    -- (handy for development and debugging):
3    DROP MATERIALIZED VIEW "subscriptions_ready";
4
5    -- Creating a materialized view:
6    CREATE MATERIALIZED VIEW "subscriptions_ready"
7    BUILD IMMEDIATE
8    REFRESH FORCE
9    START WITH (SYSDATE) NEXT (SYSDATE + 1/1440)
10   AS
11     SELECT "sb_id",
12            "s_name" AS "sb_subscriber",
13            "b_name" AS "sb_book",
14            "sb_start",
15            "sb_finish",
16            "sb_is_active"
17     FROM   "books"
18            JOIN "subscriptions"
19              ON "b_id" = "sb_book"
20            JOIN "subscribers"
21              ON "sb_subscriber" = "s_id";
```

Task 3.1.2.TSK.A: check if the caching table and views from the solution of problem 3.1.2.b{229} are updated correctly.

Task 3.1.2.TSK.B: create a caching view that allows us to get a list of all books and their genres (two columns: the first with book title, the second with book genres, listed separated by commas).

Task 3.1.2.TSK.C: create a caching view that allows us to get a list of all authors and their books (two columns: the first with the author's name, the second with the books written by the author, listed separated by commas).

Task 3.1.2.TSK.D: modify the solution{260} of problem 3.1.2.a{229} for MySQL so that it takes into account the changes in the **`subscriptions`** table caused by the cascade deletion operation (when deleting subscribers). Make sure that solutions for MS SQL Server and Oracle do not require such modification.

3.1.3. Example 28: Using Views to Obscure Database Structures and Data Values

Problem 3.1.3.a[{256}]: create a view through which it is impossible to get information about which particular reader borrowed this or that book.

Problem 3.1.3.b[{257}]: create a view that returns all the information from the `subscriptions` table by converting the dates from the `sb_start` and `sb_finish` fields into the UNIXTIME format.

Expected result 3.1.3.a.

Executing a `SELECT * FROM {view}` query gets the following result:

sb_id	sb_book	sb_start	sb_finish	sb_is_active
2	1	2011-01-12	2011-02-12	N
3	3	2012-05-17	2012-07-17	Y
42	2	2012-06-11	2012-08-11	N
57	5	2012-06-11	2012-08-11	N
61	7	2014-08-03	2014-10-03	N
62	5	2014-08-03	2014-10-03	Y
86	1	2014-08-03	2014-09-03	Y
91	1	2015-10-07	2015-03-07	Y
95	4	2015-10-07	2015-11-07	N
99	4	2015-10-08	2025-11-08	Y
100	3	2011-01-12	2011-02-12	N

Expected result 3.1.3.b.

Executing a `SELECT * FROM {view}` query gets the following result:

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
2	1	1	1294779600	1297458000	N
3	3	3	1337202000	1342472400	Y
42	1	2	1339362000	1344632400	N
57	4	5	1339362000	1344632400	N
61	1	7	1407013200	1412283600	N
62	3	5	1407013200	1412283600	Y
86	3	1	1407013200	1409691600	Y
91	4	1	1444165200	1425675600	Y
95	1	4	1444165200	1446843600	N
99	4	4	1444251600	1762549200	Y
100	1	3	1294779600	1297458000	N

Solution 3.1.3.a[{255}].

The only thing we need to do to solve this problem is to build a view that selects all the fields of the **subscriptions** table except for the **sb_subscriber** field. The internal logic of the views will be absolutely identical for all the three DBMS, and the solutions will differ only in the syntax peculiarities of creating the views.

MySQL Solution 3.1.3.a

```
1    CREATE OR REPLACE VIEW `subscriptions_anonymous`
2    AS
3      SELECT `sb_id`,
4             `sb_book`,
5             `sb_start`,
6             `sb_finish`,
7             `sb_is_active`
8      FROM   `subscriptions`
```

MS SQL Solution 3.1.3.a

```
1    CREATE OR ALTER VIEW [subscriptions_anonymous]
2    WITH SCHEMABINDING
3    AS
4      SELECT [sb_id],
5             [sb_book],
6             [sb_start],
7             [sb_finish],
8             [sb_is_active]
9      FROM   [dbo].[subscriptions]
```

In MS SQL Server, we can afford to increase the reliability of a view by specifying (query line 2) the DBMS to establish and monitor the correspondence between the use of database objects in the view code and the actual state of such objects (their existence, accessibility, etc.) not only at the moment when the view is created, but also at the moment of any modification of the database objects to which the view refers. To enable this option, we must also specify the name of the table together with the schema name (**[dbo]**) to which it belongs (line 9 of the query).

Oracle Solution 3.1.3.a

```
1    CREATE OR REPLACE VIEW "subscriptions_anonymous"
2    AS
3      SELECT "sb_id",
4             "sb_book",
5             "sb_start",
6             "sb_finish",
7             "sb_is_active"
8      FROM   "subscriptions"
```

Oracle (like MySQL) does not support the **WITH SCHEMABINDING** option, so here we create a plain view using a trivial **SELECT** query.

Solution 3.1.3.b[255].

The solution of this problem is to build a view by selecting all fields from the **subscriptions** table, where the functions of converting **sb_start** and **sb_finish** fields from the internal DBMS date-time format into the UNIXTIME format are applied to them.

MySQL Solution 3.1.3.b

```
1    CREATE OR REPLACE VIEW `subscriptions_unixtime`
2    AS
3      SELECT `sb_id`,
4             `sb_subscriber`,
5             `sb_book`,
6             UNIX_TIMESTAMP(`sb_start`)  AS `sb_start`,
7             UNIX_TIMESTAMP(`sb_finish`) AS `sb_finish`,
8             `sb_is_active`
9      FROM   `subscriptions`
```

In MySQL there is a ready-made function to represent the date-time in the UNIXTIME format, that's what we used in lines 6-7.

MS SQL Solution 3.1.3.b

```
1    CREATE OR ALTER VIEW [subscriptions_unixtime]
2    WITH SCHEMABINDING
3    AS
4      SELECT [sb_id],
5             [sb_subscriber],
6             [sb_book],
7             DATEDIFF(SECOND, CAST(N'1970-01-01' AS DATE), [sb_start])
8             AS [sb_start],
9             DATEDIFF(SECOND, CAST(N'1970-01-01' AS DATE), [sb_finish])
10            AS [sb_finish],
11            [sb_is_active]
12     FROM   [dbo].[subscriptions]
```

In MS SQL Server there is no ready function to represent the date-time in the UNIXTIME format, so in lines 7-10 we calculate the UNIXTIME value by its definition: that is, we find the number of seconds elapsed from January 1, 1970 to the specified date.

For an explanation of the **WITH SCHEMABINDING** option, see the solution[256] of problem 3.1.3.a[255].

Oracle Solution 3.1.3.b

```
1    CREATE OR REPLACE VIEW "subscriptions_unixtime"
2    AS
3      SELECT "sb_id",
4             "sb_subscriber",
5             "sb_book",
6             (("sb_start" - TO_DATE('01-01-1970','DD-MM-YYYY')) * 86400)
7             AS "sb_start",
8             (("sb_finish" - TO_DATE('01-01-1970','DD-MM-YYYY')) * 86400)
9             AS "sb_finish",
10            "sb_is_active"
11     FROM   "subscriptions"
```

In Oracle (as well as in MS SQL Server) there is no ready-made function to represent the date-time in the UNIXTIME format, so in lines 6-9 we calculate the UNIXTIME-value by its definition: that is, we find the number of seconds elapsed from January 1, 1970 to the specified date.

Despite the seeming simplicity, there is a trap in the problem's condition, which is not so much in the area of writing SQL queries as in the area of working with different date-time concepts.

If we create the above views and retrieve data with them, we will see that they are slightly different in different DBMS. E.g., the date "January 12, 2011" is converted as follows:

MySQL	MS SQL Server	Oracle
1294779600	1294790400	1294790400

The results of MS SQL Server and Oracle coincide and differ from the result of MySQL by 10800 seconds (i.e., 180 minutes, i.e., three hours). And both results are correct. It's because the MySQL solution does not consider the time zone (in our case UTC+3), while the other two solutions take it into account.

To get the same result in MySQL as in MS SQL Server and Oracle, we can use the `CONVERT_TZ` function to convert the time zone:

MySQL — Solution 3.1.3.b (option with time zone conversion)

```
1   CREATE OR REPLACE VIEW `subscriptions_unixtime_tz`
2   AS
3     SELECT `sb_id`,
4            `sb_subscriber`,
5            `sb_book`,
6            UNIX_TIMESTAMP(CONVERT_TZ(`sb_start`, '+00:00', '+03:00'))
7            AS `sb_start`,
8            UNIX_TIMESTAMP(CONVERT_TZ(`sb_finish`, '+00:00', '+03:00'))
9            AS `sb_finish`,
10           `sb_is_active`
11    FROM   `subscriptions`
```

Task 3.1.3.TSK.A: create a view through which it is impossible to get information about which particular book was taken by a reader in any of the subscriptions.

Task 3.1.3.TSK.B: create a view that returns all the information from the `subscriptions` table by converting the dates from the `sb_start` and `sb_finish` fields into the "YYYYY-MM-DD WD" format, where "WD" is the day of the week as its full name (i.e., "Monday", "Tuesday", etc.)

3.2. Using Views to Modify Data

3.2.1. Example 29: Using Updatable Views to Modify Data

Problem 3.2.1.a[{260}]: create a view that retrieves reader information by converting all text to uppercase while allowing modification of the readers list.

Problem 3.2.1.b[{265}]: create a view that retrieves information about the dates of subscriptions start and finish as a single line, while allowing us to update the information in the `subscriptions` table.

Expected result 3.2.1.a.

Executing a `SELECT * FROM {view}` query gives the result shown below, and the `INSERT`, `UPDATE`, `DELETE` operations performed on the view modify the corresponding data in the original table.

s_id	s_name
1	IVANOV I.I.
2	PETROV P.P.
3	SIDOROV S.S.
4	SIDOROV S.S.

Expected result 3.2.1.b.

Executing a `SELECT * FROM {view}` query gives the result shown below, and the `INSERT`, `UPDATE`, `DELETE` operations performed on the view modify the corresponding data in the original table.

sb_id	sb_subscriber	sb_book	sb_dates	sb_is_active
2	1	1	2011-02-12 - 2011-02-12	N
3	3	3	2012-05-17 - 2012-07-17	Y
42	1	2	2012-06-11 - 2012-08-11	N
57	4	5	2012-06-11 - 2012-08-11	N
61	1	7	2014-08-03 - 2014-10-03	N
62	3	5	2014-08-03 - 2014-10-03	Y
86	3	1	2014-08-03 - 2014-09-03	Y
91	4	1	2015-10-07 - 2015-03-07	Y
95	1	4	2015-10-07 - 2015-11-07	N
99	4	4	2015-10-08 - 2025-11-08	Y
100	1	3	2011-01-12 - 2011-02-12	N

Solution 3.2.1.a[259].

Creating a view that allows us to retrieve information from the database in the form we want is easy. We'll just use the **UPPER** function to uppercase the reader's name (line 4 of the query).

MySQL — Solution 3.2.1.a (view that allows only data deletion)

```
1    CREATE OR REPLACE VIEW `subscribers_upper_case`
2    AS
3      SELECT `s_id`,
4             UPPER(`s_name`) AS `s_name`
5      FROM   `subscribers`
```

The problem is that MySQL imposes a wide range of restrictions[12] on updatable views, among which is the use of functions to convert field values. We use the **UPPER** function, which makes it impossible to perform **INSERT** and **UPDATE** operations using the resulting view (deletion will work).

To bypass the update restriction, we can do the following: in the view we must select both the "untouched" source field and its processed copy. This will partially violate the condition that the view must return only two fields with the same names as the original table fields, but it will give us the opportunity to update the data:

MySQL — Solution 3.2.1.a (view that allows data deletion and update)

```
1    CREATE OR REPLACE VIEW `subscribers_upper_case_trick`
2    AS
3      SELECT `s_id`,
4             `s_name`,
5             UPPER(`s_name`) AS `s_name_upper`
6      FROM   `subscribers`
```

Now we already have both deletion and update working. We can run the following queries to check this statement.

MySQL — Solution 3.2.1.a (checking the update and deletion)

```
1    UPDATE `subscribers_upper_case_trick`
2    SET    `s_name` = 'Sidorov A.A.'
3    WHERE  `s_id` = 4;
4
5    UPDATE `subscribers_upper_case_trick`
6    SET    `s_id` = 10
7    WHERE  `s_id` = 4;
8
9    DELETE FROM `subscribers_upper_case`
10   WHERE  `s_id` = 10;
```

Unfortunately, we will not be able to insert the data using such a view: for the insertion to work, the view must not reference the same field of the source table more than once.

Thus, for MySQL the problem is only partially solved.

MS SQL Server also has a series of restrictions[13], imposed on the views with which we plan to modify data. However (unlike MySQL) MS SQL Server allows us to create triggers on the views, with which we can solve the problem.

[12] https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html

[13] https://docs.microsoft.com/en-us/sql/t-sql/statements/create-view-transact-sql

MS SQL Solution 3.2.1.a (view creation)

```
1    CREATE OR ALTER VIEW [subscribers_upper_case]
2    WITH SCHEMABINDING
3    AS
4      SELECT [s_id],
5             UPPER([s_name]) AS [s_name]
6      FROM   [dbo].[subscribers]
```

Such a view already allows us to retrieve data in the format specified in the problem condition, as well as to delete data. In order to make it possible to insert and update data using this view, we need to create two triggers on it for insert and update operations.

MS SQL Solution 3.2.1.a (creating a trigger to implement an insertion operation)

```
1    CREATE TRIGGER [subscribers_upper_case_ins]
2    ON [subscribers_upper_case]
3    INSTEAD OF INSERT
4    AS
5        SET IDENTITY_INSERT [subscribers] ON;
6        INSERT INTO [subscribers]
7                    ([s_id],
8                     [s_name])
9        SELECT ( CASE
10                   WHEN [s_id] IS NULL
11                        OR [s_id] = 0 THEN IDENT_CURRENT('subscribers')
12                                           + IDENT_INCR('subscribers')
13                                           + ROW_NUMBER() OVER (ORDER BY
14                                                                (SELECT 1))
15                                           - 1
16                   ELSE [s_id]
17                 END ) AS [s_id],
18               [s_name]
19       FROM   [inserted];
20       SET IDENTITY_INSERT [subscribers] OFF;
21   GO
```

Such a trigger is executed **instead** of (`INSTEAD OF`) the operation of inserting data into the view and internally performs an insertion of data into the table on which the view is built.

Some complexity is caused by the fact that we want to allow insertion of only one field (reader's name) or both fields (reader's identifier and reader's name), and we do not know beforehand whether the value of `s_id` identifier will be passed during the insertion operation.

On lines 5 and 20, respectively, we allow and prohibit again the explicit insertion of data in the `s_id` field (which is the `IDENTITY` field for the `subscribers` table).

On lines 9-17 we check if we received the explicit `s_id` value. If no explicit value was passed, the `s_id` field of the `[inserted]` pseudotable can take the `NULL` value or `0`, in this case we calculate a new value of the `s_id` field for the `subscribers` table based on functions that return the current `IDENTITY` field value (`IDENT_CURRENT`) and its increment step (`IDENT_INCR`), as well as the row number from the `[inserted]` table.

In other words, the formula for calculating a new `IDENTITY` field value is: `current_value + increment_step + inserted_row_number - 1`.

Now each of the following data insertion queries will run correctly:

MS SQL Solution 3.2.1.a (check that the insert is working properly)

```
1    INSERT INTO [subscribers_upper_case]
2                ([s_name])
3    VALUES      (N'Orlov O.O.');
4
5    INSERT INTO [subscribers_upper_case]
6                ([s_name])
7    VALUES      (N'Sokolov S.S.'),
8                (N'Berkutov B.B.');
9
10   INSERT INTO [subscribers_upper_case]
11               ([s_id],
12                [s_name])
13   VALUES      (30,
14                N'Yastrebov Y.Y.');
15
16   INSERT INTO [subscribers_upper_case]
17               ([s_id],
18                [s_name])
19   VALUES      (31,
20                N'Sinitsyn S.S.'),
21               (32,
22                N'Voronov V.V.');
```

Moving on to implementing the data update.

MS SQL Solution 3.2.1.a (creating a trigger to implement the update operation)

```
1    CREATE TRIGGER [subscribers_upper_case_upd]
2    ON [subscribers_upper_case]
3    INSTEAD OF UPDATE
4    AS
5        IF UPDATE([s_id])
6          BEGIN
7              RAISERROR ('UPDATE of Primary Key through
8                          [subscribers_upper_case_upd]
9                          view is prohibited.', 16, 1);
10             ROLLBACK;
11         END
12         ELSE
13          UPDATE [subscribers]
14          SET    [subscribers].[s_name] = [inserted].[s_name]
15          FROM   [subscribers]
16            JOIN [inserted]
17              ON [subscribers].[s_id] = [inserted].[s_id];
18   GO
```

When updating data, the “old” rows are copied to the **`[deleted]`** pseudotable and the “new” rows are copied to the **`[inserted]`** pseudotable, but there is an insurmountable problem: there is no way to guarantee that the rows in these two pseudotables match, i.e., if a primary key value changes, we cannot determine its new value.

So, in lines 5-11 of the trigger code, we check if there was an attempt to update the value of the primary key with the **`UPDATE`** function (yes, this is not an **`UPDATE`** operator here, but a function). If it was attempted, we disallow the operation and rollback the transaction. If the primary key has not been affected by the update, we can easily define a new value for the reader’s name and use it to actually update the data in the **`subscribers`** table (lines 12-17).

Now the following queries run correctly (by the way, the deletion did not require any modifications, but it is still worth checking):

MS SQL Solution 3.2.1.a (check that updates and deletions are working)

```
1     UPDATE [subscribers_upper_case]
2     SET    [s_name] = N'New name'
3     WHERE  [s_id] = 30;
4
5     UPDATE [subscribers_upper_case]
6     SET    [s_name] = N'And one more name'
7     WHERE  [s_id] >= 31;
8
9     DELETE FROM [subscribers_upper_case]
10    WHERE  [s_id] = 30;
11
12    DELETE FROM [subscribers_upper_case]
13    WHERE  [s_id] >= 31;
```

An attempt to update the value of the primary key will naturally lead to the operation blocking:

MS SQL Solution 3.2.1.a (check that the primary key value cannot be updated)

```
1     UPDATE [subscribers_upper_case]
2     SET    [s_id] = 50
3     WHERE  [s_id] = 1;
```

This query will result in the following error message:

```
Msg 50000, Level 16, State 1, Procedure subscribers_upper_case_upd, Line 7
UPDATE of Primary Key through [subscribers_upper_case_upd] view is prohibited.
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

This completes the solution for MS SQL Server, and we move on to the solution for Oracle.

Let's create a view.

Oracle Solution 3.2.1.a (view creation)

```
1     CREATE OR REPLACE VIEW "subscribers_upper_case"
2     AS
3       SELECT "s_id",
4              UPPER("s_name") AS "s_name"
5       FROM   "subscribers"
```

Using this view, you can already retrieve data in the desired format and delete data.

Important: if we search for records for deletion by **`s_name`** field, we must pass the data we are looking for in uppercase. This peculiarity concerns only Oracle, because MySQL and MS SQL Server do not impose such a restriction.

In order to be able to insert and update data using this view, we need to create two triggers on it: on insert and update operations.

Since Oracle (unlike MS SQL Server) supports row-level triggers, their code is very simple and their implementation allows us to bypass the primary key update restriction of MS SQL Server.

Oracle Solution 3.2.1.a (creating a trigger to implement an insert operation)

```
CREATE OR REPLACE TRIGGER "subscribers_upper_case_ins"
INSTEAD OF INSERT ON "subscribers_upper_case"
FOR EACH ROW
 BEGIN
    INSERT INTO "subscribers"
                ("s_id",
                 "s_name")
    VALUES      (:new."s_id",
                 :new."s_name");
 END;
```

Oracle Solution 3.2.1.a (creating a trigger to implement an update operation)

```
CREATE OR REPLACE TRIGGER "subscribers_upper_case_upd"
INSTEAD OF UPDATE ON "subscribers_upper_case"
FOR EACH ROW
 BEGIN
    UPDATE "subscribers"
    SET    "s_id" = :new."s_id",
           "s_name" = :new."s_name"
    WHERE  "s_id" = :old."s_id";
 END;
```

The code for both triggers comes down to performing an insert or update queries to the table on which the view is built. As in MySQL, we can use the **`old`** and **`new`** keywords in Oracle to refer to “old” (to be deleted or updated) and new (to be added or updated) data.

Now the following queries run correctly (by the way, the deletion did not require any modifications, but it is still worth checking):

Oracle Solution 3.2.1.a (check the functionality of data modification)

```
INSERT ALL
 INTO "subscribers" ("s_id", "s_name") VALUES (1, N'Sokolov S.S.')
 INTO "subscribers" ("s_id", "s_name") VALUES (2, N'Berkutov B.B.')
 INTO "subscribers" ("s_id", "s_name") VALUES (3, N'Filinov F.F.')
SELECT 1 FROM "DUAL";

UPDATE "subscribers_upper_case"
SET    "s_name" = N'Sinitsyn N.N.'
WHERE  "s_id" = 6;

-- This query WILL NOT find anything, since the name must be in uppercase:
UPDATE "subscribers_upper_case"
SET    "s_name" = N'Sinitsyn A.A.'
WHERE  "s_name" = N'Sinitsyn N.N.';

-- And this query will find what we're looking for:
UPDATE "subscribers_upper_case"
SET    "s_name" = N'Sinitsyn A.A.'
WHERE  "s_name" = N'SINITSYN N.N.';

DELETE FROM "subscribers_upper_case"
WHERE  "s_id" = 6;

-- This query WILL NOT find anything, since the name must be in uppercase:
DELETE FROM "subscribers_upper_case"
WHERE  "s_name" = N'Filinov F.F.';

-- And this query will find what we're looking for:
DELETE FROM "subscribers_upper_case"
WHERE  "s_name" = N'FILINOV F.F.';

DELETE FROM "subscribers_upper_case"
WHERE  "s_id" > 4;
```

Solution 3.2.1.b[259].

The solution of this problem for MySQL falls under all the restrictions typical of the solution[260] of problem 3.2.1.a[259]: we can also create a view that either satisfies the selection format and allows only data deletion, or that does not satisfy the selection format (with additional fields) and allows data update and deletion. But data insertion won't work anyway.

MySQL Solution 3.2.1.b (view that only allows data deletion)

```
CREATE OR REPLACE VIEW `subscriptions_wcd`
AS
  SELECT `sb_id`,
         `sb_subscriber`,
         `sb_book`,
         CONCAT(`sb_start`, ' - ', `sb_finish`) AS `sb_dates`,
         `sb_is_active`
  FROM   `subscriptions`
```

MySQL Solution 3.2.1.b (view that allows data deletion and updating)

```
CREATE OR REPLACE VIEW `subscriptions_wcd_trick`
AS
  SELECT `sb_id`,
         `sb_subscriber`,
         `sb_book`,
         CONCAT(`sb_start`, ' - ', `sb_finish`) AS `sb_dates`,
         `sb_start`,
         `sb_finish`,
         `sb_is_active`
  FROM   `subscriptions`
```

Except for the already repeatedly mentioned problem with insertion, which does not allow to solve the problem in MySQL completely, the code of views is absolutely trivial and is built on elementary `SELECT` queries.

The limitations of MS SQL Server regarding updatable views mentioned in the solution[260] of problem 3.2.1.a[259] are also relevant in this problem: we will again have to create `INSTEAD OF` triggers to implement data update and insertion.

MS SQL Solution 3.2.1.b (view creation)

```
CREATE OR ALTER VIEW [subscriptions_wcd]
WITH SCHEMABINDING
AS
  SELECT [sb_id],
         [sb_subscriber],
         [sb_book],
         CONCAT([sb_start], ' - ', [sb_finish]) AS [sb_dates],
         [sb_is_active]
  FROM   [dbo].[subscriptions]
```

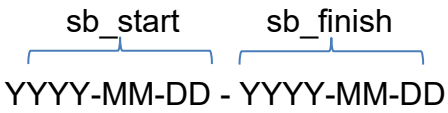
Such a view already allows us to retrieve data in the format specified in the problem condition, as well as to delete data. In order to make it possible to insert and update data using this view, we need to create two triggers on it: on insert and update operations.

MS SQL Solution 3.2.1.b (create a trigger to implement an insert operation)

```
1    CREATE TRIGGER [subscriptions_wcd_ins]
2    ON [subscriptions_wcd]
3    INSTEAD OF INSERT
4    AS
5        SET IDENTITY_INSERT [subscriptions] ON;
6        INSERT INTO [subscriptions]
7                    ([sb_id],
8                     [sb_subscriber],
9                     [sb_book],
10                    [sb_start],
11                    [sb_finish],
12                    [sb_is_active])
13       SELECT ( CASE
14                   WHEN [sb_id] IS NULL
15                        OR [sb_id] = 0 THEN IDENT_CURRENT('subscriptions')
16                                            + IDENT_INCR('subscriptions')
17                                            + ROW_NUMBER() OVER (ORDER BY
18                                                                 (SELECT 1))
19                                            - 1
20                   ELSE [sb_id]
21                 END ) AS [sb_id],
22                [sb_subscriber],
23                [sb_book],
24                SUBSTRING([sb_dates], 1, (CHARINDEX(' ', [sb_dates]) - 1))
25                AS [sb_start],
26                SUBSTRING([sb_dates], (CHARINDEX(' ', [sb_dates]) + 3),
27                          DATALENGTH([sb_dates]) -
28                          (CHARINDEX(' ', [sb_dates]) + 2))
29                AS [sb_finish],
30                [sb_is_active]
31       FROM     [inserted];
32       SET IDENTITY_INSERT [subscriptions] OFF;
33   GO
```

The non-trivial logic of getting the value of the primary key (lines 13-21) is explained in detail in the solution{260} of problem 3.2.1.a{259}.

As for getting the values of **`sb_start`** and **`sb_finish`** fields (lines 24-25 and 26-29), we can see the consequences of another MS SQL Server limitation: in the **`[inserted]`** pseudotable there are no fields that are not in the view the trigger is built on. So, the only way[14] to get their values is to extract them from **`sb_dates`** field. This is as follows (parts of string containing two dates are extracted using string functions):



This approach is slow and unreliable, but there are no alternatives. If we want to improve reliability of trigger's work, we can add additional check for correctness of "combined date" format in **`sb_dates`** field, but each such additional operation will negatively affect performance.

[14] https://technet.microsoft.com/en-us/library/ms190188%28v=sql.105%29.aspx

Let's check how the data insertion will work using the created trigger:

MS SQL Solution 3.2.1.b (check that the insert is working properly)

```
INSERT INTO [subscriptions_wcd]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_dates],
             [sb_is_active])
VALUES      (1000,
             1,
             3,
             '2017-01-12 - 2017-03-15',
             'N'),
            (2000,
             1,
             1,
             '2017-01-12 - 2017-03-15',
             'N');

INSERT INTO [subscriptions_wcd]
            ([sb_subscriber],
             [sb_book],
             [sb_dates],
             [sb_is_active])
VALUES      (1,
             3,
             '2019-01-12 - 2019-03-15',
             'N'),
            (1,
             1,
             '2019-01-12 - 2019-03-15',
             'N');

INSERT INTO [subscriptions_wcd]
            ([sb_subscriber],
             [sb_book],
             [sb_dates],
             [sb_is_active])
VALUES      (1,
             3,
             'Not dates, but something else.',
             'N');
```

The first two queries work correctly, but the third one expectedly leads to an erroneous situation:

```
Msg 241, Level 16, State 1, Procedure subscriptions_wcd_ins, Line 6
Conversion failed when converting date and/or time from character string.
```

Moving on to the implementation of the data update.

MS SQL Solution 3.2.1.b (create a trigger to implement the update operation)

```
1    CREATE TRIGGER [subscriptions_wcd_upd]
2    ON [subscriptions_wcd]
3    INSTEAD OF UPDATE
4    AS
5        IF UPDATE([sb_id])
6          BEGIN
7              RAISERROR ('UPDATE of Primary Key through
8                          [subscriptions_wcd_upd]
9                          view is prohibited.', 16, 1);
10             ROLLBACK;
11         END
12         ELSE
13          UPDATE [subscriptions]
14          SET    [subscriptions].[sb_subscriber] = [inserted].[sb_subscriber],
15                 [subscriptions].[sb_book] = [inserted].[sb_book],
16                 [subscriptions].[sb_start] =
17                                    SUBSTRING([sb_dates], 1,
18                                    (CHARINDEX(' ', [sb_dates]) - 1)),
19                 [subscriptions].[sb_finish] =
20                                    SUBSTRING([sb_dates],
21                                    (CHARINDEX(' ', [sb_dates]) + 3),
22                                    DATALENGTH([sb_dates]) -
23                                    (CHARINDEX(' ', [sb_dates]) + 2)),
24                 [subscriptions].[sb_is_active] = [inserted].[sb_is_active]
25          FROM   [subscriptions]
26            JOIN [inserted]
27              ON [subscriptions].[sb_id] = [inserted].[sb_id];
28   GO
```

The logic of prohibiting the update of the primary key (lines 5-11) is explained in detail in the solution of problem 3.2.1.a.

The need to get the values of **`sb_start`** and **`sb_finish`** fields (lines 16-23) has just been discussed in the implementation of the **`INSERT`** trigger.

For the rest, the query on lines 14-27 is a classic implementation of a **`SELECT`** based **`UPDATE`**.

It remains to make sure that the update and deletion works correctly:

MS SQL Solution 3.2.1.b (check that the update and deletion work properly)

```
1    UPDATE [subscriptions_wcd]
2    SET    [sb_dates] = '2021-01-12 - 2021-03-15'
3    WHERE  [sb_id] = 1000;
4
5    DELETE FROM [subscriptions_wcd]
6    WHERE  [sb_id] = 2000;
7
8    DELETE FROM [subscriptions_wcd]
9    WHERE  [sb_dates] = '2021-01-12 - 2021-03-15';
```

An attempt to update the value of the primary key will naturally lead to operation blocking:

MS SQL Solution 3.2.1.b (checking if the primary key value cannot be updated)

```
1    UPDATE [subscriptions_wcd]
2    SET    [sb_id] = 999
3    WHERE  [sb_id] = 1000;
```

This request will result in the following error message:

```
Msg 50000, Level 16, State 1, Procedure subscriptions_wcd_upd, Line 7
UPDATE of Primary Key through [subscriptions_wcd_upd] view is prohibited.
Msg 3609, Level 16, State 1, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

This completes the solution for MS SQL Server, and we move on to the solution for Oracle.

Oracle Solution 3.2.1.b (view creation)

```
1     CREATE OR REPLACE VIEW "subscriptions_wcd"
2     AS
3       SELECT "sb_id",
4              "sb_subscriber",
5              "sb_book",
6              TO_CHAR("sb_start", 'YYYY-MM-DD') || ' - ' ||
7              TO_CHAR("sb_finish", 'YYYY-MM-DD') AS "sb_dates",
8              "sb_is_active"
9       FROM   "subscriptions"
```

Using the `TO_CHAR` function in lines 6-7 allows us to get a string with dates in the format defined by the problem.

Oracle Solution 3.2.1.b (create a trigger to implement an insert operation)

```
1     CREATE OR REPLACE TRIGGER "subscriptions_wcd_ins"
2       INSTEAD OF INSERT ON "subscriptions_wcd"
3       FOR EACH ROW
4        BEGIN
5         INSERT INTO "subscriptions"
6                     ("sb_id",
7                      "sb_subscriber",
8                      "sb_book",
9                      "sb_start",
10                     "sb_finish",
11                     "sb_is_active")
12        VALUES       (:new."sb_id",
13                     :new."sb_subscriber",
14                     :new."sb_book",
15                     TO_DATE(SUBSTR(:new."sb_dates", 1,
16                            (INSTR(:new."sb_dates", ' ') - 1 )), 'YYYY-MM-DD'),
17                     TO_DATE(SUBSTR(:new."sb_dates",
18                            (INSTR(:new."sb_dates", ' ') + 3 )), 'YYYY-MM-DD'),
19                     :new."sb_is_active");
20       END;
```

Oracle Solution 3.2.1.b (create a trigger to implement an update operation)

```
1     CREATE OR REPLACE TRIGGER "subscriptions_wcd_ins"
2     INSTEAD OF UPDATE ON "subscriptions_wcd"
3     FOR EACH ROW
4       BEGIN
5         UPDATE "subscriptions"
6         SET    "sb_id" = :new."sb_id",
7                "sb_subscriber" = :new."sb_subscriber",
8                "sb_book" = :new."sb_book",
9                "sb_start" = TO_DATE(SUBSTR(:new."sb_dates", 1,
10                                   (INSTR(:new."sb_dates", ' ') - 1)),
11                                   'YYYY-MM-DD'),
12               "sb_finish" = TO_DATE(SUBSTR(:new."sb_dates",
13                                   (INSTR(:new."sb_dates", ' ') + 3 )),
14                                   'YYYY-MM-DD'),
15               "sb_is_active" = :new."sb_is_active"
16        WHERE  "sb_id" = :old."sb_id";
17      END;
```

Thanks to the support of row-level triggers, the solution of this problem in Oracle is quite simple: the code for both triggers comes down to executing an `INSERT` or `UPDATE` query on the table on which the view is built. As in MySQL, we can use the `old` and `new` keywords in Oracle to refer to “old” (deleted or updated) and new (added or updated) data.

The values of **sb_start** and **sb_finish** fields (same as in MS SQL Server solution) in both triggers have to be calculated using string functions, but in Oracle their syntax is a bit easier and the solution is shorter.

Now all of the following queries work correctly:

Oracle — Solution 3.2.1.b (check the functionality of data modification)

```
1   INSERT INTO "subscriptions_wcd"
2                ("sb_subscriber",
3                 "sb_book",
4                 "sb_dates",
5                 "sb_is_active")
6   VALUES       (1,
7                 3,
8                 '2019-01-12 - 2019-02-12',
9                 'N');
10
11  UPDATE "subscriptions_wcd"
12  SET    "sb_dates" = '2019-01-12 - 2019-02-12'
13  WHERE  "sb_id" = 100;
14
15  DELETE FROM "subscriptions_wcd"
16  WHERE  "sb_id" = 100;
17
18  DELETE FROM "subscriptions_wcd"
19  WHERE  "sb_dates" = '2012-05-17 - 2012-07-17';
```

Important! Oracle only allows data insertion using a view with syntax of the following kind

```
INSERT INTO ... (...) VALUES (...)
```

but not of this kind

```
INSERT ALL
 INTO ... (...) VALUES (…)
 INTO ... (...) VALUES (…)
SELECT 1 FROM "DUAL"
```

If you try to use the second option, you will get an error message “ORA-01702: a view is not appropriate here”.

Task 3.2.1.TSK.A: create a view that retrieves information about books, converting all text to uppercase, while allowing modification of the books list.

Task 3.2.1.TSK.B: create a view that retrieves information about the dates of subscriptions’ start and finish and the status of a book as a single line in the format “YYYY-MM-DD - YYYY-MM-DD - Returned” and at the same time allows us to update the information in the **subscriptions** table.

3.2.2. Example 30: Using Triggers on Views to Modify Data

Since MySQL does not allow us to create triggers on views, all problems in this example only have solutions for MS SQL Server and Oracle.

Problem 3.2.2.a{272}: create a view that extracts human-readable information from the **`subscriptions`** table (with readers' names and books' titles instead of identifiers), while allowing us to modify the data in the **`subscriptions`** table.

Problem 3.2.2.b{284}: create a view that shows a list of books with genres related to those books, while allowing us to add new genres.

Expected result 3.2.2.a.

Executing a **`SELECT * FROM {view}`** query gives the result shown below, and the **`INSERT`**, **`UPDATE`**, **`DELETE`** operations performed on the view modify the corresponding data in the original table.

sb_id	sb_subscriber	sb_book	sb_start	sb_finish	sb_is_active
2	Ivanov I.I.	Eugene Onegin	2011-01-12	2011-02-12	N
3	Sidorov S.S.	Foundation and Empire	2012-05-17	2012-07-17	Y
42	Ivanov I.I.	The Fisherman and the Golden Fish	2012-06-11	2012-08-11	N
57	Sidorov S.S.	The C++ Programming Language	2012-06-11	2012-08-11	N
61	Ivanov I.I.	The Art of Computer Programming	2014-08-03	2014-10-03	N
62	Sidorov S.S.	The C++ Programming Language	2014-08-03	2014-10-03	Y
86	Sidorov S.S.	Eugene Onegin	2014-08-03	2014-09-03	Y
91	Sidorov S.S.	Eugene Onegin	2015-10-07	2015-03-07	Y
95	Ivanov I.I.	Programming Psychology	2015-10-07	2015-11-07	N
99	Sidorov S.S.	Programming Psychology	2015-10-08	2025-11-08	Y
100	Ivanov I.I.	Foundation and Empire	2011-01-12	2011-02-12	N

Expected result 3.2.2.b.

Executing a **`SELECT * FROM {view}`** query gives the result shown below, and the **`INSERT`** operation on the view adds a new genre to the **`genres`** table.

b_id	b_name	genres
1	Eugene Onegin	Classic,Poetry
2	The Fisherman and the Golden Fish	Classic,Poetry
3	Foundation and Empire	Science Fiction
4	Programming Psychology	Programming,Psychology
5	The C++ Programming Language	Programming
6	Course of Theoretical Physics	Classic
7	The Art of Computer Programming	Programming,Classic

Solution 3.2.2.a{271}.

Unfortunately, for MySQL, this problem has no solution, because MySQL does not allow us to create triggers on views. The best we can do is to create the view itself, but we won't be able to modify data with it:

MySQL Solution 3.2.2.a (view creation)

```
1     CREATE OR REPLACE VIEW `subscriptions_with_text`
2     AS
3     SELECT `sb_id`,
4            `s_name` AS `sb_subscriber`,
5            `b_name` AS `sb_book`,
6            `sb_start`,
7            `sb_finish`,
8            `sb_is_active`
9     FROM   `subscriptions`
10           JOIN `subscribers` ON `sb_subscriber` = `s_id`
11           JOIN `books` ON `sb_book` = `b_id`
```

In MS SQL Server the code of the view itself is identical:

MS SQL Solution 3.2.2.a (view creation)

```
1     CREATE OR ALTER VIEW [subscriptions_with_text]
2     WITH SCHEMABINDING
3     AS
4     SELECT [sb_id],
5            [s_name] AS [sb_subscriber],
6            [b_name] AS [sb_book],
7            [sb_start],
8            [sb_finish],
9            [sb_is_active]
10    FROM   [dbo].[subscriptions]
11           JOIN [dbo].[subscribers] ON [sb_subscriber] = [s_id]
12           JOIN [dbo].[books] ON [sb_book] = [b_id]
```

Let's create the trigger to implement the insert operation.

Since the original **`subscriptions`** table contains digital readers' and books' identifiers in the **`sb_subscriber`** and **`sb_book`** fields, we must get the inserted values of these fields in digital form.

However, trying to get reader's or book's identifiers on the fly based on their name or title is not feasible because both readers' names and books' titles can overlap, and we cannot guarantee that we will get the correct value.

To minimize the probability of misuse of the resulting trigger, its lines 5-14 check whether non-digital values are passed into **`sb_subscriber`** or **`sb_book`** fields during insertion. If this situation occurs, an error message is displayed, and the transaction is cancelled.

Apart from that, the whole trigger code is very similar to the one described in the solution{260} of problem 3.2.1.a{259} (the logic of calculating a new primary key value if it is not explicitly passed in the data insertion request (lines 25-33 of the below trigger) is described in detail there).

MS SQL Solution 3.2.2.a (create a trigger to implement an insert operation)

```
1    CREATE TRIGGER [subscriptions_with_text_ins]
2    ON [subscriptions_with_text]
3    INSTEAD OF INSERT
4    AS
5       IF EXISTS(SELECT 1
6                 FROM   [inserted]
7                 WHERE  PATINDEX('%[^0-9]%', [sb_subscriber]) > 0
8                     OR PATINDEX('%[^0-9]%', [sb_book]) > 0)
9          BEGIN
10          RAISERROR ('Use digital identifiers for [sb_subscriber]
11                      and [sb_book]. Do not use subscribers'' names
12                      or books'' titles', 16, 1);
13          ROLLBACK;
14         END
15         ELSE
16          BEGIN
17         SET IDENTITY_INSERT [subscriptions] ON;
18         INSERT INTO [subscriptions]
19                     ([sb_id],
20                      [sb_subscriber],
21                      [sb_book],
22                      [sb_start],
23                      [sb_finish],
24                      [sb_is_active])
25         SELECT ( CASE
26                    WHEN [sb_id] IS NULL
27                          OR [sb_id] = 0 THEN IDENT_CURRENT('subscriptions')
28                                              + IDENT_INCR('subscriptions')
29                                              + ROW_NUMBER() OVER (ORDER BY
30                                                                   (SELECT 1))
31                                              - 1
32                    ELSE [sb_id]
33                  END ) AS [sb_id],
34                [sb_subscriber],
35                [sb_book],
36                [sb_start],
37                [sb_finish],
38                [sb_is_active]
39         FROM   [inserted];
40         SET IDENTITY_INSERT [subscriptions] OFF;
41         END
42   GO
```

Let's check how the following insertion queries are executed. Queries 1-2 are expectedly executed correctly, but queries 3-5 activate the code in lines 5-14 of the trigger and cancel the transaction, because passing non-digital values to the **`sb_subscriber`** and/or **`sb_book`** fields is forbidden.

MS SQL Solution 3.2.2.a (checking the functionality of the insert operation)

```
-- Query 1 (insert works here):
INSERT INTO [subscriptions_with_text]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (5000,
             1,
             3,
             '2015-01-12',
             '2015-02-12',
             'N'),
            (5005,
             1,
             1,
             '2015-01-12',
             '2015-02-12',
             'N');

-- Query 2 (insert works here):
INSERT INTO [subscriptions_with_text]
            ([sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (1,
             3,
             '2015-01-12',
             '2015-02-12',
             'N'),
            (1,
             1,
             '2015-01-12',
             '2015-02-12',
             'N');

-- Query 3 (insert does NOT work here):
INSERT INTO [subscriptions_with_text]
            ([sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (N'Ivanov I.I.',
             3,
             '2015-01-12',
             '2015-02-12',
             'N');

-- Query 4 (insert does NOT work here):
INSERT INTO [subscriptions_with_text]
            ([sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (1,
             N'Some book',
             '2015-01-12',
             '2015-02-12',
             'N');
```

MS SQL Solution 3.2.2.a (checking the functionality of the insert operation) (continued)

```
65   -- Query 5 (insert does NOT work here):
66   INSERT INTO [subscriptions_with_text]
67               ([sb_subscriber],
68                [sb_book],
69                [sb_start],
70                [sb_finish],
71                [sb_is_active])
72   VALUES      (N'Some subscriber',
73                N'Some book',
74                '2015-01-12',
75                '2015-02-12',
76                'N');
```

Let's create a trigger to implement the data update operation. Its logic will be a bit more complicated than in the just discussed **INSTEAD OF INSERT** trigger.

We should respond to non-digital values in the **sb_subscriber** and **sb_book** fields only if these values were explicitly passed in the query, this makes it necessary to create a more complex condition in lines 7-10.

If **sb_subscriber** and **sb_book** fields were not passed in the **UPDATE** query, they will naturally contain human-readable readers' names and books' titles (because the **[inserted]** and **[deleted]** pseudo-tables are filled with data from the view).

We consider this situation and fix it in lines 29-42: if there are non-digital data in fields (and trigger execution has reached this part), it means that these fields do not appear in **UPDATE** query and their values should be taken from the source table (**subscriptions**).

Finally, as it was stressed in the solution{260} of problem 3.2.1.a{259}, we cannot correctly determine the relationship between the entries in the **[inserted]** and **[deleted]** pseudotables if the primary key has been changed, so we forbid this operation (lines 19-25).

MS SQL Solution 3.2.2.a (create a trigger to implement the update operation)

```
1    CREATE TRIGGER [subscriptions_with_text_upd]
2    ON [subscriptions_with_text]
3    INSTEAD OF UPDATE
4    AS
5       IF EXISTS(SELECT 1
6                 FROM [inserted]
7                 WHERE (    UPDATE([sb_subscriber])
8                        AND PATINDEX('%[^0-9]%', [sb_subscriber]) > 0)
9                    OR (    UPDATE([sb_book])
10                       AND PATINDEX('%[^0-9]%', [sb_book]) > 0))
11          BEGIN
12           RAISERROR ('Use digital identifiers for [sb_subscriber]
13                       and [sb_book]. Do not use subscribers'' names
14                       or books'' titles', 16, 1);
15           ROLLBACK;
16          END
17          ELSE
18            BEGIN
19             IF UPDATE([sb_id])
20              BEGIN
21               RAISERROR ('UPDATE of Primary Key through
22                           [subscriptions_with_text]
23                          view is prohibited.', 16, 1);
24               ROLLBACK;
25              END
26              ELSE
27               BEGIN
28                UPDATE [subscriptions]
29                SET    [subscriptions].[sb_subscriber] =
30                       CASE
31                        WHEN (PATINDEX('%[^0-9]%',
32                              [inserted].[sb_subscriber]) = 0)
33                        THEN [inserted].[sb_subscriber]
34                        ELSE [subscriptions].[sb_subscriber]
35                       END,
36                       [subscriptions].[sb_book] =
37                       CASE
38                        WHEN (PATINDEX('%[^0-9]%',
39                              [inserted].[sb_book]) = 0)
40                        THEN [inserted].[sb_book]
41                        ELSE [subscriptions].[sb_book]
42                       END,
43                       [subscriptions].[sb_start] = [inserted].[sb_start],
44                       [subscriptions].[sb_finish] = [inserted].[sb_finish],
45                       [subscriptions].[sb_is_active] =
6                                      [inserted].[sb_is_active]
47                FROM   [subscriptions]
48                  JOIN [inserted]
49                    ON [subscriptions].[sb_id] = [inserted].[sb_id];
50               END
51              END
52   GO
```

Let's see how the following data update queries work. Queries 1-3 are successful, but queries 4-6 are not, because queries 4-5 are trying to pass non-digital values of reader's name and book's title, and query 6 is trying to update the value of the primary key.

MS SQL Solution 3.2.2.a (update operation functionality check)

```
-- Query 1 (update works here):
UPDATE [subscriptions_with_text]
SET    [sb_start] = '2021-01-12'
WHERE  [sb_id] = 5000;

-- Query 2 (update works here):
UPDATE [subscriptions_with_text]
SET    [sb_subscriber] = 3
WHERE  [sb_id] = 5000;

-- Query 3 (update works here):
UPDATE [subscriptions_with_text]
SET    [sb_book] = 4
WHERE  [sb_id] = 5000;

-- Query 4 (update does NOT work here):
UPDATE [subscriptions_with_text]
SET    [sb_subscriber] = N'Subscriber'
WHERE  [sb_id] = 5000;

-- Query 5 (update does NOT work here):
UPDATE [subscriptions_with_text]
SET    [sb_book] = N'Book'
WHERE  [sb_id] = 5000;

-- Query 6 (update does NOT work here):
UPDATE [subscriptions_with_text]
SET    [sb_id] = 5001
WHERE  [sb_id] = 5000;
```

Let's create a trigger to implement the data deletion operation. Note how much shorter and simpler this code is than the above mentioned triggers for inserting and updating data. But, unfortunately, there will be much more problems with this trigger.

MS SQL Solution 3.2.2.a (creating a trigger to implement a deletion operation)

```
CREATE TRIGGER [subscriptions_with_text_del]
ON [subscriptions_with_text]
INSTEAD OF DELETE
AS
 DELETE FROM [subscriptions]
 WHERE [sb_id] IN (SELECT [sb_id]
                   FROM   [deleted]);
GO
```

The first problem is that MS SQL Server fills the `[deleted]` table with data before passing control to the trigger. So, there is no way we can intercept the situation of passing strictly digital data in `sb_subscriber` and `sb_book` fields in `DELETE` query, and such situation leads to query execution error with resolution: cannot convert {text value} to {digital value}.

The second problem is that passing the reader's name or book's title as a number (identifier) represented by a string leads to "zero matches found" (which is quite logical, since the view retrieves readers' names and books' titles, not identifiers). Passing string (text) readers' names and/or book' titles allows us to find matches, but does not guarantee that we have found the right strings (recall: we may have readers with the same names and books with the same titles).

As already mentioned, there is nothing we can do about the first problem, yet the second one has a solution, which is neither the most reliable nor the most pretty, but it works anyway: we can get the code of the query inside the trigger, whose execution activated the trigger, and check if the **sb_subscriber** and/or **sb_book** fields are mentioned in the query. If they are mentioned, we will cancel the operation.

MS SQL Solution 3.2.2.a (creating a trigger to implement a deletion operation; a more complex option)

```
CREATE TRIGGER [subscriptions_with_text_del]
ON [subscriptions_with_text]
INSTEAD OF DELETE
AS
 -- Attempting to determine if the
 -- sb_subscriber and/or sb_book fields are passed to the DELETE query:
 SET NOCOUNT ON;
 DECLARE @ExecStr VARCHAR(50), @Qry NVARCHAR(255);
 CREATE TABLE #inputbuffer
 (
  [EventType] NVARCHAR(30),
  [Parameters] INT,
  [EventInfo] NVARCHAR(255)
 );

 SET @ExecStr = 'DBCC INPUTBUFFER(' + STR(@@SPID) + ')';
 INSERT INTO #inputbuffer EXEC (@ExecStr);
 SET @Qry = LOWER((SELECT [EventInfo] FROM #inputbuffer));

 -- For debugging, you can uncomment the following line
 -- and make sure that it contains the query
 -- that triggered the trigger:
 -- PRINT(@Qry);

 IF    ((CHARINDEX ('sb_subscriber', @Qry) > 0)
    OR  (CHARINDEX ('sb_book', @Qry) > 0))
       BEGIN
         RAISERROR ('Deletion from [subscriptions_with_text] view
                     using [sb_subscriber] and/or [sb_book]
                     is prohibited.', 16, 1);
         ROLLBACK;
       END
 SET NOCOUNT OFF;

 -- This is where the deletion itself is performed:
 DELETE FROM [subscriptions]
 WHERE [sb_id] IN (SELECT [sb_id]
                   FROM   [deleted]);
GO
```

Let's check how deletion queries work. The comments to each of the queries explain in which cases they will be executed successfully or will end with one or another error. The result will depend on which of the above two versions of the trigger you implement.

MS SQL Solution 3.2.2.a (checking the functionality of the deletion operation)

```
1    -- Query 1 (delete works here):
2    DELETE FROM [subscriptions_with_text]
3    WHERE  [sb_id] < 10;
4
5    -- Query 2 (delete works here):
6    DELETE FROM [subscriptions_with_text]
7    WHERE  [sb_start] = '2012-01-12';
8
9    -- Query 3 (delete does NOT work here
10   -- in both versions of the trigger):
11   DELETE FROM [subscriptions_with_text]
12   WHERE  [sb_book] = 2;
13
14   -- Query 4 (zero matches found
15   -- in the first version of the trigger
16   -- and transaction rollback in the second one):
17   DELETE FROM [subscriptions_with_text]
18   WHERE  [sb_book] = '2';
19
20   -- Query 5 (the first version of the trigger
21   -- may find a match, while the second version of the trigger
22   -- will always rollback the transaction):
23   DELETE FROM [subscriptions_with_text]
24   WHERE  [sb_book] = N'Eugene Onegin';
```

Let's move on to solving this problem for Oracle. The code of the view itself is the same as for MySQL and MS SQL Server:

Oracle Solution 3.2.2.a (view creation)

```
1    CREATE OR REPLACE VIEW "subscriptions_with_text"
2    AS
3    SELECT "sb_id",
4           "s_name" AS "sb_subscriber",
5           "b_name" AS "sb_book",
6           "sb_start",
7           "sb_finish",
8           "sb_is_active"
9    FROM   "subscriptions"
10          JOIN "subscribers" ON "sb_subscriber" = "s_id"
11          JOIN "books" ON "sb_book" = "b_id"
```

Let's create a trigger to implement the data insertion operation.

The check logic (lines 5-13) is similar in MS SQL Server and Oracle: initial **`subscriptions`** table contains digital reader's and book's identifiers in **`sb_subscriber`** and **`sb_book`** fields, so we must get the inserted values of these fields in the digital form, hence, we prohibit the insertion of non-digital data.

Algorithmically we perform the same actions in both DBMS, and the only difference is in the regular expression functions and error message generation.

The main part of the trigger, responsible directly for the insertion of data in Oracle is again a bit simpler than in MS SQL Server due to the ability to handle each row separately.

The mechanism for managing auto-incrementable primary keys (based on a sequence (**SEQUENCE**) and a separate trigger) allows us not to worry about how to get a new value of the primary key.

Oracle — Solution 3.2.2.a (create a trigger to implement the insert operation)

```
1    CREATE OR REPLACE TRIGGER "subscriptions_with_text_ins"
2      INSTEAD OF INSERT ON "subscriptions_with_text"
3      FOR EACH ROW
4       BEGIN
5        IF    ((REGEXP_INSTR(:new."sb_subscriber", '[^0-9]') > 0)
6          OR  (REGEXP_INSTR(:new."sb_book", '[^0-9]') > 0))
7        THEN
8         RAISE_APPLICATION_ERROR(-20001, 'Use digital identifiers for
9                                         "sb_subscriber" and "sb_book".
10                                        Do not use subscribers'' names
11                                        or books'' titles');
12        ROLLBACK;
13       END IF;
14       INSERT INTO "subscriptions"
15                   ("sb_id",
16                    "sb_subscriber",
17                    "sb_book",
18                    "sb_start",
19                    "sb_finish",
20                    "sb_is_active")
21       VALUES      (:new."sb_id",
22                    :new."sb_subscriber",
23                    :new."sb_book",
24                    :new."sb_start",
25                    :new."sb_finish",
26                    :new."sb_is_active");
27     END;
```

Let's check how the following insertion queries are executed. Queries 1 and 2 are expectedly executed correctly, but queries 3-5 activate code in lines 5-14 of the trigger and cancel the transaction, because passing non-digital values to the **sb_subscriber** and/or **sb_book** fields is forbidden.

Oracle — Solution 3.2.2.a (checking the functionality of the insert operation)

```
1    -- Query 1 (insert works here):
2    INSERT INTO "subscriptions_with_text"
3                ("sb_id",
4                 "sb_subscriber",
5                 "sb_book",
6                 "sb_start",
7                 "sb_finish",
8                 "sb_is_active")
9    VALUES      (5000,
10                1,
11                3,
12                TO_DATE('2015-01-12', 'YYYY-MM-DD'),
13                TO_DATE('2015-02-12', 'YYYY-MM-DD'),
14                'N');
15
16   -- Query 2 (insert works here):
17   INSERT INTO "subscriptions_with_text"
18               ("sb_subscriber",
19                "sb_book",
20                "sb_start",
21                "sb_finish",
22                "sb_is_active")
23   VALUES      (1,
24                3,
25                TO_DATE('2015-01-12', 'YYYY-MM-DD'),
26                TO_DATE('2015-02-12', 'YYYY-MM-DD'),
27                'N');
```

Oracle — Solution 3.2.2.a (checking the functionality of the insert operation) (continued)

```
28  -- Query 3 (insert does NOT work here):
29  INSERT INTO "subscriptions_with_text"
30              ("sb_subscriber",
31               "sb_book",
32               "sb_start",
33               "sb_finish",
34               "sb_is_active")
35  VALUES      (N'Ivanov I.I.',
36               3,
37               TO_DATE('2015-01-12', 'YYYY-MM-DD'),
38               TO_DATE('2015-02-12', 'YYYY-MM-DD'),
39               'N');
40
41  -- Query 4 (insert does NOT work here):
42  INSERT INTO "subscriptions_with_text"
43              ("sb_subscriber",
44               "sb_book",
45               "sb_start",
46               "sb_finish",
47               "sb_is_active")
48  VALUES      (1,
49               N'Some book',
50               TO_DATE('2015-01-12', 'YYYY-MM-DD'),
51               TO_DATE('2015-02-12', 'YYYY-MM-DD'),
52               'N');
53
54  -- Query 5 (insert does NOT work here):
55  INSERT INTO "subscriptions_with_text"
56              ("sb_subscriber",
57               "sb_book",
58               "sb_start",
59               "sb_finish",
60               "sb_is_active")
61  VALUES      (N'Some subscriber',
62               N'Some book',
63               TO_DATE('2015-01-12', 'YYYY-MM-DD'),
64               TO_DATE('2015-02-12', 'YYYY-MM-DD'),
65               'N');
```

Let's create a trigger that allows us to implement the data update operation.

As in the MS SQL Server solution, we have to react to non-digital values in **sb_subscriber** and **sb_book** fields only if these values were explicitly passed in the query, this causes the need for a more complex (than in the **INSERT** trigger) condition in lines 5-8.

On lines 19-30 we check again if the digital value of fields **sb_subscriber** and **sb_book** came in the new data set and use the old value available in the **subscriptions** table if the new one is not a number.

In lines 23 and 29 the "trick" of adding (concatenating) empty string to the original value of field in Unicode representation is applied, which leads to automatic conversion of data type to Unicode string. This avoids a compilation error in the trigger caused by a data type mismatch between the **sb_subscriber** and **sb_book** fields in the **subscriptions** table (**NUMBER**) and the **subscriptions_with_text** (**NVARCHAR2**) view. The resulting string representation of a number is automatically converted to **NUMBER** without errors and is used correctly to insert it into the **subscriptions** table

In contrast to MS SQL Server, where the entire set of processed data is passed to the trigger, in Oracle our trigger processes each updated row separately, and therefore we know exactly the old and new values of the primary key. This allows us to get rid of the prohibition on updating the primary key value.

Oracle Solution 3.2.2.a (create a trigger to implement the update operation)

```
CREATE OR REPLACE TRIGGER "subscriptions_with_text_upd"
  INSTEAD OF UPDATE ON "subscriptions_with_text"
  FOR EACH ROW
   BEGIN
    IF      ((:old."sb_subscriber" != :new."sb_subscriber")
        AND  (REGEXP_INSTR(:new."sb_subscriber", '[^0-9]') > 0))
        OR  ((:old."sb_book" != :new."sb_book")
        AND  (REGEXP_INSTR(:new."sb_book", '[^0-9]') > 0))
    THEN
     RAISE_APPLICATION_ERROR(-20001, 'Use digital identifiers for
                                      "sb_subscriber" and "sb_book".
                                      Do not use subscribers'' names
                                      or books'' titles');
     ROLLBACK;
    END IF;

    UPDATE "subscriptions"
    SET    "sb_id" = :new."sb_id",
           "sb_subscriber" =
                  CASE
                   WHEN (REGEXP_INSTR(:new."sb_subscriber", '[^0-9]') = 0)
                   THEN :new."sb_subscriber"
                   ELSE "sb_subscriber" || N''
                  END,
           "sb_book" =
                  CASE
                   WHEN (REGEXP_INSTR(:new."sb_book", '[^0-9]') = 0)
                   THEN :new."sb_book"
                   ELSE "sb_book" || N''
                  END,
           "sb_start" = :new."sb_start",
           "sb_finish" = :new."sb_finish",
           "sb_is_active" = :new."sb_is_active"
    WHERE  "sb_id" = :old."sb_id";
  END;
```

Let's check how the following data update queries work. Queries 1-3 and 6 are executed successfully (query 6 is not executed in MS SQL Server because the primary key value is updated), but queries 4-5 are not, because they are trying to pass non-digital values of reader name and book title.

Oracle Solution 3.2.2.a (checking the functionality of the update operation)

```
-- Query 1 (update works here):
UPDATE "subscriptions_with_text"
SET    "sb_start" = TO_DATE('2021-01-12', 'YYYY-MM-DD')
WHERE  "sb_id" = 101;

-- Query 2 (update works here):
UPDATE "subscriptions_with_text"
SET    "sb_subscriber" = 3
WHERE  "sb_id" = 101;

-- Query 3 (update works here):
UPDATE "subscriptions_with_text"
SET    "sb_book" = 4
WHERE  "sb_id" = 101;

-- Query 4 (update does NOT work here):
UPDATE "subscriptions_with_text"
SET    "sb_subscriber" = N'Subscriber'
WHERE  "sb_id" = 101;
```

Oracle Solution 3.2.2.a (checking the functionality of the update operation) (continued)

```
20    -- Query 5 (update does NOT work here):
21    UPDATE "subscriptions_with_text"
22    SET    "sb_book" = N'Book'
23    WHERE  "sb_id" = 101;
24
25    -- Query 6 (update works here):
26    UPDATE "subscriptions_with_text"
27    SET    "sb_id" = 1001
28    WHERE  "sb_id" = 101;
```

Let's create a trigger to implement data deletion operation. Its code differs from the code of similar trigger for MS SQL Server only by the logic of defining identifiers of deleted records.

Oracle Solution 3.2.2.a (create a trigger to implement the delete operation)

```
1     CREATE OR REPLACE TRIGGER "subscriptions_with_text_del"
2       INSTEAD OF DELETE ON "subscriptions_with_text"
3       FOR EACH ROW
4        BEGIN
5         DELETE FROM "subscriptions"
6         WHERE "sb_id" = :old."sb_id";
7        END;
```

Oracle (as well as MS SQL Server) performs a search operation to find the records corresponding to the deletion condition before passing control to the trigger. That is why there is no way we can intercept the situation when a strictly digital data in **sb_subscriber** and **sb_book** fields are passed to **DELETE** query, and such situation leads to an error of query execution with the "invalid digital value" resolution.

The second problem (as it happens with MS SQL Server) is that passing readers' names or books' titles as a number (identifier) represented by a string leads to "zero matches found", while passing string (text) readers' names and/or books' titles allows us to find matches, but does not guarantee that we find the right strings (recall: we may have readers with the same names and books with the same titles).

And while MS SQL Server allows us to solve the second problem by analyzing the SQL query that activated the trigger, in Oracle there is no way to get the SQL query text in triggers that react to data modification expressions (DML triggers). Thus, the **DELETE** trigger in solving this problem for Oracle is more harmful and dangerous than useful, it leads to unexpected error messages and allows us to accidentally delete some useful data.

Still, let's check how delete queries work.

Oracle Solution 3.2.2.a (checking the functionality of the delete operation)

```
1     -- Query 1 (delete works here):
2     DELETE FROM "subscriptions_with_text"
3     WHERE  "sb_id" = 103;
4
5     -- Query 2 (delete works here):
6     DELETE FROM "subscriptions_with_text"
7     WHERE  "sb_start" = TO_DATE('2011-01-12', 'YYYY-MM-DD');
8
9     -- Query 3 (delete does NOT work here:
10    -- type conversion error):
11    DELETE FROM "subscriptions_with_text"
12    WHERE  "sb_book" = 2;
```

Oracle Solution 3.2.2.a (checking the functionality of the delete operation) (continued)

```
13    -- Query 4 (no matches found):
14    DELETE FROM "subscriptions_with_text"
15    WHERE  "sb_book" = '2';
16
17    -- Query 5 (it is possible to accidently delete different
18    -- books with the same names):
19    DELETE FROM "subscriptions_with_text"
20    WHERE  "sb_book" = N'Eugene Onegin';
```

Solution 3.2.2.b{271}.

The data sampling logic in this view is a simplified version of the solution{79} of problem 2.2.2.b{74}, and the triggers themselves, compared to the previous example, are many times simpler.

Since MySQL does not allow us to create triggers on views, the most we can do is to create a view itself, but we cannot modify data with it:

MySQL Solution 3.2.2.b (view creation)

```
1     CREATE OR REPLACE VIEW `books_with_genres`
2     AS
3     SELECT `b_id`,
4            `b_name`,
5            GROUP_CONCAT(`g_name`) AS `genres`
6     FROM   `books`
7            JOIN `m2m_books_genres` USING(`b_id`)
8            JOIN `genres` USING(`g_id`)
9     GROUP BY `b_id`
```

In MS SQL Server, the code of the view itself looks like this (see the explanation of the logic of obtaining the desired result in the solution{79} of problem 2.2.2.b{74}):

MS SQL Solution 3.2.2.b (view creation)

```
1     CREATE OR ALTER VIEW [books_with_genres]
2     AS
3     WITH [prepared_data]
4          AS (SELECT [books].[b_id],
5                     [b_name],
6                              [g_name]
7               FROM   [books]
8                      JOIN [m2m_books_genres]
9                        ON [books].[b_id] = [m2m_books_genres].[b_id]
10                     JOIN [genres]
11                       ON [m2m_books_genres].[g_id] = [genres].[g_id]
12             )
13    SELECT [outer].[b_id],
14           [outer].[b_name],
15           STUFF ((SELECT DISTINCT ',' + [inner].[g_name]
16                   FROM   [prepared_data] AS [inner]
17                   WHERE  [outer].[b_id] = [inner].[b_id]
18                   ORDER  BY ',' + [inner].[g_name]
19                   FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
20                   1, 1, '')
21           AS [genres]
22    FROM   [prepared_data] AS [outer]
23    GROUP  BY [outer].[b_id],
24              [outer].[b_name]
```

Let's create a trigger to implement the data insertion operation.

MS SQL Solution 3.2.2.b (create a trigger to implement the insert operation)

```
CREATE TRIGGER [books_with_genres_ins]
ON [books_with_genres]
INSTEAD OF INSERT
AS
 INSERT INTO [genres]
             ([g_name])
 SELECT [genres]
 FROM   [inserted];
GO
```

Yes, that's it. We won't be able to pass the genre identifier using this view (there is no corresponding field in the view), nor will we be able to add new books for the same reason. And adding a new genre is really that primitive.

Let's move on to the solution for Oracle. We'll create a view (see the explanation of the logic for obtaining the desired result in the solution{79} of problem 2.2.2.b{74}):

Oracle Solution 3.2.2.b (view creation)

```
CREATE OR REPLACE VIEW "books_with_genres"
AS
SELECT "b_id", "b_name",
       UTL_RAW.CAST_TO_NVARCHAR2
       (
          LISTAGG
          (
              UTL_RAW.CAST_TO_RAW("g_name"),
              UTL_RAW.CAST_TO_RAW(N',')
          )
          WITHIN GROUP (ORDER BY "g_name")
       )
       AS "genres"
       FROM "books"
       JOIN "m2m_books_genres" USING ("b_id")
       JOIN "genres" USING ("g_id")
       GROUP  BY "b_id",
                 "b_name"
```

Let's create a trigger to implement the data insertion operation.

Oracle Solution 3.2.2.b (create a trigger to implement the insert operation)

```
CREATE OR REPLACE TRIGGER "books_with_genres_ins"
INSTEAD OF INSERT ON "books_with_genres"
FOR EACH ROW
  BEGIN
    INSERT INTO "genres"
                ("g_name")
    VALUES      (:new."genres");
  END;
```

As we can see from the code of the trigger, the solution for Oracle is just as primitive for the reasons described above in the solution for MS SQL Server.

Task 3.2.2.TSK.A: create a view that extracts human-readable information (with books' titles and authors' names instead of identifiers) from the **`m2m_books_authors`** table, and at the same time allows us to modify data in that table (in case of non-unique books' titles and authors' names in both cases use a record with the minimum value of the primary key).

Task 3.2.2.TSK.B: create a view that shows a list of books with their authors, and at the same time allows us to add new authors.

Chapter 4: Using Triggers

4.1. Using Triggers to Aggregate Data

4.1.1. Example 31: Using Triggers to Update Caching Tables and Fields

The solutions to the problems presented in examples 27{229}, 29{259} and 30{271} provide additional information for this example.

Problem 4.1.1.a{286}: modify the “Library” database schema so that the `subscribers` table keeps up-to-date information about the date of the subscriber’s last visit to the library.

Problem 4.1.1.b{295}: create a caching table `averages`, which contains the following actual information at any time:
a) How many books are on average in the reader’s hands;
b) How long it takes a reader to read a book on average (in days);
c) How many books a reader has read on average.

Expected result 4.1.1.a.

The `subscribers` table contains an additional field that stores current information about the date of the last visit to the library:

s_id	s_name	s_last_visit
1	Ivanov I.I.	2015-10-07
2	Petrov P.P.	NULL
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

Expected result 4.1.1.b.

The `averages` table contains the following information that is relevant at any given time:

books_taken	days_to_read	books_returned
1.2500	46.0000	1.5000

Solution 4.1.1.a{286}.

To solve this problem we will need to perform three steps:

- modify the `subscribers` table (by adding a field to store the date of the last visit of a reader);
- initialize the values of last visits for all readers;
- create triggers to keep this information up to date.

Important: in MySQL (even in version 8) triggers are not activated by cascade operations, so changes in `subscriptions` table caused by deleting books will remain “invisible” to triggers on this table. In task 4.1.1.TSK.D{305} you are invited to improve this solution by eliminating this problem.

For MySQL, the first two steps are performed using the following queries.

MySQL Solution 4.1.1.a (table modification and data initialization)

```
1    -- Table modification:
2    ALTER TABLE `subscribers`
3      ADD COLUMN `s_last_visit` DATE NULL DEFAULT NULL AFTER `s_name`;
4
5    -- Data initialization:
6    UPDATE `subscribers`
7           LEFT JOIN (SELECT `sb_subscriber`,
8                             MAX(`sb_start`) AS `last_visit`
9                      FROM   `subscriptions`
10                     GROUP  BY `sb_subscriber`) AS `prepared_data`
11                ON `s_id` = `sb_subscriber`
12   SET    `s_last_visit` = `last_visit`;
```

Since the value of the date of the reader's last visit depends on the information presented in the **`subscriptions`** table (specifically, on the value of the **`sb_start`** field), it is on this table that we have to create triggers.

Basically, we could use the same solution in all three triggers (**`UPDATE`** based on **`JOIN`**), but for a change in the **`INSERT`** trigger we will implement a simpler option: check if the date of the added subscription was greater than the saved date of the last visit and, if so, update the date of the last visit.

In **`UPDATE`** and **`DELETE`** triggers this solution is not suitable: if as a result of these operations it turns out that the reader has never been in the library, the date of his last visit should be **`NULL`**. This result is easiest to achieve with **`UPDATE`** based on **`JOIN`**.

For **`UPDATE`** and **`DELETE`** operations it is critically important to use exactly **`AFTER`** triggers, since **`BEFORE`** triggers will work with "old" data, which will lead to incorrect determination of the date being searched for.

Also note that in the **`UPDATE`** trigger we have to update the date of the last visit for two potentially different readers, because when we make a change, the subscription can be "transferred" from one reader to another (we will consider this situation when we check if this solution works).

MySQL Solution 4.1.1.a (creating triggers)

```
1    -- Removing old versions of triggers
2    -- (handy during development and debugging):
3    DROP TRIGGER `last_visit_on_subscriptions_ins`;
4    DROP TRIGGER `last_visit_on_subscriptions_upd`;
5    DROP TRIGGER `last_visit_on_subscriptions_del`;
6
7    -- Switching the query completion delimiter,
8    -- since the query now will create a trigger,
9    -- inside which there are their own, classic queries:
10   DELIMITER $$
```

MySQL Solution 4.1.1.a (creating triggers) (continued)

```
11  -- Creating a trigger that reacts to the addition of a subscription:
12  CREATE TRIGGER `last_visit_on_subscriptions_ins`
13  AFTER INSERT
14  ON `subscriptions`
15   FOR EACH ROW
16    BEGIN
17     IF (SELECT IFNULL(`s_last_visit`, '1970-01-01')
18         FROM   `subscribers`
19         WHERE  `s_id` = NEW.`sb_subscriber`) < NEW.`sb_start`
20     THEN
21      UPDATE `subscribers`
22      SET    `s_last_visit` = NEW.`sb_start`
23      WHERE  `s_id` = NEW.`sb_subscriber`;
24     END IF;
25    END;
26  $$
27
28  -- Creating a trigger that reacts to an update of a subscription:
29  CREATE TRIGGER `last_visit_on_subscriptions_upd`
30  AFTER UPDATE
31  ON `subscriptions`
32   FOR EACH ROW
33    BEGIN
34     UPDATE `subscribers`
35     LEFT JOIN (SELECT `sb_subscriber`,
36                       MAX(`sb_start`) AS `last_visit`
37                FROM   `subscriptions`
38                GROUP  BY `sb_subscriber`) AS `prepared_data`
39             ON `s_id` = `sb_subscriber`
40     SET    `s_last_visit` = `last_visit`
41     WHERE  `s_id` IN (OLD.`sb_subscriber`, NEW.`sb_subscriber`);
42    END;
43  $$
44
45  -- Creating a trigger that reacts to the deletion of a subscription:
46  CREATE TRIGGER `last_visit_on_subscriptions_del`
47  AFTER DELETE
48  ON `subscriptions`
49   FOR EACH ROW
50    BEGIN
51     UPDATE `subscribers`
52     LEFT JOIN (SELECT `sb_subscriber`,
53                       MAX(`sb_start`) AS `last_visit`
54                FROM   `subscriptions`
55                GROUP  BY `sb_subscriber`) AS `prepared_data`
56             ON `s_id` = `sb_subscriber`
57     SET    `s_last_visit` = `last_visit`
58     WHERE  `s_id` = OLD.`sb_subscriber`;
59    END;
60  $$
61
62  -- Restoring the query completion delimiter:
63  DELIMITER ;
```

Let's test the functionality of this solution. We will change data in the **`subscriptions`** table and monitor changes in the **`subscribers`** table.

First, let's add a subscription to the reader with identifier 2 (they have never been in the library before):

MySQL Solution 4.1.1.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (200,
             2,
             1,
             '2019-01-12',
             '2019-02-12',
             'N')
```

s_id	s_name	s_last_visit
1	Ivanov I.I.	2015-10-07
2	Petrov P.P.	2019-01-12
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

Now let's emulate the situation "the book was mistakenly registered to another reader" and change the reader identifier in the just added subscription from 2 to 1. **NULL** value of the date of the last visit of Petrov P.P. is correctly restored:

MySQL Solution 4.1.1.a (functionality check)

```
UPDATE `subscriptions`
SET    `sb_subscriber` = 1
WHERE  `sb_id` = 200
```

s_id	s_name	s_last_visit
1	Ivanov I.I.	2019-01-12
2	Petrov P.P.	NULL
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

Again, let's add the subscriptions to Petrov P. P.:

MySQL Solution 4.1.1.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (201,
             2,
             1,
             '2020-01-12',
             '2020-02-12',
             'N')
```

s_id	s_name	s_last_visit
1	Ivanov I.I.	2019-01-12
2	Petrov P.P.	2020-01-12
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

Let's change the value of the date of the previously corrected subscription (which we re-registered from Petrov P. P. to Ivanov I. I.):

MySQL Solution 4.1.1.a (functionality check)

```
UPDATE `subscriptions`
SET    `sb_start` = '2018-01-12'
WHERE  `sb_id` = 200
```

s_id	s_name	s_last_visit
1	Ivanov I.I.	2018-01-12
2	Petrov P.P.	2020-01-12
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

Let's delete this subscription:

MySQL Solution 4.1.1.a (functionality check)

```
DELETE FROM `subscriptions`
WHERE  `sb_id` = 200
```

s_id	s_name	s_last_visit
1	Ivanov I.I.	2015-10-07
2	Petrov P.P.	2020-01-12
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

Let's delete the only subscription of Petrov P.P.:

MySQL Solution 4.1.1.a (functionality check)

```
DELETE FROM `subscriptions`
WHERE  `sb_id` = 201
```

s_id	s_name	s_last_visit
1	Ivanov I.I.	2015-10-07
2	Petrov P.P.	NULL
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

So, the solution for MySQL is ready and works correctly. But before we move on to the solution for MS SQL Server, let's do a little exploration that clearly demonstrates the answer to the question "is the **BEFORE** trigger canceled in case of an error during the operation"?

Exploration 4.4.1.EXP.A. Will the data changes caused by the **BEFORE** trigger be invalidated if the operation that activated the trigger fails to complete successfully?

Let's change the **INSERT** trigger from **AFTER** to **BEFORE**:

MySQL Exploration 4.1.1.a (changing the trigger type)

```
DROP TRIGGER `last_visit_on_subscriptions_ins`;

DELIMITER $$

CREATE TRIGGER `last_visit_on_subscriptions_ins`
BEFORE INSERT
ON `subscriptions`
 FOR EACH ROW
  BEGIN
   IF (SELECT IFNULL(`s_last_visit`, '1970-01-01')
       FROM   `subscribers`
       WHERE  `s_id` = NEW.`sb_subscriber`) < NEW.`sb_start`
   THEN
    UPDATE `subscribers`
    SET    `s_last_visit` = NEW.`sb_start`
    WHERE  `s_id` = NEW.`sb_subscriber`;
   END IF;
  END;
$$

DELIMITER ;
```

Let's sequentially add two subscriptions with different dates, but the same values of primary keys:

MySQL Exploration 4.1.1.a (data insertion)

```
INSERT INTO `subscriptions`
VALUES      (500,
             2,
             1,
             '2020-01-12',
             '2020-02-12',
             'N');

INSERT INTO `subscriptions`
VALUES      (500,
             2,
             1,
             '2021-01-12',
             '2021-02-12',
             'N');
```

Now, let's check the data in the **`subscribers`** table:

s_id	s_name	s_last_visit
1	Ivanov I.I.	2015-10-07
2	Petrov P.P.	2020-01-12
3	Sidorov S.S.	2014-08-03
4	Sidorov S.S.	2015-10-08

So, the changes are cancelled if the operation cannot be completed successfully. Otherwise, the date of the last visit of Petrov P.P. would be 2021-01-12.

Let's move on to solve the problem for MS SQL Server. We'll modify the **`subscribers`** table and initialize the added field with data.

MS SQL Solution 4.1.1.a (table modification and data initialization)

```
-- Table modification:
ALTER TABLE [subscribers]
  ADD [s_last_visit] DATE NULL DEFAULT NULL;

-- Data initialization:
UPDATE [subscribers]
SET    [s_last_visit] = [last_visit]
FROM   [subscribers]
       LEFT JOIN (SELECT [sb_subscriber],
                         MAX([sb_start]) AS [last_visit]
                  FROM   [subscriptions]
                  GROUP  BY [sb_subscriber]) AS [prepared_data]
              ON [s_id] = [sb_subscriber];
```

MS SQL Server supports a very convenient syntax for creating a trigger on several operations at once, so (unlike the MySQL solution) we use the same trigger body code for all three cases (**`INSERT`**, **`UPDATE`**, **`DELETE`**):

MS SQL Solution 4.1.1.a (creating triggers)

```
CREATE TRIGGER [last_visit_on_subscriptions_ins_upd_del]
ON [subscriptions]
AFTER INSERT, UPDATE, DELETE
AS
 UPDATE [subscribers]
 SET    [s_last_visit] = [last_visit]
 FROM   [subscribers]
       LEFT JOIN (SELECT [sb_subscriber],
                         MAX([sb_start]) AS [last_visit]
                  FROM   [subscriptions]
                  GROUP  BY [sb_subscriber]) AS [prepared_data]
              ON [s_id] = [sb_subscriber];
```

We can verify the correctness of the obtained solution by running the following queries (they are performed step by step with explanations and displaying the results in the solution for MySQL):

MS SQL Solution 4.1.1.a (queries to verify functionality)

```
SET IDENTITY_INSERT [subscriptions] ON;

-- Adding a subscription to a reader with identifier 2
-- (they have never been in the library before):
INSERT INTO [subscriptions]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (200,
             2,
             1,
             '2019-01-12',
             '2019-02-12',
             'N');

-- Changing the reader identifier in the just
-- added subscription from 2 to 1:
UPDATE [subscriptions]
SET    [sb_subscriber] = 1
WHERE  [sb_id] = 200;

-- Adding one more subscription to Petrov P.P.
-- (reader identifier = 2):
INSERT INTO [subscriptions]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (201,
             2,
             1,
             '2020-01-12',
             '2020-02-12',
             'N');

-- Changing the date value of the previously corrected
-- subscription (which was re-registered from Petrov P.P.
-- to Ivanov I.I.):
UPDATE [subscriptions]
SET    [sb_start] = '2018-01-12'
WHERE  [sb_id] = 200;
```

MS SQL — Solution 4.1.1.a (queries to verify functionality) (continued)

```
47    -- Removing this corrected subscription:
48    DELETE FROM [subscriptions]
49    WHERE  [sb_id] = 200;
50
51    -- Removing the only subscription of Petrov P.P.:
52    DELETE FROM [subscriptions]
53    WHERE  [sb_id] = 201;
54
55    SET IDENTITY_INSERT [subscriptions] OFF;
```

Let's move on to the solution for Oracle, which differs from the solution for MS SQL Server only in the syntax features of the implementation of the same logic:

Oracle — Solution 4.1.1.a (table modification and data initialization)

```
1     -- Table modification:
2     ALTER TABLE "subscribers"
3       ADD ("s_last_visit" DATE DEFAULT NULL NULL);
4
5     -- Data initialization:
6     UPDATE "subscribers" "outer"
7     SET    "s_last_visit" =
8            (
9             SELECT    "last_visit"
10            FROM      "subscribers"
11            LEFT JOIN (SELECT   "sb_subscriber",
12                               MAX("sb_start") AS "last_visit"
13                        FROM     "subscriptions"
14                        GROUP BY "sb_subscriber") "prepared_data"
15            ON         "s_id" = "sb_subscriber"
16            WHERE "outer"."s_id" = "sb_subscriber");
```

Oracle — Solution 4.1.1.a (creating triggers)

```
1     CREATE TRIGGER "last_visit_on_scs_ins_upd_del"
2     AFTER INSERT OR UPDATE OR DELETE
3     ON "subscriptions"
4     BEGIN
5      UPDATE "subscribers" "outer"
6      SET    "s_last_visit" =
7             (
8              SELECT    "last_visit"
9              FROM      "subscribers"
10             LEFT JOIN (SELECT   "sb_subscriber",
11                                MAX("sb_start") AS "last_visit"
12                         FROM     "subscriptions"
13                         GROUP BY "sb_subscriber") "prepared_data"
14             ON         "s_id" = "sb_subscriber"
15             WHERE "outer"."s_id" = "sb_subscriber");
16    END;
```

In this solution, we used Oracle's ability to create so called "statement-level triggers", which work similarly to MS SQL Server triggers: such trigger is activated once after the entire operation, not for each modified row separately, as it happens, for example, in MySQL, where only "row-level triggers" are supported, which are activated separately for each modified row.

We can verify the correctness of the obtained solution by running the following queries (they are performed step by step with explanations and displaying the results in the solution for MySQL):

Oracle | Solution 4.1.1.a (queries to verify functionality)

```
ALTER TRIGGER "TRG_subscriptions_sb_id" DISABLE;

-- Adding a subscription to a reader with identifier 2
-- (they have never been in the library before):
INSERT INTO "subscriptions"
            ("sb_id",
             "sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
VALUES      (200,
             2,
             1,
             TO_DATE('2019-01-12', 'YYYY-MM-DD'),
             TO_DATE('2019-02-12', 'YYYY-MM-DD'),
             'N');

-- Changing the reader identifier in the just
-- added subscription from 2 to 1:
UPDATE "subscriptions"
SET    "sb_subscriber" = 1
WHERE  "sb_id" = 200;

-- Adding one more subscription to Petrov P.P.
-- (reader identifier = 2):
INSERT INTO "subscriptions"
            ("sb_id",
             "sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
VALUES      (201,
             2,
             1,
             TO_DATE('2020-01-12', 'YYYY-MM-DD'),
             TO_DATE('2020-02-12', 'YYYY-MM-DD'),
             'N');

-- Changing the date value of the previously corrected
-- subscription (which was re-registered from Petrov P.P.
-- to Ivanov I.I.):
UPDATE "subscriptions"
SET    "sb_start" = TO_DATE('2018-01-12', 'YYYY-MM-DD')
WHERE  "sb_id" = 200;

-- Removing this corrected subscription:
DELETE FROM "subscriptions"
WHERE  "sb_id" = 200;

-- Removing the only subscription of Petrov P.P.:
DELETE FROM "subscriptions"
WHERE  "sb_id" = 201;

ALTER TRIGGER "TRG_subscriptions_sb_id" ENABLE;
```

This completes the solution of this problem.

Solution 4.1.1.b{286}.

In many ways, the solution of this problem is similar to the solution{230} of problem 3.1.2.a{229}, which it is recommended to recall before continuing reading.

As usual, let's start with MySQL. Let's create an aggregating table:

MySQL Solution 4.1.1.b (creating an aggregating table)

```
CREATE TABLE `averages`
(
        `books_taken` DOUBLE NOT NULL,
        `days_to_read` DOUBLE NOT NULL,
        `books_returned` DOUBLE NOT NULL
)
```

Let's initialize the data in the created table:

MySQL Solution 4.1.1.b (cleaning the table and data initialization)

```
-- Cleaning the table:
TRUNCATE TABLE `averages`;

-- Data initialization:
INSERT INTO `averages`
            (`books_taken`,
             `days_to_read`,
             `books_returned`)
SELECT ( `active_count` / `subscribers_count` )   AS `books_taken`,
       ( `days_sum` / `inactive_count` )          AS `days_to_read`,
       ( `inactive_count` / `subscribers_count` ) AS `books_returned`
FROM   (SELECT COUNT(`s_id`) AS `subscribers_count`
        FROM   `subscribers`) AS `tmp_subscribers_count`,
       (SELECT COUNT(`sb_id`) AS `active_count`
        FROM   `subscriptions`
        WHERE  `sb_is_active` = 'Y') AS `tmp_active_count`,
       (SELECT COUNT(`sb_id`) AS `inactive_count`
        FROM   `subscriptions`
        WHERE  `sb_is_active` = 'N') AS `tmp_inactive_count`,
       (SELECT SUM(DATEDIFF(`sb_finish`, `sb_start`)) AS `days_sum`
        FROM   `subscriptions`
        WHERE  `sb_is_active` = 'N') AS `tmp_days_sum`;
```

Let's create triggers that modify the data in the aggregating table. The aggregation is based on the information presented in the **`subscribers`** and **`subscriptions`** tables, so we have to create triggers for both of these tables.

In order not to complicate the solution, we will use the same code for all five triggers (the **`subscribers`** table should have only **`INSERT`** and **`DELETE`** triggers, because updating this table does not affect the results of calculations; and the **`subscriptions`** table should have all three triggers: **`INSERT`**, **`UPDATE`**, **`DELETE`**).

Important: in MySQL (even in version 8) triggers are not activated by cascade operations, so changes in the **`subscriptions`** table caused by deleting books will remain "invisible" to triggers on this table. In task 4.1.1.TSK.E{305} you are invited to improve this solution, eliminating this problem.

MySQL | Solution 4.1.1.b (triggers on the **subscribers** table)

```mysql
-- Removal of old versions of triggers
-- (handy during development and debugging):
DROP TRIGGER `upd_avgs_on_subscribers_ins`;
DROP TRIGGER `upd_avgs_on_subscribers_del`;

-- Switching the query completion delimiter,
-- because now the query will create a trigger,
-- inside which there are their own, classic queries:
DELIMITER $$

-- Creating a trigger that reacts to the addition of readers:
CREATE TRIGGER `upd_avgs_on_subscribers_ins`
AFTER INSERT
ON `subscribers`
 FOR EACH ROW
  BEGIN
   UPDATE `averages`,
          (SELECT COUNT(`s_id`) AS `subscribers_count`
           FROM   `subscribers`) AS `tmp_subscribers_count`,
          (SELECT COUNT(`sb_id`) AS `active_count`
           FROM   `subscriptions`
           WHERE  `sb_is_active` = 'Y') AS `tmp_active_count`,
          (SELECT COUNT(`sb_id`) AS `inactive_count`
           FROM   `subscriptions`
           WHERE  `sb_is_active` = 'N') AS `tmp_inactive_count`,
          (SELECT SUM(DATEDIFF(`sb_finish`, `sb_start`)) AS `days_sum`
           FROM   `subscriptions`
           WHERE  `sb_is_active` = 'N') AS `tmp_days_sum`
    SET   `books_taken` = `active_count` / `subscribers_count`,
          `days_to_read` = `days_sum` / `inactive_count`,
          `books_returned` = `inactive_count` / `subscribers_count`;
  END;
$$

-- Creating a trigger that reacts to the deletion of readers:
CREATE TRIGGER `upd_avgs_on_subscribers_del`
AFTER DELETE
ON `subscribers`
 FOR EACH ROW
  BEGIN
   UPDATE `averages`,
          (SELECT COUNT(`s_id`) AS `subscribers_count`
           FROM   `subscribers`) AS `tmp_subscribers_count`,
          (SELECT COUNT(`sb_id`) AS `active_count`
           FROM   `subscriptions`
           WHERE  `sb_is_active` = 'Y') AS `tmp_active_count`,
          (SELECT COUNT(`sb_id`) AS `inactive_count`
           FROM   `subscriptions`
           WHERE  `sb_is_active` = 'N') AS `tmp_inactive_count`,
          (SELECT SUM(DATEDIFF(`sb_finish`, `sb_start`)) AS `days_sum`
           FROM   `subscriptions`
           WHERE  `sb_is_active` = 'N') AS `tmp_days_sum`
    SET   `books_taken` = `active_count` / `subscribers_count`,
          `days_to_read` = `days_sum` / `inactive_count`,
          `books_returned` = `inactive_count` / `subscribers_count`;
  END;
$$

-- Restoring the query completion delimiter:
DELIMITER ;
```

Let's create triggers on the **`subscriptions`** table.

MySQL | Solution 4.1.1.b (triggers on the **`subscriptions`** table)

```
1    -- Removal of old versions of triggers
2    -- (handy during development and debugging):
3    DROP TRIGGER `upd_avgs_on_subscriptions_ins`;
4    DROP TRIGGER `upd_avgs_on_subscriptions_upd`;
5    DROP TRIGGER `upd_avgs_on_subscriptions_del`;
6
7    -- Switching the query completion delimiter,
8    -- because now the query will create a trigger,
9    -- inside which there are their own, classic queries:
10   DELIMITER $$
11
12   -- Creating a trigger that reacts to the addition of a subscription:
13   CREATE TRIGGER `upd_avgs_on_subscriptions_ins`
14   AFTER INSERT
15   ON `subscriptions`
16    FOR EACH ROW
17     BEGIN
18      UPDATE `averages`,
19             (SELECT COUNT(`s_id`) AS `subscribers_count`
20              FROM   `subscribers`) AS `tmp_subscribers_count`,
21             (SELECT COUNT(`sb_id`) AS `active_count`
22              FROM   `subscriptions`
23              WHERE  `sb_is_active` = 'Y') AS `tmp_active_count`,
24             (SELECT COUNT(`sb_id`) AS `inactive_count`
25              FROM   `subscriptions`
26              WHERE  `sb_is_active` = 'N') AS `tmp_inactive_count`,
27             (SELECT SUM(DATEDIFF(`sb_finish`, `sb_start`)) AS `days_sum`
28              FROM   `subscriptions`
29              WHERE  `sb_is_active` = 'N') AS `tmp_days_sum`
30      SET    `books_taken` = `active_count` / `subscribers_count`,
31             `days_to_read` = `days_sum` / `inactive_count`,
32             `books_returned` = `inactive_count` / `subscribers_count`;
33     END;
34   $$
35
36   -- Creating a trigger that reacts to an update of a subscription:
37   CREATE TRIGGER `upd_avgs_on_subscriptions_upd`
38   AFTER UPDATE
39   ON `subscriptions`
40    FOR EACH ROW
41     BEGIN
42      UPDATE `averages`,
43             (SELECT COUNT(`s_id`) AS `subscribers_count`
44              FROM   `subscribers`) AS `tmp_subscribers_count`,
45             (SELECT COUNT(`sb_id`) AS `active_count`
46              FROM   `subscriptions`
47              WHERE  `sb_is_active` = 'Y') AS `tmp_active_count`,
48             (SELECT COUNT(`sb_id`) AS `inactive_count`
49              FROM   `subscriptions`
50              WHERE  `sb_is_active` = 'N') AS `tmp_inactive_count`,
51             (SELECT SUM(DATEDIFF(`sb_finish`, `sb_start`)) AS `days_sum`
52              FROM   `subscriptions`
53              WHERE  `sb_is_active` = 'N') AS `tmp_days_sum`
45      SET    `books_taken` = `active_count` / `subscribers_count`,
55             `days_to_read` = `days_sum` / `inactive_count`,
56             `books_returned` = `inactive_count` / `subscribers_count`;
57     END;
58   $$
```

MySQL Solution 4.1.1.b (triggers on the `subscriptions` table) (continued)

```
59  -- Creating a trigger that reacts to the deletion of a subscription:
60  CREATE TRIGGER `upd_avgs_on_subscriptions_del`
61  AFTER DELETE
62  ON `subscriptions`
63   FOR EACH ROW
64    BEGIN
65     UPDATE `averages`,
66            (SELECT COUNT(`s_id`) AS `subscribers_count`
67             FROM   `subscribers`) AS `tmp_subscribers_count`,
68            (SELECT COUNT(`sb_id`) AS `active_count`
69             FROM   `subscriptions`
70             WHERE  `sb_is_active` = 'Y') AS `tmp_active_count`,
71            (SELECT COUNT(`sb_id`) AS `inactive_count`
72             FROM   `subscriptions`
73             WHERE  `sb_is_active` = 'N') AS `tmp_inactive_count`,
74            (SELECT SUM(DATEDIFF(`sb_finish`, `sb_start`)) AS `days_sum`
75             FROM   `subscriptions`
76             WHERE  `sb_is_active` = 'N') AS `tmp_days_sum`
77       SET  `books_taken` = `active_count` / `subscribers_count`,
78            `days_to_read` = `days_sum` / `inactive_count`,
79            `books_returned` = `inactive_count` / `subscribers_count`;
80    END;
81  $$
82
83  -- Restoring the query completion delimiter:
84  DELIMITER ;
```

Let's check the functionality of the obtained solution. We will change the data in the **`subscribers`** and **`subscriptions`** tables and monitor changes in the **`averages`** table.

The initial state of the **`averages`** table is as follows:

books_taken	days_to_read	books_returned
1.25	46	1.5

Let's add a reader:

MySQL Solution 4.1.1.b (functionality check)

```
1  INSERT INTO `subscribers`
2              (`s_id`,
3               `s_name`)
4  VALUES      (500,
5               'Reader R.R.')
```

books_taken	days_to_read	books_returned
1	46	1.2

Now, let's delete that reader:

MySQL Solution 4.1.1.b (functionality check)

```
1  DELETE FROM `subscribers`
2  WHERE  `s_id` = 500
```

books_taken	days_to_read	books_returned
1.25	46	1.5

Let's add two subscriptions:

MySQL Solution 4.1.1.b (functionality check)

```
INSERT INTO `subscriptions`
            (`sb_id`,
             `sb_subscriber`,
             `sb_book`,
             `sb_start`,
             `sb_finish`,
             `sb_is_active`)
VALUES      (200,
             1,
             1,
             '2019-01-12',
             '2019-02-12',
             'N'),
            (201,
             2,
             1,
             '2020-01-12',
             '2020-02-12',
             'N')
```

books_taken	days_to_read	books_returned
1.25	42.25	2

Let's change the status of added subscriptions from "book returned" to "book not returned":

MySQL Solution 4.1.1.b (functionality check)

```
UPDATE `subscriptions`
SET    `sb_is_active` = 'Y'
WHERE  `sb_id` >= 200
```

books_taken	days_to_read	books_returned
1.75	46	1.5

Let's delete these two subscriptions:

MySQL Solution 4.1.1.b (functionality check)

```
DELETE FROM `subscriptions`
WHERE  `sb_id` >= 200
```

books_taken	days_to_read	books_returned
1.25	46	1.5

So, the triggers for MySQL work correctly. Let's move on to the solution for MS SQL Server.

MS SQL Solution 4.1.1.b (creating an aggregating table)

```
CREATE TABLE [averages]
(
        [books_taken] DOUBLE PRECISION NOT NULL,
        [days_to_read] DOUBLE PRECISION NOT NULL,
        [books_returned] DOUBLE PRECISION NOT NULL
)
```

Let's initialize the data in the created table. Note: here again the problem of data type conversion is relevant because the result of division will be integer (with loss of some data), if we do not first convert obtained values of **COUNT** and **SUM** to doubles.

MS SQL Solution 4.1.1.b (cleaning the table and data initialization)

```
1    -- Cleaning the table:
2    TRUNCATE TABLE [averages];
3
4    -- Data initialization:
5    INSERT INTO [averages]
6                ([books_taken],
7                 [days_to_read],
8                 [books_returned])
9    SELECT ( [active_count] / [subscribers_count] )   AS [books_taken],
10          ( [days_sum] / [inactive_count] )          AS [days_to_read],
11          ( [inactive_count] / [subscribers_count] ) AS [books_returned]
12   FROM   (SELECT CAST(COUNT([s_id]) AS DOUBLE PRECISION)
13           AS [subscribers_count]
14           FROM   [subscribers]) AS [tmp_subscribers_count],
15          (SELECT CAST(COUNT([sb_id]) AS DOUBLE PRECISION)
16           AS [active_count]
17           FROM   [subscriptions]
18           WHERE  [sb_is_active] = 'Y') AS [tmp_active_count],
19          (SELECT CAST(COUNT([sb_id]) AS DOUBLE PRECISION)
20           AS [inactive_count]
21           FROM   [subscriptions]
22           WHERE  [sb_is_active] = 'N') AS [tmp_inactive_count],
23          (SELECT CAST(SUM(DATEDIFF(day, [sb_start], [sb_finish]))
24           AS DOUBLE PRECISION) AS [days_sum]
25           FROM   [subscriptions]
26           WHERE  [sb_is_active] = 'N') AS [tmp_days_sum];
```

Let's create triggers on **`subscribers`** and **`subscriptions`** tables that modify data in the aggregating table **`averages`**. Recall that MS SQL Server allows us to specify in the trigger several activating events at once, which allows us to reduce the amount of written code.

MS SQL Solution 4.1.1.b (triggers on the **`subscribers`** table)

```
1    CREATE TRIGGER [upd_avgs_on_subscribers_ins_del]
2    ON [subscribers]
3    AFTER INSERT, DELETE
4    AS
5       UPDATE [averages]
6       SET    [books_taken] = [active_count] / [subscribers_count],
7              [days_to_read] = [days_sum] / [inactive_count],
8              [books_returned] = [inactive_count] / [subscribers_count]
9       FROM   (SELECT CAST(COUNT([s_id]) AS DOUBLE PRECISION)
10              AS [subscribers_count]
11              FROM   [subscribers]) AS [tmp_subscribers_count],
12             (SELECT CAST(COUNT([sb_id]) AS DOUBLE PRECISION)
13              AS [active_count]
14              FROM   [subscriptions]
15              WHERE  [sb_is_active] = 'Y') AS [tmp_active_count],
16             (SELECT CAST(COUNT([sb_id]) AS DOUBLE PRECISION)
17              AS [inactive_count]
18              FROM   [subscriptions]
19              WHERE  [sb_is_active] = 'N') AS [tmp_inactive_count],
20             (SELECT CAST(SUM(DATEDIFF(day, [sb_start], [sb_finish]))
21              AS DOUBLE PRECISION) AS [days_sum]
22              FROM   [subscriptions]
23              WHERE  [sb_is_active] = 'N') AS [tmp_days_sum];
```

MS SQL Solution 4.1.1.b (triggers on the `subscriptions` table)

```
CREATE TRIGGER [upd_avgs_on_subscriptions_ins_upd_del]
ON [subscriptions]
AFTER INSERT, UPDATE, DELETE
AS
   UPDATE [averages]
   SET    [books_taken] = [active_count] / [subscribers_count],
          [days_to_read] = [days_sum] / [inactive_count],
          [books_returned] = [inactive_count] / [subscribers_count]
   FROM   (SELECT CAST(COUNT([s_id]) AS DOUBLE PRECISION)
           AS [subscribers_count]
           FROM   [subscribers]) AS [tmp_subscribers_count],
          (SELECT CAST(COUNT([sb_id]) AS DOUBLE PRECISION)
           AS [active_count]
           FROM   [subscriptions]
           WHERE  [sb_is_active] = 'Y') AS [tmp_active_count],
          (SELECT CAST(COUNT([sb_id]) AS DOUBLE PRECISION)
           AS [inactive_count]
           FROM   [subscriptions]
           WHERE  [sb_is_active] = 'N') AS [tmp_inactive_count],
          (SELECT CAST(SUM(DATEDIFF(day, [sb_start], [sb_finish]))
           AS DOUBLE PRECISION) AS [days_sum]
           FROM   [subscriptions]
           WHERE  [sb_is_active] = 'N') AS [tmp_days_sum];
```

We can verify the correctness of the obtained solution by running the following queries (they are performed step by step with explanations and displaying the results in the solution for MySQL):

MS SQL Solution 4.1.1.b (queries to verify functionality)

```
SET IDENTITY_INSERT [subscribers] ON;

-- Adding a reader:
INSERT INTO [subscribers]
            ([s_id],
             [s_name])
VALUES      (500,
             N'Reader R.R.');

-- Deleting a just added reader:
DELETE FROM [subscribers]
WHERE  [s_id] = 500;

SET IDENTITY_INSERT [subscribers] OFF;
SET IDENTITY_INSERT [subscriptions] ON;

-- Adding two subscriptions:
INSERT INTO [subscriptions]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (200,
             1,
             1,
             '2019-01-12',
             '2019-02-12',
             'N'),
            (201,
             2,
             1,
             '2020-01-12',
             '2020-02-12',
             'N');
```

MS SQL — Solution 4.1.1.b (queries to verify functionality) (continued)

```
37   -- Changing the status of added subscriptions from "book returned"
38   -- to "book not returned":
39   UPDATE [subscriptions]
40   SET    [sb_is_active] = 'Y'
41   WHERE  [sb_id] >= 200;
42
43   -- Deletion of just added subscriptions:
44   DELETE FROM [subscriptions]
45   WHERE  [sb_id] >= 200;
46
47   SET IDENTITY_INSERT [subscriptions] OFF;
```

Let's move on to the solution for Oracle, which differs from the solution for MS SQL Server only in the syntax features of the implementation of the same logic:

Oracle — Solution 4.1.1.b (creating an aggregating table)

```
1    CREATE TABLE "averages"
2    (
3           "books_taken" DOUBLE PRECISION NOT NULL,
4           "days_to_read" DOUBLE PRECISION NOT NULL,
5           "books_returned" DOUBLE PRECISION NOT NULL
6    )
```

Oracle — Solution 4.1.1.b (cleaning the table and data initialization)

```
1    -- Cleaning the table:
2    TRUNCATE TABLE "averages";
3
4    -- Data initialization:
5    INSERT INTO "averages"
6                ("books_taken",
7                 "days_to_read",
8                 "books_returned")
9    SELECT ( "active_count" / "subscribers_count" )    AS "books_taken",
10          ( "days_sum" / "inactive_count" )           AS "days_to_read",
11          ( "inactive_count" / "subscribers_count" ) AS "books_returned"
12   FROM    (SELECT COUNT("s_id")  AS "subscribers_count"
13            FROM   "subscribers") "tmp_subscribers_count",
14           (SELECT COUNT("sb_id") AS "active_count"
15            FROM   "subscriptions"
16            WHERE  "sb_is_active" = 'Y') "tmp_active_count",
17           (SELECT COUNT("sb_id") AS "inactive_count"
18            FROM   "subscriptions"
19            WHERE  "sb_is_active" = 'N') "tmp_inactive_count",
20           (SELECT SUM("sb_finish" - "sb_start") AS "days_sum"
21            FROM   "subscriptions"
22            WHERE  "sb_is_active" = 'N') "tmp_days_sum";
```

Despite the fact that the logic of triggers in Oracle is completely identical to the approaches used in MySQL and MS SQL Server, due to the syntax peculiarities of this DBMS, the data update request itself looks somewhat unusual. Here we use the **`MERGE`** operator, specifying as a condition of join as **`1=1`**, i.e., knowingly fulfilling equality.

Oracle Solution 4.1.1.b (triggers on the `subscribers` table)

```
1    CREATE TRIGGER "upd_avgs_on_sbrs_ins_del"
2    AFTER INSERT OR DELETE
3    ON "subscribers"
4    BEGIN
5     MERGE INTO "averages"
6     USING
7     (
8      SELECT ( "active_count" / "subscribers_count" )    AS "books_taken",
9             ( "days_sum" / "inactive_count" )           AS "days_to_read",
10            ( "inactive_count" / "subscribers_count" ) AS "books_returned"
11     FROM    (SELECT COUNT("s_id")  AS "subscribers_count"
12              FROM   "subscribers") "tmp_subscribers_count",
13             (SELECT COUNT("sb_id") AS "active_count"
14              FROM   "subscriptions"
15              WHERE  "sb_is_active" = 'Y') "tmp_active_count",
16             (SELECT COUNT("sb_id") AS "inactive_count"
17              FROM   "subscriptions"
18              WHERE  "sb_is_active" = 'N') "tmp_inactive_count",
19             (SELECT SUM("sb_finish" - "sb_start") AS "days_sum"
20              FROM   "subscriptions"
21              WHERE  "sb_is_active" = 'N') "tmp_days_sum"
22     ) "tmp" ON (1=1)
23    WHEN MATCHED THEN UPDATE
24     SET    "averages"."books_taken" = "tmp"."books_taken",
25            "averages"."days_to_read" = "tmp"."days_to_read",
26            "averages"."books_returned" = "tmp"."books_returned";
27   END;
```

Oracle Solution 4.1.1.b (triggers on the `subscriptions` table)

```
1    CREATE TRIGGER "upd_avgs_on_sbps_ins_upd_del"
2    AFTER INSERT OR UPDATE OR DELETE
3    ON "subscriptions"
4    BEGIN
5     MERGE INTO "averages"
6     USING
7     (
8      SELECT ( "active_count" / "subscribers_count" )    AS "books_taken",
9             ( "days_sum" / "inactive_count" )           AS "days_to_read",
10            ( "inactive_count" / "subscribers_count" ) AS "books_returned"
11     FROM    (SELECT COUNT("s_id")  AS "subscribers_count"
12              FROM   "subscribers") "tmp_subscribers_count",
13             (SELECT COUNT("sb_id") AS "active_count"
14              FROM   "subscriptions"
15              WHERE  "sb_is_active" = 'Y') "tmp_active_count",
16             (SELECT COUNT("sb_id") AS "inactive_count"
17              FROM   "subscriptions"
18              WHERE  "sb_is_active" = 'N') "tmp_inactive_count",
19             (SELECT SUM("sb_finish" - "sb_start") AS "days_sum"
20              FROM   "subscriptions"
21              WHERE  "sb_is_active" = 'N') "tmp_days_sum"
22     ) "tmp" ON (1=1)
23    WHEN MATCHED THEN UPDATE
24     SET    "averages"."books_taken" = "tmp"."books_taken",
25            "averages"."days_to_read" = "tmp"."days_to_read",
26            "averages"."books_returned" = "tmp"."books_returned";
27   END;
```

As in the solution{286} of problem 4.1.1.a{286}, here we used Oracle's ability to create so-called "statement-level triggers": such a trigger is activated once after the entire operation, not for each modifiable row separately, as it happens, e.g., in MySQL, where only "row-level triggers" are supported, which are activated separately for each modifiable row.

We can verify the correctness of the obtained solution by running the following queries (they are performed step by step with explanations and displaying the results in the solution for MySQL):

Oracle | Solution 4.1.1.b (queries to verify functionality)

```
ALTER TRIGGER "TRG_subscribers_s_id" DISABLE;
ALTER TRIGGER "TRG_subscriptions_sb_id" DISABLE;

-- Adding a reader:
INSERT INTO "subscribers"
            ("s_id",
             "s_name")
VALUES      (500,
             N'Reader R.R.');

-- Deleting a just added reader:
DELETE FROM "subscribers"
WHERE  "s_id" = 500;

-- Adding two subscriptions:
INSERT ALL
 INTO "subscriptions"
            ("sb_id",
             "sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
 VALUES     (200,
             1,
             1,
             TO_DATE('2019-01-12', 'YYYY-MM-DD'),
             TO_DATE('2019-02-12', 'YYYY-MM-DD'),
             'N')
 INTO "subscriptions"
            ("sb_id",
             "sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
 VALUES     (201,
             2,
             1,
             TO_DATE('2020-01-12', 'YYYY-MM-DD'),
             TO_DATE('2020-02-12', 'YYYY-MM-DD'),
             'N')
SELECT 1 FROM "DUAL";

-- Changing the status of added subscriptions from "book returned"
-- to "book not returned":
UPDATE "subscriptions"
SET    "sb_is_active" = 'Y'
WHERE  "sb_id" >= 200;

-- Deletion of just added subscriptions:
DELETE FROM "subscriptions"
WHERE  "sb_id" >= 200;

ALTER TRIGGER "TRG_subscribers_s_id" ENABLE;
ALTER TRIGGER "TRG_subscriptions_sb_id" ENABLE;
```

This completes the solution of this problem.

Task 4.1.1.TSK.A: modify the “Library” database schema so that the `authors` table stores current information about the date of the last subscription to the author’s book by a reader.

Task 4.1.1.TSK.B: create a caching table `best_averages` that contains the following actual information at any time:
a) How many books on average are in the hands of readers who have read more than 20 books during their time with the library.
b) How long it takes, on average, to read a book (in days) for a reader who has never kept a book in their possession for more than two weeks.
c) How many books are read on average by readers who have never had an overdue subscription.

Task 4.1.1.TSK.C: optimize the MySQL triggers from the solution{295} of problem 4.1.1.b{286} so that they do not perform extra actions where they are not needed (hint: not in every case we need all the data collected by the existing queries).

Task 4.1.1.TSK.D: modify the solution{286} of problem 4.1.1.a{286} for MySQL, so that it takes into account the changes in the `subscriptions` table caused by the cascade deletion operation (when deleting books). Make sure that solutions for MS SQL Server and Oracle don’t require such modification.

Task 4.1.1.TSK.E: modify the solution{295} of problem 4.1.1.b{286} for MySQL, so that it takes into account the changes in the `subscriptions` table caused by the cascade delete operation (when deleting books). Make sure that solutions for MS SQL Server and Oracle don’t require such modification.

4.1.2. Example 32: Using Triggers to Ensure Data Consistency

Problem 4.1.2.a{306}: modify the “Library” database schema so that the `subscribers` table stores information about how many books are currently given to each of the readers.

Problem 4.1.2.b{319}: modify the “Library” database schema so that the `genres` table stores information about how many books currently belong to each genre.

Expected result 4.1.2.a.

The `subscribers` table contains an additional field that stores current information about the number of books given to each reader:

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Expected result 4.1.2.b.

The `genres` table contains an additional field that stores current information about the number of books belonging to each genre:

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	0
5	Classic	4
6	Science Fiction	1

Solution 4.1.2.a{306}.

As in the solution{286} of problem 4.1.1.a{286} here we need to perform three steps:

- modify the `subscribers` table (by adding a field to store the number of books given to the reader);
- initialize the values of the number of given books for all readers;
- create triggers to keep this information up to date.

In contrast to the solution{286} of problem 4.1.1.a{286} here the default value for the new field is not `NULL` but `0` (because “the reader has never been to the library” is “unknown”, i.e., NULL, and “the reader has no books” is quite clear and understandable, i.e., `0`). Following the same logic, we use `JOIN` rather than `LEFT JOIN` in the initializing query, as there is no point in updating data for readers who have never borrowed books.

So, for MySQL, the first two steps are performed using the following queries.

MySQL Solution 4.1.2.a (table modification and data initialization)

```
1    -- Table modification:
2    ALTER TABLE `subscribers`
3      ADD COLUMN `s_books` INT(11) NOT NULL DEFAULT 0 AFTER `s_name`;
4
5    -- Data initialization:
6    UPDATE `subscribers`
7           JOIN (SELECT `sb_subscriber`,
8                        COUNT(`sb_id`) AS `s_has_books`
9                 FROM   `subscriptions`
10                WHERE  `sb_is_active` = 'Y'
11                GROUP  BY `sb_subscriber`) AS `prepared_data`
12          ON `s_id` = `sb_subscriber`
13   SET    `s_books` = `s_has_books`;
```

As we can see from the initializing query, we can get all the information needed to form the **`s_books`** field from the **`subscriptions`** table. We will create triggers on it.

The **`INSERT`** and **`DELETE`** triggers are simple: if we add or delete an "active" subscription (the **`sb_is_active`** field is **`Y`**), we must increase or decrease by one the value of the subscriptions counter for the corresponding reader.

MySQL Solution 4.1.2.a (triggers on the **`subscriptions`** table)

```
1    DELIMITER $$
2
3    -- Reaction to the addition of a subscription:
4    CREATE TRIGGER `s_has_books_on_subscriptions_ins`
5    AFTER INSERT
6    ON `subscriptions`
7     FOR EACH ROW
8      BEGIN
9       IF (NEW.`sb_is_active` = 'Y') THEN
10       UPDATE `subscribers`
11       SET    `s_books` = `s_books` + 1
12       WHERE  `s_id` = NEW.`sb_subscriber`;
13      END IF;
14     END;
15   $$
16
17   -- Reaction to the deletion of a subscription:
18   CREATE TRIGGER `s_has_books_on_subscriptions_del`
19   AFTER DELETE
20   ON `subscriptions`
21    FOR EACH ROW
22     BEGIN
23      IF (OLD.`sb_is_active` = 'Y') THEN
24       UPDATE `subscribers`
25       SET    `s_books` = `s_books` - 1
26       WHERE  `s_id` = OLD.`sb_subscriber`;
27      END IF;
28     END;
29   $$
30
31   DELIMITER ;
```

The situation with the **`UPDATE`** trigger will be more complicated because we have two parameters that can both change or remain unchanged: the reader's identifier and the status of the subscription.

MySQL | Solution 4.1.2.a (triggers on the `subscriptions` table)

```
DELIMITER $$

-- Reaction to the update of a subscription:
CREATE TRIGGER `s_has_books_on_subscriptions_upd`
AFTER UPDATE
ON `subscriptions`
 FOR EACH ROW
  BEGIN

    -- A) The reader is the same, Y -> N
    IF ((OLD.`sb_subscriber` = NEW.`sb_subscriber`) AND
        (OLD.`sb_is_active` = 'Y') AND
        (NEW.`sb_is_active` = 'N')) THEN
     UPDATE `subscribers`
     SET    `s_books` = `s_books` - 1
     WHERE  `s_id` = OLD.`sb_subscriber`;
    END IF;

    -- B) The reader is the same, N -> Y
    IF ((OLD.`sb_subscriber` = NEW.`sb_subscriber`) AND
        (OLD.`sb_is_active` = 'N') AND
        (NEW.`sb_is_active` = 'Y')) THEN
     UPDATE `subscribers`
     SET    `s_books` = `s_books` + 1
     WHERE  `s_id` = OLD.`sb_subscriber`;
    END IF;

    -- C) Readers are different, Y -> Y
    IF ((OLD.`sb_subscriber` != NEW.`sb_subscriber`) AND
        (OLD.`sb_is_active` = 'Y') AND
        (NEW.`sb_is_active` = 'Y')) THEN
     UPDATE `subscribers`
     SET    `s_books` = `s_books` - 1
     WHERE  `s_id` = OLD.`sb_subscriber`;
     UPDATE `subscribers`
     SET    `s_books` = `s_books` + 1
     WHERE  `s_id` = NEW.`sb_subscriber`;
    END IF;

    -- D) Readers are different, Y -> N
    IF ((OLD.`sb_subscriber` != NEW.`sb_subscriber`) AND
        (OLD.`sb_is_active` = 'Y') AND
        (NEW.`sb_is_active` = 'N')) THEN
     UPDATE `subscribers`
     SET    `s_books` = `s_books` - 1
     WHERE  `s_id` = OLD.`sb_subscriber`;
    END IF;

    -- E) Readers are different, N -> Y
    IF ((OLD.`sb_subscriber` != NEW.`sb_subscriber`) AND
        (OLD.`sb_is_active` = 'N') AND
        (NEW.`sb_is_active` = 'Y')) THEN
     UPDATE `subscribers`
     SET    `s_books` = `s_books` + 1
     WHERE  `s_id` = NEW.`sb_subscriber`;
    END IF;

  END;
$$

DELIMITER ;
```

Let's depict the considered situations graphically.

	Old sb_is_active value	New sb_is_active value	Action	Action code
Reader's identifier has remained unchanged	Y	Y	-	
	Y	N	OLD-1	A
	N	Y	OLD+1	B
	N	N	-	
Reader's identifier has changed	Y	Y	OLD-1, NEW+1	C
	Y	N	OLD-1	D
	N	Y	NEW+1	E
	N	N	-	

Finally, here is a solution to the problem described in 3.2.1.TSK.D{254}, 4.1.1.TSK.D{305}, 4.1.1.TSK.E{305}. Recall that MySQL does not activate triggers by cascade operations, so deleting books (which will lead to deletion of all subscriptions to those books) is "invisible" for **`DELETE`** trigger on **`subscriptions`** table.

To account for this effect, we will create an additional trigger on the **`books`** table, reacting to the deletion of books (inserting or updating data in the **`books`** table does not affect the distribution of existing books to these or those readers, so here is enough to create only **`DELETE`** trigger).

Note: here we create a **`BEFORE`** trigger, because at the moment the **`AFTER`** trigger is activated, the information we are looking for in the **`subscriptions`** table will already be deleted.

MySQL — Solution 4.1.2.a (trigger on the `books` table)

```
DELIMITER $$

-- Reaction to book deletion:
CREATE TRIGGER `s_has_books_on_books_del`
BEFORE DELETE
ON `books`
 FOR EACH ROW
  BEGIN
   UPDATE `subscribers`
   JOIN (SELECT    `sb_subscriber`,
                   COUNT(`sb_book`) AS `delta`
         FROM      `subscriptions`
         WHERE     `sb_book` = OLD.`b_id`
              AND `sb_is_active` = 'Y'
         GROUP BY `sb_subscriber`) AS `prepared_data`
         ON `s_id` = `sb_subscriber`
   SET   `s_books` = `s_books` - `delta`;
  END;
$$

DELIMITER ;
```

Let's check the functionality of this solution. We will change data in the **`books`** and **`subscriptions`** tables and monitor changes in the **`subscribers`** table.

The initial state of the **`subscribers`** table is as follows:

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's check the reaction to adding and removing subscriptions. Let's add an active subscription to Ivanov I.I., and an inactive one to Petrov P.P.:

MySQL Solution 4.1.2.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (200,
             1,
             1,
             '2011-01-12',
             '2011-02-12',
             'Y'),
            (201,
             2,
             1,
             '2011-01-12',
             '2011-02-12',
             'N')
```

s_id	s_name	s_books
1	Ivanov I.I.	1
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Now, let's delete the added subscriptions:

MySQL Solution 4.1.2.a (functionality check)

```
DELETE FROM `subscriptions`
WHERE  `sb_id` IN ( 200, 201 )
```

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's check the reaction to updating subscriptions. First. Let's add a subscription:

MySQL Solution 4.1.2.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (300,
             1,
             1,
             '2011-01-12',
             '2011-02-12',
             'Y')
```

s_id	s_name	s_books
1	Ivanov I.I.	1
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's make the subscription inactive without changing the reader's identifier:

MySQL Solution 4.1.2.a (functionality check)

```
1    -- A
2    UPDATE `subscriptions`
3    SET    `sb_is_active` = 'N'
4    WHERE  `sb_id` = 300
```

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's make the subscription active again without changing the reader's identifier:

MySQL Solution 4.1.2.a (functionality check)

```
1    -- B
2    UPDATE `subscriptions`
3    SET    `sb_is_active` = 'Y'
4    WHERE  `sb_id` = 300
```

s_id	s_name	s_books
1	Ivanov I.I.	1
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's change the reader's identifier without changing the subscription status:

MySQL Solution 4.1.2.a (functionality check)

```
1    -- C
2    UPDATE `subscriptions`
3    SET    `sb_subscriber` = 2
4    WHERE  `sb_id` = 300
```

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	1
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Now, let's change the reader's identifier and make the subscription inactive:

MySQL Solution 4.1.2.a (functionality check)

```
1    -- D
2    UPDATE `subscriptions`
3    SET    `sb_subscriber` = 1,
4           `sb_is_active` = 'N'
5    WHERE  `sb_id` = 300
```

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	0
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's change the reader's identifier and make the subscription active:

MySQL Solution 4.1.2.a (functionality check)

```
-- E
UPDATE `subscriptions`
SET    `sb_subscriber` = 2,
       `sb_is_active` = 'Y'
WHERE  `sb_id` = 300
```

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	1
3	Sidorov S.S.	3
4	Sidorov S.S.	2

Let's delete the book with identifier 1 (such a book is now given to Petrov and both Sidorovs one copy each):

MySQL Solution 4.1.2.a (functionality check)

```
DELETE FROM `books`
WHERE `b_id` = 1
```

s_id	s_name	s_books
1	Ivanov I.I.	0
2	Petrov P.P.	0
3	Sidorov S.S.	2
4	Sidorov S.S.	1

As the exploration has shown, all operations are performed correctly and result in correct changes in the values of the **s_books** field.

Let's move on to solve the problem for MS SQL Server. We'll modify the **subscribers** table and initialize the added field with data.

MS SQL Solution 4.1.2.a (table modification and data initialization)

```
-- Table modification:
ALTER TABLE [subscribers]
  ADD [s_books] INT NOT NULL DEFAULT 0;

-- Data initialization:
UPDATE [subscribers]
SET    [s_books] = [s_has_books]
FROM   [subscribers]
       JOIN (SELECT [sb_subscriber],
                    COUNT([sb_id]) AS [s_has_books]
             FROM   [subscriptions]
             WHERE  [sb_is_active] = 'Y'
             GROUP  BY [sb_subscriber]) AS [prepared_data]
       ON [s_id] = [sb_subscriber];
```

The logic of triggers in MS SQL Server will be different, because this DBMS does not support row-level triggers, and we have to process all the changes made at once.

It will be more or less easy with **INSERT** and **DELETE** triggers (as it was with MySQL): we have to find out the identifiers of readers who received (or returned) books, the number of such books for each reader, and then increase or decrease the counters of books given to corresponding readers by corresponding values.

MS SQL Solution 4.1.2.a (triggers on the `subscriptions` table)

```
1    -- Reaction to adding a subscription:
2    CREATE TRIGGER [s_has_books_on_subscriptions_ins]
3    ON [subscriptions]
4    AFTER INSERT
5    AS
6    UPDATE [subscribers]
7    SET    [s_books] = [s_books] + [s_new_books]
8    FROM   [subscribers]
9           JOIN (SELECT [sb_subscriber],
10                       COUNT([sb_id]) AS [s_new_books]
11                FROM   [inserted]
12                       WHERE  [sb_is_active] = 'Y'
13                GROUP  BY [sb_subscriber]) AS [prepared_data]
14          ON [s_id] = [sb_subscriber];
15   GO
16
17   -- Reaction to deleting a subscription:
18   CREATE TRIGGER [s_has_books_on_subscriptions_del]
19   ON [subscriptions]
20   AFTER DELETE
21   AS
22   UPDATE [subscribers]
23   SET    [s_books] = [s_books] - [s_old_books]
24   FROM   [subscribers]
25          JOIN (SELECT [sb_subscriber],
26                       COUNT([sb_id]) AS [s_old_books]
27                FROM   [deleted]
28                       WHERE  [sb_is_active] = 'Y'
29                GROUP  BY [sb_subscriber]) AS [prepared_data]
30          ON [s_id] = [sb_subscriber];
31
32   GO
```

The **`UPDATE`** trigger is a bit more complicated, but not as complicated as the one in MySQL: here we can calculate the number of given and returned books based on information from the **`[deleted]`** and **`[inserted]`** pseudo-tables, and we don't care about changing the status of subscriptions: we just count (independently) the number of given and returned books for each reader and change their books counters using these two values.

In other words, we do not care which state to which (and from which reader to which) a subscription is switched, we are only interested in "for whom the active subscriptions have been removed", and "for whom the active subscriptions have been added".

Therefore, the **`UPDATE`** trigger would simply contain the code from **`INSERT`** trigger and **`DELETE`** trigger. A slightly more elegant solution would be to implement such logic in the trigger body that it would be enough to perform only one operation to update the **`subscribers`** table: this is what task 4.1.2.TSK.C{328} will consist of.

To check the functionality of the obtained solution, we can use the queries presented after the **`UPDATE`** trigger code (their general logic and the expected DBMS reaction are explained in the MySQL solution but note that here we operate with a bit more data in each query).

MS SQL Solution 4.1.2.a (triggers on the `subscriptions` table)

```
-- Reaction to the update of a subscription:
CREATE TRIGGER [s_has_books_on_subscriptions_upd]
ON [subscriptions]
AFTER UPDATE
AS
-- (This is actually a DELETE trigger code):
UPDATE [subscribers]
SET    [s_books] = [s_books] - [s_old_books]
FROM   [subscribers]
       JOIN (SELECT [sb_subscriber],
                    COUNT([sb_id]) AS [s_old_books]
             FROM   [deleted]
                    WHERE  [sb_is_active] = 'Y'
             GROUP  BY [sb_subscriber]) AS [prepared_data]
       ON [s_id] = [sb_subscriber];
-- (This is actually an INSERT trigger code):
UPDATE [subscribers]
SET    [s_books] = [s_books] + [s_new_books]
FROM   [subscribers]
       JOIN (SELECT [sb_subscriber],
                    COUNT([sb_id]) AS [s_new_books]
             FROM   [inserted]
                    WHERE  [sb_is_active] = 'Y'
             GROUP  BY [sb_subscriber]) AS [prepared_data]
       ON [s_id] = [sb_subscriber];
GO
```

MS SQL Solution 4.1.2.a (functionality check)

```
SET IDENTITY_INSERT [subscriptions] ON;

-- Let's add two active subscriptions to Ivanov I.I,
-- and one active and one inactive to Petrov P.P.:
INSERT INTO [subscriptions]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (200,
             1,
             3,
             '2011-01-12',
             '2011-02-12',
             'Y'),
            (201,
             1,
             4,
             '2011-01-12',
             '2011-02-12',
             'Y'),
            (202,
             2,
             3,
             '2011-01-12',
             '2011-02-12',
             'Y'),
            (203,
             2,
             4,
             '2011-01-12',
             '2011-02-12',
             'N');
```

MS SQL Solution 4.1.2.a (functionality check) (continued)

```
36   -- Let's delete just added subscriptions:
37   DELETE FROM [subscriptions]
38   WHERE  [sb_id] IN (200, 201, 202, 203);
39
40   -- Let's check the reaction to subscriptions update.
41   -- First, let's add two subscriptions:
42   INSERT INTO [subscriptions]
43               ([sb_id],
44                [sb_subscriber],
45                [sb_book],
46                [sb_start],
47                [sb_finish],
48                [sb_is_active])
49   VALUES      (300,
50                1,
51                3,
52                '2011-01-12',
53                '2011-02-12',
54                'Y'),
55               (301,
56                1,
57                4,
58                '2011-01-12',
59                '2011-02-12',
60                'Y');
61
62   -- Without changing the reader's identifier let's deactivate subscriptions:
63   UPDATE [subscriptions]
64   SET    [sb_is_active] = 'N'
65   WHERE  [sb_id] IN (300, 301);
66
67   -- Without changing the reader's identifier let's reactivate subscriptions:
68   UPDATE [subscriptions]
69   SET    [sb_is_active] = 'Y'
70   WHERE  [sb_id] IN (300, 301);
71
72   -- Let's change the reader's identifier not changing subscriptions status:
73   UPDATE [subscriptions]
74   SET    [sb_subscriber] = 2
75   WHERE  [sb_id] IN (300, 301);
76
77   -- Let's change the reader's identifier and make subscriptions inactive:
78   UPDATE [subscriptions]
79   SET    [sb_subscriber] = 1,
80          [sb_is_active] = 'N'
81   WHERE  [sb_id] IN (300, 301);
82
83   -- Let's change the reader's identifier and make subscriptions active:
84   UPDATE [subscriptions]
85   SET    [sb_subscriber] = 2,
86          [sb_is_active] = 'Y'
87   WHERE  [sb_id] IN (300, 301);
88
89   -- Let's delete the book with identifier 1 (was given one copy each to
90   -- Petrov and both Sidorovs):
91   DELETE FROM [books]
92   WHERE [b_id] = 1;
93
94   SET IDENTITY_INSERT [subscriptions] OFF;
```

Let's move on to the solution of the problem for Oracle. We'll modify the **subscribers** table and initialize the added field with data.

Oracle Solution 4.1.2.a (table modification and data initialization)

```
1     -- Table modification:
2     ALTER TABLE "subscribers"
3       ADD ("s_books" INT DEFAULT 0 NOT NULL);
4
5     -- Data initialization:
6     UPDATE "subscribers"
7     SET    "s_books" = NVL(
8            (SELECT   COUNT("sb_id") AS "s_has_books"
9             FROM     "subscriptions"
10            WHERE    "sb_is_active" = 'Y'
11                 AND "sb_subscriber" = "s_id"
12            GROUP BY "sb_subscriber"), 0);
```

Note: because of Oracle syntax features forcing us to write such a query for update, we have to use **NVL** function because correlated subquery in rows 7-16 will be executed for each row of the **subscribers** table, and in some cases will return **NULL**.

Although Oracle (like MS SQL Server) supports statement-level triggers, we cannot use the logic presented in the MS SQL solution, because Oracle doesn't have **[inserted]** and **[updated]** pseudo-tables. We have to go the way of the MySQL solution and use row-level triggers.

Oracle Solution 4.1.2.a (triggers on the subscriptions table)

```
1     -- Reaction to adding a subscription:
2     CREATE OR REPLACE TRIGGER "s_has_books_on_sbps_ins"
3     AFTER INSERT
4     ON "subscriptions"
5     FOR EACH ROW
6      BEGIN
7        IF (:new."sb_is_active" = 'Y') THEN
8         UPDATE "subscribers"
9         SET    "s_books" = "s_books" + 1
10        WHERE  "s_id" = :new."sb_subscriber";
11       END IF;
12     END;
13
14    -- Reaction to deleting a subscription:
15    CREATE OR REPLACE TRIGGER "s_has_books_on_sbps_del"
16    AFTER DELETE
17    ON "subscriptions"
18    FOR EACH ROW
19     BEGIN
20       IF (:old."sb_is_active" = 'Y') THEN
21        UPDATE "subscribers"
22        SET    "s_books" = "s_books" - 1
23        WHERE  "s_id" = :old."sb_subscriber";
24       END IF;
25     END;
```

In the **UPDATE** trigger we also use exactly the same code that was used in the MySQL solution (all the situations that this trigger has to consider were also discussed and shown graphically there).

Oracle | Solution 4.1.2.a (triggers on the `subscriptions` table)

```
1     -- Reaction to the update of a subscription:
2     CREATE OR REPLACE TRIGGER "s_has_books_on_sbps_upd"
3     AFTER UPDATE
4     ON "subscriptions"
5     FOR EACH ROW
6      BEGIN
7        -- A) Reader is the same, Y -> N
8        IF ((:old."sb_subscriber" = :new."sb_subscriber") AND
9            (:old."sb_is_active" = 'Y') AND
10           (:new."sb_is_active" = 'N')) THEN
11        UPDATE "subscribers"
12        SET    "s_books" = "s_books" - 1
13        WHERE  "s_id" = :old."sb_subscriber";
14       END IF;
15
16       -- B) Reader is the same, N -> Y
17       IF ((:old."sb_subscriber" = :new."sb_subscriber") AND
18           (:old."sb_is_active" = 'N') AND
19           (:new."sb_is_active" = 'Y')) THEN
20        UPDATE "subscribers"
21        SET    "s_books" = "s_books" + 1
22        WHERE  "s_id" = :old."sb_subscriber";
23       END IF;
24
25       -- C) Readers are different, Y -> Y
26       IF ((:old."sb_subscriber" != :new."sb_subscriber") AND
27           (:old."sb_is_active" = 'Y') AND
28           (:new."sb_is_active" = 'Y')) THEN
29        UPDATE "subscribers"
30        SET    "s_books" = "s_books" - 1
31        WHERE  "s_id" = :old."sb_subscriber";
32        UPDATE "subscribers"
33        SET    "s_books" = "s_books" + 1
34        WHERE  "s_id" = :new."sb_subscriber";
35       END IF;
36
37       -- D) Readers are different, Y -> N
38       IF ((:old."sb_subscriber" != :new."sb_subscriber") AND
39           (:old."sb_is_active" = 'Y') AND
40           (:new."sb_is_active" = 'N')) THEN
41        UPDATE "subscribers"
42        SET    "s_books" = "s_books" - 1
43        WHERE  "s_id" = :old."sb_subscriber";
44       END IF;
45
46       -- E) Readers are different, N -> Y
47       IF ((:old."sb_subscriber" != :new."sb_subscriber") AND
48           (:old."sb_is_active" = 'N') AND
49           (:new."sb_is_active" = 'Y')) THEN
50        UPDATE "subscribers"
51        SET    "s_books" = "s_books" + 1
52        WHERE  "s_id" = :new."sb_subscriber";
53       END IF;
54     END;
```

As a result, the code for Oracle triggers is completely identical to the code for MySQL triggers, so the queries for testing the obtained solution are the same for both DBMSes.

See the code of the queries themselves below, and the logic of their work with an explanation and demonstration of changing the contents of the **`subscribers`** table is presented in the solution for MySQL.

Oracle | Solution 4.1.2.a (functionality check)

```
ALTER TRIGGER "TRG_subscriptions_sb_id" DISABLE;

-- Let's add an active subscription for Ivanov, and inactive for Petrov:
INSERT INTO "subscriptions"
VALUES      (200,
             1,
             1,
             TO_DATE('2011-01-12', 'YYYY-MM-DD'),
             TO_DATE('2011-02-12', 'YYYY-MM-DD'),
             'Y');
INSERT INTO "subscriptions"
VALUES      (201,
             2,
             1,
             TO_DATE('2011-01-12', 'YYYY-MM-DD'),
             TO_DATE('2011-02-12', 'YYYY-MM-DD'),
             'N');

-- Let's delete just added subscriptions:
DELETE FROM "subscriptions"
WHERE  "sb_id" IN ( 200, 201 );

-- Let's check the reaction to subscriptions update. Add a subscription:
INSERT INTO "subscriptions"
VALUES      (300,
             1,
             1,
             TO_DATE('2011-01-12', 'YYYY-MM-DD'),
             TO_DATE('2011-02-12', 'YYYY-MM-DD'),
             'Y');

-- A) Not changing the reader's identifier we deactivate the subscription:
UPDATE "subscriptions"
SET    "sb_is_active" = 'N'
WHERE  "sb_id" = 300;

-- B) Not changing the reader's identifier we reactivate the subscription:
UPDATE "subscriptions"
SET    "sb_is_active" = 'Y'
WHERE  "sb_id" = 300;

-- C) Now change the reader's identifier not changing subscriptions status:
UPDATE "subscriptions"
SET    "sb_subscriber" = 2
WHERE  "sb_id" = 300;

-- D) Let's change the reader's identifier and make subscription inactive:
UPDATE "subscriptions"
SET    "sb_subscriber" = 1,
       "sb_is_active" = 'N'
WHERE  "sb_id" = 300;

-- E) Let's change the reader's identifier and make subscriptions active:
UPDATE "subscriptions"
SET    "sb_subscriber" = 2,
       "sb_is_active" = 'Y'
WHERE  "sb_id" = 300;

-- Let's delete the book (id=1) (1 copy given to Petrov and both Sidorovs):
DELETE FROM [books]
WHERE [b_id] = 1;

ALTER TRIGGER "TRG_subscriptions_sb_id" ENABLE;
```

So, the solution of this problem is obtained and checked for all three DBMSes.

Solution 4.1.2.b{306}.

Just like in the solution{306} of problem 4.1.2.a{306} we'll need to do the same things here: modify the table, initialize it with data, create triggers. And even the code of triggers will be somewhat similar to the previously considered solutions.

Traditionally, we start with the MySQL solution: modify the table and initialize it with data.

MySQL — Solution 4.1.2.b (table modification and data initialization)

```
-- Table modification:
ALTER TABLE `genres`
  ADD COLUMN `g_books` INT(11) NOT NULL DEFAULT 0 AFTER `g_name`;

-- Data initialization:
UPDATE `genres`
       JOIN (SELECT `g_id`,
                    COUNT(`b_id`) AS `g_has_books`
             FROM   `m2m_books_genres`
             GROUP  BY `g_id`) AS `prepared_data`
       USING (`g_id`)
SET    `g_books` = `g_has_books`;
```

The code for all three triggers will be very simple: in the **`INSERT`** trigger we increase the book count for the corresponding genre, in the **`DELETE`** trigger we decrease it, in the **`UPDATE`** trigger we decrease for the "old" genre and increase for the "new" genre. No additional checks or tricks are required here.

MySQL — Solution 4.1.2.b (triggers on the `m2m_books_genres` table)

```
DELIMITER $$

-- Reaction to adding relationships between books and genres:
CREATE TRIGGER `g_has_books_on_m2m_b_g_ins`
AFTER INSERT
ON `m2m_books_genres`
 FOR EACH ROW
  BEGIN
   UPDATE `genres`
   SET    `g_books` = `g_books` + 1
   WHERE  `g_id` = NEW.`g_id`;
  END;
$$

-- Reaction to updating the relationship between books and genres:
CREATE TRIGGER `g_has_books_on_m2m_b_g_upd`
AFTER UPDATE
ON `m2m_books_genres`
 FOR EACH ROW
  BEGIN
   UPDATE `genres`
   SET    `g_books` = `g_books` - 1
   WHERE  `g_id` = OLD.`g_id`;
   UPDATE `genres`
   SET    `g_books` = `g_books` + 1
   WHERE  `g_id` = NEW.`g_id`;
  END;
$$
```

MySQL Solution 4.1.2.b (triggers on the `m2m_books_genres` table) (continued)

```
29  -- Reaction to deletion the relationship between books and genres:
30  CREATE TRIGGER `g_has_books_on_m2m_b_g_del`
31  AFTER DELETE
32  ON `m2m_books_genres`
33   FOR EACH ROW
34    BEGIN
35     UPDATE `genres`
36     SET    `g_books` = `g_books` - 1
37     WHERE  `g_id` = OLD.`g_id`;
38    END;
39  $$
40
41  DELIMITER ;
```

Since in MySQL triggers are not activated by cascade operations, deleting a book (which will delete all its relationships with all genres) will remain invisible to the triggers on **`m2m_books_genres`** table. So, we have to create a trigger on the **`books`** table that takes into account the relevant situation.

Each book is related to each genre no more than once, so when we delete any book we need to decrease by one the book counter of each of the genres it is related to.

MySQL Solution 4.1.2.b (trigger on the `books` table)

```
1   DELIMITER $$
2
3   -- Reaction to book deletion:
4   CREATE TRIGGER `g_has_books_on_books_del`
5   BEFORE DELETE
6   ON `books`
7    FOR EACH ROW
8     BEGIN
9      UPDATE `genres`
10     SET    `g_books` = `g_books` - 1
11     WHERE  `g_id` IN (SELECT `g_id`
12                       FROM   `m2m_books_genres`
13                       WHERE  `b_id` = OLD.`b_id`);
14    END;
15  $$
16
17  DELIMITER ;
```

Let's check the correctness of the obtained solution. We will modify data in the **`m2m_books_genres`** and **`books`** tables and check changes in the **`genres`** table.

Initial state of the **`genres`** table:

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	0
5	Classic	4
6	Science Fiction	1

Let's add two relationships to the "Science" genre (identifier is 4):

MySQL Solution 4.1.2.b (functionality check)

```
INSERT INTO `m2m_books_genres`
            (`b_id`,
             `g_id`)
VALUES      (1, 4),
            (2, 4)
```

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	2
5	Classic	4
6	Science Fiction	1

Let's change the values of the book identifiers in these relationships without changing the values of the genre identifiers:

MySQL Solution 4.1.2.b (functionality check)

```
UPDATE `m2m_books_genres`
SET    `b_id` = 3
WHERE  `b_id` = 1
   AND `g_id` = 4;

UPDATE `m2m_books_genres`
SET    `b_id` = 4
WHERE  `b_id` = 2
   AND `g_id` = 4;
```

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	2
5	Classic	4
6	Science Fiction	1

Let's change the values of genre identifiers in these relationships without changing the values of book identifiers:

MySQL Solution 4.1.2.b (functionality check)

```
UPDATE `m2m_books_genres`
SET    `g_id` = 5
WHERE  `b_id` = 3
   AND `g_id` = 4;

UPDATE `m2m_books_genres`
SET    `g_id` = 5
WHERE  `b_id` = 4
   AND `g_id` = 4;
```

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	0
5	Classic	6
6	Science Fiction	1

Let's change the values of genre identifiers and book identifiers simultaneously in these relationships:

MySQL Solution 4.1.2.b (functionality check)

```
UPDATE `m2m_books_genres`
SET    `b_id` = 1,
       `g_id` = 4
WHERE  `b_id` = 3
   AND `g_id` = 5;

UPDATE `m2m_books_genres`
SET    `b_id` = 2,
       `g_id` = 4
WHERE  `b_id` = 4
   AND `g_id` = 5;
```

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	2
5	Classic	4
6	Science Fiction	1

Now, let's delete these relationships that we've just created:

MySQL Solution 4.1.2.b (functionality check)

```
DELETE FROM `m2m_books_genres`
WHERE  `b_id` = 1
   AND `g_id` = 4;

DELETE FROM `m2m_books_genres`
WHERE  `b_id` = 2
   AND `g_id` = 4;
```

g_id	g_name	g_books
1	Poetry	2
2	Programming	3
3	Psychology	1
4	Science	0
5	Classic	4
6	Science Fiction	1

Finally, let's delete the books with identifiers 1 and 2 (both belonging to the "Poetry" and "Classic" genres at the same time):

MySQL Solution 4.1.2.b (functionality check)

```
DELETE FROM `books`
WHERE `b_id` IN (1, 2)
```

g_id	g_name	g_books
1	Poetry	0
2	Programming	3
3	Psychology	1
4	Science	0
5	Classic	2
6	Science Fiction	1

So, the solution for MySQL is completed and tested.

Let's move on to the solution for MS SQL Server. We'll modify the table and initialize it with data.

MS SQL Solution 4.1.2.b (table modification and data initialization)

```
-- Table modification:
ALTER TABLE [genres]
  ADD [g_books] INT NOT NULL DEFAULT 0;

-- Data initialization:
UPDATE [genres]
SET    [g_books] = [g_has_books]
FROM   [genres]
       JOIN (SELECT [g_id],
                    COUNT([b_id]) AS [g_has_books]
             FROM   [m2m_books_genres]
             GROUP  BY [g_id]) AS [prepared_data]
         ON [genres].[g_id] = [prepared_data].[g_id];
```

In MS SQL Server triggers are activated by cascade operations, so here it will be enough to create triggers only on the **`m2m_books_genres`** table.

The **`INSERT`** and **`DELETE`** triggers are quite simple: each of them counts the number of books added to or removed from a genre and changes the counter of books in the corresponding genre to the obtained value.

MS SQL Solution 4.1.2.b (triggers on the **`m2m_books_genres`** table)

```
-- Reaction to adding relationships between books and genres:
CREATE TRIGGER [g_has_books_on_m2m_b_g_ins]
ON [m2m_books_genres]
AFTER INSERT
AS
UPDATE [genres]
SET    [g_books] = [g_books] + [g_new_books]
FROM   [genres]
       JOIN (SELECT [g_id],
                    COUNT([b_id]) AS [g_new_books]
             FROM   [inserted]
             GROUP  BY [g_id]) AS [prepared_data]
         ON [genres].[g_id] = [prepared_data].[g_id];
GO

-- Reaction to the update of the relationship between books and genres:
CREATE TRIGGER [g_has_books_on_m2m_b_g_upd]
ON [m2m_books_genres]
AFTER UPDATE
AS
UPDATE [genres]
SET    [g_books] = [g_books] + [delta]
FROM   [genres]
       JOIN (SELECT [g_id],
                    SUM([delta]) AS [delta]
             FROM   (SELECT [g_id],
                            -COUNT([b_id]) AS [delta]
                     FROM   [deleted]
                     GROUP  BY [g_id]
                     UNION
                     SELECT [g_id],
                            COUNT([b_id]) AS [delta]
                     FROM   [inserted]
                     GROUP  BY [g_id]) AS [raw_deltas]
             GROUP  BY [g_id]) AS [ready_delta]
         ON [genres].[g_id] = [ready_delta].[g_id];
GO
```

MS SQL Solution 4.1.2.b (triggers on the `m2m_books_genres` table) (continued)

```
38  -- Reaction to deleting the relationship between books and genres:
39  CREATE TRIGGER [g_has_books_on_m2m_b_g_del]
40  ON [m2m_books_genres]
41  AFTER DELETE
42  AS
43  UPDATE [genres]
44  SET    [g_books] = [g_books] - [g_old_books]
45  FROM   [genres]
46         JOIN (SELECT [g_id],
47                      COUNT([b_id]) AS [g_old_books]
48               FROM   [deleted]
49               GROUP  BY [g_id]) AS [prepared_data]
50           ON [genres].[g_id] = [prepared_data].[g_id];
51  GO
```

The **`UPDATE`** trigger logic is a bit more complicated. To avoid two separate updates of the **`genres`** table, we first get a "summary table" of deleted and added relationships between books and genres in rows 26-34 of the query. This table in some hypothetical situation might look like this:

g_id	delta
3	4
5	1
1	-3
3	-6
6	2

The cells with a minus sign show the number of deleted relationships between books and genres, and the rest cells show the number of added relationships. Note the genre with identifier 3 which had some relationships removed and some added during one update operation. In line 25 of the query these negative and positive values are summed up, thus forming the final delta of the number of relationships between genres and books.

g_id	delta
3	-2
5	1
1	-3
6	2

Line 22 of the query uses this data to change the value of the genre-book relationship counter.

We can check the functionality of the resulting solution with the following queries (which are described in detail in the solution for MySQL).

MS SQL Solution 4.1.2.b (functionality check)

```
1   -- Adding two relationships to the "Science" genre (genre identifier is 4):
2   INSERT INTO [m2m_books_genres]
3               ([b_id],
4                [g_id])
5   VALUES      (1, 4),
6               (2, 4);
```

MS SQL Solution 4.1.2.b (functionality check) (continued)

```
7     -- Changing the value of books identifiers in the added relationships,
8     -- without changing the value of genre identifiers:
9     UPDATE [m2m_books_genres]
10    SET    [b_id] = 3
11    WHERE  [b_id] = 1
12       AND [g_id] = 4;
13
14    UPDATE [m2m_books_genres]
15    SET    [b_id] = 4
16    WHERE  [b_id] = 2
17       AND [g_id] = 4;
18
19    -- Changing the value of genres identifiers in the added relationships,
20    -- without changing the value of books identifiers:
21    UPDATE [m2m_books_genres]
22    SET    [g_id] = 5
23    WHERE  [b_id] = 3
24       AND [g_id] = 4;
25
26    UPDATE [m2m_books_genres]
27    SET    [g_id] = 5
28    WHERE  [b_id] = 4
29       AND [g_id] = 4;
30
31    -- Changing the values of genres identifiers and book identifiers
32    -- simultaneously in added relationships:
33    UPDATE [m2m_books_genres]
34    SET    [b_id] = 1,
35           [g_id] = 4
36    WHERE  [b_id] = 3
37       AND [g_id] = 5;
38
39    UPDATE [m2m_books_genres]
40    SET    [b_id] = 2,
41           [g_id] = 4
42    WHERE  [b_id] = 4
43       AND [g_id] = 5;
44
45    -- Deleting previously created relationships:
46    DELETE FROM [m2m_books_genres]
47    WHERE  [b_id] = 1
48       AND [g_id] = 4;
49
50    DELETE FROM [m2m_books_genres]
51    WHERE  [b_id] = 2
52       AND [g_id] = 4;
53
54    -- Deleting books with identifiers 1 and 2 (both belong to the "Poetry" and
55    -- "Classic" genres at the same time):
56    DELETE FROM [books]
57    WHERE [b_id] IN (1, 2);
```

So, the solution for MySQL is completed and tested.

Let's move on to the solution for Oracle. Since this DBMS does not support **`[inserted]`** and **`[deleted]`** pseudo-tables, we will rely on the logic of the MySQL solution. All relevant details of this solution have already been described above, so only the SQL code will be presented here.

Also note that since Oracle triggers are activated by cascade operations, this solution (unlike the MySQL solution) does not require creating a **`DELETE`** trigger on the **`books`** table.

Oracle Solution 4.1.2.b (table modification and data initialization)

```
1     -- Table modification:
2     ALTER TABLE "genres"
3       ADD ("g_books" NUMBER(10) DEFAULT 0 NOT NULL);
4
5     -- Data initialization:
6     UPDATE "genres" "outer"
7     SET    "g_books" =
8             NVL((SELECT COUNT("b_id") AS "g_has_books"
9             FROM   "m2m_books_genres"
10            WHERE "outer"."g_id" = "g_id"
11            GROUP  BY "g_id"), 0);
```

Oracle Solution 4.1.2.b (triggers on the `m2m_books_genres` table)

```
1     -- Reaction to adding the relationship between books and genres:
2     CREATE TRIGGER "g_has_bks_on_m2m_b_g_ins"
3     AFTER INSERT
4     ON "m2m_books_genres"
5      FOR EACH ROW
6       BEGIN
7        UPDATE "genres"
8        SET    "g_books" = "g_books" + 1
9        WHERE  "g_id" = :new."g_id";
10      END;
11
12    -- Reaction to the update of the relationship between books and genres:
13    CREATE TRIGGER "g_has_bks_on_m2m_b_g_upd"
14    AFTER UPDATE
15    ON "m2m_books_genres"
16     FOR EACH ROW
17      BEGIN
18       UPDATE "genres"
19       SET    "g_books" = "g_books" + 1
20       WHERE  "g_id" = :new."g_id";
21       UPDATE "genres"
22       SET    "g_books" = "g_books" - 1
23       WHERE  "g_id" = :old."g_id";
24      END;
25
26    -- Reaction to deleting the relationship between books and genres:
27    CREATE TRIGGER "g_has_bks_on_m2m_b_g_del"
28    AFTER DELETE
29    ON "m2m_books_genres"
30     FOR EACH ROW
31      BEGIN
32       UPDATE "genres"
33       SET    "g_books" = "g_books" - 1
34       WHERE  "g_id" = :old."g_id";
35      END;
```

We can check the functionality of the resulting solution with the following queries (which are described in detail in the solution for MySQL).

Oracle Solution 4.1.2.b (functionality check)

```
1     -- Adding two relationships to the "Science" genre (genre identifier is 4):
2     INSERT INTO "m2m_books_genres"
3                 ("b_id", "g_id")
4     VALUES      (1, 4);
5
6     INSERT INTO "m2m_books_genres"
7                 ("b_id", "g_id")
8     VALUES      (2, 4);
```

Oracle | Solution 4.1.2.b (functionality check)

```
9     -- Changing the value of books identifiers in the added relationships,
10    -- without changing the value of genre identifiers:
11    UPDATE "m2m_books_genres"
12    SET    "b_id" = 3
13    WHERE  "b_id" = 1
14       AND "g_id" = 4;
15
16    UPDATE "m2m_books_genres"
17    SET    "b_id" = 4
18    WHERE  "b_id" = 2
19       AND "g_id" = 4;
20
21    -- Changing the value of genre identifiers in the added relationships,
22    -- without changing the value of book identifiers:
23    UPDATE "m2m_books_genres"
24    SET    "g_id" = 5
25    WHERE  "b_id" = 3
26       AND "g_id" = 4;
27
28    UPDATE "m2m_books_genres"
29    SET    "g_id" = 5
30    WHERE  "b_id" = 4
31       AND "g_id" = 4;
32
33    -- Changing the values of genres identifiers and book identifiers
34    -- simultaneously in added relationships:
35    UPDATE "m2m_books_genres"
36    SET    "b_id" = 1,
37           "g_id" = 4
38    WHERE  "b_id" = 3
39       AND "g_id" = 5;
40
41    UPDATE "m2m_books_genres"
42    SET    "b_id" = 2,
43           "g_id" = 4
44    WHERE  "b_id" = 4
45       AND "g_id" = 5;
46
47    -- Deleting previously created relationships:
48    DELETE FROM "m2m_books_genres"
49    WHERE  "b_id" = 1
50       AND "g_id" = 4;
51
52    DELETE FROM "m2m_books_genres"
53    WHERE  "b_id" = 2
54       AND "g_id" = 4;
55
56    -- Deleting books with identifiers 1 and 2 (both of which belong to the
57    -- "Poetry" and "Classics" genres at the same time):
58    DELETE FROM "books"
59    WHERE "b_id" IN (1, 2);
```

This completes the solution of this problem.

Task 4.1.2.TSK.A: refine the triggers from the solutions{306}, {319} of problems 4.1.2.a{306} and 4.1.2.b{306} so that during any manipulations with data the values of `s_books` (in the `subscribers` table) and `g_books` (in the `genres` table) fields could not be negative.

Task 4.1.2.TSK.B: modify the "Library" database schema so that the `subscribers` table stores information about how many times a reader has borrowed books from the library (this counter should be incremented each time a reader takes a book; reducing this counter value is not necessary).

Task 4.1.2.TSK.C: optimize the `UPDATE` trigger code from the solution{306} of problem 4.1.2.a{306} for MS SQL Server, so that there will be one operation to update the `subscribers` table (instead of two separate operations, as it is implemented now).

4.2. Using Triggers to Control Data Operations

4.2.1. Example 33: Using Triggers to Control Data Modification

Problem 4.2.1.a{329}: create a trigger that does not allow one to add information about a subscriptions to the database if at least one of the conditions is met:

- the start date of a subscription is in the future;
- the finish date of a subscription in the past (only for data insertion);
- the return date is less than the start date.

Problem 4.2.1.b{342}: create a trigger that prohibits giving a book to a subscriber who has ten or more books in their hands.

Problem 4.2.1.c{349}: create a trigger that prohibits changing the value of the `sb_is_active` field of the `subscriptions` table from the `N` value to the `Y` value.

Expected result 4.2.1.a.

If an attempt is made to make changes to the database that contradict the problem conditions, the operation (transaction) must be cancelled. An error message must also be displayed, clearly explaining the problem, e.g.:

- “Date 2038.01.12 for subscription 145 activation is in the future”.
- “Date 1983.01.12 for subscription 155 deactivation is in the past”.
- “Date 2000.01.12 for subscription 165 deactivation is less than date for its activation (2015.01.12)”.

Expected result 4.2.1.b.

If an attempt is made to make changes to the database that contradict the problem condition, the operation (transaction) must be cancelled. An error message must also be displayed, clearly explaining the problem, e.g.: “Subscriber Ivanov I.I. (id = 1) already has 23 books out of 10 allowed.”

Expected result 4.2.1.c.

If an attempt is made to make changes to the database that contradict the problem condition, the operation (transaction) must be cancelled. An error message must also be displayed, clearly explaining the problem, e.g.: “It is prohibited to activate previously deactivated subscriptions (rule violated for subscriptions 34, 89, 12).”

Solution 4.2.1.a{329}.

To solve this problem we need only `INSERT` and `UPDATE` triggers, because none of the controlled conditions can be violated in the process of data deletion.

The **INSERT** trigger code for MySQL looks like this.

MySQL Solution 4.2.1.a (trigger on the **subscriptions** table)

```
1    DELIMITER $$
2
3    CREATE TRIGGER `subscriptions_control_ins`
4    AFTER INSERT
5    ON `subscriptions`
6     FOR EACH ROW
7      BEGIN
8
9       -- Blocking the subscriptions with the start date in the future.
10      IF NEW.`sb_start` > CURDATE()
11       THEN
12        SET @msg = CONCAT('Date ', NEW.`sb_start`, ' for subscription ',
13                    NEW.`sb_id`, ' activation is in the future.');
14        SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
15       END IF;
16
17      -- Blocking the subscriptions with the finish date in the past.
18      IF NEW.`sb_finish` < CURDATE()
19       THEN
20        SET @msg = CONCAT('Date ', NEW.`sb_finish`, ' for subscription ',
21                    NEW.`sb_id`, ' deactivation is in the past.');
22        SIGNAL SQLSTATE '45002' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1002;
23       END IF;
24
25      -- Blocking the subscriptions with the finish date less than start date.
26      IF NEW.`sb_finish` < NEW.`sb_start`
27       THEN
28        SET @msg = CONCAT('Date ', NEW.`sb_finish`, ' for subscription ',
29                    NEW.`sb_id`,
30                    ' deactivation is less than the date for its activation (',
31                    NEW.`sb_start`, ').');
32        SIGNAL SQLSTATE '45003' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1003;
33       END IF;
34
35     END;
36   $$
37
38   DELIMITER ;
```

From the functionality point of view, the **BEFORE** trigger could also be used in this case, but in the case of the **AFTER** trigger the error message is more informative, because it already contains the correct value of the auto-incrementable primary key (in the **BEFORE** trigger this value is not defined, and therefore it turns into 0 in the error message).

The code in lines 25-34 is not needed at the moment, the two preceding checks will not allow the situation checked by this code to occur. But if these checks are modified or removed in the future, the code in lines 25-34 will work.

Since the MySQL trigger cannot explicitly cancel a transaction, we rise an exception (lines 14, 22, 33) that cancels the transaction that activated the trigger.

The **UPDATE** trigger code will be even a bit simpler since it does not need to check if the finish date is in the past.

The code in lines 25-34 (previously marked as “useless” for **INSERT** trigger) will work here, because by the condition of the problem when performing the update operation, it is allowed to set in the **sb_finish** field the date from the past, which allows to violate the third condition of the problem.

MySQL Solution 4.2.1.a (triggers on the `subscriptions` table)

```
DELIMITER $$

CREATE TRIGGER `subscriptions_control_upd`
AFTER UPDATE
ON `subscriptions`
 FOR EACH ROW
  BEGIN

    -- Blocking the subscriptions with the start date in the future.
    IF NEW.`sb_start` > CURDATE()
     THEN
      SET @msg = CONCAT('Date ', NEW.`sb_start`, ' for subscription ',
                 NEW.`sb_id`, ' activation is in the future.');
      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
     END IF;

    -- Blocking the subscriptions with the finish date less than start date.
    IF NEW.`sb_finish` < NEW.`sb_start`
     THEN
      SET @msg = CONCAT('Date ', NEW.`sb_finish`, ' for subscription ',
                 NEW.`sb_id`,
                 ' deactivation is less than the date for its activation (',
                 NEW.`sb_start`, ').');
      SIGNAL SQLSTATE '45003' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1003;
     END IF;

  END;
$$

DELIMITER ;
```

Let's check the functionality of the obtained solution. We will run the following queries and watch how the DBMS responds.

Let's try to add a subscriptions with the start date in the future:

MySQL Solution 4.2.1.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (500,
             1,
             1,
             '2050-01-12',
             '2050-02-12',
             'N')
```

The DBMS will block this operation and return the following error message:

```
Error Code: 1001. Date 2050-01-12 for subscription 500 activation is in the future.
```

Let's try to add a subscription with the start date in the future, without specifying the value of the primary key:

MySQL Solution 4.2.1.a (functionality check)

```
INSERT INTO `subscriptions`
            (`sb_id`,
             `sb_subscriber`,
             `sb_book`,
             `sb_start`,
             `sb_finish`,
             `sb_is_active`)
VALUES      (NULL,
             3,
             3,
             '2050-01-12',
             '2050-02-12',
             'N');
```

The DBMS will block this operation and return the following error message:

```
Error Code: 1001. Date 2050-01-12 for subscription 501 activation is in the future.
```

Let's try to add a subscription with the finish date in the past:

MySQL Solution 4.2.1.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (502,
             1,
             1,
             '2000-01-12',
             '2000-02-12',
             'N')
```

The DBMS will block this operation and return the following error message:

```
Error Code: 1002. Date 2000-02-12 for subscription 502 deactivation is in the past.
```

Let's try to add a subscription without violating any of the conditions of this problem:

MySQL Solution 4.2.1.a (functionality check)

```
INSERT INTO `subscriptions`
VALUES      (503,
             1,
             1,
             '2000-01-12',
             '2050-02-12',
             'N')
```

The DBMS will allow the insertion.
Let's try to update the added subscription so that its start date is in the future:

MySQL Solution 4.2.1.a (functionality check)

```
UPDATE `subscriptions`
SET    `sb_start` = '2050-01-01'
WHERE  `sb_id` = 503
```

The DBMS will block this operation and return the following error message:

```
Error Code: 1001. Date 2050-01-01 for subscription 503 activation is in the future.
```

Let's try to update the added subscription so that its start date is later than the finish date:

MySQL Solution 4.2.1.a (functionality check)

```
UPDATE `subscriptions`
SET    `sb_start` = '2030-01-01',
       `sb_finish` = '2025-01-01'
WHERE  `sb_id` = 503
```

The DBMS will block this operation and return the following error message:

```
Error Code: 1003. Date 2025-01-01 for subscription 503 deactivation is less than the
date for its activation (2030-01-01).
```

Let's try to update the added subscription so that its finish date is in the past (this is allowed for the update operation):

MySQL Solution 4.2.1.a (functionality check)

```
UPDATE `subscriptions`
SET    `sb_start` = '2005-01-01',
       `sb_finish` = '2006-01-01'
WHERE  `sb_id` = 503
```

The DBMS will allow the update.

Let's try to update the subscription without violating any of the conditions of the problem:

MySQL — Solution 4.2.1.a (functionality check)

```
1    UPDATE `subscriptions`
2    SET    `sb_start` = '2005-01-01',
3           `sb_finish` = '2010-01-01'
4    WHERE  `sb_id` = 503
```

The DBMS will allow the update.

So, the solution for MySQL is complete and tested. Let's move on to the solution for MS SQL Server.

In MS SQL Server, we only have statement-level triggers, so their internal logic will be different than in MySQL. Also, the error messages returned by triggers will look a bit different, because they will need to contain information about all the subscriptions: in one implementation, only the "bad" ones, and in the second, both the "bad" and "good" ones.

The first implementation of the trigger completely blocks the operation if at least one of the records violates the conditions of the problem. Information about such records is accumulated in the **`@bad_records`** string variable (for an explanation of the **`STUFF`** function logic, see the solution{75} of problem 2.2.2.a{74}).

Then the length of the produced string is checked: if it is **not** equal to zero, it means that "bad" records are detected, and the trigger must send an error message to the client (lines 23, 48, 67) and cancel the operation ("rollback the transaction") (lines 24, 49, 69).

On lines 28-32 we get information about the number of records in the **`[inserted]`** and **`[deleted]`** pseudo-tables, so that we can then determine on lines 42-43 whether an insertion operation was performed (its sign: no records in **`[deleted]`**, some records in **`[inserted]`**).

MS SQL — Solution 4.2.1.a (trigger on the **`subscriptions`** table, option one)

```
1    -- Option with complete operation block.
2    CREATE TRIGGER [subscriptions_control]
3    ON [subscriptions]
4    AFTER INSERT, UPDATE
5    AS
6     -- Variables for storing the list of "bad records" and the error message.
7     DECLARE @bad_records NVARCHAR(max);
8     DECLARE @msg NVARCHAR(max);
9
10    -- Blocking the subscriptions with the start date in the future.
11    SELECT @bad_records = STUFF((SELECT ', ' + CAST([sb_id] AS NVARCHAR) +
12                           ' (' + CAST([sb_start] AS NVARCHAR) + ')'
13                    FROM   [inserted]
14                    WHERE [sb_start] > CONVERT(date, GETDATE())
15                    ORDER  BY [sb_id]
16                    FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
17                    1, 2, '');
18    IF LEN(@bad_records) > 0
19     BEGIN
20      SET @msg =
21      CONCAT('The following subscriptions'' activation dates are
22      in the future: ', @bad_records);
23      RAISERROR (@msg, 16, 1);
24      ROLLBACK TRANSACTION;
25      RETURN
26     END;
```

MS SQL — Solution 4.2.1.a (trigger on the `subscriptions` table, option one) (continued)

```
27    -- Blocking the subscriptions with the finish date in the past.
28    DECLARE @deleted_records INT;
29    DECLARE @inserted_records INT;
30
31    SELECT @deleted_records = COUNT(*) FROM [deleted];
32    SELECT @inserted_records = COUNT(*) FROM [inserted];
33
34    SELECT @bad_records = STUFF((SELECT ', ' + CAST([sb_id] AS NVARCHAR) +
35           ' (' + CAST([sb_start] AS NVARCHAR) + ')'
36                  FROM  [inserted]
37                  WHERE [sb_finish] < CONVERT(date, GETDATE())
38                  ORDER  BY [sb_id]
39                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
40                  1, 2, '');
41    IF ((LEN(@bad_records) > 0) AND
42        (@deleted_records = 0) AND
43        (@inserted_records > 0))
44     BEGIN
45      SET @msg =
46      CONCAT('The following subscriptions'' deactivation dates are
47      in the past: ', @bad_records);
48      RAISERROR (@msg, 16, 1);
49      ROLLBACK TRANSACTION;
50      RETURN
51     END;
52
53    -- Blocking the subscriptions with the finish date less than start date.
54    SELECT @bad_records = STUFF((SELECT ', ' + CAST([sb_id] AS NVARCHAR) +
55           ' (act: ' + CAST([sb_start] AS NVARCHAR) + ', deact: ' +
56          CAST([sb_finish] AS NVARCHAR) + ')'
57                  FROM  [inserted]
58                  WHERE [sb_finish] < [sb_start]
59                  ORDER  BY [sb_id]
60                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
61                  1, 2, '');
62    IF LEN(@bad_records) > 0
63     BEGIN
64      SET @msg =
65      CONCAT('The following subscriptions'' deactivation dates are less
66              than activation dates: ', @bad_records);
67      RAISERROR (@msg, 16, 1);
68      ROLLBACK TRANSACTION;
69      RETURN
70     END;
71   GO
```

The second option of implementation of the trigger blocks only operations with "bad" records and performs operations with "good" records. It also displays a message with information about "good" records and an error message with information about "bad" records.

To achieve this effect, we will use an `INSTEAD OF` trigger, which is activated **instead** of the corresponding data operation. If no action is performed in the body of such a trigger, the original data operation will not be executed by definition, and there is even no need to "rollback the transaction". For the operation to be executed, the trigger body will need to execute a corresponding query to modify the data in the table

Unfortunately, there are foreign keys with cascade update in the `subscriptions` table, so MS SQL Server will not allow to create `INSTEAD OF UPDATE` trigger, but it is possible to demonstrate the general logic of the solution on the insertion operation alone.

If we were faced with the urgent task of applying this solution to the update operation as well, we could change the properties of the foreign keys by removing the cascade update operation there and implementing it in a separate trigger.

Note how this option of the solution implements the sequence of actions: since we do not cancel the operation or exit the trigger body (as implemented in lines 24-25, 49-50, 69-70 of the first option of the solution), the trigger will complete its body, forcing us to simultaneously consider all three problem conditions when deciding whether we have a “good” or a “bad” record.

First, we get three lists of “bad” records for each of the problem conditions (lines 14-40). In the **`INSTEAD OF`** trigger, we do not know the values of the auto-incrementable primary key (**`sb_id IDENTITY`** field), so we collect only the date values themselves. However, we need these values for the real data insertion (lines 82-106): in the solution{260} of problem 3.2.1.a{259} the logic of their obtaining and use is explained in detail.

MS SQL — Solution 4.2.1.a (trigger on the **`subscriptions`** table, option two)

```
1    -- Option with partial operation block.
2    CREATE TRIGGER [subscriptions_control]
3    ON [subscriptions]
4    INSTEAD OF INSERT
5    AS
6     -- Variables for storing messages and record lists.
7     DECLARE @bad_records_act_future NVARCHAR(max);
8     DECLARE @bad_records_deact_past NVARCHAR(max);
9     DECLARE @bad_records_act_greater_than_deact NVARCHAR(max);
10
11    DECLARE @good_records NVARCHAR(max);
12    DECLARE @msg NVARCHAR(max);
13
14    -- Blocking the subscriptions with the start date in the future.
15    SELECT @bad_records_act_future =
16                  STUFF((SELECT ', ' + CAST([sb_start] AS NVARCHAR)
17                  FROM  [inserted]
18                  WHERE [sb_start] > CONVERT(date, GETDATE())
19                  ORDER  BY [sb_start]
20                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
21                  1, 2, '');
22
23    -- Blocking the subscriptions with the finish date in the past.
24    SELECT @bad_records_deact_past =
25                  STUFF((SELECT ', ' + CAST([sb_finish] AS NVARCHAR)
26                  FROM  [inserted]
27                  WHERE [sb_finish] < CONVERT(date, GETDATE())
28                  ORDER  BY [sb_finish]
29                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
30                  1, 2, '');
31
32    -- Blocking the subscriptions with the finish date less than start date.
33    SELECT @bad_records_act_greater_than_deact =
34                  STUFF((SELECT ', (act: ' + CAST([sb_start] AS NVARCHAR) +
35                         ', deact: ' + CAST([sb_finish] AS NVARCHAR) + ')'
36                  FROM  [inserted]
37                  WHERE [sb_finish] < [sb_start]
38                  ORDER  BY [sb_start], [sb_finish]
39                         FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
40                  1, 2, '');
```

MS SQL — Solution 4.2.1.a (trigger on the `subscriptions` table, option two) (continued)

```
41     IF ((LEN(@bad_records_act_future) > 0) OR
42         (LEN(@bad_records_deact_past) > 0) OR
43         (LEN(@bad_records_act_greater_than_deact) > 0))
44      BEGIN
45       SET @msg = 'Some records were NOT inserted!';
46       IF (LEN(@bad_records_act_future) > 0)
47        BEGIN
48         SET @msg = CONCAT(@msg, CHAR(13), CHAR(10),
49         'The following activation dates are in the future: ',
50         @bad_records_act_future);
51        END;
52       IF (LEN(@bad_records_deact_past) > 0)
53        BEGIN
54         SET @msg = CONCAT(@msg, CHAR(13), CHAR(10),
55         'The following deactivation dates are in the past: ',
56         @bad_records_deact_past);
57        END;
58       IF (LEN(@bad_records_act_greater_than_deact) > 0)
59        BEGIN
60         SET @msg = CONCAT(@msg, CHAR(13), CHAR(10),
61         'The following deactivation dates are less than activation dates: ',
62         @bad_records_act_greater_than_deact);
63        END;
64       RAISERROR (@msg, 16, 1);
65      END;
66
67     SELECT @good_records = STUFF((SELECT ', ' +
68                   CAST([sb_start] AS NVARCHAR) + '/' +
69                   CAST([sb_finish] AS NVARCHAR)
70                   FROM  [inserted]
71                   WHERE (([sb_start] <= CONVERT(date, GETDATE())) AND
72                          ([sb_finish] >= CONVERT(date, GETDATE())) AND
73                          ([sb_finish] >= [sb_start]))
74                   ORDER  BY [sb_start], [sb_finish]
75                   FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
76                   1, 2, '');
77
78    IF LEN(@good_records) > 0
79     BEGIN
80      SET IDENTITY_INSERT [subscriptions] ON;
81      INSERT INTO [subscriptions]
82                  ([sb_id],
83                   [sb_subscriber],
84                   [sb_book],
85                   [sb_start],
86                   [sb_finish],
87                   [sb_is_active])
88      SELECT ( CASE
89                   WHEN [sb_id] IS NULL
90                     OR [sb_id] = 0 THEN IDENT_CURRENT('subscriptions')
91                                         + IDENT_INCR('subscriptions')
92                                         + ROW_NUMBER() OVER (ORDER BY
93                                                              (SELECT 1))
94                                         - 1
95                                    ELSE [sb_id]
96                END ) AS [sb_id],
97               [sb_subscriber],
98               [sb_book],
99               [sb_start],
100              [sb_finish],
101              [sb_is_active]
102     FROM    [inserted]
103     WHERE (([sb_start] <= CONVERT(date, GETDATE())) AND
104                   ([sb_finish] >= CONVERT(date, GETDATE())) AND
105                   ([sb_finish] >= [sb_start]));
```

MS SQL — Solution 4.2.1.a (trigger on the `subscriptions` table, option two) (ending)

```
106     SET IDENTITY_INSERT [subscriptions] OFF;
107     SET @msg =
108     CONCAT('Subscriptions with the following activation/deactivation
109     dates were inserted successfully: ', @good_records);
110     PRINT @msg;
111   END;
112 GO
```

The code on lines 42-66 checks if any bad entries were found, and if so, generates an error message that takes all three conditions into account. Since after sending the error message (line 65) we do not rollback the transaction or exit the body of the trigger, the execution continues, and we are able to insert the "good" records into the table.

Lines 68-77 form a list of such "good" records, whose data does not violate any of the conditions of the problem. If such records were found, on lines 79-112 we insert them into the table and display a message (just a message, **not** an error message) with a list of their dates. It is easy to guess that this insertion operation does not re-activate the `INSTEAD OF` trigger (otherwise we would get infinite recursion).

Important! This (second) option is shown for educational purposes to demonstrate the capabilities of MS SQL Server triggers. In real-world applications, such "partial" data processing (when some records are successfully inserted into a table, and some are not) may lead to hard-to-detect defects and other hardly predictable consequences.

So, both solutions are ready, now we have to check their functionality. Let's execute queries and keep track of received messages.

Let's insert data with explicitly specified values of the primary key and part of the records that satisfy the conditions of the problem, and part that do not satisfy:

MS SQL — Solution 4.2.1.a (functionality check)

```
1   SET IDENTITY_INSERT [subscriptions] ON;
2   INSERT INTO [subscriptions]
3               ([sb_id],
4                [sb_subscriber],
5                [sb_book],
6                [sb_start],
7                [sb_finish],
8                [sb_is_active])
9   VALUES      (500,
10               3,
11               3,
12               '2020-01-12',
13               '2020-02-12',
14               'N'),
15              (600,
16               3,
17               4,
18               '2021-01-12',
19               '2021-02-12',
20               'N'),
21              (700,
22               4,
23               4,
24               '2001-01-12',
25               '2021-02-12',
26               'N');
27  SET IDENTITY_INSERT [subscriptions] OFF;
```

Received messages:

- In the first option of the solution, we get the error message:

```
The following subscriptions' activation dates are in the future: 500
(2020-01-12), 600 (2021-01-12)
```

- In the second option of the solution, we get:
  - The error message:

```
Some records were NOT inserted! The following activation dates are
in the future: 2020-01-12, 2021-01-12
```

  - The information message:

```
Subscriptions with the following activation/deactivation dates were
inserted successfully: 2001-01-12/2021-02-12
```

Let's insert the data without specifying the values of the primary key and with part of the records satisfying the conditions of the problem and part not satisfying:

MS SQL Solution 4.2.1.a (functionality check)

```
INSERT INTO [subscriptions]
            ([sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (3,
             3,
             '2020-01-12',
             '2020-02-12',
             'N'),
            (3,
             4,
             '2021-01-12',
             '2021-02-12',
             'N'),
            (4,
             4,
             '2001-01-12',
             '2021-02-12',
             'N')
```

Received messages:

- In the first option of the solution, we get the error message:

```
The following subscriptions' deactivation dates are in the past: 704
(2001-01-12), 705 (2002-01-12)
```

- In the second option of the solution, we get:
  - The error message:

```
Some records were NOT inserted! The following deactivation dates
are in the past: 2001-02-12, 2002-02-12
```

  - The information message:

```
Subscriptions with the following activation/deactivation dates were
inserted successfully: 2001-01-12/2021-02-12
```

Let's insert the data that satisfy all the conditions of the problem:

MS SQL Solution 4.2.1.a (functionality check)

```
INSERT INTO [subscriptions]
            ([sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
VALUES      (4,
             4,
             '2001-01-12',
             '2021-02-12',
             'N')
```

Received messages:

- In the first option of the solution: no messages.
- In the second option of the solution, we get:
  - The error message: none.
  - The information message:

```
Subscriptions with the following activation/deactivation dates were
inserted successfully: 2001-01-12/2021-02-12
```

Let's update the data in violation of one of the conditions of the problem:

MS SQL Solution 4.2.1.a (functionality check)

```
UPDATE [subscriptions]
SET    [sb_finish] = '2005-01-01'
WHERE  [sb_start] > '2011-01-01'
```

The trigger in the second option of the solution does not respond to the update operation, and the trigger in the first option of the solution will give the following error message:

```
The following subscriptions' deactivation dates are less than activation dates:
2 (act: 2011-01-12, deact: 2005-01-01), 3 (act: 2012-05-17, deact: 2005-01-01),
42 (act: 2012-06-11, deact: 2005-01-01), 57 (act: 2012-06-11, deact: 2005-01-
01), 61 (act: 2014-08-03, deact: 2005-01-01), 62 (act: 2014-08-03, deact: 2005-
01-01), 86 (act: 2014-08-03, deact: 2005-01-01), 91 (act: 2015-10-07, deact:
2005-01-01), 95 (act: 2015-10-07, deact: 2005-01-01), 99 (act: 2015-10-08, de-
act: 2005-01-01), 100 (act: 2011-01-12, deact: 2005-01-01)
```

Let's update the data in compliance with all the conditions of the problem:

MS SQL Solution 4.2.1.a (functionality check)

```
UPDATE [subscriptions]
SET    [sb_finish] = '2002-01-01'
WHERE  [sb_start] = '2001-01-12';
```

The trigger in the second option does not respond to the update operation, and no messages will come from the trigger in the first option.

So, the solution of this problem for MS SQL Server is obtained and tested. Let's move on to the solution for Oracle.

Since Oracle does not support `[deleted]` and `[inserted]` pseudo-tables, we implement the same logic as in the MySQL solution, using row-level triggers.

So, the only difference between the Oracle and the MySQL solutions will be in the way the operation is cancelled (with the simultaneous output of an error message): in Oracle, it is convenient to use the `RAISE_APPLICATION_ERROR` function for such tasks.

Apart from that, the solutions for Oracle and MySQL are completely identical.

Oracle | Solution 4.2.1.a (triggers on the `subscriptions` table)

```
1     -- Reaction to adding a subscription.
2     CREATE TRIGGER "subscriptions_control_ins"
3     AFTER INSERT
4     ON "subscriptions"
5      FOR EACH ROW
6       BEGIN
7
8        -- Blocking the subscriptions with the start date in the future.
9        IF :new."sb_start" > TRUNC(SYSDATE)
10        THEN
11         RAISE_APPLICATION_ERROR(-20001, 'Date ' || :new."sb_start" ||
12                                 ' for subscription ' || :new."sb_id" ||
13                                 ' activation is in the future.');
14       END IF;
15
16       -- Blocking the subscriptions with the finish date in the past.
17       IF :new."sb_finish" < TRUNC(SYSDATE)
18        THEN
19         RAISE_APPLICATION_ERROR(-20002, 'Date ' || :new."sb_finish" ||
20                                 ' for subscription ' || :new."sb_id" ||
21                                 ' deactivation is in the past.');
22       END IF;
23
24       -- Blocking the subscriptions with the finish date less than start date.
25       IF :new."sb_finish" < :new."sb_start"
26        THEN
27         RAISE_APPLICATION_ERROR(-20003, 'Date ' || :new."sb_finish" ||
28                                 ' for subscription ' || :new."sb_id" ||
29                                 ' deactivation is less than the date
30                                   for its activation (' ||
31                                 :new."sb_start" || ').');
32       END IF;
33     END;
34
35    -- Reaction to the update of a subscription.
36    CREATE TRIGGER "subscriptions_control_upd"
37    AFTER UPDATE
38    ON "subscriptions"
39     FOR EACH ROW
40      BEGIN
41
42       -- Blocking the subscriptions with the start date in the future.
43       IF :new."sb_start" > TRUNC(SYSDATE)
44        THEN
45         RAISE_APPLICATION_ERROR(-20001, 'Date ' || :new."sb_start" ||
46                                 ' for subscription ' || :new."sb_id" ||
47                                 ' activation is in the future.');
48       END IF;
49
50       -- Blocking the subscriptions with the finish date less than start date.
51       IF :new."sb_finish" < :new."sb_start"
52        THEN
53         RAISE_APPLICATION_ERROR(-20003, 'Date ' || :new."sb_finish" ||
54                                 ' for subscription ' || :new."sb_id" ||
55                                 ' deactivation is less than the date
56                                   for its activation (' ||
57                                 :new."sb_start" || ').');
58       END IF;
59
60     END;
```

We can check the functionality of the obtained solution with the following queries (their logic and expected response of triggers are described in the MySQL solution).

Oracle Solution 4.2.1.a (functionality check)

```
-- Deactivating the trigger that forms the auto-incrementable PK value:
ALTER TRIGGER "TRG_subscriptions_sb_id" DISABLE;

-- Adding a subscription with the start date in the future:
INSERT INTO "subscriptions"
VALUES      (500,
             1,
             1,
             TO_DATE('2020-01-12', 'YYYY-MM-DD'),
             TO_DATE('2020-02-12', 'YYYY-MM-DD'),
             'N');

-- Reactivating the trigger that forms the auto-incrementable PK value:
ALTER TRIGGER "TRG_subscriptions_sb_id" ENABLE;

-- Adding a subscription with the start date in the future
-- (without specifying the value of the primary key):
INSERT INTO "subscriptions"
            ("sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
VALUES      (3,
             3,
             TO_DATE('2020-01-12', 'YYYY-MM-DD'),
             TO_DATE('2020-02-12', 'YYYY-MM-DD'),
             'N');

-- Adding a subscription with the finish date in the past:
INSERT INTO "subscriptions"
            ("sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
VALUES      (1,
             1,
             TO_DATE('2000-01-12', 'YYYY-MM-DD'),
             TO_DATE('2000-02-12', 'YYYY-MM-DD'),
             'N');

-- Adding a subscription without violating the conditions of the problem:
INSERT INTO "subscriptions"
            ("sb_subscriber",
             "sb_book",
             "sb_start",
             "sb_finish",
             "sb_is_active")
VALUES      (1,
             1,
             TO_DATE('2000-01-12', 'YYYY-MM-DD'),
             TO_DATE('2020-02-12', 'YYYY-MM-DD'),
             'N');

-- Updating the added subscription so that its start date
-- is in the future:
UPDATE "subscriptions"
SET    "sb_start" = TO_DATE('2020-01-01', 'YYYY-MM-DD')
WHERE  "sb_id" = 104;
```

Oracle Solution 4.2.1.a (functionality check) (continued)

```
61   -- Updating the added subscription so that its start date
62   -- is later than the finish date:
63   UPDATE "subscriptions"
64   SET    "sb_start" = TO_DATE('2010-01-01', 'YYYY-MM-DD'),
65          "sb_finish" = TO_DATE('2005-01-01', 'YYYY-MM-DD')
66   WHERE  "sb_id" = 104;
67
68   -- Updating the added subscription so that its
69   -- finish date is in the past
70   -- (this is allowed for the update operation):
71   UPDATE "subscriptions"
72   SET    "sb_start" = TO_DATE('2005-01-01', 'YYYY-MM-DD'),
73          "sb_finish" = TO_DATE('2006-01-01', 'YYYY-MM-DD')
74   WHERE  "sb_id" = 104;
75
76   -- Updating the added subscription without violating the conditions
77   -- of the problem:
78   UPDATE "subscriptions"
79   SET    "sb_start" = TO_DATE('2005-01-01', 'YYYY-MM-DD'),
80          "sb_finish" = TO_DATE('2010-01-01', 'YYYY-MM-DD')
81   WHERE  "sb_id" = 104;
```

This completes the solution of this problem.

Solution 4.2.1.b{329}.

With the example of this (rather simple) problem we will demonstrate a typical wrong solution, which is often the first that comes to mind. It goes down to checking whether there are subscribers for which the problem condition is violated (by analyzing all subscribers' data) in **`AFTER`** trigger and, if so, rolling back the transaction. On quite a large database, this solution can lead to a very noticeable performance drop.

The correct solution is to make the **`BEFORE`** trigger check only for that subscriber, for which the operation of inserting or updating records in the **`subscriptions`** table is performed now.

So, for all three DBMS, let's present the wrong and the right solution and compare their speed on the "Big Library (for Experiments)" database.

Let's create **`INSERT`** and **`UPDATE`** triggers with completely identical code in the wrong solution for MySQL, where we will form a list of readers for which the problem condition (not allowing more than ten books) was violated.

While being extremely non-optimal in terms of performance, this solution does have one advantage: it will also react to all violations of the problem condition, which were committed before the creation of the trigger. However, if we are not satisfied with this behavior, this advantage turns into a disadvantage, and the presented solution becomes even worse than we thought.

MySQL Solution 4.2.1.b (**wrong** solution)

```
DELIMITER $$

CREATE TRIGGER `sbs_cntrl_10_books_ins_WRONG`
AFTER INSERT
ON `subscriptions`
 FOR EACH ROW
  BEGIN

    SET @msg = IFNULL((SELECT GROUP_CONCAT(
                              CONCAT('(id=', `s_id`, ', ', `s_name`,
                              ', books=', `s_books`, ')') SEPARATOR ', ')
                              AS `list`
                       FROM   (SELECT `s_id`,
                                      `s_name`,
                                      COUNT(`sb_book`) AS `s_books`
                                FROM   `subscribers`
                                      JOIN `subscriptions`
                                      ON `s_id` = `sb_subscriber`
                                WHERE  `sb_is_active` = 'Y'
                                GROUP  BY `sb_subscriber`
                                HAVING `s_books` > 10) AS `prepared_data`),
                  '');

    IF (LENGTH(@msg) > 0)
     THEN
      SET @msg = CONCAT('The following readers have more books
                         than allowed (10 allowed): ', @msg);
      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
     END IF;

  END;
$$

CREATE TRIGGER `sbs_cntrl_10_books_upd_WRONG`
AFTER UPDATE
ON `subscriptions`
 FOR EACH ROW
  BEGIN

    SET @msg = IFNULL((SELECT GROUP_CONCAT(
                              CONCAT('(id=', `s_id`, ', ', `s_name`,
                              ', books=', `s_books`, ')') SEPARATOR ', ')
                              AS `list`
                       FROM   (SELECT `s_id`,
                                      `s_name`,
                                      COUNT(`sb_book`) AS `s_books`
                                FROM   `subscribers`
                                      JOIN `subscriptions`
                                      ON `s_id` = `sb_subscriber`
                                WHERE  `sb_is_active` = 'Y'
                                GROUP  BY `sb_subscriber`
                                HAVING `s_books` > 10) AS `prepared_data`),
                  '');

    IF (LENGTH(@msg) > 0)
     THEN
      SET @msg = CONCAT('The following readers have more books
                         than allowed (10 allowed): ', @msg);
      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
     END IF;

  END;
$$

DELIMITER ;
```

MySQL | Solution 4.2.1.b (triggers on the `subscriptions` table)

```
DELIMITER $$

CREATE TRIGGER `sbs_cntrl_10_books_ins_OK`
BEFORE INSERT
ON `subscriptions`
 FOR EACH ROW
  BEGIN

    SET @msg = IFNULL((SELECT CONCAT('Subscriber ', `s_name`,
                                     ' (id=', `sb_subscriber`, ') already has ',
                                     `sb_books`, ' books out of 10 allowed.')
                                    AS `message`
                       FROM   (SELECT `sb_subscriber`,
                                      COUNT(`sb_book`) AS `sb_books`
                               FROM   `subscriptions`
                               WHERE  `sb_is_active` = 'Y'
                               AND `sb_subscriber` = NEW.`sb_subscriber`
                               GROUP  BY `sb_subscriber`
                               HAVING `sb_books` >= 10) AS `prepared_data`
                       JOIN `subscribers`
                       ON `sb_subscriber` = `s_id`),
                  '');

    IF (LENGTH(@msg) > 0)
     THEN
      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
     END IF;

  END;
$$

CREATE TRIGGER `sbs_cntrl_10_books_upd_OK`
BEFORE UPDATE
ON `subscriptions`
 FOR EACH ROW
  BEGIN

    SET @msg = IFNULL((SELECT CONCAT('Subscriber ', `s_name`,
                                     ' (id=', `sb_subscriber`, ') already has ',
                                     `sb_books`, ' books out of 10 allowed.')
                                    AS `message`
                       FROM   (SELECT `sb_subscriber`,
                                      COUNT(`sb_book`) AS `sb_books`
                               FROM   `subscriptions`
                               WHERE  `sb_is_active` = 'Y'
                               AND `sb_subscriber` = NEW.`sb_subscriber`
                               GROUP  BY `sb_subscriber`
                               HAVING `sb_books` >= 10) AS `prepared_data`
                       JOIN `subscribers`
                       ON `sb_subscriber` = `s_id`),
                  '');

    IF (LENGTH(@msg) > 0)
     THEN
      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
     END IF;

  END;
$$

DELIMITER ;
```

In the correct MySQL solution, we only react on giving books to the reader whose identifier appears in the added/modified record. Also, we perform the check before the changes take effect, so the DBMS won't even have to undo them if the operation is forbidden (which will also save some time).

Exploration 4.2.1.EXP.A. Let's perform an exploration on the "Big Library (for Experiments)" database, comparing the speed of the presented wrong and correct solutions for MySQL.

After performing a thousand data insertion queries that violate the problem condition, the median times (in seconds) took the following values:

Wrong solution	Correct solution	Difference, times
51.177	0.009	5686

Let's move on to the solution for MS SQL Server, in which we will also present two options, wrong and correct.

The wrong option completely repeats the same wrong logic of the MySQL solution: we are trying to analyze the full set of information in the database, moreover, we do it after a data modification operation, forcing the DBMS to undo the changes obtained.

MS SQL Solution 4.2.1.b (**wrong** solution)

```
1    CREATE TRIGGER [sbs_cntrl_10_books_ins_upd_WRONG]
2    ON [subscriptions]
3    AFTER INSERT, UPDATE
4    AS
5     DECLARE @bad_records NVARCHAR(max);
6     DECLARE @msg NVARCHAR(max);
7
8     SELECT @bad_records = STUFF((SELECT ', ' + [list]
9                                  FROM  (SELECT CONCAT('(id=', [s_id], ', ',
10                                                 [s_name], ', books=',
11                                                 COUNT([sb_book]), ')') AS [list]
12                                        FROM   [subscribers]
13                                               JOIN [subscriptions]
14                                               ON [s_id] = [sb_subscriber]
15                                        WHERE  [sb_is_active] = 'Y'
16                                        GROUP  BY [s_id], [s_name]
17                                        HAVING COUNT([sb_book]) > 10)
18                                        AS [prepared_data]
19                           FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
20                           1, 2, '');
21
22    IF (LEN(@bad_records) > 0)
23     BEGIN
24      SET @msg = CONCAT('The following readers have more books
25                         than allowed (10 allowed): ', @bad_records);
26      RAISERROR (@msg, 16, 1);
27      ROLLBACK TRANSACTION;
28      RETURN;
29     END;
30   GO
```

The correct solution for MS SQL Server is subject to limitations described in detail in the solution{329} of problem 4.2.1.a{329} (impossibility to create **`INSTEAD OF UPDATE`** trigger without disabling cascade update operation on foreign keys, necessity to calculate primary key value), so here we will also limit ourselves to creating **`INSERT`** trigger only.

However, by limiting the analysis of the problem condition fulfillment to the list of readers whose identifiers are in the **`[inserted]`** pseudo-table, and by performing this analysis before the actual data insertion, we get a chance to significantly increase the speed of our trigger.

MS SQL — Solution 4.2.1.b (trigger on the **`subscriptions`** table)

```
CREATE TRIGGER [sbs_cntrl_10_books_ins_OK]
ON [subscriptions]
INSTEAD OF INSERT
AS
 DECLARE @bad_records NVARCHAR(max);
 DECLARE @msg NVARCHAR(max);

 SELECT @bad_records = STUFF((SELECT ', ' + [list]
                        FROM    (SELECT CONCAT('(id=', [s_id], ', ',
                                        [s_name], ', books=',
                                        COUNT([sb_book]), ')') AS [list]
                                 FROM   [subscribers]
                                        JOIN [subscriptions]
                                        ON [s_id] = [sb_subscriber]
                                 WHERE  [sb_is_active] = 'Y'
                                        AND [sb_subscriber] IN
                                            (SELECT [sb_subscriber]
                                             FROM   [inserted])
                                 GROUP  BY [s_id], [s_name]
                                 HAVING COUNT([sb_book]) >= 10)
                                 AS [prepared_data]
                FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
                1, 2, '');

 IF (LEN(@bad_records) > 0)
  BEGIN
   SET @msg = CONCAT('The following readers have more books
                      than allowed (10 allowed): ', @bad_records);
   RAISERROR (@msg, 16, 1);
   ROLLBACK TRANSACTION;
   RETURN;
  END;

 SET IDENTITY_INSERT [subscriptions] ON;
 INSERT INTO [subscriptions]
             ([sb_id],
              [sb_subscriber],
              [sb_book],
              [sb_start],
              [sb_finish],
              [sb_is_active])
 SELECT ( CASE
              WHEN [sb_id] IS NULL
                OR [sb_id] = 0 THEN IDENT_CURRENT('subscriptions')
                                    + IDENT_INCR('subscriptions')
                                    + ROW_NUMBER() OVER (ORDER BY
                                                         (SELECT 1))
                                    - 1
                                ELSE [sb_id]
          END ) AS [sb_id],
        [sb_subscriber],
        [sb_book],
        [sb_start],
        [sb_finish],
        [sb_is_active]
 FROM   [inserted];
 SET IDENTITY_INSERT [subscriptions] OFF;
GO
```

Exploration 4.2.1.EXP.B. Let's perform an exploration on the "Big Library (for Experiments)" database, comparing the speed of the presented wrong and correct solutions for MS SQL Server.

After performing a thousand data insertion queries that violate the problem condition, the median times (in seconds) took the following values:

Wrong solution	Correct solution	Difference, times
6.598	2.746	2.4

It's not as impressive as with MySQL, but still enough to feel the difference in performance between the wrong and the correct solution.

Let's move on to the solution for Oracle, in which we will also present two options: wrong and correct. Their internal logic will be completely equivalent to the solution of this problem for MySQL, so let's go straight to the code.

Oracle — Solution 4.2.1.b (**wrong** solution)

```
1    CREATE TRIGGER "sbs_ctr_10_bks_ins_upd_WRONG"
2    AFTER INSERT OR UPDATE
3    ON "subscriptions"
4     FOR EACH ROW
5      DECLARE
6       PRAGMA AUTONOMOUS_TRANSACTION;
7       msg NCLOB;
8      BEGIN
9       SELECT NVL((SELECT UTL_RAW.CAST_TO_NVARCHAR2
10                            (
11                             LISTAGG
12                             (
13                              UTL_RAW.CAST_TO_RAW(N'(id=' ||
14                              "sb_subscriber" || N', ' || "s_name" ||
15                              N', books=' || "s_books" || N')'),
16                              UTL_RAW.CAST_TO_RAW(N', ')
17                             )
18                             WITHIN GROUP (ORDER BY "sb_subscriber")
19                            )
20                     FROM   (SELECT "sb_subscriber",
21                                    "s_name",
22                                    COUNT("sb_book") AS "s_books"
23                             FROM   "subscribers"
24                                    JOIN "subscriptions"
25                                    ON "s_id" = "sb_subscriber"
26                             WHERE  "sb_is_active" = 'Y'
27                             GROUP  BY "sb_subscriber", "s_name"
28                             HAVING COUNT("sb_book") > 10) "prepared_data"),
29                 '')
30      INTO msg FROM dual;
31
32      IF (LENGTH(msg) > 0)
33       THEN
34        RAISE_APPLICATION_ERROR(-20001, 'The following readers have
35                                         more books than allowed
36                                         (10 allowed): ' || msg);
37       END IF;
38     END;
```

Since Oracle forbids access from the trigger to the `subscriptions` table in which the insertion is made, we run the trigger in an autonomous transaction (line 6). This peculiarity has to be considered in the correct solution, which we will see right now.

Oracle — Solution 4.2.1.b (trigger on the `subscriptions` table)

```
CREATE TRIGGER "sbs_ctr_10_bks_ins_upd_OK"
BEFORE INSERT OR UPDATE
ON "subscriptions"
 FOR EACH ROW
  DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
   msg NCLOB;
  BEGIN
   SELECT NVL((SELECT (N'Subscriber ' || "s_name" || N' (id=' ||
                       "sb_subscriber" || N') already has ' ||
                       "sb_books" || N' books out of 10 allowed.')
                      AS "message"
                      FROM    (SELECT "sb_subscriber",
                                      COUNT("sb_book") AS "sb_books"
                               FROM   "subscriptions"
                               WHERE  "sb_is_active" = 'Y'
                               AND "sb_subscriber" = :new."sb_subscriber"
                               GROUP  BY "sb_subscriber"
                               HAVING COUNT("sb_book") >= 10)
                              "prepared_data"
                      JOIN "subscribers"
                      ON "sb_subscriber" = "s_id"),
                ''')
    INTO msg FROM dual;

    IF (LENGTH(msg) > 0)
     THEN
      RAISE_APPLICATION_ERROR(-20001, msg);
     END IF;
  END;
```

It remains to check the difference in the speed of the solutions presented.

Exploration 4.2.1.EXP.C. Let's perform an exploration on the "Big Library (for Experiments)" database, comparing the speed of the presented wrong and correct Oracle solutions.

After performing a thousand data insertion queries that violate the problem condition, the median times (in seconds) took the following values:

Wrong solution	Correct solution	Difference, times
99.667	4.961	20

If we summarize all the research results in one table, we get:

DBMS	Wrong solution	Correct solution	Difference, times
MySQL	51.177	0.009	5686
MS SQL Server	6.598	2.746	2.4
Oracle	99.667	4.961	20

Also note that in all DBMSes with the wrong solution in the process of accumulating information about all readers, for which the problem condition is violated, we risk exceeding the maximum allowed string size that accumulates this information. So, the second (correct) solution is not only faster, but also more reliable.

This completes the solution of this problem.

Solution 4.2.1.c[329].

Since the only thing forbidden by the problem is to change the value of the `sb_is_active` field from `N` to `Y` for existing records, we only need the `UPDATE` trigger.

Tasks of this type can be solved very easily and conveniently using row-level triggers (supported by MySQL and Oracle): we use `old` and `new` keywords to access the old and new values of the `sb_is_active` field.

In case of statement-level triggers (only such triggers exist in MS SQL Server) we have to use `JOIN` query from `[inserted]` and `[deleted]` pseudo-tables to match old and new `sb_is_active` field values for each record. Also, we'll have to disable primary key value changes (lines 8-14 of MS SQL Server trigger code) to be able to reliably match old and new values of the monitored field.

Otherwise, the logic behind solving this problem is trivial: if an attempt is made to change data in a way that is forbidden by the problem, we display an error message and rollback the transaction.

Let's start traditionally with the code for MySQL.

MySQL Solution 4.2.1.c (trigger on the `subscriptions` table)

```
1    DELIMITER $$
2
3    CREATE TRIGGER `sbs_cntrl_is_active`
4    BEFORE UPDATE
5    ON `subscriptions`
6     FOR EACH ROW
7      BEGIN
8       IF ((OLD.`sb_is_active` = 'N') AND (NEW.`sb_is_active` = 'Y'))
9        THEN
10        SET @msg = CONCAT('It is prohibited to activate previously
11                           deactivated subscriptions (rule violated
12                           for subscription with id ', NEW.`sb_id`, ').');
13        SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
14       END IF;
15     END;
16   $$
17
18   DELIMITER ;
```

We could take a more performance-optimal way in MS SQL Server and make `INSTEAD OF` trigger, but in this case the trigger code would be more complex.

Since we have already considered such a situation in the solution[342] of problem 4.2.1.b[329] here we will sacrifice performance for the sake of brevity and clarity of the trigger code itself.

MS SQL Solution 4.2.1.c (trigger on the `subscriptions` table)

```
CREATE TRIGGER [sbs_cntrl_is_active]
ON [subscriptions]
AFTER UPDATE
AS
 DECLARE @bad_records NVARCHAR(max);
 DECLARE @msg NVARCHAR(max);

 IF (UPDATE([sb_id]))
  BEGIN
   RAISERROR ('Please, do NOT update surrogate PK
               on table [subscriptions]!', 16, 1);
   ROLLBACK TRANSACTION;
   RETURN;
  END;

 SELECT @bad_records = STUFF((SELECT ', ' +
                                     CAST([inserted].[sb_id] AS NVARCHAR)
                              FROM   [deleted]
                                     JOIN [inserted]
                                     ON [deleted].[sb_id] =
                                        [inserted].[sb_id]
                              WHERE  [deleted].[sb_is_active] = 'N'
                                     AND [inserted].[sb_is_active] = 'Y'
                 FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
                 1, 2, '');

 IF (LEN(@bad_records) > 0)
  BEGIN
   SET @msg = CONCAT('It is prohibited to activate previously
                      deactivated subscriptions (rule violated for
                      subscriptions with id ', @bad_records, ').');
   RAISERROR (@msg, 16, 1);
   ROLLBACK TRANSACTION;
   RETURN;
  END;
GO
```

Finally, we present the solution for Oracle. It differs from the solution for MySQL only syntactically, because the logic of the two solutions is completely identical.

Oracle Solution 4.2.1.c (trigger on the `subscriptions` table)

```
CREATE TRIGGER "sbs_ctr_is_active"
BEFORE UPDATE
ON "subscriptions"
 FOR EACH ROW
  BEGIN
   IF ((:old."sb_is_active" = 'N') AND (:new."sb_is_active" = 'Y'))
    THEN
     RAISE_APPLICATION_ERROR(-20001, 'It is prohibited to activate
                                      previously deactivated subscriptions
                                      (rule violated for subscription with
                                      id ' || :new."sb_id" || ').');
   END IF;
  END;
```

This completes the solution of this problem. You can check its functionality by running queries to update the data in the **`subscriptions`** table so that it either does violate or does not violate the condition of the problem.

Task 4.2.1.TSK.A: create a trigger that does not allow one to add information about a subscription to the database if at least one of the conditions is met:

- the start or finish date of a subscription falls on Sunday;
- the reader has borrowed more than 100 books in the last six months;
- the time between the start and finish dates of a subscription is less than three days.

Task 4.2.1.TSK.B: create a trigger that prohibits giving a book to a reader who has five or more books in their hands, provided that the total time remaining before the return of all books given to them is less than one month.

Task 4.2.1.TSK.C: modify the solution{349} of problem 4.2.1.c{329} for MS SQL Server, changing the `AFTER` trigger to the `INSTEAD OF` trigger.

4.2.2. Example 34: Using Triggers to Control Data Format and Values

Problem 4.2.2.a{352}: create a trigger that allows only those readers whose name contains at least two words and one dot to be registered in the library.

Problem 4.2.2.b{357}: create a trigger that allows only books published not more than a hundred years ago to be registered in the library.

Expected result 4.2.2.a:

If an attempt is made to make changes to the database that contradict the conditions of the problem, the operation (transaction) must be canceled. An error message explaining the problem should also be displayed, e.g.: “Subscribers name should contain at least two words and one point, but the following name violates this rule: IvanovII”.

Expected result 4.2.2.b:

If an attempt is made to make changes that contradict the conditions of the problem, the operation (transaction) must be canceled. An error message explaining the problem should also be displayed, e.g.: “The following issuing year is more than 100 years in the past: 1812”.

Solution 4.2.2.a{352}:

In the solution{342} of problem 4.2.1.b{329} we already discussed in detail the performance advantages of **`BEFORE`** and **`INSTEAD OF`** triggers over **`AFTER`** triggers, so we will not repeat the same reasoning here, but we will go straight to the heart of the problem.

The most difficult thing here is to count the words. And this difficulty is not so much technical as “philosophical”: what is to be considered a word? Let’s agree that as a word we will consider a continuous sequence of letters and signs `-` (minus) and `'` (apostrophe). Accepting this assumption, we can build a universal solution based on regular expressions (for DBMSes that support them).

Exploring the essence of regular expressions is beyond the scope of this book, so here is a ready-made universal variant that should work in the vast majority of DBMSes and programming languages (yes, we can write a more optimal and elegant variant, but that increases the risk of losing universality). Also note that for the same reason of universality we add Cyrillic symbols here.

```
^[a-zA-Za-яA-ЯёЁ\'-]+([^a-zA-Za-яA-ЯёЁ\'-]+[a-zA-Za-яA-ЯёЁ\'.-]+){1,}$
```

The graphical representation of this regular expression is shown in figure 4.a. The Russian letters “ё” are added to the character class as a separate character because they are not included in the “range” of letters of the Russian alphabet.

If for some reason we do not want to or cannot use regular expressions, there is a second way to make sure that there are two words in the string (assuming that the word separator is a space): we should count the number of spaces in the string where the so-called “end spaces” in the beginning and the end are guaranteed to be removed. If this number is greater than zero, then there are definitely at least two words in the string.

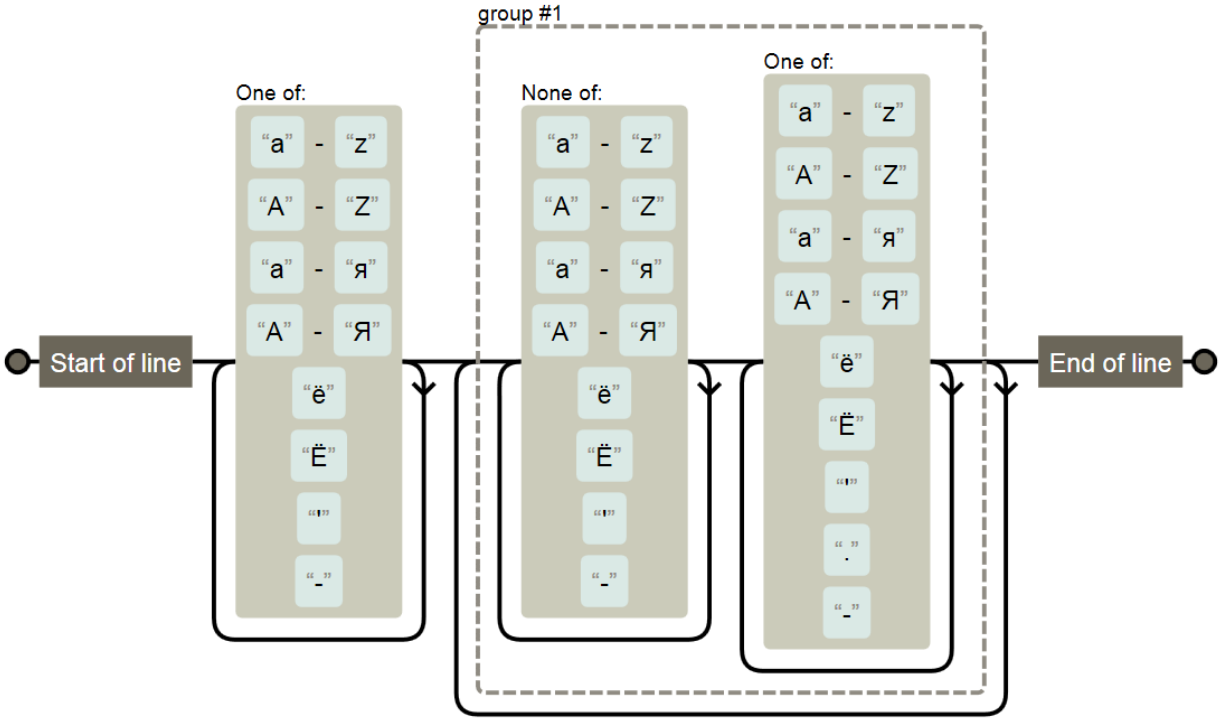


Figure 4.a — Graphical representation of the regular expression[15]

Traditionally we start with MySQL. And we immediately encounter a problem: this DBMS, before version 8.0.4, does not support multibyte strings when using regular expressions[16], so we will have to convert the analyzed values to a single-byte encoding (e.g., CP1251). For Russian alphabet this is safe, but for other languages it can lead to data distortions and malfunction of the regular expression mechanism.

MySQL — Solution 4.2.2.a (triggers on the `subscribers` table) (before version 8.0.4)

```
1   DELIMITER $$
2
3   CREATE TRIGGER `sbsrs_cntrl_name_ins`
4   BEFORE INSERT
5   ON `subscribers`
6    FOR EACH ROW
7     BEGIN
8      IF ((CAST(NEW.`s_name` AS CHAR CHARACTER SET cp1251) REGEXP
9          CAST('^[a-zA-Za-яA-ЯёЁ\'-]+([^a-zA-Za-яA-ЯёЁ\'-]+[a-zA-Za-яA-
10  ЯёЁ\'.-]+){1,}$' AS CHAR CHARACTER SET cp1251)) = 0)
11         OR (LOCATE('.', NEW.`s_name`) = 0)
12      THEN
13       SET @msg = CONCAT('Subscribers name should contain at
14                          least two words and one point, but the following
15                          name violates this rule: ', NEW.`s_name`);
16       SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
17      END IF;
18    END;
19  $$
```

[15] https://regexper.com/#^[a-zA-Z%D0%B0-%D1%8F%D0%90-%D0%AF%D1%91%D0%81\%27-]%2B%28[^a-zA-Z%D0%B0-%D1%8F%D0%90-%D0%AF%D1%91%D0%81\%27-]%2B[a-zA-Z%D0%B0-%D1%8F%D0%90-%D0%AF%D1%91%D0%81\%27-]%2B%29{1%2C}%24

[16] https://dev.mysql.com/doc/refman/8.0/en/regexp.html#operator_regexp

MySQL Solution 4.2.2.a (triggers on the `subscribers` table) (continued) (before version 8.0.4)

```
20  CREATE TRIGGER `sbsrs_cntrl_name_upd`
21  BEFORE UPDATE
22  ON `subscribers`
23   FOR EACH ROW
24    BEGIN
25     IF ((CAST(NEW.`s_name` AS CHAR CHARACTER SET cp1251) REGEXP
26          CAST('^[a-zA-Za-яA-ЯёЁ\'-]+([^a-zA-Za-яA-ЯёЁ\'-]+[a-zA-Za-яA-
27  ЯёЁ\'.-]+){1,}$' AS CHAR CHARACTER SET cp1251)) = 0)
28        OR (LOCATE('.', NEW.`s_name`) = 0)
29      THEN
30       SET @msg = CONCAT('Subscribers name should contain at
31                          least two words and one point, but the following
32                          name violates this rule: ', NEW.`s_name`);
33       SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
34      END IF;
35    END;
36  $$
37
38  DELIMITER ;
```

Starting with MySQL version 8.0.4 there is no need to change the encoding, so the trigger code will be shorter:

MySQL Solution 4.2.2.a (triggers on the `subscribers` table) (version 8.0.4 and newer)

```
1   DELIMITER $$
2
3   CREATE TRIGGER `sbsrs_cntrl_name_ins`
4   BEFORE INSERT
5   ON `subscribers`
6    FOR EACH ROW
7     BEGIN
8      IF ((NEW.`s_name` REGEXP '^[a-zA-Za-яA-ЯёЁ\'-]+([^a-zA-Za-яA-ЯёЁ\'-]+[a-
9   zA-Za-яA-ЯёЁ\'.-]+){1,}$') = 0)
10       OR (LOCATE('.', NEW.`s_name`) = 0)
11     THEN
12      SET @msg = CONCAT('Subscribers name should contain at
13                         least two words and one point, but the following
14                         name violates this rule: ', NEW.`s_name`);
15      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
16     END IF;
17   END;
18  $$
19
20  CREATE TRIGGER `sbsrs_cntrl_name_upd`
21  BEFORE UPDATE
22  ON `subscribers`
23   FOR EACH ROW
24    BEGIN
25     IF ((NEW.`s_name` REGEXP '^[a-zA-Za-яA-ЯёЁ\'-]+([^a-zA-Za-яA-ЯёЁ\'-]+[a-
26  zA-Za-яA-ЯёЁ\'.-]+){1,}$') = 0)
27       OR (LOCATE('.', NEW.`s_name`) = 0)
28     THEN
29      SET @msg = CONCAT('Subscribers name should contain at
30                         least two words and one point, but the following
31                         name violates this rule: ', NEW.`s_name`);
32      SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
33     END IF;
34   END;
35  $$
36
37  DELIMITER ;
```

If you get an error message “Error Code: 1648. Data too long for condition item 'MESSAGE_TEXT'”, try to remove all the spaces from the lines that form the **`@msg`** variable or even shorten the error message a bit and re-create the trigger.

Let’s move on to MS SQL Server. Since this DBMS does not support full-fledged (Perl-compatible, PCRE) regular expressions, here we use an alternative solution, based on the remaining spaces counting in the string.

The second problem of MS SQL Server and its statement-level triggers is that we must forbid changes to the primary key (lines 50-56) in the **`UPDATE`** trigger, otherwise we cannot guarantee that the update operation will be correctly executed in the trigger code.

The third MS SQL Server problem we are already familiar with is related to the need to calculate the value of the auto-incrementable primary key (lines 29-38) in the **`INSERT`** trigger (see explanation in the solution{260} of problem 3.2.1.a{259}).

It’s worth noting that if we have a small amount of data and performance isn’t degraded in any noticeable way by using **`AFTER`** triggers, then the solution of this problem can be made much shorter, simpler, and more versatile (the **`INSERT`** and **`UPDATE`** trigger code will be completely identical). Make sure of this yourself by completing the task 4.2.2.TSK.B{358}.

MS SQL | Solution 4.2.2.a (triggers on the **`subscribers`** table)

```
1     CREATE TRIGGER [sbsrs_cntrl_name_ins]
2     ON [subscribers]
3     INSTEAD OF INSERT
4     AS
5      DECLARE @bad_records NVARCHAR(max);
6      DECLARE @msg NVARCHAR(max);
7
8      SELECT @bad_records = STUFF((SELECT ', ' + [s_name]
9                                   FROM   [inserted]
10                                  WHERE
11                                   CHARINDEX(' ', LTRIM(RTRIM([s_name]))) = 0
12                                OR CHARINDEX('.', [s_name]) = 0
13                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
14                  1, 2, '');
15
16     IF (LEN(@bad_records) > 0)
17      BEGIN
18       SET @msg = CONCAT('Subscribers name should contain at least two
19                          words and one point, but the following names
20                          violate this rule: ', @bad_records);
21       RAISERROR (@msg, 16, 1);
22       ROLLBACK TRANSACTION;
23       RETURN;
24      END;
```

MS SQL Solution 4.2.2.a (triggers on the **subscribers** table) (continued)

```
25    SET IDENTITY_INSERT [subscribers] ON;
26    INSERT INTO [subscribers]
27                ([s_id],
28                 [s_name])
29    SELECT ( CASE
30                  WHEN [s_id] IS NULL
31                    OR [s_id] = 0 THEN IDENT_CURRENT('subscribers')
32                                       + IDENT_INCR('subscribers')
33                                       + ROW_NUMBER() OVER (ORDER BY
34                                                            (SELECT 1))
35                                        - 1
36                                  ELSE [s_id]
37               END ) AS [s_id],
38           [s_name]
39    FROM   [inserted];
40    SET IDENTITY_INSERT [subscribers] OFF;
41  GO
42
43  CREATE TRIGGER [sbsrs_cntrl_name_upd]
44  ON [subscribers]
45  INSTEAD OF UPDATE
46  AS
47   DECLARE @bad_records NVARCHAR(max);
48   DECLARE @msg NVARCHAR(max);
49
50   IF (UPDATE([s_id]))
51    BEGIN
52     RAISERROR ('Please, do NOT update surrogate PK
53                 on table [subscribers]!', 16, 1);
54     ROLLBACK TRANSACTION;
55     RETURN;
56    END;
57
58   SELECT @bad_records = STUFF((SELECT ', ' + [s_name]
59                                FROM   [inserted]
60                                WHERE
61                                 CHARINDEX(' ', LTRIM(RTRIM([s_name]))) = 0
62                               OR CHARINDEX('.', [s_name]) = 0
63                FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
64                1, 2, '');
65
66   IF (LEN(@bad_records) > 0)
67    BEGIN
68     SET @msg = CONCAT('Subscribers name should contain at least two
69                        words and one point, but the following names
70                        violate this rule: ', @bad_records);
71     RAISERROR (@msg, 16, 1);
72     ROLLBACK TRANSACTION;
73     RETURN;
74    END;
75
76   UPDATE [subscribers]
77   SET    [subscribers].[s_name] = [inserted].[s_name]
78   FROM   [subscribers]
79          JOIN [inserted]
80            ON [subscribers].[s_id] = [inserted].[s_id];
81  GO
```

Let's move on to the solution for Oracle, which fully repeats the logic of the solution for MySQL by using the **BEFORE** trigger based on the regular expression and the function of checking the existence of a substring in the string.

Oracle Solution 4.2.2.a (triggers on the **subscribers** table)

```
CREATE TRIGGER "sbsrs_cntrl_name_ins_upd"
BEFORE INSERT OR UPDATE
ON "subscribers"
 FOR EACH ROW
  BEGIN
   IF ((NOT REGEXP_LIKE(:new."s_name", '^[a-zA-Za-яA-ЯёЁ''-]+([^a-zA-Za-яA-
ЯёЁ''-]+[a-zA-Za-яA-ЯёЁ''.-]+){1,}$'))
        OR (INSTRC(:new."s_name", '.', 1, 1) = 0))
    THEN
     RAISE_APPLICATION_ERROR(-20001, 'Subscribers name should contain
                                      at least two words and one point,
                                      but the following name violates
                                      this rule: ' || :new."s_name");
   END IF;
  END;
```

This completes the solution of this problem. You can check its correctness by yourself by executing queries to the **subscribers** table to insert and update data, both violating the problem and not violating.

Solution 4.2.2.b{352}.

Since the condition of this problem is largely similar to the previous one, we will implement the simplest solution (for MS SQL Server we use **AFTER** trigger) and limit ourselves to the code without detailed explanations:

MySQL Solution 4.2.2.b (triggers on the **books** table)

```
DELIMITER $$

CREATE TRIGGER `books_cntrl_year_ins`
BEFORE INSERT
ON `books`
 FOR EACH ROW
  BEGIN
   IF ((YEAR(CURDATE()) - NEW.`b_year`) > 100)
    THEN
     SET @msg = CONCAT('The following issuing year is more than
                        100 years in the past: ', NEW.`b_year`);
     SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
    END IF;
  END;
$$

CREATE TRIGGER `books_cntrl_year_upd`
BEFORE UPDATE
ON `books`
 FOR EACH ROW
  BEGIN
   IF ((YEAR(CURDATE()) - NEW.`b_year`) > 100)
    THEN
     SET @msg = CONCAT('The following issuing year is more than
                        100 years in the past: ', NEW.`b_year`);
     SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1001;
    END IF;
  END;
$$

DELIMITER ;
```

MS SQL — Solution 4.2.2.b (triggers on the `books` table)

```
CREATE TRIGGER [books_cntrl_year_ins_upd]
ON [books]
AFTER INSERT, UPDATE
AS
 DECLARE @bad_records NVARCHAR(max);
 DECLARE @msg NVARCHAR(max);

 SELECT @bad_records = STUFF((SELECT ', ' + CAST([b_year] AS NVARCHAR)
                              FROM   [inserted]
                              WHERE  (YEAR(GETDATE()) - [b_year]) > 100
                FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
                1, 2, '');

 IF (LEN(@bad_records) > 0)
  BEGIN
   SET @msg = CONCAT('The following issuing years are more
                      than 100 years in the past: ', @bad_records);
   RAISERROR (@msg, 16, 1);
   ROLLBACK TRANSACTION;
   RETURN;
  END;
GO
```

Oracle — Solution 4.2.2.b (triggers on the `books` table)

```
CREATE TRIGGER "books_cntrl_year_ins_upd"
BEFORE INSERT OR UPDATE
ON "books"
 FOR EACH ROW
  BEGIN
   IF ((TO_NUMBER(TO_CHAR(SYSDATE, 'YYYY')) - :new."b_year") > 100)
    THEN
     RAISE_APPLICATION_ERROR(-20001, 'The following issuing year is
                                      more than 100 years in the past: '
                                      || :new."b_year");
   END IF;
  END;
```

This completes the solution of this problem. You can check its correctness by yourself by executing queries to the **`books`** table to insert and update data, both violating the problem and not violating.

Task 4.2.2.TSK.A: modify the solution{352} of problem 4.2.2.a{352} for MySQL and Oracle so that regular expressions are not used in the trigger code.

Task 4.2.2.TSK.B: rewrite the solution{352} of problem 4.2.2.a{352} for MS SQL Server using **`AFTER`** triggers.

Task 4.2.2.TSK.C: rewrite the regular expressions in the solution{352} of problem 4.2.2.a{352} for MySQL and Oracle in order to:

- eliminate the need to separately check for a dot in the reader's name;
- allow a dot in any of the words (not just in the second and further, as it is done now).

Task 4.2.2.TSK.D: create a trigger that allows only authors whose names do not contain any characters except letters, numbers, `-` (minus), `'` (apostrophe) and spaces (two or more consecutive spaces are not allowed) to be registered in the library).

4.2.3. Example 35: Using Triggers to Correct Data On-the-Fly

Problem 4.2.3.a{359}: create a trigger that checks for a dot at the end of the reader's name and adds one if it is missing.

Problem 4.2.3.b{362}: create a trigger that changes the finish date of a subscription to "current date plus two months" if the finish date is less than the start date or is in the past.

Expected result 4.2.3.a.

When performing the data modification operation, which violates the problem condition, the data must be corrected in accordance with the problem condition. Also information message like "Value [Ivanov I.I] was automatically changed to [Ivanov I.I.]" should be displayed.

Expected result 4.2.3.b.

When performing a data modification operation that violates the problem condition, the data must be corrected in accordance with the problem condition. There should also be an information message like: "Return date 2020.01.01 is less than giveaway date 2021.01.01. Return date changed to 2021.03.01."

Solution 4.2.3.a{359}.

Following MySQL triggers code differs from many similar examples discussed above only in the **`SQLSTATE`** value: values starting with **`01`** report warnings, not errors. And this is the only way to notify about performed conversion of invalid field value to valid one from MySQL trigger.

Unfortunately, before version 5.7.2[17] MySQL stores warnings not for the entire current session, but only for the last expression, so the full warning can be displayed only with MySQL 5.7.2 or newer.

MySQL — Solution 4.2.3.a (triggers on the **`subscribers`** table)

```
1   DELIMITER $$
2
3   CREATE TRIGGER `sbsrs_name_lp_ins`
4   BEFORE INSERT
5   ON `subscribers`
6    FOR EACH ROW
7     BEGIN
8      IF (SUBSTRING(NEW.`s_name`, -1) <> '.')
9       THEN
10       SET @new_value = CONCAT(NEW.`s_name`, '.');
11       SET @msg = CONCAT('Value [', NEW.`s_name`, '] was automatically
12                          changed to [', @new_value ,']');
13       SET NEW.`s_name` = @new_value;
14       SIGNAL SQLSTATE '01000' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1000;
15      END IF;
16    END;
17  $$
```

[17] http://dev.mysql.com/doc/refman/5.7/en/show-warnings.html

MySQL Solution 4.2.3.a (triggers on the **subscribers** table) (continued)

```
18  CREATE TRIGGER `sbsrs_name_lp_upd`
19  BEFORE UPDATE
20  ON `subscribers`
21   FOR EACH ROW
22    BEGIN
23     IF (SUBSTRING(NEW.`s_name`, -1) <> '.')
24      THEN
25       SET @new_value = CONCAT(NEW.`s_name`, '.');
26       SET @msg = CONCAT('Value [', NEW.`s_name`, '] was automatically
27                          changed to [', @new_value ,']');
28       SET NEW.`s_name` = @new_value;
29       SIGNAL SQLSTATE '01000' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1000;
30      END IF;
31    END;
32  $$
33
34  DELIMITER ;
```

We cannot just as easily and beautifully change the incorrect value to the correct one in MS SQL Server, because this DBMS has no row-level triggers.

So, we have to create **INSTEAD OF** triggers (with all the resulting problems and limitations in the form of "manual" generation of an auto-incrementable primary key on data insertion and prohibition of changing the primary key on data update).

But in MS SQL Server we can get messages from triggers very easily and conveniently. To demonstrate this capability, the code below implements two behaviors:

- On lines 18 and 68: using the **PRINT** keyword (which simply outputs a text message to the console).
- On lines 19 and 69: using the **RAISERROR** function (which with these parameters (see documentation[18]) generates a message that does not cause the operation to stop; moreover, we do not rollback the transaction in the trigger body).

MS SQL Solution 4.2.3.a (triggers on the **subscribers** table)

```
1   CREATE TRIGGER [sbsrs_name_lp_ins]
2   ON [subscribers]
3   INSTEAD OF INSERT
4   AS
5    DECLARE @bad_records NVARCHAR(max);
6    DECLARE @msg NVARCHAR(max);
7
8    SELECT @bad_records = STUFF((SELECT ', ' + '[' + [s_name] + '] -> [' +
9                                        [s_name] + '.]'
10                                FROM   [inserted]
11                                WHERE  RIGHT([s_name], 1) <> '.'
12                  FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
13                  1, 2, '');
14
15   IF (LEN(@bad_records) > 0)
16    BEGIN
17     SET @msg = CONCAT('Some values were changed: ', @bad_records);
18     PRINT @msg;
19     RAISERROR (@msg, 16, 0);
20    END;
```

[18] https://msdn.microsoft.com/en-us/library/ms178592.aspx

MS SQL — Solution 4.2.3.a (triggers on the `subscribers` table) (continued)

```
21    SET IDENTITY_INSERT [subscribers] ON;
22    INSERT INTO [subscribers]
23                ([s_id],
24                 [s_name])
25    SELECT ( CASE
26                 WHEN [s_id] IS NULL
27                   OR [s_id] = 0 THEN IDENT_CURRENT('subscribers')
28                                      + IDENT_INCR('subscribers')
29                                      + ROW_NUMBER() OVER (ORDER BY
30                                                           (SELECT 1))
31                                       - 1
32                                  ELSE [s_id]
33               END ) AS [s_id],
34           ( CASE
35                 WHEN RIGHT([s_name], 1) <> '.'
36                  THEN CONCAT([s_name], '.')
37                  ELSE [s_name]
38               END ) AS [s_name]
39    FROM   [inserted];
40    SET IDENTITY_INSERT [subscribers] OFF;
41  GO
42
43  CREATE TRIGGER [sbsrs_name_lp_upd]
44  ON [subscribers]
45  INSTEAD OF UPDATE
46  AS
47   DECLARE @bad_records NVARCHAR(max);
48   DECLARE @msg NVARCHAR(max);
49
50   IF (UPDATE([s_id]))
51    BEGIN
52     RAISERROR ('Please, do NOT update surrogate PK
53                 on table [subscribers]!', 16, 1);
54     ROLLBACK TRANSACTION;
55     RETURN;
56    END;
57
58   SELECT @bad_records = STUFF((SELECT ', ' + '[' + [s_name] + '] -> [' +
59                                       [s_name] + '.]'
60                                FROM   [inserted]
61                                WHERE  RIGHT([s_name], 1) <> '.'
62                 FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
63                 1, 2, '');
64
65   IF (LEN(@bad_records) > 0)
66    BEGIN
67     SET @msg = CONCAT('Some values were changed: ', @bad_records);
68     PRINT @msg;
69     RAISERROR (@msg, 16, 0);
70    END;
71
72   UPDATE [subscribers]
73   SET    [subscribers].[s_name] =
74            ( CASE
75                WHEN RIGHT([inserted].[s_name], 1) <> '.'
76                THEN CONCAT([inserted].[s_name], '.')
77                ELSE [inserted].[s_name]
78              END )
79   FROM   [subscribers]
80          JOIN [inserted]
81            ON [subscribers].[s_id] = [inserted].[s_id];
82  GO
```

The Oracle solution traditionally replicates the logic of the MySQL solution, the only difference being that here we can only output a text message (generating a warning that does not affect operation execution is currently not possible in Oracle).

To see the displayed message, we must first run the **`SET SERVEROUTPUT ON`** command, which enables displaying the output data.

Oracle — Solution 4.2.3.a (triggers on the **`subscribers`** table)

```
1    CREATE TRIGGER "sbsrs_name_lp_ins_upd"
2    BEFORE INSERT OR UPDATE
3    ON "subscribers"
4     FOR EACH ROW
5     DECLARE
6      new_value NVARCHAR2(150);
7      BEGIN
8       IF (SUBSTR(:new."s_name", -1) <> '.')
9        THEN
10        new_value := CONCAT(:new."s_name", '.');
11        DBMS_OUTPUT.PUT_LINE('Value [' || :new."s_name" ||
12        '] was automatically changed to [' || new_value || ']');
13        :new."s_name" := new_value;
14       END IF;
15     END;
```

This completes the solution of this problem. You can check its correctness by yourself by executing queries to the **`subscribers`** table to insert and update data, both violating the problem and not violating.

Solution 4.2.3.b{359}.

The general logic of solving this problem repeats the logic of the solution{359} of problem 4.2.3.a{359}, and only the following can be considered noteworthy here:

- in MySQL and MS SQL Server we have to make two separate triggers (MySQL is not able to “combine” trigger declarations for several operations, and in MS SQL Server the internal behavior of **`INSERT`** and **`UPDATE`** triggers is different), while in Oracle we get compact identical code, relevant to both operations;
- in MS SQL Server without modification of the database model we can only create **`INSTEAD OF INSERT`** trigger, and to create **`INSTEAD OF UPDATE`** trigger we have to disable cascade update on foreign keys of **`subscriptions`** table and implement corresponding operations to ensure referential integrity in the trigger itself;
- the messages displayed by triggers about automatic correction of the **`sb_finish`** field value in the implementation below may contain the start and end date values in different formats (to avoid this, we should explicitly convert both values to the same date format)).

Apart from that, all the following triggers contain only variations on the above-mentioned operations.

MySQL Solution 4.2.3.b (triggers on the `subscriptions` table)

```
DELIMITER $$

CREATE TRIGGER `sbscs_date_tm_ins`
BEFORE INSERT
ON `subscriptions`
 FOR EACH ROW
  BEGIN
   IF (NEW.`sb_finish` < NEW.`sb_start`) OR (NEW.`sb_finish` < CURDATE())
    THEN
     SET @new_value = DATE_ADD(CURDATE(), INTERVAL 2 MONTH);
     SET @msg = CONCAT('Value [', NEW.`sb_finish`, '] was automatically
                        changed to [', @new_value ,']');
     SET NEW.`sb_finish` = @new_value;
     SIGNAL SQLSTATE '01000' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1000;
    END IF;
  END;
$$

CREATE TRIGGER `sbscs_date_tm_upd`
BEFORE UPDATE
ON `subscriptions`
 FOR EACH ROW
  BEGIN
   IF (NEW.`sb_finish` < NEW.`sb_start`) OR (NEW.`sb_finish` < CURDATE())
    THEN
     SET @new_value = DATE_ADD(CURDATE(), INTERVAL 2 MONTH);
     SET @msg = CONCAT('Value [', NEW.`sb_finish`, '] was automatically
                        changed to [', @new_value ,']');
     SET NEW.`sb_finish` = @new_value;
     SIGNAL SQLSTATE '01000' SET MESSAGE_TEXT = @msg, MYSQL_ERRNO = 1000;
    END IF;
  END;
$$

DELIMITER ;
```

MS SQL Solution 4.2.3.b (triggers on the `subscriptions` table)

```
CREATE TRIGGER [sbscs_date_tm_ins]
ON [subscriptions]
INSTEAD OF INSERT
AS
 DECLARE @bad_records NVARCHAR(max);
 DECLARE @msg NVARCHAR(max);

 SELECT @bad_records =
        STUFF((SELECT ', ' + '[' + CAST([sb_finish] AS NVARCHAR) +
                      '] -> [' + FORMAT(DATEADD(month, 2, GETDATE()),
                                        'yyyy-MM-dd') + ']'
               FROM   [inserted]
               WHERE  ([sb_finish] < [sb_start]) OR
                      ([sb_finish] < GETDATE())
        FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
        1, 2, '');

 IF (LEN(@bad_records) > 0)
  BEGIN
   SET @msg = CONCAT('Some values were changed: ', @bad_records);
   PRINT @msg;
   RAISERROR (@msg, 16, 0);
  END;
```

MS SQL — Solution 4.2.3.b (triggers on the `subscriptions` table) (continued)

```
24    SET IDENTITY_INSERT [subscriptions] ON;
25    INSERT INTO [subscriptions]
26                ([sb_id],
27                 [sb_subscriber],
28                 [sb_book],
29                 [sb_start],
30                 [sb_finish],
31                 [sb_is_active])
32    SELECT ( CASE
33                 WHEN [sb_id] IS NULL
34                   OR [sb_id] = 0 THEN IDENT_CURRENT('subscriptions')
35                                       + IDENT_INCR('subscriptions')
36                                       + ROW_NUMBER() OVER (ORDER BY
37                                                            (SELECT 1))
38                                       - 1
39                                  ELSE [sb_id]
40             END ) AS [sb_id],
41             [sb_subscriber],
42                  [sb_book],
43                  [sb_start],
44           ( CASE
45                 WHEN (([sb_finish] < [sb_start]) OR
46                       ([sb_finish] < GETDATE()))
47                 THEN DATEADD(month, 2, GETDATE())
48                 ELSE [sb_finish]
49             END ) AS [sb_finish],
50            [sb_is_active]
51    FROM    [inserted];
52    SET IDENTITY_INSERT [subscriptions] OFF;
53  GO
54
55  -- Attention! To be able to create this trigger, you need to
56  -- disable cascade update on the foreign keys
57  -- of [subscriptions] table.
58  -- Although, then you would have to refine the trigger
59  -- so that it provides referential integrity.
60  CREATE TRIGGER [sbscs_date_tm_upd]
61  ON [subscriptions]
62  INSTEAD OF UPDATE
63  AS
64   DECLARE @bad_records NVARCHAR(max);
65   DECLARE @msg NVARCHAR(max);
66
67   IF (UPDATE([sb_id]))
68    BEGIN
69     RAISERROR ('Please, do NOT update surrogate PK
70                 on table [subscriptions]!', 16, 1);
71     ROLLBACK TRANSACTION;
72     RETURN;
73    END;
74
75   SELECT @bad_records =
76          STUFF((SELECT ', ' + '[' + CAST([sb_finish] AS NVARCHAR) +
77                        '] -> [' + FORMAT(DATEADD(month, 2, GETDATE()),
78                                          'yyyy-MM-dd') + ']'
79                 FROM   [inserted]
80                 WHERE  ([sb_finish] < [sb_start]) OR
81                        ([sb_finish] < GETDATE())
82          FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
83          1, 2, '');
```

MS SQL — Solution 4.2.3.b (triggers on the **subscriptions** table) (continued)

```
84    IF (LEN(@bad_records) > 0)
85     BEGIN
86      SET @msg = CONCAT('Some values were changed: ', @bad_records);
87      PRINT @msg;
88      RAISERROR (@msg, 16, 0);
89     END;
90
91    UPDATE [subscriptions]
92    SET    [subscriptions].[sb_subscriber] = [inserted].[sb_subscriber],
93           [subscriptions].[sb_book] = [inserted].[sb_book],
94                [subscriptions].[sb_start] = [inserted].[sb_start],
95                [subscriptions].[sb_finish] =
96             ( CASE
97                WHEN (([inserted].[sb_finish] < [inserted].[sb_start]) OR
98                                ([inserted].[sb_finish] < GETDATE()))
99                THEN DATEADD(month, 2, GETDATE())
100               ELSE [inserted].[sb_finish]
101              END ),
102          [subscriptions].[sb_is_active] = [inserted].[sb_is_active]
103   FROM   [subscriptions]
104          JOIN [inserted]
105            ON [subscriptions].[sb_id] = [inserted].[sb_id];
106   GO
```

Oracle — Solution 4.2.3.b (triggers on the **subscriptions** table)

```
1     CREATE TRIGGER "sbscs_date_tm_ins_upd"
2     BEFORE INSERT OR UPDATE
3     ON "subscriptions"
4      FOR EACH ROW
5      DECLARE
6       new_value DATE;
7       BEGIN
8        IF (:new."sb_finish" < :new."sb_start") OR (:new."sb_finish" < SYSDATE)
9         THEN
10         new_value := ADD_MONTHS(2, SYSDATE);
11         DBMS_OUTPUT.PUT_LINE('Value [' || :new."sb_finish" ||
12         '] was automatically changed to [' ||
13         TO_CHAR(new_value, 'YYYY-MM-DD') || ']');
14         :new."sb_finish" := new_value;
15        END IF;
16     END;
```

This completes the solution of this problem. You can check its correctness by yourself by executing queries to the **subscribers** table to insert and update data, both violating the problem and not violating.

Task 4.2.3.TSK.A: modify the solution{362} of problem 4.2.3.b{359} so that the initial and automatically obtained corrected date values in messages output by triggers are always guaranteed to represent the date in the same format (in the current implementation the format of initial and obtained values may be different).

Task 4.2.3.TSK.B: modify the solution {362} of problem 4.2.3.b{359} for MS SQL Server so as to get the possibility to create **INSTEAD OF UPDATE** trigger and at the same time not to lose cascade update of foreign keys of the **subscriptions** table (in other words: to disable cascade update on the foreign keys themselves and to implement it “manually” in **INSTEAD OF UPDATE** trigger).

Task 4.2.3.TSK.C: create a trigger that corrects the book title so that it satisfies the following conditions:

- spaces at the beginning and end of the title are not allowed;
- no repetitive spaces are allowed;
- the first letter in the title should always be in upper case.

Task 4.2.3.TSK.D: create a trigger that changes a subscription start date to the current date, if the date specified in the SQL-query is less than the current date by six months or more.

 © EPAM Systems, 2016–2025

Chapter 5: Using Stored Functions and Stored Procedures

5.1. Using Stored Functions

5.1.1. Example 36: Using Stored Functions for Data Operations

Problem 5.1.1.a{368}: create a stored function that gets as input the dates of subscriptions start and finish and returns the difference between these dates in days, and the suffixes " [OK]", " [NOTICE]", " [WARNING]", respectively, if the difference in days is less than ten, ten to thirty and more than thirty days.

Problem 5.1.1.b{370}: create a stored function that returns a list of free values of autoincrementable primary keys in a specified table (free values are primary key values that are absent in the table and are less than the maximum value used; e.g., if the table has primary keys 1, 3, 8, then 2, 4, 5, 6, 7 are considered free).

Problem 5.1.1.c{382}: create a stored function that updates the data in the `books_statistics` table (see problem 3.1.2.a{229}) and returns a number indicating the change in the quantity of books actually present in the library.

Expected result 5.1.1.a.

The result of the query that retrieves the identifier, the dates of subscription start and finish and the result of the function for cases where the book is not returned should look like this:

sb_id	sb_start	sb_finish	rdns
3	2012-05-17	2012-07-17	61 WARNING
62	2014-08-03	2014-10-03	61 WARNING
86	2014-08-03	2014-09-03	31 WARNING
91	2015-10-07	2015-03-07	-214 OK
99	2015-10-08	2025-11-08	3684 WARNING

Expected result 5.1.1.b.

Three formats for presenting the result are allowed:

- a table of two columns that hold the values of the beginning and the end of the "free keys" ranges;
- a one-column table listing all available "free keys" values;
- a string listing all available "free keys" values.

Expected result 5.1.1.c.

When the function is called, the data in the `books_statistics` table is updated, and the function returns the difference between the previous and the new value of the quantity of books actually present in the library.

Solution 5.1.1.a[{367}].

Stored procedures and functions in general can be extremely complex and non-obvious, and dozens of pages are devoted to their syntactic features in documentation for each DBMS. That is why in the problems under consideration we deliberately said to create such stored procedures, which can be implemented in all three DBMS with minimal differences.

Let's mention one "theoretical feature": in all three solutions of this problem presented below, the functions are declared as deterministic (in MS SQL Server this effect is achieved using the `WITH SCHEMABINDING` clause). It means that each time they are called with the same parameters on the same data sets (it's not relevant to our task, but if we access the data in database tables, it would be important), these functions will return the same values. Specifying this stored function property allows the DBMS to use the internal optimization mechanisms more effectively and increase performance

Otherwise, the logic of the solution is simple: we get two dates at the input, calculate their difference, and save the result to a variable, use the value of this variable to define the text part of the message, then return the result of concatenation of this variable and the text part.

And that's all, next is just the code itself.

Solution for MySQL:

MySQL Solution 5.1.1.a

```
1    DELIMITER $$
2    CREATE FUNCTION READ_DURATION_AND_STATUS(start_date DATE, finish_date DATE)
3    RETURNS VARCHAR(150) DETERMINISTIC
4    BEGIN
5     DECLARE days INT;
6     DECLARE message VARCHAR(150);
7     SET days = DATEDIFF(finish_date, start_date);
8     CASE
9      WHEN (days < 10) THEN SET message = ' OK';
10     WHEN ((days >= 10) AND (days <= 30)) THEN SET message = ' NOTICE';
11     WHEN (days > 30) THEN SET message = ' WARNING';
12    END CASE;
13    RETURN CONCAT(days, message);
14   END$$
15
16   DELIMITER ;
```

To check the correctness of this solution, use the following query:

MySQL Solution 5.1.1.a (functionality check)

```
1    SELECT `sb_id`,
2           `sb_start`,
3           `sb_finish`,
4           READ_DURATION_AND_STATUS(`sb_start`, `sb_finish`) AS `rdns`
5    FROM   `subscriptions`
6    WHERE  `sb_is_active` = 'Y'
```

Solution for MS SQL Server:

MS SQL Solution 5.1.1.a

```
CREATE FUNCTION READ_DURATION_AND_STATUS(@start_date DATE,
                                         @finish_date DATE)
RETURNS NVARCHAR(150)
WITH SCHEMABINDING
AS
BEGIN
 DECLARE @days INT;
 DECLARE @message NVARCHAR(150);

 SET @days = DATEDIFF(day, @start_date, @finish_date);
 SET @message =
  CASE
   WHEN (@days < 10) THEN ' OK'
   WHEN ((@days >= 10) AND (@days <= 30)) THEN ' NOTICE'
   WHEN (@days > 30) THEN ' WARNING'
  END;

 RETURN CONCAT(@days, @message);
END;
GO
```

To check the correctness of the resulting solution, use the following query (note the need to refer to the function by its full name, including the name of the schema):

MS SQL Solution 5.1.1.a (functionality check)

```
SELECT [sb_id],
       [sb_start],
       [sb_finish],
       dbo.READ_DURATION_AND_STATUS([sb_start], [sb_finish]) AS [rdns]
FROM   [subscriptions]
WHERE  [sb_is_active] = 'Y'
```

Solution for Oracle:

Oracle Solution 5.1.1.a

```
CREATE FUNCTION READ_DURATION_AND_STATUS(start_date IN DATE,
                                         finish_date IN DATE)
 RETURN NVARCHAR2
 DETERMINISTIC
 IS
 days NUMBER(10);
 message NVARCHAR2(150);
BEGIN
 SELECT (finish_date - start_date) INTO days FROM dual;
 SELECT CASE
         WHEN (days < 10) THEN ' OK'
         WHEN ((days >= 10) AND (days <= 30)) THEN ' NOTICE'
         WHEN (days > 30) THEN ' WARNING'
        END
 INTO message FROM dual;

 RETURN CONCAT(days, message);
END;
```

To check the correctness of this solution, use the following query:

Oracle Solution 5.1.1.a (functionality check)

```
SELECT "sb_id",
       "sb_start",
       "sb_finish",
       READ_DURATION_AND_STATUS("sb_start", "sb_finish") AS "rdns"
FROM   "subscriptions"
WHERE  "sb_is_active" = 'Y'
```

This completes the solution of this problem.

Solution 5.1.1.b[367].

The logic of solving this problem is most convenient to consider on the example of the **subscriptions** table (there are free primary key values). To make it clearer, let's first add two book issues with primary key values 200 and 202.

It should go like this.

sb_id	Free values
	1 — 1
2	
3	
	4 — 41
42	
	43 — 56
57	
	58 — 60
61	
62	
	63 — 85
86	
	87 — 90
91	
	92 — 94
95	
	96 — 98
99	
100	
	101 — 199
200	
	201 — 201
202	

Unfortunately, the stored functions in MySQL currently have a number of severe limitations, two of which greatly affect the solution of this problem:

- it is impossible to execute dynamic SQL queries inside stored functions (i.e., we cannot pass the table name, and the function will have to be strictly bound to a certain database table; in some simpler cases, this restriction can be bypassed, but, alas, not in this situation);
- stored functions cannot return tables (this limitation can partially be bypassed by returning many values as a string, which can then be processed by the MySQL built-in functions, such as **FIND_IN_SET**).

Thus, the result of the function call will look like this:

```
1,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32,33,34,35,36,37,38,39,40,41,43,44,45,46,47,48,49,50,51,52,53,54,55,56,58,59,60,
63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,87,88,89,90,
92,93,94,96,97,98,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,
117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,
137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,
157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,
177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,
197,198,199,201
```

So, the solution of this problem for MySQL is as follows.

MySQL Solution 5.1.1.b

```
1    DROP FUNCTION IF EXISTS GET_FREE_KEYS_IN_SUBSCRIPTIONS;
2    DELIMITER $$
3    CREATE FUNCTION GET_FREE_KEYS_IN_SUBSCRIPTIONS() RETURNS VARCHAR(21845)
4    DETERMINISTIC
5    BEGIN
6     DECLARE start_value INT DEFAULT 0;
7     DECLARE stop_value  INT DEFAULT 0;
8     DECLARE done        INT DEFAULT 0;
9     DECLARE free_keys_string VARCHAR(21845) DEFAULT '';
10    DECLARE free_keys_cursor CURSOR FOR
11     SELECT `start`,
12            `stop`
13     FROM   (SELECT `min_t`.`sb_id` + 1                    AS `start`,
14                    (SELECT MIN(`sb_id`) - 1
15                     FROM   `subscriptions` AS `x`
16                     WHERE  `x`.`sb_id` > `min_t`.`sb_id`) AS `stop`
17             FROM   `subscriptions` AS `min_t`
18             UNION
19             SELECT 1                                      AS `start`,
20                    (SELECT MIN(`sb_id`) - 1
21                     FROM   `subscriptions` AS `x`
22                     WHERE  `sb_id` > 0)                   AS `stop`
23            ) AS `data`
24     WHERE  `stop` >= `start`
25     ORDER  BY `start`,
26               `stop`;
27    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
28
29    OPEN free_keys_cursor;
30    BEGIN
31    read_loop: LOOP
32     FETCH free_keys_cursor INTO start_value, stop_value;
33     IF done THEN
34      LEAVE read_loop;
35     END IF;
36     for_loop: LOOP
37       SET free_keys_string = CONCAT(free_keys_string, start_value, ',');
38       SET start_value := start_value + 1;
39       IF start_value <= stop_value THEN
40         ITERATE for_loop;
41       END IF;
42       LEAVE for_loop;
43     END LOOP for_loop;
44    END LOOP read_loop;
45    END;
46
47    CLOSE free_keys_cursor;
48    RETURN SUBSTRING(free_keys_string, 1, CHAR_LENGTH(free_keys_string) - 1);
49   END;
50   $$
51   DELIMITER ;
```

This function returns a string with a maximum length of 21845 characters (the limit for **VARCHAR** in UTF encodings).

Lines 6-10 declare the variables:

- **start_value** — the beginning of the “free keys” sequence;
- **stop_value** — the end of the “free keys” sequence;
- **done** — an indication that the cursor has selected all data from the query result;
- **free_keys_string** — a string to accumulate and return the result of the function call;
- **free_keys_cursor** — a cursor for line-by-line access to query results, looking for the beginning and end of “free key” sequences.

Lines 11-26 contain the query that is the “heart” of the whole function, but we’ll come back to it a little later.

Line 27 declares a **NOT FOUND** handler for the cursor declared in line 10 (such a handler is triggered when the end of the data set is reached, or when no data is present at all).

In line 29 the cursor is opened, i.e., the query presented in lines 11-26 is executed and the result rows of its execution are made available.

Lines 31-44 contain a loop, which is executed for each row of the query result:

- in line 32 the next data set from the query result is placed in the **start_value** and **stop_value** variables;
- lines 33-35 checks if it was possible to get the data (if it was not, the loop is exited);
- lines 36-43 contain a nested loop, which is an SQL implementation of the classic FOR loop: all values from **start_value** to **stop_value** are incrementally accumulated in the **free_keys_string**;

In line 47 we close the cursor, and in line 47 we return the result of the function call (with the last comma removed).

Now consider the query in lines 11-25 separately. Its essence is expressed in lines 13-17:

- we take **key_value+1** (logically, the existing key value itself cannot be the beginning of the “free keys” range, so we assume that such a range starts with the next value), and the result is placed in the **start** field;
- then we look for the minimum key value that is greater than the **start** value and subtract 1 from it (logically, the existing value cannot be the end of the “free keys” range, so we assume that such a range ends with the previous value), the result is placed in the **stop** field;
- the condition in line 24 makes it possible to distinguish really existing “free value” ranges from false positives (in really existing ranges the upper limit cannot be smaller than the lower limit).

The **UNION** section in lines 18-22 is needed to detect “free ranges” at the beginning of a sequence of primary key values (from 1 to the first existing value).

If we run this query without the **WHERE** clause (line 24), we get the following set of data (the gray background shows “false positives”, which the **WHERE** clause helps to get rid of):

start	stop
1	1
3	2
4	41
43	56
58	60
62	61
63	85
87	90
92	94
96	98
100	99
101	199
201	201
203	NULL

We can get the result of the function call with the following query.

MySQL Solution 5.1.1.b (query to get the result of the function call)

```
SELECT GET_FREE_KEYS_IN_SUBSCRIPTIONS()
```

So, the solution for MySQL is complete. Let's move on to MS SQL Server. The logic of the main query, as well as the logic of working with cursors and loops, has just been reviewed, so we won't repeat these explanations here.

MS SQL Server does not support dynamic SQL inside stored functions either, but here we can return tables from stored functions. Because of this, in the first option of the solution we just "wrap" the query that returns the initial and final values of the free key ranges into a function.

MS SQL Solution 5.1.1.b (the first option)

```
CREATE FUNCTION GET_FREE_KEYS_IN_SUBSCRIPTIONS()
RETURNS @free_keys TABLE
   (
    [start] INT,
    [stop]  INT
   )
AS
BEGIN
 INSERT @free_keys
  SELECT [start],
         [stop]
  FROM   (SELECT [min_t].[sb_id] + 1                    AS [start],
                 (SELECT MIN([sb_id]) - 1
                  FROM   [subscriptions] AS [x]
                  WHERE  [x].[sb_id] > [min_t].[sb_id]) AS [stop]
          FROM   [subscriptions] AS [min_t]
          UNION
          SELECT 1                                      AS [start],
                 (SELECT MIN([sb_id]) - 1
                  FROM   [subscriptions] AS [x]
                  WHERE  [sb_id] > 0)                   AS [stop]
         ) AS [data]
  WHERE  [stop] >= [start]
  ORDER  BY [start],
            [stop]
 RETURN
END;
GO
```

We can get the result of the function call with the following query.

MS SQL Solution 5.1.1.b (query to get the result of the function call)

```
SELECT * FROM GET_FREE_KEYS_IN_SUBSCRIPTIONS()
```

In the second option of the solution, we return a table with a single field that contains the complete list of free keys. This solution is very similar to the MySQL solution, the only difference being that in the nested loop we do not accumulate values of free keys in a string variable, but we put them in the resulting table.

MS SQL Solution 5.1.1.b (the second option)

```
1     CREATE FUNCTION GET_FREE_KEYS_IN_SUBSCRIPTIONS()
2     RETURNS @free_keys TABLE
3     (
4      [key] INT
5     )
6     AS
7     BEGIN
8      DECLARE @start_value INT;
9      DECLARE @stop_value INT;
10     DECLARE free_keys_cursor CURSOR LOCAL FAST_FORWARD FOR
11      SELECT [start],
12             [stop]
13      FROM   (SELECT [min_t].[sb_id] + 1                    AS [start],
14                     (SELECT MIN([sb_id]) - 1
15                      FROM   [subscriptions] AS [x]
16                      WHERE  [x].[sb_id] > [min_t].[sb_id]) AS [stop]
17              FROM   [subscriptions] AS [min_t]
18              UNION
19              SELECT 1                                      AS [start],
20                     (SELECT MIN([sb_id]) - 1
21                      FROM   [subscriptions] AS [x]
22                      WHERE  [sb_id] > 0)                   AS [stop]
23             ) AS [data]
24      WHERE  [stop] >= [start]
25      ORDER  BY [start],
26                [stop];
27
28     OPEN free_keys_cursor;
29     FETCH NEXT FROM free_keys_cursor INTO @start_value, @stop_value;
30     WHILE @@FETCH_STATUS = 0
31      BEGIN
32       WHILE @start_value <= @stop_value
33        BEGIN
34            INSERT INTO @free_keys([key]) VALUES (@start_value);
35            SET @start_value = @start_value + 1;
36        END;
37       FETCH NEXT FROM free_keys_cursor INTO @start_value, @stop_value;
38      END;
39     CLOSE free_keys_cursor;
40     DEALLOCATE free_keys_cursor;
41
42     RETURN
43    END;
44    GO
```

To get the result of the function call here, as in the first option of the solution, we need to run the following query.

MS SQL Solution 5.1.1.b (query to get the result of the function call)

```
1     SELECT * FROM GET_FREE_KEYS_IN_SUBSCRIPTIONS()
```

Finally, let's implement the third option of the solution, which fully repeats the logic of the MySQL solution.

MS SQL Solution 5.1.1.b (the third option)

```
CREATE FUNCTION GET_FREE_KEYS_IN_SUBSCRIPTIONS()
RETURNS VARCHAR(max)
AS
BEGIN
 DECLARE @start_value INT;
 DECLARE @stop_value INT;
 DECLARE @free_keys_string VARCHAR(max);
 DECLARE free_keys_cursor CURSOR LOCAL FAST_FORWARD FOR
  SELECT [start],
         [stop]
  FROM   (SELECT [min_t].[sb_id] + 1                    AS [start],
                 (SELECT MIN([sb_id]) - 1
                  FROM   [subscriptions] AS [x]
                  WHERE  [x].[sb_id] > [min_t].[sb_id]) AS [stop]
          FROM   [subscriptions] AS [min_t]
          UNION
          SELECT 1                                      AS [start],
                 (SELECT MIN([sb_id]) - 1
                  FROM   [subscriptions] AS [x]
                  WHERE  [sb_id] > 0)                   AS [stop]
         ) AS [data]
  WHERE  [stop] >= [start]
  ORDER  BY [start],
            [stop];

 OPEN free_keys_cursor;
 FETCH NEXT FROM free_keys_cursor INTO @start_value, @stop_value;
 WHILE @@FETCH_STATUS = 0
  BEGIN
   WHILE @start_value <= @stop_value
    BEGIN
      SET @free_keys_string = CONCAT(@free_keys_string,
                                     @start_value, ',');
      SET @start_value = @start_value + 1;
    END;
   FETCH NEXT FROM free_keys_cursor INTO @start_value, @stop_value;
  END;
 CLOSE free_keys_cursor;
 DEALLOCATE free_keys_cursor;

 RETURN LEFT(@free_keys_string, LEN(@free_keys_string) - 1);
END;
GO
```

We can get the result of the function call with the following query.

MS SQL Solution 5.1.1.b (query to get the result of the function call)

```
SELECT dbo.GET_FREE_KEYS_IN_SUBSCRIPTIONS()
```

So, the solution for MS SQL Server is complete. Moving on to Oracle.

Here, although the solution is based on the same main query (considered in the MySQL solution), it is technologically more complex. Moreover, Oracle is the only one among the three DBMSes that allows to execute dynamic SQL queries inside stored functions, which will allow us to fully satisfy the initial problem condition and create a universal function that returns information on the free keys of any table

First of all (for uniformity reasons), we will implement a stored function similar to the solution for MS SQL Server: the function is strictly bound to one table, and the output returns a table with two fields storing the beginning and the end of free keys ranges.

Oracle allows us to implement stored functions that return tables in two ways (see the official documentation or, for example, here[19]), which we will consider:

- with preliminary preparation of all data inside the function and then transferring them to the calling code;
- with instantaneous transfer of data (as soon as they are ready) to the calling code (so called pipelined-functions).

The first solution option: the function still only focuses on one table and returns all the data after it is fully prepared.

Oracle | Solution 5.1.1.b (the first option)

```
1    -- Deleting old versions of data types:
2    DROP TYPE "t_tf_free_keys_table";
3    /
4    DROP TYPE "t_tf_free_keys_row";
5    /
6    -- Creating a data type that describes a row of a resulting table:
7    CREATE TYPE "t_tf_free_keys_row" AS OBJECT (
8      "start" NUMBER,
9      "stop"  NUMBER
10   );
11   /
12   -- Creating a data type that describes a resulting table:
13   CREATE TYPE "t_tf_free_keys_table" IS TABLE OF "t_tf_free_keys_row";
14   /
15
16   -- The function itself:
17   DROP FUNCTION GET_FREE_KEYS_IN_SUBSCRIPTIONS;
18   CREATE OR REPLACE FUNCTION GET_FREE_KEYS_IN_SUBSCRIPTIONS
19   RETURN "t_tf_free_keys_table"
20   AS
21    result_tab  "t_tf_free_keys_table" := "t_tf_free_keys_table"();
22    CURSOR free_keys_cursor IS
23     SELECT "start",
24            "stop"
25     FROM   (SELECT "min_t"."sb_id" + 1                    AS "start",
26                    (SELECT MIN("sb_id") - 1
27                     FROM   "subscriptions" "x"
28                     WHERE  "x"."sb_id" > "min_t"."sb_id") AS "stop"
29             FROM   "subscriptions" "min_t"
30             UNION
31             SELECT 1                                      AS "start",
32                    (SELECT MIN("sb_id") - 1
33                     FROM   "subscriptions" "x"
34                     WHERE  "sb_id" > 1)                   AS "stop"
35             FROM dual
36            ) "data"
37     WHERE  "stop" >= "start"
38     ORDER  BY "start",
39               "stop";
40    BEGIN
41     FOR one_row IN free_keys_cursor
42     LOOP
43      result_tab.extend;
44      result_tab(result_tab.last) :=
45                      "t_tf_free_keys_row"(one_row."start", one_row."stop");
46     END LOOP;
47
48     RETURN result_tab;
49    END;
50   /
```

[19] http://stackoverflow.com/questions/21171349/difference-between-table-function-and-pipelined-function

Before we start to consider the code of the functions themselves, let's note that Oracle requires creating special data types that allow stored functions to return tables (lines 1-14 of all presented solutions are devoted to this very subproblem).

The key differences in the implementation of this function in Oracle (compared to MS SQL Server) are in the logic of working with the cursor and the formation of the final result.

First, a full-fledged FOR loop (lines 41-46) is supported here.

Second, to form the final data set, we need to perform two operations: add a new element to the data set (line 43) and fill it with data (lines 44-45).

Apart from that, there are no fundamental differences from the implementation for MS SQL Server.

To get the result of the function call in this option of the solution we need to execute a query of the following form.

Oracle Solution 5.1.1.b (query to get the result of the function call)

```
1    SELECT * FROM TABLE(GET_FREE_KEYS_IN_SUBSCRIPTIONS)
```

The second solution option: the function returns all data after its complete preparation, but already takes the name of the table and the name of its primary key.

Here, lines 29-47 form the value of a text variable, which is an SQL query formed considering the table name and primary key name obtained through the function parameters.

The second minor difference is that instead of the "normal cursor" (which works for ready-made static SQL queries) we use the so-called **`REF CURSOR`**, which can be used for dynamic SQL.

Oracle Solution 5.1.1.b (the second option)

```
1    -- Deleting old versions of data types:
2    DROP TYPE "t_tf_free_keys_table";
3    /
4    DROP TYPE "t_tf_free_keys_row";
5    /
6    -- Creating a data type that describes a row of a resulting table:
7    CREATE TYPE "t_tf_free_keys_row" AS OBJECT (
8      "start" NUMBER,
9      "stop"  NUMBER
10   );
11   /
12   -- Creating a data type that describes a resulting table:
13   CREATE TYPE "t_tf_free_keys_table" IS TABLE OF "t_tf_free_keys_row";
14   /
15
16   -- The function itself:
17   DROP FUNCTION GET_FREE_KEYS;
18   CREATE OR REPLACE FUNCTION GET_FREE_KEYS (table_name IN VARCHAR2,
19                                             pk_name IN VARCHAR2)
20   RETURN "t_tf_free_keys_table"
21   AS
22    result_tab  "t_tf_free_keys_table" := "t_tf_free_keys_table"();
23    TYPE type_free_keys_cursor IS REF CURSOR;
24    free_keys_cursor type_free_keys_cursor;
25    start_value NUMBER;
26    stop_value NUMBER;
27    final_query VARCHAR2(1024);
```

Oracle Solution 5.1.1.b (the second option) (continued)

```
28  BEGIN
29    final_query := 'SELECT "start",
30                           "stop"
31                    FROM   (SELECT "min_t"."' || pk_name ||
32                            '" + 1                                    AS "start",
33                           (SELECT MIN("' || pk_name || '") - 1
34                            FROM   "' || table_name || '" "x"
35                            WHERE  "x"."' || pk_name || '" > "min_t"."' ||
36                                                          pk_name || '") AS "stop"
37              FROM   "' || table_name || '" "min_t"
38              UNION
39              SELECT 1                                                  AS "start",
40                     (SELECT MIN("' || pk_name || '") - 1
41                      FROM   "' || table_name || '" "x"
42                      WHERE  "' || pk_name || '" > 0)                   AS "stop"
43              FROM dual
44            ) "data"
45     WHERE  "stop" >= "start"
46     ORDER  BY "start",
47               "stop"';
48
49    OPEN free_keys_cursor FOR final_query;
50    LOOP
51     FETCH free_keys_cursor INTO start_value, stop_value;
52     EXIT WHEN free_keys_cursor%NOTFOUND;
53     result_tab.extend;
54     result_tab(result_tab.last) :=
55                                "t_tf_free_keys_row"(start_value, stop_value);
56    END LOOP;
57
58    CLOSE free_keys_cursor;
59    RETURN result_tab;
60  END;
61  /
```

We can get the result of the function call with the following query.

Oracle Solution 5.1.1.b (query to get the result of the function call)

```
1   SELECT * FROM TABLE(GET_FREE_KEYS('subscriptions', 'sb_id'))
```

The third solution option: the function still takes the table name and the primary key, but returns table data without a full generation beforehand (saves memory).

The body of the cursor loop looks different here: instead of forming a new item in the data collection, we fetch data into variables (line 49), check if the operation was successful, exit the loop if there is no more data (line 50), and pass the data to the calling code (line 51).

Oracle Solution 5.1.1.b (the third option)

```
1   -- Deleting old versions of data types:
2   DROP TYPE "t_tf_free_keys_table";
3   /
4   DROP TYPE "t_tf_free_keys_row";
5   /
6   -- Creating a data type that describes a row of a resulting table:
7   CREATE TYPE "t_tf_free_keys_row" AS OBJECT (
8     "start" NUMBER,
9     "stop"  NUMBER
10  );
11  /
12  -- Creating a data type that describes a resulting table:
13  CREATE TYPE "t_tf_free_keys_table" IS TABLE OF "t_tf_free_keys_row";
14  /
```

Oracle Solution 5.1.1.b (the third option) (continued)

```
15  -- The function itself:
16  DROP FUNCTION GET_FREE_KEYS;
17  CREATE OR REPLACE FUNCTION GET_FREE_KEYS (table_name IN VARCHAR2,
18                                            pk_name IN VARCHAR2)
19  RETURN "t_tf_free_keys_table" PIPELINED
20  AS
21   TYPE type_free_keys_cursor IS REF CURSOR;
22   free_keys_cursor type_free_keys_cursor;
23   start_value NUMBER;
24   stop_value NUMBER;
25   final_query VARCHAR2(1024);
26   BEGIN
27    final_query := 'SELECT "start",
28                           "stop"
29                    FROM   (SELECT "min_t"."' || pk_name ||
30                            '" + 1                                    AS "start",
31                    (SELECT MIN("' || pk_name || '") - 1
32                     FROM   "' || table_name || '" "x"
33                     WHERE  "x"."' || pk_name || '" > "min_t"."' ||
34                                                  pk_name || '") AS "stop"
35             FROM   "' || table_name || '" "min_t"
36             UNION
37             SELECT 1                                                 AS "start",
38                    (SELECT MIN("' || pk_name || '") - 1
39                     FROM   "' || table_name || '" "x"
40                     WHERE  "' || pk_name || '" > 0)                  AS "stop"
41             FROM dual
42            ) "data"
43     WHERE  "stop" >= "start"
44     ORDER  BY "start",
45               "stop"';
46
47    OPEN free_keys_cursor FOR final_query;
48    LOOP
49     FETCH free_keys_cursor INTO start_value, stop_value;
50     EXIT WHEN free_keys_cursor%NOTFOUND;
51     PIPE ROW("t_tf_free_keys_row"(start_value, stop_value));
52    END LOOP;
53
54    CLOSE free_keys_cursor;
55    RETURN;
56  END;
57  /
```

We can get the result of the function call with the following query.

Oracle Solution 5.1.1.b (query to get the result of the function call)

```
1   SELECT * FROM TABLE(GET_FREE_KEYS('subscriptions', 'sb_id'))
```

The second and third solutions, even being general in terms of the ability to work with any table, have one small disadvantage: in addition to the name of the processed table we must also pass the name of its primary key. This is, of course, a trifle, but it's easy enough to get rid of.

In the following solution, we consciously (to simplify the logic) do not perform the following checks: the existence of the primary key, the data type of the primary key, the type of the primary key (whether it consists of one field or is a composite, i.e., consists of several fields), etc.

The information about the primary key of the table can be obtained by a query of the form[20]:

Oracle Solution 5.1.1.b (getting information about the primary key of the table)

```
1    SELECT cols.table_name,
2           cols.column_name,
3           cols.position,
4           cons.status,
5           cons.owner
6    FROM   all_constraints cons,
7           all_cons_columns cols
8    WHERE  cols.table_name = 'TABLE_NAME'
9           AND cons.constraint_type = 'P'
10          AND cons.constraint_name = cols.constraint_name
11          AND cons.owner = cols.owner
12   ORDER  BY cols.table_name,
13              cols.position
```

Since we will be interested only in the situation with a simple primary key and only in the table belonging to the current user (on whose behalf the connection is established), we will add two restrictions: **`cons.owner = USER`** shows only data on the current user, **`ROWNUM = 1`** shows only the first field (in case of a composite primary key).

The fourth solution option: refinement of the third option with automatic detection of the primary key name of the table.

In lines 30-37 a query is formed, which will determine the name of the primary key of the processed table, and in line 42 it is executed, and its result is placed in the **`pk_name`** variable.

If you want to uncomment the debug output (lines 40, 45, 68), remember to run the **`SET SERVEROUTPUT ON`** command (otherwise the output will not be displayed).

Oracle Solution 5.1.1.b (the fourth option)

```
1    -- Deleting old versions of data types:
2    DROP TYPE "t_tf_free_keys_table";
3    /
4    DROP TYPE "t_tf_free_keys_row";
5    /
6    -- Creating a data type that describes a row of a resulting table:
7    CREATE TYPE "t_tf_free_keys_row" AS OBJECT (
8      "start" NUMBER,
9      "stop"  NUMBER
10   );
11   /
12   -- Creating a data type that describes a resulting table:
13   CREATE TYPE "t_tf_free_keys_table" IS TABLE OF "t_tf_free_keys_row";
14   /
15
16   -- The function itself:
17   DROP FUNCTION GET_FREE_KEYS;
18   CREATE OR REPLACE FUNCTION GET_FREE_KEYS (table_name IN VARCHAR2)
19   RETURN "t_tf_free_keys_table" PIPELINED
20   AS
21    TYPE type_free_keys_cursor IS REF CURSOR;
22    free_keys_cursor type_free_keys_cursor;
23    start_value NUMBER;
24    stop_value NUMBER;
25    pk_name VARCHAR2(1024);
26    getpk_query VARCHAR2(1024);
27    final_query VARCHAR2(1024);
```

[20] http://stackoverflow.com/questions/5353522/how-to-query-for-a-primary-key-in-oracle-11g

Oracle Solution 5.1.1.b (the fourth option) (continued)

```
28  BEGIN
29
30    getpk_query := 'SELECT cols.column_name FROM all_constraints cons,
31    all_cons_columns cols
32    WHERE cols.table_name = ''' || table_name || '''
33    AND cons.constraint_type = ''P''
34    AND cons.constraint_name = cols.constraint_name
35    AND cons.owner = cols.owner
36    AND cons.owner = USER
37    AND ROWNUM = 1';
38
39    -- Uncomment to see the entire query to get the PK name:
40    -- DBMS_OUTPUT.PUT_LINE(getpk_query);
41
42    EXECUTE IMMEDIATE getpk_query INTO pk_name;
43
44    -- Uncomment to see the PK name:
45    -- DBMS_OUTPUT.PUT_LINE(pk_name);
46
47    final_query := 'SELECT "start",
48                           "stop"
49                    FROM   (SELECT "min_t"."' || pk_name ||
50                            '" + 1                                    AS "start",
51                           (SELECT MIN("' || pk_name || '") - 1
52                            FROM   "' || table_name || '" "x"
53                            WHERE  "x"."' || pk_name || '" > "min_t"."' ||
54                                                      pk_name || '") AS "stop"
55             FROM   "' || table_name || '" "min_t"
56             UNION
57             SELECT 1                                                 AS "start",
58                    (SELECT MIN("' || pk_name || '") - 1
59                     FROM   "' || table_name || '" "x"
60                     WHERE  "' || pk_name || '" > 0)                  AS "stop"
61             FROM dual
62            ) "data"
63     WHERE  "stop" >= "start"
64     ORDER  BY "start",
65               "stop"';
66
67    -- Uncomment to see the final query:
68    -- DBMS_OUTPUT.PUT_LINE(final_query);
69
70    OPEN free_keys_cursor FOR final_query;
71    LOOP
72     FETCH free_keys_cursor INTO start_value, stop_value;
73     EXIT WHEN free_keys_cursor%NOTFOUND;
74     PIPE ROW("t_tf_free_keys_row"(start_value, stop_value));
75    END LOOP;
76
77    CLOSE free_keys_cursor;
78    RETURN;
79  END;
80  /
```

We can get the result of the function call with the following query.

Oracle Solution 5.1.1.b (query to get the result of the function call)

```
1   SELECT * FROM TABLE(GET_FREE_KEYS('subscriptions'))
```

This completes the solution of this problem.

To implement two more options for the behavior of the function, in which it returns a table of one field with a list of keys or a string with a list of keys you are asked to do in the task 5.1.1.TSK.C{384}.

Solution 5.1.1.c[367].

Since the solution of this problem is based on the solution[230] of problem 3.1.2.a[229], we only need to "wrap" the **UPDATE** query into a function. We will get the initial and final values of the quantity of books in the library by the usual **SELECT** query before and after the data update.

We should also note that:

- there is no solution of this problem for MS SQL Server, because this DBMS does not allow to perform data modification operations in stored functions;
- the solution of this problem for Oracle has to be implemented by deleting the previously created materialized view and creating an aggregating table by analogy with MySQL, because otherwise the problem makes no sense, as the data in the materialized view is already up to date, so function will always return the 0 value.

In problem 3.1.2.a[229] we needed to create a view that stores the actual values of the aggregated data, but MySQL does not support such views, which is very convenient for solving this problem: in MySQL, the real table, whose data we will update, plays the role of such a view.

MySQL Solution 5.1.1.c (table creation and data initialization)

```
1    -- Table creation:
2    CREATE TABLE `books_statistics`
3    (
4          `total` INTEGER UNSIGNED NOT NULL,
5          `given` INTEGER UNSIGNED NOT NULL,
6          `rest`  INTEGER UNSIGNED NOT NULL
7    );
8
9    -- Data initialization:
10   INSERT INTO `books_statistics`
11               (`total`,
12                `given`,
13                `rest`)
14   SELECT IFNULL(`total`, 0),
15          IFNULL(`given`, 0),
16          IFNULL(`total` - `given`, 0) AS `rest`
17   FROM    (SELECT (SELECT SUM(`b_quantity`)
18                    FROM   `books`)              AS `total`,
19                   (SELECT COUNT(`sb_book`)
20                    FROM   `subscriptions`
21                    WHERE  `sb_is_active` = 'Y') AS `given`)
22         AS `prepared_data`;
```

Inside the function code, we only need to put a query to update the data. We can get the result of the function call with the following query.

MySQL Solution 5.1.1.c (query to get the result of the function call)

```
1    SELECT BOOKS_DELTA()
```

MySQL Solution 5.1.1.c (the function code)

```
DROP FUNCTION IF EXISTS BOOKS_DELTA;
DELIMITER $$
CREATE FUNCTION BOOKS_DELTA() RETURNS INT
BEGIN
 DECLARE old_books_count INT DEFAULT 0;
 DECLARE new_books_count INT DEFAULT 0;

 SET old_books_count := (SELECT `total` FROM `books_statistics`);

 UPDATE `books_statistics`
 JOIN
 (SELECT IFNULL(`total`, 0) AS `total`,
         IFNULL(`given`, 0) AS `given`,
         IFNULL(`total` - `given`, 0) AS `rest`
  FROM   (SELECT (SELECT SUM(`b_quantity`)
                  FROM   `books`)              AS `total`,
                 (SELECT COUNT(`sb_book`)
                  FROM   `subscriptions`
                  WHERE  `sb_is_active` = 'Y') AS `given`)
         AS `prepared_data`) AS `src`
 SET `books_statistics`.`total` = `src`.`total`,
     `books_statistics`.`given` = `src`.`given`,
     `books_statistics`.`rest` = `src`.`rest`;

 SET new_books_count := (SELECT `total` FROM `books_statistics`);

 RETURN (new_books_count - old_books_count);
END;
$$
DELIMITER ;
```

Solution for MySQL is ready, and since there is no solution for MS SQL Server, we go straight to Oracle, where we reproduce the entire logic of the solution for MySQL.

Oracle Solution 5.1.1.c (table creation and data initialization)

```
-- Table creation:
CREATE TABLE "books_statistics"
(
      "total" NUMBER(10),
      "given" NUMBER(10),
      "rest"  NUMBER(10)
);

-- Data initialization:
INSERT INTO "books_statistics"
            ("total",
             "given",
             "rest")
SELECT "total",
       "given",
       ("total" - "given") AS "rest"
FROM   (SELECT SUM("b_quantity") AS "total"
        FROM   "books")
  JOIN (SELECT COUNT("sb_book") AS "given"
        FROM   "subscriptions"
        WHERE  "sb_is_active" = 'Y')
  ON 1 = 1;
```

Oracle Solution 5.1.1.c (the function code)

```
1   CREATE OR REPLACE FUNCTION BOOKS_DELTA RETURN NUMBER IS
2    PRAGMA AUTONOMOUS_TRANSACTION;
3    old_books_count NUMBER;
4    new_books_count NUMBER;
5   BEGIN
6    SELECT "total" INTO old_books_count FROM "books_statistics";
7    COMMIT;
8
9    UPDATE "books_statistics"
10   SET ("total", "given", "rest") =
11   (SELECT "total",
12           "given",
13           ("total" - "given") AS "rest"
14    FROM   (SELECT SUM("b_quantity") AS "total"
15            FROM   "books")
16      JOIN (SELECT COUNT("sb_book") AS "given"
17            FROM   "subscriptions"
18            WHERE  "sb_is_active" = 'Y')
19      ON 1 = 1);
20   COMMIT;
21
22   SELECT "total" INTO new_books_count FROM "books_statistics";
23   COMMIT;
24
25   RETURN (new_books_count - old_books_count);
26  END;
```

Since Oracle has a number of restrictions on modifying data from stored functions code, we need to implement two ideas:

- perform the function in autonomous transaction (line 2);
- commit the transaction in the body of the function after each query execution (lines 7, 20, 23), because otherwise there is a situation of the deadlock between the queries for reading and updating the data).

We can get the result of the function call by running the following query.

Oracle Solution 5.1.1.c (query to get the result of the function call)

```
1   SELECT BOOKS_DELTA FROM dual
```

This completes the solution of this problem.

Task 5.1.1.TSK.A: create a stored function that receives a reader's identifier and returns a list of identifiers for the books they have already read and returned to the library.

Task 5.1.1.TSK.B: create a stored function that returns a list of the first range of free values of autoincrementable primary keys in the specified table (e.g., if a table has primary keys 1, 4, 8, then the first free range are values 2 and 3).

Task 5.1.1.TSK.C: supplement the solution{370} of problem 5.1.1.b{367} for Oracle with two options for implementing the stored function, in which:

- the function returns a table of one field that stores the entire list of free key values;
- the function returns a comma-separated string listing all the values of free keys.

Task 5.1.1.TSK.D: create a stored function that updates the data in the `subscriptions_ready` table (see problem 3.1.2.b{229}) and returns a number that shows the difference in subscriptions count.

5.1.2. Example 37: Using Stored Functions for Data Control

Problem 5.1.2.a{385}: create a stored function that automates the checking of conditions of problem 4.2.1.a{329}, i.e., it must return value 1 (all conditions are met) or -1, -2, -3 (if at least one condition is violated, the number module corresponds to the condition number), depending on whether the following conditions are met:

- the start date of a subscription is in the future;
- the finish date of a subscription in the past (only for data insertion);
- the return date is less than the start date.

Problem 5.1.2.b{388}: create a stored function that automates the checking of conditions of problem 4.2.2.a{352}, i.e., it must return 1 if the reader's name contains at least two words and one dot, and 0 if this condition is violated.

Expected result 5.1.2.a.

The function returns 1 if all conditions are met, and -1, -2, -3 if a corresponding condition is violated.

Expected result 5.1.2.b.

The function returns 1 if the condition is met, and 0 if it is violated.

Solution 5.1.2.a{385}.

In all three DBMSes the logic for solving this problem will be the same: we will pass three parameters into the function (subscription start date, subscription finish date, the sign of data update), we will check the fulfillment of conditions in the body of the function and return the corresponding number.

MySQL | Solution 5.1.2.a (the function code)

```
1    DELIMITER $$
2    CREATE FUNCTION CHECK_SUBSCRIPTION_DATES(sb_start DATE,
3                                             sb_finish DATE,
4                                             is_insert INT)
5    RETURNS INT
6    DETERMINISTIC
7    BEGIN
8     DECLARE result INT DEFAULT 1;
9
10    -- Blocking the subscriptions with the start date in the future.
11    IF (sb_start > CURDATE())
12     THEN
13      SET result = -1;
14     END IF;
15
16    -- Blocking the subscriptions with the finish date in the past.
17    IF ((sb_finish < CURDATE()) AND (is_insert = 1))
18     THEN
19      SET result = -2;
20     END IF;
```

MySQL Solution 5.1.2.a (the function code) (continued)

```
21    -- Blocking the subscriptions with the finish date less than start date.
22    IF (sb_finish < sb_start)
23     THEN
24      SET result = -3;
25     END IF;
26
27     RETURN result;
28    END;
29    $$
30    DELIMITER ;
```

MySQL Solution 5.1.2.a (code to check the functionality of the solution)

```
1     SELECT CHECK_SUBSCRIPTION_DATES('2025-01-01', '2026-01-01', 1);
2     SELECT CHECK_SUBSCRIPTION_DATES('2025-01-01', '2026-01-01', 0);
3
4     SELECT CHECK_SUBSCRIPTION_DATES('2005-01-01', '2006-01-01', 1);
5     SELECT CHECK_SUBSCRIPTION_DATES('2005-01-01', '2006-01-01', 0);
6
7     SELECT CHECK_SUBSCRIPTION_DATES('2005-01-01', '2004-01-01', 1);
8     SELECT CHECK_SUBSCRIPTION_DATES('2005-01-01', '2004-01-01', 0);
```

MS SQL Solution 5.1.2.a (the function code)

```
1     CREATE FUNCTION CHECK_SUBSCRIPTION_DATES(@sb_start DATE,
2                                              @sb_finish DATE,
3                                              @is_insert INT)
4     RETURNS INT
5     WITH SCHEMABINDING
6     AS
7     BEGIN
8      DECLARE @result INT = 1;
9
10    -- Blocking the subscriptions with the start date in the future.
11    IF (@sb_start > CONVERT(date, GETDATE()))
12     BEGIN
13      SET @result = -1;
14     END;
15
16    -- Blocking the subscriptions with the finish date in the past.
17    IF ((@sb_finish < CONVERT(date, GETDATE())) AND (@is_insert = 1))
18     BEGIN
19      SET @result = -2;
20     END;
21
22    -- Blocking the subscriptions with the finish date less than start date.
23    IF (@sb_finish < @sb_start)
24     BEGIN
25      SET @result = -3;
26     END;
27
28     RETURN @result;
29    END;
```

MS SQL Solution 5.1.2.a (code to check the functionality of the solution)

```
1     SELECT dbo.CHECK_SUBSCRIPTION_DATES('2025-01-01', '2026-01-01', 1);
2     SELECT dbo.CHECK_SUBSCRIPTION_DATES('2025-01-01', '2026-01-01', 0);
3
4     SELECT dbo.CHECK_SUBSCRIPTION_DATES('2005-01-01', '2006-01-01', 1);
5     SELECT dbo.CHECK_SUBSCRIPTION_DATES('2005-01-01', '2006-01-01', 0);
6
7     SELECT dbo.CHECK_SUBSCRIPTION_DATES('2005-01-01', '2004-01-01', 1);
8     SELECT dbo.CHECK_SUBSCRIPTION_DATES('2005-01-01', '2004-01-01', 0);
```

Oracle Solution 5.1.2.a (the function code)

```
CREATE OR REPLACE FUNCTION CHECK_SUBSCRIPTION_DATES (sb_start DATE,
                                                     sb_finish DATE,
                                                     is_insert INT)
RETURN NUMBER DETERMINISTIC IS
 result_value NUMBER := 1;
BEGIN
 -- Blocking the subscriptions with the start date in the future.
 IF (sb_start > TRUNC(SYSDATE))
  THEN
   result_value := -1;
 END IF;

 -- Blocking the subscriptions with the finish date in the past.
 IF ((sb_finish < TRUNC(SYSDATE)) AND (is_insert = 1))
  THEN
   result_value := -2;
 END IF;

 -- Blocking the subscriptions with the finish date less than start date.
 IF (sb_finish < sb_start)
  THEN
   result_value := -3;
 END IF;

 RETURN result_value;
END;
```

Oracle Solution 5.1.2.a (code to check the functionality of the solution)

```
SELECT CHECK_SUBSCRIPTION_DATES(TO_DATE('2025-01-01', 'YYYY-MM-DD'),
TO_DATE('2026-01-01', 'YYYY-MM-DD'), 1) FROM dual;
SELECT CHECK_SUBSCRIPTION_DATES(TO_DATE('2025-01-01', 'YYYY-MM-DD'),
TO_DATE('2026-01-01', 'YYYY-MM-DD'), 0) FROM dual;

SELECT CHECK_SUBSCRIPTION_DATES(TO_DATE('2005-01-01', 'YYYY-MM-DD'),
TO_DATE('2006-01-01', 'YYYY-MM-DD'), 1) FROM dual;
SELECT CHECK_SUBSCRIPTION_DATES(TO_DATE('2005-01-01', 'YYYY-MM-DD'),
TO_DATE('2006-01-01', 'YYYY-MM-DD'), 0) FROM dual;

SELECT CHECK_SUBSCRIPTION_DATES(TO_DATE('2005-01-01', 'YYYY-MM-DD'),
TO_DATE('2004-01-01', 'YYYY-MM-DD'), 1) FROM dual;
SELECT CHECK_SUBSCRIPTION_DATES(TO_DATE('2005-01-01', 'YYYY-MM-DD'),
TO_DATE('2004-01-01', 'YYYY-MM-DD'), 0) FROM dual;
```

The function code itself is primitive and does not need explanation, but it is worth noting the difference between the logic of the solution presented here and the logic of the solution{329} of problem 4.2.1.a{329}.

Previously, we stopped the operation (rolled back the transaction) upon detecting the first violation of the condition, so it was extremely difficult to get a message that the finish date was less than the start date: to fulfill this condition, two previous checks (that the start date is in the future and that the return date is in the past) must not be triggered.

Here we return the result only at the very end of the function body in the solutions for all three DBMSes, so the failure of each subsequent check will cancel the previous check's failure indication.

If this behavior appears inconvenient, we can always rewrite the code, returning the result after the first check, which detects a violation of the problem condition.

This completes the solution of this problem.

Solution 5.1.2.b[385].

Compared to the previous problem, this is even simpler: we just need to “wrap” two checks into a function, and based on their results return 1 or 0.

But there will be one unusual detail in the solution for MS SQL Server: we have to use an intermediate variable and return its value at the very end of the function body, because this DBMS requires that the last expression inside the stored function is the **RETURN** operator.

MySQL Solution 5.1.2.b (the function code)

```
DROP FUNCTION IF EXISTS CHECK_SUBSCRIBER_NAME;
DELIMITER $$
CREATE FUNCTION CHECK_SUBSCRIBER_NAME(subscriber_name VARCHAR(150)) RETURNS
INT DETERMINISTIC
BEGIN
   IF ((CAST(subscriber_name AS CHAR CHARACTER SET cp1251) REGEXP
        CAST('^[a-zA-Za-яA-ЯёЁ\'-]+([^a-zA-Za-яA-ЯёЁ\'-]+[a-zA-Za-яA-
ЯёЁ\'.-]+){1,}$' AS CHAR CHARACTER SET cp1251)) = 0)
      OR (LOCATE('.', subscriber_name) = 0)
    THEN
     RETURN 0;
    ELSE
     RETURN 1;
    END IF;
END;
$$
DELIMITER ;
```

MySQL Solution 5.1.2.b (code to check the functionality of the solution)

```
SELECT CHECK_SUBSCRIBER_NAME('Ivanov');
SELECT CHECK_SUBSCRIBER_NAME('Ivanov I');
SELECT CHECK_SUBSCRIBER_NAME('Ivanov I.');
```

MS SQL Solution 5.1.2.b (the function code)

```
CREATE FUNCTION CHECK_SUBSCRIBER_NAME(@subscriber_name NVARCHAR(150))
RETURNS INT
WITH SCHEMABINDING
AS
BEGIN
 DECLARE @result INT = -1;

 IF ((CHARINDEX(' ', LTRIM(RTRIM(@subscriber_name))) = 0) OR
     (CHARINDEX('.', @subscriber_name) = 0))
  BEGIN
   SET @result = 0;
  END
 ELSE
  BEGIN
   SET @result = 1;
  END;

 RETURN @result;
END;
GO
```

MS SQL Solution 5.1.2.b (code to check the functionality of the solution)

```
SELECT dbo.CHECK_SUBSCRIBER_NAME('Ivanov');
SELECT dbo.CHECK_SUBSCRIBER_NAME('Ivanov I');
SELECT dbo.CHECK_SUBSCRIBER_NAME('Ivanov I.');
```

Oracle | Solution 5.1.2.b (the function code)

```
1     CREATE OR REPLACE
2     FUNCTION CHECK_SUBSCRIBER_NAME (subscriber_name NVARCHAR2)
3     RETURN NUMBER DETERMINISTIC IS
4     BEGIN
5      IF ((NOT REGEXP_LIKE(subscriber_name, '^[a-zA-Za-яA-ЯёЁ''-]+([^a-zA-Za-яA-
6     ЯёЁ''-]+[a-zA-Za-яA-ЯёЁ''.-]+){1,}$'))
7           OR (INSTRC(subscriber_name, '.', 1, 1) = 0))
8         THEN
9          RETURN 0;
10        ELSE
11         RETURN 1;
12        END IF;
13    END;
```

Oracle | Solution 5.1.2.b (code to check the functionality of the solution)

```
1     SELECT CHECK_SUBSCRIBER_NAME(N'Ivanov') FROM dual;
2     SELECT CHECK_SUBSCRIBER_NAME(N'Ivanov I') FROM dual;
3     SELECT CHECK_SUBSCRIBER_NAME(N'Ivanov I.') FROM dual;
```

This completes the solution of this problem.

Task 5.1.2.TSK.A: rewrite the solutions{329}, {352} of problems 4.2.1.a{329} and 4.2.2.a{352} using the stored functions created in solutions{385}, {388} of problems 5.1.2.a{385} and 5.1.2.b{385} respectively.

Task 5.1.2.TSK.B: create a stored function that automates the checking of the conditions of problem 4.2.1.b{329}, i.e., it must return 1 if the a reader now has less than ten books in hand, and 0 otherwise.

Task 5.1.2.TSK.C: create a stored function that automates the checking of the conditions of problem 4.2.2.b{352}, i.e., it must return 1 if a book was published less than a hundred years ago, and 0 otherwise.

5.2. Using Stored Procedures

5.2.1. Example 38: Using Stored Procedures to Execute Dynamic Queries

Problem 5.2.1.a{390}: create a stored procedure that eliminates gaps in the sequence of primary key values for a given table (e.g., if the primary key values were 4, 7, 9, after executing the stored procedure they will become 1, 2, 3)).

Problem 5.2.1.b{397}: create a stored procedure that generates a list of views, triggers and foreign keys for a specified table.

Expected result 5.2.1.a.

After execution of the stored procedure, in which the first two parameters are the name of the processed table and its primary key, the primary key values in the table take the form 1, 2, 3, ... (i.e., they start from 1 and go without gaps), and the stored procedure itself returns information about how many primary key values were changed.

Expected result 5.2.1.b.

After execution of the stored procedure, in which the first parameter is the name of the table to be processed, the resulting table is formed and returned in the form of:

object_type	object_name
foreign_key	FK_1
foreign_key	FK_2
trigger	TRG_1
trigger	TRG_2
view	VIEW_1
view	VIEW_2

Solution 5.2.1.a{390}.

Traditionally, let's start solving the problem with MySQL. Unlike stored functions, stored procedures of this DBMS allow us to form and execute dynamic SQL queries.

Before we start discussing the stored procedure code, let's make two important remarks:

- we can perform dynamic queries and put their results into variables only by using so called "session variables[21]" (the names of which begin with the `@` sign);
- the names of variables in which the result of query execution is placed must **not** coincide with the names of parameters of the stored procedure (and, in some cases, with the names of fields returned by the query[22]).

[21] http://stackoverflow.com/questions/1009954/mysql-variable-vs-variable-whats-the-difference

[22] http://dba.stackexchange.com/questions/112285/select-into-variable-results-in-null-or-idk

MySQL | Solution 5.2.1.a (the procedure code)

```
DROP PROCEDURE COMPACT_KEYS;
DELIMITER $$

CREATE PROCEDURE COMPACT_KEYS (IN table_name VARCHAR(150),
                               IN pk_name VARCHAR(150),
                               OUT keys_changed INT)
BEGIN
 SET keys_changed = 0;
 SELECT
 CONCAT('Point 1. table_name = ', table_name, ', pk_name = ',
        pk_name, ', keys_changed = ', IFNULL(keys_changed, 'NULL'));

 SET @empty_key_query =
 CONCAT('SELECT MIN(`empty_key`) AS `empty_key` INTO @empty_key_value
        FROM   (SELECT `left`.`', pk_name, '` + 1 AS `empty_key`
                FROM   `', table_name, '` AS `left`
                       LEFT OUTER JOIN `', table_name, '` AS `right`
                                    ON `left`.`', pk_name,
                                       '` + 1 = `right`.`', pk_name, '`
         WHERE  `right`.`', pk_name, '` IS NULL
         UNION
         SELECT 1 AS `empty_key`
         FROM   `', table_name, '`
         WHERE  NOT EXISTS(SELECT `', pk_name, '`
                           FROM   `', table_name, '`
                           WHERE  `', pk_name, '` = 1)) AS `prepared_data`
        WHERE  `empty_key` < (SELECT MAX(`', pk_name, '`)
                              FROM   `', table_name, '`)');

 SET @max_key_query =
     CONCAT('SELECT MAX(`', pk_name, '`)
             INTO @max_key_value FROM `', table_name, '`');
 SELECT CONCAT('Point 2. empty_key_query = ', @empty_key_query,
               'max_key_query = ', @max_key_query);

 PREPARE empty_key_stmt FROM @empty_key_query;
 PREPARE max_key_stmt FROM @max_key_query;

 while_loop: LOOP
  EXECUTE empty_key_stmt;
  SELECT CONCAT('Point 3. @empty_key_value = ', @empty_key_value);

  IF (@empty_key_value IS NULL)
   THEN LEAVE while_loop;
  END IF;

  EXECUTE max_key_stmt;
  SET @update_key_query =
      CONCAT('UPDATE `', table_name, '` SET `', pk_name,
      '` = @empty_key_value WHERE `', pk_name, '` = ', @max_key_value);
  SELECT CONCAT('Point 4. @update_key_query = ', @update_key_query);

  PREPARE update_key_stmt FROM @update_key_query;
  EXECUTE update_key_stmt;
  DEALLOCATE PREPARE update_key_stmt;

  SET keys_changed = keys_changed + 1;
  ITERATE while_loop;
 END LOOP while_loop;

 DEALLOCATE PREPARE max_key_stmt;
 DEALLOCATE PREPARE empty_key_stmt;
END;
$$
DELIMITER ;
```

Despite the overall cumbersomeness and apparent complexity, the logic of this procedure is very simple. Let's consider it in detail.

The queries on lines 10-11, 33-34, 41, and 51 are presented solely for debugging and clarity, and may be omitted.

Line 8 initializes the value of the output parameter, which will accumulate information about the number of changes made in the table.

In lines 13-28 the text of the SQL query that will search for the first free value in the sequence of primary key values is formed based on the names of the processed table and its primary key passed to the stored procedure. Let's consider this query separately (by the example of **`subscriptions`** table).

MySQL — Solution 5.2.1.a (query code searching for the first free value of the primary key)

```
1    SELECT MIN(`empty_key`) AS `empty_key`
2      FROM   (SELECT `left`.`sb_id` + 1 AS `empty_key`
3              FROM   `subscriptions` AS `left`
4                     LEFT OUTER JOIN `subscriptions` AS `right`
5                                  ON `left`.`sb_id` + 1 = `right`.`sb_id`
6              WHERE  `right`.`sb_id` IS NULL
7              UNION
8              SELECT 1 AS `empty_key`
9              FROM   `subscriptions`
10             WHERE  NOT EXISTS(SELECT `sb_id`
11                               FROM   `subscriptions`
12                               WHERE  `sb_id` = 1)) AS `prepared_data`
13     WHERE  `empty_key` < (SELECT MAX(`sb_id`)
14                           FROM   `subscriptions`)
```

The main section of the query on lines 2-4 searches for missing values of the primary key, following right after the really existing ones.

An additional **`UNION`** section in lines 8-12 checks for free primary key values at the beginning of the sequence (starting with 1).

In line 1 the minimum value is chosen from the whole set of found ones.

Finally, on lines 13-14 a condition is applied to ensure that the detected free value does not exceed the real maximum value of the primary key.

Back to the stored procedure code.

Lines 30-32 form the query text that finds the maximum value of the primary key in the table being processed, this value will be changed to the first free value at each step of the cycle.

In lines 36-37, based on the text representation of SQL queries, in which the names of the processed table and its primary key are substituted, the executable expressions are created.

Lines 39-59 present a loop that runs as long as at least one free value of the primary key exists:

- at line 40 the main query is executed, it searches for the first (minimum) free value of the primary key and places that value into the **`@empty_key_value variable`**;
- in lines 43-45 the obtained value is checked for being **`NULL`** (if the condition is met, there are no more free values, and the loop is exited);
- on line 47 the runs the query that puts the current maximum value of the primary key into the **`@max_key_value`** variable (we have to use an intermediate variable to get around the MySQL restriction on using the same table simultaneously in the **`UPDATE`** and **`SELECT`** parts of the query);
- lines 48-50 form the text representation of the query to update the value of the primary key;

- for this query, lines 53, 54, 55, respectively, form an executable expression, the query is executed, the executable expression is freed (because the next step of the cycle will be different);
- the counter of updated values of the primary key is incremented in line 57;
- on line 58 the next iteration of the loop is activated.

After the loop is complete, we only have to free the previously prepared executable expressions, which is what happens in lines 61-62.

To execute the stored procedure and find out how many values of the primary key were changed, we can use the following queries (in the first case the value 9 will be returned, in the second the value 0 will be returned, because there are no free values of the primary key in the **books** table).

MySQL — Solution 5.2.1.a (code to execute the stored procedure and get the result of its work)

```
1    CALL COMPACT_KEYS ('subscriptions', 'sb_id', @keys_changed);
2    SELECT @keys_changed;
3
4    CALL COMPACT_KEYS ('books', 'b_id', @keys_changed);
5    SELECT @keys_changed;
```

If we examine this stored procedure step by step (there are nine steps for the **subscriptions** table), we get the following situation:

	Initial	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9
Primary key values	2	1	1	1	1	1	1	1	1	1
	3	2	2	2	2	2	2	2	2	2
	42	3	3	3	3	3	3	3	3	3
	57	42	4	4	4	4	4	4	4	4
	61	57	42	5	5	5	5	5	5	5
	62	61	57	42	6	6	6	6	6	6
	86	62	61	57	42	7	7	7	7	7
	91	86	62	61	57	42	8	8	8	8
	95	91	86	62	61	57	42	9	9	9
	99	95	91	86	62	61	57	42	10	10
	100	99	95	91	86	62	61	57	42	11
Free	**1**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**NULL**
Upd.	**100**	**99**	**95**	**91**	**86**	**62**	**61**	**57**	**42**	**11**

To further study the logic of the presented solution, it is recommended to create the considered stored procedure without removing the debug output, and trace on its basis the logic of operation and change of variables values inside the procedure.

This completes the MySQL solution.

Let's move on to MS SQL Server. The overall solution logic will be very similar to that just discussed for MySQL, except for one insurmountable problem: MS SQL Server does not allow to update **IDENTITY** fields of a table (enabling **IDENTITY_INSERT** only allows us to insert values into such fields, but not to update them).

We could bypass this restriction by deleting the old row of the table and inserting the new one (with the substituted value of the primary key), but such a solution would have fatal consequences if the modified primary key is referenced by foreign keys of other tables.

There is no simple general solution to disable and re-enable the **IDENTITY** property of a column using SQL queries. The only more or less available option is to disable and re-enable this property through the MS SQL Server Management Studio GUI.

Thus, in our stored procedure we will check if the field to be processed is an **IDENTITY** field and prohibit the operation if it is (lines 14-22 of the stored procedure code).

MS SQL | Solution 5.2.1.a (the procedure code)

```
CREATE PROCEDURE COMPACT_KEYS
                 @table_name NVARCHAR(150),
                 @pk_name NVARCHAR(150),
                 @keys_changed INT OUTPUT
WITH EXECUTE AS OWNER
AS
 DECLARE @empty_key_query NVARCHAR(1000) = '';
 DECLARE @max_key_query NVARCHAR(1000) = '';
 DECLARE @empty_key_value INT = NULL;
 DECLARE @max_key_value INT = NULL;
 DECLARE @update_key_query NVARCHAR(1000) = '';
 DECLARE @error_message NVARCHAR(1000) = '';

 IF (COLUMNPROPERTY(OBJECT_ID(@table_name), @pk_name, 'IsIdentity') = 1)
  BEGIN
   SET @keys_changed = -1;
   SET @error_message = CONCAT('Remove identity property for column [',
                               @pk_name,'] of table [', @table_name,
                               '] via MS SQL Server Management Studio.');
   RAISERROR (@error_message, 16, 1);
   RETURN -1;
  END;

 SET @keys_changed = 0;

 PRINT(CONCAT('Point 1. @table_name = ', @table_name, ', @pk_name = ',
        @pk_name, ', @keys_changed = ', ISNULL(@keys_changed, 'NULL')));

 SET @empty_key_query =
 CONCAT('SET @empty_k_v = (SELECT MIN([empty_key]) AS [empty_key]
        FROM   (SELECT [left].[', @pk_name, '] + 1 AS [empty_key]
                FROM   [', @table_name, '] AS [left]
                       LEFT OUTER JOIN [', @table_name, '] AS [right]
                                    ON [left].[', @pk_name,
                                       '] + 1 = [right].[', @pk_name, ']
         WHERE  [right].[', @pk_name, '] IS NULL
         UNION
         SELECT 1 AS [empty_key]
         FROM   [', @table_name, ']
         WHERE  NOT EXISTS(SELECT [', @pk_name, ']
                           FROM   [', @table_name, ']
                           WHERE  [', @pk_name, '] = 1)
        ) AS [prepared_data]
        WHERE  [empty_key] < (SELECT MAX([', @pk_name, '])
                              FROM   [', @table_name, ']))');

 SET @max_key_query =
     CONCAT('SET @max_k_v = (SELECT MAX([', @pk_name, ']) FROM [',
            @table_name, '])');

 PRINT(CONCAT('Point 2. @empty_key_query = ', @empty_key_query,
              CHAR(13), CHAR(10), '@max_key_query = ', @max_key_query));
WHILE (1 = 1)
  BEGIN
   EXECUTE sp_executesql @empty_key_query,
                         N'@empty_k_v INT OUT',
                         @empty_key_value OUTPUT;

   IF (@empty_key_value IS NULL)
    BREAK;

   EXECUTE sp_executesql @max_key_query,
                         N'@max_k_v INT OUT',
                         @max_key_value OUTPUT;

```

MS SQL Solution 5.2.1.a (the procedure code) (continued)

```
66      SET @update_key_query =
67          CONCAT('UPDATE [', @table_name, '] SET [', @pk_name,
68                 '] = ', @empty_key_value, ' WHERE [', @pk_name, '] = ',
69                 @max_key_value);
70
71      PRINT(CONCAT('Point 3. @update_key_query = ', @update_key_query));
72
73      EXECUTE sp_executesql @update_key_query;
74
75      SET @keys_changed = @keys_changed + 1;
76
77    END;
78  GO
```

SQL queries to determine the first free value of the primary key, the maximum value of the primary key and update the value of the primary key are similar to the solution for MySQL.

The slight difference is how to get the result of a dynamic query into a variable: **`SET … = (SELECT ...)`** is used instead of **`SELECT ... INTO ...`**, and when executing dynamic SQL with **`sp_executesql`** additional parameters are passed to place the query result in the specified variable (lines 55-57, 62-64).

Since MS SQL Server does not support **`DO ... WHILE`** loops, we have to use the infinite **`WHILE`** loop (lines 53-77), inside which we will check the exit condition and (if it is satisfied) forcibly terminate the loop (lines 59-60).

The last difference from MySQL is in the way the debugging information is output: in MS SQL Server we can use the **`PRINT`** clause (lines 26-27, 51-52, 71).

We can check the functionality of the obtained solution with the following queries.

MS SQL Solution 5.2.1.a (code to execute the stored procedure and get the result of its work)

```
1   DECLARE @res INT;
2   EXECUTE COMPACT_KEYS 'subscriptions', 'sb_id', @res OUTPUT;
3   SELECT @res;
4   GO
5
6   DECLARE @res INT;
7   EXECUTE COMPACT_KEYS 'books', 'b_quantity', @res OUTPUT;
8   SELECT @res;
9   GO
10
11  SELECT * FROM [books] ORDER BY [b_quantity];
```

In the first case we will get an error message asking to remove **`IDENTITY`** property from **`sb_id`** field of the **`subscriptions`** table; in the second case the procedure will run (yes, removing “free values” from the field storing the number of books does not make any sense, but it works fine to check if the stored procedure works).

This completes the solution for MS SQL Server.

Let’s move on to Oracle. There are no problems typical for MS SQL Server (it’s not even necessary to disable triggers, which provide auto-increment of primary key during insertion, because they are **`INSERT`** triggers, and we will perform **`UPDATE`**)).

Oracle | Solution 5.2.1.a (the procedure code)

```
1    CREATE PROCEDURE COMPACT_KEYS (table_name IN VARCHAR, pk_name IN VARCHAR,
2                                   keys_changed OUT NUMBER) AS
3     empty_key_query VARCHAR(1000) := '';
4     max_key_query VARCHAR(1000) := '';
5     empty_key_value NUMBER := NULL;
6     max_key_value NUMBER := NULL;
7     update_key_query VARCHAR(1000) := '';
8    BEGIN
9     keys_changed := 0;
10
11    DBMS_OUTPUT.PUT_LINE('Point 1. table_name = ' || table_name ||
12    ' || pk_name = ' || pk_name || ', keys_changed = ' || keys_changed);
13
14    empty_key_query :=
15    'SELECT MIN("empty_key") AS "empty_key"
16     FROM   (SELECT "left"."' || pk_name || '" + 1 AS "empty_key"
17             FROM   "' || table_name || '" "left"
18                    LEFT OUTER JOIN "' || table_name || '" "right"
19                                 ON "left"."' || pk_name ||
20                                 '" + 1 = "right"."' || pk_name || '"
21             WHERE  "right"."' || pk_name || '" IS NULL
22             UNION
23             SELECT 1 AS "empty_key"
24             FROM   "' || table_name || '"
25             WHERE  NOT EXISTS(SELECT "' || pk_name || '"
26                               FROM   "' || table_name || '"
27                               WHERE  "' || pk_name ||
28                               '" = 1)) "prepared_data"
29     WHERE  "empty_key" < (SELECT MAX("' || pk_name || '")
30                           FROM   "' || table_name || '")';
31
32    max_key_query :=
33    'SELECT MAX("' || pk_name || '") FROM "' || table_name || '"';
34
35    DBMS_OUTPUT.PUT_LINE('Point 2. empty_key_query = ' || empty_key_query ||
36    CHR(13) || CHR(10) || ' max_key_query = ' || max_key_query);
37
38    LOOP
39     EXECUTE IMMEDIATE empty_key_query INTO empty_key_value;
40     EXIT WHEN empty_key_value IS NULL;
41     EXECUTE IMMEDIATE max_key_query INTO max_key_value;
42     update_key_query :=
43         'UPDATE "' || table_name || '" SET "' || pk_name ||
44         '" = ' || TO_CHAR(empty_key_value) || ' WHERE "' || pk_name ||
45         '" = ' || TO_CHAR(max_key_value);
46     DBMS_OUTPUT.PUT_LINE('Point 3. update_key_query = ' || update_key_query);
47     EXECUTE IMMEDIATE update_key_query;
48     keys_changed := keys_changed + 1;
49   END LOOP;
50   END;
51   /
```

It turns out that the only two differences between the Oracle and MySQL solutions are in the way debugging information is output (lines 11-12, 35-36, 46) and in the syntax describing the logic of loop exit (line 40).

We can check the functionality of the obtained solution with the following queries (don't forget to enable the display of messages received from the server by executing **`SET SERVEROUTPUT ON`**).

Oracle — Solution 5.2.1.a (code to execute the stored procedure and get the result of its work)

```
1    DECLARE
2     keys_changed_in_table NUMBER;
3    BEGIN
4     COMPACT_KEYS('books', 'b_id', keys_changed_in_table);
5     DBMS_OUTPUT.PUT_LINE('Keys changed: ' || keys_changed_in_table);
6
7     COMPACT_KEYS('subscriptions', 'sb_id', keys_changed_in_table);
8     DBMS_OUTPUT.PUT_LINE('Keys changed: ' || keys_changed_in_table);
9    END;
```

This completes the solution of this problem.

Solution 5.2.1.b{390}.

In the solution{390} of problem 5.2.1.a{390} we have already considered the logic of forming and executing dynamic SQL queries in stored procedures. The difference in solving this problem will be that the result of the stored procedure will not be a change in the database and return the number of edits, but the return of a table.

Solution for MySQL looks like this.

MySQL — Solution 5.2.1.b (the procedure code)

```
1    DELIMITER $$
2    CREATE PROCEDURE SHOW_TABLE_OBJECTS (IN table_name VARCHAR(150))
3    BEGIN
4     SET @query_text = '
5     SELECT \'foreign_key\'   AS `object_type`,
6            `constraint_name` AS `object_name`
7     FROM   `information_schema`.`table_constraints`
8     WHERE  `table_schema` = DATABASE()
9            AND `table_name` = \'_FP_TABLE_NAME_PLACEHOLDER_\'
10           AND `constraint_type` = \'FOREIGN KEY\'
11    UNION
12    SELECT \'trigger\'    AS `object_type`,
13           `trigger_name` AS `object_name`
14    FROM   `information_schema`.`triggers`
15    WHERE  `event_object_schema` = DATABASE()
16           AND `event_object_table` = \'_FP_TABLE_NAME_PLACEHOLDER_\'
17    UNION
18    SELECT \'view\'     AS `object_type`,
19           `table_name` AS `object_name`
20    FROM   `information_schema`.`views`
21    WHERE  `table_schema` = DATABASE()
22           AND `view_definition` LIKE \'%`_FP_TABLE_NAME_PLACEHOLDER_`%\'';
23    SET @query_text = REPLACE(@query_text,
24                              '_FP_TABLE_NAME_PLACEHOLDER_', table_name);
25
26    PREPARE query_stmt FROM @query_text;
27    EXECUTE query_stmt;
28    DEALLOCATE PREPARE query_stmt;
29   END;
30   $$
31   DELIMITER ;
```

Here (for colorfulness) we get the text of the resulting query not using the `CONCAT` function, but by replacing the `_FP_TABLE_NAME_PLACEHOLDER_` placeholder with the real table name in the previously prepared full query text.

The logic of obtaining the list of objects is completely trivial for foreign keys and triggers (see query text in lines 5-16), and only for views we need to analyze their source code to detect there the name of the table obtained as a parameter of our procedure (because views are not associated directly with tables, but are independent objects).

We can check the functionality of the obtained solution with the following query.

MySQL Solution 5.2.1.b (code to execute the stored procedure and get the result of its work)

```
CALL SHOW_TABLE_OBJECTS('subscriptions')
```

This completes the MySQL solution.

Let's move on to MS SQL Server. Here the logic of the solution is exactly the same as for MySQL, except for the fact that information about triggers has to be extracted from **`[sys].[triggers]`** as an analogue of **`` `information_schema`.`triggers` ``**.

MS SQL Solution 5.2.1.b (the procedure code)

```
CREATE PROCEDURE SHOW_TABLE_OBJECTS
                 @table_name NVARCHAR(150)
WITH EXECUTE AS OWNER
AS
 DECLARE @query_text NVARCHAR(1000) = '';
 SET @query_text =
 'SELECT ''foreign_key''   AS [object_type],
         [constraint_name] AS [object_name]
  FROM   [information_schema].[table_constraints]
  WHERE  [table_catalog] = DB_NAME()
         AND [table_name] = ''_FP_TABLE_NAME_PLACEHOLDER_''
         AND [constraint_type] = ''FOREIGN KEY''
  UNION
  SELECT ''trigger'' AS [object_type],
         [name]      AS [object_name]
  FROM   [sys].[triggers]
  WHERE  OBJECT_NAME([parent_id]) = ''_FP_TABLE_NAME_PLACEHOLDER_''
  UNION
  SELECT ''view''     AS [object_type],
         [table_name] AS [object_name]
  FROM   [information_schema].[views]
  WHERE  [table_catalog] = DB_NAME()
         AND [view_definition] LIKE ''%[_FP_TABLE_NAME_PLACEHOLDER_]%''';

 SET @query_text = REPLACE(@query_text, '_FP_TABLE_NAME_PLACEHOLDER_',
                                        @table_name);

 EXECUTE sp_executesql @query_text;
GO
```

We can check the functionality of the obtained solution with the following query.

MS SQL Solution 5.2.1.b (code to execute the stored procedure and get the result of its work)

```
EXECUTE SHOW_TABLE_OBJECTS 'subscriptions';
```

This completes the solution for MS SQL Server.

Let's move on to Oracle. And here is where the radical differences will appear. The solution will still be based on the same query that we used for MySQL and MS SQL Server, but Oracle has one very unpleasant problem, which complicates the solution many times over.

The text of the view (in which we are looking for a table reference) is stored in a field of **`LONG`** type (this is not a "long integer", it is an obsolete but still sometimes used text data type[23]), and data of this type can neither be used in expressions of **`LIKE`** type, nor converted to another type (e.g., **`VARCHAR2`**) in a simple way.

The only more or less adequate way to extract **`LONG`** data with conversion to **`VARCHAR2`** is to create a stored function. There is a general solution[24], but for simplicity we will implement an option that is bound to a certain data source.

[23] https://docs.oracle.com/cd/E11882_01/appdev.112/e25519/datatypes.htm#LNPLS346

[24] https://asktom.oracle.com/pls/apex/f?p=100:11:0::NO::P11_QUESTION_ID:839298816582

In the code below, we create a stored function that returns a table (see the solution{370} of problem 5.1.1.b{367} for the ways of creating and using such functions). The key idea here is that the `TEXT` field of `all_views_row` object is declared as `VARCHAR2`, and that is the type of data that will be in the output table. And with `VARCHAR2` data we can already perform comparison operations.

In this particular case, the function first prepares the complete data set and then returns it as a generated table, and the `PIPLINED` solution will be presented next.

Oracle — Solution 5.2.1.b (function code to convert `LONG` to `VARCHAR2`)

```
1    CREATE OR REPLACE TYPE "all_views_row" AS OBJECT
2    (
3       "VIEW_NAME" VARCHAR2(500),
4       "TEXT"      VARCHAR2(32767)
5    );
6    /
7    CREATE TYPE "all_views_table" IS TABLE OF "all_views_row";
8    /
9
10   CREATE OR REPLACE FUNCTION ALL_VIEWS_VARCHAR2
11   RETURN "all_views_table"
12   AS
13    result_table  "all_views_table" := "all_views_table"();
14    CURSOR all_views_table_cursor IS
15     SELECT VIEW_NAME,
16            TEXT
17     FROM   ALL_VIEWS
18     WHERE  OWNER = USER;
19   BEGIN
20     FOR one_row IN all_views_table_cursor
21     LOOP
22      result_table.extend;
23      result_table(result_table.last) :=
24                      "all_views_row"(one_row."VIEW_NAME", one_row."TEXT");
25     END LOOP;
26     RETURN result_table;
27   END;
28   /
```

Now that the problem of applying the `LIKE` expression to the text of the view is solved, all that is left is to create a stored procedure similar to the solutions for MySQL and MS SQL Server.

Oracle Solution 5.2.1.b (the procedure code)

```
CREATE OR REPLACE PROCEDURE SHOW_TABLE_OBJECTS(table_name IN VARCHAR2,
                                               final_rc OUT SYS_REFCURSOR)
IS
 query_text VARCHAR2(1000);
BEGIN
 query_text := '
  SELECT ''foreign_key'' AS "object_type",
         CONSTRAINT_NAME AS "object_name"
  FROM ALL_CONSTRAINTS
  WHERE OWNER = USER
  AND TABLE_NAME = ''_FP_TABLE_NAME_PLACEHOLDER_''
  AND CONSTRAINT_TYPE = ''R''
  UNION
  SELECT ''trigger''  AS "object_type",
         TRIGGER_NAME AS "object_name"
  FROM ALL_TRIGGERS
  WHERE OWNER = USER
  AND TABLE_NAME = ''_FP_TABLE_NAME_PLACEHOLDER_''
  UNION
  SELECT ''view''    AS "object_type",
         "VIEW_NAME" AS "object_name"
  FROM TABLE(ALL_VIEWS_VARCHAR2)
  WHERE "TEXT" LIKE ''%"_FP_TABLE_NAME_PLACEHOLDER_"%''';

  query_text := REPLACE(query_text, '_FP_TABLE_NAME_PLACEHOLDER_',
                                    table_name);

 OPEN final_rc FOR query_text;
END;
/
```

But one inconvenience remains: to get the result of such a procedure, we have to use a rather non-trivial code:

Oracle Solution 5.2.1.b (code to execute the stored procedure and get the result of its work)

```
DECLARE
  rc SYS_REFCURSOR;
  object_type VARCHAR2(500);
  object_name VARCHAR2(500);
BEGIN
  SHOW_TABLE_OBJECTS('subscriptions', rc);

  LOOP
    FETCH rc INTO object_type, object_name;
    EXIT WHEN rc%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(object_type || ' | ' || object_name);
  END LOOP;
  CLOSE rc;
END;
```

And even with this non-trivial code, we get the result as text, while we would like to get a full-fledged table. This is possible, but we have to abandon the stored procedure and implement a stored function.

A bit above we created such a function to convert the data type of the field in which the view text is stored. Now we will improve it and get a complete solution in a function that fully meets the conditions of the current problem.

Along the way we will change the function type to `PIPELINED`, which will reduce the load on RAM and slightly increase performance.

Oracle | Solution 5.2.1.b (alternative solution in the form of a single stored function)

```
CREATE OR REPLACE TYPE "show_table_objects_row" AS OBJECT
(
   "field_a"  VARCHAR2(500),
   "field_b"  VARCHAR2(32767)
);
/
CREATE TYPE "show_table_objects_table"
IS TABLE OF "show_table_objects_row";
/

CREATE OR REPLACE FUNCTION SHOW_TABLE_OBJECTS_FNC(table_name IN VARCHAR2)
RETURN "show_table_objects_table" PIPELINED
AS
 TYPE type_rc IS REF CURSOR;
 rc type_rc;
 field_a    VARCHAR2(500);
 field_b    VARCHAR2(32767);
 query_text VARCHAR2(1000);
BEGIN
 query_text := '
  SELECT ''foreign_key'' AS "object_type",
         CONSTRAINT_NAME AS "object_name"
  FROM ALL_CONSTRAINTS
  WHERE OWNER = USER
  AND TABLE_NAME = ''_FP_TABLE_NAME_PLACEHOLDER_''
  AND CONSTRAINT_TYPE = ''R''
  UNION
  SELECT ''trigger''  AS "object_type",
         TRIGGER_NAME AS "object_name"
  FROM ALL_TRIGGERS
  WHERE OWNER = USER
  AND TABLE_NAME = ''_FP_TABLE_NAME_PLACEHOLDER_''';
  query_text := REPLACE(query_text, '_FP_TABLE_NAME_PLACEHOLDER_',
                                    table_name);

  OPEN rc FOR query_text;
  LOOP
   FETCH rc INTO field_a, field_b;
   EXIT WHEN rc%NOTFOUND;
   PIPE ROW("show_table_objects_row"(field_a, field_b));
  END LOOP;
  CLOSE rc;

 query_text := '
  SELECT VIEW_NAME,
         TEXT
  FROM   ALL_VIEWS
  WHERE  OWNER = USER';

  OPEN rc FOR query_text;
  LOOP
   FETCH rc INTO field_a, field_b;
   EXIT WHEN rc%NOTFOUND;
   IF (INSTR(field_b, '"' || table_name || '"') > 0)
    THEN
     PIPE ROW("show_table_objects_row"('view', field_a));
    END IF;
  END LOOP;
  CLOSE rc;

  RETURN;
END;
```

The key idea of this function is to output the data received separately from two different queries.

The first query (lines 20-42) simply fetches data about foreign keys and triggers without any special complications and nuances: a string constant (`'foreign_key'`, `'trigger'`) is placed in the `field_a` variable, the name of corresponding foreign key or trigger is placed in the `field_b` variable. Then these variables are used to initialize `show_table_objects_row` object fields.

In the second query (lines 44-59), using the same variables and the same object as in the first query, we do the following:

- first the name and text of a view are placed in the variables `field_a` and `field_b`, respectively (line 52);
- then the value of the `field_b` variable is used to check the fact that the view text contains the name of the analyzed table (line 54), and we do not need and do not use this value of the `field_b` variable anywhere else;
- finally (on line 56) we use the text constant `'view'` and the view name stored in the `field_a` variable to initialize the fields of the `show_table_objects_row` object.

Now we can get the result we need in the form of a table.

Oracle — Solution 5.2.1.b (code to execute the stored function and get the result as a table)

```
1    SELECT * FROM TABLE(SHOW_TABLE_OBJECTS_FNC('subscriptions'));
```

This completes the solution of this problem.

Task 5.2.1.TSK.A: create a stored procedure that updates all fields of `DATE` type (if any) of all records in the specified table to the current date.

Task 5.2.1.TSK.B: create a stored procedure that generates a list of child tables (and their foreign keys), depending on the parent table specified in the parameter.

5.2.2. Example 39: Using Stored Procedures for Performance Optimization

Problem 5.2.2.a[403]: create a stored procedure that runs on schedule every hour and updates data in the aggregating **books_statistics** table (see problem 3.1.2.a[229]).

Problem 5.2.2.b[408]: create a stored procedure that runs on schedule every day and optimizes (defragments, compactifies) all database tables.

Expected result 5.2.2.a.

At the beginning of each hour the created stored procedure is started. The data in the **books_statistics** table is brought up to date.

Expected result 5.2.2.b.

At the beginning of each day the created stored procedure is started. All database tables are brought into an optimized state.

Solution 5.2.2.a[403].

In the MySQL solution, the main query (lines 14-29) that updates data in the **books_statistics** table is based on the solution[230] of problem 3.1.2.a[229].

However, for reliability reasons, we will check the existence of the **books_statistics** table, which we are going to update. This check is implemented in lines 5-13. If the table does not exist, we terminate the stored procedure by returning the corresponding error message.

MySQL | Solution 5.2.2.a (the procedure code)

```
1    DELIMITER $$
2    CREATE PROCEDURE UPDATE_BOOKS_STATISTICS()
3    BEGIN
4
5     IF (NOT EXISTS(SELECT *
6                    FROM `information_schema`.`tables`
7                    WHERE `table_schema` = DATABASE()
8                          AND `table_name` = 'books_statistics'))
9      THEN
10      SIGNAL SQLSTATE '45001'
11      SET MESSAGE_TEXT = 'The `books_statistics` table is missing.',
12      MYSQL_ERRNO = 1001;
13     END IF;
```

MySQL Solution 5.2.2.a (the procedure code) (continued)

```
14    UPDATE `books_statistics`
15    JOIN
16    (SELECT IFNULL(`total`, 0) AS `total`,
17            IFNULL(`given`, 0) AS `given`,
18            IFNULL(`total` - `given`, 0) AS `rest`
19     FROM   (SELECT (SELECT SUM(`b_quantity`)
20                     FROM   `books`)               AS `total`,
21                    (SELECT COUNT(`sb_book`)
22                     FROM   `subscriptions`
23                     WHERE  `sb_is_active` = 'Y') AS `given`)
24            AS `prepared_data`
25    ) AS `src`
26    SET
27     `books_statistics`.`total` = `src`.`total`,
28     `books_statistics`.`given` = `src`.`given`,
29     `books_statistics`.`rest` = `src`.`rest`;
30  END;
31  $$
32  DELIMITER ;
```

Let's check its functionality (in advance, we can run a separate query to update the data in the **books_statistics** table and set all values to zero).

MySQL Solution 5.2.2.a (running the functionality check)

```
1   CALL UPDATE_BOOKS_STATISTICS;
```

Setup the resulting stored procedure to run according to the schedule is as follows. Preliminarily in line 1 we enable the MySQL task scheduler (in order to keep it running even after MySQL restart it is necessary to add the **event_scheduler = on** line to the MySQL settings **my.ini** file).

MySQL Solution 5.2.2.a (setting the scheduled start)

```
1   SET GLOBAL event_scheduler = ON;
2
3   CREATE EVENT `update_books_statistics_hourly`
4   ON SCHEDULE
5   EVERY 1 HOUR
6   STARTS DATE(NOW()) + INTERVAL (HOUR(NOW())+1) HOUR + INTERVAL 1 MINUTE
7   ON COMPLETION PRESERVE
8   DO
9   CALL UPDATE_BOOKS_STATISTICS;
```

To make sure that the corresponding task is added to the scheduler, run the following query:

MySQL Solution 5.2.2.a (checking the schedule)

```
1   SELECT * FROM `information_schema`.`events`
```

This completes the MySQL solution.

Let's move to MS SQL Server. The work logic of the stored procedure here is fully equivalent to the MySQL solution: we check if the **books_statistics** table exists in lines 3-10 (and terminate the stored procedure if it does not exist), and in lines 12-30 we execute a query that updates the data.

So far, everything looks quite simple and trivial, but adding a task to the scheduler in this DBMS is implemented in a much more complex way.

MS SQL Solution 5.2.2.a (the procedure code)

```
1    CREATE PROCEDURE UPDATE_BOOKS_STATISTICS
2    AS
3     IF (NOT EXISTS(SELECT *
4                    FROM [information_schema].[tables]
5                    WHERE [table_catalog] = DB_NAME()
6                      AND [table_name] = 'books_statistics'))
7      BEGIN
8       RAISERROR ('The [books_statistics] table is missing.', 16, 1);
9       RETURN;
10     END;
11
12    UPDATE [books_statistics]
13    SET
14     [books_statistics].[total] = [src].[total],
15     [books_statistics].[given] = [src].[given],
16     [books_statistics].[rest] = [src].[rest]
17     FROM [books_statistics]
18    JOIN
19    (SELECT ISNULL([total], 0) AS [total],
20            ISNULL([given], 0) AS [given],
21            ISNULL([total] - [given], 0) AS [rest]
22     FROM    (SELECT (SELECT SUM([b_quantity])
23                      FROM   [books])              AS [total],
24                     (SELECT COUNT([sb_book])
25                      FROM   [subscriptions]
26                      WHERE  [sb_is_active] = 'Y') AS [given])
27            AS [prepared_data]
28    ) AS [src]
29    ON 1=1;
30   GO
```

Let's check its functionality (in advance, we can run a separate query to update the data in the **books_statistics** table and set all values to zero).

MS SQL Solution 5.2.2.a (running the functionality check)

```
1    EXECUTE UPDATE_BOOKS_STATISTICS
```

Setting a scheduled execution of a stored procedure in MS SQL Server not only looks much more complicated but will have no effect if you use Express Edition of this DBMS: this version lacks SQL Server Agent component, which is responsible for scheduled tasks execution.

There is a lot written in the official documentation for each of the following commands (there is a special link to the corresponding documentation section before each command), but if we express the algorithm in simple words, it is as follows.

We need to:

- Create a task.
- Create a task step describing the start of our stored procedure.
- Create a schedule in which we specify the required periodicity of execution.
- Attach the previously created task to the newly created schedule.
- Pass the created task for processing to SQL Server Agent.

MS SQL Solution 5.2.2.a (setting the scheduled start)

```
1    USE msdb ;
2    GO
3    -- https://msdn.microsoft.com/en-us/library/ms182079.aspx
4    EXEC dbo.sp_add_job
5        @job_name = N'Hourly [books_statistics] update';
6    GO
7    -- https://msdn.microsoft.com/en-us/library/ms187358.aspx
8    EXEC sp_add_jobstep
9        @job_name = N'Hourly [books_statistics] update',
10       @step_name = N'Execute UPDATE_BOOKS_STATISTICS stored procedure',
11       @subsystem = N'TSQL',
12       @command = N'EXECUTE UPDATE_BOOKS_STATISTICS',
13       @database_name = N'library_ex_2015_mod';
14   GO
15   -- https://msdn.microsoft.com/en-us/library/ms187320.aspx
16   EXEC dbo.sp_add_schedule
17       @schedule_name = N'UpdateBooksStatistics',
18       @freq_type = 4,
19       @freq_interval = 4,
20       @freq_subday_type = 8,
21       @freq_subday_interval = 1,
22       @active_start_time = 000100 ;
23   USE msdb ;
24   GO
25   -- https://msdn.microsoft.com/en-us/library/ms186766.aspx
26   EXEC sp_attach_schedule
27      @job_name = N'Hourly [books_statistics] update',
28      @schedule_name = N'UpdateBooksStatistics';
29   GO
30   -- https://msdn.microsoft.com/en-us/library/ms178625.aspx
31   EXEC dbo.sp_add_jobserver
32       @job_name = N'Hourly [books_statistics] update';
33   GO
```

To make sure that the corresponding task is added to the scheduler, run the following query (it will show the list of tasks even in MS SQL Server Express Edition):

MS SQL Solution 5.2.2.a (checking the schedule)

```
1    SELECT * FROM msdb.dbo.sysschedules
```

This discussion[25] presents a very good ready-made SQL-script for getting information about all scheduled tasks in MS SQL Server in a convenient form.

This completes the solution for MS SQL Server.

Let's move on to Oracle. The work logic of the stored procedure here is fully equivalent to the MySQL and MS SQL Server solutions: we check if the **`books_statistics`** table exists in lines 5-15 (and exit the stored procedure, if the table does not exist), and in lines 17-27 we execute a query that updates data.

[25] http://www.sqlservercentral.com/Forums/Topic410557-116-1.aspx

Oracle Solution 5.2.2.a (the procedure code)

```
CREATE OR REPLACE PROCEDURE UPDATE_BOOKS_STATISTICS
AS
 rows_count NUMBER;
BEGIN
 SELECT COUNT(1) INTO rows_count
 FROM ALL_TABLES
 WHERE OWNER = USER
 AND TABLE_NAME = 'books_statistics';

 IF (rows_count = 0)
  THEN
   RAISE_APPLICATION_ERROR(-20001,
                           'The "books_statistics" table is missing.');
   RETURN;
  END IF;

 UPDATE "books_statistics"
   SET ("total", "given", "rest") =
   (SELECT NVL("total", 0) AS "total",
           NVL("given", 0) AS "given",
           NVL("total" - "given", 0) AS "rest"
    FROM   (SELECT (SELECT SUM("b_quantity")
                    FROM   "books")              AS "total",
                   (SELECT COUNT("sb_book")
                    FROM   "subscriptions"
                    WHERE  "sb_is_active" = 'Y') AS "given"
           FROM dual) "prepared_data");
END;
/
```

Let's check its functionality (in advance, we can run a separate query to update the data in the table books_statistics and set all values to zero).

Oracle Solution 5.2.2.a (running the functionality check)

```
EXECUTE UPDATE_BOOKS_STATISTICS
```

Setting the schedule to run the resulting stored procedure is as follows.

Oracle Solution 5.2.2.a (setting the scheduled start)

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB (
    job_name            =>  'hourly_update_books_statistics',
    job_type            =>  'STORED_PROCEDURE',
    job_action          =>  'UPDATE_BOOKS_STATISTICS',
    start_date          =>  '01-APR-16 1.00.00 AM',
    repeat_interval     =>  'FREQ=HOURLY;INTERVAL=1',
    auto_drop           =>  FALSE,
    enabled             =>  TRUE);
END;
```

To make sure that the corresponding task is added to the scheduler, run the following query:

Oracle Solution 5.2.2.a (checking the schedule)

```
SELECT * FROM ALL_SCHEDULER_JOBS WHERE OWNER=USER
```

This completes the solution of this problem.

Solution 5.2.2.b{403}.

The solution of this problem is both very simple and very complex, because we need to get a list of database tables and… do something with them.

The question of optimizing the performance of databases and DBMSes deserves a separate book, so here we will take the path of least resistance and assume that:

- For MySQL it will be enough to perform `OPTIMIZE` process for all tables.
- For MS SQL Server it will be enough to perform `REORGANIZE` or `REBUILD` processes for all clustered indexes (which leads to optimization of the corresponding tables on which the indexes are built).
- For Oracle it will be enough to perform `SHRINK SPACE COMPACT CASCADE` process for all tables.

We would like to emphasize once again that the solution of this problem is for demonstration purposes only and should **not** be regarded as a recommendation for general performance optimization. In some cases, the implementation of the actions shown below may reduce the database performance, so be sure to carefully study the official documentation on the relevant DBMS and professional recommendations on performance optimization in this or that real situation.

Traditionally, we start with MySQL.

MySQL | Solution 5.2.2.b (the procedure code)

```
1    DELIMITER $$
2    CREATE PROCEDURE OPTIMIZE_ALL_TABLES()
3    BEGIN
4     DECLARE done INT DEFAULT 0;
5     DECLARE tbl_name VARCHAR(200) DEFAULT '';
6     DECLARE all_tables_cursor CURSOR FOR
7      SELECT `table_name`
8      FROM   `information_schema`.`tables`
9      WHERE  `table_schema` = DATABASE()
10     AND    `table_type` = 'BASE TABLE';
11    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
12
13    OPEN all_tables_cursor;
14
15    tables_loop: LOOP
16     FETCH all_tables_cursor INTO tbl_name;
17     IF done THEN
18      LEAVE tables_loop;
19     END IF;
20
21     SET @table_opt_query = CONCAT('OPTIMIZE TABLE `', tbl_name, '`');
22     PREPARE table_opt_stmt FROM @table_opt_query;
23     EXECUTE table_opt_stmt;
24     DEALLOCATE PREPARE table_opt_stmt;
25
26    END LOOP tables_loop;
27
28    CLOSE all_tables_cursor;
29   END;
30   $$
31
32   DELIMITER ;
```

The query on lines 7-10 gets the list of tables in the current database (the part of the condition in line 10 allows us to distinguish tables from views), then:

- a cursor is opened for the obtained list of tables (line 13);
- a loop (lines 15-26) is executed for all cursor lines;
- in the loop body, a query is formed (line 21) and executed (lines 22-23), which implements the table optimization.

The resulting stored procedure can be run as follows.

MySQL Solution 5.2.2.b (running the functionality check)

```
1    CALL OPTIMIZE_ALL_TABLES
```

Now it remains to create an event triggered by the MySQL task scheduler according to a certain schedule. This is implemented by the following code.

MySQL Solution 5.2.2.b (setting the scheduled start)

```
1    SET GLOBAL event_scheduler = ON;
2
3    CREATE EVENT `optimize_all_tables_daily`
4    ON SCHEDULE
5    EVERY 1 DAY
6    STARTS DATE(NOW()) + INTERVAL 1 HOUR
7    ON COMPLETION PRESERVE
8    DO
9    CALL OPTIMIZE_ALL_TABLES;
```

To make sure that the event is added, you can use the following query.

MySQL Solution 5.2.2.b (checking the schedule)

```
1    SELECT * FROM `information_schema`.`events`
```

If we look at the selected fields of the result of such a query, we get the following picture. Here we also see the event that once an hour activates a stored procedure created while solving{403} the problem 5.2.2.a{403}.

EVENT_NAME	EVENT_TYPE	INTERVAL_VALUE	INTERVAL_FIELD	STARTS	ENDS	STATUS	ON_COMPLETION
update_books_statistics_hourly	RECURRING	1	HOUR	2016-04-27 14:01:00	NULL	ENABLED	PRESERVE
optimize_all_tables_daily	RECURRING	1	DAY	2016-04-27 01:00:00	NULL	ENABLED	PRESERVE

This completes the MySQL solution.

Let's move on to MS SQL Server. The concept of the solution below is based on this recommendation[26].

So, we will need to get a list of clustered indexes of the current database, find out the degree of fragmentation (`avg_fragmentation_in_percent`) and, depending on this value, perform a reorganization (`REORGANIZE`) or rebuilding (`REBUILD`) of the index.

To start with, let's create a query that provides information on all indexes, corresponding tables, fields, etc. Unfortunately, MS SQL Server does not provide a ready-made solution, so we have to do everything "manually".

[26] http://blog.sqlauthority.com/2010/01/12/sql-server-fragmentation-detect-fragmentation-and-eliminate-fragmentation/

MS SQL Solution 5.2.2.b (obtaining data for all indexes)

```
SELECT [tables].[name]                                  AS [table_name],
       [indexes].[name]                                 AS [index_name],
       [indexes].[type]                                 AS [index_type],
       [stats].[index_type_desc]                        AS [index_type_desc],
       [indexes].[object_id]                            AS [index_object_id],
       [columns].[name]                                 AS [column_name],
       [stats].[avg_fragmentation_in_percent]   AS [avg_fragm_perc],
       [stats].[avg_page_space_used_in_percent] AS [avg_space_perc]
FROM   sys.indexes AS [indexes]
       INNER JOIN sys.index_columns AS [index_columns]
               ON [indexes].[object_id] = [index_columns].[object_id]
                  AND [indexes].[index_id] = [index_columns].[index_id]
       INNER JOIN sys.columns AS [columns]
               ON [index_columns].[object_id] = [columns].[object_id]
                  AND [index_columns].[column_id] = [columns].[column_id]
       INNER JOIN sys.tables AS [tables]
               ON [indexes].[object_id] = [tables].[object_id]
       INNER JOIN sys.dm_db_index_physical_stats(DB_ID(DB_NAME()),
                                                 NULL, NULL, NULL,
                                                 'SAMPLED') AS [stats]
               ON [indexes].[object_id] = [stats].[object_id]
                        AND [indexes].[index_id] = [stats].[index_id]
ORDER  BY [tables].[name],
          [indexes].[name],
          [indexes].[index_id],
          [index_columns].[index_column_id]
```

As a result of such a query we will get the following data.

table_name	index_name	index_type	index_type_desc	index_object_id	column_name	avg_fragm_perc	avg_space_perc
authors	PK_authors	1	CLUSTERED INDEX	245575913	a_id	0	3.22461082283173
books	PK_books	1	CLUSTERED INDEX	277576027	b_id	0	5.72028663207314
genres	PK_genres	1	CLUSTERED INDEX	309576141	g_id	0	2.59451445515196
genres	UQ_genres_g_name	2	NONCLUSTERED INDEX	309576141	g_name	0	2.37212750185322
m2m_books_authors	PK_m2m_books_authors	1	CLUSTERED INDEX	341576255	b_id	0	1.86557944156165
m2m_books_authors	PK_m2m_books_authors	1	CLUSTERED INDEX	341576255	a_id	0	1.86557944156165
m2m_books_genres	PK_m2m_books_genres	1	CLUSTERED INDEX	373576369	b_id	0	2.28564368668149
m2m_books_genres	PK_m2m_books_genres	1	CLUSTERED INDEX	373576369	g_id	0	2.28564368668149
subscribers	PK_subscribers	1	CLUSTERED INDEX	405576483	s_id	0	1.95206325673338
subscriptions	PK_subscriptions	1	CLUSTERED INDEX	437576597	sb_id	0	5.16431924882629

This result is good because it is human-readable, but inside the stored procedure we will need only three fields: table name, index name, fragmentation percentage. Besides, composite cluster indexes (e.g., `PK_m2m_books_authors`) should be represented only once (not for each of the index fields, as it is now).

Let's simplify the query to leave only the necessary data.

MS SQL Solution 5.2.2.b (obtaining a concise data set for all indexes)

```
1    SELECT DISTINCT
2           [tables].[name]                              AS [table_name],
3           [indexes].[name]                             AS [index_name],
4           [stats].[avg_fragmentation_in_percent]       AS [avg_fragm_perc]
5    FROM   sys.indexes AS [indexes]
6           INNER JOIN sys.tables AS [tables]
7                   ON [indexes].[object_id] = [tables].[object_id]
8           INNER JOIN sys.dm_db_index_physical_stats(DB_ID(DB_NAME()),
9                                                     NULL, NULL, NULL,
10                                                    'SAMPLED') AS [stats]
11                  ON [indexes].[object_id] = [stats].[object_id]
12                                AND [indexes].[index_id] = [stats].[index_id]
13   WHERE  [indexes].[type] = 1
14   ORDER  BY [tables].[name],
15             [indexes].[name]
```

As a result of such a query we will get the following data.

table_name	index_name	avg_fragm_perc
authors	PK_authors	0
books	PK_books	0
genres	PK_genres	0
m2m_books_authors	PK_m2m_books_authors	0
m2m_books_genres	PK_m2m_books_genres	0
subscribers	PK_subscribers	0
subscriptions	PK_subscriptions	0

Now we will use the obtained query in the body of the stored procedure. Going through all rows returned by the cursor (the loop in lines 29-56), we will analyze the value of **`avg_fragm_perc`** and either perform one of two optimization actions or perform no action.

MS SQL Solution 5.2.2.b (the procedure code)

```
1    CREATE PROCEDURE OPTIMIZE_ALL_TABLES
2    AS
3    BEGIN
4     DECLARE @table_name NVARCHAR(200);
5     DECLARE @index_name NVARCHAR(200);
6     DECLARE @avg_fragm_perc DOUBLE PRECISION;
7     DECLARE @query_text NVARCHAR(2000);
8     DECLARE indexes_cursor CURSOR LOCAL FAST_FORWARD FOR
9      SELECT DISTINCT
10            [tables].[name]                              AS [table_name],
11            [indexes].[name]                             AS [index_name],
12            [stats].[avg_fragmentation_in_percent]       AS [avg_fragm_perc]
13     FROM   sys.indexes AS [indexes]
14            INNER JOIN sys.tables AS [tables]
15                    ON [indexes].[object_id] = [tables].[object_id]
16            INNER JOIN sys.dm_db_index_physical_stats(DB_ID(DB_NAME()),
17                                                      NULL, NULL, NULL,
18                                                      'SAMPLED') AS [stats]
19                    ON [indexes].[object_id] = [stats].[object_id]
20                                  AND [indexes].[index_id] = [stats].[index_id]
21     WHERE  [indexes].[type] = 1
22     ORDER  BY [tables].[name],
23               [indexes].[name];
24
25    OPEN indexes_cursor;
26    FETCH NEXT FROM indexes_cursor INTO @table_name,
27                                        @index_name,
28                                        @avg_fragm_perc;
```

MS SQL Solution 5.2.2.b (the procedure code) (continued)

```
29    WHILE @@FETCH_STATUS = 0
30      BEGIN
31       IF (@avg_fragm_perc >= 5.0) AND (@avg_fragm_perc <= 30.0)
32        BEGIN
33         SET @query_text = CONCAT('ALTER INDEX [', @index_name,
34                                  '] ON [', @table_name, '] REORGANIZE');
35         PRINT CONCAT('Index [', @index_name,'] on [', @table_name,
36                      '] will be REORGANIZED...');
37         EXECUTE sp_executesql @query_text;
38        END;
39       IF (@avg_fragm_perc > 30.0)
40        BEGIN
41         SET @query_text = CONCAT('ALTER INDEX [', @index_name,'] ON [',
42                                   @table_name, '] REBUILD');
43         PRINT CONCAT('Index [', @index_name,'] on [', @table_name,
44                      '] will be REBUILT...');
45         EXECUTE sp_executesql @query_text;
46        END;
47       IF (@avg_fragm_perc < 5.0)
48        BEGIN
49         PRINT CONCAT('Index [', @index_name,'] on [', @table_name,
50                      '] needs no optimization...');
51        END;
52
53       FETCH NEXT FROM indexes_cursor INTO @table_name,
54                                           @index_name,
55                                           @avg_fragm_perc;
56      END;
57     CLOSE indexes_cursor;
58     DEALLOCATE indexes_cursor;
59    END;
60    GO
```

We can run the resulting stored procedure as follows.

MS SQL Solution 5.2.2.b (running the functionality check)

```
1     EXECUTE OPTIMIZE_ALL_TABLES
```

The logic of setting up a scheduled task run was discussed in the solution{403} of problem 5.2.2.a{403}, so here we will simply give the code by which this operation is performed.

MS SQL Solution 5.2.2.b (setting the scheduled start)

```
1     USE msdb ;
2     GO
3     -- https://msdn.microsoft.com/en-us/library/ms182079.aspx
4     EXEC dbo.sp_add_job
5         @job_name = N'DailyOptimizeAllTables';
6     GO
7     -- https://msdn.microsoft.com/en-us/library/ms187358.aspx
8     EXEC sp_add_jobstep
9         @job_name = N'DailyOptimizeAllTables',
10        @step_name = N'Execute OPTIMIZE_ALL_TABLES stored procedure',
11        @subsystem = N'TSQL',
12        @command = N'EXECUTE OPTIMIZE_ALL_TABLES',
13        @database_name = N'library_ex_2015_mod';
14    GO
15    -- https://msdn.microsoft.com/en-us/library/ms187320.aspx
16    EXEC dbo.sp_add_schedule
17        @schedule_name = N'UpdateBooksStatistics',
18        @freq_type = 4,
19        @freq_interval = 4,
20        @freq_subday_type = 1,
21        @freq_subday_interval = 1,
22        @active_start_time = 000105 ;
```

MS SQL Solution 5.2.2.b (setting the scheduled start) (continued)

```
23  USE msdb ;
24  GO
25  -- https://msdn.microsoft.com/en-us/library/ms186766.aspx
26  EXEC sp_attach_schedule
27     @job_name = N'DailyOptimizeAllTables',
28     @schedule_name = N'DailyOptimizeAllTables';
29  GO
30  -- https://msdn.microsoft.com/en-us/library/ms178625.aspx
31  EXEC dbo.sp_add_jobserver
32      @job_name = N'DailyOptimizeAllTables';
33  GO
```

To make sure that the event is added, we can use the following query.

MS SQL Solution 5.2.2.b (checking the schedule)

```
1   SELECT * FROM msdb.dbo.sysschedules
```

If we look at the selected fields of the result of such a query, we get the following picture. Here we also see the event that activates once an hour the stored procedure created while solving{403} the problem 5.2.2.a{403}.

Name	enabled	freq_type	freq_interval	freq_subday_type	freq_subday_interval	active_start_date	active_start_time
UpdateBooksStatistics	1	4	4	8	1	20160427	000105
DailyOptimizeAllTables	1	4	4	1	1	20160427	000100

This completes the solution for MS SQL Server.

Let's go to Oracle. The code to create the stored procedure we need looks like this.

Oracle Solution 5.2.2.b (the procedure code)

```
1   CREATE OR REPLACE PROCEDURE OPTIMIZE_ALL_TABLES
2   AS
3    table_name VARCHAR(150) := '';
4    query_text VARCHAR(1000) := '';
5    CURSOR tables_cursor IS
6     SELECT TABLE_NAME AS "table_name"
7     FROM   ALL_TABLES
8     WHERE OWNER=USER;
9   BEGIN
10   FOR one_row IN tables_cursor
11   LOOP
12    query_text := 'ALTER TABLE "' || one_row."table_name" ||
13                  '" ENABLE ROW MOVEMENT';
14    DBMS_OUTPUT.PUT_LINE('Enabling row movement for "' ||
15                         one_row."table_name" || '"...');
16    EXECUTE IMMEDIATE query_text;
17
18    query_text := 'ALTER TABLE "' || one_row."table_name" ||
19                  '" SHRINK SPACE COMPACT CASCADE';
20    DBMS_OUTPUT.PUT_LINE('Performing SHRINK SPACE COMPACT CASCADE on "' ||
21                         one_row."table_name" || '"...');
22    EXECUTE IMMEDIATE query_text;
23
24    query_text := 'ALTER TABLE "' || one_row."table_name" ||
25                  '" DISABLE ROW MOVEMENT';
26    DBMS_OUTPUT.PUT_LINE('Disabling row movement for "' ||
27                         one_row."table_name" || '"...');
28    EXECUTE IMMEDIATE query_text;
29   END LOOP;
30  END;
31  /
```

The concept of the presented solution is based on this recommendation[27]. So, we will have to get a list of all tables, and then for each of them, first allow the relocation of rows, then perform compactification and, finally, prohibit the relocation of rows again. Unlike MS SQL Server, all queries here are completely trivial.

We can run the resulting stored procedure as follows.

Oracle — Solution 5.2.2.b (running the functionality check)

```
1    SET SERVEROUTPUT ON;
2    EXECUTE OPTIMIZE_ALL_TABLES;
```

Now it remains to create an event triggered by the Oracle task scheduler according to a certain schedule. This is implemented by the following code.

Oracle — Solution 5.2.2.b (setting the scheduled start)

```
1    BEGIN
2      DBMS_SCHEDULER.CREATE_JOB (
3        job_name            =>  'daily_optimize_all_tables',
4        job_type            =>  'STORED_PROCEDURE',
5        job_action          =>  'OPTIMIZE_ALL_TABLES',
6        start_date          =>  '25-APR-16 4.00.00 PM',
7        repeat_interval     =>  'FREQ=DAILY;INTERVAL=1',
8        auto_drop           =>  FALSE,
9        enabled             =>  TRUE);
10   END;
```

To make sure that the event is added, we can use the following query.

Oracle — Solution 5.2.2.b (checking the schedule)

```
1    SELECT * FROM ALL_SCHEDULER_JOBS WHERE OWNER=USER
```

If we look at the selected fields of the result of such a query, we get the following picture. Here we also see the event that activates a stored procedure once an hour, created while solving{403} the problem 5.2.2.a{403}.

JOB_NAME	JOB_STYLE	JOB_TYPE	JOB_ACTION	START_DATE	REPEAT_INTERVAL	ENABLED	STATE
DAILY_OPTIMIZE_ALL_TABLES	REGULAR	STORED_PROCEDURE	OPTIMIZE_ALL_TABLES	25-APR-27 04.00.00.000000000 PM -03:00	FREQ=DAILY;INTERVAL=1	TRUE	SCHEDULED
HOURLY_UPDATE_BOOKS_STATISTICS	REGULAR	STORED_PROCEDURE	UPDATE_BOOKS_STATISTICS	22-APR-27 04.00.00.000000000 PM -03:00	FREQ=HOURLY;INTERVAL=1	TRUE	SCHEDULED

This completes the solution of this problem.

Task 5.2.2.TSK.A: create a stored procedure that runs on schedule every 12 hours and updates the data in the **`subscriptions_ready`** aggregating table (see problem 3.1.2.b{229}).

Task 5.2.2.TSK.B: create a stored procedure that runs on schedule once a week and optimizes (defragments, compactifies) all database tables that contain at least one million records.

[27] https://asktom.oracle.com/pls/asktom/f?p=100:11:0::::P11_QUESTION_ID:17312316112393#1765387500346472492

5.2.3. Example 40: Using Stored Procedures to Manipulate Database Objects

Problem 5.2.3.a{415}: create a stored procedure that automatically creates and fills the `books_statistics` aggregating table with data (see problem 3.1.2.a{229}).

Problem 5.2.3.b{419}: create a stored procedure that automatically creates and fills the `tables_rc` aggregating table with information about the number of records in all tables of the database in the format (`table_name`, `number_of_records`).

Expected result 5.2.3.a.

When we call a stored procedure, the `books_statistics` table is created and filled with data. If the table already exists at the time the stored procedure is called, the data in it is updated (brought up to date).

For an example of the contents of the `books_statistics` table, see problem 3.1.2.a{229}.

Expected result 5.2.3.b.

When a stored procedure is called, the `tables_rc` table is created and filled with data. If the table already exists at the time the stored procedure is called, the data in it is updated (brought up to date).

An example of the contents of the `tables_rc` table:

table_name	rows_count
authors	7
books	7
books_statistics	1
genres	6
m2m_books_authors	9
m2m_books_genres	11
subscribers	4
subscriptions	11
tables_rc	8

Solution 5.2.3.a{415}.

The solution{382} of problem 5.1.1.c{367} already contains ready-made queries for creating the `books_statistics` table, filling it with data, and updating its data. Now we only need to place these queries inside a stored procedure and add checking the existence of the `books_statistics` table itself.

Solutions for MySQL and MS SQL Server are completely identical, but in Oracle we have to execute all queries via `EXECUTE IMMEDIATE` clause, because if the `books_statistics` table is missing, the database automatically considers stored procedure invalid if it contains queries that access this table directly.

Now all that remains is to provide the code.

Solution for MySQL looks like this.

MySQL Solution 5.2.3.a (the procedure code)

```
DELIMITER $$
CREATE PROCEDURE CREATE_BOOKS_STATISTICS()
BEGIN

 IF NOT EXISTS
  (SELECT `table_name`
   FROM   `information_schema`.`tables`
   WHERE  `table_schema` = DATABASE()
   AND    `table_type` = 'BASE TABLE'
   AND    `table_name` = 'books_statistics')
 THEN
  CREATE TABLE `books_statistics`
   (
       `total` INTEGER UNSIGNED NOT NULL,
       `given` INTEGER UNSIGNED NOT NULL,
       `rest`  INTEGER UNSIGNED NOT NULL
   );
  INSERT INTO `books_statistics`
              (`total`,
               `given`,
               `rest`)
  SELECT IFNULL(`total`, 0),
         IFNULL(`given`, 0),
         IFNULL(`total` - `given`, 0) AS `rest`
  FROM   (SELECT (SELECT SUM(`b_quantity`)
                  FROM   `books`)               AS `total`,
                 (SELECT COUNT(`sb_book`)
                  FROM   `subscriptions`
                  WHERE  `sb_is_active` = 'Y') AS `given`)
         AS `prepared_data`;
 ELSE
  UPDATE `books_statistics`
  JOIN
  (SELECT IFNULL(`total`, 0) AS `total`,
          IFNULL(`given`, 0) AS `given`,
          IFNULL(`total` - `given`, 0) AS `rest`
   FROM   (SELECT (SELECT SUM(`b_quantity`)
                   FROM   `books`)               AS `total`,
                  (SELECT COUNT(`sb_book`)
                   FROM   `subscriptions`
                   WHERE  `sb_is_active` = 'Y') AS `given`)
          AS `prepared_data`) AS `src`
  SET `books_statistics`.`total` = `src`.`total`,
      `books_statistics`.`given` = `src`.`given`,
      `books_statistics`.`rest` = `src`.`rest`;
 END IF;
END;
$$
DELIMITER ;
```

To check the correctness of the obtained solution, we can run the following queries.

MySQL Solution 5.2.3.a (functionality check)

```
DROP TABLE `books_statistics`;
CALL CREATE_BOOKS_STATISTICS;
SELECT * FROM `books_statistics`;
```

Solution for MS SQL Server looks like this.

MS SQL Solution 5.2.3.a (the procedure code)

```
CREATE PROCEDURE CREATE_BOOKS_STATISTICS
AS
BEGIN
 IF NOT EXISTS
  (SELECT [name]
   FROM   sys.tables
   WHERE  [name] = 'books_statistics')
 BEGIN
  CREATE TABLE [books_statistics]
  (
   [total] INTEGER NOT NULL,
   [given] INTEGER NOT NULL,
   [rest]  INTEGER NOT NULL
  );
  INSERT INTO [books_statistics]
              ([total],
               [given],
               [rest])
  SELECT ISNULL([total], 0) AS [total],
         ISNULL([given], 0) AS [given],
         ISNULL([total] - [given], 0) AS [rest]
  FROM   (SELECT (SELECT SUM([b_quantity])
                  FROM   [books])              AS [total],
                 (SELECT COUNT([sb_book])
                  FROM   [subscriptions]
                  WHERE  [sb_is_active] = 'Y') AS [given])
         AS [prepared_data];

  END
   ELSE
  BEGIN
   UPDATE [books_statistics]
   SET
    [books_statistics].[total] = [src].[total],
    [books_statistics].[given] = [src].[given],
    [books_statistics].[rest] = [src].[rest]
    FROM [books_statistics]
   JOIN
   (SELECT ISNULL([total], 0) AS [total],
           ISNULL([given], 0) AS [given],
           ISNULL([total] - [given], 0) AS [rest]
    FROM   (SELECT (SELECT SUM([b_quantity])
                    FROM   [books])              AS [total],
                   (SELECT COUNT([sb_book])
                    FROM   [subscriptions]
                    WHERE  [sb_is_active] = 'Y') AS [given])
           AS [prepared_data]
   ) AS [src]
    ON 1=1;
  END;
END;
GO
```

To check the correctness of the obtained solution, we can run the following queries.

MS SQL Solution 5.2.3.a (functionality check)

```
DROP TABLE [books_statistics];
EXECUTE CREATE_BOOKS_STATISTICS;
SELECT * FROM [books_statistics];
```

Solution for Oracle is as follows. Pay attention to the way the `books_statistics` table existence check is implemented: as Oracle doesn't support `IF NOT EXISTS` scenario, we are forced to put into variable number of rows found and then check its value in the `IF` block.

Oracle | Solution 5.2.3.a (the procedure code)

```
CREATE OR REPLACE PROCEDURE CREATE_BOOKS_STATISTICS
AS
 table_found NUMBER(1) :=0;
BEGIN

 SELECT COUNT(1) INTO table_found
 FROM   ALL_TABLES
 WHERE  OWNER=USER
 AND    TABLE_NAME = 'books_statistics';

 IF (table_found = 0)
 THEN
  EXECUTE IMMEDIATE 'CREATE TABLE "books_statistics"
   (
        "total" NUMBER(10),
        "given" NUMBER(10),
        "rest"  NUMBER(10)
   )';

  EXECUTE IMMEDIATE 'INSERT INTO "books_statistics"
              ("total",
               "given",
               "rest")
   SELECT "total",
          "given",
          ("total" - "given") AS "rest"
   FROM   (SELECT SUM("b_quantity") AS "total"
           FROM   "books")
     JOIN (SELECT COUNT("sb_book") AS "given"
           FROM   "subscriptions"
           WHERE  "sb_is_active" = ''Y'')
     ON 1 = 1';

  ELSE
    EXECUTE IMMEDIATE 'UPDATE "books_statistics"
    SET ("total", "given", "rest") =
    (SELECT "total",
            "given",
            ("total" - "given") AS "rest"
     FROM   (SELECT SUM("b_quantity") AS "total"
             FROM   "books")
       JOIN (SELECT COUNT("sb_book") AS "given"
             FROM   "subscriptions"
             WHERE  "sb_is_active" = ''Y'')
       ON 1 = 1)';
 END IF;
END;
/
```

To check the correctness of the obtained solution, we can run the following queries.

Oracle | Solution 5.2.3.a (functionality check)

```
DROP TABLE "books_statistics";
EXECUTE CREATE_BOOKS_STATISTICS;
SELECT * FROM "books_statistics";
```

This completes the solution of this problem.

Solution 5.2.3.b{415}.

The logic for solving this problem is a combination of the approaches presented in the solutions{408}, {415} of problems 5.2.2.b{403} and 5.2.3.a{415}.

We will:

- check the existence of the target `tables_rc` table;
- create it if it is not found;
- get the list of database tables and for each of them perform the required operation, i.e., get the number of rows and put this number together with the name of the analyzed table into the `tables_rc` aggregating table.

It is important to note that here, to simplify the code, we perform the `TRUNCATE` operation followed by the addition of rows. This saves us from having to implement an algorithm with adding information about new tables, deleting information about old ones, and updating information about existing ones.

However, in real design problems some code may not expect that there is no data in the `tables_rc` table (which is observed in the time interval between the `TRUNCATE` operation and the finish of the fill cycle), and this can lead to errors.

Solution for MySQL looks like this.

MySQL | Solution 5.2.3.b (the procedure code)

```
1    DELIMITER $$
2    CREATE PROCEDURE CACHE_TABLES_RC()
3    BEGIN
4     DECLARE done INT DEFAULT 0;
5     DECLARE tbl_name VARCHAR(200) DEFAULT '';
6     DECLARE all_tables_cursor CURSOR FOR
7      SELECT `table_name`
8      FROM   `information_schema`.`tables`
9      WHERE  `table_schema` = DATABASE()
10     AND    `table_type` = 'BASE TABLE';
11    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
12
13    IF NOT EXISTS
14     (SELECT `table_name`
15      FROM   `information_schema`.`tables`
16      WHERE  `table_schema` = DATABASE()
17      AND    `table_type` = 'BASE TABLE'
18      AND    `table_name` = 'tables_rc')
19    THEN
20     CREATE TABLE `tables_rc`
21     (
22      `table_name` VARCHAR(200),
23      `rows_count` INT
24     );
25    END IF;
26
27    TRUNCATE TABLE `tables_rc`;
28
29    OPEN all_tables_cursor;
```

MySQL Solution 5.2.3.b (the procedure code) (continued)

```
30    tables_loop: LOOP
31     FETCH all_tables_cursor INTO tbl_name;
32     IF done THEN
33      LEAVE tables_loop;
34     END IF;
35
36     SET @table_rc_query = CONCAT('SELECT COUNT(1) INTO @tbl_rc FROM `',
37                                  tbl_name, '`');
38     PREPARE table_opt_stmt FROM @table_rc_query;
39     EXECUTE table_opt_stmt;
40     DEALLOCATE PREPARE table_opt_stmt;
41
42     INSERT INTO `tables_rc` (`table_name`,
43                              `rows_count`)
44                      VALUES (tbl_name,
45                              @tbl_rc);
46
47    END LOOP tables_loop;
48
49    CLOSE all_tables_cursor;
50   END;
51   $$
52   DELIMITER ;
```

To check the correctness of the obtained solution, we can run the following queries.

MySQL Solution 5.2.3.b (functionality check)

```
1    CALL CACHE_TABLES_RC;
2    SELECT * FROM `tables_rc`;
```

Solution for MS SQL Server looks like this.

MS SQL Solution 5.2.3.b (the procedure code)

```
1    CREATE PROCEDURE CACHE_TABLES_RC
2    AS
3    BEGIN
4     DECLARE @table_name NVARCHAR(200);
5     DECLARE @table_rows INT;
6     DECLARE @query_text NVARCHAR(2000);
7     DECLARE tables_cursor CURSOR LOCAL FAST_FORWARD FOR
8      SELECT [name]
9      FROM   sys.tables;
10
11    IF NOT EXISTS
12     (SELECT [name]
13      FROM   sys.tables
14      WHERE  [name] = 'tables_rc')
15    BEGIN
16     CREATE TABLE [tables_rc]
17     (
18      [table_name] VARCHAR(200),
19      [rows_count] INT
20     );
21    END;
22
23    TRUNCATE TABLE [tables_rc];
24
25    OPEN tables_cursor;
26    FETCH NEXT FROM tables_cursor INTO @table_name;
```

MS SQL Solution 5.2.3.b (the procedure code) (continued)

```
27    WHILE @@FETCH_STATUS = 0
28     BEGIN
29      SET @query_text = CONCAT('SELECT @cnt = COUNT(1) FROM [',
30                               @table_name, ']');
31      EXECUTE sp_executesql @query_text, N'@cnt INT OUT', @table_rows OUTPUT;
32
33      INSERT INTO [tables_rc] ([table_name],
34                               [rows_count])
35                       VALUES (@table_name,
36                                       @table_rows);
37
38      FETCH NEXT FROM tables_cursor INTO @table_name;
39     END;
40    CLOSE tables_cursor;
41    DEALLOCATE tables_cursor;
42   END;
43   GO
```

To check the correctness of the obtained solution, we can run the following queries.

MS SQL Solution 5.2.3.b (functionality check)

```
1    EXECUTE CACHE_TABLES_RC;
2    SELECT * FROM [tables_rc];
```

Solution for Oracle looks like this.

Oracle Solution 5.2.3.b (the procedure code)

```
1    CREATE OR REPLACE PROCEDURE CACHE_TABLES_RC
2    AS
3     table_name VARCHAR(150) := '';
4     table_rows NUMBER(10) := 0;
5     table_found NUMBER(1) :=0;
6     query_text VARCHAR(1000) := '';
7     CURSOR tables_cursor IS
8      SELECT TABLE_NAME AS "table_name"
9      FROM   ALL_TABLES
10     WHERE OWNER=USER;
11   BEGIN
12    SELECT COUNT(1) INTO table_found
13    FROM   ALL_TABLES
14    WHERE  OWNER=USER
15    AND    TABLE_NAME = 'tables_rc';
16
17    IF (table_found = 0)
18    THEN
19     EXECUTE IMMEDIATE 'CREATE TABLE "tables_rc"
20     ("table_name" VARCHAR(200),
21      "rows_count" NUMBER(10))';
22    END IF;
23    EXECUTE IMMEDIATE 'TRUNCATE TABLE "tables_rc"';
24
25    FOR one_row IN tables_cursor
26    LOOP
27     query_text := 'SELECT COUNT(1) FROM "' || one_row."table_name" ||
28                   '"';
29     EXECUTE IMMEDIATE query_text INTO table_rows;
30
31     query_text := 'INSERT INTO "tables_rc" ("table_name", "rows_count")
32                    VALUES (''' || one_row."table_name" || ''', ' ||
33                    table_rows || ')';
34     EXECUTE IMMEDIATE query_text;
35    END LOOP;
36   END;
37   /
```

To check the correctness of the obtained solution, we can run the following queries.

Oracle Solution 5.2.3.b (functionality check)

```
1    EXECUTE CACHE_TABLES_RC;
2    SELECT * FROM "tables_rc";
```

Task 5.2.3.TSK.A: create a stored procedure that automatically creates and fills the `arrears` table with data; this table should contain the identifiers and names of readers who still have at least one book for which the return date is set in the past.

Task 5.2.3.TSK.B: create a stored procedure that deletes all indexes (except for primary keys) built on the current database tables and including more than one field.

Task 5.2.3.TSK.C: create a stored procedure that deletes all views for which `SELECT COUNT(1) FROM view` returns a value less than ten.

 © EPAM Systems, 2016–2025

Chapter 6: Using Transactions

6.1. Using Implicit and Explicit Transactions

6.1.1. Example 41: Managing Implicit Transactions

Problem 6.1.1.a{423}: demonstrate DBMS behavior when executing data modification operations while the implicit transaction autocommit is enabled and disabled.

Problem 6.1.1.b{428}: create a stored procedure that performs the following actions:

- determine whether autocommit mode of implicit transactions is enabled;
- disable this mode, if required (if a corresponding parameter with a corresponding value is passed to the procedure);
- perform insertion of N records into the `subscribers` table (N is passed to the procedure by corresponding parameter);
- restore the initial value of implicit transactions autocommit mode (if it has been changed);
- return the time it took to perform the insertion.

Expected result 6.1.1.a.

If the implicit transactions autocommit mode is enabled, data modification is immediately committed; if the implicit transactions autocommit mode is disabled, data modification will not take effect until the transaction is explicitly committed.

Expected result 6.1.1.b.

The stored procedure outputs debug messages for all steps of the algorithm described in the problem and returns information about the amount of time spent on the insertion operation after its completion.

Solution 6.1.1.a{423}.

The implicit transactions autocommit mode is relevant for MySQL[28] and MS SQL Server[29] (and not relevant for Oracle[30], where this behavior is completely at the discretion of the client software) in case transactions are not explicitly framed by expressions to start and commit or rollback.

MySQL by default works with autocommit of implicit transactions enabled, i.e., any data changes take effect immediately. The `autocommit` parameter is responsible for changing this behavior (it can be controlled locally during a session or globally by changing the corresponding setting in the configuration file).

[28] https://dev.mysql.com/doc/refman/8.0/en/commit.html
[29] https://docs.microsoft.com/en-us/sql/t-sql/statements/set-implicit-transactions-transact-sql
[30] https://asktom.oracle.com/pls/apex/f?p=100:11:0%3A%3A%3A%3AP11_QUESTION_ID:314816776423

To solve this problem in MySQL, we need to use the following set of queries.

MySQL Solution 6.1.1.a

```
1    -- Autocommit disabled:
2    SET autocommit = 0;
3
4    SELECT COUNT(*)
5    FROM   `subscribers`; -- 4
6
7    INSERT INTO `subscribers`
8                (`s_name`)
9    VALUES      ('Ivanov I.I.');
10
11   SELECT COUNT(*)
12   FROM   `subscribers`; -- 5
13
14   ROLLBACK;
15
16   SELECT COUNT(*)
17   FROM   `subscribers`; -- 4
18
19   -- Autocommit enabled:
20   SET autocommit = 1;
21
22   SELECT COUNT(*)
23   FROM   `subscribers`; -- 4
24
25   INSERT INTO `subscribers`
26               (`s_name`)
27   VALUES      ('Ivanov I.I.');
28
29   SELECT COUNT(*)
30   FROM   `subscribers`; -- 5
31
32   ROLLBACK;
33
34   SELECT COUNT(*)
35   FROM   `subscribers`; -- 5
```

On lines 1-17 the queries are executed in the mode of disabled autocommit of implicit transactions: that is why the rollback of the transaction in line 14 is successful and the insertion of data made in lines 7-9 is cancelled.

On lines 19-35 the queries are executed in the mode of autocommit of implicit transactions enabled, and therefore rollback of the transaction in line 32 has no effect: the data insertion made in lines 25-27 remains effective.

MS SQL Server (as well as MySQL) works with autocommit of implicit transactions enabled by default, i.e., any data changes take effect immediately. To change this behavior the **`IMPLICIT_TRANSACTIONS`** parameter is responsible (which in general can only be controlled locally during a session; general ideas on how to manage this parameter are described here[31]).

To solve this problem in MS SQL Server we need to use the following set of queries. Note that the **`IMPLICIT_TRANSACTIONS`** parameter in MS SQL Server is logically opposite to the **`autocommit`** parameter in MySQL (i.e., to turn off the autocommit of implicit transactions we must run the **`SET IMPLICIT_TRANSACTIONS ON`** command).

[31] https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-user-options-server-configuration-option

MS SQL Solution 6.1.1.a

```
1    -- Autocommit disabled:
2    SET IMPLICIT_TRANSACTIONS ON;
3
4    SELECT COUNT(*)
5    FROM   [subscribers]; -- 4
6
7    INSERT INTO [subscribers]
8                ([s_name])
9    VALUES      (N'Ivanov I.I.');
10
11   SELECT COUNT(*)
12   FROM   [subscribers]; -- 5
13
14   ROLLBACK;
15
16   SELECT COUNT(*)
17   FROM   [subscribers]; -- 4
18
19   -- Autocommit enabled:
20   SET IMPLICIT_TRANSACTIONS OFF;
21
22   SELECT COUNT(*)
23   FROM   [subscribers]; -- 4
24
25   INSERT INTO [subscribers]
26               ([s_name])
27   VALUES      (N'Ivanov I.I.');
28
29   SELECT COUNT(*)
30   FROM   [subscribers]; -- 5
31
32   ROLLBACK; -- Error! There is no corresponding transaction that
33             -- could be rolled back.
34
35   SELECT COUNT(*)
36   FROM   [subscribers]; -- 5
```

On lines 1-17 the queries are executed in the mode of disabled autocommit of implicit transactions: that is why the rollback of the transaction in line 14 is successful and the insertion of data made in lines 7-9 is cancelled.

On lines 19-36 the queries are executed in the mode of enabled autocommit of implicit transactions, so rollback of the transaction in line 32 does not affect anything: the data insertion made in lines 25-27 remains effective.

In MS SQL Server there is one important feature to consider. If we use the **`BEGIN TRANSACTION`** clause in **`IMPLICIT_TRANSACTIONS ON`** mode, the DBMS reads the created transaction as a nested one (**`@@TRANCOUNT`** takes value 2) and to successfully commit its execution we should use **`COMMIT TRANSACTION`** clause twice. Otherwise, we risk either getting a “zombie” transaction (which is never terminated) or losing the data modification results (if we close the connection with the DBMS). **`ROLLBACK TRANSACTION`** works the same in both modes, rolling back all transactions regardless of their nesting depth.

This problem is exacerbated by the fact that when debugging queries in tools like MS SQL Server Management Studio, we tend to work within the same connection, and instead of a “zombie” transaction we get a continuation of the previous one (not previously closed). That’s why in most cases, when debugging, everything works fine, but in real applications, the behavior becomes incorrect.

Let's demonstrate the MS SQL Server behavior just described.

MS SQL Solution 6.1.1.a (demonstration of MS SQL Server features)

```
-- Default mode
SET IMPLICIT_TRANSACTIONS OFF;
PRINT @@TRANCOUNT; -- 0
-- Start of the first ("parent") transaction
BEGIN TRANSACTION;
 PRINT @@TRANCOUNT; -- 1
 -- Start of the second ("child") transaction
 BEGIN TRANSACTION;
  PRINT @@TRANCOUNT; -- 2
 -- Commit of the second ("child") transaction
 COMMIT TRANSACTION;
 PRINT @@TRANCOUNT; -- 1
-- Commit of the first ("parent") transaction
COMMIT TRANSACTION;
PRINT @@TRANCOUNT; -- 0

-- "Implicit transactions" mode
SET IMPLICIT_TRANSACTIONS ON;
PRINT @@TRANCOUNT; -- 0
-- Start of the first ("parent") transaction
BEGIN TRANSACTION;
 PRINT @@TRANCOUNT; -- 2
 -- Start of the second ("child") transaction
 BEGIN TRANSACTION;
  PRINT @@TRANCOUNT; -- 3
 -- Commit of the second ("child") transaction
 COMMIT TRANSACTION;
 PRINT @@TRANCOUNT; -- 2
-- Commit of the first ("parent") transaction
COMMIT TRANSACTION;
PRINT @@TRANCOUNT; -- 1
-- This COMMIT is also needed
COMMIT TRANSACTION;
PRINT @@TRANCOUNT; -- 0

-- Default mode
SET IMPLICIT_TRANSACTIONS OFF;
-- Start of the first ("parent") transaction
BEGIN TRANSACTION;
 PRINT @@TRANCOUNT; -- 1
 -- Start of the second ("child") transaction
 BEGIN TRANSACTION;
  PRINT @@TRANCOUNT; -- 2
-- Rollback of all transactions
ROLLBACK TRANSACTION;
PRINT @@TRANCOUNT; -- 0

-- "Implicit transactions" mode
SET IMPLICIT_TRANSACTIONS ON;
-- Start of the first ("parent") transaction
BEGIN TRANSACTION;
 PRINT @@TRANCOUNT; -- 2
 -- Start of the second ("child") transaction
 BEGIN TRANSACTION;
  PRINT @@TRANCOUNT; -- 3
-- Rollback of all transactions
ROLLBACK TRANSACTION;
PRINT @@TRANCOUNT; -- 0
```

Oracle (unlike MySQL and MS SQL Server) does not operate with such things as “implicit transaction” and its autocommit. This DBMS only automatically commits the current transaction if an expression modifying the database structure is executed.

However, the client software organizing the interaction with Oracle may have its own settings responsible for automatic commit of transactions that are not explicitly framed by start and commit or rollback expressions.

In a tool like Oracle SQL Developer, e.g., the corresponding effect is achieved by running the **`SET AUTOCOMMIT ON / OFF`** command (the effect of which is equivalent to changing the **`autocommit`** parameter in MySQL).

To solve this problem in Oracle, we need to use the following set of queries.

Oracle | Solution 6.1.1.a

```
1    -- Autocommit disabled:
2    SET AUTOCOMMIT OFF;
3
4    SELECT COUNT(*)
5    FROM   "subscribers"; -- 4
6
7    INSERT INTO "subscribers"
8                ("s_name")
9    VALUES       (N'Ivanov I.I.');
10
11   SELECT COUNT(*)
12   FROM   "subscribers"; -- 5
13
14   ROLLBACK;
15
16   SELECT COUNT(*)
17   FROM   "subscribers"; -- 4
18
19   -- Autocommit enabled:
20   SET AUTOCOMMIT ON;
21
22   SELECT COUNT(*)
23   FROM   "subscribers"; -- 4
24
25   INSERT INTO "subscribers"
26               ("s_name")
27   VALUES       (N'Ivanov I.I.');
28
29   SELECT COUNT(*)
30   FROM   "subscribers"; -- 5
31
32   ROLLBACK;
33
34   SELECT COUNT(*)
35   FROM   "subscribers"; -- 5
```

On lines 1-17 the queries are executed in the mode of disabled autocommit of implicit transactions: that is why the rollback of the transaction in line 14 is successful and the insertion of data made in lines 7-9 is cancelled.

On lines 19-35 the queries are executed in the mode of autocommit of implicit transactions enabled, and therefore rollback of the transaction in line 32 has no effect: the data insertion made in lines 25-27 remains effective.

This completes the solution of this problem.

Solution 6.1.1.b{423}.

This problem is intended not only to remind the principles of stored procedures and control logic of implicit transaction autocommit, but also to demonstrate the difference in DBMS performance in situations when each data modification operation is executed separately, and when such operations are submitted when the whole group of operations is completed.

Traditionally, we start the solution with MySQL and immediately look at the code.

MySQL Solution 6.1.1.b (the procedure code)

```
DELIMITER $$
CREATE PROCEDURE TEST_INSERT_SPEED(IN records_count INT,
                                   IN use_autocommit INT,
                                   OUT total_time TIME(6))
BEGIN
 DECLARE counter INT DEFAULT 0;

 SET @old_autocommit = (SELECT @@autocommit);
 SELECT CONCAT('Old autocommit value = ', @old_autocommit);
 SELECT CONCAT('New autocommit value = ', use_autocommit);

 IF (use_autocommit != @old_autocommit)
 THEN
  SELECT CONCAT('Switching autocommit to ', use_autocommit);
  SET autocommit = use_autocommit;
 ELSE
  SELECT 'No changes in autocommit mode needed.';
 END IF;

 SELECT CONCAT('Starting insert of ', records_count, ' records...');
 SET @start_time = (SELECT NOW(6));
 WHILE counter < records_count DO
  INSERT INTO    `subscribers`
                 (`s_name`)
         VALUES (CONCAT('New subscriber ', (counter + 1)));
  SET counter = counter + 1;
 END WHILE;
 SET @finish_time = (SELECT NOW(6));
 SELECT CONCAT('Finished insert of ', records_count, ' records...');

 IF ((SELECT @@autocommit) = 0)
 THEN
  SELECT 'Current autocommit mode is 0. Performing explicit commit.';
  COMMIT;
 END IF;

 IF (use_autocommit != @old_autocommit)
 THEN
  SELECT CONCAT('Switching autocommit back to ', @old_autocommit);
  SET autocommit = @old_autocommit;
 ELSE
  SELECT 'No changes in autocommit mode were made. No restore needed.';
 END IF;

 SET total_time = (SELECT TIMEDIFF(@finish_time, @start_time));
 SELECT CONCAT('Time used: ', total_time);

 SELECT total_time;
END;
$$
DELIMITER ;
```

Line 8 determines the current autocommit value of implicit transactions (in MySQL this information can be extracted from the `@@autocommit` variable).

Lines 12-18 check the necessity of changing the autocommit mode of implicit transactions and the change itself (if necessary). Lines 37-34 recheck and return the original value if it was changed.

The time spent on the insertion operation is determined by taking the current time before (line 21) and after (line 28) the insertion loop (lines 22-27), and then calculating the difference of these values (line 45).

Lines 31-35 check the current value of the autocommit mode for implicit transactions, and the confirmation is done explicitly in line 34 if autocommit is turned off (here we are not interested in whether it was turned off initially or during our procedure).

Now all that remains is to return the value of the time spent on the insertion loop as a result of the stored procedure (line 48).

We can use the following queries to test the functionality and performance of MySQL in the two implicit transaction modes.

MySQL Solution 6.1.1.b (queries to test the functionality)

```
1    CALL TEST_INSERT_SPEED(100000, 1, @tmp);
2    SELECT @tmp;
3
4    CALL TEST_INSERT_SPEED(100000, 0, @tmp);
5    SELECT @tmp;
```

You can perform a corresponding performance exploration yourself. Let's just note here that disabling the autocommit of implicit transactions can speed up this insertion operation dozens of times.

This completes the MySQL solution.

Let's move on to MS SQL Server. The internal logic of the stored procedure will be very similar to the MySQL solution, the only difference being the way[32] to determine the autocommit mode for implicit transactions. Since this database does not provide this information explicitly, we will try to define it indirectly.

This determination is based on information about the nesting level of the current transaction (`@@TRANCOUNT`) and the current connection settings (`@@OPTIONS`). In lines 12-31 of the stored procedure, we consider all possible combinations of values of these parameters of interest, display debugging information and determine whether the implicit transactions autocommit is enabled.

In lines 36-45 we determine whether we need to change the autocommit mode and change it if necessary.

Lines 47-57 are exactly the same as in the solution for MySQL, and the insertion loop for the specified number of records is executed.

Lines 59-64 check what mode the stored procedure is running in (in case of MySQL, we were guided by the current value of `@@autocommit` variable, but since MS SQL Server does not have it, and determining the current mode is quite nontrivial (see lines 11-31), we assume that we work in the mode that is specified when calling the stored procedure).

It only remains to restore the original `IMPLICIT_TRANSACTIONS` value (lines 65-74) and determine how long the insertion loop took (lines 76-80). Such a cumbersome construction using `CONVERT`, `DATEADD`, and `DATEDIFF` functions is necessary to get the time spent in a human-readable form at the output.

[32] http://stackoverflow.com/questions/2919018/in-sql-server-how-do-i-know-what-transaction-mode-im-currently-using

So, here is the procedure code.

MS SQL Solution 6.1.1.b (the procedure code)

```
1    CREATE PROCEDURE TEST_INSERT_SPEED @records_count INT,
2                                       @use_autocommit INT,
3                                       @total_time TIME OUTPUT
4    AS
5    BEGIN
6     DECLARE @counter INT = 0;
7     DECLARE @old_autocommit INT = 0;
8     DECLARE @start_time TIME;
9     DECLARE @finish_time TIME;
10
11    IF (@@TRANCOUNT = 0 AND (@@OPTIONS & 2 = 0))
12     BEGIN
13      PRINT 'IMPLICIT_TRANSACTIONS = OFF, no transaction is running.';
14      SET @old_autocommit = 1;
15     END
16    ELSE IF (@@TRANCOUNT = 0 AND (@@OPTIONS & 2 = 2))
17     BEGIN
18      PRINT 'IMPLICIT_TRANSACTIONS = ON, no transaction is running.';
19      SET @old_autocommit = 0;
20     END
21    ELSE IF (@@OPTIONS & 2 = 0)
22     BEGIN
23      PRINT 'IMPLICIT_TRANSACTIONS = OFF, explicit transaction is running.';
24      SET @old_autocommit = 1;
25     END
26    ELSE
27     BEGIN
28      PRINT 'IMPLICIT_TRANSACTIONS = ON, implicit or explicit transaction
29             is running.';
30      SET @old_autocommit = 0;
31     END;
32
33    PRINT CONCAT('Old autocommit value = ', @old_autocommit);
34    PRINT CONCAT('New autocommit value = ', @use_autocommit);
35
36    IF (@use_autocommit != @old_autocommit)
37     BEGIN
38      PRINT CONCAT('Switching autocommit to ', @use_autocommit);
39      IF (@use_autocommit = 1)
40       SET IMPLICIT_TRANSACTIONS OFF;
41      ELSE
42       SET IMPLICIT_TRANSACTIONS ON;
43     END
44    ELSE
45     PRINT 'No changes in autocommit mode needed.';
46
47    PRINT CONCAT('Starting insert of ', @records_count, ' records...');
48    SET @start_time = GETDATE();
49    WHILE (@counter < @records_count)
50     BEGIN
51      INSERT INTO [subscribers]
52                  ([s_name])
53           VALUES (CONCAT('New subscriber ', (@counter + 1)));
54      SET @counter = @counter + 1;
55     END;
```

MS SQL Solution 6.1.1.b (the procedure code) (continued)

```
56    SET @finish_time = GETDATE();
57    PRINT CONCAT('Finished insert of ', @records_count, ' records...');
58
59    IF (@use_autocommit = 0)
60     BEGIN
61      PRINT 'Current autocommit mode is 0 (IMPLICIT_TRANSACTIONS = ON).
62             Performing explicit commit.';
63      COMMIT;
64     END;
65    IF (@use_autocommit != @old_autocommit)
66     BEGIN
67      PRINT CONCAT('Switching autocommit back to ', @old_autocommit);
68      IF (@old_autocommit = 1)
69       SET IMPLICIT_TRANSACTIONS OFF;
70      ELSE
71       SET IMPLICIT_TRANSACTIONS ON;
72     END
73    ELSE
74     PRINT 'No changes in autocommit mode needed.';
75
76    SET @total_time = CONVERT(VARCHAR(12),
77                              DATEADD(ms,
78                                      DATEDIFF(ms, @start_time, @finish_time),
79                                      0),
80                              114);
81    PRINT CONCAT('Time used: ', @total_time);
82    RETURN;
83   END;
84   GO
```

We can use the following queries to test the functionality and evaluate the performance of MS SQL Server in the two modes of operation with implicit transactions.

MS SQL Solution 6.1.1.b (queries to test the functionality)

```
1    DECLARE @t TIME;
2    SET IMPLICIT_TRANSACTIONS ON
3    EXECUTE TEST_INSERT_SPEED 10, 1, @t OUTPUT;
4    PRINT CONCAT ('Stored procedure has returned the following value: ', @t);
5
6    DECLARE @t TIME;
7    SET IMPLICIT_TRANSACTIONS ON
8    EXECUTE TEST_INSERT_SPEED 10, 0, @t OUTPUT;
9    PRINT CONCAT ('Stored procedure has returned the following value: ', @t);
```

This completes the MySQL solution.

Let's move on to Oracle. Since this DBMS does not operate with such a concept as "autocommit of implicit transactions" at all, we can work only in one of two modes (which we explicitly choose and implement ourselves):

- performing transaction commit after each operation (similar to the enabled autocommit of implicit transactions);
- performing transaction commit after a series of transactions (similar to the disabled autocommit of implicit transactions)

Given this fact, we implement a solution for Oracle similar to MySQL and MS SQL Server, but without defining the current autocommit mode of transactions.

Since Oracle's interaction with the client software usually takes place in transaction mode (i.e., the execution of any expression to modify data takes place within a transaction), at the beginning of our stored procedure we must perform the COMMIT operation (line 12) to end the current transaction (if there is one).

In lines 16-27 we execute the insertion loop, in which we can explicitly initiate an insertion commit for each individual record (line 24) if we need to emulate the autocommit mode of implicit transactions. If such emulation is not needed, all insertions performed in the loop are committed as a set of operations (line 35).

Oracle | Solution 6.1.1.b (the procedure code)

```
1    CREATE OR REPLACE PROCEDURE TEST_INSERT_SPEED(records_count IN INT,
2                                                  use_autocommit IN INT,
3                                                  total_time OUT NVARCHAR2)
4    AS
5     counter INT := 0;
6     start_time TIMESTAMP;
7     finish_time TIMESTAMP;
8     diff_time INTERVAL DAY TO SECOND;
9    BEGIN
10    DBMS_OUTPUT.PUT_LINE('Autocommit value = ' || use_autocommit);
11    DBMS_OUTPUT.PUT_LINE('Committing previous transaction...');
12    COMMIT;
13    DBMS_OUTPUT.PUT_LINE('Starting insert of ' || records_count ||
14                         ' records...');
15    start_time := CURRENT_TIMESTAMP;
16    WHILE (counter < records_count)
17     LOOP
18      INSERT INTO "subscribers"
19                  ("s_name")
20           VALUES (CONCAT('New subscriber ', (counter + 1)));
21      IF (use_autocommit = 1)
22       THEN
23        DBMS_OUTPUT.PUT_LINE('Committing small transaction...');
24        COMMIT;
25       END IF;
26      counter := counter + 1;
27     END LOOP;
28    finish_time := CURRENT_TIMESTAMP;
29    DBMS_OUTPUT.PUT_LINE('Finished insert of ' || records_count ||
30                         ' records...');
31
32    IF (use_autocommit = 0)
33    THEN
34     DBMS_OUTPUT.PUT_LINE('Committing one big transaction...');
35     COMMIT;
36    END IF;
37
38    diff_time := finish_time - start_time;
39    total_time := TO_CHAR(EXTRACT(hour FROM diff_time)) || ':' ||
40                  TO_CHAR(EXTRACT(minute FROM diff_time)) || ':' ||
41                  TO_CHAR(EXTRACT(second FROM diff_time ), 'fm00.000000' );
42
43    DBMS_OUTPUT.PUT_LINE('Time used: ' || total_time);
44   END;
```

We can use the following queries to test the functionality and evaluate the performance of Oracle in the two implicit transaction modes.

Oracle | Solution 6.1.1.b (queries to test the functionality)

```
DECLARE
 t NVARCHAR2(100);
BEGIN
 TEST_INSERT_SPEED(10, 1, t);
 DBMS_OUTPUT.PUT_LINE('Stored procedure has returned
                       The following value: ' || t);
END;

DECLARE
 t NVARCHAR2(100);
BEGIN
 TEST_INSERT_SPEED(10, 0, t);
 DBMS_OUTPUT.PUT_LINE('Stored procedure has returned
                       The following value: ' || t);
END;
```

This completes the solution of this problem.

Task 6.1.1.TSK.A: compare the speed of the stored procedure presented in the solution{428} of problem 6.1.1.b{423} when inserting in both autocommit modes of implicit transactions of 10, 100, 1000, 10000, 100000 records in all three DBMSes.

6.1.2. Example 42: Managing Explicit Transactions

Problem 6.1.2.a{434}: create a stored procedure that:
- adds three random books to each reader with the subscriptions start date equal to the current date and the subscription finish date equal to “current date plus month”;
- cancels the actions performed if at least one reader has more than ten books on hand at the end of the operation.

Problem 6.1.2.b{440}: create a stored procedure that:
- changes all subscriptions finish dates to “plus three months”;
- cancels the action performed if the average reading time of all books exceeds 4 months.

Expected result 6.1.2.a.

The stored procedure performs the actions specified in the problem condition, at the end of its work signaling the commit or rollback of changes made.

Expected result 6.1.2.b.

The stored procedure performs the actions specified in the problem condition, at the end of its work signaling the commit or rollback of changes made.

Solution 6.1.2.a{434}.

The solution to this MySQL problem will be notable for the way to handle multiple cursors within the same stored procedure.

In order to obtain the solution, we’ll have to:
- start the transaction (line 5);
- open the cursor to retrieve all readers’ identifiers (line 15);
- for each reader identifier execute a nested loop (lines 16-55), in which:
  - open the cursor to extract three random books’ identifiers (line 13);
  - insert each of the obtained book identifier into the `subscriptions` table (lines 33-51);
  - close the cursor that gets three random books’ identifiers (line 52);
- close the cursor that retrieves the identifiers of all readers (line 56);
- check whether the condition of not allowing more than ten books in the hands of one reader (lines 58-70) was violated, and:
  - if the condition was violated, rollback the transaction (line 66);
  - if the condition has not been violated, commit the transaction (line 69).

Despite the cumbersome syntax and long description, the algorithm itself is trivial: it is a usual nested loop. What is interesting about this problem is the previously mentioned work with two cursors.

Please note that the `DECLARE CONTINUE HANDLER FOR NOT FOUND SET` clause does not imply specifying a cursor name, i.e., it is supposed to be a single cursor. When it is necessary to use several cursors, the so called “code blocks”, which limit the scope of variables, are used. In our case, there are two such blocks, the second one being enclosed in the first one and located on lines 7-57 and 22-53, respectively).

MySQL Solution 6.1.2.a (the procedure code)

```
DELIMITER $$
CREATE PROCEDURE THREE_RANDOM_BOOKS()
BEGIN
 SELECT 'Starting transaction...';
 START TRANSACTION;

 USERS: BEGIN
  DECLARE s_id_value INT DEFAULT 0;
  DECLARE subscribers_done INT DEFAULT 0;
  DECLARE subscribers_cursor CURSOR FOR
   SELECT `s_id`
   FROM   `subscribers`;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET subscribers_done = 1;

  OPEN subscribers_cursor;
  read_users_loop: LOOP
   FETCH subscribers_cursor INTO s_id_value;
   IF subscribers_done THEN
    LEAVE read_users_loop;
   END IF;

   BOOKS: BEGIN
    DECLARE b_id_value INT DEFAULT 0;
    DECLARE books_done INT DEFAULT 0;
    DECLARE books_cursor CURSOR FOR
     SELECT `b_id`
     FROM   `books`
     ORDER BY RAND()
     LIMIT 3;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET books_done = 1;
    OPEN books_cursor;

    read_books_loop: LOOP
    FETCH books_cursor INTO b_id_value;
     IF books_done THEN
      LEAVE read_books_loop;
     END IF;

     INSERT INTO `subscriptions`
                 (`sb_subscriber`,
                  `sb_book`,
                  `sb_start`,
                  `sb_finish`,
                  `sb_is_active`)
          VALUES (s_id_value,
                  b_id_value,
                  NOW(),
                  NOW() + INTERVAL 1 MONTH,
                  'Y');

    END LOOP read_books_loop;
    CLOSE books_cursor;
   END BOOKS;

  END LOOP read_users_loop;
  CLOSE subscribers_cursor;
 END USERS;
```

MySQL Solution 6.1.2.a (the procedure code) (continued)

```
58    IF EXISTS (SELECT 1
59                FROM  `subscriptions`
60                WHERE `sb_is_active`='Y'
61                GROUP BY `sb_subscriber`
62                HAVING COUNT(1)>10
63                LIMIT 1)
64     THEN
65      SELECT 'Rolling transaction back...';
66      ROLLBACK;
67     ELSE
68      SELECT 'Committing transaction...';
69      COMMIT;
70    END IF;
71
72   END;
73   $$
74   DELIMITER ;
```

Note the names of the variables into which `` `s_id` `` and `` `b_id` `` field values are extracted: **`s_id_value`** and **`b_id_value`**. The **`_value`** part was not added there by accident, because if such variables have the same name as the table fields, MySQL will not place data into them.

We can use the following queries to test the functionality of the resulting solution. If we execute them on the initial data set of the “Library” database, then the operation will complete successfully twice, and the third and subsequent calls will end with the rollback of the transaction.

MySQL Solution 6.1.2.a (queries to test the functionality)

```
1    CALL THREE_RANDOM_BOOKS();
2    SELECT * FROM `subscriptions`;
```

This completes the MySQL solution.

Now let’s turn to MS SQL Server. The general solution logic for this database is the same as for MySQL, but since we have to work with nested cursors in a different way, let’s repeat the algorithm with references to the corresponding code fragments.

So, in order to get the solution, we’ll have to:

- start the transaction (line 17);
- open the cursor to retrieve identifiers of all readers (line 19);
- for each reader identifier perform a nested loop (lines 23-49), in which:
  - open the cursor to retrieve three random books’ identifiers (line 25);
  - for each obtained book identifier make an insertion in the **`subscriptions`** table (lines 28-44);
  - close the cursor that retrieves three random books’ identifiers (line 45);
- close the cursor that retrieves identifiers of all readers (line 50);
- check whether the condition that one reader may not hold more than ten books (lines 53-66) has been violated, and:
  - if the condition has been violated, rollback the transaction (line 60);
  - if the condition has not been violated, commit the transaction (line 65).

With MS SQL Server we do not need to use separate blocks of code for each cursor. Instead, we save the value of **`@@FETCH_STATUS`** parameter (which provides information about the last data extraction operation from a cursor) into a separate variable for each loop (lines 21, 27, 43, 48) and then use these variables to organize loops.

MS SQL Solution 6.1.2.a (the procedure code)

```
1    CREATE PROCEDURE THREE_RANDOM_BOOKS
2    AS
3    BEGIN
4     DECLARE @s_id_value INT;
5     DECLARE @b_id_value INT;
6     DECLARE subscribers_cursor CURSOR LOCAL FAST_FORWARD FOR
7      SELECT [s_id]
8      FROM   [subscribers];
9     DECLARE books_cursor CURSOR LOCAL FAST_FORWARD FOR
10     SELECT TOP 3 [b_id]
11     FROM         [books]
12     ORDER BY     NEWID();
13    DECLARE @fetch_subscribers_cursor INT;
14    DECLARE @fetch_books_cursor INT;
15
16    PRINT 'Starting transaction...';
17    BEGIN TRANSACTION;
18
19    OPEN subscribers_cursor;
20    FETCH NEXT FROM subscribers_cursor INTO @s_id_value;
21    SET @fetch_subscribers_cursor = @@FETCH_STATUS;
22
23    WHILE @fetch_subscribers_cursor = 0
24     BEGIN
25      OPEN books_cursor;
26      FETCH NEXT FROM books_cursor INTO @b_id_value;
27      SET @fetch_books_cursor = @@FETCH_STATUS;
28      WHILE @fetch_books_cursor = 0
29       BEGIN
30        INSERT INTO [subscriptions]
31                    ([sb_subscriber],
32                     [sb_book],
33                     [sb_start],
34                     [sb_finish],
35                     [sb_is_active])
36             VALUES (@s_id_value,
37                     @b_id_value,
38                     GETDATE(),
39                     DATEADD(month, 1, GETDATE()),
40                     N'Y');
41
42        FETCH NEXT FROM books_cursor INTO @b_id_value;
43        SET @fetch_books_cursor = @@FETCH_STATUS;
44       END;
45      CLOSE books_cursor;
46
47      FETCH NEXT FROM subscribers_cursor INTO @s_id_value;
48      SET @fetch_subscribers_cursor = @@FETCH_STATUS;
49     END;
50    CLOSE subscribers_cursor;
51    DEALLOCATE subscribers_cursor;
52    DEALLOCATE books_cursor;
```

MS SQL Solution 6.1.2.a (the procedure code) (continued)

```
53     IF EXISTS (SELECT TOP 1 1
54                FROM  [subscriptions]
55                WHERE [sb_is_active]='Y'
56                GROUP BY [sb_subscriber]
57                HAVING COUNT(1)>10)
58      BEGIN
59       PRINT 'Rolling transaction back...';
60       ROLLBACK TRANSACTION;
61      END
62     ELSE
63      BEGIN
64       PRINT 'Committing transaction...';
65       COMMIT TRANSACTION;
66      END;
67
68   END;
69   GO
```

The following queries can be used to check the functionality of the resulting solution.

MS SQL Solution 6.1.2.a (queries to test the functionality)

```
1    EXECUTE THREE_RANDOM_BOOKS;
2    SELECT * FROM [subscriptions];
```

This completes the solution for MS SQL Server.

Let's move on to Oracle. Although we have already reviewed the solution algorithm twice, we will repeat it here again, so that by following the references to the code, we can see how simple and elegant the work with nested cursors is implemented in Oracle.

So, in order to get the solution, we'll have to:

- commit the previous transaction (line 17) (recall that in Oracle a transaction is always activated by the first data modification operation);
- create a loop to go through the cursor rows to retrieve the identifiers of all readers (lines 19-35), and inside this loop:
  - create a loop to pass through the cursor rows to retrieve three random book identifiers (lines 21-34);
  - insert each obtained book identifier into the **`subscriptions`** table (lines 23-33);
- check whether the condition of not allowing more than ten books in the hands of one reader (lines 37-52) has been violated and:
  - if the condition has been violated, rollback the transaction (line 48);
  - if the condition has not been violated, commit the transaction (line 51).

The only little inconvenience in this solution is the necessity to find out the existence of records violating the problem condition through an intermediate variable and a subquery (lines 37-43), which is associated with impossibility of using the **`IF EXISTS`** clause in Oracle. Otherwise, the entire stored procedure code looks no more complicated than a primitive example in any common programming language.

Oracle — Solution 6.1.2.a (the procedure code)

```
CREATE OR REPLACE PROCEDURE THREE_RANDOM_BOOKS
AS
 counter INT := 0;
 CURSOR subscribers_cursor IS
  SELECT "s_id"
  FROM   "subscribers";
 CURSOR books_cursor IS
  SELECT "b_id"
  FROM
   (SELECT "b_id"
    FROM   "books"
    ORDER BY DBMS_RANDOM.VALUE)
  WHERE ROWNUM <= 3;

BEGIN
 DBMS_OUTPUT.PUT_LINE('Committing previous transaction...');
 COMMIT;

 FOR one_subscriber IN subscribers_cursor
 LOOP
  FOR one_book IN books_cursor
  LOOP
   INSERT INTO "subscriptions"
               ("sb_subscriber",
                "sb_book",
                "sb_start",
                "sb_finish",
                "sb_is_active")
        VALUES (one_subscriber."s_id",
                one_book."b_id",
                SYSDATE,
                ADD_MONTHS(SYSDATE, 1),
                'Y');
  END LOOP;
 END LOOP;

 SELECT COUNT(1) INTO counter
 FROM
 (SELECT COUNT(1)
  FROM "subscriptions"
  WHERE "sb_is_active"='Y'
  GROUP BY "sb_subscriber"
  HAVING COUNT(1)>10);

 IF (counter > 0)
  THEN
   DBMS_OUTPUT.PUT_LINE('Rolling transaction back...');
   ROLLBACK;
  ELSE
   DBMS_OUTPUT.PUT_LINE('Committing transaction...');
   COMMIT;
 END IF;

END;
```

The following queries can be used to check the functionality of the resulting solution.

Oracle — Solution 6.1.2.a (queries to test the functionality)

```
SET SERVEROUTPUT ON;
EXECUTE THREE_RANDOM_BOOKS;
SELECT * FROM "subscriptions";
```

This completes the solution of this problem.

Solution 6.1.2.b{434}.

In contrast to the solution{434} of problem 6.1.2.a{434}, here we don't even need cursors. So, the solution is reduced to a series of simple actions:

- make changes;
- check whether the condition of the problem is violated, and:
  - rollback changes if violated;
  - commit changes if not violated.

All that remains is to look at the stored procedure code. The only differences will be in the way the date intervals are calculated and (in Oracle) the way the transaction starts. Otherwise, the solutions for all three DBMSes are completely equivalent.

Solution for MySQL looks like this.

MySQL Solution 6.1.2.b (the procedure code)

```
DELIMITER $$
CREATE PROCEDURE CHANGE_DATES()
BEGIN
 SELECT 'Starting transaction...';
 START TRANSACTION;

 UPDATE `subscriptions`
    SET `sb_finish` = DATE_ADD(`sb_finish`, INTERVAL 3 MONTH);

 SET @avg_read = (SELECT AVG(DATEDIFF(`sb_finish`, `sb_start`))
                    FROM `subscriptions`);

 IF (@avg_read > 120)
  THEN
   SELECT 'Rolling transaction back...';
   ROLLBACK;
  ELSE
   SELECT 'Committing transaction...';
   COMMIT;
 END IF;

END;
$$
DELIMITER ;
```

We can use the following query to check the functionality of the resulting solution.

MySQL Solution 6.1.2.b (queries to test the functionality)

```
CALL CHANGE_DATES()
```

Solution for MS SQL Server looks like this.

MS SQL Solution 6.1.2.b (the procedure code)

```
1    CREATE PROCEDURE CHANGE_DATES
2    AS
3    BEGIN
4     DECLARE @avg_read DOUBLE PRECISION;
5
6     PRINT 'Starting transaction...';
7     BEGIN TRANSACTION;
8
9     UPDATE [subscriptions]
10       SET [sb_finish] = DATEADD(month, 3, [sb_finish]);
11
12    SET @avg_read = (SELECT AVG(DATEDIFF (month, [sb_start], [sb_finish]))
13                        FROM [subscriptions]);
14
15    IF (@avg_read > 4)
16     BEGIN
17      PRINT 'Rolling transaction back...';
18      ROLLBACK TRANSACTION;
19     END
20    ELSE
21     BEGIN
22      PRINT 'Committing transaction...';
23      COMMIT TRANSACTION;
24     END;
25
26   END;
27   GO
```

We can use the following query to check the functionality of the resulting solution.

MS SQL Solution 6.1.2.b (queries to test the functionality)

```
1    EXECUTE CHANGE_DATES
```

Solution for Oracle looks like this.

Oracle Solution 6.1.2.b (the procedure code)

```
1    CREATE OR REPLACE PROCEDURE CHANGE_DATES
2    AS
3     avg_read NUMBER(5, 3) := 0.0;
4
5    BEGIN
6     DBMS_OUTPUT.PUT_LINE('Committing previous transaction...');
7     COMMIT;
8
9     UPDATE "subscriptions"
10       SET "sb_finish" = ADD_MONTHS("sb_finish", 3);
11
12    SELECT AVG(MONTHS_BETWEEN("sb_finish", "sb_start")) INTO avg_read
13    FROM "subscriptions";
14
15    IF (avg_read > 4.0)
16     THEN
17      DBMS_OUTPUT.PUT_LINE('Rolling transaction back...');
18      ROLLBACK;
19     ELSE
20      DBMS_OUTPUT.PUT_LINE('Committing transaction...');
21      COMMIT;
22    END IF;
23
24   END;
```

We can use the following query to check the functionality of the resulting solution.

Oracle — Solution 6.1.2.b (queries to test the functionality)

```
1    EXECUTE CHANGE_DATES
```

This completes the solution of this problem.

Task 6.1.2.TSK.A: create a stored procedure that:
- adds two random genres to each book;
- rolls back the performed actions if at least one insertion operation has failed due to duplicated primary key value of the **`m2m_books_genres`** table (i.e., such a book already had such a genre).

Task 6.1.2.TSK.B: create a stored procedure that:
- increases the value of the **`b_quantity`** field for all books by a factor of two;
- rolls back the performed actions if the average number of copies of books exceeds 50 as a result of the operation.

6.2. Understanding Transactions Concurrency

6.2.1. Example 43: Managing Transactions Isolation Level

Problem 6.2.1.a{443}: create queries that, if executed in parallel, would provide the following effect:

- the first query should add one day to all subscriptions finish dates and not depend on read queries from the subscriptions table (i.e., it must not wait for such queries to complete);
- the second query should read all finish dates from the subscriptions table and not depend on the first query (i.e., it must not wait for the first query to complete).

Problem 6.2.1.b{446}: create two queries, each of which will count the number of books given to each reader, but:

- the first query should be executed as quickly as possible (even at the cost of providing not entirely reliable data);
- the second query must provide guaranteed reliable data (even at the cost of a long execution time).

Expected result 6.2.1.a.

With any start order (“first, then second” or “second, then first”) queries run in parallel, and neither query waits for the other to finish.

Expected result 6.2.1.b.

The first query never waits for any other queries to finish, the second query can be put in the waiting queue.

Solution 6.2.1.a{443}.

In MySQL, when using the InnoDB storage engine, queries to update data by default have a higher priority than queries to read data.

If a read operation is already in progress when we start the update, the update will wait for read completion only in one case: if it is started in a transaction with the `SERIALIZABLE` isolation level. Considering that the default isolation level of transactions is `REPEATABLE READ`, we have no issues with the first part of the problem: the update will start immediately.

Now we need to get DBMS to work in such a way that read query will be executed in parallel with update query. This is not a problem either, if we don’t run it in the `SERIALIZABLE` mode. It remains to be decided, do we want to get “raw data” (changes that have not yet taken effect) or do we want to get only the data really stored in the database?

In the first case, the read query should be performed in a transaction with the `READ UNCOMMITTED` isolation level; in the second case, the read query should be performed in a transaction with the `READ COMMITTED` or `REPEATABLE READ` isolation levels.

All that remains is to implement this idea in the code. The following two blocks of code need to be executed in separate connections to the DBMS (separate sessions), so make sure that the queries in the first lines of each block return different session identifiers. In MySQL Workbench, we can open multiple copies of the same DBMS connection[33], and they will run in different sessions.

MySQL Solution 6.2.1.a (the first block)

```
1    SELECT CONNECTION_ID();
2    SET autocommit = 0;
3    START TRANSACTION;
4    UPDATE `subscriptions`
5    SET    `sb_finish` = DATE_ADD(`sb_finish`, INTERVAL 1 DAY);
6    -- SELECT SLEEP(10);
7    COMMIT;
```

MySQL Solution 6.2.1.a (the second block)

```
1    SELECT CONNECTION_ID();
2    SET autocommit = 0;
3    SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
4    START TRANSACTION;
5    SELECT `sb_finish`
6    FROM   `subscriptions`;
7    -- SELECT SLEEP(10);
8    COMMIT;
```

By uncommenting the line with `SELECT SLEEP(10)` in the corresponding code block, we will simulate its long execution, which will allow us to calmly execute the second block several times (where the similar line will remain commented) and look at the result.

This completes the MySQL solution, but to better understand the logic of transactions it is highly recommended to conduct a series of experiments, changing the isolation level of transactions in the second block of code and observing the changes in the behavior of the DBMS.

Let's move on to MS SQL Server. The logic here is almost identical to that of MySQL solution, yet there are some differences:

- the transaction isolation level in MS SQL is `READ COMMITTED` by default (it does not affect the solution of this problem);
- when executing a query to read in a transaction with the `READ COMMITTED` isolation level MS SQL Server (unlike MySQL) will not return immediately the current actual data but will wait for the completion of concurrent transactions performing data modification (it follows that to meet the condition of the problem we must perform a read query in a transaction with the `READ UNCOMMITTED` isolation level).

Let's look at the code. The following two blocks of code must be executed in separate connections to the DBMS (separate sessions), so make sure that the queries in the first lines of each block return different session identifiers. In MS SQL Server Management Studio separate windows for executing SQL queries will run in separate sessions[34].

[33] https://dev.mysql.com/doc/workbench/en/wb-mysql-connections-new.html

[34] https://docs.microsoft.com/en-us/sql/ssms/understand-sql-server-management-studio-windows-management

MS SQL Solution 6.2.1.a (the first block)

```
SELECT @@SPID;
SET IMPLICIT_TRANSACTIONS ON;
BEGIN TRANSACTION;
UPDATE [subscriptions]
SET    [sb_finish] = DATEADD(day, 1, [sb_finish]);
-- WAITFOR DELAY '00:00:10';
COMMIT TRANSACTION;
```

MS SQL Solution 6.2.1.a (the second block)

```
SELECT @@SPID;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRANSACTION;
SELECT [sb_finish]
FROM   [subscriptions];
-- WAITFOR DELAY '00:00:10';
COMMIT TRANSACTION;
```

By uncommenting the line with `WAITFOR DELAY '00:00:10'` in the corresponding code block, we will simulate its long execution, which will allow us to calmly execute the second block several times (in which the similar line will remain commented) and look at the result.

This completes the solution of this problem for MS SQL Server.

Let's move on to Oracle. Continuing the analogy with the MySQL and MS SQL solutions just discussed, we note that:

- the default transaction isolation level in Oracle is `READ COMMITTED` (like in MS SQL Server);
- unlike MySQL and MS SQL Server, Oracle has no `READ UNCOMMITTED` level of transaction isolation;
- operations of data reading and modification in Oracle do not block each other[35], so the solution of the current problem is reduced to simply performing the necessary queries (but to maintain consistency, we will stick to the same set of commands that was used in MySQL and MS SQL Server).

Let's look at the code. The following two blocks of code must be executed in separate connections to the DBMS (separate sessions), so make sure that the queries in the second line of each block return different session identifiers. In Oracle SQL Developer we can open a new window for executing queries in a separate session by pressing Ctrl+Shift+N.

In the first lines of both code blocks, the `COMMIT` operation is executed to ensure the execution of further queries in a new separate transaction.

Oracle Solution 6.2.1.a (the first block)

```
COMMIT;
SELECT SYS_CONTEXT('userenv', 'sessionid')
FROM   DUAL;
UPDATE "subscriptions"
SET    "sb_finish" = "sb_finish" + 1;
-- EXEC DBMS_LOCK.SLEEP(10);
COMMIT;
```

[35] https://asktom.oracle.com/pls/apex/f?p=100:11:::NO::P11_QUESTION_ID:9526589900346732664

Oracle Solution 6.2.1.a (the second block)

```
1     COMMIT;
2     SELECT SYS_CONTEXT('userenv', 'sessionid')
3     FROM   DUAL;
4     SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
5     SELECT "sb_finish"
6     FROM   "subscriptions" ORDER BY "sb_finish" ASC;
7     -- EXEC DBMS_LOCK.SLEEP(10);
8     COMMIT;
```

By uncommenting the line with **`EXEC DBMS_LOCK.SLEEP(10)`** in the corresponding code block, we will simulate its long execution, which allows us to calmly execute the second block several times (in which the similar line will remain commented) and look at the result.

This completes the solution of this problem.

Solution 6.2.1.b{443}.

The solution of this problem obeys the general logic of transaction isolation levels:

- the lower the level, the more possibilities the DBMS has to execute the query in parallel with others, but the higher the probability to get an incorrect result;
- the higher the level, the fewer possibilities the DBMS has to execute the query in parallel with others, but the lower the probability of getting an incorrect result;
- in MySQL and MS SQL Server the lowest level is **`READ UNCOMMITTED`**, in Oracle it is **`READ COMMITTED`**;
- in all three DBMSes the highest level is **`SERIALIZABLE`**.

Given these facts, all we have to do is create code to execute the same query at the lowest and the highest levels of transaction isolation, and prepare test code to see the difference between the two options for executing the main code.

The code for MySQL looks like this.

MySQL Solution 6.2.1.b (the fastest possible execution, incorrect results are possible)

```
1     SELECT CONNECTION_ID();
2     SET autocommit = 0;
3     SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
4     START TRANSACTION;
5     SELECT `sb_subscriber`,
6            COUNT(`sb_book`) AS `sb_has_books`
7     FROM   `subscriptions`
8     WHERE  `sb_is_active` = 'Y'
9     GROUP BY `sb_subscriber`;
10    COMMIT;
```

MySQL Solution 6.2.1.b (correct results, possible long execution)

```
1     SELECT CONNECTION_ID();
2     SET autocommit = 0;
3     SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
4     START TRANSACTION;
5     SELECT `sb_subscriber`,
6            COUNT(`sb_book`) AS `sb_has_books`
7     FROM   `subscriptions`
8     WHERE  `sb_is_active` = 'Y'
9     GROUP BY `sb_subscriber`;
10    COMMIT;
```

MySQL Solution 6.2.1.b (test code)

```
SELECT CONNECTION_ID();
SET autocommit = 0;
START TRANSACTION;
UPDATE `subscriptions`
SET    `sb_is_active` =
       CASE
        WHEN `sb_is_active` = 'Y' THEN 'N'
        WHEN `sb_is_active` = 'N' THEN 'Y'
       END;
SELECT SLEEP(10);
COMMIT;
```

The code for MS SQL Server looks like this.

MS SQL Solution 6.2.1.b (the fastest possible execution, incorrect results are possible)

```
SELECT @@SPID;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRANSACTION;
SELECT [sb_subscriber],
       COUNT([sb_book]) AS [sb_has_books]
FROM   [subscriptions]
WHERE  [sb_is_active] = 'Y'
GROUP BY [sb_subscriber];
COMMIT TRANSACTION;
```

MS SQL Solution 6.2.1.b (correct results, possible long execution)

```
SELECT @@SPID;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
SELECT [sb_subscriber],
       COUNT([sb_book]) AS [sb_has_books]
FROM   [subscriptions]
WHERE  [sb_is_active] = 'Y'
GROUP BY [sb_subscriber];
COMMIT TRANSACTION;
```

MS SQL Solution 6.2.1.b (test code)

```
SELECT @@SPID;
SET IMPLICIT_TRANSACTIONS ON;
BEGIN TRANSACTION;
UPDATE [subscriptions]
SET    [sb_is_active] =
       CASE
        WHEN [sb_is_active] = 'Y' THEN 'N'
        WHEN [sb_is_active] = 'N' THEN 'Y'
       END;
WAITFOR DELAY '00:00:10';
COMMIT TRANSACTION;
```

The code for Oracle looks like this.

Oracle Solution 6.2.1.b (the fastest possible execution, incorrect results are possible)

```
COMMIT;
SELECT SYS_CONTEXT('userenv','sessionid') FROM DUAL;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT "sb_subscriber",
       COUNT("sb_book") AS "sb_has_books"
FROM   "subscriptions"
WHERE  "sb_is_active" = 'Y'
GROUP BY "sb_subscriber";
COMMIT;
```

Oracle — Solution 6.2.1.b (correct results, possible long execution)

```
1    COMMIT;
2    SELECT SYS_CONTEXT('userenv','sessionid') FROM DUAL;
3    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
4    SELECT "sb_subscriber",
5           COUNT("sb_book") AS "sb_has_books"
6    FROM   "subscriptions"
7    WHERE  "sb_is_active" = 'Y'
8    GROUP BY "sb_subscriber";
9    COMMIT;
```

Oracle — Solution 6.2.1.b (test code)

```
1    COMMIT;
2    SELECT SYS_CONTEXT('userenv','sessionid') FROM DUAL;
3    UPDATE "subscriptions"
4    SET    "sb_is_active" =
5           CASE
6            WHEN "sb_is_active" = 'Y' THEN 'N'
7            WHEN "sb_is_active" = 'N' THEN 'Y'
8           END;
9    EXEC DBMS_LOCK.SLEEP(10);
10   COMMIT;
```

For all three DBMSes, the test code must be executed in a separate session (see explanations in the solution{443} of problem 6.2.1.a{443}), and the main code must be executed before, during and after the test code, allowing us to clearly see what data the DBMS will retrieve from the database and at what point in time.

Another DBMS behavior can be seen by replacing the last command in the test code from **`COMMIT`** to **`ROLLBACK`**.

Pay special attention to the difference between Oracle's behavior and that of MySQL and MS SQL Server: even in **`SERIALIZABLE`** mode, the query will return results without delay.

This completes the solution of this problem.

Task 6.2.1.TSK.A: create queries that, if executed in parallel, would provide the following effect:

- the first query should count the number of books taken from and returned to the library and not depend on queries to update the **`subscriptions`** table (no waiting for their completion);
- the second query should invert the values of the **`sb_is_active`** field of the **`subscriptions`** table from **`Y`** to **`N`** and vice versa and not depend on the first query (do not wait for its completion).

Task 6.2.1.TSK.B: create queries that, if executed in parallel, would provide the following effect:

- the first query should count the number of books taken from and returned to the library;
- the second query should invert the values of the **`sb_is_active`** field of the **`subscriptions`** table from **`Y`** to **`N`** and vice versa for readers with odd identifiers, then pause for ten seconds and undo this change (rollback the transaction).

Examine the behavior of all three DBMSes when the first query is executed before, during, and after the second query is finished, repeating this experiment for all transaction isolation levels supported by the particular DBMS.

6.2.2. Example 44: Concurrent Transactions Interaction

Problem 6.2.2.a{449}: demonstrate in all three DBMSes all concurrent access anomalies for all possible combinations of transaction isolation levels.

Problem 6.2.2.b{477}: demonstrate in all three DBMSes the situation of guaranteed transactions mutual deadlock and DBMS response to such a situation.

Expected result 6.2.2.a.

Since the solution of this problem is the expected result, see solution{449} 6.2.2.a.

Expected result 6.2.2.b.

Since the solution of this problem is the expected result, see solution{477} 6.2.2.b.

Solution 6.2.2.a{449}.

Concurrent access anomalies include:

- dirty read — reading the intermediate state of the data before the modifying transaction is committed or rolled back;
- lost update — modification of the same information by two or more transactions, in which the changes made by the transaction that was committed last take effect (and the changes made by other transactions are lost);
- non-repeatable read — obtaining different results for the same read request within a single transaction;
- phantom read — temporary appearance (disappearance) of some records in the data set, with which a transaction works, due to their change by another transaction.

For ease of navigation, here is a table showing the page numbers from which a particular anomaly begins to be considered in each DBMS.

	Dirty read	Lost update	Non-repeatable read	Phantom read
MySQL	{450}	{452}	{455}	{457}
MS SQL Server	{460}	{462}	{465}	{467}
Oracle	{470}	{472}	{474}	{476}

We also note that since the experiment logs will look the same in all DBMS, to save space we give them below only for MySQL; and within the exploration of each concurrent access anomaly for the first transaction we'll show only one level of isolation, and for the second transaction we'll show all levels of isolation supported by this DBMS.

Traditionally, we start with MySQL. This DBMS supports four levels of transaction isolation, combinations of which we will consider:

- `READ UNCOMMITTED`;
- `READ COMMITTED`;
- `REPEATABLE READ`;
- `SERIALIZABLE`.

To perform the experiment, we use such a batch file:

```
start cmd.exe /c "mysql -uUSER -pPASSWORD DATABASE < a.sql & pause"
start cmd.exe /c "mysql -uUSER -pPASSWORD DATABASE < b.sql & pause"
```

Dirty read in MySQL can be examined by executing the following code blocks in two separate sessions:

MySQL — Solution 6.2.2.a (code to investigate the dirty read anomaly)

```
-- Transaction A:
SELECT CONCAT('Tr A ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
START TRANSACTION;
SELECT CONCAT('Tr A START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';

SELECT SLEEP(5);


SELECT CONCAT('Tr A UPDATE: ',
              CURTIME());
UPDATE `subscriptions`
SET    `sb_is_active` =
 CASE
  WHEN `sb_is_active` = 'Y' THEN 'N'
  WHEN `sb_is_active` = 'N' THEN 'Y'
 END
WHERE  `sb_id` = 2;

SELECT SLEEP(20);


SELECT CONCAT('Tr A ROLLBACK: ',
              CURTIME());
ROLLBACK;
```

```
-- Transaction B:
SELECT CONCAT('Tr B ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
START TRANSACTION;
SELECT CONCAT('Tr B START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';
SELECT CONCAT('Tr B SELECT 1: ',
              CURTIME());
SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;


SELECT SLEEP(10);


SELECT CONCAT('Tr B SELECT 2: ',
              CURTIME());
SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;
SELECT CONCAT('Tr B COMMIT: ',
              CURTIME());
COMMIT;
```

Here is an example of the execution log of this code for a situation where transaction A is executed at the `READ UNCOMMITTED` isolation level and concurs to transaction B, sequentially executed at all MySQL-supported isolation levels.

A: READ UNCOMMITTED	B: READ UNCOMMITTED
Tr A ID = 13 Tr A START: 17:21:01 in READ-UNCOMMITTED Tr A UPDATE: 17:21:06 Tr A ROLLBACK: 17:21:26	Tr B ID = 14 Tr B START: 17:21:01 in READ-UNCOMMITTED Tr B SELECT 1: 17:21:01 sb_is_active = Y Tr B SELECT 2: 17:21:11 **sb_is_active = N** Tr B COMMIT: 17:21:11

A: READ UNCOMMITTED	B: READ COMMITTED
Tr A ID = 15 Tr A START: 17:38:06 in READ-UNCOMMITTED Tr A UPDATE: 17:38:12 Tr A ROLLBACK: 17:38:32	Tr B ID = 16 Tr B START: 17:38:06 in READ-COMMITTED Tr B SELECT 1: 17:38:06 sb_is_active = Y Tr B SELECT 2: 17:38:16 **sb_is_active = Y** Tr B COMMIT: 17:38:16

A: READ UNCOMMITTED	B: REPEATABLE READ
Tr A ID = 18 Tr A START: 17:42:37 in READ-UNCOMMITTED Tr A UPDATE: 17:42:42 Tr A ROLLBACK: 17:43:02	Tr B ID = 17 Tr B START: 17:42:37 in REPEATABLE-READ Tr B SELECT 1: 17:42:37 sb_is_active = Y Tr B SELECT 2: 17:42:47 **sb_is_active = Y** Tr B COMMIT: 17:42:47

A: READ UNCOMMITTED	B: SERIALIZABLE
Tr A ID = 20 Tr A START: 17:48:19 in READ-UNCOMMITTED Tr A UPDATE: 17:48:24 Tr A ROLLBACK: 17:48:49	Tr B ID = 19 Tr B START: 17:48:19 in SERIALIZABLE Tr B SELECT 1: 17:48:19 sb_is_active = Y Tr B SELECT 2: 17:48:29 **sb_is_active = Y** Tr B COMMIT: 17:48:29

The final results of transactions interaction are as follows.

		Transaction B isolation level			
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for transaction B finish
	READ COMMITTED	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for transaction B finish
	REPEATABLE READ	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for transaction B finish
	SERIALIZABLE	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for transaction B finish

Lost update in MySQL can be investigated by executing the following code blocks in two separate sessions:

MySQL — Solution 6.2.2.a (code to investigate the lost update anomaly)

```
-- Transaction A:
SELECT CONCAT('Tr A ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
START TRANSACTION;
SELECT CONCAT('Tr A START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';
SELECT CONCAT('Tr A, SELECT: ',
              CURTIME());
SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;


SELECT SLEEP(10);


SELECT CONCAT('Tr A UPDATE: ',
              CURTIME());
UPDATE `subscriptions`
SET    `sb_is_active` = 'Y'
WHERE  `sb_id` = 2;
SELECT CONCAT('Tr A COMMIT: ',
              CURTIME());
COMMIT;


SELECT SLEEP(10);


SELECT CONCAT('After A, SELECT: ',
              CURTIME());
SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;
```

```
-- Transaction B:
SELECT CONCAT('Tr B ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
START TRANSACTION;
SELECT CONCAT('Tr B START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';


SELECT SLEEP(5);


SELECT CONCAT('Tr B, SELECT: ',
              CURTIME());
SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;


SELECT SLEEP(10);


SELECT CONCAT('Tr B UPDATE: ',
              CURTIME());
UPDATE `subscriptions`
SET    `sb_is_active` = 'N'
WHERE  `sb_id` = 2;
SELECT CONCAT('Tr B COMMIT: ',
              CURTIME());
COMMIT;
SELECT CONCAT('After B, SELECT: ',
              CURTIME());
SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;
```

Here is an example of the execution log of this code for a situation where transaction A is executed at the **`READ COMMITTED`** isolation level and concurs to transaction B, sequentially executed at all MySQL-supported isolation levels.

A: READ COMMITTED	B: READ UNCOMMITTED
Tr A ID = 77	Tr B ID = 76
Tr A START: 19:12:19 in READ-COMMITTED	Tr B START: 19:12:19 in READ-UNCOMMITTED
Tr A, SELECT: 19:12:19	Tr B, SELECT: 19:12:24
sb_is_active = N	sb_is_active = N
Tr A UPDATE: 19:12:29	Tr B UPDATE: 19:12:34
Tr A COMMIT: 19:12:29	Tr B COMMIT: 19:12:34
After A, SELECT: 19:12:39	After B, SELECT: 19:12:34
sb_is_active = N	**sb_is_active = N**

A: READ COMMITTED	B: READ COMMITTED
Tr A ID = 80 Tr A START: 19:14:43 in READ-COMMITTED Tr A, SELECT: 19:14:43 sb_is_active = N Tr A UPDATE: 19:14:53 Tr A COMMIT: 19:14:53 After A, SELECT: 19:15:03 **sb_is_active = N**	Tr B ID = 79 Tr B START: 19:14:43 in READ-COMMITTED Tr B, SELECT: 19:14:48 sb_is_active = N Tr B UPDATE: 19:14:58 Tr B COMMIT: 19:14:58 After B, SELECT: 19:14:58 **sb_is_active = N**

A: READ COMMITTED	B: REPEATABLE READ
Tr A ID = 83 Tr A START: 19:17:00 in READ-COMMITTED Tr A, SELECT: 19:17:00 sb_is_active = N Tr A UPDATE: 19:17:10 Tr A COMMIT: 19:17:10 After A, SELECT: 19:17:20 **sb_is_active = N**	Tr B ID = 82 Tr B START: 19:17:00 in REPEATABLE-READ Tr B, SELECT: 19:17:05 sb_is_active = N Tr B UPDATE: 19:17:15 Tr B COMMIT: 19:17:15 After B, SELECT: 19:17:15 **sb_is_active = N**

A: READ COMMITTED	B: SERIALIZABLE
Tr A ID = 86 Tr A START: 19:19:07 in READ-COMMITTED Tr A, SELECT: 19:19:07 sb_is_active = N Tr A UPDATE: 19:19:17 Tr A COMMIT: 19:19:17 After A, SELECT: 19:19:27 **sb_is_active = N**	Tr B ID = 85 Tr B START: 19:19:06 in SERIALIZABLE Tr B, SELECT: 19:19:11 sb_is_active = N Tr B UPDATE: 19:19:21 Tr B COMMIT: 19:19:21 After B, SELECT: 19:19:21 **sb_is_active = N**

The final results of transactions interaction are as follows.

		Transaction B isolation level			
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost
	READ COMMITTED	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost
	REPEATABLE READ	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost
	SERIALIZABLE	Transaction A update is lost	Transaction A update is lost	Transaction A update is lost	Mutual deadlock with cancellation of transaction B is possible

If in this experiment we remove reading before updating (lines 15-19 for transaction A, and lines 20-24 for transaction B), then with any combination of isolation levels the result will be the same: the changes made by transaction A will be lost.

To get another option of DBMS behavior, we need to explicitly lock the records to be read (**`SELECT ... LOCK IN SHARE MODE`** or **`SELECT ... FOR UPDATE`**)[36] in the first read operation. This was not done in this case to demonstrate the most typical MySQL behavior. But if we add the specified locks, the behavior of MySQL will change to look like this.

[36] https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html

The final results of transaction interaction when using `LOCK IN SHARE MODE` for the first read operation. It is important to note that in some cases the deadlock disrupts both transactions, but in most cases the DBMS cancels transaction B, allowing transaction A to execute successfully.

		Transaction B isolation level			
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions
	READ COMMITTED	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions
	REPEATABLE READ	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions
	SERIALIZABLE	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions	Mutual deadlock of transactions

Final results of transaction interaction when using `FOR UPDATE` for the first read operation.

		Transaction B isolation level			
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A
	READ COMMITTED	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A
	REPEATABLE READ	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A
	SERIALIZABLE	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A	Transaction A update is lost, transaction B waits for the completion of A

Obviously, using different combinations of ways to execute (or not to execute at all) the first read operation at the beginning of each transaction we can get even more options of DBMS behavior.

Non-repeatable read in MySQL can be investigated by executing the following code blocks in two separate sessions:

MySQL — Solution 6.2.2.a (code to investigate the anomaly of non-repeatable read)

```
-- Transaction A:
SELECT CONCAT('Tr A ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
START TRANSACTION;
SELECT CONCAT('Tr A START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';


SELECT SLEEP(5);


SELECT CONCAT('Tr A UPDATE: ',
              CURTIME());
UPDATE `subscriptions`
SET    `sb_is_active` =
 CASE
  WHEN `sb_is_active` = 'Y' THEN 'N'
  WHEN `sb_is_active` = 'N' THEN 'Y'
 END
WHERE  `sb_id` = 2;
SELECT CONCAT('Tr A COMMIT: ',
              CURTIME());
COMMIT;
```

```
-- Transaction B:
SELECT CONCAT('Tr B ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
START TRANSACTION;
SELECT CONCAT('Tr B START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';
SELECT CONCAT('Tr A SELECT-1: ',
              CURTIME());

SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;




SELECT SLEEP(10);




SELECT CONCAT('Tr A SELECT-2: ',
              CURTIME());

SELECT `sb_is_active`
FROM   `subscriptions`
WHERE  `sb_id` = 2;
SELECT CONCAT('Tr B COMMIT: ',
              CURTIME());
COMMIT;
```

Here is an example of the execution log of this code for a situation where transaction A is executed at the **`REPEATABLE READ`** isolation level and concurs to transaction B, sequentially executed at all MySQL-supported isolation levels.

A: REPEATABLE READ	B: READ UNCOMMITTED
Tr A ID = 151 Tr A START: 20:24:12 in REPEATABLE-READ Tr A UPDATE: 20:24:17 Tr A COMMIT: 20:24:17	Tr B ID = 152 Tr B START: 20:24:12 in READ-UNCOMMITTED Tr B SELECT-1: 20:24:12 **sb_is_active = N** Tr B SELECT-2: 20:24:22 **sb_is_active = Y** Tr B COMMIT: 20:24:22

A: REPEATABLE READ	B: READ COMMITTED
Tr A ID = 153 Tr A START: 20:25:29 in REPEATABLE-READ Tr A UPDATE: 20:25:34 Tr A COMMIT: 20:25:34	Tr B ID = 154 Tr B START: 20:25:29 in READ-COMMITTED Tr B SELECT-1: 20:25:29 **sb_is_active = Y** Tr B SELECT-2: 20:25:39 **sb_is_active = N** Tr B COMMIT: 20:25:39

A: REPEATABLE READ	B: REPEATABLE READ
Tr A ID = 156 Tr A START: 20:26:24 in REPEATABLE-READ Tr A UPDATE: 20:26:29 Tr A COMMIT: 20:26:29	Tr B ID = 155 Tr B START: 20:26:24 in REPEATABLE-READ Tr B SELECT-1: 20:26:24 **sb_is_active = N** Tr B SELECT-2: 20:26:34 **sb_is_active = N** Tr B COMMIT: 20:26:34

A: REPEATABLE READ	B: SERIALIZABLE
Tr A ID = 157 Tr A START: 20:27:15 in REPEATABLE-READ Tr A UPDATE: 20:27:20 Tr A COMMIT: 20:27:25	Tr B ID = 158 Tr B START: 20:27:15 in SERIALIZABLE Tr B SELECT-1: 20:27:15 **sb_is_active = Y** Tr B SELECT-2: 20:27:25 **sb_is_active = Y** Tr B COMMIT: 20:27:25

The final results of transactions interaction are as follows.

		Transaction B isolation level			
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	READ COMMITTED	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	REPEATABLE READ	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	SERIALIZABLE	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish

To get another option of DBMS behavior, we need to explicitly lock the records to be read (**`SELECT ... LOCK IN SHARE MODE`** or **`SELECT ... FOR UPDATE`**)[37] in the first read operation. This was not done in this case to demonstrate the most typical MySQL behavior. You are encouraged to check the other cases of DBMS behavior on your own in task 6.2.2.TSK.D{479}.

[37] https://dev.mysql.com/doc/refman/8.0/en/innodb-locking-reads.html

Phantom read in MySQL can be investigated by executing the following code blocks in two separate sessions:

MySQL — Solution 6.2.2.a (code to investigate phantom read anomaly)

```
-- Transaction A:
SELECT CONCAT('Tr A ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
```

```
-- Transaction B:
SELECT CONCAT('Tr B ID = ',
              CONNECTION_ID());
SET autocommit = 0;
SET SESSION TRANSACTION
ISOLATION LEVEL {LEVEL};
```

```
START TRANSACTION;
SELECT CONCAT('Tr A START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';
```

```
START TRANSACTION;
SELECT CONCAT('Tr B START: ',
              CURTIME(), ' in ');
SELECT `VARIABLE_VALUE`
FROM   `information_schema`.
       `session_variables`
WHERE  `VARIABLE_NAME` =
       'tx_isolation';
```

```

SELECT SLEEP(5);

```

```
SELECT CONCAT('Tr B COUNT-1: ',
              CURTIME());
SELECT COUNT(*)
FROM   `subscriptions`
WHERE  `sb_id` > 500;
```

```
SELECT CONCAT('Tr A INSERT: ',
              CURTIME());
INSERT INTO `subscriptions`
           (`sb_id`,
            `sb_subscriber`,
            `sb_book`,
            `sb_start`,
            `sb_finish`,
            `sb_is_active`)
     VALUES (1000,
             1,
             1,
             '2025-01-12',
             '2026-02-12',
             'N');
```

```

SELECT SLEEP(10);
```

```
SELECT CONCAT('Tr B COUNT-2: ',
              CURTIME());
SELECT COUNT(*)
FROM   `subscriptions`
WHERE  `sb_id` > 500;
```

```
SELECT CONCAT('Tr A ROLLBACK: ',
              CURTIME());
ROLLBACK;
```

```
SELECT SLEEP(15);
```

```

```

```
SELECT CONCAT('Tr B COUNT-3: ',
              CURTIME());
SELECT COUNT(*)
FROM   `subscriptions`
WHERE  `sb_id` > 500;
SELECT CONCAT('Tr B COMMIT: ',
              CURTIME());
COMMIT;
```

Here is an example of the execution log of this code for a situation where transaction A is executed at the **`SERIALIZABLE`** isolation level and concurs to transaction B, sequentially executed at all MySQL-supported isolation levels.

A: SERIALIZABLE	B: READ UNCOMMITTED
Tr A ID = 198 Tr A START: 21:08:32 in SERIALIZABLE Tr A INSERT: 21:08:37 Tr A ROLLBACK: 21:08:47	Tr B ID = 197 Tr B START: 21:08:32 in READ-UNCOMMITTED Tr B COUNT-1: 21:08:32 **COUNT(*) = 0** Tr B COUNT-2: 21:08:42 **COUNT(*) = 1** Tr B COUNT-3: 21:08:57 **COUNT(*) = 0** Tr B COMMIT: 21:08:57

A: SERIALIZABLE	B: READ COMMITTED
Tr A ID = 200 Tr A START: 21:09:34 in SERIALIZABLE Tr A INSERT: 21:09:39 Tr A ROLLBACK: 21:09:49	Tr B ID = 199 Tr B START: 21:09:34 in READ-COMMITTED Tr B COUNT-1: 21:09:34 **COUNT(*) = 0** Tr B COUNT-2: 21:09:44 **COUNT(*) = 0** Tr B COUNT-3: 21:09:59 **COUNT(*) = 0** Tr B COMMIT: 21:09:59

A: SERIALIZABLE	B: REPEATABLE READ
Tr A ID = 201 Tr A START: 21:10:28 in SERIALIZABLE Tr A INSERT: 21:10:33 Tr A ROLLBACK: 21:10:43	Tr B ID = 202 Tr B START: 21:10:28 in REPEATABLE-READ Tr B COUNT-1: 21:10:28 **COUNT(*) = 0** Tr B COUNT-2: 21:10:38 **COUNT(*) = 0** Tr B COUNT-3: 21:10:53 **COUNT(*) = 0** Tr B COMMIT: 21:10:53

A: SERIALIZABLE	B: SERIALIZABLE
Tr A ID = 203 Tr A START: 21:11:29 in SERIALIZABLE Tr A INSERT: 21:11:34 Tr A ROLLBACK: 21:11:44	Tr B ID = 204 Tr B START: 21:11:29 in SERIALIZABLE Tr B COUNT-1: 21:11:29 **COUNT(*) = 0** Tr B COUNT-2: 21:11:39 **COUNT(*) = 0** Tr B COUNT-3: 21:11:**59** **COUNT(*) = 0** Tr B COMMIT: 21:11:59

The final results of transactions interaction are as follows.

		Transaction B isolation level			
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction B manages to process the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”, COUNT-3 waits for transaction A finish
	READ COMMITTED	Transaction B manages to process the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”, COUNT-3 waits for transaction A finish
	REPEATABLE READ	Transaction B manages to process the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”, COUNT-3 waits for transaction A finish
	SERIALIZABLE	Transaction B manages to process the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”	Transaction B does not access the “phantom record”, COUNT-3 waits for transaction A finish

This completes the MySQL solution.

Let's move on to MS SQL Server. This DBMS supports five levels of transaction isolation, combinations of which we will consider:

- `READ UNCOMMITTED`;
- `READ COMMITTED`;
- `REPEATABLE READ`;
- `SNAPSHOT`;
- `SERIALIZABLE`.

To perform the experiment, we use such a batch file:

```
start cmd.exe /c "sqlcmd -S COMPUTER\SERVER -i a.sql & pause"
start cmd.exe /c "sqlcmd -S COMPUTER\SERVER -i b.sql & pause"
```

To simplify the code of the following queries, let's create the functions `GET_ISOLATION_LEVEL` and `GET_CT`, which return, respectively, the current transaction isolation level and the current time value.

MS SQL Solution 6.2.2.a (code and query to check the functionality of the service functions)

```
1   CREATE FUNCTION GET_ISOLATION_LEVEL()
2   RETURNS NVARCHAR(50)
3   BEGIN
4    DECLARE @IsolationLevel NVARCHAR(50);
5    SET @IsolationLevel = (
6    SELECT CASE [transaction_isolation_level]
7    WHEN 0 THEN 'Unspecified'
8    WHEN 1 THEN 'Read Uncommitted'
9    WHEN 2 THEN 'Read Committed'
10   WHEN 3 THEN 'Repeatable Read'
11   WHEN 4 THEN 'Serializable'
12   WHEN 5 THEN 'Snapshot' END AS TRANSACTION_ISOLATION_LEVEL
13   FROM [sys].[dm_exec_sessions]
14   WHERE [session_id] = @@SPID);
15   RETURN @IsolationLevel;
16  END;
17  GO
18
19  CREATE FUNCTION GET_CT()
20  RETURNS NVARCHAR(50)
21  BEGIN
22   DECLARE @CT NVARCHAR(50);
23   SET @CT = CONVERT(NVARCHAR(12), GETDATE(), 114);
24   RETURN @CT;
25  END;
26  GO
27
28  PRINT dbo.GET_ISOLATION_LEVEL();
29  PRINT dbo.GET_CT();
```

Note two important points:

- to ensure that the `SNAPSHOT` transaction isolation level works, it is necessary to run the command `ALTER DATABASE [database_name] SET ALLOW_SNAPSHOT_ISOLATION ON`;
- in the code below we will commit each transaction twice to show the current transaction nesting level (`@@TRANCOUNT`), which is caused by a feature of MS SQL Server in `IMPLICIT_TRANSACTIONS ON` mode: in this mode, the beginning of a transaction sets `@@TRANCOUNT` to 2 instead of 1.

Dirty read in MS SQL Server can be investigated by executing the following code blocks in two separate sessions:

MS SQL — Solution 6.2.2.a (code to investigate the dirty read anomaly)

1	`-- Transaction A:`	`-- Transaction B:`
2	`PRINT CONCAT('Tr A ID = ', @@SPID);`	`PRINT CONCAT('Tr B ID = ', @@SPID);`
3	`SET IMPLICIT_TRANSACTIONS ON;`	`SET IMPLICIT_TRANSACTIONS ON;`
4	`SET TRANSACTION ISOLATION`	`SET TRANSACTION ISOLATION`
5	`LEVEL {LEVEL};`	`LEVEL {LEVEL};`
6	`BEGIN TRANSACTION;`	`BEGIN TRANSACTION;`
7	`PRINT CONCAT('Tr A START: ',`	`PRINT CONCAT('Tr B START: ',`
8	`dbo.GET_CT(), ' in ',`	`dbo.GET_CT(), ' in ',`
9	`dbo.GET_ISOLATION_LEVEL());`	`dbo.GET_ISOLATION_LEVEL());`
10		`PRINT CONCAT('Tr B SELECT-1: ',`
11		`dbo.GET_CT());`
12	`WAITFOR DELAY '00:00:05';`	`SELECT [sb_is_active]`
13		`FROM [subscriptions]`
14		`WHERE [sb_id] = 2;`
15	`PRINT CONCAT('Tr A UPDATE: ',`	
16	`dbo.GET_CT());`	
17	`UPDATE [subscriptions]`	
18	`SET [sb_is_active] =`	`WAITFOR DELAY '00:00:10';`
19	`CASE`	
20	`WHEN [sb_is_active] = 'Y' THEN 'N'`	
21	`WHEN [sb_is_active] = 'N' THEN 'Y'`	
22	`END`	
23	`WHERE [sb_id] = 2;`	
24		`PRINT CONCAT('Tr B SELECT-2: ',`
25		`dbo.GET_CT());`
26		`SELECT [sb_is_active]`
27		`FROM [subscriptions]`
28	`WAITFOR DELAY '00:00:20';`	`WHERE [sb_id] = 2;`
29		`PRINT CONCAT('Tr B COMMIT: ',`
30		`dbo.GET_CT());`
31		`COMMIT;`
32		`PRINT CONCAT('TrC = ', @@TRANCOUNT);`
33		`COMMIT;`
34		`PRINT CONCAT('TrC = ', @@TRANCOUNT);`
35	`PRINT CONCAT('Tr A ROLLBACK: ',`	
36	`dbo.GET_CT());`	
37	`ROLLBACK;`	
38	`PRINT CONCAT('TrC = ', @@TRANCOUNT);`	

The final results of transactions interaction are as follows.

		Transaction B isolation level				
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SNAPSHOT	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value, SELECT-2 in transaction B waits for A to finish	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish
	READ COMMITTED	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value, SELECT-2 in transaction B waits for A to finish	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish
	REPEATABLE READ	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value, SELECT-2 in transaction B waits for A to finish	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish
	SNAPSHOT	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value, SELECT-2 in transaction B waits for A to finish	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish
	SERIALIZABLE	Transaction B manages to read an uncommitted value	Transaction B both times reads the original (correct) value, SELECT-2 in transaction B waits for A to finish	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value, UPDATE in transaction A waits for B to finish

Lost update in MS SQL Server can be investigated by executing the following code blocks in two separate sessions:

MS SQL — Solution 6.2.2.a (code to investigate the lost update anomaly)

```
-- Transaction A:
PRINT CONCAT('Tr A ID = ', @@SPID);
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION
LEVEL {LEVEL};
BEGIN TRANSACTION;
PRINT CONCAT('Tr A START: ',
dbo.GET_CT(), ' in ',
dbo.GET_ISOLATION_LEVEL());
PRINT CONCAT('Tr A SELECT: ',
dbo.GET_CT());
SELECT [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] = 2;


WAITFOR DELAY '00:00:10';


PRINT CONCAT('Tr A UPDATE: ',
dbo.GET_CT());
UPDATE [subscriptions]
SET    [sb_is_active] = 'Y'
WHERE  [sb_id] = 2;
PRINT CONCAT('Tr A COMMIT: ',
dbo.GET_CT());
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);


WAITFOR DELAY '00:00:10';


PRINT CONCAT('Tr A SELECT AFTER: ',
dbo.GET_CT());
SELECT [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] = 2;
```

```
-- Transaction B:
PRINT CONCAT('Tr B ID = ', @@SPID);
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION
LEVEL {LEVEL};
BEGIN TRANSACTION;
PRINT CONCAT('Tr B START: ',
dbo.GET_CT(), ' in ',
dbo.GET_ISOLATION_LEVEL());


WAITFOR DELAY '00:00:05';


PRINT CONCAT('Tr B SELECT: ',
dbo.GET_CT());
SELECT [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] = 2;


WAITFOR DELAY '00:00:10';


PRINT CONCAT('Tr B UPDATE: ',
dbo.GET_CT());
UPDATE [subscriptions]
SET    [sb_is_active] = 'N'
WHERE  [sb_id] = 2;
PRINT CONCAT('Tr B COMMIT: ',
dbo.GET_CT());
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);


PRINT CONCAT('Tr B SELECT AFTER: ',
dbo.GET_CT());
SELECT [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] = 2;
```

The final results of transactions interaction are as follows.

		Transaction B isolation level				
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SNAPSHOT	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction A update is lost	Transaction A update is lost	Transaction A update stays effective, UPDATE in transaction A waits for B to finish	Transaction A update stays effective, transaction B aborted (attempt to update a locked record in SNAPSHOT mode)	Transaction A update stays effective, UPDATE in transaction A waits for B to finish
	READ COMMITTED	Transaction A update is lost	Transaction A update is lost	Transaction A update stays effective, UPDATE in transaction A waits for B to finish	Transaction A update stays effective, transaction B aborted (attempt to update a locked record in SNAPSHOT mode)	Transaction A update stays effective, UPDATE in transaction A waits for B to finish
	REPEATABLE READ	Transaction A update is lost	Transaction A update is lost	Mutual deadlock of transactions	Transaction A update stays effective, transaction B aborted (attempt to update a locked record in SNAPSHOT mode)	Mutual deadlock of transactions
	SNAPSHOT	Transaction A update is lost	Transaction A update is lost	Transaction A aborted (attempt to update a locked record in SNAPSHOT mode)	Transaction A update stays effective, transaction B aborted (attempt to update a locked record in SNAPSHOT mode)	Transaction A aborted (attempt to update a locked record in SNAPSHOT mode)
	SERIALIZABLE	Transaction A update is lost, COMMIT in transaction A waits for B to finish	Transaction A update is lost, COMMIT in transaction A waits for B to finish	Mutual deadlock of transactions	Transaction A update stays effective, transaction B aborted (attempt to update a locked record in SNAPSHOT mode)	Mutual deadlock of transactions

For educational purposes, consider what would happen if we “forgot” to add a second `COMMIT` to the code of both transactions (see details in the solution{423} of problem 6.1.1.a{423}).

The final **erroneous** transactions interaction results would be as follows.

		Transaction B isolation level				
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SNAPSHOT	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost
	READ COMMITTED	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost
	REPEATABLE READ	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Mutual deadlock of transactions
	SNAPSHOT	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost
	SERIALIZABLE	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Mutual deadlock of transactions	Transaction A update is lost, UPDATE in transaction B waits for session of A to finish	Mutual deadlock of transactions

Note the wording “UPDATE in transaction B waits for session of A to finish”. This means the whole session of interaction with the DBMS, not just a transaction. Because of the “forgotten” `COMMIT`, both transactions are actually terminated at the moment of closing the session with the DBMS.

To get another option of DBMS behavior, it is necessary to explicitly lock the records to be read (`UPDLOCK`) in the first operation (read). In this case, it was not done to demonstrate the most typical MS SQL Server behavior. To check the other cases of DBMS behavior you are encouraged to do it yourself in the task 6.2.2.TSK.E{479}.

Non-repeatable read in MS SQL Server can be investigated by executing the following code blocks in two separate sessions:

MS SQL — Solution 6.2.2.a (code to investigate the anomaly of non-repeatable read)

```
-- Transaction A:
PRINT CONCAT('Tr A ID = ', @@SPID);
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION
LEVEL {LEVEL};
BEGIN TRANSACTION;
PRINT CONCAT('Tr A START: ',
dbo.GET_CT(), ' in ',
dbo.GET_ISOLATION_LEVEL());


WAITFOR DELAY '00:00:05';


PRINT CONCAT('Tr A UPDATE: ',
dbo.GET_CT());
UPDATE [subscriptions]
SET    [sb_is_active] =
 CASE
  WHEN [sb_is_active] = 'Y' THEN 'N'
  WHEN [sb_is_active] = 'N' THEN 'Y'
 END
WHERE  [sb_id] = 2;
PRINT CONCAT('Tr A COMMIT: ',
dbo.GET_CT());
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
```

```
-- Transaction B:
PRINT CONCAT('Tr B ID = ', @@SPID);
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION
LEVEL {LEVEL};
BEGIN TRANSACTION;
PRINT CONCAT('Tr B START: ',
dbo.GET_CT(), ' in ',
dbo.GET_ISOLATION_LEVEL());
PRINT CONCAT('Tr B SELECT-1: ',
dbo.GET_CT());
SELECT [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] = 2;


WAITFOR DELAY '00:00:10';


PRINT CONCAT('Tr B SELECT-2: ',
dbo.GET_CT());
SELECT [sb_is_active]
FROM   [subscriptions]
WHERE  [sb_id] = 2;
PRINT CONCAT('Tr B COMMIT: ',
dbo.GET_CT());
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
```

The final results of transactions interaction are as follows.

		Transaction B isolation level				
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SNAPSHOT	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data, transaction A waits for B to finish	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	READ COMMITTED	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data, transaction A waits for B to finish	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	REPEATABLE READ	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data, transaction A waits for B to finish	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	SNAPSHOT	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data, transaction A waits for B to finish	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish
	SERIALIZABLE	The first and second SELECT returned different data	The first and second SELECT returned different data	The first and second SELECT returned the same data, transaction A waits for B to finish	The first and second SELECT returned the same data	The first and second SELECT returned the same data, transaction A waits for B to finish

To get another option of DBMS behavior, it is necessary to explicitly lock the records to be read (`UPDLOCK`) in the first read operation. In this case, it was not done to demonstrate the most typical MS SQL Server behavior. To check the other cases of DBMS behavior you are encouraged to do it yourself in the task 6.2.2.TSK.E{479}.

Phantom read in MS SQL Server can be investigated by executing the following code blocks in two separate sessions:

MS SQL — Solution 6.2.2.a (code to investigate phantom read anomaly)

```
-- Transaction A:
PRINT CONCAT('Tr A ID = ', @@SPID);
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION
LEVEL {LEVEL};
BEGIN TRANSACTION;
PRINT CONCAT('Tr A START: ',
dbo.GET_CT(), ' in ',
dbo.GET_ISOLATION_LEVEL());


WAITFOR DELAY '00:00:05';


PRINT CONCAT('Tr A INSERT: ',
dbo.GET_CT());

SET IDENTITY_INSERT
[subscriptions] ON;

INSERT INTO [subscriptions]
            ([sb_id],
             [sb_subscriber],
             [sb_book],
             [sb_start],
             [sb_finish],
             [sb_is_active])
     VALUES (1000,
             1,
             1,
             '2025-01-12',
             '2026-02-12',
             'N');

SET IDENTITY_INSERT
[subscriptions] OFF;


WAITFOR DELAY '00:00:10';


PRINT CONCAT('Tr A ROLLBACK: ',
dbo.GET_CT());
ROLLBACK;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
```

```
-- Transaction B:
PRINT CONCAT('Tr B ID = ', @@SPID);
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION
LEVEL {LEVEL};
BEGIN TRANSACTION;
PRINT CONCAT('Tr B START: ',
dbo.GET_CT(), ' in ',
dbo.GET_ISOLATION_LEVEL());
PRINT CONCAT('Tr B COUNT-1: ',
dbo.GET_CT());
SELECT COUNT(*)
FROM   [subscriptions]
WHERE  [sb_id] > 500;


WAITFOR DELAY '00:00:10';


PRINT CONCAT('Tr B COUNT-2: ',
dbo.GET_CT());

SELECT COUNT(*)
FROM   [subscriptions]
WHERE  [sb_id] > 500;

WAITFOR DELAY '00:00:15';


PRINT CONCAT('Tr B COUNT-3: ',
dbo.GET_CT());
SELECT COUNT(*)
FROM   [subscriptions]
WHERE  [sb_id] > 500;
PRINT CONCAT('Tr B COMMIT: ',
dbo.GET_CT());
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
COMMIT;
PRINT CONCAT('TrC = ', @@TRANCOUNT);
```

The final results of transactions interaction are as follows.

		Transaction B isolation level				
		READ UNCOMMITTED	READ COMMITTED	REPEATABLE READ	SNAPSHOT	SERIALIZABLE
Transaction A isolation level	READ UNCOMMITTED	Transaction B manages to process the "phantom record"	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record", INSERT in transaction A waits for B to finish
	READ COMMITTED	Transaction B manages to process the "phantom record"	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record", INSERT in transaction A waits for B to finish
	REPEATABLE READ	Transaction B manages to process the "phantom record", INSERT in transaction A waits for B to finish	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record", INSERT in transaction A waits for B to finish
	SNAPSHOT	Transaction B manages to process the "phantom record"	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record", INSERT in transaction A waits for B to finish
	SERIALIZABLE	Transaction B manages to process the "phantom record"	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record", its COUNT-2 waits for A to finish	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record", INSERT in transaction A waits for B to finish

This completes the solution for MS SQL Server.

Let's move on to Oracle. This DBMS supports two transaction isolation levels and two transaction modes, the combinations of which we will consider:

- Isolation levels:
  - **`READ COMMITTED`**;
  - **`SERIALIZABLE`**.
- Modes:
  - **`READ ONLY`** (in fact, this is also the isolation level equivalent to **`REPEATABLE READ`** and/or **`SERIALIZABLE`** in other DBMSes);
  - **`READ WRITE`**.

To perform the experiment, we use such a batch file:

```
start cmd.exe /c "echo exit | sqlplus USER/PASSWORD@COMPUTER @a.sql & pause"
start cmd.exe /c "echo exit | sqlplus USER/PASSWORD@COMPUTER @b.sql & pause"
```

To simplify the code of the following queries, let's create the functions **GET_IDS_AND_ISOLATION_LEVEL** and **GET_CT**, which return, respectively, the value of the transaction parameters and the current time value.

Oracle — Solution 6.2.2.a (code and query to check the functionality of the service functions)

```
1   CREATE FUNCTION GET_IDS_AND_ISOLATION_LEVEL
2    RETURN NVARCHAR2
3    IS
4    session_id NUMBER(10);
5    session_sn NUMBER(10);
6    session_isolation_level NVARCHAR2(150);
7   BEGIN
8    SELECT "session".sid AS "session_id",
9           "session".serial# AS "session_sn",
10          CASE BITAND("transaction".flag, POWER(2, 28))
11            WHEN 0 THEN 'READ COMMITTED'
12            ELSE 'SERIALIZABLE'
13          END AS "session_isolation_level"
14   INTO   session_id,
15          session_sn,
16          session_isolation_level
17   FROM   v$transaction "transaction"
18          JOIN v$session "session"
19            ON "transaction".addr = "session".taddr
20               AND "session".sid = SYS_CONTEXT('USERENV', 'SID');
21
22   RETURN 'ID = ' || session_id || ', SN = ' || session_sn || ', in ' ||
23          session_isolation_level;
24  END;
25
26  CREATE FUNCTION GET_CT
27   RETURN NVARCHAR2
28   IS
29  BEGIN
30   RETURN TO_CHAR(SYSTIMESTAMP,'HH24:MI:SS.FF');
31  END;
32
33  SELECT GET_CT FROM DUAL;
34  SELECT GET_IDS_AND_ISOLATION_LEVEL FROM DUAL;
```

Unfortunately, there is no documented way in Oracle to distinguish between **READ ONLY** and **READ WRITE** transactions (do not confuse this with the typical incorrect solution **SELECT OPEN_MODE FROM V$DATABASE**, such a solution does not refer to the transaction mode!), nor is there a way to combine these parameters with the specification of isolation level. And if the second problem can be avoided by setting the session and transaction parameters separately, nothing can be done about the first problem (yet?).

Since Oracle (like MySQL[38]) does not support nested transactions, here (unlike with MS SQL Server) we will not monitor the current transaction level.

[38] There are ways to get behavior in MySQL that is "similar" to nested transactions (see https://www.burnison.ca/notes/fun-mysql-fact-of-the-day-no-nested-transactions), but it is still exactly an emulation.

Dirty read in Oracle does not exist, but we can check it by executing the following code blocks in two separate sessions:

Oracle — Solution 6.2.2.a (code to investigate the dirty read anomaly)

```
-- Transaction A:
ALTER SESSION SET
ISOLATION_LEVEL = {LEVEL};
SET TRANSACTION {MODE};
SELECT 'Tr A: ' ||
GET_IDS_AND_ISOLATION_LEVEL
FROM DUAL;
SELECT 'Tr A START: ' ||
GET_CT FROM DUAL;
```

```
-- Transaction B:
ALTER SESSION SET
ISOLATION_LEVEL = {LEVEL};
SET TRANSACTION {MODE};
SELECT 'Tr B: ' ||
GET_IDS_AND_ISOLATION_LEVEL
FROM DUAL;
SELECT 'Tr B START: ' ||
GET_CT FROM DUAL;
```

```

EXEC DBMS_LOCK.SLEEP(5);
```

```
SELECT 'Tr B SELECT-1: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;
```

```
SELECT 'Tr A UPDATE: ' ||
GET_CT FROM DUAL;
UPDATE "subscriptions"
SET    "sb_is_active" =
 CASE
  WHEN "sb_is_active" = 'Y' THEN 'N'
  WHEN "sb_is_active" = 'N' THEN 'Y'
 END
WHERE  "sb_id" = 2;
```

```

EXEC DBMS_LOCK.SLEEP(10);
```

```


EXEC DBMS_LOCK.SLEEP(20);


```

```
SELECT 'Tr B SELECT-2: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;
SELECT 'Tr B COMMIT: ' ||
GET_CT FROM DUAL;
COMMIT;
```

```
SELECT 'Tr A ROLLBACK: ' ||
GET_CT FROM DUAL;
ROLLBACK;
```

The final results of transactions interaction are as follows.

			Transaction B isolation level			
			READ COMMITTED		SERIALIZABLE	
			READ ONLY	READ WRITE	READ ONLY	READ WRITE
Transaction A isolation level	READ COMMITTED	READ ONLY	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)
		READ WRITE	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value
	SERIALIZABLE	READ ONLY	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)	Transaction B both times reads the original (correct) value, UPDATE in transaction A is forbidden (R/O)
		READ WRITE	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value	Transaction B both times reads the original (correct) value

Lost update in Oracle can be investigated by executing the following code blocks in two separate sessions:

Oracle — Solution 6.2.2.a (code to investigate the lost update anomaly)

```
-- Transaction A:
ALTER SESSION SET
ISOLATION_LEVEL = {LEVEL};
SET TRANSACTION {MODE};
SELECT 'Tr A: ' ||
GET_IDS_AND_ISOLATION_LEVEL
FROM DUAL;
SELECT 'Tr A START: ' ||
GET_CT FROM DUAL;
SELECT 'Tr A SELECT: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;


EXEC DBMS_LOCK.SLEEP(10);


SELECT 'Tr A UPDATE: ' ||
GET_CT FROM DUAL;

UPDATE "subscriptions"
SET    "sb_is_active" = 'Y'
WHERE  "sb_id" = 2;
SELECT 'Tr A COMMIT: ' ||
GET_CT FROM DUAL;
COMMIT;


EXEC DBMS_LOCK.SLEEP(10);


SELECT 'Tr A SELECT AFTER: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;
```

```
-- Transaction B:
ALTER SESSION SET
ISOLATION_LEVEL = {LEVEL};
SET TRANSACTION {MODE};
SELECT 'Tr B: ' ||
GET_IDS_AND_ISOLATION_LEVEL
FROM DUAL;
SELECT 'Tr B START: ' ||
GET_CT FROM DUAL;


EXEC DBMS_LOCK.SLEEP(5);


SELECT 'Tr B SELECT: ' ||
GET_CT FROM DUAL;

SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;


EXEC DBMS_LOCK.SLEEP(10);


SELECT 'Tr B UPDATE: ' ||
GET_CT FROM DUAL;
UPDATE "subscriptions"
SET    "sb_is_active" = 'N'
WHERE  "sb_id" = 2;
SELECT 'Tr B COMMIT: ' ||
GET_CT FROM DUAL;
COMMIT;


SELECT 'Tr B SELECT AFTER: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;
```

The final results of transactions interaction are as follows.

			Transaction B isolation level			
			READ COMMITTED		SERIALIZABLE	
			READ ONLY	READ WRITE	READ ONLY	READ WRITE
Transaction A isolation level	READ COMMITTED	READ ONLY	UPDATE in both transactions is forbidden (R/O)	UPDATE in transaction A is forbidden (R/O)	UPDATE in both transactions is forbidden (R/O)	UPDATE in transaction A is forbidden (R/O)
		READ WRITE	UPDATE in transaction B is forbidden (R/O)	Transaction A update is lost	UPDATE in transaction B is forbidden (R/O)	Transaction A update stays effective, UPDATE in transaction B failed (see below)
	SERIALIZABLE	READ ONLY	UPDATE in both transactions is forbidden (R/O)	UPDATE in transaction A is forbidden (R/O)	UPDATE in both transactions is forbidden (R/O)	UPDATE in transaction A is forbidden (R/O)
		READ WRITE	UPDATE in transaction B is forbidden (R/O)	Transaction A update is lost	UPDATE in transaction B is forbidden (R/O)	Transaction A update stays effective, UPDATE in transaction B failed (see below)

In transaction B, `UPDATE` causes an error “ORA-08177: can't serialize access for this transaction”, because the corresponding record is locked by transaction A.

Non-repeatable read in Oracle can be investigated by executing the following code blocks in two separate sessions:

Oracle — Solution 6.2.2.a (code to investigate the anomaly of non-repeatable read)

```
1    -- Transaction A:
2    ALTER SESSION SET
3    ISOLATION_LEVEL = {LEVEL};
4    SET TRANSACTION {MODE};
5    SELECT 'Tr A: ' ||
6    GET_IDS_AND_ISOLATION_LEVEL
7    FROM DUAL;
8    SELECT 'Tr A START: ' ||
9    GET_CT FROM DUAL;
10
11
12   EXEC DBMS_LOCK.SLEEP(5);
13
14
15   SELECT 'Tr A UPDATE: ' ||
16   GET_CT FROM DUAL;
17   UPDATE "subscriptions"
18   SET    "sb_is_active" =
19    CASE
20     WHEN "sb_is_active" = 'Y' THEN 'N'
21     WHEN "sb_is_active" = 'N' THEN 'Y'
22    END
23   WHERE  "sb_id" = 2;
24   SELECT 'Tr A COMMIT: ' ||
25   GET_CT FROM DUAL;
26   COMMIT;
27
28
29
30
31
32
33
34
```

```
-- Transaction B:
ALTER SESSION SET
ISOLATION_LEVEL = {LEVEL};
SET TRANSACTION {MODE};
SELECT 'Tr B: ' ||
GET_IDS_AND_ISOLATION_LEVEL
FROM DUAL;
SELECT 'Tr B START: ' ||
GET_CT FROM DUAL;
SELECT 'Tr B SELECT-1: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;


EXEC DBMS_LOCK.SLEEP(10);


SELECT 'Tr B SELECT-2: ' ||
GET_CT FROM DUAL;
SELECT "sb_is_active"
FROM   "subscriptions"
WHERE  "sb_id" = 2;
SELECT 'Tr B COMMIT: ' ||
GET_CT FROM DUAL;
COMMIT;
```

The final results of transactions interaction are as follows.

			Transaction B isolation level			
			READ COMMITTED		SERIALIZABLE	
			READ ONLY	READ WRITE	READ ONLY	READ WRITE
Transaction A isolation level	READ COMMITTED	READ ONLY	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)
		READ WRITE	The first and second SELECT returned the same data	The first and second SELECT returned different data	The first and second SELECT returned the same data	The first and second SELECT returned the same data
	SERIALIZABLE	READ ONLY	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)	The first and second SELECT returned the same data, UPDATE in transaction A is forbidden (R/O)
		READ WRITE	The first and second SELECT returned the same data	The first and second SELECT returned different data	The first and second SELECT returned the same data	The first and second SELECT returned the same data

Phantom read in Oracle can be investigated by executing the following code blocks in two separate sessions:

Oracle — Solution 6.2.2.a (code to investigate phantom read anomaly)

-- Transaction A: ALTER SESSION SET ISOLATION_LEVEL = {LEVEL}; SET TRANSACTION {MODE}; SELECT 'Tr A: ' || GET_IDS_AND_ISOLATION_LEVEL FROM DUAL; SELECT 'Tr A START: ' || GET_CT FROM DUAL;	-- Transaction B: ALTER SESSION SET ISOLATION_LEVEL = {LEVEL}; SET TRANSACTION {MODE}; SELECT 'Tr B: ' || GET_IDS_AND_ISOLATION_LEVEL FROM DUAL; SELECT 'Tr B START: ' || GET_CT FROM DUAL;
	SELECT 'Tr B COUNT-1: ' || GET_CT FROM DUAL; SELECT COUNT(*) FROM "subscriptions" WHERE "sb_id" > 500;
EXEC DBMS_LOCK.SLEEP(5);	
SELECT 'Tr A INSERT: ' || GET_CT FROM DUAL; INSERT INTO "subscriptions" ("sb_id", "sb_subscriber", "sb_book", "sb_start", "sb_finish", "sb_is_active") VALUES (1000, 1, 1, TO_DATE('2025-01-12', 'YYYY-MM-DD'), TO_DATE('2026-01-12', 'YYYY-MM-DD'), 'N');	
	EXEC DBMS_LOCK.SLEEP(10);
	SELECT 'Tr B COUNT-2: ' || GET_CT FROM DUAL; SELECT COUNT(*) FROM "subscriptions" WHERE "sb_id" > 500;
EXEC DBMS_LOCK.SLEEP(10);	
SELECT 'Tr A ROLLBACK: ' || GET_CT FROM DUAL; ROLLBACK;	
	EXEC DBMS_LOCK.SLEEP(15);
	SELECT 'Tr B COUNT-3: ' || GET_CT FROM DUAL; SELECT COUNT(*) FROM "subscriptions" WHERE "sb_id" > 500; SELECT 'Tr B COMMIT: ' || GET_CT FROM DUAL; COMMIT;

Before executing the above code blocks, we should disable the trigger that provides auto incrementation of the primary key in the `subscriptions` table (`ALTER TRIGGER "TRG_subscriptions_sb_id" DISABLE`), and after the experiment we should enable the trigger again (`ALTER TRIGGER "TRG_subscriptions_sb_id" ENABLE`).

We cannot add these commands immediately before and after `INSERT` in transaction A because `ALTER TRIGGER` automatically commits the previous transaction and starts a new one.

The final results of transactions interaction are as follows.

			Transaction B isolation level			
			READ COMMITTED		SERIALIZABLE	
			READ ONLY	READ WRITE	READ ONLY	READ WRITE
Transaction A isolation level	READ COMMITTED	READ ONLY	INSERT in transaction A is forbidden (R/O)	INSERT in transaction A is forbidden (R/O)	INSERT in transaction A is forbidden (R/O)	INSERT in transaction A is forbidden (R/O)
		READ WRITE	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record"
	SERIALIZABLE	READ ONLY	INSERT in transaction A is forbidden (R/O)	INSERT in transaction A is forbidden (R/O)	INSERT in transaction A is forbidden (R/O)	INSERT in transaction A is forbidden (R/O)
		READ WRITE	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record"	Transaction B does not access the "phantom record"

This completes the solution of this problem.

Solution 6.2.2.b{449}.

In the solution{449} of problem 6.2.2.a{449} we got the situation of transactions mutual deadlock in some cases, but now we will consider the code that is guaranteed to lead to such a situation in all three DBMSes.

At low isolation levels, a DBMS may be able to avoid the deadlocks, which is why we use the **`SERIALIZABLE`** level. You are encouraged to explore DBMS behavior at other isolation levels on your own in task 6.2.2.TSK.F{479}.

It is important to note that only MS SQL Server allows us to specify the priority of transactions; it is taken into account when deciding which of the two deadlocked transactions will be canceled, MySQL and Oracle make this decision entirely independently.

The code below works according to the following algorithm:

- transaction A updates the first table;
- transaction B updates the second table;
- transaction A tries to update the second table (the row blocked by transaction B);
- transaction B tries to update the first table (the row blocked by transaction A);
- transactions mutual deadlock occurs.

Let's look at the code that implements this algorithm.

Solution for MySQL looks like this.

MySQL Solution 6.2.2.b

```
1     -- Transaction A:                          -- Transaction B:
2     SET autocommit = 0;                        SET autocommit = 0;
3     SET SESSION TRANSACTION                    SET SESSION TRANSACTION
4     ISOLATION LEVEL SERIALIZABLE;              ISOLATION LEVEL SERIALIZABLE;
5     START TRANSACTION;                         START TRANSACTION;
6     UPDATE `books`
7     SET    `b_name` =                          SELECT SLEEP(3);
8            CONCAT(`b_name`, '.')
9     WHERE  `b_id` = 1;
10                                               UPDATE `subscribers`
11    SELECT SLEEP(5);                           SET    `s_name` =
12                                                      CONCAT(`s_name`, '.')
13                                               WHERE  `s_id` = 1;
14    UPDATE `subscribers`
15    SET    `s_name` =                          SELECT SLEEP(3);
16           CONCAT(`s_name`, '.')
17    WHERE  `s_id` = 1;
18    COMMIT;                                    UPDATE `books`
19                                               SET    `b_name` =
20                                                      CONCAT(`b_name`, '.')
21                                               WHERE  `b_id` = 1;
22                                               COMMIT;
```

Solution for MS SQL Server looks as follows. Note line 5, where the first transaction is given a higher priority and the second is given a lower one, which means that the DBMS will always cancel the second transaction, allowing the first one to finish successfully.

MS SQL Solution 6.2.2.b

```
1     -- Transaction A:                          -- Transaction B:
2     SET IMPLICIT_TRANSACTIONS ON;              SET IMPLICIT_TRANSACTIONS ON;
3     SET TRANSACTION ISOLATION                  SET TRANSACTION ISOLATION
4     LEVEL SERIALIZABLE;                        LEVEL SERIALIZABLE;
5     SET DEADLOCK_PRIORITY HIGH;                SET DEADLOCK_PRIORITY LOW;
6     BEGIN TRANSACTION;                         BEGIN TRANSACTION;
6     UPDATE [books]
7     SET    [b_name] =                          WAITFOR DELAY '00:00:03';
8            CONCAT([b_name], '.')
9     WHERE  [b_id] = 1;
10                                               UPDATE [subscribers]
11    WAITFOR DELAY '00:00:05';                  SET    [s_name] =
12                                                      CONCAT([s_name], '.')
13                                               WHERE  [s_id] = 1;
14    UPDATE [subscribers]
15    SET    [s_name] =                          WAITFOR DELAY '00:00:03';
16           CONCAT([s_name], '.')
17    WHERE  [s_id] = 1;
18    COMMIT;                                    UPDATE [books]
19                                               SET    [b_name] =
20                                                      CONCAT([b_name], '.')
21                                               WHERE  [b_id] = 1;
22                                               COMMIT;
```

Solution for Oracle looks like this.

Oracle | Solution 6.2.2.b

1	`-- Transaction A:`	`-- Transaction B:`
2	`ALTER SESSION SET`	`ALTER SESSION SET`
3	`ISOLATION_LEVEL = SERIALIZABLE;`	`ISOLATION_LEVEL = SERIALIZABLE;`
4	`SET TRANSACTION READ WRITE;`	`SET TRANSACTION READ WRITE;`
5	`UPDATE "books"`	
6	`SET "b_name" =`	`EXEC DBMS_LOCK.SLEEP(3);`
7	`CONCAT("b_name", '.')`	
8	`WHERE "b_id" = 1;`	
9		`UPDATE "subscribers"`
10	`EXEC DBMS_LOCK.SLEEP(5);`	`SET "s_name" =`
11		`CONCAT("s_name", '.')`
12		`WHERE "s_id" = 1;`
14	`UPDATE "subscribers"`	
15	`SET "s_name" =`	`EXEC DBMS_LOCK.SLEEP(3);`
16	`CONCAT("s_name", '.')`	
17	`WHERE "s_id" = 1;`	
18	`COMMIT;`	`UPDATE "books"`
19		`SET "b_name" =`
20		`CONCAT("b_name", '.')`
21		`WHERE "b_id" = 1;`
22		`COMMIT;`

This completes the solution of this problem.

Task 6.2.2.TSK.A: repeat the experiment presented in the solution{449} of problem 6.2.2.a{449} and personally check the behavior of all three DBMSes in all the situations considered.

Task 6.2.2.TSK.B: repeat the experiment presented in the solution{477} of problem 6.2.2.b{449} and personally check the behavior of all three DBMSes in all the situations considered.

Task 6.2.2.TSK.C: create code in which a query that inverts the values of the **`sb_is_active`** field of the **`subscriptions`** table from **`Y`** to **`N`** and vice versa will have the best chance of successful completion in case of transactions mutual deadlock.

Task 6.2.2.TSK.D: investigate MySQL behavior in the context of the non-repeatable read anomaly by performing the first operation in each transaction in **`LOCK IN SHARE MODE`** and **`FOR UPDATE`** modes (see solution{449} of problem 6.2.2.a{449}).

Task 6.2.2.TSK.E: investigate MS SQL Server behavior in the context of lost update and non-repeatable read anomalies by performing the first operation in each transaction using the **`UPDLOCK`** “table hint[39]” (see solution{449} of problem 6.2.2.a{449}).

Task 6.2.2.TSK.F: repeat the solution{477} of problem 6.2.2.b{449} for all three DBMSes in the remaining transaction isolation levels they support, find such combinations of isolation levels that transactions mutual deadlock does not occur.

[39] https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-table

6.2.3. Example 45: Managing Transactions in Triggers, Stored Functions and Procedures

Problem 6.2.3.a{480}: create a trigger on the **books** table that determines the isolation level of the transaction in which the insertion operation is currently running, and cancels the operation if the isolation level of the transaction is other than **SERIALIZABLE**.

Problem 6.2.3.b{484}: create a stored function that raises an exception in case of being called in autocommit transaction mode.

Problem 6.2.3.c{486}: create a stored procedure that counts the number of records in a specified table so that the query is executed as fast as possible (regardless of the queries being executed in parallel), even if it returns not quite correct data in the end.

Expected result 6.2.3.a.

If a data insertion operation into the **books** table is performed in a transaction with a level of isolation other than **SERIALIZABLE**, the trigger cancels the operation and raises an exception.

Expected result 6.2.3.b.

If the stored function is called at the moment when the current DBMS session is in the autocommit transaction mode, the function must raise an exception and stop operating.

Expected result 6.2.3.c.

The stored procedure must count records in the specified table in a transaction with the isolation level that provides the minimum probability of waiting for concurrent transactions or individual operations in them to complete.

Solution 6.2.3.a{480}.

For simplicity (no need to perform the insert operation manually) and consistency (for all three DBMSes), we use **AFTER** triggers.

Thus, the following solutions will differ only in the logic of determining the transaction isolation level, because in each DBMS the corresponding mechanism is implemented in a very special way, incompatible with other DBMSes.

Solution for MySQL looks like this.

MySQL Solution 6.2.3.a (trigger code)

```
1    DELIMITER $$
2
3    CREATE TRIGGER `books_ins_trans`
4    AFTER INSERT
5    ON `books`
6     FOR EACH ROW
7      BEGIN
8       DECLARE isolation_level VARCHAR(50);
9
10      SET isolation_level =
11      (
12       SELECT `VARIABLE_VALUE`
13       FROM   `information_schema`.
14              `session_variables`
15       WHERE  `VARIABLE_NAME` =
16              'tx_isolation'
17       );
18
19      IF (isolation_level != 'SERIALIZABLE')
20       THEN
21        SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = 'Please, switch your
22        transaction to SERIALIZABLE isolation level and rerun this
23        INSERT again.', MYSQL_ERRNO = 1001;
24      END IF;
25
26     END;
27   $$
28
29   DELIMITER ;
```

Since new versions of MySQL will (most likely) stop using the `session_variables` approach, we can use the alternative (line 10 of the query below).

MySQL Solution 6.2.3.a (trigger code) (alternative for newer versions)

```
1    DELIMITER $$
2
3    CREATE TRIGGER `books_ins_trans`
4    AFTER INSERT
5    ON `books`
6     FOR EACH ROW
7      BEGIN
8       DECLARE isolation_level VARCHAR(50);
9
10      SET isolation_level = @@transaction_isolation;
11
12      IF (isolation_level != 'SERIALIZABLE')
13       THEN
14        SIGNAL SQLSTATE '45001' SET MESSAGE_TEXT = 'Please, switch your
15        transaction to SERIALIZABLE isolation level and rerun this
16        INSERT again.', MYSQL_ERRNO = 1001;
17      END IF;
18
19     END;
20   $$
21
22   DELIMITER ;
```

We can check if the presented solution works and is correct by executing the following code: the first attempt to execute the insertion will result in an exception raised in the trigger, and the second attempt will be successful.

MySQL Solution 6.2.3.a (code to check if the solution works)

```
SET SESSION TRANSACTION
ISOLATION LEVEL READ COMMITTED;

INSERT INTO `books`
            (`b_name`,
             `b_year`,
             `b_quantity`)
VALUES      ('And another book',
             1985,
             2);

SET SESSION TRANSACTION
ISOLATION LEVEL SERIALIZABLE;

INSERT INTO `books`
            (`b_name`,
             `b_year`,
             `b_quantity`)
VALUES      ('And another book',
             1985,
             2);
```

Solution for MS SQL Server looks like this.

MS SQL Solution 6.2.3.a (trigger code)

```
CREATE TRIGGER [books_ins_trans]
ON [books]
AFTER INSERT
AS
 DECLARE @isolation_level NVARCHAR(50);

 SET @isolation_level =
 (
  SELECT [transaction_isolation_level]
  FROM [sys].[dm_exec_sessions]
  WHERE [session_id] = @@SPID
 );

 IF (@isolation_level != 4)
  BEGIN
   RAISERROR ('Please, switch your transaction to SERIALIZABLE isolation
               level and rerun this INSERT again.', 16, 1);
   ROLLBACK TRANSACTION;
   RETURN
  END;
GO
```

We can check if the presented solution works and is correct by executing the following code: the first attempt to execute the insertion will result in an exception raised in the trigger, and the second attempt will be successful.

MS SQL Solution 6.2.3.a (code to check if the solution works)

```
SET TRANSACTION ISOLATION
LEVEL READ COMMITTED;

INSERT INTO [books]
            ([b_name],
             [b_year],
             [b_quantity])
VALUES      ('And another book',
             1985,
             2);
```

MS SQL Solution 6.2.3.a (code to check if the solution works) (continued)

```
SET TRANSACTION ISOLATION
LEVEL SERIALIZABLE;

INSERT INTO [books]
            ([b_name],
             [b_year],
             [b_quantity])
VALUES      ('And another book',
             1985,
             2);
```

Solution for Oracle looks like this.

Oracle Solution 6.2.3.a (trigger code)

```
CREATE OR REPLACE TRIGGER "books_ins_trans"
AFTER INSERT
ON "books"
 FOR EACH ROW
  DECLARE
   isolation_level NVARCHAR2(150);
   trans_id VARCHAR(100);
  BEGIN
   trans_id := DBMS_TRANSACTION.LOCAL_TRANSACTION_ID(FALSE);
   SELECT CASE BITAND("transaction".flag, POWER(2, 28))
            WHEN 0 THEN 'READ COMMITTED'
            ELSE 'SERIALIZABLE'
          END AS "session_isolation_level"
   INTO   isolation_level
   FROM   v$transaction "transaction"
          JOIN v$session "session"
            ON "transaction".addr = "session".taddr
               AND "session".sid = SYS_CONTEXT('USERENV', 'SID');

   IF (isolation_level != 'SERIALIZABLE')
    THEN
     RAISE_APPLICATION_ERROR(-20001, 'Please, switch your transaction
     to SERIALIZABLE isolation level and rerun this INSERT again.');
    END IF;

  END;
```

We can check if the presented solution works and is correct by executing the following code: the first attempt to execute the insertion will result in an exception raised in the trigger, and the second attempt will be successful.

Oracle Solution 6.2.3.a (code to check if the solution works)

```
ALTER SESSION SET
ISOLATION_LEVEL = READ COMMITTED;

INSERT INTO "books"
            ("b_name",
             "b_year",
             "b_quantity")
VALUES      ('And another book',
             1985,
             2);
```

Oracle Solution 6.2.3.a (code to check if the solution works) (continued)

```
11  ALTER SESSION SET
12  ISOLATION_LEVEL = SERIALIZABLE;
13
14  INSERT INTO "books"
15              ("b_name",
16               "b_year",
17               "b_quantity")
18  VALUES      ('And another book',
19               1985,
20               2);
```

This completes the solution of this problem.

Solution 6.2.3.b{480}.

Since the condition of the problem does not say exactly what the function should do, we will limit ourselves to checking for transaction autocommit mode and raising an exception if it is turned on.

Solution for MySQL looks like this.

MySQL Solution 6.2.3.b (the function code)

```
1   DELIMITER $$
2   CREATE FUNCTION NO_AUTOCOMMIT()
3   RETURNS INT DETERMINISTIC
4   BEGIN
5    IF ((SELECT @@autocommit) = 1)
6    THEN
7     SIGNAL SQLSTATE '45001'
8      SET MESSAGE_TEXT = 'Please, turn the autocommit off.',
9      MYSQL_ERRNO = 1001;
10    RETURN -1;
11   END IF;
12
13   -- There may be some useful code here :).
14
15   RETURN 0;
16  END$$
17
18  DELIMITER ;
```

We can check the functionality and correctness of the presented solution by executing the following code: the first function call will end with an exception, and the second one will succeed.

MySQL Solution 6.2.3.b (code to check if the solution works)

```
1   SET autocommit = 1;
2   SELECT NO_AUTOCOMMIT();
3
4   SET autocommit = 0;
5   SELECT NO_AUTOCOMMIT();
```

Solution for MS SQL Server looks like this.

Note the following important points specific to MS SQL Server:

- the state of transaction autocommit can only be determined indirectly (lines 8-23 of the code);
- it is impossible to explicitly raise an exception in the function, so, we have to use a workaround (lines 27-31 of the code);
- it is impossible to rollback a transaction in the function, but due to the exception raise, the transaction will be rolled back.

MS SQL Solution 6.2.3.b (the function code)

```
CREATE FUNCTION NO_AUTOCOMMITT()
RETURNS INT
WITH SCHEMABINDING
AS
BEGIN
 DECLARE @autocommit INT;

 IF (@@TRANCOUNT = 0 AND (@@OPTIONS & 2 = 0))
  BEGIN
   SET @autocommit = 1;
  END
 ELSE IF (@@TRANCOUNT = 0 AND (@@OPTIONS & 2 = 2))
  BEGIN
   SET @autocommit = 0;
  END
 ELSE IF (@@OPTIONS & 2 = 0)
  BEGIN
   SET @autocommit = 1;
  END
 ELSE
  BEGIN
   SET @autocommit = 0;
  END;

 IF (@autocommit = 1)
  BEGIN
   -- We cannot use RAISEERROR in MS SQL Server functions!
   -- RAISERROR ('Please, turn the autocommit off.', 16, 1);

   -- A workaround to raise an exception:
   RETURN CAST('Please, turn the autocommit off.' AS INT);

   -- We cannot rollback a transaction from MS SQL Server function either.
   -- ROLLBACK TRANSACTION;
  END;

 -- There may be some useful code here :).

 RETURN 0;
END;
GO
```

We can check the functionality and correctness of the presented solution by executing the following code: the first function call will end with an exception, and the second one will succeed.

MS SQL Solution 6.2.3.b (code to check if the solution works)

```
SET IMPLICIT_TRANSACTIONS OFF;
SELECT dbo.NO_AUTOCOMMITT();

SET IMPLICIT_TRANSACTIONS ON;
SELECT dbo.NO_AUTOCOMMITT();
```

The solution for Oracle looks like this. Yes, it looks exactly like that, because in Oracle there is no such thing as transaction autocommit, this effect can be implemented by some DBMS tools, but the DBMS itself always waits for explicit COMMITT or ROLLBACK.

Oracle Solution 6.2.3.b (the function code)

```
1    CREATE FUNCTION NO_AUTOCOMMITT
2     RETURN INT
3     DETERMINISTIC
4     IS
5    BEGIN
6     DBMS_OUTPUT.PUT_LINE('Have a nice day :)');
7     RETURN 1;
8    END;
```

We can check the functionality and correctness of the presented solution by executing the following code.

Oracle Solution 6.2.3.b (code to check if the solution works)

```
1    SET SERVEROUTPUT ON;
2    SELECT NO_AUTOCOMMITT FROM DUAL;
```

This completes the solution of this problem.

Solution 6.2.3.c{480}.

The idea to solve this problem is to use a transaction isolation level that is least likely to be affected by locks generated by other transactions.

In MySQL this level is **`READ UNCOMMITTED`**, in MS SQL Server it is also **`READ UNCOMMITTED`** or **`SNAPSHOT`** (but **`SNAPSHOT`** can lead to additional resource consumption), in Oracle data reading is always an independent process, so in this DBMS we can use **`READ COMMITTED`** (especially since there is no **`READ UNCOMMITTED`** in Oracle).

The solution for MySQL looks like this.

MySQL Solution 6.2.3.c (the procedure code)

```
1    DELIMITER $$
2    CREATE PROCEDURE COUNT_ROWS(IN table_name VARCHAR(150),
3                                OUT rows_in_table INT)
4    BEGIN
5     SET SESSION TRANSACTION
6     ISOLATION LEVEL READ UNCOMMITTED;
7
8     SET @count_query =
9     CONCAT('SELECT COUNT(1) INTO @rows_found
10            FROM ', table_name);
11
12    PREPARE count_stmt FROM @count_query;
13    EXECUTE count_stmt;
14    DEALLOCATE PREPARE count_stmt;
15
16    SET rows_in_table := @rows_found;
17   END;
18   $$
19   DELIMITER ;
```

We can check the functionality and correctness of the presented solution by executing the following code.

MySQL Solution 6.2.3.c (code to check if the solution works)

```
CALL COUNT_ROWS('subscriptions', @rows_in_table);
SELECT @rows_in_table;
```

The solution for MS SQL Server looks like this.

MS SQL Solution 6.2.3.c (the procedure code)

```
CREATE PROCEDURE COUNT_ROWS
                  @table_name NVARCHAR(150),
                  @rows_in_table INT OUTPUT
AS
 DECLARE @count_query NVARCHAR(1000) = '';

 SET TRANSACTION ISOLATION
 LEVEL READ UNCOMMITTED;

 SET @count_query =
 CONCAT('SET @rows_f = (SELECT COUNT(1) FROM [', @table_name, '])');
 EXECUTE sp_executesql @count_query,
                       N'@rows_f INT OUT',
                       @rows_in_table OUTPUT;
GO
```

We can check the functionality and correctness of the presented solution by executing the following code.

MySQL Solution 6.2.3.c (code to check if the solution works)

```
DECLARE @res INT;
EXECUTE COUNT_ROWS 'subscriptions', @res OUTPUT;
SELECT @res;
```

The solution for Oracle is as follows.

Oracle Solution 6.2.3.c (the procedure code)

```
CREATE PROCEDURE COUNT_ROWS (table_name IN VARCHAR,
                             rows_in_table OUT NUMBER) AS
 count_query VARCHAR(1000) := '';
BEGIN

 EXECUTE IMMEDIATE 'ALTER SESSION SET
 ISOLATION_LEVEL = READ COMMITTED';

 count_query :=
 'SELECT COUNT(1) FROM "' || table_name || '"';
 EXECUTE IMMEDIATE count_query INTO rows_in_table;
END;
/
```

We can check the functionality and correctness of the presented solution by executing the following code.

Oracle Solution 6.2.3.c (code to check if the solution works)

```
DECLARE
 res NUMBER;
BEGIN
 COUNT_ROWS('subscriptions', res);
 DBMS_OUTPUT.PUT_LINE('Rows: ' || res);
END;
```

This completes the solution of this problem.

Task 6.2.3.TSK.A: create a trigger on the `subscriptions` table that determines the transaction isolation level in which the update operation is currently running, and cancels the operation if the transaction isolation level is different from `REPEATABLE READ`.

Task 6.2.3.TSK.B: create a stored function that raises an exception if both conditions are met:

- transaction autocommit mode is disabled;
- the function is called from a nested transaction.

Hint: this problem has a solution only for MS SQL Server.

Task 6.2.3.TSK.C: create a stored procedure that counts the number of records in a specified table so that it returns the most correct data possible, even if we have to sacrifice performance to achieve this result.

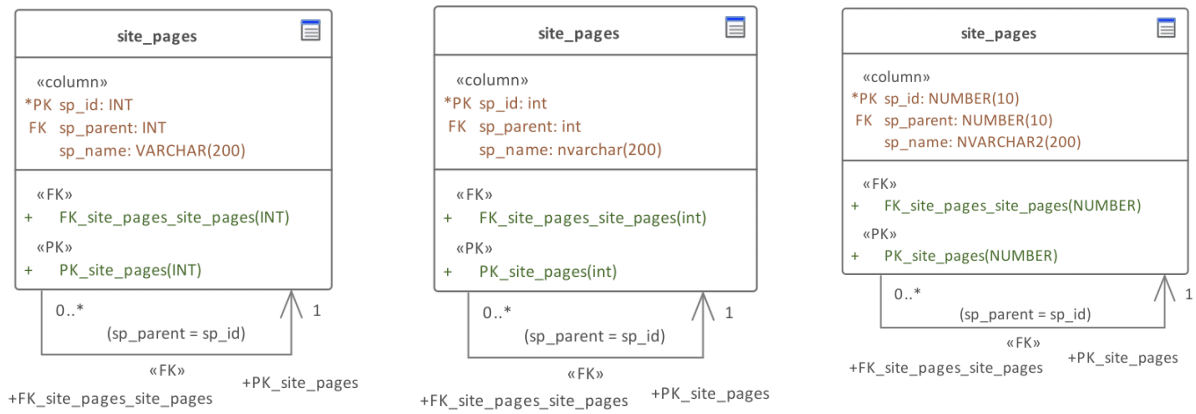
Chapter 7: Solving Typical Tasks and Performing Typical Operations

7.1. Operations with Hierarchical and Linked Data

7.1.1. Example 46: Managing Hierarchical Structures

To solve the problems in this example, we will need a new table, which we will create in the “Exploration” database. This table will store a tree of elements (let’s assume that it will be the tree structure of our hypothetical library website).

There are many ways to store tree structures in relational databases[40], but we use recursive foreign keys as one of the most common solutions. The corresponding fragments of the database schema for all three DBMSes are shown in figure 7.a.



MySQL MS SQL Server Oracle

Figure 7.a — The `site_pages` table in all three DBMSes

Let’s save the following data set in the `site_pages` table, visually represented in figure 7.b.

sp_id	sp_parent	sp_name
1	NULL	Main
2	1	For subscribers
3	1	For sponsors
4	1	For advertisers
5	2	News
6	2	Statistics
7	3	Offers
8	3	Success stories
9	4	Events
10	1	Contacts
11	3	Documents
12	6	Current
13	6	Archive
14	6	Non-official

[40] http://www.amazon.com/dp/1558609202/

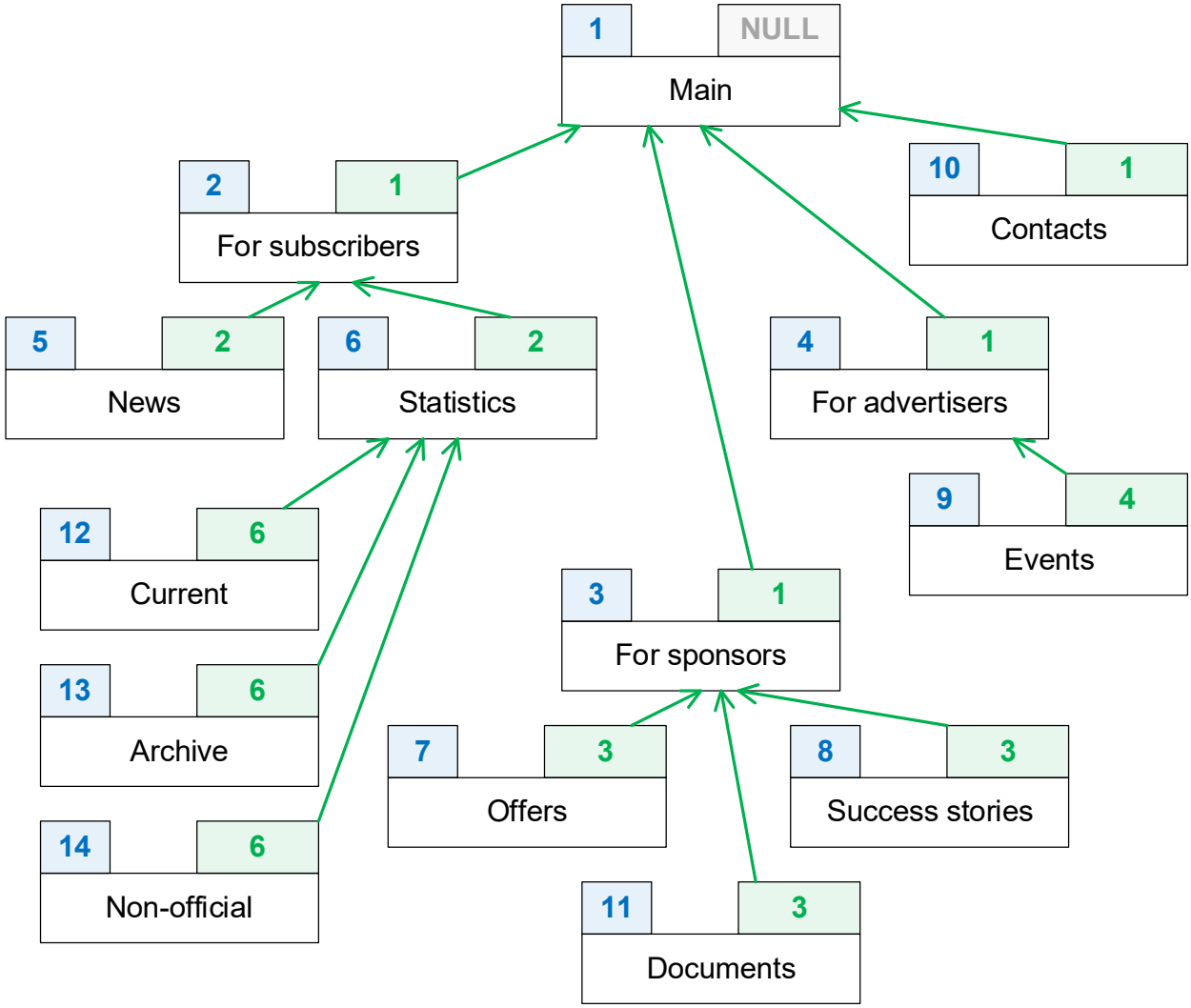


Figure 7.b — Visual representation of the sitemap

Now that all the data have been prepared, we can move on to the problems.

Problem 7.1.1.a{491}: create a function that returns a list of identifiers of all child nodes of a given node (e.g., identifiers of all subpages of the “For subscribers” page).

Problem 7.1.1.b{498}: create a query to show the entire subtree of a given node in the tree including the parent node itself (e.g., all the subpages of the “For subscribers” page including the page itself).

Problem 7.1.1.c{501}: create a function that returns a list of node identifiers on the path from a given node to the tree root (e.g., identifiers of all nodes on the path from the “Archive” table page to the “Main” page).

Expected result 7.1.1.a.

For the node with identifier 2 (the “For subscribers” page) the function should return the following data:

shildren_of_2
5,6,12,13,14

Expected result 7.1.1.b.

For the node with identifier 2 (the “For subscribers” page) the query should return the following data:

sp_id	sp_name
2	For subscribers
5	News
6	Statistics
12	Current
13	Archive
14	Non-official

Expected result 7.1.1.c.

For the node with identifier 13 (the “Archive” page) the query should return the following data:

path
13
6
2
1

This option is also allowed:

path
13,6,2,1

Solution 7.1.1.a{490}.

The solution of this problem for MS SQL Server and Oracle can be very easily built on the basis of recursive Common Table Expressions. In MySQL, however, Common Table Expressions are supported only since version 8, so for this DBMS we will consider both the classical solution and the non-trivial one.

Let’s start with a non-trivial one and first give the ready-made code of the function and an example of how to use it.

MySQL Solution 7.1.1.a (the function code) (before version 8)

```
1    DELIMITER $$
2    CREATE FUNCTION GET_ALL_CHILDREN(start_node INT)
3     RETURNS TEXT
4     BEGIN
5      DECLARE result TEXT;
6      SELECT GROUP_CONCAT(`children_ids` SEPARATOR ',') INTO result
7      FROM   (
8                   SELECT `sp_id`, @parent_values :=
9                          (
10                                SELECT GROUP_CONCAT(`sp_id` SEPARATOR ',')
11                                FROM   `site_pages`
12                                WHERE  FIND_IN_SET(`sp_parent`,
13                                                   @parent_values) > 0
14                         ) AS `children_ids`
15                  FROM   `site_pages`
16                  JOIN   (SELECT @parent_values := start_node)
17                                                        AS `initialisation`
18                  WHERE  `sp_id` IN (@parent_values)
19            ) AS `data`;
20     RETURN result;
21    END$$
22   DELIMITER ;
```

We can check the functionality and correctness of the presented solution by executing the following code.

MySQL Solution 7.1.1.a (function usage sample) (before version 8)

```
1    -- Functionality check:
2    SELECT GET_ALL_CHILDREN(2) AS `children_of_2`;
3
4    -- Function usage:
5    SELECT `sp_id`, `sp_name`, GET_ALL_CHILDREN(`sp_id`) AS `children`
6    FROM `site_pages`;
```

The second query will return the following data (a list of all pages of the library site with all their subpages' identifiers).

sp_id	sp_name	children
1	Main	2,3,4,10,5,6,7,8,9,11,12,13,14
2	For subscribers	5,6,12,13,14
3	For sponsors	7,8,11
4	For advertisers	9
5	News	NULL
6	Statistics	12,13,14
7	Offers	NULL
8	Success stories	NULL
9	Events	NULL
10	Contacts	NULL
11	Documents	NULL
12	Current	NULL
13	Archive	NULL
14	Non-official	NULL

Now let's see how this solution works.

Obviously, the main part of the presented function is the query in lines 6-19. Let's rewrite it without the function.

MySQL Solution 7.1.1.a (query on which the function is based) (before version 8)

```
1     SELECT GROUP_CONCAT(`children_ids` SEPARATOR ',') AS `children_ids`
2     FROM    (
3                     SELECT `sp_id`, @parent_values :=
4                            (
5                                   SELECT GROUP_CONCAT(`sp_id` SEPARATOR ',')
6                                   FROM   `site_pages`
7                                   WHERE  FIND_IN_SET(`sp_parent`,
8                                                      @parent_values) > 0
9                            ) AS `children_ids`
10                    FROM   `site_pages`
11                    JOIN   (SELECT @parent_values := {parent_node})
12                                                        AS `initialisation`
13                    WHERE  `sp_id` IN (@parent_values)
14          ) AS `prepared_data`
```

In this form this query will return the data presented in the expected result of the problem (if the value of `{parent_node}` is 2).

Let's consider the solution in parts.

1) The code in lines 11-12 initializes the value of `@parent_values` variable to the identifier of the node for which the list of identifiers of child nodes is built. Later the list of nodes will be stored here, but only one value is placed here at the beginning.
2) The code in lines 5-8 forms a set of child nodes' identifiers of those nodes listed in the `@parent_values` variable. The result is used as a new value of `@parent_values` variable.
3) The condition in line 13 narrows the selection to only to those nodes whose children are searched for in the current step.
4) The algorithm terminates when the value of `@parent_values` becomes `NULL`.
5) Thanks to `GROUP_CONCAT` on line 1, the entire result is presented as a single list of identifiers, in which they are listed separated by commas. This presentation allows us to use the `FIND_IN_SET` function to further manipulate the results.

If we run lines 3-13 of this query separately, we will get the following result (the result itself is shown on a gray background so as not to confuse it with explanations).

Step	sp_id	children_ids	New value of @parent_values
Initial state, search for the children of the node 2.			
1	2	5,6	5,6
Now we need to find children of nodes 5 and 6 (the line with node 6 "collapses" because of `GROUP_CONCAT`, i.e., in such results we lose all `sp_id` values except the very first one, but `@parent_values` thanks to the same `GROUP_CONCAT` gets children of all `sp_id`, even those ones we "lost").			
2	5	12,13,14	12,13,14
Now we need to find children of nodes 13, 13 and 14 (the lines with nodes 13 and 14 "collapse" because of `GROUP_CONCAT`, i.e., in such results we lose all `sp_id` values except the very first one, but `@parent_values` thanks to the same `GROUP_CONCAT` gets children of all `sp_id`, even those ones we "lost").			
3	12	NULL	NULL
Now we have to find the children of nodes… `NULL`, i.e., none: the end of the algorithm.			

If we use MySQL version 8 or newer, we can implement a solution based on the so-called recursive Common Table Expression[41].

The solution code looks like this.

MySQL Solution 7.1.1.a (the function code) (version 8 and newer)

```
1   DELIMITER $$
2   CREATE FUNCTION GET_ALL_CHILDREN (parent INT)
3   RETURNS VARCHAR(16000) READS SQL DATA
4   BEGIN
5    DECLARE result VARCHAR(16000);
6
7    WITH RECURSIVE `tree` (`sp_id`, `sp_parent`)
8     AS
9     (
10     SELECT `sp_id`,
11            `sp_parent`
12     FROM   `site_pages`
13     WHERE  `sp_id` = parent
14     UNION ALL
15     SELECT `inner`.`sp_id`,
16            `inner`.`sp_parent`
17     FROM   `site_pages` AS `inner`
18           JOIN `tree`
19             ON `inner`.`sp_parent` = `tree`.`sp_id`
20    )
21    SELECT GROUP_CONCAT(`sp_id` SEPARATOR ',') INTO result
22    FROM   `tree`
23    WHERE  `sp_id` != parent;
24
25    RETURN result;
26  END$$
27
28  DELIMITER ;
```

We can check the functionality and correctness of the presented solution by executing the following code.

MySQL Solution 7.1.1.a (function usage sample) (version 8 and newer)

```
1   SELECT GET_ALL_CHILDREN(2);
```

The logic of recursive Common Table Expression (lines 7-23) is described in detail further (in the solution for MS SQL Server).

Here we also note that (since MySQL has no way to return a table from a function) we have to convert the entire result from a column to a string (using the **`GROUP_CONCAT`** function in line 21 of the query) and place it in the resulting text variable (using the **`INTO`** clause in line 21 of the query).

Let's move on to MS SQL Server. Here the solution will be different, because, firstly, this DBMS does not support some of the syntax needed to emulate MySQL behavior, and secondly, using the features of MS SQL Server, this task is easier to solve (despite the fact that there will be more code).

[41] https://dev.mysql.com/doc/refman/8.0/en/with.html#common-table-expressions-recursive

MS SQL Solution 7.1.1.a (the function code)

```
CREATE FUNCTION GET_ALL_CHILDREN(@parent INT, @mode VARCHAR(50))
RETURNS @all_children TABLE
    (
     id VARCHAR(max)
    )
AS
BEGIN
 IF (@mode = 'TABLE')
  BEGIN
   WITH [tree] ([sp_id], [sp_parent])
   AS
   (
    SELECT [sp_id],
           [sp_parent]
    FROM   [site_pages]
    WHERE  [sp_id] =  @parent
    UNION ALL
    SELECT [inner].[sp_id],
           [inner].[sp_parent]
    FROM   [site_pages] AS [inner]
           JOIN [tree]
             ON [inner].[sp_parent] = [tree].[sp_id]
   )
   INSERT @all_children
   SELECT CAST([sp_id] AS VARCHAR)
   FROM   [tree]
   WHERE  [sp_id] != @parent
 END
  ELSE
 BEGIN
  WITH [tree] ([sp_id], [sp_parent])
  AS
  (
   SELECT [sp_id],
          [sp_parent]
   FROM   [site_pages]
   WHERE  [sp_id] = 2
   UNION ALL
   SELECT [inner].[sp_id],
          [inner].[sp_parent]
   FROM   [site_pages] AS [inner]
          JOIN [tree]
            ON [inner].[sp_parent] = [tree].[sp_id]
  )
   INSERT @all_children
   SELECT STUFF((SELECT ',' + CAST([sp_id] AS VARCHAR)
                 FROM   [tree]
                        WHERE  [sp_id] != 2
                 FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
                 1, 1, '');
 END;
 RETURN
END;
```

We can verify the functionality and correctness of the presented solution by executing the following code. The first query will return the result as required by the condition (comma-separated list of identifiers) and the second query will return a single-column table with each identifier on a separate line.

MS SQL Solution 7.1.1.a (function usage sample)

```
SELECT * FROM GET_ALL_CHILDREN(2, 'STRING');
SELECT * FROM GET_ALL_CHILDREN(2, 'TABLE');
```

So, let's take a look at this solution. We created exactly **table** function to be able to return data not only as a comma-separated list of identifiers, but also as a single-column table whose rows are identifiers.

In the case where we still need a comma-separated list of identifiers, we still return a single-column table, but it will have exactly one row containing our entire list.

The solution is based on the so-called recursive Common Table Expression[42]. Let's consider it separately.

MS SQL Solution 7.1.1.a (recursive Common Table Expression)

```
1    WITH [tree] ([sp_id], [sp_parent])
2    AS
3    (
4     SELECT [sp_id],
5            [sp_parent]
6     FROM   [site_pages]
7     WHERE  [sp_id] = {parent_node}
8     UNION ALL
9     SELECT [inner].[sp_id],
10           [inner].[sp_parent]
11    FROM   [site_pages] AS [inner]
12           JOIN [tree]
13             ON [inner].[sp_parent] = [tree].[sp_id]
14   )
15   SELECT [sp_id]
16   FROM   [tree]
17   WHERE  [sp_id] != {parent_node}
```

A recursive Common Table Expression must contain two parts:

- the first part (lines 4-7) is the selection of the parent record (for which we will search for the children);
- the second part (lines 8-13) recursively refers to the result of the Common Table Expression, which allows us to get the whole subtree of a given node.

Since the first part (before the `UNION` operator) is mandatory, and the identifier of the parent node itself should not be included in the result, we exclude the corresponding record in the final selection by the condition in line 17.

We placed the resulting recursive Common Table Expression in lines 10-27 of the function code, ensuring that its results (line 24) are inserted into the resulting table returned by the function.

Lines 31-50 of the function code contains the same recursive Common Table Expression, but when getting the results of its execution, we apply the technique described in detail in the solution{75} of problem 2.2.2.a{74} for emulating the `GROUP_CONCAT` MySQL function.

Let's move on to Oracle. Here the solution will be almost identical to the solution for MS SQL Server. The only difference is in the way the table is returned from the function (we dealt with this issue earlier, see solution{370} of problem 5.1.1.b{367}). Here too, we have to emulate the behavior of the `GROUP_CONCAT` function of MySQL by using the `LISTAGG` function in Oracle (see solution{75} of problem 2.2.2.a{74}).

[42] https://docs.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql

Oracle | Solution 7.1.1.a (the function code)

```
CREATE TYPE "tree_node" AS OBJECT("id" VARCHAR(32767));
/

CREATE TYPE "nodes_collection" AS TABLE OF "tree_node";
/

CREATE OR REPLACE FUNCTION GET_ALL_CHILDREN(parent_id NUMBER,
                                            function_mode VARCHAR)
RETURN "nodes_collection"
AS
 result_collection "nodes_collection";
BEGIN
 IF (function_mode = 'TABLE')
  THEN
   WITH "tree" ("sp_id", "sp_parent")
   AS
   (
    SELECT "sp_id",
           "sp_parent"
    FROM   "site_pages"
    WHERE  "sp_id" = parent_id
    UNION ALL
    SELECT "inner"."sp_id",
           "inner"."sp_parent"
    FROM   "site_pages" "inner"
           JOIN "tree"
             ON "inner"."sp_parent" = "tree"."sp_id"
   )
   SELECT "tree_node"(TO_CHAR("sp_id"))
   BULK COLLECT INTO result_collection
   FROM   "tree"
   WHERE  "sp_id" != parent_id;
  ELSE
   WITH "tree" ("sp_id", "sp_parent")
   AS
   (
    SELECT "sp_id",
           "sp_parent"
    FROM   "site_pages"
    WHERE  "sp_id" = parent_id
    UNION ALL
    SELECT "inner"."sp_id",
           "inner"."sp_parent"
    FROM   "site_pages" "inner"
           JOIN "tree"
             ON "inner"."sp_parent" = "tree"."sp_id"
   )
   SELECT "tree_node"(LISTAGG(TO_CHAR("sp_id"), ',' )
                      WITHIN GROUP (ORDER BY "sp_id"))
   BULK COLLECT INTO result_collection
   FROM   "tree"
   WHERE  "sp_id" != parent_id;
  END IF;
 RETURN result_collection;
END;
/
```

We can check the functionality and correctness of the presented solution by executing the following code.

The first query will return the result as required by the problem (comma-separated list of identifiers), while the second query will return a single-column table with each identifier on a separate line.

Oracle — Solution 7.1.1.a (function usage sample)

```
SELECT *
FROM TABLE(CAST(GET_ALL_CHILDREN(2, 'STRING') AS "nodes_collection"));

SELECT *
FROM TABLE(CAST(GET_ALL_CHILDREN(2, 'TABLE') AS "nodes_collection"));
```

This completes the solution of this problem.

Solution 7.1.1.b{490}.

It is easy to see that the solution of this problem is based on the considerations presented in the solution{491} of problem 7.1.1.a{490}. Assuming that we already have the corresponding functions, the code for all three DBMSes will look like this.

MySQL — Solution 7.1.1.b (using the **GET_ALL_CHILDREN** function from solution 7.1.1.a)

```
SELECT `sp_id`,
       `sp_name`
FROM   `site_pages`
WHERE  `sp_id` = {parent_node}
       OR FIND_IN_SET(`sp_id`, GET_ALL_CHILDREN({parent_node}))
```

MS SQL — Solution 7.1.1.b (using the **GET_ALL_CHILDREN** function from solution 7.1.1.a)

```
SELECT [sp_id],
       [sp_name]
FROM   [site_pages]
WHERE  [sp_id] = {parent_node}
       OR [sp_id] IN
                  (SELECT [id]
                   FROM GET_ALL_CHILDREN({parent_node}, 'TABLE'))
```

Oracle — Solution 7.1.1.b (using the **GET_ALL_CHILDREN** function from solution 7.1.1.a)

```
SELECT "sp_id",
       "sp_name"
FROM   "site_pages"
WHERE  "sp_id" = {parent_node}
       OR "sp_id" IN
                  (SELECT "id"
                   FROM TABLE(CAST(
                              GET_ALL_CHILDREN({parent_node},
                                               'TABLE')
                              AS "nodes_collection")))
```

If we assume that we do not have the **GET_ALL_CHILDREN** function, the solution can be constructed as follows.

In MySQL, we take the query around which the function in the solution{491} of problem 7.1.1.a{490} was built and use it directly as a source of the list of child nodes' identifiers (you are encouraged to create this solution using the recursive Common Table Expression in task 7.1.1.TSK.D{506}).

MySQL Solution 7.1.1.b

```
SELECT `sp_id`,
       `sp_name`
FROM   `site_pages`
WHERE  `sp_id` = {parent_node}
       OR FIND_IN_SET
       (`sp_id`,
        (SELECT GROUP_CONCAT(`children_ids`)
         FROM   (SELECT `sp_id`,
                 @parent_values :=
                  (
                   SELECT GROUP_CONCAT(`sp_id` SEPARATOR ',')
                   FROM   `site_pages`
                   WHERE  FIND_IN_SET(`sp_parent`, @parent_values) > 0
                  ) AS `children_ids`
                 FROM   `site_pages`
                 JOIN   (SELECT @parent_values := {parent_node})
                                                      AS `initialisation`
                 WHERE  `sp_id` IN (@parent_values)
         ) AS `prepared_data`)
       )
```

In MS SQL Server, we take the recursive Common Table Expression, around which the function in the solution{491} of problem 7.1.1.a{490} is built, and after adding the page name (`sp_name` field) to the selection, we use the result as a source of the required data. Here we don't even have to exclude the parent node itself from the result because by this problem it should be in the final data set.

Note that MS SQL Server (as well as MySQL) allows creating recursive Common Table Expressions without an explicit list of columns, while in Oracle this list is mandatory.

MS SQL Solution 7.1.1.b

```
WITH [tree]
AS
(
 SELECT [sp_id],
        [sp_parent],
             [sp_name]
 FROM   [site_pages]
 WHERE  [sp_id] = {parent_node}
 UNION ALL
 SELECT [inner].[sp_id],
        [inner].[sp_parent],
             [inner].[sp_name]
 FROM   [site_pages] AS [inner]
        JOIN [tree]
          ON [inner].[sp_parent] = [tree].[sp_id]
)
SELECT [sp_id],
       [sp_name]
FROM   [tree]
```

Oracle Solution 7.1.1.b

```
WITH "tree" ("sp_id", "sp_parent", "sp_name")
AS
(
 SELECT "sp_id",
        "sp_parent",
        "sp_name"
 FROM   "site_pages"
 WHERE  "sp_id" = {parent_node}
 UNION ALL
 SELECT "inner"."sp_id",
        "inner"."sp_parent",
        "inner"."sp_name"
 FROM   "site_pages" "inner"
        JOIN "tree"
          ON "inner"."sp_parent" = "tree"."sp_id"
)
SELECT "sp_id",
       "sp_name"
FROM   "tree"
```

If we move away from the tradition, in which we consider the most similar solutions in all three DBMSes, then in Oracle this problem can be solved by using the **`CONNECT BY`** clause.

For clarity, let's select **all** the information about website pages, and even add an indication of the level at which each of the pages is located in relation to the original parent, the list of subpages of which we are looking for.

Oracle Solution 7.1.1.b (alternative option)

```
SELECT "sp_id",
       "sp_parent",
       "sp_name",
       LEVEL
FROM   "site_pages"
START WITH "sp_id" = {parent_node}
CONNECT BY PRIOR "sp_id" = "sp_parent";
```

For the page with identifier 2, this query will return the following data.

sp_id	sp_parent	sp_name	LEVEL
2	1	For subscribers	1
5	2	News	2
6	2	Statistics	2
12	6	Current	3
13	6	Archive	3
14	6	Non-official	3

But that's not all: the **`SYS_CONNECT_BY_PATH`** function allows each node to form a full path linking it to a parent node (for which a list of children is built).

Oracle Solution 7.1.1.b (alternative option)

```
SELECT LPAD(' ', 2 * LEVEL, ' ') || "sp_name" "debug",
       "sp_id",
       "sp_parent",
       "sp_name",
       SYS_CONNECT_BY_PATH("sp_name", '/') "path_with_names",
       SYS_CONNECT_BY_PATH("sp_id", ',')   "path_with_ids"
FROM   "site_pages"
START WITH "sp_id" = {parent_node}
CONNECT BY PRIOR "sp_id" = "sp_parent"
ORDER SIBLINGS BY "sp_name"
```

For the page with identifier 2, this query will return the following data.

debug	sp_id	sp_ par- ent	sp_name	path_with_names	path_wit h_ids
For subscribers	2	1	For subscribers	/For subscribers	,2
News	5	2	News	/For subscribers/News	,2,5
Statistics	6	2	Statistics	/For subscribers/Statistics	,2,6
Archive	13	6	Archive	/For subscribers/Statistics/Archive	,2,6,13
Non-official	14	6	Non-official	/For subscribers/Statistics/Non-official	,2,6,14
Current	12	6	Current	/For subscribers/Statistics/Current	,2,6,12

This completes the solution of this problem.

Solution 7.1.1.c[490].

In MySQL before version 8 there are no recursive queries and Common Table Expressions, and even in version 8 there is no way to return a table from a function.

So, let's start with a general solution that works even in "old" versions of this DBMS, built on an algorithm with a loop, in which at each step we move up one level of the hierarchy, until we reach the root node (the reference to the parent is **NULL**).

Here we successively substitute (lines 12-15) the value of the current node identifier and accumulate (lines 16-19) all the obtained values in a string variable, which is the result of the function.

Note that MySQL allows us to specify **RETURN** right in the declaration of the "query returned empty result" handler (line 7), so there is no need to make **RETURN** explicitly further in the function code.

MySQL Solution 7.1.1.c (the function code) (before version 8)

```
1    DELIMITER $$
2    CREATE FUNCTION GET_PATH_TO_ROOT(start_node INT) RETURNS TEXT
3    NOT DETERMINISTIC
4     BEGIN
5      DECLARE path_to_root TEXT;
6      DECLARE current_node INT;
7      DECLARE EXIT HANDLER FOR NOT FOUND RETURN path_to_root;
8
9      SET current_node = start_node;
10     SET path_to_root = start_node;
11     LOOP
12      SELECT  `sp_parent`
13      INTO    current_node
14      FROM    `site_pages`
15      WHERE   `sp_id` = current_node;
16      IF (current_node IS NOT NULL)
17       THEN
18        SET path_to_root = CONCAT(path_to_root, ',', current_node);
19      END IF;
20    END LOOP;
21   END$$
22   DELIMITER ;
```

The function can be used as follows.

MySQL Solution 7.1.1.c (function usage sample)

```
1    SELECT GET_PATH_TO_ROOT(14)
```

In MySQL version 8 and newer, we can use the recursive Common Table Expression approach, the code of which looks like this (the logic of recursive Common Table Expressions is explained in the solution{491} of problem 7.1.1.a{490}).

MySQL | Solution 7.1.1.c (the function code) (version 8 and newer)

```
1   DELIMITER $$
2   CREATE FUNCTION PATH_TO_ROOT(start_node INT)
3   RETURNS VARCHAR(16000) READS SQL DATA
4   BEGIN
5    DECLARE result VARCHAR(16000);
6
7    WITH RECURSIVE `path_to_root` AS
8    (
9     SELECT `sp_id`,
10           `sp_parent`
11    FROM   `site_pages`
12    WHERE  `sp_id` = start_node
13    UNION ALL
14    SELECT `inner`.`sp_id`,
15           `inner`.`sp_parent`
16    FROM   `site_pages` AS `inner`
17      JOIN `path_to_root` ON `path_to_root`.`sp_parent` = `inner`.`sp_id`
18   )
19   SELECT GROUP_CONCAT(`sp_id` SEPARATOR ',') INTO result
20   FROM `path_to_root`;
21
22   RETURN result;
23  END$$
24
25  DELIMITER ;
```

This completes the MySQL solution.

Let's move on to MS SQL Server. Since this DBMS has the ability to return the function result as a table, we will implement both table return and row return behaviors.

In lines 9-17 we implement the same algorithm as in the MySQL solution, but we do not accumulate all of the considered node identifier values into a string, we put them into the resulting table.

In lines 19-28 we check if we need to return the result as a comma-separated set of identifiers, and if so, we collect all the data from the resulting table into a line (lines 21-24 of code), clear the resulting table (line 25 of code), and put the resulting line back into it (lines 26-27 of code).

MS SQL Solution 7.1.1.c (the function code)

```
1    CREATE FUNCTION GET_PATH_TO_ROOT(@current_node INT, @mode VARCHAR(50))
2    RETURNS @path TABLE
3     (
4      id VARCHAR(max)
5     )
6    AS
7    BEGIN
8     DECLARE @all_as_string VARCHAR(max) = '';
9     WHILE (@current_node IS NOT NULL)
10    BEGIN
11     INSERT INTO @path
12     SELECT CAST(@current_node AS VARCHAR);
13
14     SET @current_node = (SELECT [sp_parent]
15                          FROM   [site_pages]
16                          WHERE  [sp_id] = @current_node);
17    END;
18
19    IF (@mode = 'STRING')
20     BEGIN
21      SET @all_as_string  = (SELECT STUFF((SELECT ',' + CAST([id] AS VARCHAR)
22                    FROM   @path
23                    FOR XML PATH(''), TYPE).value('.', 'nvarchar(max)'),
24                    1, 1, ''));
25      DELETE FROM @path;
26      INSERT INTO @path
27      SELECT @all_as_string;
28     END;
29
30    RETURN
31   END;
```

We can check the functionality and correctness of the obtained solution with the following queries, the first of which will return the column of identifiers, and the second will return a table of one cell, which will have a line with identifiers listed separated by commas.

MS SQL Solution 7.1.1.c (function usage sample)

```
1    SELECT * FROM GET_PATH_TO_ROOT(14, 'TABLE');
2    SELECT * FROM GET_PATH_TO_ROOT(14, 'STRING');
```

Since MS SQL Server supports recursive Common Table Expressions, this problem can also be solved using them. For the sake of brevity, we will not create a separate function, but here is a query that even by itself returns exactly the result required by the problem.

It is exactly the **JOIN** in line 11 that provides the recursive behavior we need.

MS SQL Solution 7.1.1.c (alternative option)

```
1    WITH [path_to_root] AS
2    (
3     SELECT [sp_id],
4            [sp_parent]
5     FROM   [site_pages]
6     WHERE  [sp_id] = {starting_node}
7     UNION ALL
8     SELECT [inner].[sp_id],
9            [inner].[sp_parent]
10    FROM   [site_pages] AS [inner]
11      JOIN [path_to_root] ON [path_to_root].[sp_parent] = [inner].[sp_id]
12   )
13   SELECT [sp_id] FROM [path_to_root]
```

This completes the solution for MS SQL Server.

Let's go to Oracle and implement a solution similar to MS SQL Server.

Oracle Solution 7.1.1.c (the function code)

```
1    CREATE TYPE "ptr_tree_node" AS OBJECT("id" VARCHAR(32767));
2    /
3
4    CREATE TYPE "ptr_nodes_collection" AS TABLE OF "ptr_tree_node";
5    /
6
7    CREATE OR REPLACE FUNCTION GET_PATH_TO_ROOT(start_id NUMBER,
8                                                function_mode VARCHAR)
9    RETURN "ptr_nodes_collection"
10   AS
11    result_collection  "ptr_nodes_collection" := "ptr_nodes_collection"();
12    all_as_string VARCHAR(32767);
13    temp_int_value NUMBER(10);
14   BEGIN
15    temp_int_value := start_id;
16
17    WHILE (temp_int_value IS NOT NULL)
18     LOOP
19      IF (function_mode = 'TABLE')
20       THEN
21        result_collection.extend;
22        result_collection(result_collection.last) :=
23                                      "ptr_tree_node"(TO_CHAR(temp_int_value));
24       ELSE
25        all_as_string := all_as_string || ',' || temp_int_value;
26      END IF;
27      SELECT "sp_parent" INTO temp_int_value
28      FROM "site_pages"
29      WHERE "sp_id" = temp_int_value;
30     END LOOP;
31
32     all_as_string := SUBSTR(all_as_string, 2);
33
34     IF (function_mode != 'TABLE')
35     THEN
36      SELECT "ptr_tree_node"(TO_CHAR(all_as_string))
37      BULK COLLECT INTO result_collection
38      FROM DUAL;
39     END IF;
40
41    RETURN result_collection;
42   END;
```

Since we cannot assign a new value to an input parameter of a function in Oracle, we use a temporary variable (lines 13 and 15) to store the values of the node identifiers and use it in the loop.

Unlike MS SQL Server, where inside the function we have a full table to store data, in Oracle we use a collection, so to simplify the code we immediately check in the loop body if we have to return a column of identifiers (then we add them to the collection) or a string with their enumeration (then we accumulate them in a string variable).

In lines 34-39, we check the function mode again and place the line with the identifiers enumeration (from which we remove the first extra comma in line 32) into the collection (which is now empty if the function is called in the identifiers enumeration return mode).

We can check the functionality and correctness of the obtained solution with the following queries, the first of which will return the column of identifiers, and the second will return a table of one cell, which will have a line with identifiers listed separated by commas.

Oracle Solution 7.1.1.c (function usage sample)

```
SELECT *
FROM TABLE(CAST(GET_PATH_TO_ROOT(14, 'TABLE') AS "ptr_nodes_collection"));

SELECT *
FROM TABLE(CAST(GET_PATH_TO_ROOT(14, 'STRING') AS "ptr_nodes_collection"));
```

Since Oracle (like MS SQL Server and MySQL since version 8) supports recursive Common Table Expressions, this problem can also be solved using them. For the sake of brevity, we will not create a separate function, but here is a query that, even by itself, returns exactly the result required by the problem.

Oracle Solution 7.1.1.c (alternative option)

```
WITH "path_to_root" ("sp_id", "sp_parent") AS
(
 SELECT "sp_id",
        "sp_parent"
 FROM   "site_pages"
 WHERE  "sp_id" = {starting_node}
 UNION ALL
 SELECT "inner"."sp_id",
        "inner"."sp_parent"
 FROM   "site_pages" "inner"
   JOIN "path_to_root" ON "path_to_root"."sp_parent" = "inner"."sp_id"
)
SELECT "sp_id" FROM "path_to_root"
```

As in the solution{498} of problem 7.1.1.b{490}, we will consider another option, available only in Oracle. Based on the **CONNECT BY** clause and the **SYS_CONNECT_BY_PATH** function, we have a pretty easy solution to present the result as a string of identifiers.

Oracle Solution 7.1.1.c (alternative option)

```
SELECT SUBSTR(SYS_CONNECT_BY_PATH("sp_id", ','), 2) "path_with_ids"
FROM   "site_pages"
WHERE  "sp_id" = {starting_node}
START WITH "sp_id" = (SELECT "sp_id"
                      FROM   "site_pages"
                      WHERE  "sp_parent" IS NULL)
CONNECT BY PRIOR "sp_id" = "sp_parent"
ORDER SIBLINGS BY "sp_name"
```

The only peculiarity is in the sequence of identifiers placement: in all previously considered variants the root node was on the right (e.g., for node 14 the sequence was 14,6,2,1), but here the root node will be on the left (i.e., we get 1,2,6,14).

This completes the solution of this problem.

Task 7.1.1.TSK.A: create a function that returns a list of identifiers of all child nodes of a given node (e.g., identifiers of all subpages of the “For subscribers” page) to a depth not exceeding the specified one.

Task 7.1.1.TSK.B: create a query to show the entire subtree of a given node in the tree, including the parent node itself (e.g., all the subpages of the “For subscribers” page, including the page itself), with no more than one node from each hierarchy level to be selected.

Task 7.1.1.TSK.C: create a function that returns a list of node identifiers on the path from the tree root to the given node (e.g., identifiers of all nodes on the path from the “Main” page to the “Archive” page).

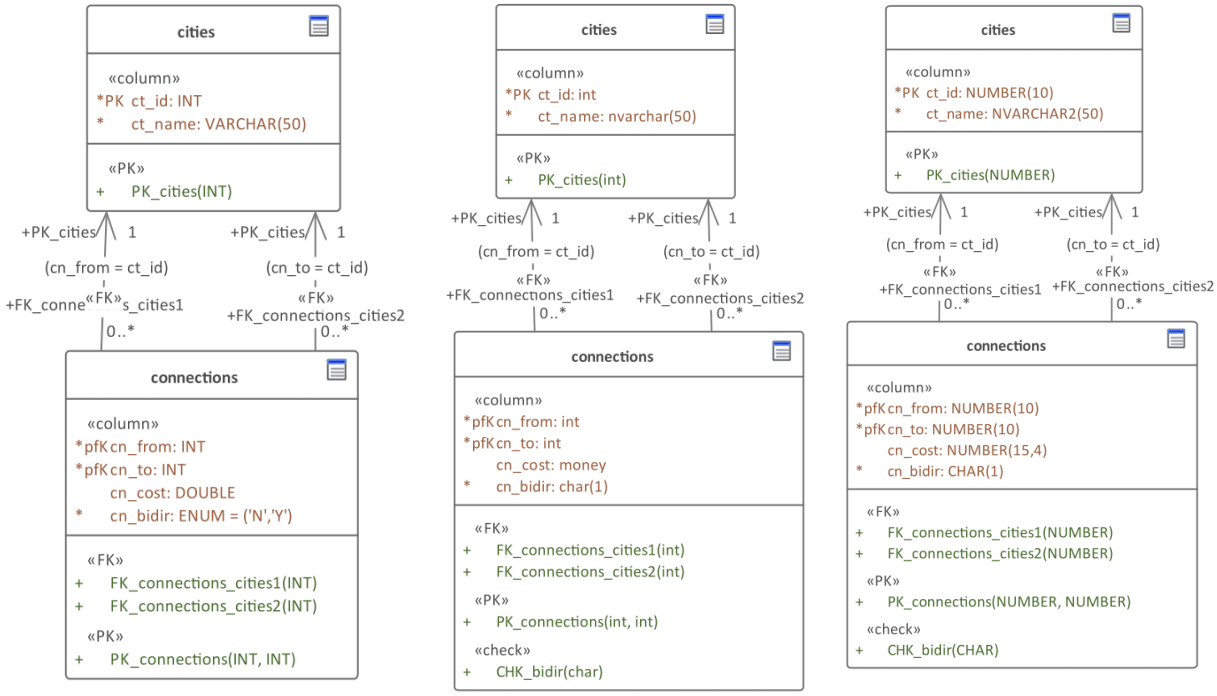
Task 7.1.1.TSK.D: rewrite the solution 7.1.1.b{498} for MySQL using the recursive Common Table Expression.

7.1.2. Example 47: Managing Graph Structures

To solve the problems in this example, we will need a new table, which we will create in the “Exploration” database. This table will store a graph (let’s assume that it will be information about the cost of shipping books from one city to another in order to organize cooperation with other libraries).

There are many ways to store hierarchical structures in relational databases[43], but we use a connections table as one of the most common solutions. The corresponding fragments of the database schema for all three DBMSes are shown in figure 7.c.

The `cn_bidir` field in the `connections` table is a sign that shipping costs are the same whether the book is sent from `cn_from` city to `cn_to` city, or from `cn_to` to `cn_from`.



MySQL MS SQL Server Oracle

Figure 7.c — The `cities` and `connections` tables in all three DBMSes

Let’s save the following data set in the `cities` table.

ct_id	ct_name
1	London
2	Paris
3	Madrid
4	Tokyo
5	Moscow
6	Kiev
7	Minsk
8	Riga
9	Warsaw
10	Berlin

[43] http://www.amazon.com/dp/1558609202/

Let's save the following data set in the `connections` table.

cn_from	cn_to	cn_cost	cn_bidir
1	5	10	Y
1	7	20	N
7	1	25	N
7	2	15	Y
2	6	50	N
6	8	40	Y
8	4	30	N
4	8	35	N
8	9	15	Y
9	1	20	N
7	3	5	N
3	6	5	N

Now that all the data have been prepared, we can move on to the problems.

Problem 7.1.2.a{509}: refine the database model so that for direct routes (without transfers), the cost of travel on which "there" and "back" is the same, in a search query for such a route, we can swap the points of departure and destination.

Problem 7.1.2.b{510}: create a stored procedure that checks the existence of a route (with possible transfers) between two specified cities, and calculates the cost of sending a book by such a route (if it exists).

Expected result 7.1.2.a.

A query like this

```
1    SELECT *
2    FROM   {data_source}
3    WHERE  {from} = 5
4       AND {to} = 1
```

should return this result (note: in the above data there is no route from city 5 to city 1, there is only a route from 1 to 5, but this route is bidirectional):

cn_from	cn_to	cn_cost	cn_bidir
5	1	10	Y

Expected result 7.1.2.b.

E.g., for cities with identifiers 1 and 6, the stored procedure should return the following data.

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	6	31	N	3	1,7,3,6
1	6	85	N	3	1,7,2,6

Solution 7.1.2.a{508}.

To solve this problem, we need to ensure that for bidirectional routes when the search condition is `cn_from=A ADN cn_to=B`, the result also includes routes for which the condition `cn_from=B AND cn_to=A` is satisfied. The easiest way to achieve this effect is to use the views.

On line 8 of the MySQL code below we could have omitted the `DISTINCT` keyword (since by default (without the `ALL` keyword) the `UNION` statement works in `DISTINCT` mode), but it is there for clarity, to emphasize the need to eliminate duplicate records.

MySQL Solution 7.1.2.a

```
1   CREATE OR REPLACE VIEW `connections_bidir`
2   AS
3     SELECT `cn_from`,
4            `cn_to`,
5            `cn_cost`,
6            `cn_bidir`
7     FROM   `connections`
8     UNION DISTINCT
9     SELECT `cn_to`,
10           `cn_from`,
11           `cn_cost`,
12           `cn_bidir`
13    FROM   `connections`
14    WHERE  `cn_bidir` = 'Y'
```

On the example of the MySQL solution, let's see in detail how such a view works. If we select all the data from it, we get the following picture. The lines that appeared as a result of the `UNION` part of the query are marked in gray: records with inverted points of departure and destination were added for all bidirectional routes.

cn_from	cn_to	cn_cost	cn_bidir
1	**5**	**10**	**Y**
1	7	20	N
2	6	50	N
3	6	6	N
4	8	35	N
6	8	40	Y
7	1	25	N
7	2	15	Y
7	3	5	N
8	4	30	N
8	9	15	Y
9	1	20	N
5	**1**	**10**	**Y**
8	6	40	Y
2	7	15	Y
9	8	15	Y

Now such a query execution

MySQL Solution 7.1.2.a (functionality check)

```
1    SELECT *
2    FROM   `connections_bidir`
3    WHERE  `cn_from` = 5
4           AND `cn_to` = 1
```

will return the correct expected result.

cn_from	cn_to	cn_cost	cn_bidir
5	1	10	Y

In MS SQL Server and Oracle, the syntax of **UNOIN** operator does not allow specifying the **DISTINCT** keyword explicitly (line 8 of both queries shown below), but this is not a problem because by default (without **ALL** keyword) **UNION** operator works in **DISTINCT** mode.

MS SQL Solution 7.1.2.a

```
1    CREATE OR ALTER VIEW [connections_bidir]
2    AS
3      SELECT [cn_from],
4             [cn_to],
5             [cn_cost],
6             [cn_bidir]
7      FROM   [connections]
8      UNION
9      SELECT [cn_to],
10            [cn_from],
11            [cn_cost],
12            [cn_bidir]
13     FROM   [connections]
14     WHERE  [cn_bidir] = 'Y'
```

Oracle Solution 7.1.2.a

```
1    CREATE OR REPLACE VIEW "connections_bidir"
2    AS
3      SELECT "cn_from",
4             "cn_to",
5             "cn_cost",
6             "cn_bidir"
7      FROM   "connections"
8      UNION
9      SELECT "cn_to",
10            "cn_from",
11            "cn_cost",
12            "cn_bidir"
13     FROM   "connections"
14     WHERE  "cn_bidir" = 'Y'
```

This completes the solution of this problem.

Solution 7.1.2.b{508}.

To solve this problem, we should start by emphasizing the fact that relational DBMSes are not optimized for storing graph structures and performing such operations on them. That is why the following solutions may seem to be too cumbersome (using classical programming languages one can create much more compact and optimal code).

Let's start traditionally with MySQL and consider two solutions, the first of which maximizes the capabilities of the DBMS, and the second emulates a classic algorithmic solution.

In the first option the following approach[44] is implemented:

- a temporary table is created in RAM (**`ENGINE = HEAP`**) to store the paths found (lines 10-18);
- all data from the **`connections`** table are transferred into the created table, taking into account the bidirectional nature of some connections (lines 21-40; a similar subquery, taking into account bidirectional connections, is used in lines 68-80; in fact, it is nothing but a view body from the solution{509} of problem 7.1.2.a{508});
- the search loop for derived paths (lines 45-92) is performed, in which:
  - the exit condition is the absence of new routes (MySQL function **`ROW_COUNT`** returns the number of records affected by the last data modification operation);
  - the idea of finding new routes is based on adding steps to the already found routes (the end point of the found route coincides with the starting point of the connection between cities, which is checked in the condition of **`JOIN`** in line 81; the condition in lines 82-83 excludes the generation of cyclic routes; the condition in lines 88-89 excludes the endlessly repeated addition of duplicate routes between any two cities).
- after all possible derived paths are built, only those routes are returned as the result of the stored procedure, the points of departure and destination of which coincide with the parameters passed to the stored procedure (lines 94-98).

This solution has two disadvantages:

- preliminary build of all possible routes is redundant and leads to meaningless waste of memory and loss of performance;
- alternative routes can be detected only if they are of the same length (i.e., they are found in the same step of the loop).

MySQL Solution 7.1.2.b (procedure code, the first option)

```
1    DELIMITER $$
2    CREATE PROCEDURE FIND_PATH(IN start_node INT,
3                               IN finish_node INT)
4    BEGIN
5     DECLARE rows_inserted INT DEFAULT 0;
6
7     -- Recreation of a temporary table for storing routes
8     -- (exactly DROP/CREATE in case there was such a table):
9     DROP TABLE IF EXISTS `connections_temp`;
10    CREATE TABLE IF NOT EXISTS `connections_temp`
11    (
12     `cn_from` INT,
13     `cn_to` INT,
14     `cn_cost` DOUBLE,
15     `cn_bidir` CHAR(1),
16     `cn_steps` SMALLINT,
17     `cn_route` VARCHAR(1000)
18    ) ENGINE = MEMORY;
```

[44] https://www.artfulsoftware.com/mysqlbook/sampler/mysqled1ch20.html

MySQL Solution 7.1.2.b (procedure code, the first option) (continued)

```
19    -- Initial filling of the temporary table
20    -- with existing paths:
21    INSERT INTO `connections_temp`
22    SELECT `cn_from`,
23           `cn_to`,
24           `cn_cost`,
25           `cn_bidir`,
26           1,
27           CONCAT(`cn_from`, ',', `cn_to`)
28    FROM   (SELECT `cn_from`,
29                   `cn_to`,
30                   `cn_cost`,
31                   `cn_bidir`
32            FROM   `connections`
33            UNION DISTINCT
34            SELECT `cn_to`,
35                   `cn_from`,
36                   `cn_cost`,
37                   `cn_bidir`
38            FROM   `connections`
39            WHERE  `cn_bidir` = 'Y'
40           ) AS `connections_bidir`;
41
42     -- Filling of the temporary table with
43     -- derived paths:
44     SET rows_inserted = ROW_COUNT();
45     WHILE (rows_inserted > 0)
46     DO
47       INSERT INTO `connections_temp`
48       SELECT `connections_next`.`cn_from`,
49              `connections_next`.`cn_to`,
50              `connections_next`.`cn_cost`,
51              `connections_next`.`cn_bidir`,
52              `connections_next`.`cn_steps`,
53              `connections_next`.`cn_route`
54       FROM (SELECT `connections_temp`.`cn_from` AS `cn_from`,
55                    `connections`.`cn_to` AS `cn_to`,
56                   (`connections_temp`.`cn_cost` +
57                    `connections`.`cn_cost`) AS `cn_cost`,
58                   CASE
59                    WHEN (`connections_temp`.`cn_bidir` = 'Y')
60                     AND (`connections`.`cn_bidir` = 'Y')
61                    THEN 'Y'
62                    ELSE 'N'
63                   END AS  `cn_bidir`,
64                   (`connections_temp`.`cn_steps` + 1) AS `cn_steps`,
65                   CONCAT(`connections_temp`.`cn_route`, ',',
66                          `connections`.`cn_to`) AS `cn_route`
67             FROM `connections_temp`
```

MySQL Solution 7.1.2.b (procedure code, the first option) (continued)

```
68              JOIN (SELECT `cn_from`,
69                           `cn_to`,
70                           `cn_cost`,
71                           `cn_bidir`
72                    FROM   `connections`
73                           UNION DISTINCT
74                    SELECT `cn_to`,
75                           `cn_from`,
76                           `cn_cost`,
77                           `cn_bidir`
78                    FROM   `connections`
79                    WHERE  `cn_bidir` = 'Y'
80                   ) AS `connections`
81                ON `connections_temp`.`cn_to` = `connections`.`cn_from`
82                   AND FIND_IN_SET(`connections`.`cn_to`,
83                                   `connections_temp`.`cn_route`) = 0
84            ) AS `connections_next`
85       LEFT JOIN `connections_temp`
86              ON `connections_next`.`cn_from` = `connections_temp`.`cn_from`
87                 AND `connections_next`.`cn_to` = `connections_temp`.`cn_to`
88       WHERE `connections_temp`.`cn_from` IS NULL
89         AND `connections_temp`.`cn_to` IS NULL;
90
91       SET rows_inserted = ROW_COUNT();
92     END WHILE;
93     -- Retrieving routes that match the search condition:
94     SELECT *
95     FROM `connections_temp`
96     WHERE `cn_from` = start_node
97       AND `cn_to` = finish_node
98     ORDER BY `cn_cost` ASC;
99     DROP TABLE IF EXISTS `connections_temp`;
100  END;
101  $$
102  DELIMITER ;
```

An alternative solution based on the classical algorithm “depth-first search” requires preliminary preparation: the creation of two tables in the RAM (**`ENGINE = MEMORY`**) and setting the maximum level of nesting of recursive calls.

MySQL Solution 7.1.2.b (preparation for the second option of the solution)

```
1    -- Creating a table to store the current path:
2    CREATE TABLE IF NOT EXISTS `current_path`
3     (
4      `cp_id` INT PRIMARY KEY AUTO_INCREMENT,
5      `cp_from` INT,
6      `cp_to` INT,
7      `cp_cost` DOUBLE,
8      `cp_bidir` CHAR(1)
9     ) ENGINE = MEMORY;
10
11   -- Creating a table to store ready-made paths:
12   CREATE TABLE IF NOT EXISTS `final_paths`
13    (
14     `fp_id` DOUBLE,
15     `fp_from` INT,
16     `fp_to` INT,
17     `fp_cost` DOUBLE,
18     `fp_bidir` CHAR(1)
19    ) ENGINE = MEMORY;
20
21   -- Setting the maximum level of nesting of recursive calls:
22   SET @@SESSION.max_sp_recursion_depth = 255;
```

Now we can implement the algorithm:

- if the current path is empty, the departure node is the start node, otherwise the departure node is the arrival point of the last connection in the path (lines 40-54);
- open the cursor to select all connections between cities (lines 18-32, 56);
- for all connections repeat the loop in which:
  - check whether the start node of the connection in question coincides with the current start node (lines 69-72) and, if not, go to the next iteration of the loop;
  - check whether the connection in question is already present in the path (lines 74-80) and whether following this connection leads to a cyclic route (lines 83-88); if any of these conditions is met, go to the next iteration of the loop;
  - check (line 91) if the end node of the connection is the same as the finish node:
    - if it matches, we found a path for which we generate a unique identifier (line 93) and transfer it to the table to store the paths found (lines 95-118), not forgetting to add to the end the connection itself, which we have just considered (lines 108-118);
    - if it does not match, the path has not yet been found, so: add the connection in question to the current path (lines 121-131), perform a recursive call (line 134), after which we remove the last connection from the current path (lines 137-140: MySQL does not allow simultaneously read and delete data from the table, so we put the identifier of the last record in a variable in lines 137-138, and then use in the condition in line 140).

When finished, the **`final_paths`** table will contain all the paths found between the two specified cities.

MySQL | Solution 7.1.2.b (procedure code, the second option)

```
1    DELIMITER $$
2    CREATE PROCEDURE FIND_PATH (IN start_node INT,
3                                IN finish_node INT)
4    BEGIN
5     -- Sign of the cursor loop exit:
6     DECLARE done INT DEFAULT 0;
7
8     -- Variables for extracting data from the cursor:
9     DECLARE cn_from_value INT DEFAULT 0;
10    DECLARE cn_to_value INT DEFAULT 0;
11    DECLARE cn_cost_value DOUBLE DEFAULT 0;
12    DECLARE cn_bidir_value CHAR(1) DEFAULT 0;
13
14    -- Current "start point"
15    -- IMPORTANT: We cannot make this variable @global!
16    DECLARE from_node INT DEFAULT 0;
```

MySQL Solution 7.1.2.b (procedure code, the second option) (continued)

```
17    -- Cursor
18     DECLARE nodes_cursor CURSOR FOR
19     SELECT *
20     FROM (SELECT `cn_from`,
21                  `cn_to`,
22                  `cn_cost`,
23                  `cn_bidir`
24           FROM   `connections`
25           UNION DISTINCT
26           SELECT `cn_to`,
27                  `cn_from`,
28                  `cn_cost`,
29                  `cn_bidir`
30           FROM   `connections`
31           WHERE  `cn_bidir` = 'Y')
32           AS `connections_bidir`;
33           -- Here we can add
34           -- WHERE `cn_from` = from_node
35           -- and then remove
36           -- IF (cn_from_value != from_node)
37
38     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
39
40     IF ((SELECT COUNT(1)
41          FROM `current_path`) = 0)
42      THEN
43       -- If the current path is empty, the departure node
44       -- is the start node
45       SET from_node = start_node;
46      ELSE
47       -- If the current path is NOT empty, the departure node
48       -- is the arrival point of the last connection in the path
49       SET from_node = (SELECT `cp_to`
50                        FROM `current_path`
51                        WHERE `cp_id` = (SELECT MAX(`cp_id`)
52                                         FROM `current_path`)
53                       );
54     END IF;
55
56     OPEN nodes_cursor;
57
58     nodes_loop: LOOP
59     FETCH nodes_cursor INTO cn_from_value,
60                             cn_to_value,
61                             cn_cost_value,
62                             cn_bidir_value;
63      IF done THEN
64       LEAVE nodes_loop;
65      END IF;
66
67      -- The departure node of the connection is not the same as the current
68      -- departure node, skip
69      IF (cn_from_value != from_node)
70       THEN
71        ITERATE nodes_loop;
72       END IF;
```

MySQL Solution 7.1.2.b (procedure code, the second option) (continued)

```
73      -- This connection is already in the current path, skip
74      IF EXISTS (SELECT 1
75                 FROM `current_path`
76                 WHERE `cp_from` = cn_from_value
77                   AND `cp_to` = cn_to_value)
78       THEN
79        ITERATE nodes_loop;
80       END IF;
81
82      -- Such a connection leads to a cycle, skip
83      IF EXISTS (SELECT 1
84                 FROM `current_path`
85                 WHERE `cp_from` = cn_to_value)
86       THEN
87        ITERATE nodes_loop;
88       END IF;
89
90      -- Connection end point coincided with finish node, the path was found
91      IF (cn_to_value = finish_node)
92       THEN
93        SET @rand_value = RAND();
94
95        INSERT INTO `final_paths`
96                    (`fp_id`,
97                     `fp_from`,
98                     `fp_to`,
99                     `fp_cost`,
100                    `fp_bidir`)
101       SELECT @rand_value,
102              `cp_from`,
103              `cp_to`,
104              `cp_cost`,
105              `cp_bidir`
106       FROM   `current_path`;
107
108       INSERT INTO `final_paths`
109                   (`fp_id`,
110                    `fp_from`,
111                    `fp_to`,
112                    `fp_cost`,
113                    `fp_bidir`)
114       VALUES (@rand_value,
115               cn_from_value,
116               cn_to_value,
117               cn_cost_value,
118               cn_bidir_value);
119      ELSE
```

MySQL Solution 7.1.2.b (procedure code, the second option) (continued)

```
120       -- Adding a connection to the current path
121       INSERT INTO `current_path`
122                   (`cp_id`,
123                    `cp_from`,
124                    `cp_to`,
125                    `cp_cost`,
126                    `cp_bidir`)
127       VALUES (NULL,
128               cn_from_value,
129               cn_to_value,
130               cn_cost_value,
131               cn_bidir_value);
132
133       -- Continue to recursively search for the more connections
134       CALL FIND_PATH (start_node, finish_node);
135
136       -- Deleting the last connection from the current path
137       SET @max_cp_id = (SELECT MAX(`cp_id`)
138                         FROM `current_path`);
139       DELETE FROM  `current_path`
140              WHERE `cp_id` = @max_cp_id;
141     END IF;
142    END LOOP nodes_loop;
143    CLOSE nodes_cursor;
144  END;
145  $$
146  DELIMITER ;
```

Let's check how these solutions work by running the following code.

MySQL Solution 7.1.2.b (queries to test the functionality)

```
1    -- For the first solution option:
2    CALL FIND_PATH({start_node}, {finish_node});
3
4    -- For the second solution option:
5    TRUNCATE TABLE `current_path`;
6    TRUNCATE TABLE `final_paths`;
7    CALL FIND_PATH({start_node}, {finish_node});
8    SELECT * FROM `final_paths`;
```

On the data set presented at the beginning of this example, to find a path from city 1 to city 6, both solutions return the same (though differently presented) results.

The result of the first solution option:

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	6	30	N	3	1,7,3,6
1	6	85	N	3	1,7,2,6

The result of the second solution option:

fp_id	fp_from	fp_to	fp_cost	fp_bidir
0.42358866466543516	1	7	20	N
0.42358866466543516	7	2	15	Y
0.42358866466543516	2	6	50	N
0.34713028031134074	1	7	20	N
0.34713028031134074	7	3	5	N
0.34713028031134074	3	6	5	N

But if we put the following data in the `connections` table

cn_from	cn_to	cn_cost	cn_bidir
1	3	100	N
1	5	100	N
3	5	20	N
5	3	200	N

and search for the path between cities 1 and 5, the results will be different.

The result of the first solution option:

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	5	100	N	1	1,5

The result of the second solution option:

fp_id	fp_from	fp_to	fp_cost	fp_bidir
0.6203666074391143	1	3	100	N
0.6203666074391143	3	5	20	N
0.2569376912498266	1	5	100	N

As mentioned above, the first version of the solution does not find alternative paths of different lengths, while the second option manages this.

This completes the MySQL solution.

Let's go to MS SQL Server and implement the same solution as above for MySQL.

Algorithm of the first solution option:

- delete (if it exists) and create a temporary table to store the paths found (lines 7-19);
- all data from the `connections` table are transferred into the created table, taking into account the bidirectional nature of some connections (lines 23-42; a similar subquery, taking into account bidirectional connections, is used in lines 69-81; in fact, it is nothing but a view body from the solution{509} of problem 7.1.2.a{508});
- the search loop for derived paths (lines 46-93) is performed, in which:
  - exit condition is absence of new routes (MS SQL Server `@@ROWCOUNT` variable contains number of records affected by the last data modification operation);
  - the idea of finding new routes is based on adding steps to the already found routes (the finish point of the found route coincides with the start node of the connection between cities, which is checked in the union condition in line 82; the condition in lines 83-84 excludes the generation of cyclic routes; the condition in lines 89-90 excludes endlessly duplicated routes between any two cities).
- after all possible derived paths are built, only those routes are returned as the result of the stored procedure call, the start points and final points of which coincide with the parameters passed to the stored procedure (lines 96-100).

Note how the route placed in the `cn_route` field of the `connections_temp` table is formed and analyzed: MS SQL Server has no direct analogue of the MySQL function `FIND_IN_SET`, and when searching for an occurrence of a substring in the string there is a chance, e.g., to "find" the number 12 in the number 123, etc.

Therefore, each value of the identifier is taken in square brackets (the path will look like "[1][7][3][6]") and the search is also performed with prior enclosing of the sought identifier in square brackets, which guarantees the absence of false positives.

MS SQL Solution 7.1.2.b (procedure code, the first option)

```
CREATE PROCEDURE FIND_PATH
                 @start_node INT,
                 @finish_node INT
AS
 DECLARE @rows_inserted INT = 0;

 -- Creating a temporary table to store routes:
 IF OBJECT_ID('tempdb.dbo.#connections_temp', 'U') IS NOT NULL
  DROP TABLE #connections_temp;

 CREATE TABLE #connections_temp
 (
  [cn_from] INT,
  [cn_to] INT,
  [cn_cost] DOUBLE PRECISION,
  [cn_bidir] CHAR(1),
  [cn_steps] SMALLINT,
  [cn_route] VARCHAR(1000)
 );

 -- Initial filling of the temporary table
 -- with existing paths:
 INSERT INTO #connections_temp
 SELECT [cn_from],
        [cn_to],
        [cn_cost],
        [cn_bidir],
        1,
        CONCAT('[', [cn_from], '][', [cn_to], ']')
 FROM   (SELECT [cn_from],
                [cn_to],
                [cn_cost],
                [cn_bidir]
         FROM   [connections]
         UNION
         SELECT [cn_to],
                [cn_from],
                [cn_cost],
                [cn_bidir]
         FROM   [connections]
         WHERE  [cn_bidir] = 'Y'
        ) AS [connections_bidir];
```

MS SQL Solution 7.1.2.b (procedure code, the first option) (continued)

```
43      -- Filling the temporary table with
44      -- derived paths:
45      SET @rows_inserted = @@ROWCOUNT;
46      WHILE (@rows_inserted > 0)
47      BEGIN
48        INSERT INTO #connections_temp
49        SELECT [connections_next].[cn_from],
50               [connections_next].[cn_to],
51               [connections_next].[cn_cost],
52               [connections_next].[cn_bidir],
53               [connections_next].[cn_steps],
54               [connections_next].[cn_route]
55        FROM (SELECT #connections_temp.[cn_from] AS [cn_from],
56                     [connections].[cn_to] AS [cn_to],
57                    (#connections_temp.[cn_cost] +
58                     [connections].[cn_cost]) AS [cn_cost],
59                    CASE
60                     WHEN (#connections_temp.[cn_bidir] = 'Y')
61                      AND ([connections].[cn_bidir] = 'Y')
62                     THEN 'Y'
63                     ELSE 'N'
64                    END AS  [cn_bidir],
65                    (#connections_temp.[cn_steps] + 1) AS [cn_steps],
66                    CONCAT(#connections_temp.[cn_route], '[',
67                           [connections].[cn_to], ']') AS [cn_route]
68              FROM #connections_temp
69              JOIN (SELECT [cn_from],
70                           [cn_to],
71                           [cn_cost],
72                           [cn_bidir]
73                    FROM   [connections]
74                           UNION
75                    SELECT [cn_to],
76                           [cn_from],
77                           [cn_cost],
78                           [cn_bidir]
79                    FROM   [connections]
80                    WHERE  [cn_bidir] = 'Y'
81                   ) AS [connections]
82                ON #connections_temp.[cn_to] = [connections].[cn_from]
83                   AND CHARINDEX(CONCAT('[', [connections].[cn_to], ']'),
84                                 #connections_temp.[cn_route]) = 0
85             ) AS [connections_next]
86        LEFT JOIN #connections_temp
87               ON [connections_next].[cn_from] = #connections_temp.[cn_from]
88                  AND [connections_next].[cn_to] = #connections_temp.[cn_to]
89        WHERE #connections_temp.[cn_from] IS NULL
90          AND #connections_temp.[cn_to] IS NULL;
91
92        SET @rows_inserted = @@ROWCOUNT;
93      END; -- WHILE
94
95      -- Retrieving routes that match the search condition:
96      SELECT *
97      FROM #connections_temp
98      WHERE [cn_from] = @start_node
99        AND [cn_to] = @finish_node
100     ORDER BY [cn_cost] ASC;
101   GO
```

For the second solution, as with MySQL, we need auxiliary tables to store the current and final paths.

MS SQL Solution 7.1.2.b (preparation for the second solution option)

```
1    -- Creating a table to store the current path:
2    IF OBJECT_ID('tempdb.dbo.#current_path', 'U') IS NOT NULL
3      DROP TABLE #current_path;
4    CREATE TABLE #current_path
5     (
6      [cp_id] INT NOT NULL IDENTITY (1, 1),
7      [cp_from] INT,
8      [cp_to] INT,
9      [cp_cost] DOUBLE PRECISION,
10     [cp_bidir] CHAR(1)
11    );
12
13   -- Creating a table to store final paths:
14   IF OBJECT_ID('tempdb.dbo.#final_paths', 'U') IS NOT NULL
15     DROP TABLE #final_paths;
16   CREATE TABLE #final_paths
17    (
18     [fp_id] DOUBLE PRECISION,
19     [fp_from] INT,
20     [fp_to] INT,
21     [fp_cost] DOUBLE PRECISION,
22     [fp_bidir] CHAR(1)
23    );
```

The algorithm of the second solution option:

- if the current path is empty, the departure node is the start node, otherwise the departure node is the arrival point of the last connection in the path (lines 34-45);
- open the cursor to select all connections between cities (lines 18-32, 47);
- for all connections repeat the loop in which:
    - check whether the start node of the connection in question coincides with the current start node (lines 60-61) and, if not, go to the next iteration of the loop;
    - check whether the connection in question is already present in the path (lines 63-67) and whether following this connection leads to a cyclic route (lines 70-73); if any of these conditions is met, go to the next iteration of the loop;
    - check (line 76), if the end node of the connection is the same as the finish node:
        - if it matches, we found a path for which we generate a unique identifier (line 78) and transfer it to the table to store the paths found (lines 79-90), not forgetting to add to the end the connection itself, which we have just considered (lines 91-101);
        - if it does not match, the path has not yet been found, so: add the connection in question to the current path (lines 106-114), perform a recursive call (line 117), after which we remove the last connection from the current path (lines 120-121).

When finished, the **`final_paths`** table will contain all the paths found between the two specified cities.

MS SQL Solution 7.1.2.b (procedure code, the second option)

```
1    CREATE PROCEDURE FIND_PATH
2                     @start_node INT,
3                     @finish_node INT
4    AS
5     -- Variables for extracting data from the cursor:
6     DECLARE @cn_from_value INT = 0;
7     DECLARE @cn_to_value INT = 0;
8     DECLARE @cn_cost_value DOUBLE PRECISION = 0;
9     DECLARE @cn_bidir_value CHAR(1) = '0';
10
11    -- Current "from node"
12    DECLARE @from_node INT = 0;
13
14    -- Identifier of the path found
15    DECLARE @rand_value DOUBLE PRECISION = 0;
16
17    -- Cursor to loop through connections between cities
18    DECLARE nodes_cursor CURSOR LOCAL FAST_FORWARD FOR
19     SELECT *
20      FROM (SELECT [cn_from],
21                   [cn_to],
22                   [cn_cost],
23                   [cn_bidir]
24            FROM   [connections]
25            UNION
26            SELECT [cn_to],
27                   [cn_from],
28                   [cn_cost],
29                   [cn_bidir]
30            FROM   [connections]
31            WHERE  [cn_bidir] = 'Y'
32           ) AS [connections_bidir];
33
34    IF ((SELECT COUNT(1)
35         FROM #current_path) = 0)
36      -- If the current path is empty, the departure node is the start node
37      SET @from_node = @start_node;
38     ELSE
39      -- If the current path is NOT empty, the departure node
40      -- is the arrival point of the last connection in the path
41      SET @from_node = (SELECT [cp_to]
42                        FROM   #current_path
43                        WHERE  [cp_id] = (SELECT MAX([cp_id])
44                                          FROM #current_path)
45                       );
46
47    OPEN nodes_cursor;
48
49    WHILE (1 = 1)
50     BEGIN
51      FETCH NEXT FROM nodes_cursor INTO @cn_from_value,
52                                        @cn_to_value,
53                                        @cn_cost_value,
54                                        @cn_bidir_value;
55      IF (@@FETCH_STATUS != 0)
56       BREAK;
57
58      -- The connection departure node is not the same as the current
59      -- departure node, skip
60      IF (@cn_from_value != @from_node)
61       CONTINUE;
```

MS SQL Solution 7.1.2.b (procedure code, the second option) (continued)

```
62      -- This connection is already in the current path, skip
63      IF EXISTS (SELECT 1
64                 FROM #current_path
65                 WHERE [cp_from] = @cn_from_value
66                   AND [cp_to] = @cn_to_value)
67       CONTINUE;
68
69      -- Such a connection leads to a loop, skip
70      IF EXISTS (SELECT 1
71                 FROM #current_path
72                 WHERE [cp_from] = @cn_to_value)
73       CONTINUE;
74
75     -- Connection end point coincided with finish node, the path was found
76      IF (@cn_to_value = @finish_node)
77       BEGIN
78        SET @rand_value = RAND();
79        INSERT INTO #final_paths
80                    ([fp_id],
81                     [fp_from],
82                     [fp_to],
83                     [fp_cost],
84                     [fp_bidir])
85        SELECT @rand_value,
86               [cp_from],
87               [cp_to],
88               [cp_cost],
89               [cp_bidir]
90        FROM   #current_path;
91        INSERT INTO #final_paths
92                    ([fp_id],
93                     [fp_from],
94                     [fp_to],
95                     [fp_cost],
96                     [fp_bidir])
97        VALUES (@rand_value,
98                @cn_from_value,
99                @cn_to_value,
100               @cn_cost_value,
101               @cn_bidir_value);
102      END
103     ELSE
104      BEGIN
105       -- Adding a connection to the current path
106       INSERT INTO #current_path
107                   ([cp_from],
108                    [cp_to],
109                    [cp_cost],
110                    [cp_bidir])
111       VALUES (@cn_from_value,
112               @cn_to_value,
113               @cn_cost_value,
114               @cn_bidir_value);
115
116      -- Continue to recursively search for more connections
117      EXEC FIND_PATH @start_node, @finish_node;
118
119      -- Deleting the last connection from the current path
120      DELETE FROM  #current_path
121             WHERE [cp_id] = (SELECT MAX([cp_id]) FROM #current_path);
122     END;
123    END;
124   CLOSE nodes_cursor;
125   DEALLOCATE nodes_cursor;
126  GO
```

Let's check how these solutions work by running the following code.

MS SQL Solution 7.1.2.b (queries to test the functionality)

```
-- For the first solution option:
EXEC FIND_PATH {start_node}, {finish_node};

-- For the second solution option:
TRUNCATE TABLE #current_path;
TRUNCATE TABLE #final_paths;
EXEC FIND_PATH {start_node}, {finish_node};
SELECT * FROM #final_paths;
```

The behavior of MS SQL Server is completely equivalent to that of MySQL.

On the data set presented at the beginning of this example, both solutions return the same (though differently presented) results for finding a path from city 1 to city 6.

The result of the first solution option:

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	6	30	N	3	[1][7][3][6]
1	6	85	N	3	[1][7][2][6]

The result of the second solution option:

fp_id	fp_from	fp_to	fp_cost	fp_bidir
0.159113517642648	1	7	20	N
0.159113517642648	7	2	15	Y
0.159113517642648	2	6	50	N
0.716279602109358	1	7	20	N
0.716279602109358	7	3	5	N
0.716279602109358	3	6	5	N

But if we put the following data in the `connections` table

cn_from	cn_to	cn_cost	cn_bidir
1	3	100	N
1	5	100	N
3	5	20	N
5	3	200	N

and search for the path between cities 1 and 5, the results will be different.

The result of the first solution option:

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	5	100	N	1	[1][5]

The result of the second solution option:

fp_id	fp_from	fp_to	fp_cost	fp_bidir
0.948062188221448	1	3	100	N
0.948062188221448	3	5	20	N
0.562740117282937	1	5	100	N

So, as mentioned above, in MS SQL Server the first solution option fails to find alternative paths of different lengths, too, while the second one does it.

This completes the solution for MS SQL Server.

Let's go to Oracle and implement the solution presented above for MySQL and MS SQL Server.

Unlike the other two DBMSes, Oracle does not allow us to compile a stored procedure that refers to non-existent objects inside it, even if the objects are created in the same procedure a few lines above. This limitation can be bypassed by **`EXECUTE IMMEDIATE`** clause and other fancy ways, but to keep the code simple, we will put the creation of the temporary table that the stored procedure works with into a separate code.

Oracle — Solution 7.1.2.b (preparation for the first solution option)

```
1    CREATE GLOBAL TEMPORARY TABLE "connections_temp"
2     (
3      "cn_from" NUMBER(10),
4      "cn_to" NUMBER(10),
5      "cn_cost" DOUBLE PRECISION,
6      "cn_bidir" CHAR(1),
7      "cn_steps" NUMBER(5),
8      "cn_route" VARCHAR(1000)
9     )
10    ON COMMIT PRESERVE ROWS;
```

Algorithm of the first solution option:

- all data from the **`connections`** table are transferred to the temporary table created before compiling the stored procedure, taking into account the bidirectional nature of some connections (lines 10-28; a similar subquery, taking into account bidirectional connections, is used in lines 55-67; in fact, it is nothing but a view body from the solution{509} of problem 7.1.2.a{508});
- the search loop for derived paths (lines 32-79) is performed, in which:
  - the exit condition is the absence of new routes (in Oracle the **`SQL%ROWCOUNT`** variable contains the number of records affected by the last data modification operation);
  - the idea of finding new routes is based on adding steps to the already found routes (the end point of the found route coincides with the starting point of the connection between cities, which is checked in the condition of union in line 68; the condition in lines 69-70 excludes the generation of cyclic routes; the condition in lines 75-76 excludes the endlessly repeated addition of duplicate routes between any two cities).
- after all possible derived paths are built, only those routes are returned as the result of the stored procedure, the start points and final points of which coincide with the parameters passed to the stored procedure (lines 82-87).

Pay attention to how the route, placed in the **`cn_route`** field of the **`connections_temp`** table, is formed and analyzed: in Oracle (as well as in MS SQL Server) there is no direct analogue of the MySQL function **`FIND_IN_SET`**, and when searching for occurrence of a substring in a string there is a chance, e.g., to "find" the number 12 in the number 123, etc.

Therefore, each value of the identifier is enclosed in square brackets (the path will be something like "[1][7][3][6]") and the search is also performed with a preliminary enclosure of the sought identifier in square brackets, which guarantees the absence of false positives.

Oracle | Solution 7.1.2.b (procedure code, the first option)

```
CREATE OR REPLACE PROCEDURE FIND_PATH (start_node IN NUMBER,
                                       finish_node IN NUMBER,
                                       final_paths OUT SYS_REFCURSOR)
AS
 rows_inserted NUMBER := 0;
BEGIN

 -- Initial filling of the temporary table with existing paths:
 INSERT INTO "connections_temp"
 SELECT "cn_from",
        "cn_to",
        "cn_cost",
        "cn_bidir",
        1,
        ('[' || "cn_from" || '][' || "cn_to" || ']')
 FROM   (SELECT "cn_from",
                "cn_to",
                "cn_cost",
                "cn_bidir"
         FROM   "connections"
         UNION
         SELECT "cn_to",
                "cn_from",
                "cn_cost",
                "cn_bidir"
         FROM   "connections"
         WHERE  "cn_bidir" = 'Y'
        ) "connections_bidir";

 -- Filling the temporary table with derived paths:
 rows_inserted := SQL%ROWCOUNT;
 WHILE (rows_inserted > 0)
  LOOP
   INSERT INTO "connections_temp"
   SELECT "connections_next"."cn_from",
          "connections_next"."cn_to",
          "connections_next"."cn_cost",
          "connections_next"."cn_bidir",
          "connections_next"."cn_steps",
          "connections_next"."cn_route"
   FROM (SELECT "connections_temp"."cn_from" AS "cn_from",
                  "connections"."cn_to" AS "cn_to",
                ("connections_temp"."cn_cost" +
                 "connections"."cn_cost") AS "cn_cost",
                CASE
                 WHEN ("connections_temp"."cn_bidir" = 'Y')
                  AND ("connections"."cn_bidir" = 'Y')
                 THEN 'Y'
                 ELSE 'N'
                END AS  "cn_bidir",
                ("connections_temp"."cn_steps" + 1) AS "cn_steps",
                ("connections_temp"."cn_route" || '[' ||
                 "connections"."cn_to" || ']') AS "cn_route"
          FROM "connections_temp"
```

Oracle Solution 7.1.2.b (procedure code, the first option) (continued)

```
55              JOIN (SELECT "cn_from",
56                           "cn_to",
57                           "cn_cost",
58                           "cn_bidir"
59                    FROM   "connections"
60                           UNION
61                    SELECT "cn_to",
62                           "cn_from",
63                           "cn_cost",
64                           "cn_bidir"
65                    FROM   "connections"
66                    WHERE  "cn_bidir" = 'Y'
67                   ) "connections"
68                ON "connections_temp"."cn_to" = "connections"."cn_from"
69                   AND INSTR("connections_temp"."cn_route",
70                             '[' || "connections"."cn_to" || ']') = 0
71            ) "connections_next"
72       LEFT JOIN "connections_temp"
73              ON "connections_next"."cn_from" = "connections_temp"."cn_from"
74                 AND "connections_next"."cn_to" = "connections_temp"."cn_to"
75       WHERE "connections_temp"."cn_from" IS NULL
76         AND "connections_temp"."cn_to" IS NULL;
77
78       rows_inserted := SQL%ROWCOUNT;
79     END LOOP;
80
81     -- Retrieving paths that match the search condition:
82     OPEN final_paths FOR
83     SELECT *
84     FROM "connections_temp"
85     WHERE "cn_from" = start_node
86       AND "cn_to" = finish_node
87     ORDER BY "cn_cost" ASC;
88   END;
```

For the second solution option, as with MySQL and MS SQL Server, we need auxiliary tables to store the current and final paths.

Oracle Solution 7.1.2.b (preparation for the second solution option)

```
1     -- Creating a table to store the current path:
2    CREATE GLOBAL TEMPORARY TABLE "current_path"
3     (
4      "cp_id" NUMBER(10),
5      "cp_from" NUMBER(10),
6      "cp_to" NUMBER(10),
7      "cp_cost" NUMBER(15,4),
8      "cp_bidir" CHAR(1)
9     );
10
11   -- Creating a table to store ready-made paths:
12   CREATE GLOBAL TEMPORARY TABLE "final_paths"
13    (
14     "fp_id" NUMBER(15,4),
15     "fp_from" NUMBER(15,4),
16     "fp_to" NUMBER(15,4),
17     "fp_cost" NUMBER(15,4),
18     "fp_bidir" CHAR(1)
19    );
```

Algorithm of the second solution option:

- if the current path is empty, the departure node is the start node, otherwise the departure node is the arrival point of the last connection in the path (lines 26-40; note how Oracle checks query results for emptiness, the classic expression `IF (NOT) EXISTS` does not work here);
- open the cursor to select all connections between cities (lines 8-24, 42);
- for all connections repeat the loop in which:
  - check whether the start node of the connection in question coincides with the current start node (lines 46-49) and, if not, go to the next iteration of the loop;
  - check whether the connection in question is already present in the path (lines 52-60) and whether following this connection leads to a cyclic route (lines 63-70); if any of these conditions is met, go to the next iteration of the loop;
  - check (line 73), if the end node of the connection is the same as the finish node:
    - if it matches, we found a path for which we generate a unique identifier (line 75) and transfer it to the table to store the paths found (lines 77-88), not forgetting to add to the end the connection itself, which we have just considered (lines 89-99);
    - if it does not match, the path has not yet been found, so: add the connection in question to the current path (lines 102-113), perform a recursive call (line 116), after which we remove the last connection from the current path (lines 119-121).

When finished, the `final_paths` table will contain all the paths found between the two specified cities.

Note how lines 108-109 emulate an autoincrementable primary key without using a trigger.

Oracle | Solution 7.1.2.b (procedure code, the second option)

```
1    CREATE OR REPLACE PROCEDURE FIND_PATH (start_node IN NUMBER,
2                                           finish_node IN NUMBER)
3    AS
4     from_node NUMBER(10) := 0;
5     rows_count NUMBER(10) := 0;
6     rand_value NUMBER(15,4) := 0;
7
8     CURSOR nodes_cursor IS
9      SELECT "cn_from",
10            "cn_to",
11            "cn_cost",
12            "cn_bidir"
13     FROM (SELECT "cn_from",
14                  "cn_to",
15                  "cn_cost",
16                  "cn_bidir"
17           FROM   "connections"
18           UNION
19           SELECT "cn_to",
20                  "cn_from",
21                  "cn_cost",
22                  "cn_bidir"
23           FROM   "connections"
24           WHERE  "cn_bidir" = 'Y');
```

Oracle Solution 7.1.2.b (procedure code, the second option) (continued)

```
25  BEGIN
26   SELECT COUNT(1) INTO rows_count
27   FROM "current_path";
28
29   IF (rows_count = 0)
30    THEN
31     -- If the current path is empty, the departure node is the start node
32     from_node := start_node;
33    ELSE
34     -- If the current path is NOT empty, the departure node
35     -- is the arrival point of the last connection in the path
36     SELECT "cp_to" INTO from_node
37     FROM "current_path"
38     WHERE "cp_id" = (SELECT MAX("cp_id")
39                      FROM "current_path");
40   END IF;
41
42   FOR one_link IN nodes_cursor
43    LOOP
44     -- The connection departure node is not the same as the current
45     -- departure node, skip
46     IF (one_link."cn_from" != from_node)
47     THEN
48      CONTINUE;
49     END IF;
50
51     -- This connection is already in the current path, skip
52     SELECT COUNT(1) INTO rows_count
53     FROM (SELECT 1
54           FROM  "current_path"
55           WHERE "cp_from" = one_link."cn_from"
56             AND "cp_to" = one_link."cn_to");
57     IF (rows_count > 0)
58      THEN
59       CONTINUE;
60      END IF;
61
62     -- Such a connection leads to a cycle, skip
63     SELECT COUNT(1) INTO rows_count
64     FROM (SELECT 1
65           FROM  "current_path"
66           WHERE "cp_from" = one_link."cn_to");
67     IF (rows_count > 0)
68      THEN
69       CONTINUE;
70      END IF;
71
72     -- Connection end point coincided with finish node, the path was found
73     IF (one_link."cn_to" = finish_node)
74      THEN
75       rand_value := DBMS_RANDOM.VALUE(1,10);
76
77       INSERT INTO "final_paths"
78                   ("fp_id",
79                    "fp_from",
80                    "fp_to",
81                    "fp_cost",
82                    "fp_bidir")
83       SELECT rand_value,
84              "cp_from",
85              "cp_to",
86              "cp_cost",
87              "cp_bidir"
88       FROM   "current_path";
```

Oracle Solution 7.1.2.b (procedure code, the second option) (continued)

```
89        INSERT INTO "final_paths"
90                    ("fp_id",
91                     "fp_from",
92                     "fp_to",
93                     "fp_cost",
94                     "fp_bidir")
95        VALUES (rand_value,
96                one_link."cn_from",
97                one_link."cn_to",
98                one_link."cn_cost",
99                one_link."cn_bidir");
100     ELSE
101       -- Adding a connection to the current path
102       INSERT INTO "current_path"
103                   ("cp_id",
104                    "cp_from",
105                    "cp_to",
106                    "cp_cost",
107                    "cp_bidir")
108       VALUES (NVL((SELECT MAX("cp_id") + 1
109                    FROM "current_path"), 1),
110               one_link."cn_from",
111               one_link."cn_to",
112               one_link."cn_cost",
113               one_link."cn_bidir");
114
115       -- Continue to recursively search for more connections
116       FIND_PATH (start_node, finish_node);
117
118       -- Deleting the last connection from the current path
119       DELETE FROM  "current_path"
120              WHERE "cp_id" = (SELECT MAX("cp_id")
121                               FROM "current_path");
122     END IF;
123   END LOOP;
124 END;
```

Let's check how these solutions work by running the code below. The behavior of Oracle is fully equivalent to that of MySQL and MS SQL Server.

On the data set presented at the beginning of this example, to find a path from city 1 to city 6, both solutions return the same (though differently presented) results.

The result of the first solution option:

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	6	30	N	3	[1][7][3][6]
1	6	85	N	3	[1][7][2][6]

The result of the second solution option:

fp_id	fp_from	fp_to	fp_cost	fp_bidir
4.5847	1	7	20	N
4.5847	7	2	15	Y
4.5847	2	6	50	N
8.5731	1	7	20	N
8.5731	7	3	5	N
8.5731	3	6	5	N

Oracle | Solution 7.1.2.b (queries to test the functionality)

```
-- For the first solution option::
TRUNCATE TABLE "connections_temp";
DECLARE
  fp SYS_REFCURSOR;
  cn_from NUMBER(10);
  cn_to NUMBER(10);
  cn_cost DOUBLE PRECISION;
  cn_bidir CHAR(1);
  cn_steps NUMBER(5);
  cn_route VARCHAR(1000);
BEGIN
  FIND_PATH({start_node}, {finish_node}, fp);
  LOOP
    FETCH fp INTO cn_from,
                  cn_to,
                  cn_cost,
                  cn_bidir,
                  cn_steps,
                  cn_route;
    EXIT WHEN fp%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(cn_from || ' | ' ||
                         cn_to || ' | ' ||
                         cn_cost || ' | ' ||
                         cn_bidir || ' | ' ||
                         cn_steps || ' | ' ||
                         cn_route);
  END LOOP;
  CLOSE fp;
END;

-- For the second solution option::
TRUNCATE TABLE "current_path";
TRUNCATE TABLE "final_paths";
BEGIN
 FIND_PATH({start_node}, {finish_node});
END;
SELECT * FROM "final_paths";
```

But if we put the following data in the **`connections`** table

cn_from	cn_to	cn_cost	cn_bidir
1	3	100	N
1	5	100	N
3	5	20	N
5	3	200	N

and search for the path between cities 1 and 5, the results will be different.

The result of the first solution option::

cn_from	cn_to	cn_cost	cn_bidir	cn_steps	cn_route
1	5	100	N	1	[1][5]

The result of the second solution option::

fp_id	fp_from	fp_to	fp_cost	fp_bidir
3.5748	1	3	100	N
3.5748	3	5	20	N
5.881	1	5	100	N

So, as mentioned above, in Oracle the first solution option also fails to find alternative paths of different lengths, while the second solution option manages it.

This completes the solution of this problem.

Task 7.1.2.TSK.A: implement the solution{510} of problem 7.1.2.b{508} not using a stored procedure, but using a stored function in those DBMSes that support the corresponding functionality.

Task 7.1.2.TSK.B: compare the performance of the first and second solutions options{510} of problem 7.1.2.b{508} for MySQL.

Task 7.1.2.TSK.C: implement the first solution option{510} of problem 7.1.2.b{508} for Oracle using the `CONNECT BY` clause.

7.2. Operations with Databases

7.2.1. Example 48: Database Backup and Restore

Problem 7.2.1.a{533}: create a batch file to automate database backup.

Problem 7.2.1.b{535}: create a batch file to automate database restore from a backup.

Problem 7.2.1.c{536}: create a batch file to automate the creation of a database working copy.

Expected result 7.2.1.a:

Since the solution itself is the expected result, see solution 7.2.1.a{533}.

Expected result 7.2.1.b:

Since the solution itself is the expected result, see solution 7.2.1.b{535}.

Expected result 7.2.1.c:

Since the solution itself is the expected result, see solution 7.2.1.c{536}.

Solution 7.2.1.a{533}.

In this case, by “database backup” we mean a set of SQL-commands, which, if executed (possibly with prior editing) on an arbitrary server, will give us a full working copy of the original database.

We will **not** talk here about backing up a specific server and/or its specific databases, scheduled backups, incremental backups, and other ways to automate backups and restores within the same system.

In MySQL, there is a utility called `mysqldump` to perform the corresponding operation. Make sure that the path to it is in the `PATH` environment variable, and then the following batch file will create a full backup of the specified database.

MySQL Solution 7.2.1.a (cmd file) (general concept)

```
1    mysqldump -uUSERNAME -pPASSWORD DATABASE > DATABASE.sql
```

MySQL Solution 7.2.1.a (cmd file) (example)

```
1    mysqldump -uabc -pdef library > library.sql
```

In MS SQL Server we can use MS SQL Server Management Studio[45] or SQL Server Management Objects[46], to back up our database as SQL code, but it is not as simple and elegant as MySQL. Still, it is possible to get a binary backup using the following batch file (make sure that the path to the `sqlcmd` utility is in the `PATH` environment variable).

The code below must be written in **one line** and the file extension must be "**bak**".

MS SQL — Solution 7.2.1.a (cmd file) (general concept)

```
1    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -Q "BACKUP DATABASE [DATA-
     BASE] TO DISK='FULL_PATH_TO_THE_FILE.bak'"
```

MS SQL — Solution 7.2.1.a (cmd file) (example)

```
1    sqlcmd -U abc -P def -S COMP\MSSQLSRV -Q "BACKUP DATABASE [library] TO
     DISK='C:\backup\library.bak'"
```

Oracle (as well as MS SQL Server) does not have an easy way to get from the command line the full-fledged SQL code with a full database backup (but we can use Oracle SQL Developer[47]), still, there is the `expdp` utility which creates a binary backup. We can use the code presented below (make sure to put the path for `expdp` in the `PATH` environment variable).

Also note that the first two lines in the command file are responsible for executing SQL queries, which make the necessary preparations at the DBMS level.

Oracle — Solution 7.2.1.a (cmd file) (general concept)

```
1    @echo GRANT CREATE ANY DIRECTORY TO USERNAME; | sqlplus USERNAME/PASS-
2    WORD@SERVER
     @echo CREATE DIRECTORY LOGICAL_DIRECTORY_NAME AS 'FULL_PATH_TO_THE_DIRECTO-
3    RY'; | sqlplus USERNAME/PASSWORD@SERVER
     expdp USERNAME/PASSWORD@SERVER schemas=USERNAME directory=LOGICAL_DIREC-
     TORY_NAME dumpfile=FILE_NAME.dmp logfile=LOG_FILE_NAME.log
```

Oracle — Solution 7.2.1.a (cmd file) (example)

```
1    @echo GRANT CREATE ANY DIRECTORY TO abc; | sqlplus abc/def@COMP
2    @echo CREATE DIRECTORY LIBRARY_BACKUP AS 'C:\backup'; | sqlplus
     abc/def@COMP
3    expdp abc/def@COMP schemas=abc directory=LIBRARY_BACKUP dumpfile=li-
     brary.dmp logfile=library_log.log
```

We would like to take this opportunity to remind you: be sure to make backups of your databases. The resulting backups should be checked to see if they are usable for recovery (i.e., can be restored on a test server).

This completes the solution of this problem.

[45] http://blog.sqlauthority.com/2011/05/07/sql-server-2008-2008-r2-create-script-to-copy-database-schema-and-all-the-objects-data-schema-stored-procedure-functions-triggers-tables-views-constraints-and-all-other-database-objects/

[46] https://www.simple-talk.com/sql/database-administration/automated-script-generation-with-powershell-and-smo/

[47] http://docs.oracle.com/cd/E17781_01/server.112/e18804/impexp.htm#ADMQS256

Solution 7.2.1.b[533].

In MySQL, we use the console client that comes with the DBMS to solve this problem.

Deleting and recreating (lines 1-2) is a way to quickly get an empty database: usually, the database to be restored from a backup is already damaged enough to be deleted.

Be careful with the database names in the following batch file. There is a risk of deleting the “wrong” database.

If the database we are about to “recreate” contains useful information, it is recommended that we back it up separately anyway before proceeding with the restore.

MySQL Solution 7.2.1.b (cmd file) (general concept)

```
1    mysql -uUSERNAME -pPASSWORD -e "DROP SCHEMA `DATABASE`;";
2    mysql -uUSERNAME -pPASSWORD -e "CREATE SCHEMA `DATABASE` DEFAULT CHARACTER
     SET utf8 COLLATE utf8_general_ci;";
3    mysql -uUSERNAME -pPASSWORD DATABASE < DATABASE.sql
```

MySQL Solution 7.2.1.b (cmd file) (example)

```
1    mysql -uabc -pdef -e "DROP SCHEMA `library`;";
2    mysql -uabc -pdef -e "CREATE SCHEMA `library` DEFAULT CHARACTER SET utf8
     COLLATE utf8_general_ci;";
3    mysql -uabc -pdef library < library.sql
```

MS SQL Server has two ways to restore a database from backup. The first way is relevant if we have the full SQL-code of the database with all its structures and data.

MS SQL Solution 7.2.1.b (cmd file) (general concept)

```
1    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -i DATABASE.sql
```

MS SQL Solution 7.2.1.b (cmd file) (example)

```
1    sqlcmd -U abc -P def -S COMP\MSSQLSRV -i library.sql
```

The second way is relevant if we want to restore from a binary backup. The following code is a **single line**.

MS SQL Solution 7.2.1.b (cmd file) (general concept)

```
1    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -Q "RESTORE DATABASE [DA-
     TABASE] FROM DISK='FULL_PATH_TO_THE_FILE.bak'"
```

MS SQL Solution 7.2.1.b (cmd file) (example)

```
1    sqlcmd -U abc -P def -S COMP\MSSQLSRV -Q "RESTORE DATABASE [library] FROM
     DISK='C:\backup\library.bak'"
```

In Oracle, a restore can be performed using the `impdp` utility (assuming the backup was performed by the `expdp` utility). The following code is a **single line**.

Oracle Solution 7.2.1.b (cmd file) (general concept)

```
1    impdp USERNAME/PASSWORD@SERVER schemas=USERNAME directory=LOGICAL_DIREC-
     TORY_NAME dumpfile=FILE_NAME.dmp logfile=LOG_FILE_NAME.log
```

Oracle Solution 7.2.1.b (cmd file) (example)

```
1    impdp abc/def@COMP schemas=abc directory=LIBRARY_BACKUP dumpfile=li-
     brary.dmp logfile=library_log.log
```

This completes the solution of this problem.

Solution 7.2.1.c[533].

The simplest, fastest and most general solution of this problem for all three DBMSes is the combination of the solution[533] of problem 7.2.1.a[533] and the solution[535] of problem 7.2.1.b[533] with a small modification: we will restore the database under a new name.

In MySQL, no additional preparations and tricks are required, we just implement the logic just described.

MySQL Solution 7.2.1.b (cmd file) (general concept)

```
1    rem Export
2    mysqldump -uUSERNAME -pPASSWORD SOURCE_DATABASE > SOURCE_DATABASE.sql
3    rem Import
4    mysql -uUSERNAME -pPASSWORD -e "DROP SCHEMA `NEW_DATABASE`;";
5    mysql -uUSERNAME -pPASSWORD -e "CREATE SCHEMA `NEW_DATABASE` DEFAULT CHAR-
     ACTER SET utf8 COLLATE utf8_general_ci;";
6    mysql -uUSERNAME -pPASSWORD NEW_DATABASE < SOURCE_DATABASE.sql
```

MySQL Solution 7.2.1.b (cmd file) (example)

```
1    rem Export
2    mysqldump -uabc -pdef library > library.sql
3    rem Import
4    mysql -uabc -pdef -e "DROP SCHEMA `library_copy`;";
5    mysql -uabc -pdef -e "CREATE SCHEMA `library_copy` DEFAULT CHARACTER SET
     utf8 COLLATE utf8_general_ci;";
6    mysql -uabc -pdef library_copy < library.sql
```

In MS SQL Server, the export is a normal backup, and during the import process, we must use the **`WITH MOVE`** construct to specify new database file names.

Line 5 is needed to identify the logical names present in the backup and to write them into line 6. That is, we can comment out line 6 the first time we run it, since it probably won't work anyway. And when we run it again, we can comment out line 5, since it is no longer needed.

MS SQL Solution 7.2.1.b (cmd file) (general concept)

```
1    rem Export
2    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -Q "BACKUP DATABASE
     [SOURCE_DATABASE] TO DISK='FULL_PATH_TO_THE_FILE.bak'"
3    rem Import
4    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -Q "DROP DATABASE
     [NEW_DATABASE]"
5    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -Q "RESTORE FILELISTONLY
     FROM DISK='C:\!\FULL_PATH_TO_THE_FILE.bak'"
6    sqlcmd -U USERNAME -P PASSWORD -S SERVER\SERVICE -Q "RESTORE DATABASE
     [NEW_DATABASE] FROM DISK='FULL_PATH_TO_THE_FILE.bak' WITH
     MOVE 'LOGICAL_NAME_1' TO 'NEW_FULL_PATH_TO_THE_FILE_1',
     MOVE 'LOGICAL_NAME_2' TO 'NEW_FULL_PATH_TO_THE_FILE_2'"
```

MS SQL Solution 7.2.1.b (cmd file) (example)

```
1    rem Export
2    sqlcmd -U abc -P def -S COMP\MSSQLSRV -Q "BACKUP DATABASE [library] TO
     DISK='C:\backup\library.bak'"
3    rem Import
4    sqlcmd -U abc -P def -S COMP\MSSQLSRV -Q "DROP DATABASE [library_copy]"
5    sqlcmd -U abc -P def -S COMP\MSSQLSRV -Q "RESTORE FILELISTONLY FROM
     DISK='C:\backup\library.bak'"
6    sqlcmd -U abc -P def -S COMP\MSSQLSRV -Q "RESTORE DATABASE [library_copy]
     FROM DISK='C:\backup\library.bak' WITH
     MOVE 'library' TO 'C:\new\library_copy.mdf',
     MOVE 'library_log' TO 'C:\new\library_copy_log.ldf'"
```

In Oracle, the solution is the most voluminous.

Oracle Solution 7.2.1.b (cmd file) (general concept)

```
1   rem Export
2   del FILE_NAME
3   del LOG_FILE_NAME
4   @echo GRANT CREATE ANY DIRECTORY TO USERNAME; | sqlplus USERNAME/PASS-
    WORD@SERVER
5   @echo CREATE DIRECTORY LOGICAL_DIRECTORY_NAME AS 'FULL_PATH_TO_DIRECTORY'; |
    sqlplus USERNAME/PASSWORD@SERVER
6   expdp USERNAME/PASSWORD@SERVER schemas=USERNAME
    directory=LOGICAL_DIRECTORY_NAME dumpfile=FILE_NAME.dmp
    logfile=LOG_FILE_NAME.log
7   rem Import
8   @echo DROP TABLESPACE COPY_TABLESPACE INCLUDING CONTENTS AND DATAFILES; |
    sqlplus USERNAME/PASSWORD@SERVER
9   @echo DROP USER COPY_USERNAME CASCADE; | sqlplus USERNAME/PASSWORD@SERVER
    @echo CREATE TABLESPACE COPY_TABLESPACE DATAFILE 'COPY_DATA_FILE.dat' SIZE 5M
10  REUSE AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED; | sqlplus USERNAME/PASS-
    WORD@SERVER
    @echo CREATE USER COPY_USERNAME IDENTIFIED BY COPY_PASSWORD DEFAULT TABLESPACE
11  COPY_TABLESPACE TEMPORARY TABLESPACE temp; | sqlplus USERNAME/PASSWORD@SERVER
    @echo GRANT ALL PRIVILEGES TO COPY_USERNAME; | sqlplus USERNAME/PASSWORD@SERVER
12  impdp USERNAME/PASSWORD@SERVER schemas=USERNAME
    directory= LOGICAL_DIRECTORY_NAME remap_schema=USERNAME:COPY_USERNAME remap_ta-
13  blespace=TABLESPACE: COPY_TABLESPACE dumpfile=FILE_NAME.dmp
    logfile=LOG_FILE_NAME.log
```

Oracle Solution 7.2.1.b (cmd file) (example)

```
1   rem Export
2   del "C:\backup\library.dmp"
3   del "C:\backup\library_log.log"
4   @echo GRANT CREATE ANY DIRECTORY TO abc; | sqlplus abc/def@COMP
5   @echo CREATE DIRECTORY LIBRARY_BACKUP AS 'C:\backup'; | sqlplus abc/def@COMP
    expdp abc/def@COMP schemas=abc directory=LIBRARY_BACKUP dumpfile=library.dmp
6   logfile=library_log.log
    rem Import
7   @echo DROP TABLESPACE library_copy_ts INCLUDING CONTENTS AND DATAFILES; |
    sqlplus abc/def@COMP
8   @echo DROP USER library_copy CASCADE; | sqlplus abc/def@COMP
9   @echo CREATE TABLESPACE library_copy_ts DATAFILE 'library_copy_ts.dat' SIZE 5M
    REUSE AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED; | sqlplus abc/def@COMP
10  @echo CREATE USER library_copy IDENTIFIED BY library_copy_pwd  DEFAULT TABLE-
    SPACE library_copy_ts TEMPORARY TABLESPACE temp; | sqlplus abc/def@COMP
11  @echo GRANT ALL PRIVILEGES TO library_copy; | sqlplus abc/def@COMP
12  impdp abc/def@COMP schemas=abc directory=LIBRARY_BACKUP remap_schema=abc: li-
13  brary_copy remap_tablespace=library_ts:library_copy_ts dumpfile=library.dmp
    logfile=library_log.log
```

First, we need to delete the existing backups (lines 2-3) and export them (lines 4-6), which is the usual backup discussed in the solution{535} of problem 7.2.1.b{533}.

Then we:

- delete the tablespace of the database copy, in case it already exists (line 8);
- delete the database copy user/schema, in case it already exists (line 9);
- create the tablespace for the database copy (line 10);
- create a database copy (user/schema) (line 11);
- give the database copy user all privileges (line 12): don't do this on real servers, but in a training context it is acceptable;
- import (line 13), specifying the change of schema (**`remap_schema`**) and tablespace (**`remap_tablespace`**).

This completes the solution of this problem.

Task 7.2.1.TSK.A: rewrite command files from solutions 7.2.1.a{533}, 7.2.1.b{535}, and 7.2.1.c{536} to run on Linux.

Task 7.2.1.TSK.B: implement the solution{536} of problem 7.2.1.c{533} without creating intermediate files on disk (in those DBMSes that support such functionality).

7.3. Operations with DBMS

7.3.1. Example 49: User Management, DBMS Start and Stop

Problem 7.3.1.a{539}: create a new DBMS user and grant them the full set of privileges to the “Library” database.

Problem 7.3.1.b{540}: change the password for the user just created in problem 7.3.1.a.

Problem 7.3.1.c{540}: create batch files to start, stop, and restart the DBMS.

Expected result 7.3.1.a.

Since the solution itself is the expected result, see solution 7.3.1.a{539}.

Expected result 7.3.1.b.

Since the solution itself is the expected result, see solution 7.3.1.b{540}.

Expected result 7.3.1.c.

Since the solution itself is the expected result, see solution 7.3.1.c{540}.

Solution 7.3.1.a{539}.

In MySQL, the solution of this problem is achieved by two commands. In line 1, we create a user (the '`%`' clause after `@` means that the user is allowed access from any host), and in line 2, we grant the user full privileges to the “Library” database.

MySQL Solution 7.3.1.a

```
1    CREATE USER 'new_user'@'%' IDENTIFIED BY 'new_password';
2    GRANT ALL PRIVILEGES ON `library`.* TO 'new_user'@'%';
```

In MS SQL Server we must first create a login to connect to the DBMS (lines 2-3), and then (lines 6-8) we can create a user associated with the previously created login and make this user the owner of the “Library” database, which will grant this user full privileges to it.

MS SQL Solution 7.3.1.a

```
1    -- Creating a login (to connect to the server)
2    USE [master];
3    CREATE LOGIN [new_login] WITH PASSWORD = 'new_password';
4
5    -- Creating a user and granting full privileges
6    USE [library];
7    CREATE USER [new_user] FOR LOGIN [new_login];
8    EXEC sp_addrolemember N'db_owner', N'new_user';
```

In Oracle, there is no easy way to give one user all permissions to another user's entire schema. There are workarounds[48], but they are too cumbersome.

However, we can take advantage of an interesting feature of this DBMS (also supported by MySQL), which is a proxy access, when the authorization occurs on behalf of one user, and the work continues on behalf of another user.

On lines 1-2 we create a new user, line 4 allows them to work as the owner of the "Library" database, and line 5 allows them to establish connections to the database.

Oracle Solution 7.3.1.a

```
1    CREATE USER new_user IDENTIFIED BY new_password
2    DEFAULT TABLESPACE library_ts TEMPORARY TABLESPACE temp;
3
4    ALTER USER library GRANT CONNECT THROUGH new_user;
5    GRANT CREATE SESSION TO new_user;
```

Now we can check if this solution works by connecting to the database via **sqlplus** with the following credentials: **new_user[library]/new_password@SERVER**. After starting **sqlplus**, we can check if we are connected as the **library** user by executing the **SHOW USER** command.

This completes the solution of this problem.

Solution 7.3.1.b{539}.

In all three DBMSes this problem is solved in one line (in MySQL we must use the **PASSWORD** function to hash the initial password value).

MySQL Solution 7.3.1.b

```
1    SET PASSWORD FOR 'new_user'@'%' = PASSWORD('some_password')
```

MS SQL Solution 7.3.1.b

```
1    ALTER LOGIN [new_login] WITH PASSWORD = 'some_password_3@'
```

Oracle Solution 7.3.1.b

```
1    ALTER USER new_user IDENTIFIED BY some_password
```

This completes the solution of this problem.

Solution 7.3.1.c{539}.

The solution of this problem for all three DBMSes is based on the use of **net start**[49] and **net stop**[50] commands. The only difference will be in the service names, and for MS SQL Server (probably) we will also have to manage its related services, the set and names of which depend on the specific DBMS version.

The code in the batch files below will restart the database. Obviously, we can use the parts responsible for stopping and starting the DBMS separately to achieve the appropriate result.

[48] https://community.oracle.com/thread/2386990?start=0&tstart=0
[49] https://technet.microsoft.com/en-us/library/bb490713.aspx
[50] https://technet.microsoft.com/en-us/library/bb490715.aspx

MySQL Solution 7.3.1.c

```
rem Stop
net stop MySQL56

rem Start
net start MySQL56
```

MS SQL Solution 7.3.1.c

```
rem Stop
net stop "SQL Server (SQLEXPRESS)"

rem Start
net start "SQL Server (SQLEXPRESS)"
```

Oracle Solution 7.3.1.c

```
rem Stop
net stop OracleServiceXE

rem Start
net start OracleServiceXE
```

This completes the solution of this problem.

Task 7.3.1.TSK.A: remove access permissions to the “Library” database from the user created in the solution 7.3.1.b{539}.

Task 7.3.1.TSK.B: delete the user created in the solution{539} of problem 7.3.1.a{539}.

Task 7.3.1.TSK.C: create a batch file showing whether each of the three DBMSes is currently running.

Task 7.3.1.TSK.D: rewrite the solution{540} of problem 7.3.1.c{539} to run on Linux.

7.3.2. Example 50: Charset Management

Problem 7.3.2.a{542}: display all information about the current DBMS settings regarding the default encodings (charsets).

Problem 7.3.2.b{543}: change all DBMS default encoding (charset) settings to UTF8.

Expected result 7.3.2.a.

As a result of query(s) execution the current DBMS settings are displayed with respect to the encodings (charsets) used by default.

Expected result 7.3.2.b.

As a result of the query(s) execution the default DBMS encodings (charsets) are changed to UTF8.

Solution 7.3.2.a{542}:

In MySQL we can get information about encodings (charsets) by the following query (note: the symbol `_` is escaped with `\`, because `_` stands for “any character”, in this case it is not critical, but we should still write it correctly).

MySQL Solution 7.3.2.a

```
SHOW GLOBAL VARIABLES
WHERE `variable_name` LIKE 'character\_set%'
   OR `variable_name` LIKE 'collat%'
```

The result of such a query will be a table with the following data.

Variable_name	Value
character_set_client	utf8
character_set_connection	utf8
character_set_database	utf8
character_set_filesystem	Binary
character_set_results	utf8
character_set_server	utf8
character_set_system	utf8
character_sets_dir	C:\Program Files\MySQL\MySQL Server 8.0\share\charsets\
collation_connection	utf8_general_ci
collation_database	utf8_general_ci
collation_server	utf8_general_ci

In MS SQL Server we can get information on encodings (charsets) with the following query.

MS SQL Solution 7.3.2.a

```
SELECT [collation_name]
FROM [sys].[databases]
WHERE [name] = 'master'
```

The result of such a query will be a table with the following data.

collation_name
Cyrillic_General_CI_AS

MS SQL Server will apply this encoding (charset, collation) by default to all new databases and their structures, unless a different encoding is specified in the corresponding queries.

In Oracle we can get information about encodings (charsets) by the following query.

Oracle | Solution 7.3.2.a

```
1    -- Simplified option:
2    SELECT value$
3    FROM   sys.props$
4    WHERE  name = 'NLS_CHARACTERSET';
5
6    -- Extended option:
7    SELECT parameter,
8           value
9    FROM   nls_database_parameters
10   WHERE  parameter = 'NLS_CHARACTERSET'
11       OR parameter = 'NLS_NCHAR_CHARACTERSET';
```

The first query will return the following information.

VALUE$
AL32UTF8

The second query will return the following information.

PARAMETER	VALUE
NLS_CHARACTERSET	AL32UTF8
NLS_NCHAR_CHARACTERSET	AL16UTF16

The **NLS_CHARACTERSET** parameter value is responsible for encoding “normal” data (without **N** prefix, i.e., **CHAR**, **VARCHAR2**, etc.), and the **NLS_NCHAR_CHARACTERSET** value is responsible for encoding “national data” (i.e., **NCHAR**, **NVARCHAR2**, etc.)

This completes the solution of this problem.

Solution 7.3.2.b{542}.

Changing encodings (charsets, collations) on the whole server may lead to temporary or permanent data corruption in the existing databases. Be sure to make a full backup before performing the corresponding experiments.

In MySQL we can change the default DBMS encoding settings either in the **my.ini** configuration file (permanent changes) or by the following queries (changes are valid until the DBMS is restarted):

MySQL | Solution 7.3.1.b

```
1    -- Repeat for all necessary variables:
2    SET character_set_database = 'utf8';
```

In MS SQL Server we can change the default DBMS encoding settings only in a rather complicated way:

- stop the DBMS;
- run the following command from the console (its parameters may differ depending on the DBMS version):

MySQL | Solution 7.3.1.b

```
1    sqlservr -m -T4022 -T3659 -s"SERVICE" -q"ENCODING"
```

- run the DBMS.

As for the encoding itself, UTF8 is not supported in “pure form” in MS SQL Server, but it is possible to store and process corresponding data using other encodings[51].

Finally, in Oracle changing the default encodings is only possible with a rather non-trivial procedure[52] (which description is clearly beyond the scope of this book), and therefore it is recommended to specify all necessary parameters by default at the DBMS installation stage, as well as to specify encodings when creating individual databases and tables.

Task 7.3.2.TSK.A: conduct an experiment in all three DBMSes with changing the encoding of an existing database containing strings with Cyrillic characters; check how the display, search and ordering of such data will change.

Task 7.3.2.TSK.B: conduct an experiment in all three DBMSes with changing the connection encoding to a different one from the encoding of the database you are going to work with; check how the display, search and ordering of strings containing Cyrillic characters will change.

[51] https://msdn.microsoft.com/en-us/library/ms143726%28v=sql.110%29.aspx

[52] https://docs.oracle.com/cd/E11882_01/server.112/e10729/ch11charsetmig.htm#NLSPG011

7.4. Useful Operations with Data

7.4.1. Example 51: Date Operations

Problem 7.4.1.a{545}: create a query that shows for each subscription start date the number of the day of the week (1 — Monday, 2 — Tuesday, etc.) and the sequence number of the corresponding day of the week within the month (e.g., first Thursday of the month, second Wednesday of the month, etc.)

Problem 7.4.1.b{549}: rework the solution of problem 7.4.1.a as a function that receives a date and returns the result as a single line (instead of two separate columns).

Expected result 7.4.1.a.

For each subscription start date, the information required by the problem is presented in the columns “W” (the sequence number of the corresponding day of the week within the month) and “D” (the number of the day of the week):

sb_id	sb_start	W	D
2	2011-01-12	2	3
3	2012-05-17	3	4
42	2012-06-11	2	1
57	2012-06-11	2	1
61	2014-08-03	1	7
62	2014-08-03	1	7
86	2014-08-03	1	7
91	2015-10-07	1	3
95	2015-10-07	1	3
99	2015-10-08	2	4
100	2011-01-12	2	3

Expected result 7.4.1.b.

Since the solution itself is the expected result, see solution 7.4.1.b.

Solution 7.4.1.a{545}. (Also see the comments in the solution 7.4.1.b{549}.)

First, note that this problem is an extended version of the classic “Determine if a date is the third Tuesday of a month” type, here we just get the result for each date, rather than checking them for a single given date.

It should also be explained how the calculation is made. Consider the October 2016 calendar. Note the following:

- the second week of the month begins on the 3rd of October (Monday) (unfortunately, intuitively, many people think that the second week always begins on the 8th day of any month, and this is not true);
- even though Monday the 3rd of October is the second week, it is the first Monday of the month (while Saturday the 8th is the second Saturday of the month, although it is in the same second week).

1	2	3	4	5	6	7	← Day number
MON	**TUE**	**WED**	**THU**	**FRI**	**SAT**	**SUN**	
					1	2	1st week
3	4	5	6	7	8	9	2nd week
10	11	12	13	14	15	16	3rd week
17	18	19	20	21	22	23	4th week
24	25	26	27	28	29	30	5th week
31							6th week

Considering the notes just discussed, we will create the code. For MySQL it looks like this:

MySQL | Solution 7.4.1.a

```
1    SELECT `sb_id`,
2           `sb_start`,
3           CASE
4            WHEN WEEKDAY(`sb_start` - INTERVAL DAY(`sb_start`)-1 DAY) <=
5                 WEEKDAY(`sb_start`)
6            THEN WEEK(`sb_start`, 5) -
7                 WEEK(`sb_start` - INTERVAL DAY(`sb_start`)-1 DAY, 5) + 1
8            ELSE WEEK(`sb_start`, 5) -
9                 WEEK(`sb_start` - INTERVAL DAY(`sb_start`)-1 DAY, 5)
10          END AS `W`,
11          WEEKDAY(`sb_start`) + 1 AS `D`
12   FROM `subscriptions`
```

Let's start with lines 8-9, which determine the number of the week of the month. To get this result we perform the following operations (let's consider the date 2016.10.20 as an example).

Code	Idea	Result
`WEEK(`sb_start`, 5)`	Getting for the analyzed date the number of the week of the year in which it falls.	42
`WEEK(`sb_start` - INTERVAL DAY(`sb_start`)-1 DAY, 5)`	Getting the number of the week of the year for the first day of the month on which the analyzed date falls. To obtain the value of the first day of the month (2016.10.01), we subtract a value that is 1 less than the "day number of the month" (20 - 19 = 1) of the analyzed date from the analyzed date (2016.10.20).	39
`WEEK(`sb_start`, 5) - WEEK(`sb_start` - INTERVAL DAY(`sb_start`)-1 DAY, 5)`	The whole operation. Yes, value 3 is not correct in terms of "week of the month", but it is correct in the context of "2016.10.20 is the 3rd Thursday of the month".	3

Parameter "5" of the `WEEK` function defines the form of the result return (the number of the week in the year is 0-53, the beginning of the week is Monday, see documentation for details).

It remains to consider lines 3-10, in which the `CASE` expression with two alternatives is applied. The condition in lines 4-5 compares the numbers of the day of the week for the analyzed date and the first day of the month. If the day number of the first day of

the month is ≤ the day number of the analyzed date, the “week number” must be increased by one. So, for 2016.10.20 the days from Monday to Friday will be the third Monday of the month, the third Tuesday of the month, etc., and Saturday and Sunday will be, respectively, the fourth Saturday and the fourth Sunday of the month.

The **`WEEKDAY`** function in line 11 of the query returns the sequence number of the day of the week, where the numbering is as follows: MON = 0, TUE = 1, WED= 2, etc. To get a more familiar result, where MON = 1, TUE = 2, WED= 3, etc., we add one to the result of the function.

This completes the MySQL solution, moving on to MS SQL Server.

Here we have to compensate for the “week starts on Sunday” effect, which will be present by default in most cases. In theory, in MS SQL Server we can run the **`SET DATEFIRST 1`** command, which will make Monday the beginning of the week, but in practice, many sources noted that this may not work. Also, in MS SQL Server ISO number of the week is calculated differently than in MySQL (the first day of the year may not be the zero week, but 52nd or 53rd)

The following code is created for the case when MS SQL Server considers Sunday to be the first day of the week.

MS SQL Solution 7.4.1.a

```
1    WITH [iso_week_data] AS
2     (
3      SELECT CASE
4               WHEN DATEPART(iso_week,
5                             CONVERT(VARCHAR(6), [sb_start], 112) + '01') >
6                    DATEPART(dy,
7                             CONVERT(VARCHAR(6), [sb_start], 112) + '01')
8               THEN 0
9               ELSE DATEPART(iso_week,
10                            CONVERT(VARCHAR(6), [sb_start], 112) + '01')
11            END AS [real_iso_week_of_month_start],
12            CASE
13             WHEN DATEPART(iso_week, [sb_start]) > DATEPART(dy, [sb_start])
14             THEN 0
15             ELSE DATEPART(iso_week, [sb_start])
16            END AS [real_iso_week_of_this_date],
17            [sb_start],
18            [sb_id]
19     FROM [subscriptions]
20    )
21    SELECT [sb_id],
22           [sb_start],
23           CASE
24            WHEN DATEPART(dw,
25                          DATEADD(day, -1,
26                                  CONVERT(VARCHAR(6), [sb_start], 112)
27                                  + '01')) <=
28                 DATEPART(dw, DATEADD(day, -1, [sb_start]))
29            THEN [real_iso_week_of_this_date] -
30                 [real_iso_week_of_month_start] + 1
31            ELSE [real_iso_week_of_this_date] - [real_iso_week_of_month_start]
32           END AS [W],
33           DATEPART(dw, DATEADD(day, -1, [sb_start])) AS [D]
34    FROM [iso_week_data]
```

Let’s start with lines 29-31, which contain the same idea as in lines 6-9 of MySQL solution: from the number within the year of the week on which the analyzed date falls, we subtract the number within the year of the week on which the first day of the month to which the analyzed date falls.

Lines 24-28 follow the same logic as lines 4-5 of the MySQL solution: the numbers of the days of the week for the analyzed date and the first day of the month are compared

(if the day number of the first day of the month is ≤ the day number of the analyzed date, "week number" must be increased by one).

The `CONVERT(VARCHAR(6), [sb_start], 112)` operation allows us to get the value of year and month as six characters, e.g.: 2016.10.20 give 201610. After performing `+ '01'` we will get 20161001, i.e., the first day of October. The number `112` is another "magic value" (see documentation for details).

The `DATEADD(day, -1, [sb_start])` operation, which occurs many times throughout the query, is needed to mark Monday as the beginning of the week, because MS SQL Server now considers Sunday the beginning of the week, and the weekday number turns out to be "incorrect" (SUN = 1, SAT = 7, etc.).

It remains to consider the Common Table Expression in lines 1-20: this code is needed to ensure that for the first days of the year MS SQL Server does not return the number of the week as 52 or 53. To do this, the number of the week is compared to the number of the day of the year. If the week number is greater, it must be converted to 0.

This completes the solution for MS SQL Server, moving on to Oracle.

Here, too, we have to compensate for the "start of the week from Sunday" effect, which will be present by default in most cases. We can use `ALTER SESSION SET NLS_TERRITORY=Russia` command in Oracle to set the beginning of day to Monday, but we'll again assume the worst (e.g., we cannot switch this parameter) and create the solution for the situation when first day of week is Sunday.

Oracle | Solution 7.4.1.a

```
1    WITH "iso_week_data" AS
2     (
3      SELECT CASE
4               WHEN TO_NUMBER(TO_CHAR(TRUNC("sb_start", 'mm'), 'IW')) >
5                    TO_NUMBER(TO_CHAR(TRUNC("sb_start", 'mm'), 'DDD'))
6               THEN 0
7               ELSE TO_NUMBER(TO_CHAR(TRUNC("sb_start", 'mm'), 'IW'))
8             END AS "real_iso_week_of_month_start",
9             CASE
10             WHEN TO_NUMBER(TO_CHAR("sb_start", 'IW')) >
11                  TO_NUMBER(TO_CHAR("sb_start", 'DDD'))
12             THEN 0
13             ELSE TO_NUMBER(TO_CHAR("sb_start", 'IW'))
14            END AS "real_iso_week_of_this_date",
15            "sb_start",
16            "sb_id"
17     FROM "subscriptions"
18    )
19    SELECT "sb_id", "sb_start",
20           CASE
21            WHEN (TRUNC("sb_start", 'mm') -
22                  NEXT_DAY(TRUNC("sb_start", 'mm') - 7, 'MON'))
23              <= ("sb_start" - NEXT_DAY("sb_start" - 7, 'MON'))
24             THEN "real_iso_week_of_this_date" -
25                  "real_iso_week_of_month_start" + 1
26             ELSE "real_iso_week_of_this_date" -
27                  "real_iso_week_of_month_start"
28            END AS "W",
29          "sb_start" - NEXT_DAY("sb_start" - 7, 'MON') + 1 AS "D"
30    FROM "iso_week_data"
```

In Oracle (as well as in MS SQL Server) ISO week number is calculated differently than in MySQL (the first day of the year may not be zero week, but 52nd or 53rd), this must also be compensated by additional calculations similar to the solution for MS SQL Server (CTE in rows 1-18 is similar to CTE in rows 1-20 solutions for MS SQL Server except for data type conversion method and how to get the first day of the month: in Oracle we can use the `TRUNC` function for this).

The main part of the query on lines 20-30 is also similar to the solution for MS SQL Server except for the way to get the correct day of the week number. Let's take a closer look at line 29.

The `NEXT_DAY` function returns the date of the next day of the week after the specified date (e.g., `NEXT_DAY(2016.10.20, Saturday)` will return the date of the next Saturday, i.e., 2016.10.22).

To get the number of the day within the week, we need to know the difference in days between the analyzed date and the previous (preceding) Monday, whose date we get as `NEXT_DAY("sb_start" - 7, 'MON')` (this expression can be read as "return the date of the nearest Monday in the past from the analyzed date").

So, the number of the day within the week is (almost) ready: `"sb_start" - NEXT_DAY("sb_start" - 7, 'MON')`. It remains to add 1, so that the numbering does not start from zero. This is how we get the final value of the day number within the week.

This completes the solution of this problem.

Solution 7.4.1.b{545}.

The solution of this problem is built on the logic of date processing presented in the solution 7.4.1.a{545}, and the function code itself is specially written not optimally, but in the most detailed way and commented (all explanatory results are obtained for the date 2016.10.20).

The function code for MySQL is as follows:

The function code for MySQL looks like this

MySQL Solution 7.4.1.b

```
DELIMITER $$
CREATE FUNCTION GET_WEEK_AND_DAY (date_var DATE)
RETURNS CHAR(4)
DETERMINISTIC
BEGIN
 DECLARE day_number TINYINT;    -- weekday number (4)
 DECLARE week_number TINYINT;   -- number of the full week of the month (3)
 DECLARE current_wn TINYINT;    -- number within the year of the week,
                                -- which includes the
                                -- analyzed date (42)
 DECLARE first_dom_wn TINYINT; -- the number within the year of the week to
                                -- which the first day of the month with
                                -- the analyzed date belongs (39)
 DECLARE current_dom TINYINT;  -- day number within the month (20)
 SET day_number = WEEKDAY(date_var) + 1; -- numbering goes from 0,
                                         -- therefore we need +1 (3+1 = 4)
 SET current_dom = DAY(date_var); -- (20)
 SET current_wn = WEEK(date_var, 5); -- see the documentation for "5" (42)
 SET first_dom_wn = WEEK(date_var - INTERVAL current_dom-1 DAY, 5); -- (39)
 SET week_number = current_wn - first_dom_wn; -- (3)

 IF WEEKDAY(date_var - INTERVAL current_dom-1 DAY) <= WEEKDAY(date_var)
  THEN
   SET week_number = week_number + 1;
  END IF;
 RETURN CONCAT('W', week_number, 'D', day_number); -- W3D4
END;
$$
DELIMITER ;
```

We can check the functionality of this function by running the following code:

MySQL Solution 7.4.1.b (functionality check)

```
SELECT `sb_id`,
       `sb_start`,
       GET_WEEK_AND_DAY(`sb_start`) AS `DW`
FROM   `subscriptions`
```

The function code for MS SQL Server looks like this:

MS SQL | Solution 7.4.1.b

```
CREATE FUNCTION GET_WEEK_AND_DAY(@date_var DATETIME)
RETURNS VARCHAR(4)
AS
BEGIN
 DECLARE @day_number TINYINT;     -- weekday number (4)
 DECLARE @week_number TINYINT;    -- number of the full week of the month (3)
 DECLARE @current_wn TINYINT;     -- number within the year of the week,
                                  -- which includes the
                                  -- analyzed date (42)
 DECLARE @first_dom_wn TINYINT;   -- the number within the year of
                                  -- the week to which the first
                                  -- day of the month with
                                  -- the analyzed date belongs (39)
 DECLARE @current_doy INT;        -- day number within the year (294)
 DECLARE @first_dom_doy INT;      -- number within the year of the first day
                                  -- the month to which the
                                  -- analyzed date belongs (275)
 DECLARE @real_iso_week_of_month_start TINYINT;
 -- ISO number within the year of the week to which the first day of
 -- the month with the analyzed date refers (39), for the first days of
 -- January it will be 0, not 52 or 53, as calculated by MS SQL Server
 DECLARE @real_iso_week_of_this_date TINYINT;
 -- ISO number within the year of the week to which the analyzed
 -- date belongs (42), for the first days of January it will be 0,
 -- not 52 or 53, as calculated by MS SQL Server

 SET @day_number = DATEPART(dw, DATEADD(day, -1, @date_var));
  -- @day_number = DATEPART(dw, (2016.10.20 - 1)) =
  --               DATEPART(dw, 2016.10.19) = 4

 SET @current_wn = DATEPART(iso_week, @date_var); -- 42
 SET @first_dom_wn = DATEPART(iso_week,
                              CONVERT(VARCHAR(6), @date_var, 112) + '01');
  -- @first_dom_wn = DATEPART(iso_week, '201610' + '01') =
  --                 DATEPART(iso_week, '20161001') = 39

 SET @current_doy = DATEPART(dy, @date_var); -- 294
 SET @first_dom_doy = DATEPART(dy,
                               CONVERT(VARCHAR(6), @date_var, 112) + '01');
  -- @first_dom_doy = DATEPART(dy, '201610' + '01') =
                      DATEPART(dy, '20161001') = 275

 -- Relevant, e.g., for 2017.01.01
 IF (@first_dom_wn > @first_dom_doy)
       SET @real_iso_week_of_month_start = 0
  ELSE SET @real_iso_week_of_month_start = @first_dom_wn;

 -- Relevant, e.g., for 2017.01.02
 IF (@current_wn > @current_doy)
       SET @real_iso_week_of_this_date = 0
  ELSE SET @real_iso_week_of_this_date = @current_wn;
SET @week_number = @real_iso_week_of_this_date -
                   @real_iso_week_of_month_start; -- (3)

 IF (DATEPART(dw, DATEADD(day, -1,
                  CONVERT(VARCHAR(6), @date_var, 112) + '01'))
     <= @day_number)
  SET @week_number = @week_number + 1;

 RETURN CONCAT('W', @week_number, 'D', @day_number); -- W3D4
END;
```

We can check the functionality of this function by running the following code:

MS SQL Solution 7.4.1.b (functionality check)

```
SELECT [sb_id],
       [sb_start],
       dbo.GET_WEEK_AND_DAY([sb_start]) AS [DW]
FROM   [subscriptions];
```

The function code for Oracle looks like this:

Oracle Solution 7.4.1.b

```
CREATE OR REPLACE FUNCTION GET_WEEK_AND_DAYISO(date_var IN DATE)
RETURN VARCHAR
IS
 day_number NUMBER(1);     -- weekday number (4)
 week_number NUMBER(1);    -- number of the full week of the month (3)
 current_wn NUMBER(3);     -- number within the year of the week,
                           -- which includes the
                           -- analyzed date (42)
 first_dom_wn NUMBER(3);   -- the number within the year of the week
                           -- to which the first day of the month with
                           -- the analyzed date belongs (39)
 current_doy NUMBER(5);    -- day number within the year (294)
 first_dom_doy NUMBER(5); -- number within the year of the first day the
                           -- month to which the analyzed date belongs (275)

 real_iso_week_of_month_start NUMBER(3);
 -- ISO number within the year of the week to which the first day of
 -- the month with the analyzed date refers (39), for the first days of
 -- January it will be 0, not 52 or 53, as calculated by Oracle

 real_iso_week_of_this_date NUMBER(3);
 -- ISO number within the year of the week to which the analyzed
 -- date belongs (42), for the first days of January it will be 0,
 -- not 52 or 53, as calculated by Oracle
BEGIN
 day_number := date_var - NEXT_DAY(date_var - 8, 'MON') + 1;
 -- day_number := 2016.10.20 - NEXT_DAY(2016.10.20 - 8, 'MON') + 1 =
 --               2016.10.20 - NEXT_DAY(2016.10.12, 'MON') + 1
 --               2016.10.20 - 2016.10.17 + 1 = 4

 current_wn := TO_NUMBER(TO_CHAR(date_var, 'IW')); -- 42
 first_dom_wn := TO_NUMBER(TO_CHAR(TRUNC(date_var, 'mm'), 'IW')); -- 39
 -- first_dom_wn := TO_NUMBER(TO_CHAR(TRUNC(2016.10.20, 'mm'), 'IW')) =
 --                 TO_NUMBER(TO_CHAR(2016.10.01, 'IW')) = 39

 current_doy := TO_NUMBER(TO_CHAR(date_var, 'DDD')); -- 294
 first_dom_doy := TO_NUMBER(TO_CHAR(TRUNC(date_var, 'mm'), 'DDD'));
 -- first_dom_doy := TO_NUMBER(TO_CHAR(TRUNC(2016.10.20, 'mm'), 'DDD')) =
 --                  TO_NUMBER(TO_CHAR(2016.10.01, 'DDD')) = 275

-- Relevant, e.g., for 2017.01.01
 IF (first_dom_wn > first_dom_doy)
  THEN real_iso_week_of_month_start := 0;
  ELSE real_iso_week_of_month_start := first_dom_wn;
 END IF;
```

Oracle Solution 7.4.1.b (continued)

```
46    -- Relevant, e.g., for 2017.01.02
47    IF (current_wn > current_doy)
48     THEN real_iso_week_of_this_date := 0;
49     ELSE real_iso_week_of_this_date := current_wn;
50    END IF;
51
52    week_number := real_iso_week_of_this_date - real_iso_week_of_month_start;
53    -- (3)
54
55    IF (TRUNC(date_var, 'mm') - NEXT_DAY(TRUNC(date_var, 'mm') - 7, 'MON')
56              <= day_number)
57     THEN week_number := week_number + 1;
58    END IF;
59
60    RETURN 'W' || week_number || 'D' || day_number;
61  END;
```

We can check the functionality of this function by running the following code:

Oracle Solution 7.4.1.b (functionality check)

```
1   SELECT "sb_id",
2          "sb_start",
3          GET_WEEK_AND_DAY("sb_start") AS "DW"
4   FROM   "subscriptions";
```

This completes the solution of this problem.

Task 7.4.1.TSK.A: rewrite the solution{545} of problem 7.4.1.a{545} so that the query returns the number of the week and the day of the week as one string (e.g., W5D1), similar to the result returned by the function obtained in the solution{549} of problem 7.4.1.b{545}.

Task 7.4.1.TSK.B: rewrite the solution{545} of problem 7.4.1.a{545} for MS SQL Server and Oracle without using of Common Table Expression.

7.4.2. Example 52: Acquiring Non-repeatable Unique Values

Problem 7.4.2.a{554}: create a query that generates a pair of guaranteed non-repeating readers' identifiers.

Problem 7.4.2.b{556}: rework the solution of problem 7.4.2.a{554} using Common Table Expressions.

Expected result 7.4.2.a.

After executing the query, a pair of readers' identifiers should be selected, whose values are guaranteed not to be equal (duplicated) (provided that there are at least two records in the `subscribers` table), e.g.:

id_first	id_second
3	1

Expected result 7.4.2.b.

Since the solution itself is the expected result, see the solution 7.4.2.b{556}.

Solution 7.4.2.a{554}.

Suppose that we have no information about readers' identifiers (minimum value, maximum value, total number, sequence continuity, etc.) except for the fact that there are at least two entries in the `subscribers` table (otherwise, the problem has no solution).

The logic of the solution is based on the fact that the Cartesian product of two sets, one of which has at least two unique values, is guaranteed to contain a pair of unique values.

Let's explain this idea with an example. Suppose we randomly choose two sets of readers' identifiers, and we have "bad luck" and both sets have the same value.

Set 1	Set 2
5	5
3	

The Cartesian product of such sets will give the following result:

Values from set 1	Values from set 2
5	5
3	5

It is easy to see that the second combination contains non-repeating values (which is required by the problem). Obviously, if all three identifier values in the initial sets are different, then the resulting Cartesian product will also contain at least one (in fact, two) combinations with non-repeating values.

This logic can be broken if and only if in the first set both values of identifiers coincide, but we cannot fear this scenario, since we will build this set based on the values of the primary key, which are unique by definition.

So, the solution of this problem for MySQL before version 8 is as follows.

MySQL Solution 7.4.2.a

```
1    SELECT `id_first`,
2           `id_second`
3    FROM   (SELECT `s_id` AS `id_first`
4            FROM   `subscribers`
5            ORDER  BY RAND()
6            LIMIT  2) AS `set1`
7           CROSS JOIN
8           (SELECT `s_id` AS `id_second`
9            FROM   `subscribers`
10           ORDER  BY RAND()
11           LIMIT  1) AS `set2`
12   WHERE  `id_first` != `id_second`
13   LIMIT  1
```

In lines 3-6 we get a pair of values for the first set, in lines 8-11 we get one value for the second set, and the **`CROSS JOIN`** operator in line 7 ensures that we get the Cartesian product of the sets. The condition in line 12 ensures that the problem condition (the values of the identifiers must not coincide) is satisfied.

Solution for MS SQL Server differs only in the way to specify the number of returned rows (**`TOP`** instead of **`LIMIT`**) and random ordering (because in MS SQL Server **`RAND`** function will return the same number for all calls within one query).

MS SQL Solution 7.4.2.a

```
1    SELECT TOP 1 [id_first],
2                 [id_second]
3    FROM   (SELECT TOP 2 [s_id] AS [id_first]
4            FROM   [subscribers]
5            ORDER  BY NEWID()) AS [set1]
6           CROSS JOIN
7           (SELECT TOP 1 [s_id] AS [id_second]
8            FROM   [subscribers]
9            ORDER  BY NEWID()) AS [set2]
10   WHERE  [id_first] != [id_second]
```

Solution for Oracle also differs only in the way of specifying the number of rows to be returned (subquery with **`ROW_NUMBER`**) and the way of ordering by random number (**`DBMS_RANDOM.RANDOM`**).

The first option of the solution is so “syntactically overloaded” because Oracle starts to support the **`OFFSET ... FETCH`** syntax only since version 12.

Oracle | Solution 7.4.2.a (before version 12)

```
SELECT "id_first",
       "id_second"
FROM   (SELECT "id_first",
               "id_second",
               ROW_NUMBER() OVER (ORDER BY NULL) AS "rn"
        FROM   (SELECT "id_first"
                FROM   (SELECT "s_id" AS "id_first",
                               ROW_NUMBER()
                               OVER (ORDER BY DBMS_RANDOM.RANDOM) AS "rn"
                        FROM   "subscribers") "pre_set1"
                WHERE  "rn" <= 2) "set1"
               CROSS JOIN
               (SELECT "id_second"
                FROM   (SELECT "s_id" AS "id_second",
                               ROW_NUMBER()
                               OVER (ORDER BY DBMS_RANDOM.RANDOM) AS "rn"
                        FROM   "subscribers") "pre_set2"
                WHERE  "rn" = 1) "set2"
        WHERE  "id_first" != "id_second") "prepared_data"
WHERE  "rn" = 1
```

Starting from version 12, it is possible to implement a more compact solution in this DBMS.

Oracle | Solution 7.4.2.a (version 12 and newer)

```
SELECT "id_first",
       "id_second"
FROM   (SELECT "id_first",
               "id_second"
        FROM   (SELECT "s_id" AS "id_first"
                FROM "subscribers"
                ORDER BY DBMS_RANDOM.RANDOM
                FETCH FIRST 2 ROWS ONLY) "set1"
               CROSS JOIN
               (SELECT "s_id" AS "id_second"
                FROM "subscribers"
                ORDER BY DBMS_RANDOM.RANDOM
                FETCH FIRST 1 ROWS ONLY) "set2"
        WHERE  "id_first" != "id_second") "prepared_data"
FETCH FIRST 1 ROWS ONLY
```

This completes the solution of this problem.

Solution 7.4.2.b{554}.

Attention: the solution of this problem is given as an example of what must **not** be done. Below it will be explained why exactly this must **not** be done.

MySQL supports Common Table Expressions only from version 8, so there is no solution of this problem for earlier versions of the DBMS.

It should be noted that MySQL is the only DBMS considered in this book in which the following solution is correct and works without any problems.

The solution looks like this.

MySQL Solution 7.4.2.b (version 8 and newer)

```
1    WITH `first_source`
2         AS (SELECT `s_id`
3             FROM   `subscribers`
4             ORDER  BY RAND()
5             LIMIT 1),
6         `second_source`
7         AS (SELECT `s_id`
8             FROM   `subscribers`
9             WHERE  `s_id` != (SELECT `s_id`
10                              FROM   `first_source`)
11            ORDER  BY RAND()
12            LIMIT 1)
13   SELECT `first_source`.`s_id`  AS `id_first`,
14          `second_source`.`s_id` AS `id_second`
15   FROM       `first_source`
16   CROSS JOIN `second_source`
```

In the first Common Table Expression (lines 1-5), we extract one random identifier. In the second Common Table Expression (lines 6-12), we also extract one random identifier, and use the condition in lines 9-10 to ensure that it does not coincide with the identifier previously extracted in the first Common Table Expression.

Next, all we have to do is take the previously obtained identifiers from two separate "tables" (as Common Table Expression actually returns a "table") get the result as one line of two columns, which is accomplished by the code in lines 13-16.

In MySQL this solution works because the results of Common Table Expressions in this DBMS are persistent, i.e., they are not recalculated each time the Common Table Expression is called, and therefore the data set in the `WHERE` section subquery is the same as the data set returned elsewhere in the query.

This behavior (albeit extremely convenient) is uncharacteristic of many other DBMSes, as we will see below. It is also important to understand that this behavior of MySQL may be changed in the future, and then this solution will not work correctly.

Solutions for MS SQL Server and Oracle are the same, the only difference is in the way of ordering and getting the first record, but these features have already been discussed in the solution{554} of problem 7.4.2.a{554}).

First, let's look at the code (reminder: this is incorrect code) for both DBMSes.

MS SQL Solution 7.4.2.b (incorrect solution)

```
1    WITH [first_source]
2         AS (SELECT TOP 1 [s_id]
3             FROM   [subscribers]
4             ORDER  BY NEWID()),
5         [second_source]
6         AS (SELECT TOP 1 [s_id]
7             FROM   [subscribers]
8             WHERE  [s_id] != (SELECT [s_id]
9                               FROM   [first_source])
10            ORDER  BY NEWID())
11   SELECT [first_source].[s_id]  AS [id_first],
12          [second_source].[s_id] AS [id_second]
13   FROM       [first_source]
14   CROSS JOIN [second_source]
```

Oracle Solution 7.4.2.b (incorrect solution)

```
1    WITH "first_source"
2         AS (SELECT "s_id",
3                    ROW_NUMBER() OVER (ORDER BY DBMS_RANDOM.RANDOM) AS "rn"
4               FROM   "subscribers"),
5         "second_source"
6         AS (SELECT "s_id",
7                    ROW_NUMBER() OVER (ORDER BY DBMS_RANDOM.RANDOM) AS "rn"
8               FROM   "subscribers"
9               WHERE  "s_id" != (SELECT "s_id"
10                                FROM   "first_source"
11                                WHERE  "rn" = 1))
12   SELECT "first_source"."s_id"  AS "id_first",
13          "second_source"."s_id" AS "id_second"
14   FROM       "first_source"
15   CROSS JOIN "second_source"
16   WHERE "first_source"."rn" = 1
17     AND "second_source"."rn" = 1
```

At first glance, the code should work (the queries run without errors, and therein lies the biggest problem: it seems to work, when in fact it does not).

In the first Common Table Expression (lines 1-4 of both queries) we get one random record (random identifier). In the second Common Table Expression (lines 5-10 of the MS SQL Server query and lines 5-11 of the Oracle query) we get one more random record that does not coincide with the one in the first Common Table Expression, the `WHERE` clause is responsible for this.

The main query code, which comes after the Common Table Expressions, only prepares the result as a single row.

Let's try to execute the queries several times. If there is not much data (and we have only four records in our subscribers table), very soon we will see a situation like this:

id_first	id_second
3	3

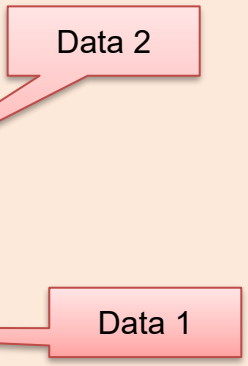
But how?! The second Common Table Expression clearly says in the `WHERE` condition not to take the record, which is already selected by the first Common Table Expression!

The point is that the results of calculating Common Table Expressions in MS SQL Server and Oracle are **non-persistent**. In other words, they are generated anew each time we access the Common Table Expression, so the data set in the `WHERE` section subquery is **not** the same as the data set returned elsewhere in the query.

Let's explain it again (graphically) on the example of MS SQL Server solution.

MS SQL Solution 7.4.2.b (incorrect solution)

```
1    WITH [first_source]
2         AS (SELECT TOP 1 [s_id]
3             FROM   [subscribers]
4             ORDER  BY NEWID()),
5         [second_source]
6         AS (SELECT TOP 1 [s_id]
7             FROM   [subscribers]
8             WHERE  [s_id] != (SELECT [s_id]
9                               FROM   [first_source])
10            ORDER  BY NEWID())
11   SELECT [first_source].[s_id]  AS [id_first],
12          [second_source].[s_id] AS [id_second]
13   FROM        [first_source]
14   CROSS JOIN [second_source]
```



"Data 1" and "Data 2" are **different** data. Suppose that "Data 1" is a randomly chosen identifier with a value of 3. But when the second Common Table Expression is executed, "Data 2" gives a different number (e.g., 1) and the second random identifier is compared already to that number (1), not to the one chosen earlier (3).

In such a situation, nothing prevents the second Common Table Expression from returning the same value that was returned by the first Common Table Expression. Thus, we get duplicated identifier values, which contradicts the problem condition.

Unfortunately, this problem has no solution in MS SQL Server, it is impossible to "materialize" the results of Common Table Expressions calculations in this DBMS. In Oracle, however, there is a workaround.

Oracle Solution 7.4.2.b

```
1    WITH "first_source"
2         AS (SELECT /*+ MATERIALIZE */ "s_id",
3                    ROW_NUMBER() OVER (ORDER BY DBMS_RANDOM.RANDOM) AS "rn"
4             FROM   "subscribers"),
5         "second_source"
6         AS (SELECT /*+ MATERIALIZE */ "s_id",
7                    ROW_NUMBER() OVER (ORDER BY DBMS_RANDOM.RANDOM) AS "rn"
8             FROM   "subscribers"
9             WHERE  "s_id" != (SELECT "s_id"
10                              FROM   "first_source"
11                              WHERE  "rn" = 1))
12   SELECT "first_source"."s_id"  AS "id_first",
13          "second_source"."s_id" AS "id_second"
14   FROM        "first_source"
15   CROSS JOIN "second_source"
16   WHERE "first_source"."rn" = 1
17     AND "second_source"."rn" = 1
```

Note the `/*+ MATERIALIZE */` construct in lines 2 and 6: it is a so-called "query hint" telling the DBMS to "materialize" (by analogy with materialized views, see Example 27{229}) the results of a Common Table Expression execution. Now the results of Common Table Expression evaluation are fixed, and the problem described above disappears, i.e., "Data 1" and "Data 2" are the same.

Warning! The `/*+ MATERIALIZE */` construct is an undocumented feature of Oracle and may be discontinued at any time. So, although it works, it cannot be considered reliable.

Task 7.4.2.TSK.A: optimize the solution{554} of problem 7.4.2.a{554} so that two random entries for the first set and one random entry for the second set are obtained without data ordering by a random number.

Task 7.4.2.TSK.B: implement the solution{554} of problem 7.4.2.a{554} as a stored procedure without obtaining the Cartesian product of the sets.

7.4.3. Example 53: Working with JSON

Problem 7.4.3.a{562}: rewrite the solution of problem 2.2.2.b{74} so that information (including identifiers) about the authors and genres of each book is presented in JSON format.

Problem 7.4.3.b{564}: extract information about books, genres and authors from the text in JSON format and present it as a table with separate columns for books' names and lists of authors' identifiers, authors' names, genres' identifiers, genres' names.

Expected result 7.4.3.a.

book	author(s)	genre(s)
Eugene Onegin	[{"id": 7, "name": "Alexander Pushkin"}]	[{"id": 1, "name": "Poetry"},{"id": 5, "name": "Classic"}]
The Art of Computer Programming	[{"id": 1, "name": "Donald Knuth"}]	[{"id": 2, "name": "Programming"},{"id": 5, "name": "Classic"}]
Course of Theoretical Physics	[{"id": 4, "name": "Lev Landau"},{"id": 5, "name": "Evgeny Lifshitz"}]	[{"id": 5, "name": "Classic"}]
Foundation and Empire	[{"id": 2, "name": "Isaac Asimov"}]	[{"id": 6, "name": "Science Fiction"}]
Programming Psychology	[{"id": 3, "name": "Dale Carnegie"},{"id": 6, "name": "Bjarne Stroustrup"}]	[{"id": 2, "name": "Programming"},{"id": 3, "name": "Psychology"}]
The Fisherman and the Golden Fish	[{"id": 7, "name": "Alexander Pushkin"}]	[{"id": 1, "name": "Poetry"},{"id": 5, "name": "Classic"}]
The C++ Programming Language	[{"id": 6, "name": "Bjarne Stroustrup"}]	[{"id": 2, "name": "Programming"}]

Expected result 7.4.3.b.

book	author(s)_id(s)	author(s)_names(s)	genre(s)_id(s)	genre(s)_names(s)
Eugene Onegin	7	Alexander Pushkin	1,5	Classic,Poetry
The Art of Computer Programming	1	Donald Knuth	2,5	Classic, Programming
Course of Theoretical Physics	4,5	Evgeny Lifshitz, Lev Landau	5	Classic
Foundation and Empire	2	Isaac Asimov	6	Science Fiction
Programming Psychology	3,6	Bjarne Stroustrup, Dale Carnegie	2,3	Programming, Psychology
The Fisherman and the Golden Fish	7	Alexander Pushkin	1,5	Classic,Poetry
The C++ Programming Language	6	Bjarne Stroustrup	2	Programming

Solution 7.4.3.a{561}.

Traditionally, let's start the solution with MySQL and consider an "unscientific", but extremely convenient general solution based on "manual" JSON data generation (for the example of JSON format data, see the solution 7.4.3.b{564}).

To get the code below, just take the solution of problem 2.2.2.b{74} and add the JSON syntax formation to the **SELECT** part.

MySQL Solution 7.4.3.a (simplified option)

```
SELECT `b_name`
       AS `book`,
       CONCAT('[', GROUP_CONCAT(DISTINCT
        CONCAT('{"id": ', `a_id`, ', "name": "', `a_name`, '"}')
        ORDER BY `a_id` SEPARATOR ','), ']')
       AS `author(s)`,
       CONCAT('[', GROUP_CONCAT(DISTINCT
        CONCAT('{"id": ', `g_id`, ', "name": "', `g_name`, '"}')
        ORDER BY `g_id` SEPARATOR ','), ']')
       AS `genre(s)`
FROM   `books`
       JOIN `m2m_books_authors` USING(`b_id`)
       JOIN `authors` USING(`a_id`)
       JOIN `m2m_books_genres` USING(`b_id`)
       JOIN `genres` USING(`g_id`)
GROUP  BY `b_id`
ORDER  BY `b_name`
```

If we approach this question "scientifically", the solution turns out to be a bit more cumbersome. This is caused by the fact that the **JSON_ARRAYAGG** function in MySQL does not support the **DISTINCT** mode, i.e., we have to separately prepare information about authors and by genres, and then combine this information with the final result.

MySQL Solution 7.4.3.a ("scientific" option)

```
WITH `deduplicated_authors` AS
(
    SELECT `b_id`,
           JSON_ARRAYAGG(JSON_OBJECT('id', `a_id`, 'name', `a_name`))
           AS `author(s)`
    FROM   `books`
           JOIN `m2m_books_authors` USING(`b_id`)
           JOIN `authors` USING(`a_id`)
    GROUP  BY `b_id`),
`deduplicated_genres` AS
(
    SELECT `b_id`,
           JSON_ARRAYAGG(JSON_OBJECT('id', `g_id`, 'name', `g_name`))
           AS `genre(s)`
    FROM   `books`
           JOIN `m2m_books_genres` USING(`b_id`)
           JOIN `genres` USING(`g_id`)
    GROUP  BY `b_id`
)
SELECT `b_name` AS `book`,
       `author(s)`,
       `genre(s)`
  FROM `books`
  JOIN `deduplicated_authors` USING(`b_id`)
  JOIN `deduplicated_genres` USING(`b_id`)
```

Moving on to MS SQL Server.

Despite the presence of such a powerful mechanism in MS SQL Server as `FOR JSON PATH`, in this particular case it will not help us in any way, because we still cannot overcome a lot of restrictions of `STRING_AGG` function, so here we have to form a solution, which is a mixture of just discussed for MySQL "unscientific" and "scientific" approaches.

We'll take the logic of forming JSON format "manually" (lines 4-8 and 18-22 of the query) from "unscientific" option, and the logic of successive preparation of information (in `deduplicated_authors` and `deduplicated_genres` Common Table Expressions) about authors and genres and then combining these data into a single result (lines 30-37 of the query) from "scientific" option.

In total, the solution is as follows.

MS SQL | Solution 7.4.3.a

```
1    WITH [deduplicated_authors] AS
2    (
3        SELECT [books].[b_id],
4        CONCAT('[', STRING_AGG(CONCAT('{"id": ', [authors].[a_id],
5                                      ', "name": "', [authors].[a_name], '"}'),
6                                      ',') WITHIN GROUP
7                                      (ORDER BY [authors].[a_id] ASC), ']')
8        AS [author(s)]
9        FROM [books]
10            JOIN [m2m_books_authors]
11             ON [books].[b_id] = [m2m_books_authors].[b_id]
12            JOIN [authors]
13             ON [m2m_books_authors].[a_id] = [authors].[a_id]
14       GROUP  BY [books].[b_id]),
15   [deduplicated_genres] AS
16   (
17       SELECT [books].[b_id],
18       CONCAT('[', STRING_AGG(CONCAT('{"id": ', [genres].[g_id],
19                                      ', "name": "', [genres].[g_name], '"}'),
20                                      ',') WITHIN GROUP
21                                      (ORDER BY [genres].[g_id] ASC), ']')
22       AS [genre(s)]
23       FROM [books]
24            JOIN [m2m_books_genres]
25             ON [books].[b_id] = [m2m_books_genres].[b_id]
26            JOIN [genres]
27             ON [m2m_books_genres].[g_id] = [genres].[g_id]
28       GROUP  BY [books].[b_id]
29   )
30   SELECT [b_name] AS [book],
31          [author(s)],
32          [genre(s)]
33     FROM [books]
34     JOIN [deduplicated_authors]
35           ON [books].[b_id] = [deduplicated_authors].[b_id]
36     JOIN [deduplicated_genres]
37           ON [deduplicated_authors].[b_id] = [deduplicated_genres].[b_id]
```

Moving on to Oracle.

Solution for this DBMS will be completely similar to the "scientific" solution for MySQL, except for the need to explicitly specify the key location and values in the `JSON_OBJECT` function arguments.

Oracle | Solution 7.4.3.a

```
WITH "deduplicated_authors" AS
(
    SELECT "b_id",
             JSON_ARRAYAGG(
             JSON_OBJECT(KEY 'id' VALUE "a_id", KEY 'name' VALUE "a_name"))
           AS "author(s)"
    FROM   "books"
           JOIN "m2m_books_authors" USING("b_id")
           JOIN "authors" USING("a_id")
    GROUP  BY "b_id"),
"deduplicated_genres" AS
(
    SELECT "b_id",
           JSON_ARRAYAGG(
           JSON_OBJECT(KEY 'id' VALUE "g_id", KEY 'name' VALUE "g_name"))
           AS "genre(s)"
    FROM   "books"
           JOIN "m2m_books_genres" USING("b_id")
           JOIN "genres" USING("g_id")
    GROUP  BY "b_id"
)
SELECT "b_name" AS "book",
       "author(s)",
       "genre(s)"
  FROM "books"
  JOIN "deduplicated_authors" USING("b_id")
  JOIN "deduplicated_genres" USING("b_id")
```

This completes the solution of this problem.

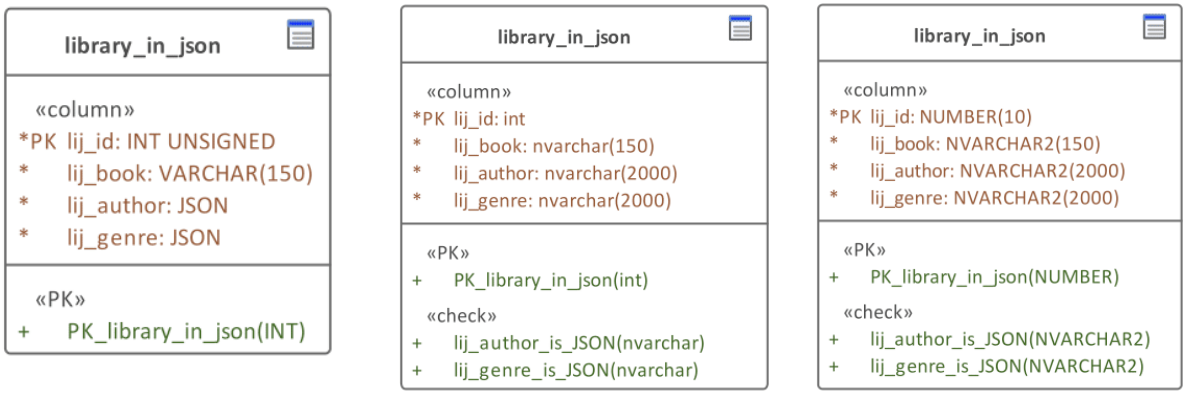
Solution 7.4.3.b[561].

To solve this problem, we need data in JSON format. Suppose they look like this.

book	author(s)	genre(s)
Eugene Onegin	[{"id": 7, "name": "Alexander Pushkin"}]	[{"id": 1, "name": "Poetry"},{"id": 5, "name": "Classic"}]
The Art of Computer Programming	[{"id": 1, "name": "Donald Knuth"}]	[{"id": 2, "name": "Programming"},{"id": 5, "name": "Classic"}]
Course of Theoretical Physics	[{"id": 4, "name": "Lev Landau"},{"id": 5, "name": "Evgeny Lifshitz"}]	[{"id": 5, "name": "Classic"}]
Foundation and Empire	[{"id": 2, "name": "Isaac Asimov"}]	[{"id": 6, "name": "Science Fiction"}]
Programming Psychology	[{"id": 3, "name": "Dale Carnegie"},{"id": 6, "name": "Bjarne Stroustrup"}]	[{"id": 2, "name": "Programming"},{"id": 3, "name": "Psychology"}]
The Fisherman and the Golden Fish	[{"id": 7, "name": "Alexander Pushkin"}]	[{"id": 1, "name": "Poetry"},{"id": 5, "name": "Classic"}]
The C++ Programming Language	[{"id": 6, "name": "Bjarne Stroustrup"}]	[{"id": 2, "name": "Programming"}]

Figure 7.d shows the schemas of the `library_in_json` table for all three DBMSes (we will create it in the "Exploration" database), and then we will show the SQL code for creating it and filling it with data.

Note: in MySQL we use `JSON` data type explicitly, while in MS SQL Server and Oracle (where it does not exist), we use `CHECK` to control the data format.



MySQL MS SQL Server Oracle

Figure 7.d — The `library_in_json` table in all three DBMSes

In the case of MySQL, we create the `library_in_json` table and fill it with data using these queries.

MySQL Solution 7.4.3.b (creating and filling the table)

```
CREATE TABLE `library_in_json`
(
    `lij_id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    `lij_book` VARCHAR(150) NOT NULL,
    `lij_author` JSON,
    `lij_genre` JSON,
    PRIMARY KEY (`lij_id`)
)
ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

INSERT INTO `library_in_json`
(   `lij_book`,
    `lij_author`,
    `lij_genre`)
VALUES
('Eugene Onegin', '[{"id":7,"name":"Alexander Pushkin"}]',
 '[{"id":1,"name":"Poetry"},{"id":5,"name":"Classic"}]'),
('The Art of Computer Programming', '[{"id":1,"name":"Donald Knuth"}]',
 '[{"id":2,"name":"Programming"},{"id":5,"name":"Classic"}]'),
('Course of Theoretical Physics',
 '[{"id":4,"name":"Lev Landau"},{"id":5,"name":"Evgeny Lifshitz"}]',
 '[{"id":5,"name":"Classic"}]'),
('Foundation and Empire', '[{"id":2,"name":"Isaac Asimov"}]',
 '[{"id":6,"name":"Science Fiction"}]'),
('Programming Psychology',
 '[{"id":3,"name":"Dale Carnegie"},{"id":6,"name":"Bjarne Stroustrup"}]',
 '[{"id":2,"name":"Programming"},{"id":3,"name":"Psychology"}]'),
('The Fisherman and the Golden Fish', '[{"id":7,"name":"Alexander Push-
kin"}]',
 '[{"id":1,"name":"Poetry"},{"id":5,"name":"Classic"}]'),
('The C++ Programming Language', '[{"id":6,"name":"Bjarne Stroustrup"}]',
 '[{"id":2,"name":"Programming"}]');
```

To retrieve data from such a table, we can use the following query.

MySQL Solution 7.4.3.b (*almost* correct solution)

```
SELECT `lij_book` AS `book`,
       JSON_EXTRACT(`lij_author`, '$**.id')     AS `author(s)_id(s)`,
       JSON_EXTRACT(`lij_author`, '$**.name')   AS `author(s)_names(s)`,
       JSON_EXTRACT(`lij_genre`, '$**.id')      AS `genre(s)_id(s)`,
       JSON_EXTRACT(`lij_genre`, '$**.name')    AS `genre(s)_names(s)`
  FROM `library_in_json`
```

This query produces almost correct results, except the data is enclosed in square brackets and quotation marks, which cannot be removed by simple means.

book	author(s)_id(s)	author(s)_names(s)	genre(s)_id(s)	genre(s)_names(s)
Eugene Onegin	[7]	["Alexander Pushkin"]	[1, 5]	["Poetry", "Classic"]
The Art of Computer Programming	[1]	["Donald Knuth"]	[2, 5]	["Programming", "Classic"]
Course of Theoretical Physics	[4, 5]	["Lev Landau", "Evgeny Lifshitz"]	[5]	["Classic"]
Foundation and Empire	[2]	["Isaac Asimov"]	[6]	["Science Fiction"]
Programming Psychology	[3, 6]	["Dale Carnegie", "Bjarne Stroustrup"]	[2, 3]	["Programming", "Psychology"]
The Fisherman and the Golden Fish	[7]	["Alexander Pushkin"]	[1, 5]	["Poetry", "Classic"]
The C++ Programming Language	[6]	["Bjarne Stroustrup"]	[2]	["Programming"]

But we can create a very simple function that will form the final representation of the data exactly as it is required by the problem.

The function code looks like this.

MySQL Solution 7.4.3.b (function code for the completely correct solution)

```
DELIMITER $$
CREATE FUNCTION EXTRACT_JSON_AS_PLAIN_TEXT(json_text VARCHAR(21845),
                                           field_name VARCHAR(21845))
RETURNS VARCHAR(21845) DETERMINISTIC
BEGIN
 DECLARE final_result VARCHAR(21845);
 DECLARE i INT;
 SET final_result = '';
 SET i = 0;
 WHILE (i < JSON_LENGTH(json_text, '$') ) DO
    SET final_result = CONCAT(final_result, ',',
                              JSON_UNQUOTE(JSON_EXTRACT(json_text,
                               CONCAT( '$[', i, '].', field_name))));
    SET i = i + 1;
  END WHILE;
 RETURN SUBSTR(final_result, 2);
END$$
DELIMITER ;
```

Now it remains to apply the created function, and the query, which gives exactly the result required by the problem, will look like this.

MySQL Solution 7.4.3.b (solution using the function)

```
SELECT `lij_book` AS `book`,
       EXTRACT_JSON_AS_PLAIN_TEXT(`lij_author`, 'id')
        AS `author(s)_id(s)`,
       EXTRACT_JSON_AS_PLAIN_TEXT(`lij_author`, 'name')
        AS `author(s)_names(s)`,
       EXTRACT_JSON_AS_PLAIN_TEXT(`lij_genre`, 'id')
        AS `genre(s)_id(s)`,
       EXTRACT_JSON_AS_PLAIN_TEXT(`lij_genre`, 'name')
        AS `genre(s)_names(s)`
  FROM `library_in_json`
```

Moving on to MS SQL Server.
Here we also start by creating a table and filling it with data.

MS SQL Solution 7.4.3.b (creating and filling the table)

```
1    CREATE TABLE [library_in_json]
2    (
3        [lij_id] int NOT NULL IDENTITY (1, 1),
4        [lij_book] NVARCHAR(150) NOT NULL,
5        [lij_author] NVARCHAR(2000) NOT NULL,
6        [lij_genre] NVARCHAR(2000) NOT NULL
7    );
8
9    ALTER TABLE [library_in_json]
10       ADD CONSTRAINT [PK_lij_id]
11           PRIMARY KEY CLUSTERED ([lij_id] ASC);
12
13   ALTER TABLE [library_in_json]
14       ADD CONSTRAINT [lij_author_is_JSON] CHECK (ISJSON([lij_author])=1);
15
16   ALTER TABLE [library_in_json]
17       ADD CONSTRAINT [lij_genre_is_JSON] CHECK (ISJSON([lij_genre])=1);
18
19   INSERT INTO [library_in_json]
20   (
21       [lij_book],
22       [lij_author],
23       [lij_genre]
24   ) VALUES
25   (N'Eugene Onegin', N'[{"id":7,"name":"Alexander Pushkin"}]',
26    N'[{"id":1,"name":"Poetry"},{"id":5,"name":"Classic"}]'),
27   (N'The Art of Computer Programming', N'[{"id":1,"name":"Donald Knuth"}]',
28    N'[{"id":2,"name":"Programming"},{"id":5,"name":"Classic"}]'),
29   (N'Course of Theoretical Physics',
30    N'[{"id":4,"name":"Lev Landau"},{"id":5,"name":"Evgeny Lifshitz"}]',
31    N'[{"id":5,"name":"Classic"}]'),
32   (N'Foundation and Empire', N'[{"id":2,"name":"Isaac Asimov"}]',
33    N'[{"id":6,"name":"Science Fiction"}]'),
34   (N'Programming Psychology',
35    N'[{"id":3,"name":"Dale Carnegie"},{"id":6,"name":"Bjarne Stroustrup"}]',
36    N'[{"id":2,"name":"Programming"},{"id":3,"name":"Psychology"}]'),
37   (N'The Fisherman and the Golden Fish', N'[{"id":7,"name":"Alexander Push-
38   kin"}]',
39    N'[{"id":1,"name":"Poetry"},{"id":5,"name":"Classic"}]'),
40   (N'The C++ Programming Language', N'[{"id":6,"name":"Bjarne Stroustrup"}]',
      N'[{"id":2,"name":"Programming"}]');
```

We will then base our solution on the approach considered in the solution{79} of problem 2.2.2.b{74}. Unlike MySQL, in MS SQL Server we can get exactly the required result without creating an additional function.

The query will look like this (see next page).

The **`OPENJSON`** function (lines 13-15 and 33-33 of the query), which returns the result of JSON format processing as a table, allows us to extract data from JSON format.

Then, using the **`STRING_AGG`** function (lines 4-9 and 22-27 of the query), we get the data in aggregated form (converting the table representation into a string representation by grouping the data)).

Two Common Table Expressions (lines 1-18 and 19-36 of the query) are needed to get information about authors and genres without duplication (this is shown in detail in the solution{79} of problem 2.2.2.b{74}).

And finally, in lines 37-46 of the query we combine all the pre-prepared data and get the final result.

MS SQL Solution 7.4.3.b

```
WITH [authors_ready] AS
(
 SELECT [main_table].[lij_id],
        STRING_AGG([authors_data].[id], ',')
         WITHIN GROUP (ORDER BY [authors_data].[id] ASC)
          AS [author(s)_id(s)],
        STRING_AGG([authors_data].[name], ',')
         WITHIN GROUP (ORDER BY [authors_data].[id] ASC)
          AS [author(s)_name(s)]
 FROM [library_in_json] AS [main_table]
 CROSS APPLY (
  SELECT *
  FROM OPENJSON([main_table].[lij_author], '$')
  WITH ([id] nvarchar(100) '$.id',
        [name] nvarchar(100) '$.name')
 ) AS [authors_data]
 GROUP BY [main_table].[lij_id]
),
[genres_ready] AS
(
 SELECT [main_table].[lij_id],
        STRING_AGG([genres_data].[id], ',')
         WITHIN GROUP (ORDER BY [genres_data].[id] ASC)
          AS [genre(s)_id(s)],
        STRING_AGG([genres_data].[name], ',')
         WITHIN GROUP (ORDER BY [genres_data].[id] ASC)
          AS [genre(s)_name(s)]
 FROM [library_in_json] AS [main_table]
 CROSS APPLY (
  SELECT *
  FROM OPENJSON([main_table].[lij_genre], '$')
  WITH ([id] nvarchar(100) '$.id',
        [name] nvarchar(100) '$.name')
 ) AS [genres_data]
 GROUP  BY [main_table].[lij_id]
)
SELECT [library_in_json].[lij_book] AS [book],
       [author(s)_id(s)],
       [author(s)_name(s)],
       [genre(s)_id(s)],
       [genre(s)_name(s)]
  FROM [library_in_json]
  JOIN [authors_ready]
    ON [library_in_json].[lij_id] = [authors_ready].[lij_id]
  JOIN [genres_ready]
    ON [authors_ready].[lij_id] = [genres_ready].[lij_id]
```

Moving on to Oracle.

Here, too, we'll start by creating a table and filling it with data. To simplify the code (to avoid creating a sequence and a trigger), we will not make the primary key autoincrementable in this table, which will not affect any further solution.

Oracle Solution 7.4.3.b (creating and filling the table)

```
CREATE TABLE "library_in_json"
(
 "lij_id" NUMBER(10,0) NOT NULL PRIMARY KEY,
 "lij_book" VARCHAR2(150) NOT NULL,
 "lij_author" VARCHAR2 (2000)
  CONSTRAINT "lij_author_is_JSON" CHECK ("lij_author" IS JSON),
 "lij_genre" VARCHAR2 (2000)
  CONSTRAINT "lij_genre_is_JSON" CHECK ("lij_genre" IS JSON)
);
```

Oracle Solution 7.4.3.b (creating and filling the table) (continued)

```
  INSERT ALL
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (1, N'Eugene Onegin', N'[{"id":7,"name":"Alexander Pushkin"}]',
                N'[{"id":1,"name":"Poetry"},{"id":5,"name":"Classic"}]')
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (2, N'The Fisherman and the Golden Fish',
                N'[{"id":7,"name":"Alexander Pushkin"}]',
                N'[{"id":1,"name":"Poetry"},{"id":5,"name":"Classic"}]')
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (3, N'Foundation and Empire', N'[{"id":2,"name":"Isaac Asi-
mov"}]',
                N'[{"id":6,"name":"Science Fiction"}]')
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (4, N'Programming Psychology',
                N'[{"id":3,"name":"Dale Carnegie"},
                   {"id":6,"name":"Bjarne Stroustrup"}]',
                N'[{"id":2,"name":"Programming"},
                   {"id":3,"name":"Psychology"}]')
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (5, N'The C++ Programming Language',
                N'[{"id":6,"name":"Bjarne Stroustrup"}]',
                N'[{"id":2,"name":"Programming"}]')
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (6, N'Course of Theoretical Physics',
                N'[{"id":4,"name":"Lev Landau"},
                   {"id":5,"name":"Evgeny Lifshitz"}]',
                N'[{"id":5,"name":"Classic"}]')
    INTO "library_in_json" ("lij_id", "lij_book", "lij_author", "lij_genre")
     VALUES (7, N'The Art of Computer Programming',
                N'[{"id":1,"name":"Donald Knuth"}]',
                N'[{"id":2,"name":"Programming"},
                   {"id":5,"name":"Classic"}]')
  SELECT 1 FROM "DUAL";
```

If for some reason we are satisfied with a result that does not fully meet the conditions of the problem, we can use this simplified and not completely correct solution.

Oracle Solution 7.4.3.b (simplified and not completely correct solution)

```
  WITH "prepared_data" AS
  (
   SELECT "inner_data_source"."lij_book",
          TREAT ("inner_data_source"."lij_author" AS JSON) AS "authors_data",
          TREAT ("inner_data_source"."lij_genre" AS JSON) AS "genres_data"
     FROM "library_in_json" "inner_data_source"
  )
  SELECT "prepared_data_source"."lij_book" AS "book",
         "prepared_data_source"."authors_data".id AS "author(s)_id(s)",
         "prepared_data_source"."authors_data".name AS "author(s)_name(s)",
         "prepared_data_source"."genres_data".id AS "genre(s)_id(s)",
         "prepared_data_source"."genres_data".name AS "genre(s)_name(s)"
    FROM "prepared_data" "prepared_data_source"
```

Here, data representing a “set of objects” will be framed in quotes and/or square brackets, which is not very convenient (and very difficult to eliminate, despite the apparent simplicity of the task).

book	author(s)_id(s)	author(s)_name(s)	genre(s)_id(s)	genre(s)_name(s)
Eugene Onegin	7	Alexander Push-kin	[1,5]	["Poetry", "Classic"]
The Fisherman and the Golden Fish	7	Alexander Push-kin	[1,5]	["Poetry", "Classic"]
Foundation and Empire	2	Isaac Asimov	6	Science Fiction
Programming Psy-chology	[3,6]	["Dale Carnegie", "Bjarne Strou-strup"]	[2,3]	["Programming", "Psychology"]
The C++ Program-ming Language	6	Bjarne Stroustrup	2	Programming
Course of Theoreti-cal Physics	[4,5]	["Lev Landau", "Evgeny Lifshitz"]	5	Classic
The Art of Com-puter Programming	1	Donald Knuth	[2,5]	["Programming", "Classic"]

But if we still want to get the data exactly as it is required by the condition of the problem, we have to go the other way.

Despite its great volume, the query (see next page) is simple and based on a few trivial principles:

- four Common Table Expressions (lines 1-13, 14-26, 27-39, 40-52 of the query) generate ready sets of data on the same principle (this approach was chosen in order to avoid complicating the internal logic);
- within each Common Table Expression using the **`JSON_TABLE`** function, a table representation of the data being searched for is generated (lines 8-10, 21-23, 34-36, 47-49 of the query);
- this table representation is then converted into an aggregated (string) form using the **`LISTAGG`** function (lines 4-6, 17-19, 30-32, 43-45 of the query); if in some case the result of processing text data will be presented with an encoding violation, we can use an additional encoding conversion, as shown in the solution{79} of problem 2.2.2.b{74};
- then the prepared data are combined into the final set (lines 53-62 of the query).

Oracle | Solution 7.4.3.b

```
WITH "authors_ids" AS
(
 SELECT "library_in_json"."lij_id",
        LISTAGG("json_extracted"."id", ',')
         WITHIN GROUP (ORDER BY "json_extracted"."id")
          AS "author(s)_id(s)"
 FROM   "library_in_json",
        JSON_TABLE("lij_author", '$'
                   COLUMNS (NESTED PATH '$.id[*]'
                                COLUMNS ("id" varchar2(2000) PATH '$')))
        "json_extracted"
 GROUP  BY "library_in_json"."lij_id"
),
"authors_names" AS
(
 SELECT "library_in_json"."lij_id",
        LISTAGG("json_extracted"."name", ',')
         WITHIN GROUP (ORDER BY "json_extracted"."name")
          AS "author(s)_name(s)"
 FROM   "library_in_json",
        JSON_TABLE("lij_author", '$'
                   COLUMNS (NESTED PATH '$.name[*]'
                                COLUMNS ("name" varchar2(2000) PATH '$')))
        "json_extracted"
 GROUP  BY "library_in_json"."lij_id"
),
"genres_ids" AS
(
 SELECT "library_in_json"."lij_id",
        LISTAGG("json_extracted"."id", ',')
         WITHIN GROUP (ORDER BY "json_extracted"."id")
           AS "genres(s)_id(s)"
 FROM   "library_in_json",
        JSON_TABLE("lij_genre", '$'
                   COLUMNS (NESTED PATH '$.id[*]'
                                COLUMNS ("id" varchar2(2000) PATH '$')))
        "json_extracted"
 GROUP  BY "library_in_json"."lij_id"
),
"genres_names" AS
(
 SELECT "library_in_json"."lij_id",
        LISTAGG("json_extracted"."name", ',')
         WITHIN GROUP (ORDER BY "json_extracted"."name")
          AS "genres(s)_name(s)"
 FROM   "library_in_json",
        JSON_TABLE("lij_genre", '$'
                   COLUMNS (NESTED PATH '$.name[*]'
                                COLUMNS ("name" varchar2(2000) PATH '$')))
        "json_extracted"
 GROUP  BY "library_in_json"."lij_id"
)
SELECT "library_in_json"."lij_book",
       "authors_ids"."author(s)_id(s)",
       "authors_names"."author(s)_name(s)",
       "genres_ids"."genres(s)_id(s)",
       "genres_names"."genres(s)_name(s)"
  FROM "library_in_json"
  JOIN "authors_ids" USING("lij_id")
  JOIN "authors_names" USING("lij_id")
  JOIN "genres_ids" USING("lij_id")
  JOIN "genres_names" USING("lij_id")
```

This completes the solution of this problem.

Task 7.4.3.TSK.A: rewrite the “scientific” version of the solution 7.4.3.a{562} for MySQL without using Common Table Expressions.

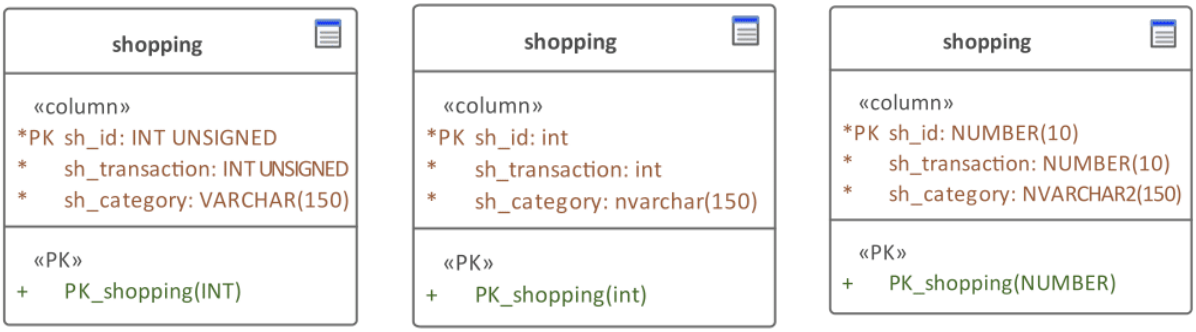
Task 7.4.3.TSK.B: rewrite the solution 7.4.3.a{562} for Oracle without using `JSON_ARRAYAGG` and `JSON_OBJECT` functions (i.e., implement “manual” JSON format generation).

Task 7.4.3.TSK.C: rewrite the solution 7.4.3.b{564} for Oracle using a function (similar to the one in the solution 7.4.3.b{564} for MySQL).

7.4.4. Example 54: Working with Pivot Data

To solve the first two problems in this example, we will need a new table, which we will create in the “Exploration” database. This table (taken out of context, so it looks a bit unusual) will store information about customers’ purchases of some hypothetical store.

It is known that at one time the customer could buy any number of goods from any categories. For simplicity and clarity of the solution we will store information about a group of purchases as a “transaction identifier” (receipt identifier), and we will store the names of product categories explicitly, i.e., as text (yes, this situation will never occur in real life, but the solutions shown below are already complicated, so we will not overcomplicate it).



MySQL MS SQL Server Oracle

Figure 7.e — The `shopping` table in all three DBMSes

Let’s save the following data set in the `shopping` table.

sh_id	sh_transaction	sh_category	
1	1	Bag	These three items were bought together.
2	1	Dress	
3	1	Bag	
4	2	Bag	These two items were bought together.
5	2	Skirt	
6	3	Dress	These four items were bought together.
7	3	Skirt	
8	3	Shoes	
9	3	Hat	
10	4	Bag	This item was bought separately.

Now that all the data have been prepared, we can move on to the problems.

Problem 7.4.4.a{574}: extract as a pivot table the information about how many times customers bought items of a certain category together with the items of other categories.

Problem 7.4.4.b{578}: calculate how many times each item was bought “by itself” and “with other items” (and the percentage of such purchases).

Problem 7.4.4.c{581}: extract as a pivot table the number of books borrowed from the library in each month (arranged horizontally) of each year (arranged vertically).

Expected result 7.4.4.a.

	Dress	Bag	Shoes	Hat	Skirt
Dress	2	1	1	1	1
Bag	1	3	0	0	1
Shoes	1	0	1	1	1
Hat	1	0	1	1	1
Skirt	1	1	1	1	2

Expected result 7.4.4.b.

category	multi_count	single_count	multi_perc	single_perc
Dress	2	0	100	0
Bag	2	1	66.6667	33.3333
Shoes	1	0	100	0
Hat	1	0	100	0
Skirt	2	0	100	0

Expected result 7.4.4.c.

year	1	2	3	4	5	6	7	8	9	10	11	12
2011	2	0	0	0	0	0	0	0	0	0	0	0
2012	0	0	0	0	1	2	0	0	0	0	0	0
2014	0	0	0	0	0	0	0	3	0	0	0	0
2015	0	0	0	0	0	0	0	0	0	3	0	0

Solution 7.4.4.a{573}.

From a mathematical point of view, the basis for solving this problem is to get the so-called Cartesian product[53] of the set of purchases on itself. Using SQL capabilities, the desired result can be obtained by performing a query with `FULL OUTER JOIN`.

Since MySQL still does not support such an operation, the desired result in this DBMS is obtained by combining the results of queries with `LEFT OUTER JOIN` and `RIGHT OUTER JOIN` through the `UNION` operator.

`UNION` operator works in `DISTINCT` mode by default, so in MySQL we immediately get the desired result, but in MS SQL Server and Oracle we must explicitly add the `DISTINCT` keyword, to avoid getting duplicates in the final data set.

So, in all three DBMS we get the following data set.

[53] "Cartesian Product" [https://en.wikipedia.org/wiki/Cartesian_product]

id	c1	c2
1	Dress	Dress
1	Dress	Bag
1	Bag	Dress
1	Bag	Bag
2	Bag	Bag
2	Bag	Skirt
2	Skirt	Bag
2	Skirt	Skirt
3	Dress	Dress
3	Dress	Shoes
3	Dress	Hat
3	Dress	Skirt
3	Shoes	Dress
3	Shoes	Shoes
3	Shoes	Hat
3	Shoes	Skirt
3	Hat	Dress
3	Hat	Shoes
3	Hat	Hat
3	Hat	Skirt
3	Skirt	Dress
3	Skirt	Shoes
3	Skirt	Hat
3	Skirt	Skirt
4	Bag	Bag

Further logic is to calculate for each element (without duplications) from column `c1` how many times each element from column `c2` occurs. This is exactly the idea of pivot tables.

MySQL Solution 7.4.4.a (fixed set of columns)

```
WITH `cartesian_product` AS
(
 SELECT `t1`.`sh_transaction` AS `id`,
        `t1`.`sh_category`    AS `c1`,
        `t2`.`sh_category`    AS `c2`
 FROM   `shopping` AS `t1`
        LEFT OUTER JOIN `shopping` AS `t2`
                     ON `t1`.`sh_transaction` = `t2`.`sh_transaction`
 UNION
 SELECT `t1`.`sh_transaction` AS `id`,
        `t1`.`sh_category`    AS `c1`,
        `t2`.`sh_category`    AS `c2`
 FROM   `shopping` AS `t1`
        RIGHT OUTER JOIN `shopping` AS `t2`
                     ON `t1`.`sh_transaction` = `t2`.`sh_transaction`
)
SELECT
  `c1`,
  COUNT(IF(`c2` = 'Dress', 1, NULL)) AS `Dress`,
  COUNT(IF(`c2` = 'Bag', 1, NULL)) AS `Bad`,
  COUNT(IF(`c2` = 'Shoes', 1, NULL)) AS `Shoes`,
  COUNT(IF(`c2` = 'Hat', 1, NULL)) AS `Hat`,
  COUNT(IF(`c2` = 'Skirt', 1, NULL)) AS `Skirt`
FROM `cartesian_product`
GROUP BY `c1`
ORDER BY `c1`
```

Since MySQL does not support the `PIVOT` operator, here we can emulate its behavior by using conditional constructions inside the parameter of the `COUNT` function (lines 19-23 of the query).

But this solution cannot be called a general one, because it is designed for a strictly defined set of product categories (and if the category name is changed or a new category is added, this query will return incorrect results).

To make this solution more general, the part of the query that generates the pivot table will have to be generated dynamically.

It looks like this.

MySQL — Solution 7.4.4.a (dynamic set of columns)

```
1   SELECT
2     GROUP_CONCAT(
3      CONCAT(', COUNT(IF(`c2` = ''',
4             `sh_category`, ''', 1, NULL)) AS `',
5             `sh_category`, '`')
6       SEPARATOR '') INTO @pivot_query
7   FROM (SELECT DISTINCT `sh_category`
8                    FROM `shopping`
9                ORDER BY `sh_category`) AS `tmp`;
10
11  SET @pivot_query = CONCAT('WITH `cartesian_product` AS
12  (
13   SELECT `t1`.`sh_transaction` AS `id`,
14          `t1`.`sh_category`    AS `c1`,
15          `t2`.`sh_category`    AS `c2`
16   FROM   `shopping` AS `t1`
17          LEFT OUTER JOIN `shopping` AS `t2`
18                       ON `t1`.`sh_transaction` = `t2`.`sh_transaction`
19   UNION
20   SELECT `t1`.`sh_transaction` AS `id`,
21          `t1`.`sh_category`    AS `c1`,
22          `t2`.`sh_category`    AS `c2`
23   FROM   `shopping` AS `t1`
24          RIGHT OUTER JOIN `shopping` AS `t2`
25                       ON `t1`.`sh_transaction` = `t2`.`sh_transaction`
26  ) SELECT
27    `c1`', @pivot_query, ' FROM `cartesian_product`
28  GROUP BY `c1`
29  ORDER BY `c1`');
30
31  SELECT @pivot_query;
32
33  PREPARE pivot_statement FROM @pivot_query;
34  EXECUTE pivot_statement;
35  DEALLOCATE PREPARE pivot_statement;
```

Of course, such a solution is difficult to consider convenient, because it requires several separate queries, so in task 7.4.4.TSK.C{584} you are suggested to make some improvements.

Moving on to MS SQL Server.

This DBMS supports both `FULL OUTER JOIN` and `PIVOT`, which greatly simplifies the solution syntax. But there are still problems with dynamic formation of a list of columns in the pivot table.

Consider two solutions (fully analogous to those just discussed for MySQL).

In the first option, we specify a fixed list of pivot table columns.

MS SQL Solution 7.4.4.a (fixed set of columns)

```
WITH [cartesian_product] AS
(
 SELECT DISTINCT [t1].[sh_transaction] AS [id],
        [t1].[sh_category]    AS [c1],
        [t2].[sh_category]    AS [c2]
 FROM   [shopping] AS [t1]
        FULL OUTER JOIN [shopping] AS [t2]
                     ON [t1].[sh_transaction] = [t2].[sh_transaction]
)
SELECT *
FROM   (SELECT [c1],
               [c2] AS [linked],
               [c2] AS [what_to_count]
        FROM   [cartesian_product]) AS [tmp]
        PIVOT ( COUNT([what_to_count])
                FOR [linked]
                 IN ([Dress], [Bag], [Shoes], [Hat], [Skirt])
              ) AS [pivot]
ORDER BY [c1]
```

In the second option, we dynamically generate the part of the query that is responsible for the set of columns of the pivot table.

MS SQL Solution 7.4.4.a (dynamic set of columns)

```
DECLARE @cols  NVARCHAR(max),
        @query NVARCHAR(max);

SET @cols = STUFF((SELECT DISTINCT ',',
                    QUOTENAME([cname]) AS [cname]
                     FROM   (SELECT [sh_category] AS [cname]
                             FROM   [shopping]) AS [tmp]
                     ORDER  BY [cname]
                     FOR XML PATH(''),
                     type).value('.', 'NVARCHAR(MAX)'), 1, 1, '');

SET @query = 'WITH [cartesian_product] AS
(
 SELECT DISTINCT [t1].[sh_transaction] AS [id],
        [t1].[sh_category]    AS [c1],
        [t2].[sh_category]    AS [c2]
 FROM   [shopping] AS [t1]
        FULL OUTER JOIN [shopping] AS [t2]
                     ON [t1].[sh_transaction] = [t2].[sh_transaction]
)
SELECT *
FROM   (SELECT [c1],
               [c2] AS [linked],
               [c2] AS [what_to_count]
        FROM   [cartesian_product]) AS [tmp]
        PIVOT ( COUNT([what_to_count])
                FOR [linked] IN (' + @cols + ')
              ) AS [pivot]
ORDER BY [c1]';

EXECUTE(@query);
```

As with MySQL, this solution is difficult to consider convenient, because it requires several separate queries, so in task 7.4.4.TSK.C{584} you are suggested to make some improvements.

Moving on to Oracle.

And let's look at the ready-made query (it is derived from the solution for MS SQL Server) through minimal syntax adjustments.

Oracle | Solution 7.4.4.a (fixed set of columns)

```
1   SELECT * FROM
2   (
3    SELECT "c1", "c2" FROM
4    (
5    SELECT DISTINCT "t1"."sh_transaction" AS "id",
6           "t1"."sh_category"    AS "c1",
7           "t2"."sh_category"    AS "c2"
8    FROM   "shopping" "t1"
9           FULL OUTER JOIN "shopping" "t2"
10                        ON "t1"."sh_transaction" = "t2"."sh_transaction"
11   ) "tmp_inner"
12  )
13  PIVOT ( COUNT("c2")
14          FOR "c2" IN ('Dress', 'Bag', 'Shoes', 'Hat', 'Skirt')
15        ) "pivot"
16  ORDER BY "c1"
```

Unfortunately, in Oracle there is no easy way to perform a dynamic query with the substitution of variables outside the body of the stored routines, so we will leave the alternative solution (similar to that considered for MySQL and MS SQL Server and based on the dynamic generation of column names) to be developed by yourself in task 7.4.4.TSK.C{584}.

This completes the solution of this problem.

Solution 7.4.4.b{573}.

This problem looks easier compared to the previous one, because the set of columns of the pivot table is always strictly fixed, and therefore there is no need to generate it dynamically.

Nevertheless, a number of data preparation operations are required here. Solutions for all three DBMSes follow the same logic, so let's consider it in detail on the example of MySQL, and for MS SQL Server and Oracle just give ready-made queries.

So, in MySQL the query looks like this (see next page).

In the first Common Table Expression (lines 1-11 of the query) we get information about whether the purchase is "single", i.e., one item was purchased, or "multi", i.e., several items were purchased in it.

The result of this Common Table Expression is as follows.

sh_transaction	type
1	multi
2	multi
3	multi
4	single

In the second Common Table Expression (lines 12-20 of the query) we get information about whether the transaction was made as part of a "single" or "multi" purchase.

The result of this Common Table Expression is as follows.

sh_transaction	sh_category	type
1	Dress	multi
1	Bag	multi
2	Bag	multi
2	Skirt	multi
3	Dress	multi
3	Shoes	multi
3	Hat	multi
3	Skirt	multi
4	Bag	single

Now it remains for each category (without duplications) to count how many times it participated in “single” and “multi” purchases, this is done in the third Common Table Expression (lines 21-31).

MySQL Solution 7.4.4.b

```
1    WITH `transaction_types` AS
2    (
3     SELECT `sh_transaction`,
4            CASE
5             WHEN COUNT(*) = 1
6                  THEN 'single'
7                  ELSE 'multi'
8            END AS `type`
9     FROM   `shopping`
10    GROUP  BY `sh_transaction`
11   ),
12   `transaction_with_types` AS
13   (
14    SELECT DISTINCT `shopping`.`sh_transaction`,
15                    `sh_category`,
16                    `type`
17    FROM   `shopping`
18      JOIN `transaction_types`
19        ON `shopping`.`sh_transaction` = `transaction_types`.`sh_transaction`
20   ),
21   `transaction_pivoted`
22   AS
23   (
24    SELECT
25     `sh_category`,
26     COUNT(IF(`type` = 'single', 1, NULL)) AS `single`,
27     COUNT(IF(`type` = 'multi', 1, NULL)) AS `multi`
28    FROM `transaction_with_types`
29    GROUP BY `sh_category`
30    ORDER BY `sh_category`
31   )
32   SELECT `sh_category` AS `category`,
33          `multi`       AS `multi_count`,
34            `single`      AS `single_count`,
35            `multi` * 100 / (`multi` + `single`) AS `multi_perc`,
36            `single` * 100 / (`multi` + `single`) AS `single_perc`
37   FROM `transaction_pivoted`
38   ORDER BY `sh_category`
```

The result of this third Common Table Expression is as follows.

sh_category	multi	single
Dress	2	0
Bag	2	1
Shoes	1	0
Hat	1	0
Skirt	2	0

All that remains is to calculate the percentage of “single” and “multi” purchases to the total number of purchases in each category, which is done in lines 35-36 of the query.

In MS SQL Server all the same actions are performed within this query.

MS SQL Solution 7.4.4.b

```
1    WITH [transaction_types] AS
2    (
3     SELECT [sh_transaction],
4            CASE
5             WHEN COUNT(*) = 1
6                  THEN 'single'
7                  ELSE 'multi'
8            END AS [type]
9     FROM   [shopping]
10    GROUP  BY [sh_transaction]
11   ),
12   [transaction_with_types] AS
13   (
14    SELECT DISTINCT [shopping].[sh_transaction],
15                    [sh_category],
16                    [type]
17    FROM   [shopping]
18      JOIN [transaction_types]
19        ON [shopping].[sh_transaction] = [transaction_types].[sh_transaction]
20   ),
21   [transaction_pivoted]
22   AS
23   (
24    SELECT *
25    FROM   (SELECT [sh_category],
26                   [type] AS [linked],
27                   [type] AS [what_to_count]
28            FROM   [transaction_with_types]) AS [tmp]
29            PIVOT  (COUNT([what_to_count])
30                    FOR [linked] IN ([multi], [single])
31                   ) AS [pivot]
32   )
33   SELECT [sh_category] AS [category],
34          [multi]       AS [multi_count],
35            [single]      AS [single_count],
36            CAST([multi] AS FLOAT) * 100 / ([multi] + [single]) AS
37   [multi_perc],
38            CAST([single] AS FLOAT) * 100 / ([multi] + [single]) AS [sin-
39   gle_perc]
40   FROM [transaction_pivoted]
41   ORDER BY [sh_category]
```

In Oracle all the same actions are performed within this query.

Oracle | Solution 7.4.4.b

```
1   WITH "transaction_types" AS
2   (
3    SELECT "sh_transaction",
4           CASE
5            WHEN COUNT(*) = 1
6                 THEN 'single'
7                 ELSE 'multi'
8           END AS "type"
9    FROM   "shopping"
10   GROUP  BY "sh_transaction"
11  ),
12  "transaction_with_types" AS
13  (
14   SELECT DISTINCT "shopping"."sh_transaction",
15                   "sh_category",
16                   "type"
17   FROM   "shopping"
18     JOIN "transaction_types"
19       ON "shopping"."sh_transaction" = "transaction_types"."sh_transaction"
20  ),
21  "transaction_pivoted"
22  AS
23  (
24   SELECT *
25   FROM   (SELECT "sh_category",
26                  "type" AS "linked",
27                  "type" AS "what_to_count"
28           FROM   "transaction_with_types") "tmp"
29           PIVOT  (COUNT("what_to_count")
30                   FOR "linked" IN ('multi', 'single')
31                  ) "pivot"
32  )
33  SELECT "sh_category" AS "category",
34         "'multi'"       AS "multi_count",
35           "'single'"      AS "single_count",
36           "'multi'" * 100 / ("'multi'" + "'single'") AS "multi_perc",
37           "'single'" * 100 / ("'multi'" + "'single'") AS "single_perc"
38  FROM "transaction_pivoted"
39  ORDER BY "sh_category"
```

This completes the solution of this problem.

Solution 7.4.4.c[573].

In this Problem we return to work with the “Library” database.

Solutions for all three DBMSes follow the same logic, so let’s consider it in detail on the example of MySQL, and for MS SQL Server and Oracle just give ready-made queries.

First, we need to prepare, in classical relational form, information about how many books were borrowed from the library in each month of each year. We get this information in a Common Table Expression (lines 1-9 of the query), and it looks like this.

year	month	books
2011	1	2
2012	5	1
2012	6	2
2014	8	3
2015	10	3

Now we use this information to produce a pivot table, just as we did in solution 7.4.4.a{574}. The number of months in a year is always fixed, so there is no need to dynamically form a set of pivot table columns here.

So, the solution for MySQL looks like this.

MySQL Solution 7.4.4.c

```
WITH `prepared_data` AS
(
 SELECT YEAR(`sb_start`) AS `year`,
        MONTH(`sb_start`) AS `month`,
        COUNT(`sb_id`) AS `books`
 FROM   `subscriptions`
 GROUP  BY `year`,
           `month`
)
 SELECT
  `year`,
  SUM(IF(`month` = '1', `books`, 0)) AS `1`,
  SUM(IF(`month` = '2', `books`, 0)) AS `2`,
  SUM(IF(`month` = '3', `books`, 0)) AS `3`,
  SUM(IF(`month` = '4', `books`, 0)) AS `4`,
  SUM(IF(`month` = '5', `books`, 0)) AS `5`,
  SUM(IF(`month` = '6', `books`, 0)) AS `6`,
  SUM(IF(`month` = '7', `books`, 0)) AS `7`,
  SUM(IF(`month` = '8', `books`, 0)) AS `8`,
  SUM(IF(`month` = '9', `books`, 0)) AS `9`,
  SUM(IF(`month` = '10', `books`, 0)) AS `10`,
  SUM(IF(`month` = '11', `books`, 0)) AS `11`,
  SUM(IF(`month` = '12', `books`, 0)) AS `12`
 FROM `prepared_data`
 GROUP BY `year`
 ORDER BY `year`
```

In MS SQL Server all the same actions are performed within this query.

MS SQL Solution 7.4.4.c

```
WITH [prepared_data] AS
(
 SELECT YEAR([sb_start]) AS [year],
        MONTH([sb_start]) AS [month],
        COUNT([sb_id]) AS [books]
 FROM   [subscriptions]
 GROUP  BY YEAR([sb_start]),
           MONTH([sb_start])
),
[pivoted_data] AS
(
 SELECT *
 FROM   (SELECT [year],
                [month] AS [linked],
                [books] AS [what_to_count]
         FROM   [prepared_data]) AS [tmp]
         PIVOT  (SUM([what_to_count])
                 FOR [linked] IN ([1], [2], [3], [4], [5], [6],
                                  [7], [8], [9], [10], [11], [12])
                ) AS [pivot]
)
SELECT [year],
       ISNULL([1], 0) AS [1],
         ISNULL([2], 0) AS [2],
         ISNULL([3], 0) AS [3],
         ISNULL([4], 0) AS [4],
         ISNULL([5], 0) AS [5],
         ISNULL([6], 0) AS [6],
         ISNULL([7], 0) AS [7],
         ISNULL([8], 0) AS [8],
         ISNULL([9], 0) AS [9],
         ISNULL([10], 0) AS [10],
         ISNULL([11], 0) AS [11],
         ISNULL([12], 0) AS [12]
  FROM [pivoted_data]
  ORDER BY [year] ASC
```

In Oracle all the same actions are performed within such a query.

Oracle | Solution 7.4.4.c

```
WITH "prepared_data" AS
(
 SELECT EXTRACT(YEAR FROM "sb_start") AS "year",
        EXTRACT(MONTH FROM "sb_start") AS "month",
        COUNT("sb_id") AS "books"
 FROM   "subscriptions"
 GROUP  BY EXTRACT(YEAR FROM "sb_start"),
           EXTRACT(MONTH FROM "sb_start")
),
"pivoted_data" AS
(
 SELECT *
 FROM   (SELECT "year",
                "month" AS "linked",
                "books" AS "what_to_count"
         FROM   "prepared_data") "tmp"
         PIVOT  (SUM("what_to_count")
                 FOR "linked" IN ('1', '2', '3', '4', '5', '6', '7', '8',
'9', '10', '11', '12')
                ) "pivot"
)
SELECT "year",
       NVL("'1'", 0) AS "1",
       NVL("'2'", 0) AS "2",
       NVL("'3'", 0) AS "3",
       NVL("'4'", 0) AS "4",
       NVL("'5'", 0) AS "5",
       NVL("'6'", 0) AS "6",
       NVL("'7'", 0) AS "7",
       NVL("'8'", 0) AS "8",
       NVL("'9'", 0) AS "9",
       NVL("'10'", 0) AS "10",
       NVL("'11'", 0) AS "11",
       NVL("'12'", 0) AS "12"
  FROM "pivoted_data"
  ORDER BY "year" ASC
```

This completes the solution of this problem.

Task 7.4.4.TSK.A: retrieve information about how many books were returned to the library each month (arranged vertically) of each year (arranged horizontally) as a pivot table.

Task 7.4.4.TSK.B: retrieve information about the number of books borrowed from the library in each month (vertically) of each year (horizontally) as a pivot table. (In fact, rewrite solution 7.4.4.c{581}, by swapping rows and columns.)

Task 7.4.4.TSK.C: rewrite solution 7.4.4.a{574} for all three DBMSes so that all operations on forming the dynamic part of the query, which is responsible for creating the list of columns, take place inside the stored routine.

7.4.5. Example 55: Suggest Your Example

It is impossible to cover all the details in any one book. But some of the examples presented above came about because of questions and suggestions from readers.

If you think there are more examples you would like to add to the book as we update and refine it, please contact dbb@svyatoslav.biz.

Chapter 8: Brief Comparison of MySQL, MS SQL Server, Oracle

This chapter is a cheat sheet on the main differences between the three DBMSes mentioned in this book. References to the corresponding examples and problems allow you to quickly jump to the desired part of the material and read the details.

Features of queries for data selection and modification.

Essence	MySQL	MS SQL Server	Oracle	Reference
`JOIN` conditions	We can use `ON` or `USING (the same name fields in joined tables)`	We can only use `ON`	We can use `ON` or `USING (the same name fields in joined tables)`	2.2.1.a{70}
Data types for `AVG` and other numeric operations	The resulting type is selected automatically, conversion is not necessary	Conversion is needed, otherwise results will be wrong (e.g., `AVG` from an integer field will also be an integer, i.e., the fractional part will be lost)	The resulting type is selected automatically, conversion is not necessary	2.1.5.a{33}
The default location of `NULL` values when ordering data	At the end	At the end	At the beginning; we can change it via `NULLS FIRST` and `NULLS LAST`	2.1.6.EXP.A{39}
Converting the string date representation to the date-time type	Automatic	Automatic	Explicit conversion is required	2.1.7.b{42}
Showing the first few rows of the result	`LIMIT x`	`TOP x` or `OFFSET y ROWS FETCH NEXT x ROWS ONLY`	Up to 12c we need a nested query with `ROW_NUMBER()`, versions f12c supports `OFFSET y ROWS FETCH NEXT x ROWS ONLY`	2.1.8.b{46}
		There is also support of `TOP ... WITH TIES`		2.2.7.b{121}
Ranking (window) functions	Supported since version 8	Supported	Supported	2.1.8.EXP.D{53}, 2.2.7.d{126}, 2.2.9.d{149}
Naming subqueries that are the source of tabular data (in the `FROM` and `JOIN` sections)	Mandatory	Mandatory	Mandatory, and we cannot write `AS` before the subquery name	2.1.8.EXP.D{53}, 2.1.8.c{48}, 2.1.8.d{50}

Features of queries for data selection and modification (continued).

Essence	MySQL	MS SQL Server	Oracle	Reference
Common Table Expressions	Supported since version 8	Supported	Supported	2.1.8.d{50}, 2.2.6.b{107}
Determining the difference in days between two dates	`DATEDIFF(greater, lower)`	`DATEDIFF(day, lower, greater)`	`(greater - lower)`	2.1.9.d{60}
Getting the current date	`CURDATE()`	`CONVERT(date, GETDATE()`	`TRUNC(SYSDATE)`	2.1.9.d{60}
Features of `GROUP BY`	We can specify fields in `SELECT` that are not aggregation functions and are not listed in `GROUP BY`. In `GROUP BY` we can refer to the name (alias) of the expression used in `SELECT`.	We cannot specify fields in `SELECT` that are not aggregation functions and are not listed in `GROUP BY`. `GROUP BY` cannot refer to the name (alias) of an expression used in `SELECT`.	We cannot specify fields in `SELECT` that are not aggregation functions and are not listed in `GROUP BY`. `GROUP BY` cannot refer to the name (alias) of an expression used in `SELECT`.	2.2.2.a{75}, 2.2.2.b{79}
Group concatenation	`GROUP_CONCAT()`	`STRING_AGG` or `STUFF (FOR XML…)`	`LISTAGG()`	2.2.5.c{104}
Selecting data from multiple tables with fields of the same name	Usually, it is not necessary to specify the table name, the DBMS automatically determines from where the data is to be retrieved	It is always necessary to specify the table name explicitly, if the `SELECT` includes two or more fields with the same name	It is always necessary to specify the table name explicitly, if the `SELECT` includes two or more fields with the same name	2.2.6.b{107}
"Protection against `NULL`"	`IFNULL(field, substitution value)`	`ISNULL(field, substitution value)`	`NVL(field, substitution value)`	2.2.7.e{131}
`TRUE` / `FALSE`	Yes, interpreted as 1 and 0	No explicit `TRUE` / `FALSE`	No explicit `TRUE` / `FALSE`	Example 20{159}
`JOIN` varieties	`JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `CROSS JOIN`, Not supported: `FULL OUTER JOIN` and `CROSS/OUTER APPLY`	`JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `CROSS JOIN`, `FULL OUTER JOIN`, `CROSS APPLY` and `OUTER APPLY`	`JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `CROSS JOIN`, `FULL OUTER JOIN`, supported since version 12c: `CROSS APPLY` and `OUTER APPLY`	2.2.7.e{131} 2.3.5.b{216} 6.1.2.a{434}
Checking the fact that the query returned a (non)empty result	`(NOT) EXISTS (query)`	`(NOT) EXISTS (query)`	`EXISTS (query)`; instead of `NOT EXISTS` we need a variable, `SELECT COUNT` into it, compare it to zero	2.3.5.c{218}
Inserting a value into an auto-incrementable primary key	Just pass the value, that's all	Enable the `IDENTITY_INSERT` for the table	Disable the trigger that implements auto incrementation	2.3.1.a{195}
Insert or update depending on the primary key value match	`REPLACE` and `ON DUPLICATE KEY UPDATE`	`MERGE`	`MERGE`	2.3.4.a{208}, 2.3.4.b{210}

Features of views, triggers, stored routines, transactions.

Essence	MySQL	MS SQL Server	Oracle	Reference
Creating views	We cannot use subqueries in the `FROM` section	We can use subqueries in the `FROM` section	We can use subqueries in the `FROM` section	3.1.1.a{223}
Caching (materialized, indexed) views	Not supported	Supported	Supported	Example 27{229}
Triggers	We can create any number of `BEFORE` and `AFTER` triggers on `INSERT/UPDATE/DELETE`	There are no `BEFORE` triggers, but we can create any number of `AFTER` and `INSTEAD OF` triggers on `INSERT/UPDATE/DELETE`	We can create any number of `BEFORE` and `AFTER` triggers on `INSERT/UPDATE/DELETE`, triggers of the `INSTEAD OF` type can only be created on views	3.1.2.b{246}
	Work only at row level (activated each time for each record of data to be processed)	Work only at the statement level (activated once for the entire expression)	There are both row-level triggers and statement-level triggers (with some limitations: no `[inserted]` and `[updated]` pseudo-tables)	4.1.1.a{286}, 4.1.1.b{295}
	We can't assign one trigger body to multiple operations	We can assign one trigger body to multiple operations	We can assign one trigger body to multiple operations	4.1.1.b{295}
	Not activated by cascade operations	Activated by cascade operations	Activated by cascade operations	4.1.1.a{286}, 4.1.2.a{306}
Triggers on views	Not supported	Only `INSTEAD OF`	Only `INSTEAD OF`	Example 30{271}
Implicit transactions and their autocommit	Yes	Yes	No	Example 41{423}
Levels of transaction isolation	`READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, `SERIALIZABLE`	`READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ`, `SNAPSHOT`, `SERIALIZABLE`	Levels: `READ COMMITTED`, `SERIALIZABLE`; modes: `READ ONLY`, `READ WRITE`	Example 44{449}

Features of views, triggers, stored routines, transactions (continued).

Essence	MySQL	MS SQL Server	Oracle	Reference
Default isolation level	`REPEATABLE READ`	`READ COMMITTED`	`READ COMMITTED`	6.2.1.a{443}
Output debugging information	`SELECT x`	`PRINT x`	`DBMS_OUTPUT.PUT_LINE(x)`	6.2.2.a{449}
Pause in script execution	`SELECT SLEEP(s)`	`WAITFOR DELAY 'hh:mm:ss'`	`EXEC DBMS_LOCK.SLEEP(s)`	6.2.1.a{443}
Modifying data in stored functions	Yes	No	Yes	5.1.1.c{382}

Task 8.TSK.A: supplement the table above with lists of indexes supported by each of the three DBMSes.

Task 8.TSK.B: create a table of such a kind, and write down the features of each DBMS, primarily those that you encounter and have to look for additional information.

Chapter 9: License and Distribution

This book is distributed under the “Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International[54]” license.

The text of the book is periodically updated and revised. If you would like to share this book, please share the link to the most up-to-date version available here: https://svyatoslav.biz/database_book/.

Source materials (schemas, scripts, etc.) are available here: https://svyatoslav.biz/database_book_download/src.zip

If you have any questions or find errors, typos, or other deficiencies in the book, please email at: dbb@svyatoslav.biz.

* * *

If you liked this book, check out two others written in the same style:

“Relational Databases by Examples”

All the key ideas of relational DBMS — from the concept of data to the logic of transactions, the fundamental theory and illustrative practice of database design: tables, keys, connections, normal forms, views, triggers, stored procedures, and much more by examples. The book will be useful to those who: have learned databases some time ago but have forgotten something; have not much practical experience, but wants to expand their knowledge; wants to start using relational databases in their work in a very short time.

Download: https://svyatoslav.biz/relational_databases_book/

“Software Testing. Base Course.”

The book is based on ten years of experience in conducting trainings for testers, which allowed to summarize the typical questions, problems, and difficulties for many beginners. This book will be useful both for those who are just starting out in software testing, and for experienced professionals — to systematize their existing knowledge and to organize training in their team.

Download: https://svyatoslav.biz/software_testing_book/

In addition to the text of this book, it is recommended that you take a free online course with a series of video lessons, quizzes, and self-study tasks.

The course is intended for about 100 academic hours, of which about half of your time should be spent on practical tasks.

With Russian voiceover: https://svyatoslav.biz/urls/sql_online_rus/
With English voiceover: https://svyatoslav.biz/urls/sql_online_eng/

[54] “Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International”. [https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode]

USING MYSQL, MS SQL SERVER, AND ORACLE BY EXAMPLES

2nd EDITION

About the author:

Svyatoslav Kulikov

EdTech specialist, EPAM Systems.
PhD, associate professor, Belarusian State University of Informatics and Radioelectronics.

Author of "Software Testing Introduction", "Automated Testing", and "PHP Web Development" enterprise trainings.

20+ years of experience in IT, 10+ years of experience in testers and web-developers mentoring.

Site: https://svyatoslav.biz

<epam> | TRAINING CENTER