Svyatoslav Kulikov

RELATIONAL DATABASES

BY EXAMPLES

© EPAM Systems, RD Dep.

Relational Databases
by Examples

Practical Guide
for Software Engineers and Testers

Book version 1.0.27 as of 2025-07-07

Table of Contents

FOREWORD **4**

CHAPTER 1: DATABASE FUNDAMENTALS **5**

1.1. DATA AND DATABASES **5**

1.2. DATABASE MODELING **7**

1.3. RELATIONAL DATA MODEL **17**

CHAPTER 2: RELATIONS, KEYS, RELATIONSHIPS, INDEXES **19**

2.1. RELATIONS **19**

2.1.1. General Information on the Relations 19
2.1.2. Creating and Using the Relations 25

2.2. KEYS **32**

2.2.1. General Information on the Keys 32
2.2.2. Creating and Using the Keys 46

2.3. RELATIONSHIPS **53**

2.3.1. General Information on the Relationships 53
2.3.2. Referential Integrity and Database Consistency 67
2.3.3. Creating and Using the Relationships 73

2.4. INDEXES **103**

2.4.1. General Information on the Indexes 103
2.4.2. Creating and Using the Indexes 135

CHAPTER 3: NORMALIZATION AND NORMAL FORMS **157**

3.1. DATA OPERATION ANOMALIES **157**

3.1.1. Theoretical Review of Data Operations Anomalies 157
3.1.2. Ways to Detect Data Operation Anomalies 162

3.2. GENERAL IDEAS ABOUT NORMALIZATION **170**

3.2.1. Dependency Theory 170
3.2.2. Normalization Requirements in the Database Design Context 207
3.2.3. Normalization Process from a Practical Point of View 218
3.2.4. Denormalization 228

3.3. NORMAL FORMS **235**

3.3.1. First Normal Form 236
3.3.2. Second Normal Form 241
3.3.3. Third Normal Form 247
3.3.4. Boyce-Codd Normal Form 253
3.3.5. Fourth Normal Form 258
3.3.6. Fifth Normal Form 262
3.3.7. Domain-Key Normal Form 267
3.3.8. Sixth Normal Form 271
3.3.9. Normal Forms Hierarchy and Non-Canonical Normal Forms 274

CHAPTER 4: DATABASE DESIGN **278**

4.1. DESIGN AT THE INFOLOGICAL (CONCEPTUAL) LEVEL **278**

4.1.1. Design Goals and Objectives at the Infological (Conceptual) Level 278
4.1.2. Design Tools and Techniques at the Infological (Conceptual) Level 280
4.1.3. Example of Design at the Infological (Conceptual) Level 287

4.2. DESIGN AT THE DATALOGICAL (LOGICAL) LEVEL 295

4.2.1. Design Goals and Objectives at the Datalogical (Logical) Level 295
4.2.2. Design Tools and Techniques at the Datalogical (Logical) Level 296
4.2.3. Example of Design at the Datalogical (Logical) Level 299

4.3. DESIGN AT THE PHYSICAL LEVEL 305

4.3.1. Design Goals and Objectives at the Physical Level 305
4.3.2. Design Tools and Techniques at the Physical Level 310
4.3.3. Example of Design at the Physical Level 312

4.4. REVERSE ENGINEERING 323

4.4.1. Purpose and Objectives of Reverse Engineering 323
4.4.2. Tools and Techniques of Reverse Engineering 324
4.4.3. Example of Reverse Engineering 327

CHAPTER 5: ADDITIONAL DATABASE OBJECTS AND PROCESSES 331

5.1. VIEWS 331

5.1.1. General Information on Views 331
5.1.2. Creating and Using Views 335

5.2. CHECKS 338

5.2.1. General Information on Checks 338
5.2.2. Creating and Using Checks 339

5.3. TRIGGERS 341

5.3.1. General Information on Triggers 341
5.3.2. Creating and Using Triggers 346

5.4. STORED ROUTINES 353

5.4.1. General Information on Stored Routines 353
5.4.2. Creating and Using Stored Routines 356

5.5. TRANSACTIONS 364

5.5.1. General Information on Transactions 364
5.5.2. Transactions Management 373

CHAPTER 6: DATABASE QUALITY ASSURANCE 376

6.1. DATABASE QUALITY ASSURANCE AT THE DESIGN STAGE 376

6.1.1. General Approaches to Database Quality Assurance at the Design Stage 376
6.1.2. Tools and Techniques of Database Quality Assurance at the Design Stage 379

6.2. DATABASE QUALITY ASSURANCE AT THE PRODUCTION STAGE 381

6.2.1. General Approaches to Database Quality Assurance at the Production Stage 381
6.2.2. Tools and Techniques of Database Quality Assurance at the Production Stage 384

6.3. ADDITIONAL DATA QUALITY AND DATABASE QUALITY ISSUES 388

6.3.1. Typical Misconceptions About Data 388
6.3.2. Typical Mistakes When Working with Databases and Ways to Eliminate Them 392

CHAPTER 7: APPENDIXES 395

7.1. DATABASE DESCRIPTION FOR INDIVIDUAL ASSIGNMENTS 395

7.2. DATABASE CODE FOR INDIVIDUAL ASSIGNMENTS 399

CHAPTER 8: LICENSE AND DISTRIBUTION 408

Foreword

Many thanks to colleagues in the EPAM Software Testing Division for their valuable comments and recommendations during the preparation of the material.

This book is devoted to a practical look at relational theory and relational database design. It does not cover fundamentals such as relational algebra and relational calculus, but with a variety of examples and explanations it shows the basic concepts and approaches needed to design databases.

This book will primarily be useful to those who:

- never studied databases;
- once studied databases, but have forgotten a lot;
- want to systematize their existing knowledge.

All database schemas in this book are given in UML 2.1 notation, created using Sparx Enterprise Architect and (if we're talking about design levels for which this is relevant) focused on MySQL 8.0, Microsoft SQL Server 2019, Oracle 18c. Most likely, the given solutions will work successfully on newer versions of these DBMS, **but not on older ones**.

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

The conventions used in this book are as follows:



Definitions and other important information to remember. Most of the definitions will be given, albeit in an adapted, but sufficiently rigorous form.

For ease of understanding, each definition will be given a simplified form (sometimes simplification will border on incorrectness, so you should still focus on the more rigorous formulation and use the simplified form only as a hint).



Additional information or reference to relevant sources. Anything that is useful to know. The original definitions will be footnoted.



Cautions and common mistakes. It is not enough to show you "the right thing to do", examples of what not to do are often very helpful.



Assignments for self-study. It is highly recommended that you do them (even if they seem very simple to you; and especially if the task looks complicated and incomprehensible).

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

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

Let's begin!

Chapter 1: Database Fundamentals

1.1. Data and Databases

Let's begin traditionally with definitions that will hardly tell you anything new but will allow us to avoid explaining the same thoughts over and over again in the future.



Data[1] — a versatile, interpretable representation of information in a formalized form suitable for transmission, communication, or processing.

Simplified: information organized according to certain rules.

There are two key points here.

First, the information must be presented in a formalized form (in other words, subject to certain rules). For illustration, here is an example of unformalized and formalized representation of information:

Unformalized representation	Formalized representation
"We have Smith and Jones in our department, and Smith's phone number has recently changed, from 123-44-55 to 999-87-32. And Taylor is going to work next month, but it will be hard to reach him, because he doesn't answer 333-55-66 very often, and more often another Jones answers there, but he's from another department."	**employee** **pass** | **name** | **department** | **phone** 178 | Smith | Dept-1 | *NULL* 223 | Smith | Dept-1 | 999-87-32 243 | Taylor | Dept-1 | 333-55-66 318 | Jones | Dept-2 | 333-55-66

Second, different interpretations allow us to perceive the same data differently in different contexts. For example, the "pass" field can be interpreted as:

- employee pass number;
- field with a unique value;
- number;
- natural primary key of the relation;
- clustered index{107};
- etc.

When the data becomes sufficiently large, it becomes necessary to organize it into more complex structures.



Database[2] — a collection of data organized according to a conceptual structure describing the characteristics of the data and the relationships among corresponding entities, supporting one or more application areas.

Simplified: a large amount of data, the relationship between which is built by special rules.

So, a database is a large set of data subject to additional rules. Such rules depend on the type of database, and the database management system (DBMS) is responsible for enforcing them.

[1] **Data** — reinterpretable representation of information in a formalized manner suitable for communication, interpretation, or processing (ISO/IEC 2382:2015, Information technology — Vocabulary).

[2] **Database** — collection of data organized according to a conceptual structure describing the characteristics of these data and the relationships among their corresponding entities, supporting one or more application areas (ISO/IEC 2382:2015, Information technology — Vocabulary).



Database management system[3], **DBMS** — a system, based on hardware and software, for defining, creating, manipulating, controlling, managing, and using databases.

Simplified: a software that manages databases.



It is worth noting an important fact for beginners (because very often one may hear a request to "show a DBMS"): the vast majority of DBMSes have no "human interface", are a service (a daemon in *nix systems) and interact with the outside world through special protocols (most often built over TCP/IP). Such well-known products as MySQL Workbench, Microsoft SQL Server Management Studio, Oracle SQL Developer and the like are **not DBMSes**, they are just client software that allows us to interact with the DBMS.

So, a DBMS is a special software designed to manage databases. Since in this book we will talk exclusively about relational databases and relational DBMSes, we can immediately note that interaction with such DBMSes is done by executing SQL queries.



Studying the SQL and its dialects is beyond the scope of this book, but if you are familiar with the general concepts, you can expand your knowledge by reading the material in the book[4] "Using MySQL, MS SQL Server and Oracle by Examples".



Task 1.1.a: to broaden your horizons, find information about how client applications interact with non-relational DBMSes, i.e., what "replaces" SQL queries in such DBMSes.



Task 1.1.b: based on the information presented in Appendix 1{395}, as well as on the handouts{4} design a logical model of the "Bank{395}" database for MySQL. If this task is too difficult at the moment, put it off for a while and continue studying the material in the book.



Task 1.1.c: based on the information presented in Appendix 2{399}, as well as on the handouts{4} design a physical model of the "Bank{395}" database for MySQL. If this task is too difficult at the moment, put it off for a while and continue studying the material in the book.

[3] **Database Management System, DBMS** — system, based on hardware and software, for defining, creating, manipulating, controlling, managing, and using databases (ISO/IEC 2382:2015, Information technology — Vocabulary).

[4] "Using MySQL, MS SQL Server and Oracle by Examples" (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

1.2. Database Modeling

Traditionally, this section in most books on databases is skimmed over without reading. I ask you to pay attention to it this time, because it will be as concise and to the point as possible. And it will help you better understand where certain ideas come from in the following sections.



Database model, data model[5] — a way to describe a database using a formalized (including graphical) language at some level of abstraction.

Simplified: a description of the database by which it will then be created (similar to the design of a building by which it will then be built).

Of particular interest in this definition is the mention of the level of abstraction. It turns out that the same database can have several models with different levels of details and purpose (e.g., a general description of data and relationships, a description of data structures, a description of how data are stored and processed, etc.).

From here we move on to the notion of levels (stages) of database modelling and design, i.e., the construction of several interrelated database models, each of which is designed to serve its specific purpose.



You can find many different (sometimes — contradictory) classifications of database modeling levels. This confusion is due to the fact that the author of each classification was based on a view relevant to its time and technology, and in its own way related to the general issues of modeling information systems. The following classification is no exception and is also subject to distortions associated with the author's point of view.

The only thing that inviolably unites all classifications is the idea of top-down{10} and bottom-up{10} design possibilities, and the fact that with each next, lower level of modelling, the model details increase and its binding to a specific type of DBMS, a specific manufacturer, a specific version, etc., becomes stronger.

It is also worth noting that in many cases it is difficult to clearly define the boundary between the levels, i.e., they flow smoothly into each other (at least at the time of the relevant preparatory actions in the design process).

Before we consider each level (see figure 1.2.a), it is necessary to give another very important clarification, without which all this information may be useless.

As a rule, engineering students who study databases and design them as part of laboratory, term and diploma papers, sincerely do not understand why we need any levels, if you can "just design a database".

You can. If the database is primitive (up to a few dozen tables), the subject area is limited to the level of complexity of university work, the initial requirements are simple and do not change. That is, if you design the same type and / or simple databases, all this "scientific nerd" is not needed. (Especially if you are both the customer and the contractor, and nothing prevents you from giving up and changing the requirements as soon as a fairly complex task arises).

But imagine that the subject area is completely unknown to you (for example, you are not designing another library database, but an information repository for a cardiac surgery research center).

Add to this the fact that the customer (in the face of the representatives of that research center) is not very knowledgeable in information technology, and explains a lot

[5] **Data model** — pattern of structuring data in a database according to the formal descriptions in its information system and according to the requirements of the database management system to be applied (ISO/IEC 2382:2015, Information technology — Vocabulary).

to you “in hand-waving terms”, and in the process it often turns out that something was forgotten, you have misunderstood, something has changed, something was not known to the customer, etc.

And in this situation, usually you can’t refuse to implement a particular requirement simply because it’s too complicated and/or you don’t really understand how to implement it.

Add to this the fact that the database may consist of hundreds (or even thousands) of tables, that it has rather strict requirements for reliability, performance and other quality indicators.

Keep in mind that you won’t be designing such a database by yourself, but as part of a team (and all participants will have to be sure to understand each other, coordinate their actions, substitute for each other).

For reasons of optimism, let’s stop this enumeration (although it could go on for several pages).

In such a situation, a structured, controlled, regimented approach to modelling and design is critical — you cannot create a workable database without it. No way.

Back to the levels of modelling.

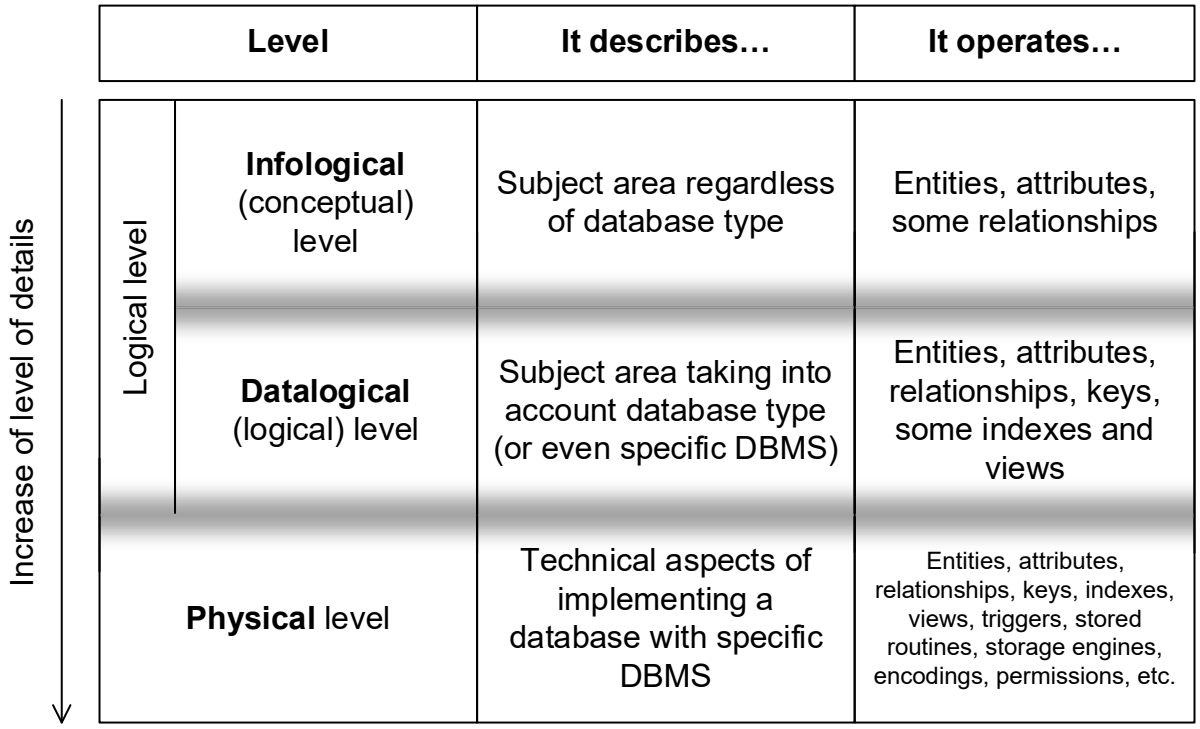


Figure 1.2.a — Generalized schema of database modelling levels



Infological (conceptual[6]) modelling level aims to create a conceptual model[7] that reflects the main entities of the domain (subject area), their attributes and relationships (perhaps not all of them yet) between the entities.

Simplified: a description of real-world objects and phenomena, the data about which will then be placed in a database.

[6] **Conceptual level** — level of consideration at which all aspects deal with the interpretation and manipulation of information describing a particular universe of discourse or entity world in an information system (ISO/IEC 2382:2015, Information technology — Vocabulary).

[7] **Conceptual model** — representation of the characteristics of a universe of discourse by means of entities and entity relationships (ISO/IEC 2382:2015, Information technology — Vocabulary).

If we extend figure 1.2.a “upwards”, to an even lower level of detail, from working directly with databases, we move on to business analysis, identifying common customer requirements and parameters of the subject area.



All the relevant material is described clearly and in detail in Karl E. Wiegers’ book “Software Requirements”.

Both the whole subject area and the data at this level can be described by various models (e.g., semantic, graph, entity-relationship). As a way to represent the model, UML or a simple verbal description is most often used (which is especially useful for discussing the model with a customer representative who is not an IT specialist).

The details of the implementation of this task will be discussed in the “Design at the Infological Level” chapter{278}.



Datalogical (often called just “logical”[8]) modelling level details the conceptual model, turning it into a logical schema[9], in which the previously identified entities, attributes and relationships are designed according to the rules of modeling for the chosen type of database (perhaps even taking into account specific DBMS).

Simplified: a description of real-world objects and phenomena according to the rules of the selected DBMS.

In many database design tools that support a division between only “logical” and “physical” design, it is this level that is called “logical”.

The usual tables{19}, fields{19}, keys{32}, relationships{53}, some indexes{103} and views{331} — everything that is the essence of the database — already appear here. Since the implementation of these objects depends on the kind of database (and even the specific DBMS), it is no longer possible to build a strictly abstract model here, and we have to take into account the typical features of the selected database type and DBMS.

UML or gradually losing popularity IDEF1X notation, Chen’s notation and the like will most often be used as a way to represent the model at this level.

The details of the implementation of this task will be discussed in the “Design at the Datalogical Level{295}” chapter.



It is worth noting that there are many logical models (document-oriented, fact-oriented, graph-theoretical, set-theoretical, object-oriented, based on inverted files, etc.). And each has spawned a different kind of database (sometimes several kinds).

This book focuses on relational databases, but let’s at least list the others (the main ones): cardboards, network databases, hierarchical databases, multidimensional databases, object-oriented databases, deductive databases, NoSQL databases, etc.

[8] **Logical level** — level of consideration at which all aspects deal with a database and its architecture, consistent with a conceptual schema and the corresponding information base but abstract from its physical implementation (ISO/IEC 2382:2015, Information technology — Vocabulary).

[9] **Logical schema** — part of the database schema that pertains to the logical level (ISO/IEC 2382:2015, Information technology — Vocabulary).



Physical[10] modelling level continues to detail and allows you to create the so-called physical schema[11], which takes into account the technical features of a particular database management system and its capabilities to organize and manage the structures of the developed database and the data in it.

Simplified: a description of the components of the database so that it can be used to automatically generate SQL code to create a database.

If we extend figure 1.2.a "down" to an even greater level of detail, from working directly with databases, we move to the field of system administration (configuring DBMS, operating system, network infrastructure, etc.)

At this level, the data model can be presented as at the previous (datalogical) level — most often in the form of UML, but just a graphical form is not enough, so we have to use SQL-scripts, verbal descriptions of the necessary changes and settings, fragments of configuration files, prepared cmd/bash-scripts, Windows registry files, etc.

The details of the implementation of this task will be discussed in the "Design at the Physical Level{305}" chapter.

It was mentioned earlier that any classification of modeling levels allows for both top-down and bottom-up design. Let's explain in more detail.



Top-down[12] design (in the context of databases often called "design from the subject area") involves moving from the highest level of modeling (infological, conceptual) down to the lowest (physical).

Simplified: we start communicating with the customer, and then we think about how to implement their requirements.



Bottom-up[13] design (in the context of databases often called "design from queries") involves moving from the lowest level of modeling (physical) up to the highest (infological, conceptual).

Simplified: we look at what and how we can implement, and then think about how to use it to meet the customer's requirements.

In practice, in order for the resulting database to satisfy all the key requirements (to be discussed very soon{10}), both design directions are applied. It follows that the design process is iterative, and it is impossible to "make and forget" a model at one level by switching completely to another.

If you try to make that mistake by choosing strictly one design direction (and without reviewing the path you have traveled to detect and correct errors), you are almost guaranteed to violate one of the following requirements for any database (and, by extension, its models):

- adequacy to the subject area;
- usability;
- performance;
- data safety.

[10] **Physical level** — level of consideration at which all aspects deal with the physical representation of data structures and with mapping them on corresponding storage organizations and their access operations in a data processing system (ISO/IEC 2382:2015, Information technology — Vocabulary).

[11] **Physical schema** — part of the database schema that pertains to the physical level (ISO/IEC 2382:2015, Information technology — Vocabulary).

[12] **Top-down** — pertaining to a method or procedure that starts at the highest level of abstraction and proceeds towards the lowest level (ISO/IEC 2382:2015, Information technology — Vocabulary).

[13] **Bottom-up** — pertaining to a method or procedure that starts at the lowest level of abstraction and proceeds towards the highest level (ISO/IEC 2382:2015, Information technology — Vocabulary).

There are no canonical definitions here, but the ideas themselves are extremely important, so we will consider them in detail and with examples.

The adequacy to the subject area is expressed in the fact that the database must allow us to perform all the necessary operations that are objectively needed in real life in the context of the work for which the database is intended.

This requirement is best met by modeling at the infological (conceptual) level, as this level is closest to the subject area and customer requirements, but to identify some errors you will inevitably have to analyze the models at other levels. And the essence of this requirement is most clearly reflected by examples of its violation.

First, let's consider a completely trivial example. Suppose for storing information about a person, a relation represented by the following schema (Figure 1.2.b) has been designed:

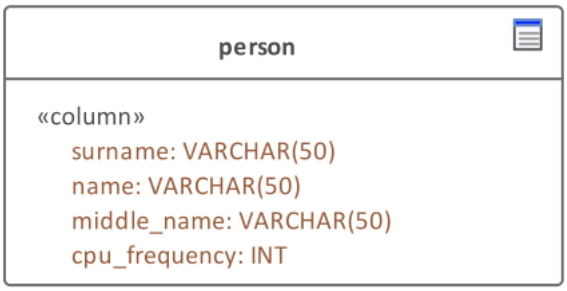


Figure 1.2.b — A trivial example of a violation of the adequacy to the subject area

It is immediately striking that `cpu_frequency` is not a human characteristic (at least for now). And therefore, it is completely unclear what to save in this field.

If you think about it a little bit, it also becomes apparent that there are many properties missing in this relation, because the characteristics of a person do not end with the last name, first name and middle name.

Let's consider a slightly closer to reality example of a design mistake (we won't even bother with a picture here). Let's say we are developing a database to automate the work of the university dean's office. The corresponding application has already been put into operation, some time has passed, and then our customers call us and ask: "And how do I mark that a student has gone on academic leave?"

And we suddenly realize that… there is no way. That we hadn't thought of such a situation, and neither the application (which the dean's office staff work with), nor our database has the ability to save the relevant data.

This situation is unpleasant (we'll have to redo a lot), but it pales in comparison to the next — the worst that can happen.

Let's consider a third (the deadliest) example, continuing the topic of automating the work of the dean's office. Suppose that information about the relationship between subjects, tutors, and students was stored in a database represented (simplified) by the following schema (figure 1.2.c):

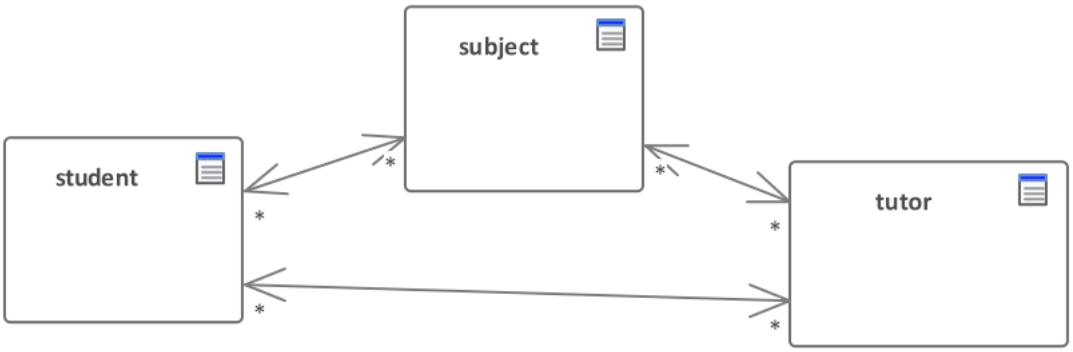


Figure 1.2.c — Simplified database schema that hides a mistake

All three relations are connected with “many to many{56}” relationships, for indeed:

- each student studies many subjects, and each subject is studied by many students;
- each subject can be taught by many tutors, and each tutor can teach many subjects;
- each student is taught by many tutors, and each tutor teaches many students.

Let’s assume that the developed product has been successfully put into operation, many years have passed, and then the customer calls and asks how to show (for a very important report!) a list of students that studied higher mathematics with professor Smith.

There’s no way to make such a report. This information was not stored in the database. There is only information that professor Smith (along with three dozen other colleagues) taught higher mathematics (and eight other subjects). There is information about which students were taught by professor Smith and which students studied higher mathematics. But there is no way to determine that this particular student studied higher mathematics (and not another subject) exactly with professor Smith (and not with any of his colleagues). (An alternative schema, which eliminates this problem, will be discussed below{163}.)

This example is called the deadliest, because there is nothing to fix. Even if we refine the application and the database, the information required by the customer will still be lost forever. But the customer was absolutely sure that it was there and available.

Basically, with a competent approach to the collection of requirements such mistakes (as in the examples about the dean’s office) can also be identified in the top-down design, but it is much easier to notice them in the bottom-up design, when we think about the specific tasks the user will solve and the SQL queries those tasks will lead to.

There is another form of violation of the adequacy to the subject area — violation of normalization, which leads to anomalies in data operations{157}. But this issue is so important and extensive that a separate chapter is devoted to it{157}.

Usability (in the context of database design) is not related to usability for the end user (who may not even know that there are any databases in the world). The term refers to the use of the database in the following situations:

- when interacting with applications (the emphasis here is mainly on simplicity and ease of writing error-free queries);
- in the process of its further development (and here we talk about quality indicators such as maintainability and supportability).

Thus, it is a question of “usability (convenience) for the software engineer”. Why is this important? The better the database meets this requirement, the easier and faster software engineers can organize interaction with it, allowing a minimum of errors and with less cost solving the issues of performance, security, etc.

A nice example to illustrate the difference in usability of a database depending on its model is determining the number of the day and number of the week in a month based on the specified date (see a detailed description of this problem and its solution in a corresponding book[14]).

While the date is stored as a single value, you have to use a “whole-screen” query. And also, each database has its own “magic” in the form of numerical constants, poorly documented functions and the like — all that is extremely dangerous in terms of the possibility of making a mistake.

But you can change the database model. You can store the values you’re looking for in the table itself (and compute them with triggers{341}) or create a view{331} (and “disguise” the corresponding query in it). Then, for the programmer organizing the interaction between the application and the database, the task is reduced to a trivial SELECT query (just a couple of lines long). The idea is shown schematically in figure 1.2.d.

MS SQL — Before the model modification

```
WITH [iso_week_data] AS
 (SELECT CASE
           WHEN DATEPART(iso_week,
                 CONVERT(VARCHAR(6), [sb_start], 112) + '01') >
                DATEPART(dy,
                 CONVERT(VARCHAR(6), [sb_start], 112) + '01')
           THEN 0
           ELSE DATEPART(iso_week,
                 CONVERT(VARCHAR(6), [sb_start], 112) + '01')
         END AS [real_iso_week_of_month_start],
         CASE
           WHEN DATEPART(iso_week, [sb_start]) >
                DATEPART(dy, [sb_start])
           THEN 0
           ELSE DATEPART(iso_week, [sb_start])
         END AS [real_iso_week_of_this_date],
         [sb_start], [sb_id]
  FROM [subscriptions])
 SELECT [sb_id], [sb_start],
        CASE
          WHEN DATEPART(dw,
               DATEADD(day, -1, CONVERT(VARCHAR(6), [sb_start], 112)
               + '01')) <= DATEPART(dw, DATEADD(day, -1, [sb_start]))
          THEN [real_iso_week_of_this_date] -
               [real_iso_week_of_month_start] + 1
          ELSE [real_iso_week_of_this_date] -
               [real_iso_week_of_month_start]
        END AS [W],
        DATEPART(dw, DATEADD(day, -1, [sb_start])) AS [D]
FROM [iso_week_data]
```

MS SQL — After the model modification

```
SELECT [sb_id],
       [sb_start],
       [sb_start_week]
       [sb_start_day]
FROM   [subscriptions]
```

Figure 1.2.d — Changing the query after database model modification

Let’s look at another very simple and illustrative example of a usability violation. Suppose that in some database the names and initials of people are stored as follows (see figure 1.2.e).

person

p_id	p_name	...
1	A.Jr. Smith	...
2	JJ Taylor	...
3	N Black	...
4	...	...

Figure 1.2.e — Non-optimal data storage

[14] “Using MySQL, MS SQL Server and Oracle by Examples” (Svyatoslav Kulikov), task 7.4.1.a [https://svyatoslav.biz/database_book/]

Here the last name and initials are stored in one column, with the initials going before the last name (and for clarity, presented in several variants, which may well occur in real life). Now, how do you organize a list of people alphabetically by last name?

Theoretically, you could write a function that takes into account many variants of initials (e.g., “A.A.”, “AA”, “A. A.”, “A .”, etc., etc.), removes this information, and returns only the last name. And use this function in queries of the following kind:

MySQL — Arranging the list of people alphabetically by last name with initials removed

```
1      SELECT *
2        FROM `person`
3    ORDER BY REMOVE_INITIALS(`p_name`) ASC
```

Yet such a function would be complicated, slow, might not take into account some variants of initials, and would also make it impossible to use indexes{103} to speed up the query (or require creating a separate index over the results of the function’s calculation).

But just put surnames and initials in separate columns (see figure 1.2.f), and the situation is instantly simplified.

person

p_id	p_surname	p_initials	...
1	Smith	A.Jr.	...
2	Taylor	JJ	...
3	Black	N	...
4	...	...	...

Figure 1.2.f — Optimal data storage

Now there is no need to pre-process the information in any way, and the value of the last name can be used directly, which eliminates all the disadvantages of the solution with the function usage. And the query is simplified to the following form:

MySQL — Arranging the list of people alphabetically by last name

```
1      SELECT *
2        FROM `person`
3    ORDER BY `p_surname` ASC
```

Such situations may occur more often than they seem. And considering them can significantly improve the usability of the database.

Performance can easily be considered one of the most painful issues in working with databases, and the solution to the corresponding problems can be described in several separate books. What is worth doing at any stage of the design process is, at a minimum, to keep performance in mind and avoid creating models where it is compromised in the first place.

For example, we know that an application will very often need to figure out the number of entries in different tables (both total and the number of some specific entries) in order to work. In general, such operations will be very slow (see the detailed description of this situation in the corresponding book[15]), but by using caching tables{230} and materialized views{331} it is possible to reduce the query execution time to almost zero, at the price of a small loss of performance of other operations in this particular case.

Usually, usability problems{12} lead to more performance problems, since complex queries are harder to optimize.

In a general case, it needs a comprehensive analysis to create a database model that is guaranteed to provide high performance — too many factors play a role here, some of which are generally beyond the scope of database design (e.g., the structure of the disk storage).

But once again, the most important thing is to avoid at least the most obvious silly mistakes, such as (we exaggerate deliberately) creating a 2GB artificial primary key in a table storing the names of days of week.

Data safety should be understood not only in the context of access restrictions, but also in the sense that no accidental, unforeseen changes should happen to the data. To achieve this goal, it is sometimes necessary to think through a huge number of constraints implemented through specific database objects and even through a separate interface in the form of a set of stored procedures{353}.

As an example, let's mention the classic — storing a tree in a relational database (see the schema in figure 1.2.g).

At first glance, everything looks nice and logical, but in order to ensure that all operations (adding a node to an arbitrary place, deleting a node, moving and/or copying a subtree, etc.) and following all the rules (strictly one root node, no orphan nodes, no cycles, etc.) are performed correctly, you have to create dozens of additional database objects and prohibit direct changes of data in the corresponding table. Without all these measures, the tree structure may be broken at any time.

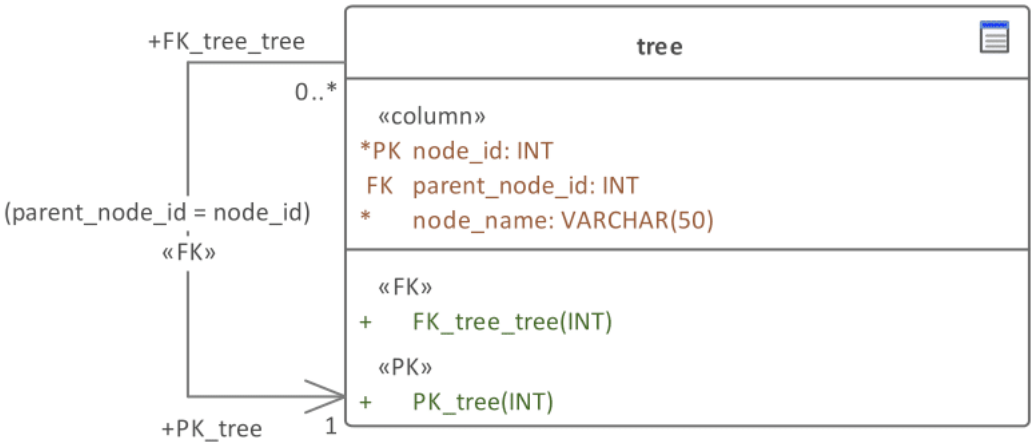


Figure 1.2.g — Classic schema for storing a tree

Here again we mention data operation anomalies{157} as another good example of data safety problems.

[15] "Using MySQL, MS SQL Server and Oracle by Examples" (Svyatoslav Kulikov), task 2.1.3.a [https://svyatoslav.biz/database_book/]

At the end of this chapter, we emphasize again that in order to ensure that the database meets all of the requirements discussed above, a careful modeling process is necessary, which should not rely on luck. And the results of such modeling must be carefully documented.



Task 1.2.a: for a better understanding of the UML diagrams in this book, get acquainted with at least the basic principles of UML. A lot of good quick references can be found here:
http://modeling-languages.com/best-uml-cheatsheets-and-reference-guides/



Task 1.2.b: give some examples of database schemas that would violate key database requirements{10}.



Task 1.2.c: what defects do you see in the infological (conceptual) and datalogical (logical) schemas of the “Bank{395}” database? Make appropriate changes to these schemas.

1.3. Relational Data Model

This chapter is probably the most theoretical in the whole book. But here we will consider the most important properties of the relational model, that for more than half a century (the model was proposed in 1969) have allowed it to take the lead in many indicators.



Relational model[16] — a mathematical theory describing data structures, data integrity control logic and data management rules.

Simplified: a model for describing relational databases.

At the time of its invention, this particular model provided a complete set of solutions for database description, data integrity control, and data management — in contrast to many other alternative approaches that allowed only part of the tasks to be implemented. Now it seems strange that there were some other “incomplete solutions”, but let’s remember: this was in the days when a computer occupied several rooms and cost as much as half the building it was in.

In the context of describing data structures, the relational model introduces the notion of a normalized{157} relation{19}. Moreover, a normalized relation is the only basic structure that this model operates on.

In the context of data integrity, this refers to the integrity of entities (tables, relations{19}) and foreign keys{32}.

Finally, in the context of data management rules, we are talking about relational algebra and relational calculus. For all the beauty of these phenomena, their consideration, alas, is beyond the scope of this book.

Why is the relational model so convenient? Because of the following advantages:

- it is logical (rather than physical, because the physical implementation is given to the DBMS developers), which allows you to design a database at a sufficiently high level of abstraction, without reference to the specific physical implementation of tables, relationships and other objects (and as a result, for example, choose a specific DBMS after the construction of the database schema);
- it is based on a rigorous mathematical apparatus, which allows not only to determine the applicability, the basic possibility and correctness of certain operations, but also to evaluate their complexity (and even serve as a basis for calculating performance indicators);
- it describes both the declarative approach (the software developer specifies the final goal of the solution without describing the sequence of actions) and the imperative (procedural) approach (familiar to everyone from “classical programming”, i.e., a description of the sequence of steps to achieve the goal).



Declarative approach (in the form of SQL) is implemented almost identically in the vast majority of relational DBMSes, while imperative (procedural) approach (with triggers, stored procedures and functions) has so many features that we can safely talk about the complete incompatibility of appropriate solutions in different DBMSes.

At the everyday level, the main advantage of the relational model is its ubiquity: there are many well-proven DBMSes, many ready-made proven solutions, many qualified specialists, a lot of literature, etc.

[16] **Relational model** — data model whose structure is based on a set of relations (ISO/IEC 2382:2015, Information technology — Vocabulary).

Speaking of disadvantages of the relational model, it is worth noting that:

- it concedes to other models in optimality of description and use of some data structures (e.g., implementation of trees or graphs in a hierarchical or graphical model turns out to be by orders of magnitude easier);
- it is not relieved of the problem, also typical of other models, of the high complexity (for humans) of developing large databases (although this is partially compensated for by the capabilities of design tools);
- databases built on the basis of the relational model when performing many operations are very demanding on the amount of memory and computing resources (it is this drawback was one of the prerequisites for the appearance of NoSQL-solutions).

Yet even now, the relational model has no alternative that combines all of its advantages and versatility and is free of its disadvantages. Therefore, for the vast majority of cases in which you need a database to store and process data, we are automatically talking about relational databases.

This concludes the general theory, and next we will discuss the fundamental concepts of relational databases with as many practical examples as possible.



However, if you are still interested in the science part, check “An Introduction to Database Systems, 8th edition” book by C.J. Date.



Task 1.3.a: name five other advantages and disadvantages of the relational model (look for relevant information in the book mentioned above).



Task 1.3.b: is it possible to represent the “Bank{395}” database schema in a form other than relational? Try to represent the alternative solution graphically in any form you like.



Task 1.3.c: why do you think the relational model was used to describe the “Bank{395}” database? What are the advantages of this solution compared to the available alternatives?

Chapter 2: Relations, Keys, Relationships, Indexes

2.1. Relations

2.1.1. General Information on the Relations

Previously{17} we noted that a normalized{157} relation is the only basic structure operated by the relational data model. However, such terms as “data type”, “data domain”, “attribute”, “tuple”, “primary key{36}” are inextricably linked to the concept of “relation”, most of which we will consider sequentially in this chapter. And let’s start with the basic term.



Relation[17] — a set of entities with the same set of attributes. In the context of relational databases, a relation consists of a header (schema) and a body (a set of tuples).

Simplified: a mathematical model of a database table.



It is important to distinguish between “relation” and “relationship,” even though they are often confused in everyday speech. Technically speaking, the term (for example) “the many to many relation” is incorrect; it should be “the many to many relationship”.

Given the note just made, another typical terminological problem needs to be clarified. The difficulty is that the concept of “entity” is extremely vague and is often defined as “everything that really exists in the world”. Let’s aggravate the situation and restate the concept of relation, so: “a relation is an entity that is a set of entities with a certain set of attributes.” This is already getting weird.

Now it is worth noting that an attribute is also an entity. Bottom line: “a relation is an entity that is a set of entities with a certain set of entities”. (You can continue the transformation further and see what the result is).

Let’s stop this madness. In the context of databases, all we’ll care about is that infological (conceptual) modeling tends to talk about “entities and relations”, while datalogical (logical) and physical modeling talk about “relations and relationships” and “tables and relationships”.

Now again, very briefly and simplistically: “entity”, “relation”, “table” are synonyms (as long as you don’t specify the exact context in which you use these terms). Then, to make it easier to remember, you can formulate a simpler definition of a relation.



Relation — a set of tuples (entries, table rows) that have the same set of attributes (properties, fields, table columns).

From here, it is logical to move on to the components of the relation. And we’ll start with the most fundamental, the data type.

[17] **Relation** — set of entity occurrences that have the same attributes, together with these attributes (ISO/IEC 2382:2015, Information technology — Vocabulary).



Data type[18] — a set of data objects of a certain structure and a set of allowed operations, in any of which such objects can act as operands.

Simplified, by example: numbers, strings, dates, etc.

The concept of "data type" here is generally equivalent to that in programming languages (for example, "strings" or "numbers" are treated almost identically in databases and programming languages, and to understand data types such as "datetime" or "geometry" we can imagine that we are dealing with instances of the corresponding classes).

All values handled by the database are typed (i.e., their type is known at any time). For most data types, there are conversion operations (e.g., the string "12.34" may be converted into a fractional number 12.34).

Additional restrictions can be imposed on data types, which leads to the concept of "attribute domain".



Attribute domain[19] — a set of all possible values of a relation attribute.

Simplified, by example: phone number, last name, street name, etc.

The essence of this concept is easiest to explain with an example. Suppose there is a "Product code" attribute{20} in some relation, which is represented by a string. And we also know that all product codes are formed according to the rule "three letters of the English alphabet in upper case, four digits, two letters of the English alphabet in lower case" (for example, "ABC1234yz"). Obviously, the string "Monday" is not a product code, i.e., is not included in the corresponding domain, although it belongs to the same type of data (to strings).

The concept of domain is important due to the fact that the data are considered comparable only if they belong to the same domain, that is, it makes no sense to compare the product code and the name of the day of the week (although technically any DBMS allow you to perform such an operation).

In the vast majority of DBMSes triggers{341} or checks{338} have to be used to control whether the data belongs to the same domain. Despite some complexity of such control and the fact that its implementation reduces the performance of some data operations, it can significantly improve data safety{15} by reducing the risk of transferring (and storing) incorrect values.

For the remaining two concepts, let's give definitions and go straight to the graphic explanation.



Attribute[20] — a named property of an entity (relation).

Simplified: a column of a table.



Tuple[21] — a part of a relation, which is a unique interrelated combination of values, each of which corresponds to a different attribute.

Simplified: a row (entry) of a table.

At a common level, an attribute can be understood as a "table column" and a tuple as a "table row".

[18] **Data type** — defined set of data objects of a specified data structure and a set of permissible operations, such that these data objects act as operands in the execution of any one of these operations (ISO/IEC 2382:2015, Information technology — Vocabulary).

[19] **Attribute domain** — set of all possible attribute values (ISO/IEC 2382:2015, Information technology — Vocabulary).

[20] **Attribute** — named property of an entity (ISO/IEC 2382:2015, Information technology — Vocabulary).

[21] **Tuple** — in a relational database, part of a relation that uniquely describes an entity occurrence and its attributes (ISO/IEC 2382:2015, Information technology — Vocabulary).

When we next talk about normalizing relationships, the terms “key attribute” and “nonkey attribute” will come up very often. Let’s consider them.



Key attribute[22] (prime attribute[23]) — an attribute of a relation, which is a part of at least one candidate{34} key of this relation.

Simplified: a column of a table, which is a part of the candidate key of that table.



Nonkey attribute[24] — an attribute of a relation, which is not part of any of the candidate{34} keys of this relation.

Simplified: a column of a table that is not part of any of the candidate keys of that table.



Primary key attribute[25] — an attribute of a relation, which is part of the primary key{36} of that relation.

Simplified: a column of a table, which is part of the primary key of that table.



In some books the term “key attribute” can be understood both as an attribute included in any candidate{34} key of the table and an attribute included in the primary{36} key of the table. Correspondingly, the term “nonkey attribute” may be understood to mean both an attribute that is not part of any candidate key and an attribute that is not part of a primary key.

Further in this book, the term “key attribute” will be used in the meaning designated above{21}.

[22] **Key attribute** — an attribute of a given relvar that’s part of at least one key of that relvar. (“The New Relational Database Dictionary”, C.J. Date)

[23] **Prime attribute** — old fashioned and somewhat deprecated term for a key attribute (not necessarily a primary key attribute). (“The New Relational Database Dictionary”, C.J. Date)

[24] **Nonkey attribute** — an attribute of a given relvar that isn’t part of any key of that relvar. (“The New Relational Database Dictionary”, C.J. Date)

[25] **Primary key attribute** — an attribute of a given relvar that participates in the primary key (if any) for that relvar. (“The New Relational Database Dictionary”, C.J. Date)

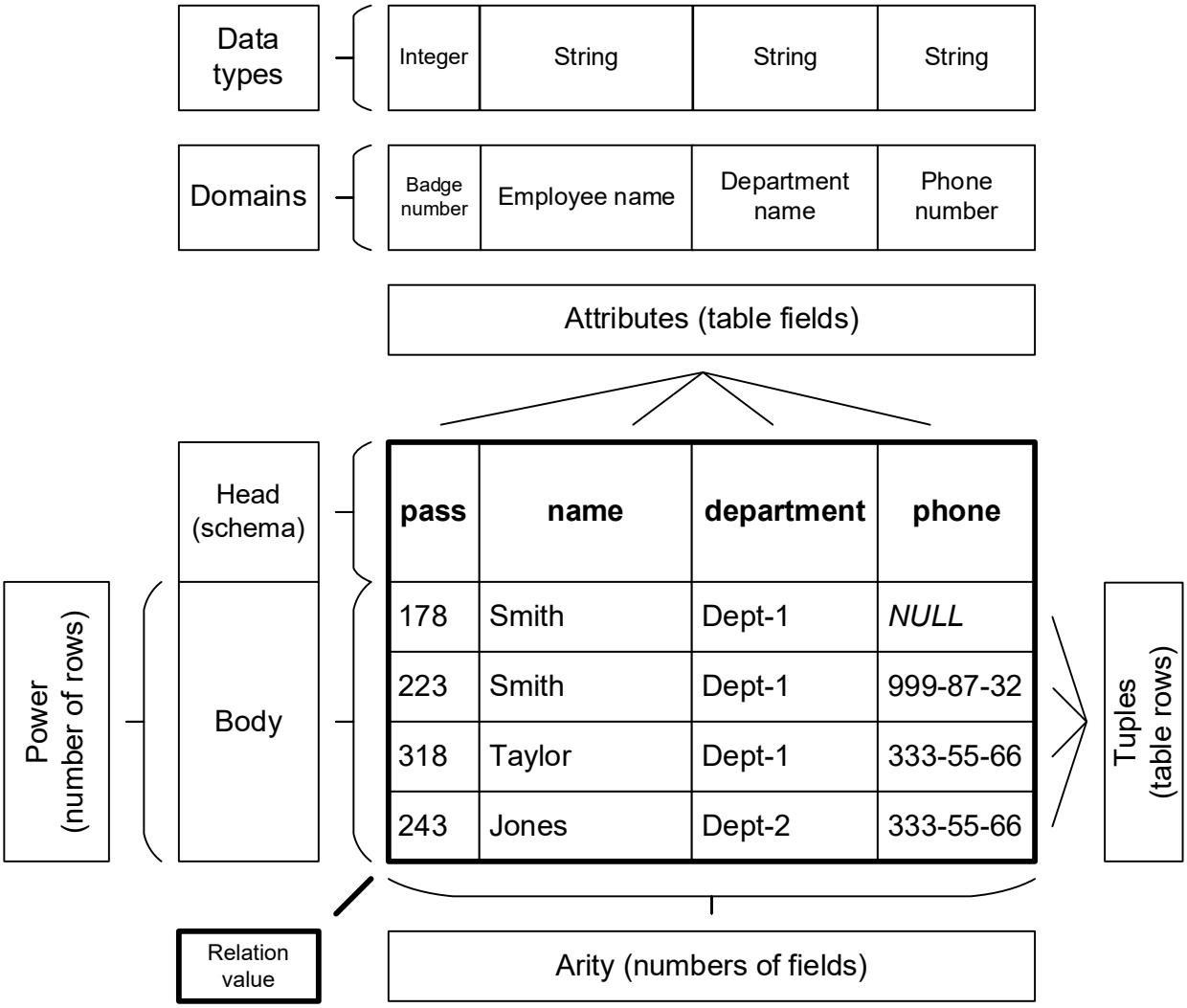


Figure 2.1.a — Relation and its components



It is important to remember that the terms “table”, “column”, “row” are only simplified and more convenient in everyday speech analogues of the corresponding concepts of relational theory. But if one approaches the question strictly mathematically, such a simplification is wrong.



“An Introduction to Database Systems, 8th edition” book by C. J. Date devotes a total of over three hundred pages to relations and related concepts. If you are just beginning to learn about databases, this material may not be of much use to you, but if you have at least 2-3 years of experience with databases, it is highly recommended that you read this comprehensive work.

To conclude this chapter, let’s emphasize an important idea that is often overlooked: the difference between “relation”, “relation schema”, “relation variable”, and “relation value”.

As one can easily guess, the first term (“relation”) is commonly used instead of the other three (although it is a synonym for “relation value”). Even more often we just say “table” (which is quite true in the context of the SQL).

So, what’s the difference? (And let’s add another term.)

- Relation schema — a description of the attributes of the relation (specifying their names, data types and other properties).
- Relation variable — a table actually created in the DBMS.

- Relation value (or just “relation”) — the data that is currently stored in the relation variable.
- Derived relation variable (view{331}) — an additional form of extracting data from the relation.

Obviously, the relation variable has many values (changing at least one bit of the data already gives a new value).

Let’s explain all the terms graphically (Figure 2.1.b), indicating the sequence of implementation: based on the relation model, a relation schema is built, which can be used to form an SQL query to create a relation variable (and then a view{331}), and then the relation variable can be filled with data, which will give us the relation value and the view value.

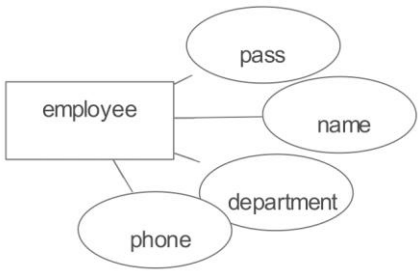
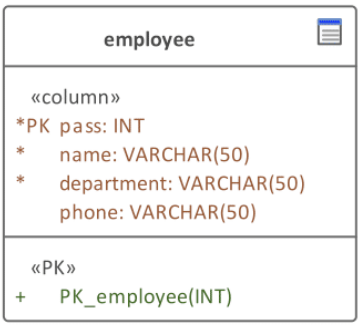
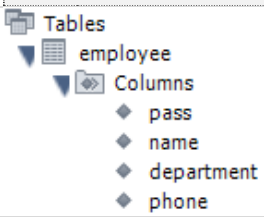
1) In the process of creating a data model, entities and their attributes are described:	
2) The model is detailed to the relation schema:	
3) Based on the existing relation schema, SQL code is generated to create a relation variable (i.e., a real table in a real database under the control of the DBMS):	MySQL Table creation `CREATE TABLE employee ( pass INT(11) NOT NULL, name VARCHAR(50) NOT NULL, department VARCHAR(50) NOT NULL, phone VARCHAR(50) DEFAULT NULL, PRIMARY KEY (pass) ) ENGINE=InnoDB DEFAULT CHARSET=utf8;`
4) A real table appears in the DBMS:	
5) Now we can build a view from the table, i.e., a derived relation variable:	MySQL View creation `CREATE VIEW pass_and_name AS SELECT pass, UPPER(name) AS name FROM employee`

Figure 2.1.b — All terms related to the concept of “relation”

6) An example of relation value looks like this:

pass	name	department	Phone
178	Smith	Dept-1	*NULL*
223	Smith	Dept-1	999-87-32
243	Taylor	Dept-1	333-55-66
318	Jones	Dept-2	333-55-66

7) An example of the view value (the derived relation variable) looks like this:

pass	name
178	SMITH
223	SMITH
318	TAYLOR
243	JONES

Figure 2.1.b (continued) — All terms related to the concept of “relation”

In order not to confuse the terminology, you can use the following cheat sheet[26] (this division is completely arbitrary, but it is well remembered):

Point of view	Mutual correspondence of terms		
	Group 1	**Group 2**	**Group 3**
Subject area	Entity	Instance	Property
Relational theory	Relation	Tuple	Attribute
Physical database	Table	String	Column
Physical storage	File	Record	Field

This concludes the theoretical part, and now we will look at how things work in reality.



Task 2.1.a: give 15–20 examples of relations describing certain real-world objects. Pay special attention to the data types you choose for each attribute.



Task 2.1.b: what relations do you think are missing in the “Bank{395}” database? Complete its schema with the missing relations.



Task 2.1.c: what relations attributes do you think are missing in the “Bank{395}” database? Complete the corresponding relations with the missing attributes.

[26] Unfortunately, it has not been possible to establish the source of this table, which in no way detracts from the merits of its author and the benefits of its use.

2.1.2. Creating and Using the Relations

Let's begin immediately with a not very technological, yet critically important remark.



Always use **only English** for naming database objects. If you do not speak it, write in "transliteration", but still use only the English alphabet.

Despite the fact that the vast majority of database design tools and DBMSes support the possibility of naming objects in many other languages, there can always be a situation in which the entire system will stop working because of encoding problems in one of the many software tools.

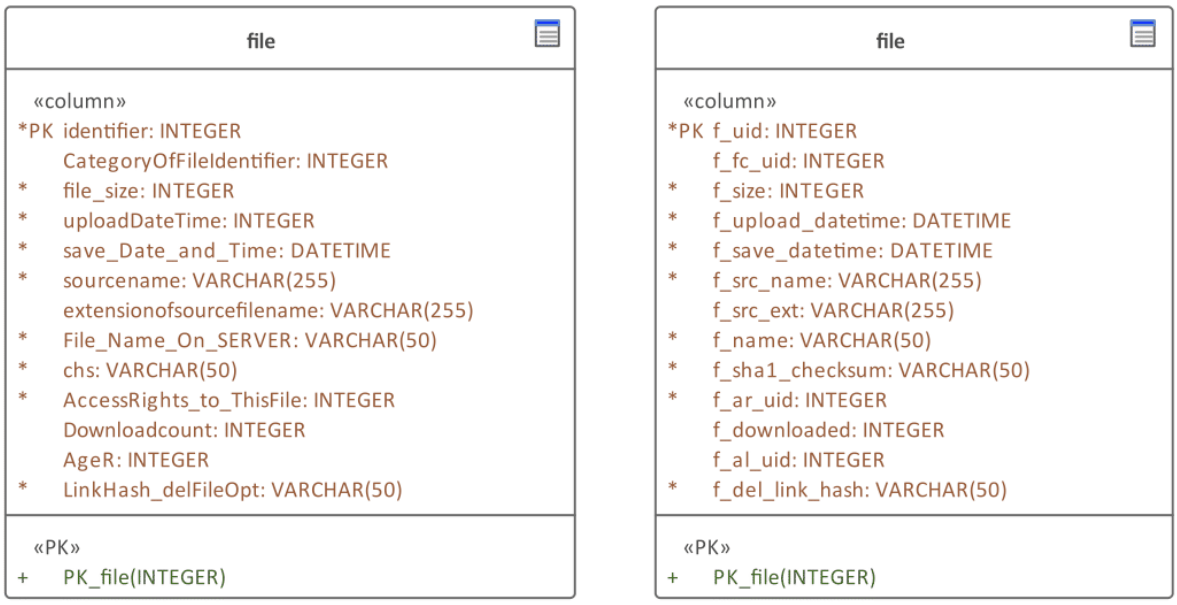
In addition, the use of English in the IT is an immutable generally accepted international tradition.

In this chapter, we will deliberately not consider modeling relations at the conceptual level (the corresponding section{278} is devoted to this matter), but will talk about the technical implementation of the logical and physical levels.

Specific actions on these levels will depend on the capabilities of your design tool, but in Sparx Enterprise Architect (where, in the context of database modeling, the capabilities of the logical level are almost identical to those of the physical level) on the logical level it is worth doing the following.

Develop a standard for naming database objects. To avoid useless arguments about which standard is better, let's emphasize only the main idea: it is necessary to follow some (even if created by you personally) one standard, and only that one.

Compare two relation schemas describing a file in some file-sharing service (figure 2.1.c):



Bad example Good example

Figure 2.1.c — Example of relation schemas without and with compliance with the naming standard

On the scale of the entire database schema, the problem of noncompliance with naming standards turns into a disaster: in such a "jumble" it is very difficult to navigate, very easy to miss a mistake, and for professionals accustomed to compliance with the standards, such a spectacle is almost physically painful.

Create relations and fill them with attributes. All necessary relations and all (at least obvious, “informational”) attributes are already known by this moment (obtained at the previous, conceptual level of modeling). So, you can safely transfer this information into your schema. An example of what the resulting relation will look like can be seen in figure 2.1.c.

Specify the most obvious constraints. At this level of design, you can afford not to dive into the details of the technical implementation of database tables and take into account only those features that clearly visible from the subject area (e.g., the requirement of uniqueness of some attribute values or the requirement of an explicit attribute value (`NOT NULL` property)). In figure 2.1.b attributes with a “*” on the left side of their name have the `NOT NULL` property enabled.

Specify the most obvious keys{32}**, relationships**{53}**, indexes**{103}**.** The same logic that was mentioned in the previous paragraph continues here: we mark only the most important things — those without which the data schema will obviously start to lose the adequacy of the subject area{11}. So, for example, in figure 2.1.b the `f_uid` attribute has the `PK` property, i.e., it is the primary key of the relation.

And that’s it with the (data)logical level. The difficulty here is not in the technical implementation (this is really a routine operation, requiring almost no mental effort), but in fully reflecting the data model of the subject area to the database schema, while respecting all the necessary requirements{10}.

On the physical level, however, the situation is more complicated. First, we still have to keep in mind the subject area. Secondly, we must have a very good understanding of the internal logic of the chosen DBMS, because here we will be making many technical decisions.



Very often one can hear the question about whether it is possible to automatically convert the (data)logical model into a physical one. To our great regret, yes, it is possible. And to take advantage of this opportunity is to make a huge mistake.

The (data)logical model is close to the infological (conceptual) model and primarily meets the requirement of adequacy to the subject area{11}. The physical model (without violating the adequacy requirements) must, in particular, provide usability{12}, performance{14}, and data safety{15}.

Obviously, automatic conversion will create a “technical copy” of the (data)logical model, nothing more. And only thoughtful “manual” creation of the physical model allows one to get a truly high-quality database.

So, what needs to be done with the relations at the physical level.

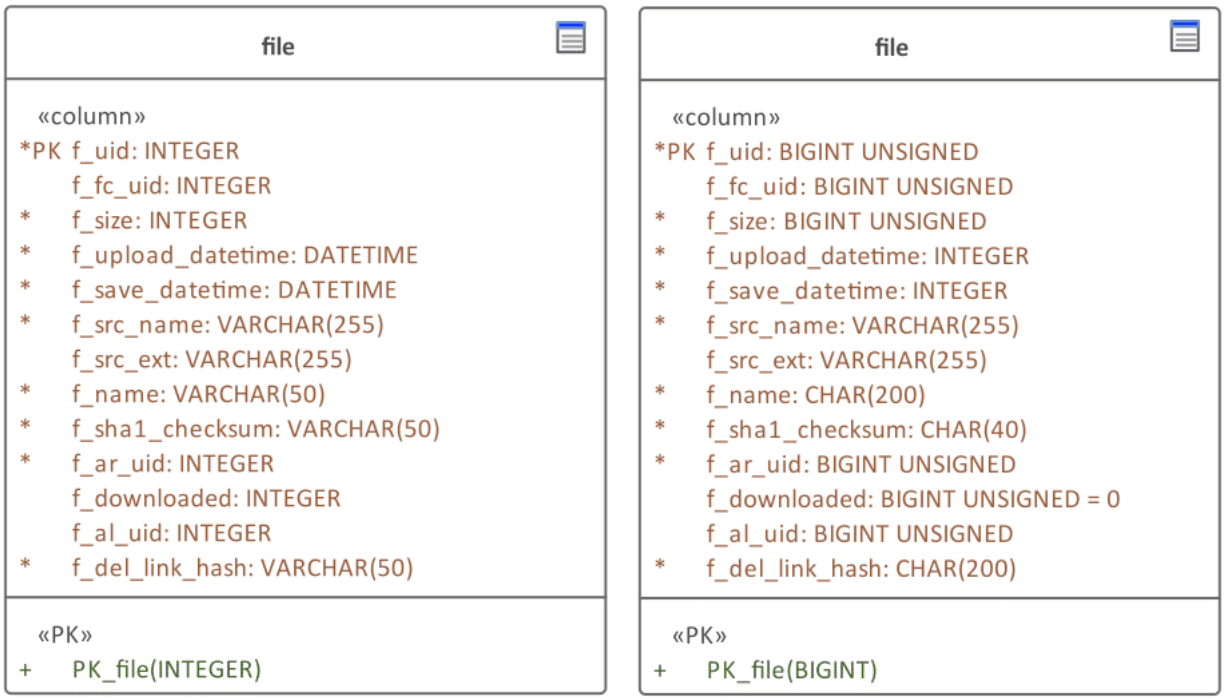
Specify data types. It is no longer enough to understand in general terms that “here is a string, and here is a number”. Strings, numbers, datetime and monetary values (and many others) are expressed in quite specific (and several different!) data types supported by a particular DBMS.

It is important to think carefully about data types, because they affect both the performance (the amount of processed information and the number of additional operations performed by the DBMS during data analysis and conversion) and the adequacy to the subject area (there is a risk of using a data type that will not save some objectively existing values or will save them with distortions).

For example, if we choose “too big” integer type (let’s say `BIGINT`) to store primary key values, we will decrease performance. But if we choose “too small” type (let’s say `TINYINT`), we can store not more than 256 different values (i.e., we can put not more than 256 records into the table).

Another perfect illustration of data type dangers is represented by `DATETIME` type in MS SQL Server, which stores fractional values of seconds rounded up to one of three values: .000, .003 or .007, i.e., you won’t be able to store thousandths of a second values, which are .001, .002, .004, .005, .006, .008 or .009.

An approximate representation of how the data types change when moving from the (data)logical level to the physical level is shown in figure 2.1.d.



Logical (datalogical) level Physical level

Figure 2.1.d — Data types at (data)logical and physical levels

Specify all constraints. And here we are not just talking about the uniqueness of values or `NOT NULL`, we need to create **everything**: primary and foreign keys{32} (and finalize all necessary relationships{53}), unique indexes{106}, checks {338}, triggers{341}.

Specify the available approaches and tools of performance enhancement. Those generally include indexes{103} (not to be confused with unique indexes{106}, which are created not so much for performance as to ensure uniqueness of values), storage engines{30} and their parameters, materialized views {331}, etc.

Define specific properties of relations and their attributes. Such properties include encoding settings for string data types, default attribute values, initial values and increment step of autoincrementable values, etc. (This list depends very much on the selected DBMS).

Seeing the number of references to other chapters of the book, you may have already realized that it is extremely difficult to try to implement “relations in their pure form”, in isolation from other database objects. That’s why many ideas, only outlined here in the abstract, will be discussed in detail later.

Now it is worth mentioning a simple thing that produces many questions and mistakes — the difference between “relation” (as a concept of relational theory) and “table” (as an everyday reality of working with DBMS).



See chapter 6 of the “Introduction to Database Systems, 8th edition” book by C. J. Date for a very detailed discussion of this matter.

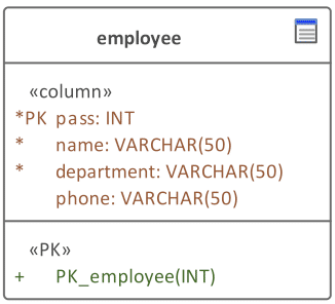
Let’s start with what “relation” and “table” have in common.

Similarity 1. A relation has named typed attributes, their counterpart in table are columns with their own names and types.

Similarity 2. A relation has a set of tuples, their counterpart in table are rows.

This is where the similarities end, and the differences begin.

Difference 1. The attributes of a relation are not ordered, while in table the column sequence is set during the table creation (yet it can be changed within the SQL queries execution and may not coincide with the actual data storage image). Suppose we have a relation represented by the following schema and created at DBMS level by the following SQL code (Figure 2.1.e).



MySQL — Table creation

```mysql
CREATE TABLE `employee`
(
  `e_pass` INT NOT NULL,
  `e_name` VARCHAR(100) NOT NULL,
  `e_department` VARCHAR(10) NOT NULL,
  `e_phone` VARCHAR(50) NULL,
  CONSTRAINT `PK_employee`
  PRIMARY KEY (`e_pass`)
)
```

Figure 2.1.e — Schema of the relation and SQL-code to create a table

Performing a query, we can write like this:

MySQL — The query in which the order of the fields is the same as the actual order of the fields in the table

```mysql
SELECT `e_pass`,
       `e_name`
FROM   `employee`
```

But we can also write it like this (changing the order of the extracted fields):

MySQL — The query in which the order of the fields does **not** match the actual order of the fields in the table

```mysql
SELECT `e_name`,
       `e_pass`
FROM   `employee`
```

As for the physical location of the data on disk, there is no uniform rule — the DBMS (or rather, the storage engine{30}) for optimization purposes may place data relating to each column in an entirely unexpected (for humans) way.

Difference 2. The tuples are not ordered, while the rows in tables are stored in some sequence (as a rule, in order determined by the clustered index{107}, if the table has one).

This difference is associated with a very common (and fundamentally wrong) question: “How can I insert a new row into the table between rows so-and-so and so-and-so?” You cannot.

The fact that the DBMS has to store data in a certain sequence (this is not only an objective necessity, but also a way to optimize search operations) does not mean that you will know anything about this sequence, much less that the DBMS will allow you to manage it.

Assume we have rows

e_name
Smith
Smith
Taylor
Jones

and we want to put the “Jackson" row between the “Taylor” and “Jones” rows, there is no way we can do this. And what for?! If we need alphabetical ordering (for example), it’s enough to write a query with `ORDER BY`:

MySQL — The query that specifies a rule for ordering the result by a field already present in the table

```
1    SELECT   `e_name`
2    FROM     `employee`
3    ORDER BY `e_name`
```

But if we need some completely artificial ordering that is not based on existing data in the table, we have to add a new field and then do the ordering by its value, for example, like this:

e_name	e_order
Smith	10
Smith	20
Taylor	30
Jones	50
Jackson	40

MySQL — The query that specifies a rule for ordering the result by a specially added field

```
1    SELECT   `e_name`
2    FROM     `employee`
3    ORDER BY `e_order`
```

The result will be as follows (the “Jackson” row will appear between “Taylor” and “Jones” rows, as required, but the physical location of data on disk will not change, i.e., this effect will be relevant only within the execution of a separate SQL-query):

e_name
Smith
Smith
Taylor
Jackson
Jones

Difference 3. A relation is not allowed to have attributes with duplicating names, or duplicating tuples, but a table (at least a derived one, i.e., obtained by executing a query) may violate this requirement.

Let’s start with duplicate rows. Almost any modern relational DBMS will allow you to create a table without primary keys{36} and/or unique indexes{106}, and then place completely identical rows there. Why you need this, and how you are now going to distinguish these rows are a separate question (hint: never do that, it causes lots of problems), but technologically there are no obstacles.

As for attributes with duplicating names, the DBMS will not allow that when creating a table, but it is possible when executing SQL queries with `JOIN`. Let’s consider a corresponding example.

Figure 2.1.f shows schemas of two relations in which the same sets of fields have been deliberately created:

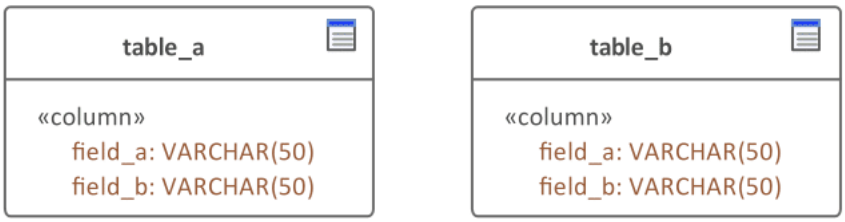


Figure 2.1.f — Schemas of relations to demonstrate the effect of attribute names duplication when executing an SQL query with `JOIN`

Let's create the corresponding tables and fill them with data (figure 2.1.g):

table_a

field_a	field_b
One	1
Two	2

table_b

field_a	field_b
One	1000
Two	2000

Figure 2.1.g — Data in tables to demonstrate the effect of attribute names duplication when executing an SQL query with `JOIN`

Now let's execute the following query (the code is given in MySQL syntax, but note that MS SQL Server behaves the same way, and only Oracle automatically adds the postfix "_1" to the end of the duplicated name):

MySQL — The query resulting with fields (columns) names duplication

```
1    SELECT *
2    FROM   `table_a`
3    JOIN   `table_b`
4    USING (`field_a`)
```

The result of the query is as follows (note that there are two columns `field_b` with the same name but different data):

field_a	field_b	field_b
One	1	1000
Two	2	2000

That's the end of the similarities and differences between relations and tables.

Yet there is one more point that is traditionally neglected in books on databases, because it is more relevant to DBMS documentation. Let's talk about what storage engine is, and how it relates to relations and tables.



Storage engine (database engine[27]) — a software component used by the DBMS to add, read, update, and delete data.

Simplified: a special part of the DBMS with which it works with the disk (files).

Unfortunately, it is almost pointless to describe "storage engines in principle", because their set, specific implementation, structure, capabilities, etc. are very different and strictly related to a particular DBMS. At the same time, many DBMSes allow you to use several storage engines simultaneously, which expands the database design capabilities at the physical level, but also complicates it.

[27] **Database engine (storage engine)** — the underlying software component that a database management system (DBMS) uses to create, read, update and delete (CRUD) data from a database. ("Wikipedia") [https://en.wikipedia.org/wiki/Database_engine]

The choice of storage engine determines the performance of various operations, the availability (and specificity of implementation) of the transaction{364} mechanism, the amount of disk space needed to store data, the requirements to the CPU and RAM, etc., etc.

If this seems too complicated (and it is), you can relax with the thought that any modern DBMS uses by default a “most balanced” storage engine, focused primarily on reliability of data operations (usually at the cost of lost performance and increased requirements to hardware resources). If you seriously want to optimize your database on the physical level, you have to study the relevant section of the documentation of the DBMS you are working with.

At the end of this chapter, we would like to stress once again that successful creation of a database schema requires a good understanding of the operation principles and capabilities of both the selected design tool and the target DBMS. Therefore, you should carefully study the relevant documentation to save a lot of time and avoid many mistakes.

And the continuation of the logic of creating and using relations will be in the sections on the practical implementation of keys{46}, relationships{73} and triggers{341}.



Task 2.1.d: using any design tool, create 3-5 schemas of relations you came up with in task 2.1.a{24}. After creating the schemas, generate the appropriate SQL code and execute it in the DBMS of your choice to create the tables. Fill the resulting tables with sample data.



Task 2.1.e: do all relationships and their attributes in the “Bank{395}” database follow the same standard for database objects naming? Suggest your improvements.



Task 2.1.f: are the optimal data types selected for all attributes of the relationship in the “Bank{395}” database? Suggest your improvements.

2.2. Keys

2.2.1. General Information on the Keys

As each type of key has its own special properties, the definition of a key as such can only be formulated in general terms:



Key[28] — an identifier that is part of a set of data elements. In the context of relational databases, a key is a set of relation attributes (or a separate relation attribute), which has certain properties depending on the type of the key.

Simplified: a field (or a set of fields) subject to certain rules (specific rules depend on the type of key).

Figure 2.2.a shows a general list of keys and their interrelationship with each other.

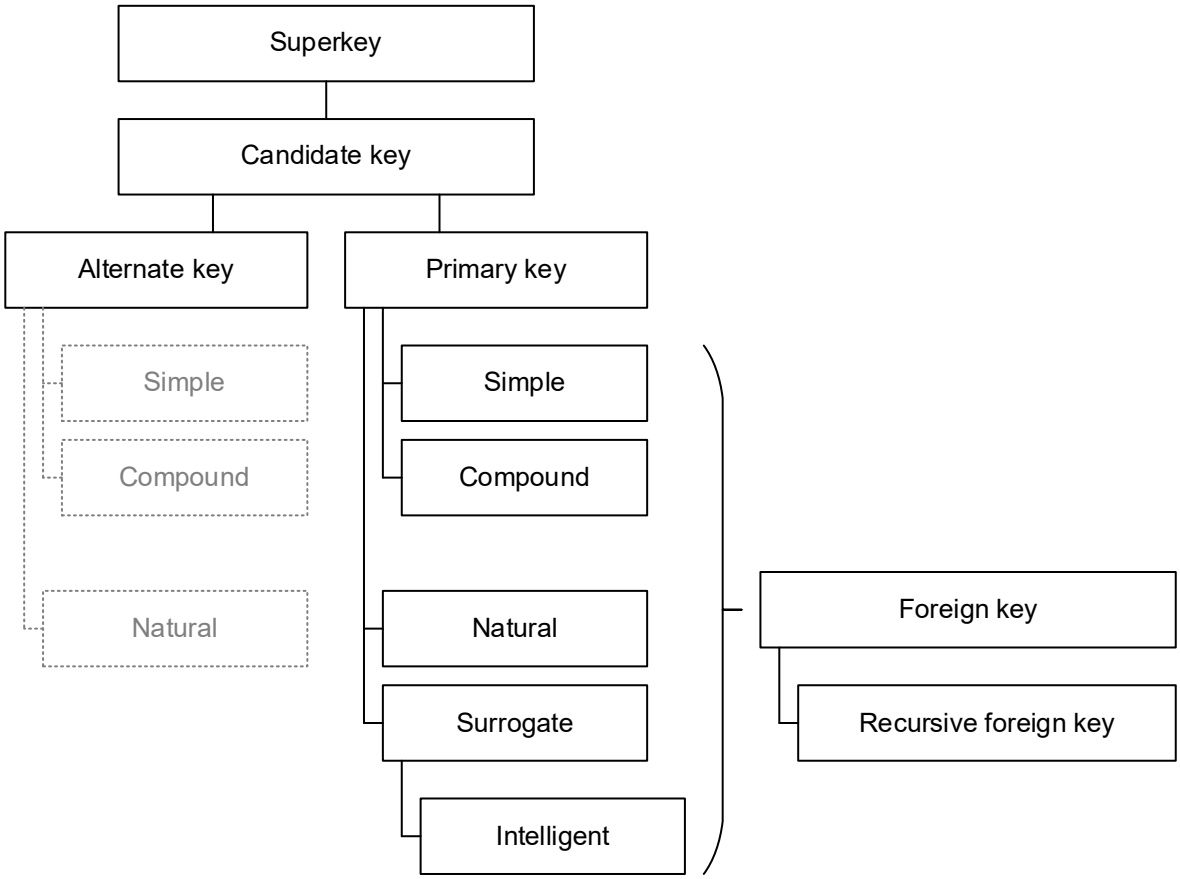


Figure 2.2.a — Types of keys and their interrelationship

Note at the outset that surrogate (and intelligent) alternate keys may exist in theory, but in practice they make no sense (moreover, they are rather harmful, increasing the load on the DBMS).

So, now let's look at all the definitions one by one and give the corresponding examples.

[28] **Key** — identifier that is part of a set of data elements (ISO/IEC 2382:2015, Information technology — Vocabulary).



Superkey[29] — a subset of attributes of the relation, uniquely identifying any tuple. A superkey is often called a candidate key superset, because it may contain "extra" elements, the removal of which will not lead to the loss of uniqueness (because the superkey does not have the property of irreducibility).

Simplified: a field or a set of fields whose values can be used to reliably distinguish one table row from another; there may be extra fields in such a set, but if we remove them, we can still reliably distinguish one row from another.

Although in practice we will rarely be interested in superkeys (they are usually needed only in the context of normalization{157}), let's explain the idea using the example of the `employee` relation already discussed many times (figure 2.2.b).

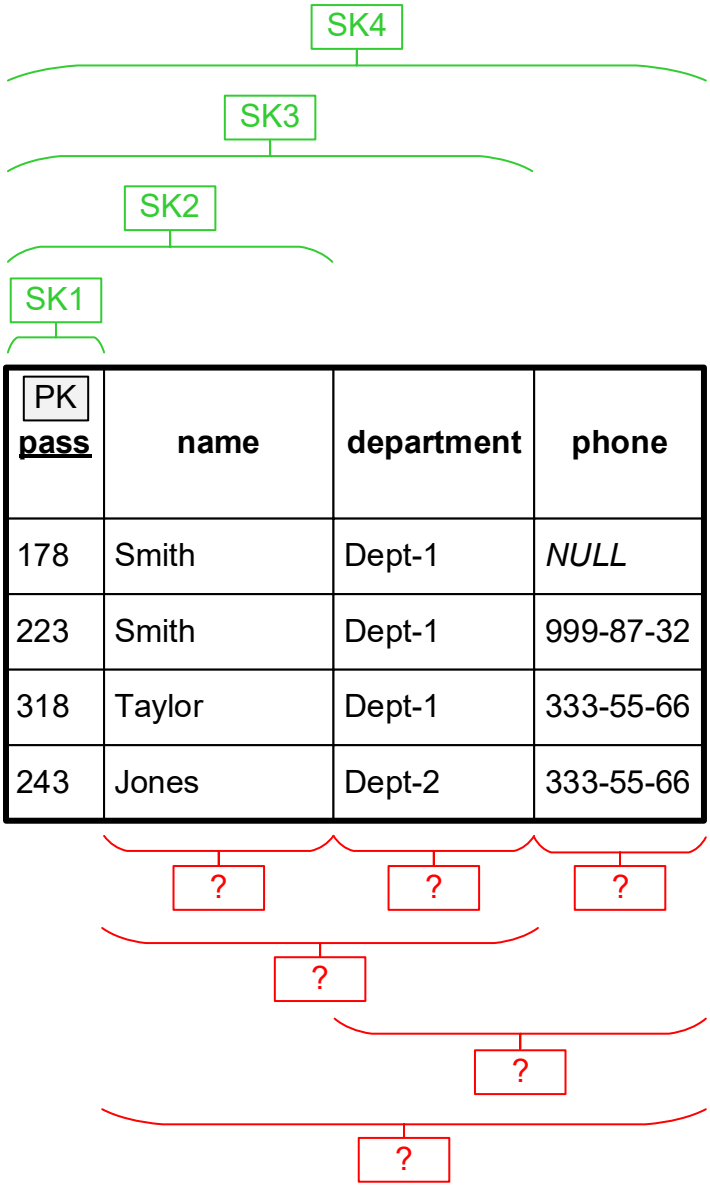


Figure 2.2.b — Superkeys of `employee` relation

[29] **Superkey** — a superset of a candidate key. A superkey has the uniqueness property but not necessarily the irreducibility property. Of course, a candidate key is a special case of a superkey. (C.J. Date, "An Introduction to Database Systems", 8th edition).

Note that all the sets (“SK1”, “SK2”, “SK3”, “SK4”) that include the primary key (`pass` attribute) are objectively superkeys (uniquely identify any tuple), while sets marked with “?” are not superkeys: these attributes or such sets may contain non-unique values.



Candidate key[30] — an irreducible subset of attributes of a relation, uniquely identifying any tuple. In other words, a candidate key is such a superkey that has no “extra” elements. As a rule, one of the candidate keys later becomes the primary key (and other candidate keys become alternative keys).

Simplified: a primary key candidate, i.e., a field or a set of fields with values that can be used to reliably distinguish one table row from another; such a set has no extra fields, i.e., if we remove at least one field from the set, we cannot reliably distinguish table rows anymore.

In the example considered in figure 2.2.b only the “SK1” superkey is a candidate key, because only the `pass` attribute contains guaranteed unique values for each tuple. Other attribute combinations either do not guarantee uniqueness (such combinations are marked with “?” in figure 2.2.b), or include extra elements, removal of which will not lead to loss of uniqueness (“SK2”, “SK3”, “SK4”).

In general, a relation can have several candidate keys, which is very easy to demonstrate in the example of `car` relation (figure 2.2.c).

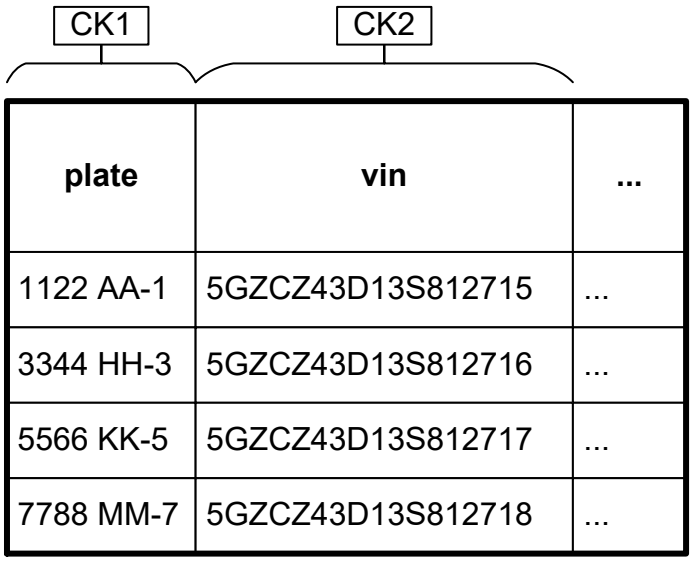


Figure 2.2.c — Example of a relation with several candidate keys

As a car has a unique state registration plate (`plate` attribute) and VIN (`vin` attribute), the value of any of these attributes is guaranteed to identify any tuple of the relation, i.e., each of these attributes is a candidate key.

As just noted in the definition of a candidate key, as a rule, one of these keys becomes the primary key, and the others become alternate keys.

[30] **Candidate key** — a set of attributes that has both of the following properties: uniqueness and irreducibility. (C.J. Date, “An Introduction to Database Systems”, 8th edition).



Alternate key[31] — a candidate relation key that is not selected as a primary key.

Simplified: a primary key candidate that never became a primary key.

If we continue looking at the `car` relation in figure 2.2.c, after choosing the attribute `plate` as the primary key (it will be explained soon why `vin` is worse suited for this role {36}), the remaining candidate key (the `vin` attribute) becomes an alternate key.

Alternate keys require special attention. By definition, their values are guaranteed to identify the relation tuples (i.e., their values cannot be duplicated), but the DBMS (so far) "does not know" anything about it and controls the uniqueness of the primary key values only.

To bring the DBMS behavior in compliance with the subject area requirements, we have to create so-called "uniqueness constraints" on alternate keys (`CHECK UNIQUE`), which in most DBMS means building a "unique index{106}" on the given key.

Let's depict everything just described in figure 2.2.d.

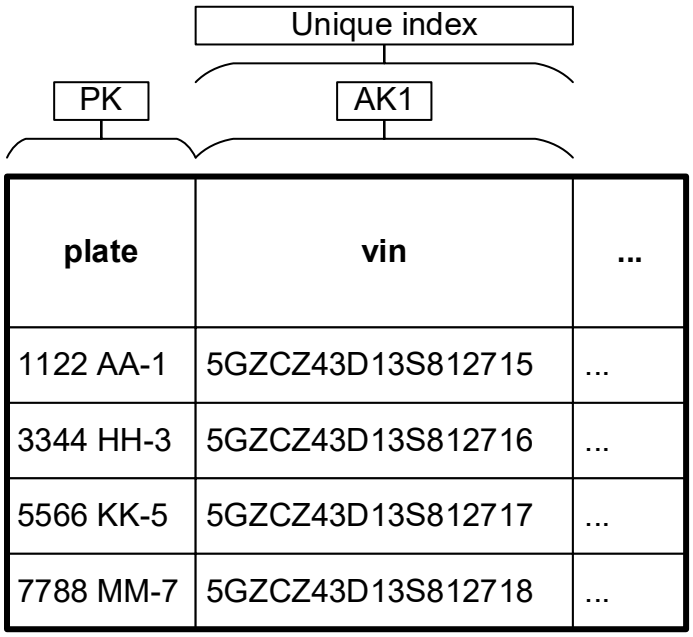


Figure 2.2.d — Primary and alternate keys

Note that the alternate key name "AK1" has a postfix ("1"), because a relation can theoretically have many alternate keys, while there may only be one primary key ("PK") in any relation.

The varieties of alternate keys ("simple", "compound", "natural") coincide with the similar varieties of primary keys (and will be discussed later).

Also note that though the "surrogate" (and its variety — "intelligent") alternate key may exist in theory, in practice it makes no sense, because it does not give any advantages for data management, but only creates an additional useless load on the DBMS.

[31] **Alternate key.** It is possible for a given relvar to have two or more candidate keys. In such a case the relational model requires that exactly one of those keys be chosen as the primary key, and the others are than called **alternate keys**. (C.J. Date, "An Introduction to Database Systems", 8th edition).



Primary key[32] (PK) — the candidate key chosen as the main means of guaranteed identification of the tuples of the relation.

Simplified: a unique identifier of the row in the table.

From the previously considered definitions it follows that a primary key must have the property of irreducibility (i.e., it must not include “extra” elements that can be discarded without loss of uniqueness of values).

Also (since the notion of a “primary key” is related not so much to the general theory of databases as to the DBMS practice) we should mention that a primary key must have the property of minimality, which should be interpreted literally[33]: if there are some candidate keys consisting of the same number of attributes, we should choose as a primary the candidate key that is “physically smaller”, i.e., contains less data.

That is why in the previously considered example{35} (figure 2.2.d) we chose the `plate` attribute as a primary key instead of `vin`: the less data the DBMS has to compare (to ensure the uniqueness of values), the faster it will perform this operation.

So, the most basic facts about the primary key that are important to understand at this point:

- the values of the primary key cannot be duplicated (i.e., they are unique for each row of the table);
- the primary key must be irreducible (i.e., it cannot contain “extra” elements) and minimal (i.e., the “smallest” candidate key must be chosen as the primary key);
- the primary key is used for guaranteed unambiguous identification of a table row (which follows from the uniqueness property of its values) and for organizing relationships (to be discussed later, see "foreign key{43}” and “relationships{53}”);
- the primary key cannot contain an undefined value (`NULL`) — this property follows from the logic of referential integrity{67}.

Further discussion of primary keys we will conduct in the context of their varieties. And the first of the classifications of primary keys divides them into simple and compound.



Simple key[34] — a key consisting of a single relation attribute.

Simplified: a key built on exactly one table column.



Compound key (composite key[35]) — a key consisting of two or more relation attributes.

Simplified: a key built on two (or more) table columns.

As you may have noticed, the word “primary” is missing from these definitions, since both of these definitions apply equally to primary, alternate, and foreign keys.



Some authors emphasize that the term “compound key” refers strictly to foreign keys, while “composite key” refers to candidate keys (and then to primary and alternate keys). Precisely to eliminate these terminological disputes, the definitions in the footnotes are taken from “The New Relational Database Dictionary” by C.J. Date, where there is no such division, and the terms “compound key” and “composite key” are clearly marked as synonyms.

---

[32] **Primary key** — key that identifies one record (ISO/IEC 2382:2015, Information technology — Vocabulary).

[33] In relational theory, the concepts of “irreducibility” and “minimality” are often used interchangeably, but here we deliberately separate them in order to emphasize two, in fact, different properties of the primary key.

[34] **Simple key** — a key that’s not composite. (“The New Relational Database Dictionary”, C.J. Date)

[35] **Compound key / composite key** — terms used interchangeably to mean a key consisting of two or more attributes. Contrast simple key. (“The New Relational Database Dictionary”, C.J. Date)

As an example of a simple and a compound key, let's consider the familiar `employee` relation.

Previously, we assumed that `pass` numbers are unique (not repeated throughout the firm), and this is the situation reflected in figure 2.2.e — the `pass` attribute is a classic case of a simple primary key.

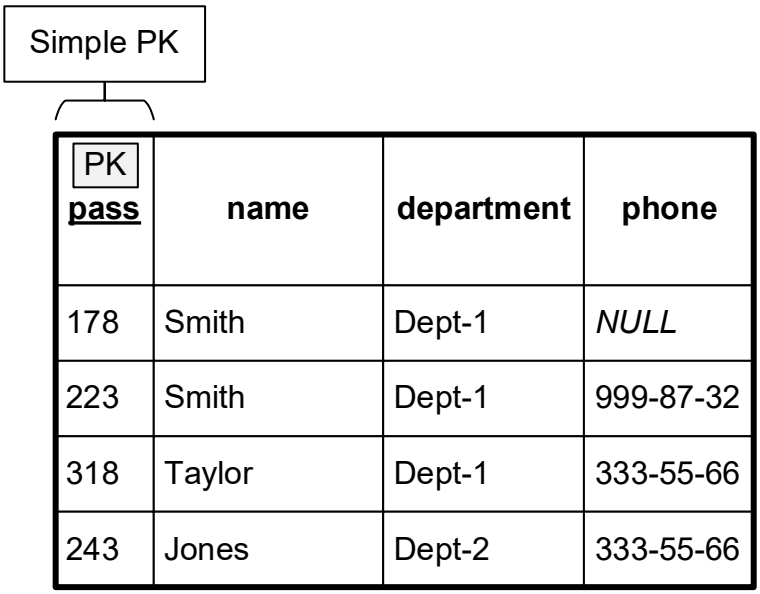


Figure 2.2.e — Simple primary key

But if `pass` numbers were unique only within departments, we would need to know **both** `pass` number **and** `department` name to be guaranteed to distinguish one employee from another, which would force us to create a compound primary key on `pass` and `department` attributes. This situation is shown in figure 2.2.f.

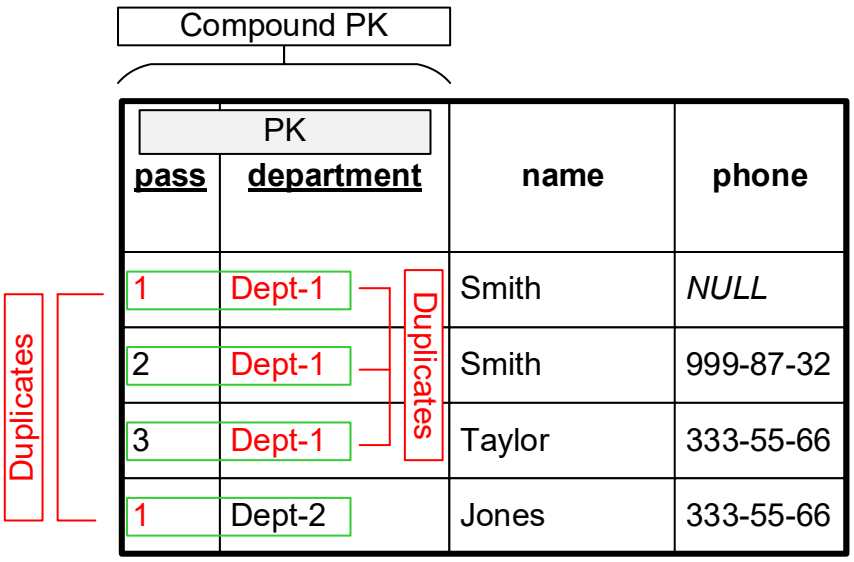


Figure 2.2.f — Compound primary key

As follows from figure 2.2.f, both the `pass` attribute and the `department` attribute separately allow duplicate values (i.e., they cannot act as a primary key on their own), but their aggregate meets the requirement of uniqueness of values, and therefore can be a primary key.



From the theoretical point of view (since attributes of a relation are considered unordered) it does not matter which attribute in the compound primary key goes first (it would be even more correct to say that in the relational approach it is impossible to figure out which attribute goes first, second, etc.), but in practice the sequence of fields in compound keys and indexes is **critically important** — this will be touched upon very soon{47}.

Now let's talk about natural, surrogate and intelligent keys.

For alternate keys from this classification there is only one variant: alternate keys in reality are only natural keys.

Surrogate and intellectual keys are very rarely "visible" at the infological modelling level and "appear" at the (data)logical or even physical level.

To be fair, it is worth noting that intelligent keys in recent years are practically not used.

So, let's consider some definitions.



Natural key[36] (business key) — a key, built on a set of relation attributes, carrying a semantic load.

Simplified: if a relationship has attributes with unique values in the real world (e.g., passport id, vehicle registration plate number), the key built on them will be natural.



Surrogate key[37] — a key built on an attribute artificially added to a relation for the sole purpose of guaranteeing that the tuples of the relation are identified.

Simplified: a new attribute is added to the relation; this attribute has nothing to do with the subject area but is only needed to guarantee the identification of individual table rows.



Intelligent key[38] — a key with values that not only identify the tuples of the relation, but also carry additional information.

Simplified: a key with unique values carrying some additional information about the table row.

Graphically the natural primary key is shown in figure 2.2.g.

[36] **Natural key** — term sometimes used to refer to a key that's not a surrogate key. ("The New Relational Database Dictionary", C.J. Date)

[37] **Surrogate key** — a single-attribute (i.e., simple) key with the property that its values serve solely as surrogates for the entities they're supposed to stand for. In other words, those surrogate key values serve merely to represent the fact that the corresponding entities exist, and they carry absolutely no additional information or meaning. ("The New Relational Database Dictionary", C.J. Date)

[38] **Intelligent key** — a single-attribute key whose values, in addition to their main purpose of serving as unique identifiers (typically for certain real world "entities"), carry some kind of encoded information embedded within themselves. ("The New Relational Database Dictionary", C.J. Date)

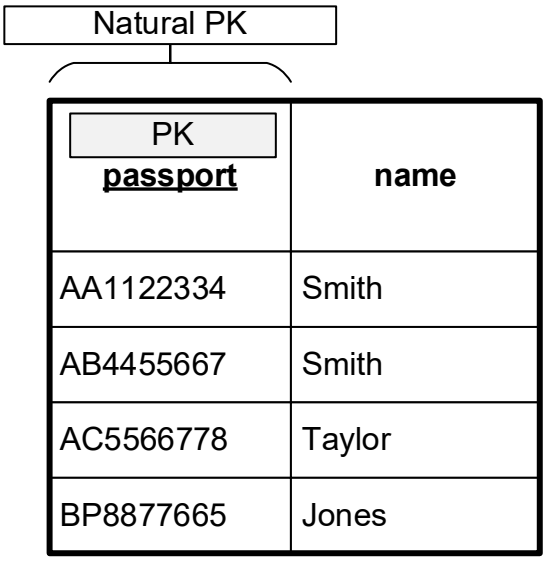


Figure 2.2.g — Natural primary key

The advantages of such a key are as follows:

- a corresponding attribute already exists in the relation, i.e., there is no need to add anything (and store it, wasting unnecessary memory);
- in cases where it does not contradict security requirements, the corresponding value can be shown to users (and used by them to facilitate their work);
- in most DBMSes and storage engines{30} the use of the natural primary key provides a very useful primary index{107}, which is actively used in most operations on the table.

Alas, the disadvantages of the natural key are many more, and they are very serious:

- if for security reasons the value of the attribute selected as the natural primary key cannot be used in some operations, it makes the corresponding operations impossible;
- the size of a natural primary key (with few exceptions) is larger than the size of a surrogate primary key (often many times larger);
- if for some reason in real life we do not know the value of a natural primary key, we either cannot add a record to the table, or we have to “invent” a fake value (which is even worse);
- in some relations a natural primary key can only be compound;
- if the value of a natural primary key changes (e.g., a person’s passport id changes), the DBMS will have to perform multiple cascade operations{68} to maintain referential integrity{67} (this case is illustrated in figure 2.2.h: if an employee’s passport id changes, the DBMS will have to update all payment records to still “know” which employee those payments belong to).

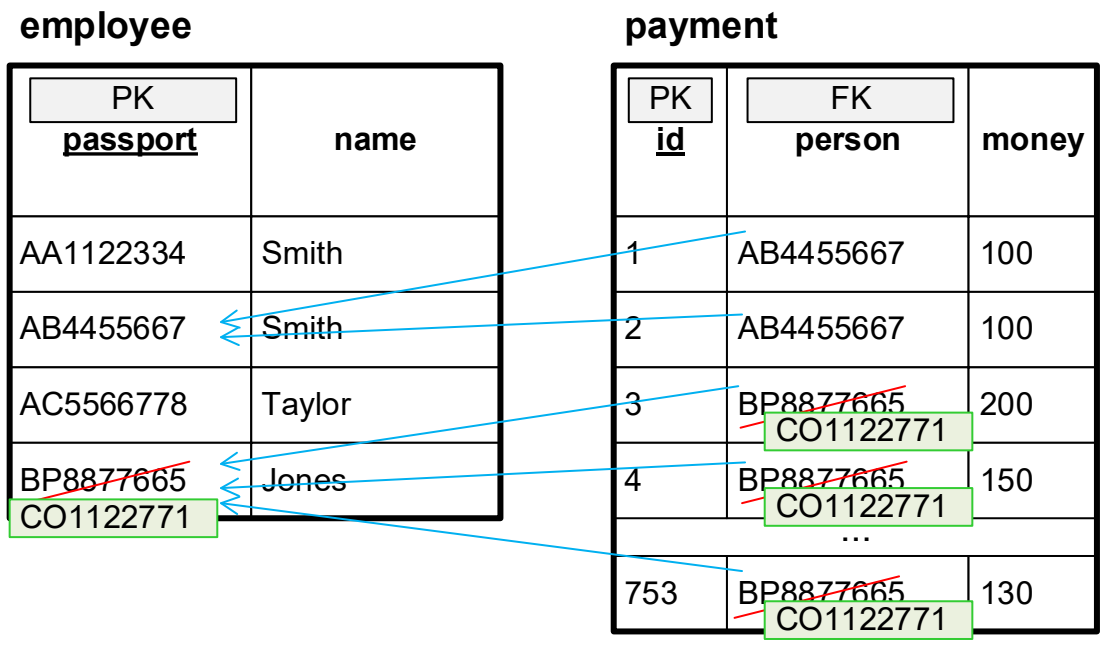


Figure 2.2.h — Cascade update when the value of the natural primary key changes

The advantages and disadvantages of the natural primary key turn into, respectively, the disadvantages and advantages of the surrogate primary key (shown in figure 2.2.i).

The disadvantages of the surrogate primary key are as follows:

- an appropriate attribute must be added to a relation (and stored, wasting extra memory);
- a value of this attribute has no “real meaning” and so even if it is not against security rules to show it to users, they still (most likely) cannot use it to simplify their work;
- in some DBMSes and storage engines{30} it is not possible to create a primary index{107} separate from the primary key, and therefore the physical organization of the table and many operations with it will not be implemented efficiently.

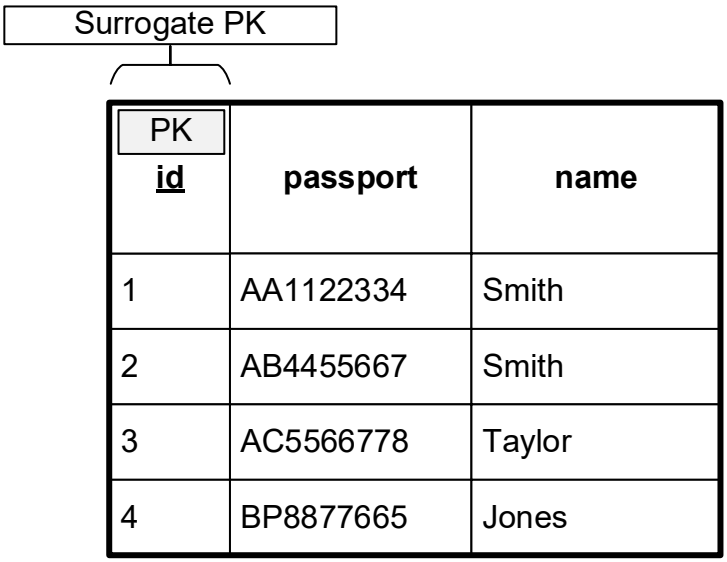


Figure 2.2.i — Surrogate primary key

The advantages of a surrogate primary key:

- because the value of such a key has no real meaning (it does not contain any “real data”), there are usually far fewer security restrictions on its use;
- the size of a surrogate primary key (with few exceptions) is smaller than the size of a natural primary key (often many times smaller);
- the value of a surrogate primary key (as a rule) are generated automatically, so the probability of the situation that we “don’t know” this value when adding a record to a table is vanishingly small;
- a surrogate primary key will almost always be simple;
- the value of a surrogate primary key does not change, so the DBMS does not need to perform any cascade operations{68} to maintain referential integrity{67} (this case is illustrated in figure 2.2.j).

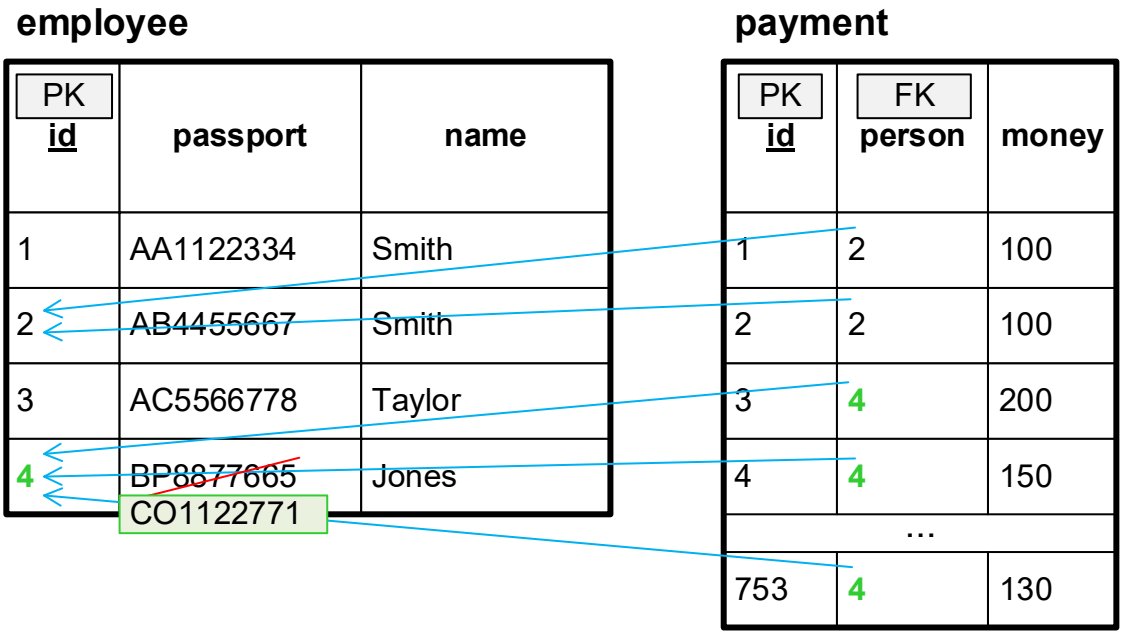


Figure 2.2.j — No cascade update needed when changing the value of the surrogate primary key

Due to the fact that the advantages of a surrogate primary key are many and extremely weighty, it is common practice to create such keys even in relations that have natural candidate keys.



If it is possible to use a natural key, which does not have the mentioned above disadvantages{39}, it is still worth using it — it is logical at least from the point of view of database theory.

Why then, in many databases, do their developers “forcibly” put a surrogate primary key in every table (except for intermediate “many to many{56}” relationship tables)? Because of usability{12}: yes, you can argue as much as you want that this approach is inefficient and illogical, but if it helps reduce errors in the product being developed, it is used.

In any case, when deciding in favor of a surrogate or a natural primary key, it is worth weighing the advantages and disadvantages of both solutions.

And even rarer than natural primary keys you can find intelligent primary keys (an example of such a key is shown in figure 2.2.k).

The essence of the idea of an intelligent primary key is to generate, using some algorithm, a relatively small amount of data[39] that is not only guaranteed to identify any tuple of the relation, but also contains some useful information.

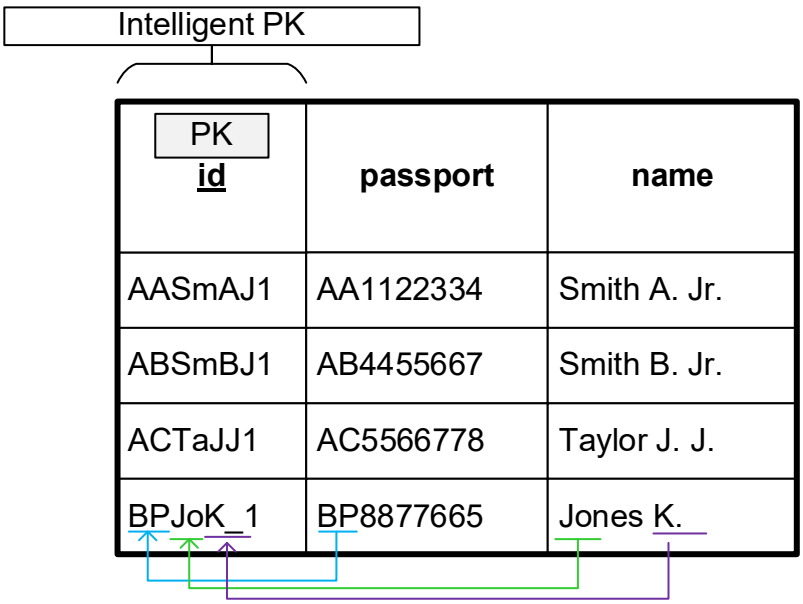


Figure 2.2.k — Intelligent primary key

In figure 2.2.k such a key contains a part of passport id, the first two letters of the last name and the initials of each person, and also a numeric postfix at the end is added in case of so-called “collisions” (if all the data for two or more people coincide, then the postfix will be increased by one for each such coincidence, so that the final value of the key is still unique).

In theory, intelligent keys were supposed to combine the advantages of natural and surrogate keys, while being free of their disadvantages. In reality, it turned out to be exactly the opposite — such keys almost always combine the disadvantages of natural and surrogate keys, and do not have their advantages. In addition to that it’s necessary to implement a rather complicated (and slow) mechanism of generation of such keys every time.

Although in some cases intelligent keys can be useful, the common practice of using surrogate keys has completely superseded intelligent keys.

So, that’s it with primary keys. Now we move on to a totally different kind of keys: foreign keys (and their variety: recursive foreign keys).

[39] In fact, intelligent key can be considered as a partially reversible hash function (a part of the original data can be restored from the hash value).



Foreign key[40] (FK) — a relation attribute (or a group of attributes) that contains copies of the values of the primary key of the other relation.

Simplified: a field in a child (subordinate) table that contains copies of the values of the primary key of the parent (main) table.



Recursive foreign key[41] (RFK) — a relation attribute (or a group of attributes) that contains copies of the values of the primary key of the same relation.

Simplified: a foreign key that is in the same table as the referenced primary key; a variant of the foreign key in the case where the table refers to itself.

We have already seen foreign keys in figures 2.2.h and 2.2.j (the `person` attribute) when we discussed natural and surrogate keys. This idea is illustrated again in figure 2.2.l.

employee

PK **id**	**passport**	**name**
1	AA1122334	Smith
2	AB4455667	Smith
3	AC5566778	Taylor
4	BP8877665	Jones

payment

PK **id**	FK **person**	**money**
1	2	100
2	2	100
3	4	200
4	4	150
…		
753	4	130

Figure 2.2.l — Foreign key

Here the `employee` table is the parent (main) table, and the `payment` table is the child (subordinate) table. Accordingly, the primary key of the parent table (`id` attribute) migrated to the child table and became the foreign key (`person` attribute) there. It is already obvious at this point that the foreign key name **does not** have to be the same as the name of the referenced primary key.

Continuing with figure 2.2.l, note the arrows that show to which entry of the parent table each entry of the child table refers (payments 1 and 2 refer to employee 2, and payments 3, 4, and 753 refer to employee 4).

Obviously, the foreign key does not have to possess the property of uniqueness of values (it is easy to notice that the `person` attribute has values 2 and 4 repeated several times), although it can have it (in the case of “one to one{58}” relationship).

[40] **Foreign key** — in a relation, one or a group of attributes that corresponds to a primary key in another relation (ISO/IEC 2382:2015, Information technology — Vocabulary).

[41] **Recursive foreign key** — a foreign key that references some key of relation itself. (“The New Relational Database Dictionary”, C.J. Date)

Now let's consider how recursive foreign keys work. A classic case of their use is the organization of hierarchical structures, such as the sitemap, which is shown in figure 2.2.m.

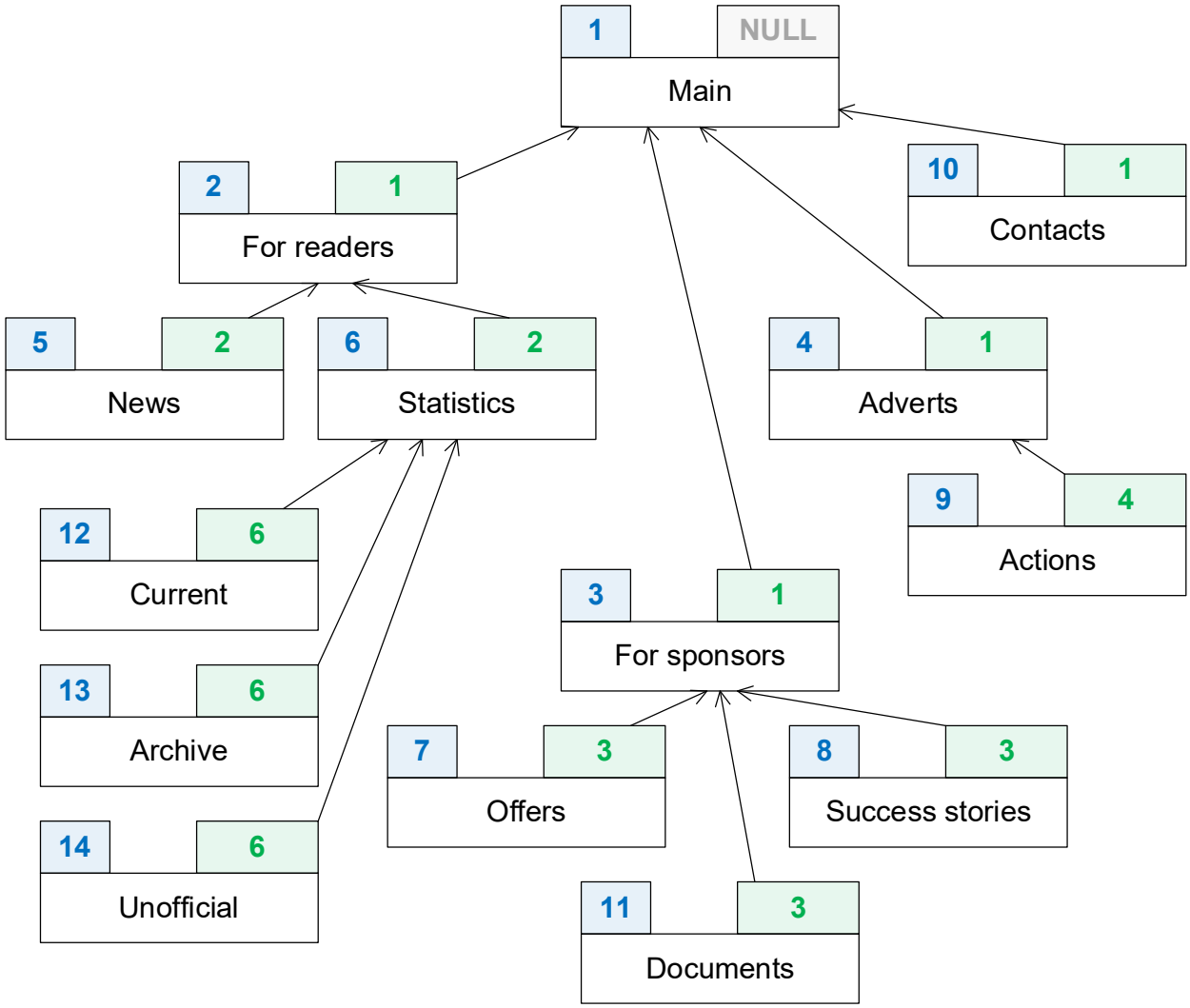


Figure 2.2.m — Visual representation of the sitemap

Technically, to organize such a structure in a relational database you can use the relation shown in figure 2.2.n.

So, "Main" page has no parent page (the value of the foreign key of this record is `NULL`), and for all other records the `parent` field contains the value of the primary key of the corresponding parent record.

Despite the fact that relational databases are poorly suited to organize such hierarchical structures, the solution presented in this example is widespread and is very actively used.

There are several problems (with solutions) just for handling this very situation in "Using MySQL, MS SQL Server and Oracle by examples"[42] book.

[42] "Using MySQL, MS SQL Server and Oracle by examples" (Svyatoslav Kulikov), example 46 [https://svyatoslav.biz/database_book/]

sitemap

PK **id**	RFK **parent**	**name**
1	NULL	Main
2	1	For readers
3	1	For sponsors
4	1	Adverts
5	2	News
6	2	Statistics
7	3	Offers
8	3	Success stories
9	4	Actions
10	1	Contacts
11	3	Documents
12	6	Current
13	6	Archive
14	6	Unofficial

Figure 2.2.n — Organizing a sitemap using a recursive foreign key

To summarize the above in one phrase, it turns out that foreign keys are used to organize connections between tables (or even within a table), and further discussion on this subject will de continued in the appropriate chapter{53}.

And now we move on to working with the keys in practice.



Task 2.2.a: once again review the relations and tables created in 2.1.a{24} and 2.1.d{31}. Name their candidate keys, primary keys, alternate keys.



Task 2.2.b: in which relations of the “Bank{395}” database should other keys have been chosen? Make appropriate edits to the schema if you have enough arguments to implement such changes.



Task 2.2.c: in which relations of the “Bank{395}” database would it be worth creating composite keys? Make appropriate edits to the schema if you have enough arguments to implement such changes.

2.2.2. Creating and Using the Keys

First of all, it should be noted that not all of the theoretical ideas just discussed find their way into everyday applications (see figure 2.2.o).

Let's also emphasize that this chapter deals with design at the (data)logical and physical levels (and there is practically no difference between them in terms of key implementation). What happens to the keys at the infological (conceptual) modelling level will be described in the corresponding chapter{278}.

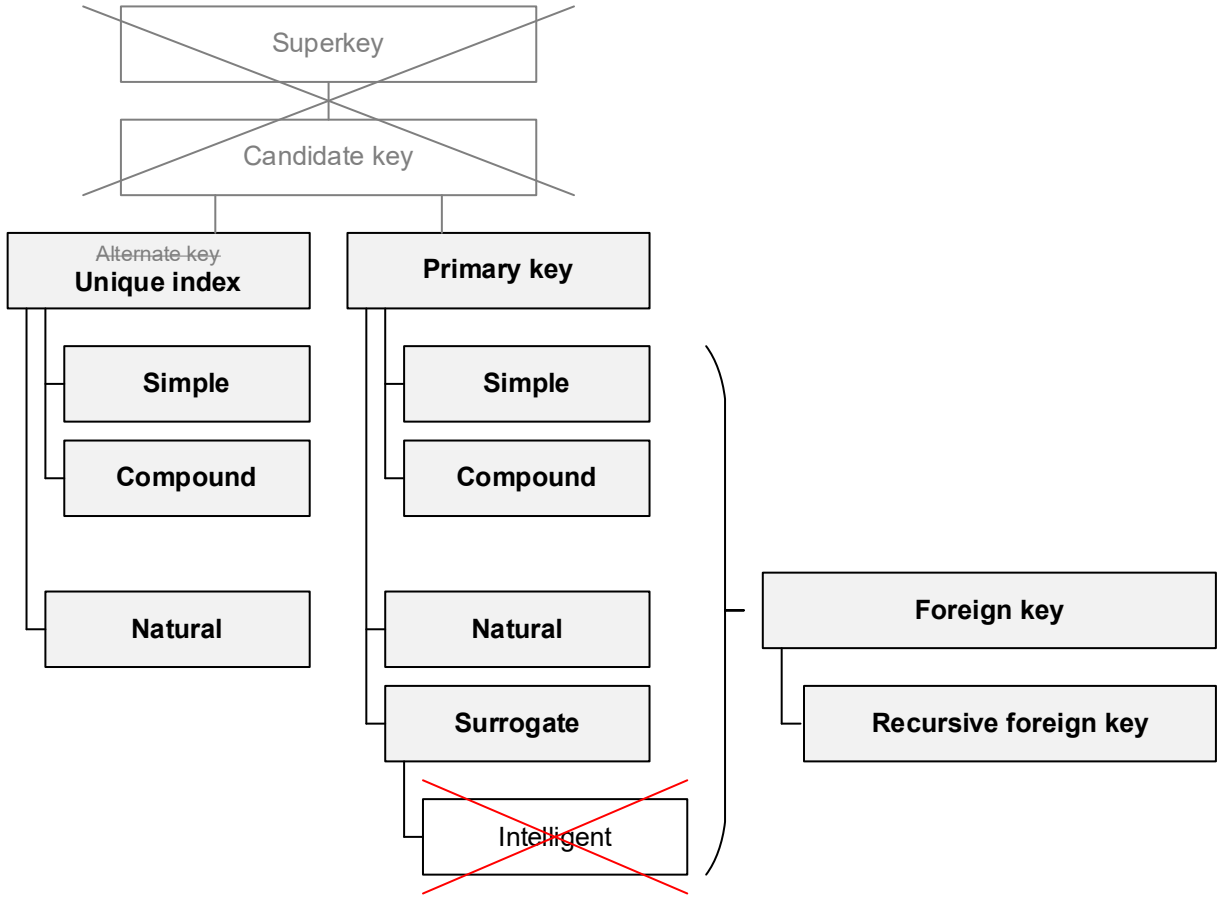


Figure 2.2.o — Types of keys and their interrelationship (in practice)

So, superkeys are almost never thought of, and candidate keys are only considered when deciding how to organize a primary key in a table. Alternate keys turn into unique indexes{106}. And intelligent keys are almost never used due to the reasons discussed above{42}.

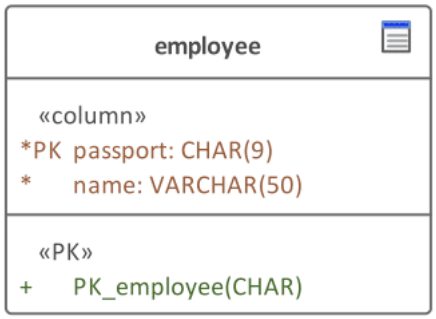
To create a simple primary key in a table, you need to:

- for a natural primary key, mark the table field selected as such a key (figure 2.2.p);
- for a surrogate primary key first add a new field to the table, and then mark the field accordingly (figure 2.2.q).

Usually, in relations schemas, the primary key is placed at the very top, i.e., to make it the first field of the table (or the first field**s**, if the key is compound).

In the vast majority of database design tools to "turn" a field into a primary key, it is sufficient to simply check the appropriate checkbox, because the creation of primary keys is a very popular and often performed operation.

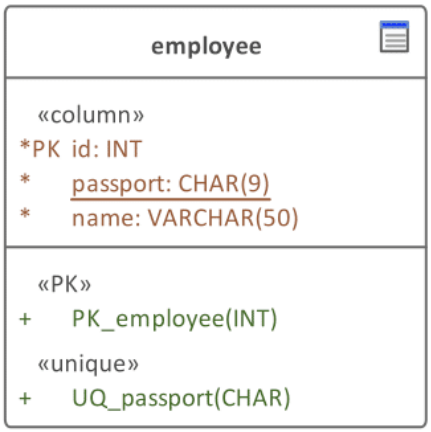
In UML notation schemas, the primary key is marked with the acronym "PK". Its name is sometimes underlined, but in general the underlined attribute name means that control of uniqueness of its values is enabled.



MySQL — Creating a simple natural primary key

```
1   CREATE TABLE `employee`
2   (
3     `passport` CHAR(9) NOT NULL,
4     `name` VARCHAR(50) NOT NULL,
5     CONSTRAINT `PK_employee`
6     PRIMARY KEY (`passport`)
7   )
```

Figure 2.2.p — Creating a simple natural primary key



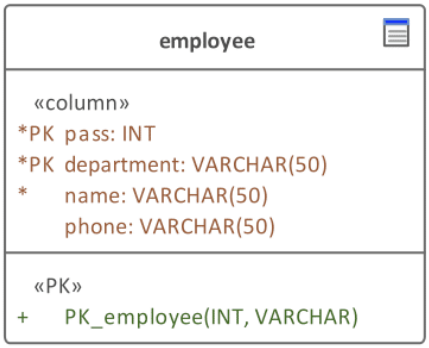
MySQL — Creating a surrogate primary key

```
1   CREATE TABLE `employee`
2   (
3     `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
4     `passport` CHAR(9) NOT NULL,
5     `name` VARCHAR(50) NOT NULL,
6     CONSTRAINT `PK_employee`
7     PRIMARY KEY (`id`)
8   );
9
10  ALTER TABLE `employee`
11    ADD CONSTRAINT `UQ_passport`
12    UNIQUE (`passport`);
```

Figure 2.2.q — Creating a surrogate primary key

Note that in both relations primary keys and uniqueness constraints (**`UQ_passport`**) attributed (according to UML syntax) not to properties of relations, but to operations. And formally this is correct, because what from human point of view looks like “this field will be a primary key”, from DBMS point of view looks like “for this field from now on you need to perform a certain set of operations”.

To create a compound natural primary key, you must follow the same steps — mark all necessary fields as parts of a primary key (figure 2.2.r).



MySQL — Creating a compound natural primary key

```
1   CREATE TABLE `employee`
2   (
3     `pass` INT UNSIGNED NOT NULL AUTO_INCREMENT,
4     `department` VARCHAR(50) NOT NULL,
5     `name` VARCHAR(50) NOT NULL,
6     `phone` VARCHAR(50) NULL,
7     CONSTRAINT `PK_employee`
8     PRIMARY KEY (`pass`,`department`)
9   )
```

Figure 2.2.r — Creating a compound natural primary key

And now it is time to explain the previously mentioned fact{38}: the sequence of fields in the compound primary key (as well as in the composite index{105}) plays a very important role.

Jumping ahead, let’s say that most often the primary key is a clustered index{107}, i.e., the DBMS physically arranges the data on disk by the values of the primary key. If such a key consists of a single field, there are no problems.

But if it consists of several fields, the DBMS can effectively search only by a combination of the fields included in the primary key or by the first field separately. But for the second (third, etc.) field alone, such an effective search does not work.

Let's illustrate this idea visually (figure 2.2.s).

Compound primary key			
pass	**department**	**name**	**Phone**
1	Dept-1	Ivanov I.I.	111-22-33
1	Dept-6	Petrov P.P.	111-22-44
1	Dept-18	Sidorov S.S.	111-22-55
1	Dept-21	Sidorov S.S.	222-22-33
2	Dept-1	Orlov O.O.	222-22-11
2	Dept-2	Berkutov B.B.	222-22-00
2	Dept-3	Boborov B.B.	222-66-33
4	Dept-34	Sinitsyn S.S.	999-12-11
5	Dept-1	Vorobyov V.V.	999-12-12
5	Dept-3	Voronov V.V.	999-12-15
5	Dept-7	Lvov L.L.	999-12-17
6	Dept-4	Volkov V.V.	888-10-01
7	Dept-3	Zaytsev Z.Z.	888-10-02
7	Dept-5	Okunev O.O.	888-10-03
8	Dept-2	Karasev K.K.	888-10-05
11	Dept-3	Shchukin S.S.	888-10-91
12	Dept-1	Petrov P.P.	888-10-81
12	Dept-5	Sidorov S.S.	888-10-81
13	Dept-6	Sidorov S.S.	999-12-11
13	Dept-9	Orlov O.O.	999-12-12
14	Dept-17	Berkutov B.B.	999-12-15
18	Dept-17	Boborov B.B.	999-12-17
23	Dept-9	Sinitsyn S.S.	888-10-01
23	Dept-12	Vorobyov V.V.	888-10-02
31	Dept-11	Voronov V.V.	999-12-11
32	Dept-14	Lvov L.L.	999-12-12
34	Dept-17	Volkov V.V.	999-12-15
45	Dept-1	Zaytsev Z.Z.	999-12-17
45	Dept-2	Okunev O.O.	888-10-01
46	Dept-1	Karasev K.K.	888-10-02
56	Dept-3	Orlov O.O.	999-12-12
56	Dept-53	Berkutov B.B.	999-12-15
71	Dept-3	Boborov B.B.	999-12-17
82	Dept-1	Sinitsyn S.S.	888-10-01

Figure 2.2.s — Illustration of the logic of compound primary keys' work

Try to answer the following questions quickly:

- Is there an employee with the “87” pass?
- How many employees do have the “1” pass?
- What department does an employee with the “34” pass work in?
- What is the name of the employee with the “1” pass and the “Dept-18” department?
- What is the phone number of the employee with the “7” pass and the “Dept-3” department?

Even for a human, such a task is not difficult: first you (very quickly) look up the pass number in the **`pass`** column and (if you find it) just as quickly look up several rows in the **`department`** column in search of the right department. No complications.

Now try to answer the following questions just as quickly:

- How many employees are there in the “Dept-3” department?
- Is there at least one employee in the “Dept-15” department?
- What is the largest pass number in the “Dept-2” department?

These operations cannot be performed as quickly: you have to look through the entire **`department`** column each time (from the very top to the very bottom) and temporarily switch to other actions when you find each match you are looking for.

Even though in reality the DBMS performs slightly different operations, the concept remains the same: all the key fields or the first key field separately are usable for search spud up, but the second, third, etc. key fields separately are not.

Hence a very simple conclusion: if your database often runs queries in which the search is performed on a separate field of the composite key (or index), this field should be the first in such a key (or index).

That’s all for the primary keys. Let’s move on to the unique indexes{106} into which the alternative keys were “turned”.

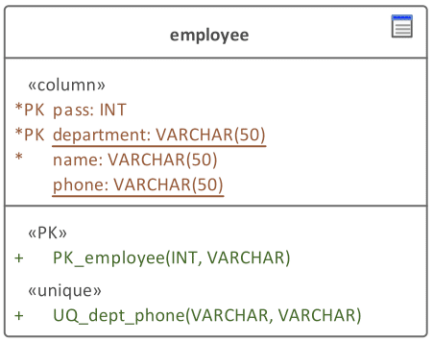
Technically, creating unique indexes in most database design tools is done by marking the “unique” property of the corresponding table field(s). In some design tools, you have to create a separate operation (with “**`unique`**” type) for the table and specify the field(s) that it should work with.

It has already been shown above (albeit implicitly) how to create a simple natural unique index — in figure 2.2.q such an index is created on the **`passport`** field to guarantee the uniqueness of its values.

If there is a need to create a composite natural unique index, the “**`unique`**” operation needs specification of the whole set of fields that it should work with (here it is not enough just to mark each field with the “**`unique`**” property, because the design tool may think that you are creating not a single composite index{105}, but many separate simple indexes{105}).

Suppose some business rules in some company forbid two or more employees to have the same phone numbers in the same department (i.e., the combination of **`department`** and **`phone`** fields’ values should be unique).

Figure 2.2.t shows the corresponding situation (remember the sequence of fields{47} — all the same reasoning that was given for keys is typical for indexes too).



MySQL — Creating a composite natural unique index

```
1   CREATE TABLE `employee`
2   (
3    `pass` INT UNSIGNED NOT NULL AUTO_INCREMENT,
4    `department` VARCHAR(50) NOT NULL,
5    `name` VARCHAR(50) NOT NULL,
6    `phone` VARCHAR(50)      NULL,
7    CONSTRAINT `PK_employee`
8    PRIMARY KEY (`pass`,`department`)
9   );
10
11  ALTER TABLE `employee `
12   ADD CONSTRAINT `UQ_dept_phone`
13    UNIQUE (`department`,`phone`);
```

Figure 2.2.t — Creating a composite natural unique index

Don't be confused by the fact that the `department` field is now part of both the primary key and the unique index. In the general case (before we start talking about normalization{157}) this is quite a typical and acceptable situation.

Another interesting fact: with the appearance of the {`department`, `phone`} index (where the `department` field is the first), the DBMS is now able to quickly search by the `department` field separately, i.e., the previously mentioned "questions that cannot be answered quickly{49}" are no longer a problem for the DBMS.

We're almost done with primary keys and unique indexes. If you are still interested in whether you can force modern DBMS to use intelligent keys, then yes, you can (though you don't need to, because you are likely to have more problems than benefits): the classic solution is to use triggers{341} (which will produce the key value) or indexes on computed columns{125}, if the DBMS supports them.

It remains to consider the creation of foreign keys. This issue is closely related to the concept of "relationships", and therefore it is recommended to read, among other things, the relevant chapter{53}.

The DBMS begins to "understand" that a certain field (or set of fields) of a table is a foreign key only when the relationship between the tables is explicitly established. Therefore, to create a foreign key you must create a relationship between the corresponding child and parent tables (figure 2.2.u).

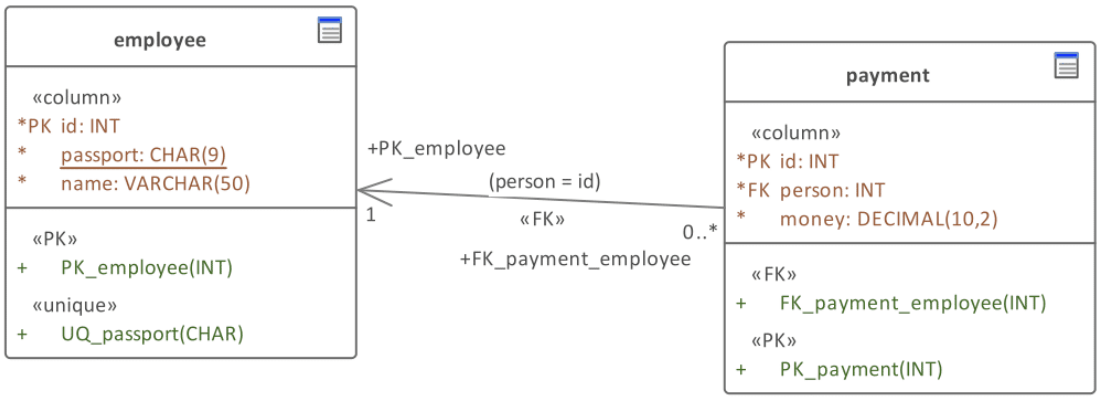


Important! The direction of the relationship matters. It determines which table is the child one and which is the parent one. So, be sure to read the documentation for your design tool to find out which direction ("parent to child" or "child to parent") the relationships should be in.

Note that most design tools follow the "child to parent" rule.



Another complication is that some design tools automatically create a field (or set of fields) in the child table that will be a foreign key, some assume that this field is already created (and offer to select it), and some even know how to work both ways. Again: read the documentation!

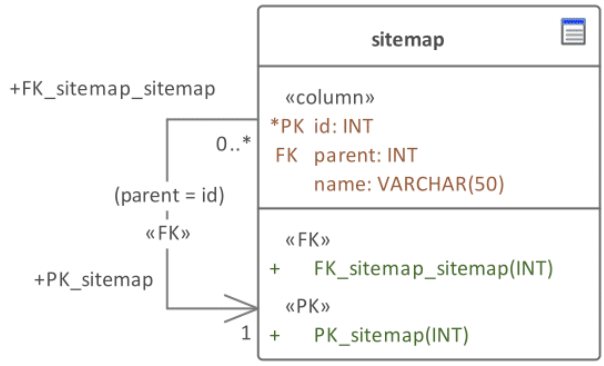


MySQL — Creating a foreign key (and the corresponding relationship)

```
1   CREATE TABLE `employee`
2   (
3           `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
4           `passport` CHAR(9) NOT NULL,
5           `name` VARCHAR(50) NOT NULL,
6           CONSTRAINT `PK_employee` PRIMARY KEY (`id`)
7   );
8
9   CREATE TABLE `payment`
10  (
11          `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
12          `person` INT UNSIGNED NOT NULL,
13          `money` DECIMAL(10,2) NOT NULL,
14          CONSTRAINT `PK_payment` PRIMARY KEY (`id`)
15  );
16
17  ALTER TABLE `employee`
18   ADD CONSTRAINT `UQ_passport` UNIQUE (`passport`);
19
20  ALTER TABLE `payment`
21   ADD CONSTRAINT `FK_payment_employee`
22    FOREIGN KEY (`person`) REFERENCES `employee` (`id`)
23     ON DELETE CASCADE ON UPDATE RESTRICT;
```

Figure 2.2.u — Creating a foreign key (and the corresponding relationship)

With a recursive foreign key, the relationship “returns” to the same table where it begins (figure 2.2.v).



MySQL — Creating a recursive foreign key (and the corresponding relationship)

```
1   CREATE TABLE `sitemap`
2   (
3    `id` INT UNSIGNED
4         NOT NULL AUTO_INCREMENT,
5    `parent` INT UNSIGNED NULL,
6    `name` VARCHAR(50) NULL,
7    CONSTRAINT `PK_sitemap`
8    PRIMARY KEY (`id`)
9   );
10
11  ALTER TABLE `sitemap`
12   ADD CONSTRAINT
13   `FK_sitemap_sitemap`
14    FOREIGN KEY (`parent`)
15    REFERENCES `sitemap` (`id`)
16    ON DELETE CASCADE
17    ON UPDATE RESTRICT;
```

Figure 2.2.v — Creating a recursive foreign key (and the corresponding relationship)

And it remains to emphasize two more important points, often unobvious to those who are just beginning to work with databases.



In UML (and many other notations) relationship arrows go simply “from one relation to another”, not “from a particular field to a particular field”. That is, it is meaningless to try to understand which fields are related by looking up “where the relationship arrow comes from and where it goes to”. The relevant information is written on the relationship itself, and the geometry of the lines is determined solely by the convenience of arrangement in the schema.



Crucially! The data types of the primary key and the corresponding foreign keys must be completely identical (i.e., type, size, signed or unsigned, encoding, format, etc. — any properties that the corresponding fields have should match (except for autoincrementability, which the foreign key should not have)). This is important because in this case the DBMS does not spend additional resources on data conversion when comparing the values of the primary and foreign keys. It is also recommended to create indexes{103} on foreign keys, which may speed up the `JOIN`-query execution.

Almost all modern DBMSes allow violating this rule very roughly (e.g., making the primary key a `BIGINT` number and the foreign key a `VARCHAR` string), but such mockery of the database at least leads to a catastrophic drop in performance, and at worst leads to errors in the execution of some queries.

At the end of this chapter, let’s note again that at the (data)logical and physical levels of design, the work with keys looks almost identical.

Of the possible differences, we can mention only the wider range of key properties available at the physical level in some design tools (although most of the time there are no such differences at all).

This concludes the discussion of keys, but in the following sections (see practical implementation of relationships{73} and triggers{341}) a great number of the thoughts expressed here will be continued and presented in more detail.



Task 2.2.d: study the documentation of your DBMS for different ways to create keys (as a rule, there are several alternative syntax forms that allow you to get the same result).



Task 2.2.e: if in task 2.2.c{45} you suggested using compound keys, are the elements optimally arranged in them? Make appropriate changes to the schema if you have sufficient arguments to implement such changes.



Task 2.2.f: is it possible to use an alternative syntax for generating keys in the “Bank{399}” database code? If you think “yes”, write such code and check if it works.

2.3. Relationships

2.3.1. General Information on the Relationships

In this chapter we need two basic definitions, upon which all subsequent considerations are based.



Relationship[43] — an association that connects several entities.

Simplified: an indication of the fact that relations are in some kind of connection with each other.



Relationship cardinality[44] — a relationship property that defines the admissible capacity of connected subsets of tuples of relations united by the given relationship.

Simplified: a specification of the number of related rows in tables connected by a relationship (e.g.: "one to many", "many to many", "one to three", etc.)

As mentioned above, relationships are organized using foreign keys{43}. And for the "many to many" case, in addition to the foreign keys, you also need to create the so-called "association entry" ("association relation") (see below{56}).

Now let's consider the three classic types of relationships.



One to many relationship[45] — an association that connects two relations in such a way that one tuple of the parent relation (or even zero of such tuples) can correspond to any number of tuples of the child relation.

Many to one relationship[46] — an association that connects two relations in such a way that an arbitrary number of tuples of the child relation can correspond to one tuple of the parent relation (or even zero of such tuples).

Simplified: one record in table A can correspond to many records in table B; in this case there can be a situation when some records in table B do not correspond to any records in table A.

It is not for nothing that these two definitions are given together. Note: the footnoted definitions of these two situations are completely identical (and this is not a typo). The only "difference" is in how (in which direction) to analyze the relationship.

Let's look at an example (figures 2.3.a and 2.3.b; see also the practical implementation in the corresponding section{73}). If we start the relationship analysis on the **`employee`** relation side, we get a "one to many relationship", i.e., "one employee has many payments". If you start analyzing the relationship from the **`payment`** relation side, we get a "many to one" relationship, i.e., "many payments belong to one employee". But in fact, it is the same relationship.

[43] **Relationship** — an association among entities. ("The New Relational Database Dictionary", C.J. Date)

[44] **Cardinality** — the number of elements in a set. ("The New Relational Database Dictionary", C.J. Date)

[45] **One to many correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to at least one element of s2 and each element of s2 corresponds to exactly one element of s1; equivalently, that pairing itself. ("The New Relational Database Dictionary", C.J. Date)

[46] **Many to one correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to exactly one element of s2 and each element of s2 corresponds to at least one element of s1; equivalently, that pairing itself. ("The New Relational Database Dictionary", C.J. Date)

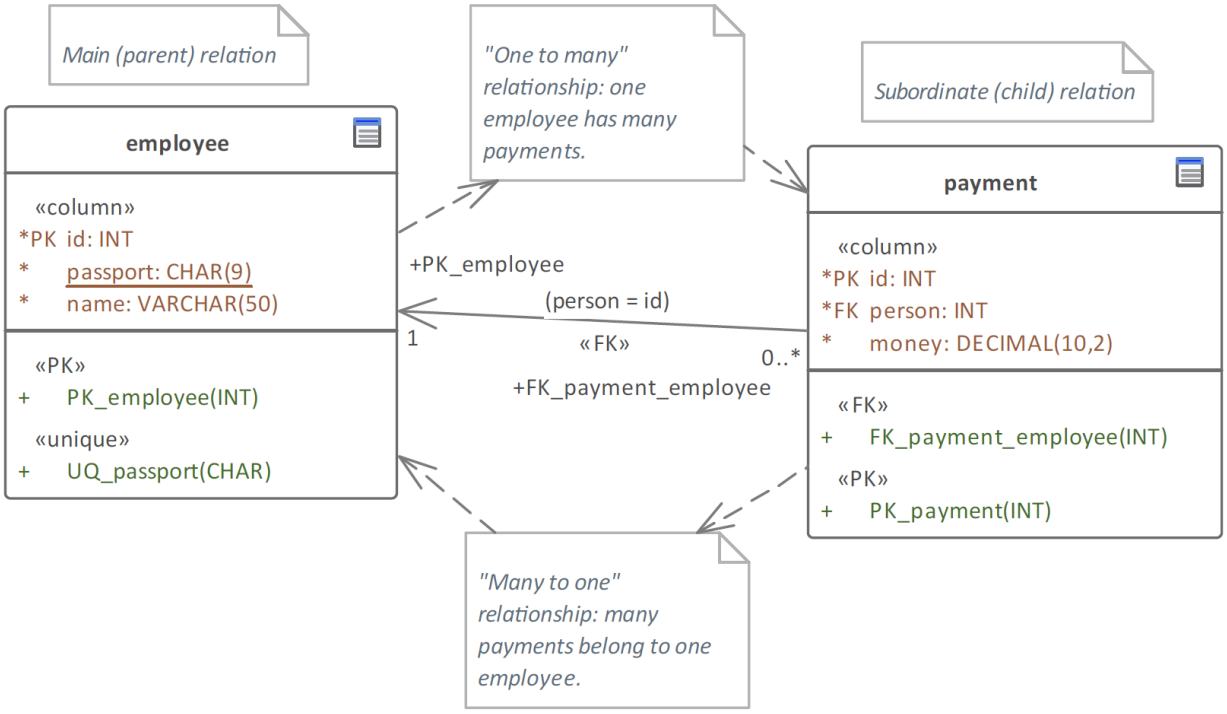


Figure 2.3.a — Schema of “one to many” (AKA “many to one”) relationship

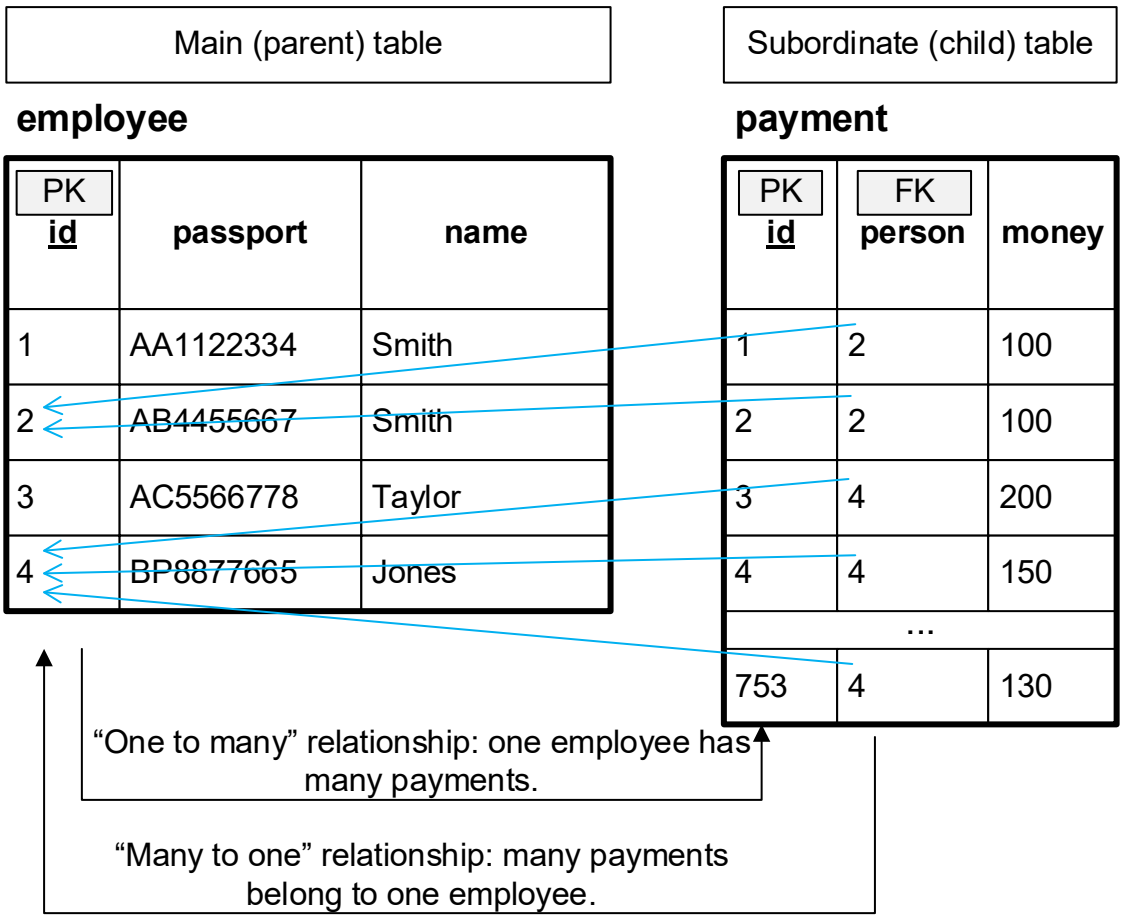


Figure 2.3.b — Tables showing “one to many” (AKA “many to one”) relationship



An important conclusion: in reality (at DBMS level) there is no difference between “one to many” and “many to one” relationships. For ease of perception, almost everywhere this relationship is called “one to many” (it is easier and more familiar to start analysis from the parent table side).

To make it easy to remember: the main (parent) table is the one with the “one” side, and the subordinate (child) table is the one with the “many” side.

To organize this relationship, we must use a foreign key[43], which is located in the child table. The child table also specifies the additional properties of the connection.

Let’s continue the example shown in figure 2.3.b and assume that payments can be cash and wire. I.e., it is not enough just to show that “Smith has been credited with some amount of money”, it is necessary to show “how exactly” this amount of money has been credited.

This “how exactly” in this case is the relationship property, which in the case of “one to many” relationship is specified in the child table (see figure 2.3.c). Objectively this property cannot be specified in the parent table, because then it would be possible to pay a certain employee either **only in cash**, or **only by wire**.

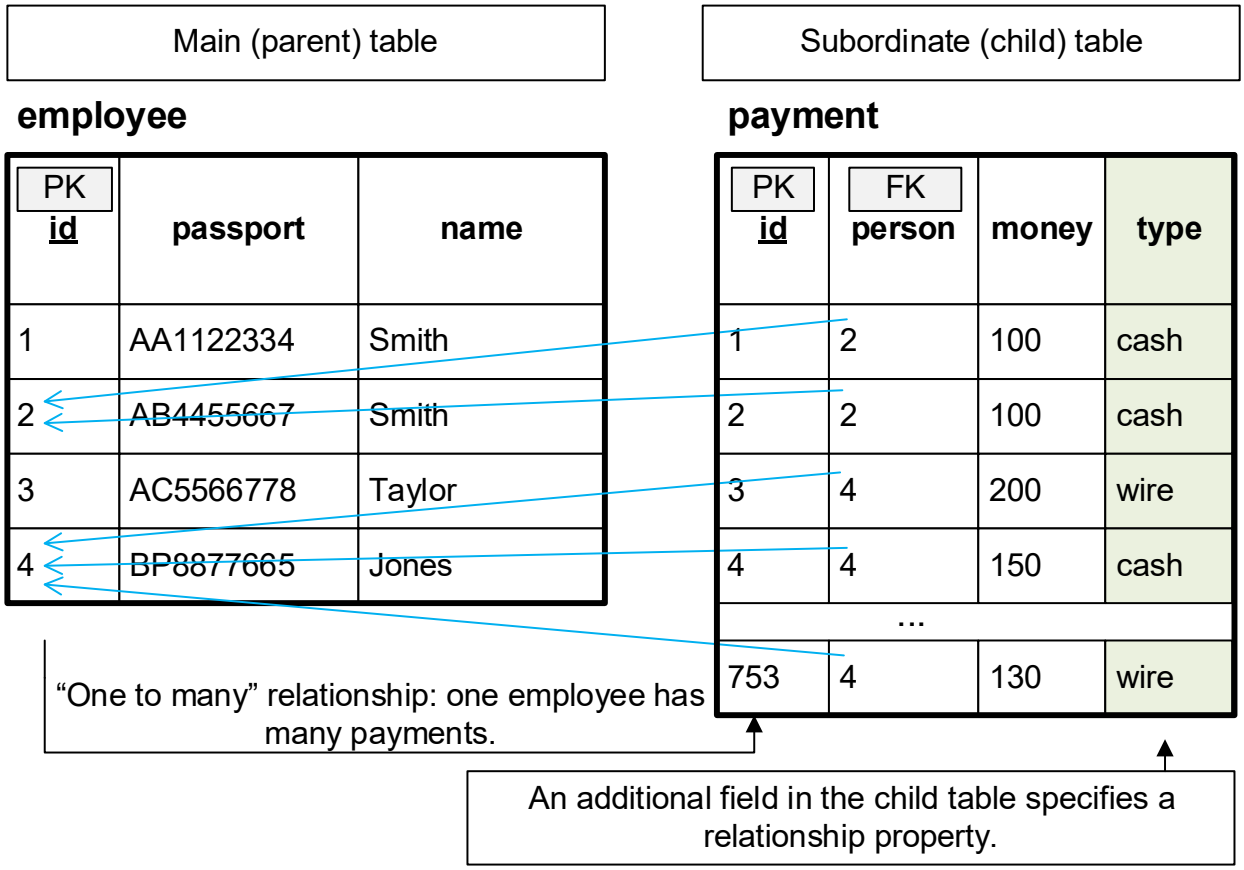


Figure 2.3.c — The way of “one to many” relationship properties specification

If the relationship has other properties (e.g., whether the payment was an advance payment, whether it was charged to the main account or an additional account, what currency it was charged in, etc.) — all these properties will be specified by additional fields in the child table.

For ease of reference, we can assume that in the case of “one to many” relationship, the properties of the relationship are equivalent to the properties of the entries in the child table.



Many to many relationship[47] — an association that connects two relations in such a way that one tuple of any of the connected relations can correspond to any number of tuples of the other relation.

Simplified: one record in table A can correspond to many records in table B, and at the same time one record in table B can correspond to many records in table A.

Consider the following classic example: students and subjects. Each student studies many subjects, and each subject is studied by many students.

It is impossible to specify such a situation in a database with “one to many” relationship, because there are no explicit parent and child entities (and an attempt to perform mutual migration of primary keys would lead to very serious technical violations and complete loss of the adequacy to the subject area{11}).

In this case, a so-called “association relation” is created (it appears in most design tools at the physical level), which is a child of both connected tables. And it is this table into which the primary keys of the connected tables are migrated.

At the (data)logical level (see figure 2.3.d) such a relationship does not even have such technical parameters as keys, operations, etc. — all those simply do not exist here yet[48] (and can only appear together with the association relation).

On the physical level (see figures 2.3.e and 2.3.f; also see the practical implementation in the corresponding section{83}) we actually get two separate “one to many” relationships, through which we form the desired “many to many” relationship).

This produces two foreign keys in the association relation, which are often combined into a compound natural primary key (the need for this operation will be discussed below{84}).

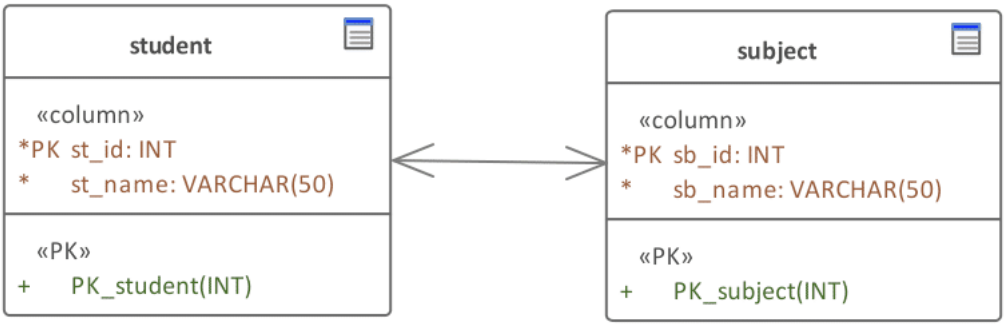


Figure 2.3.d — Schema of “many to many” relationship at the (data)logical level

[47] **Many to many correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to at least one element of s2 and each element of s2 corresponds to at least one element of s1; equivalently, that pairing itself. (“The New Relational Database Dictionary”, C.J. Date)

[48] Yes, often this information appears in the process of modeling at the (data)logical level, although formally it belongs to the physical one: it’s just more convenient, and this once again illustrates the fact that the boundaries between modeling levels are very arbitrary.

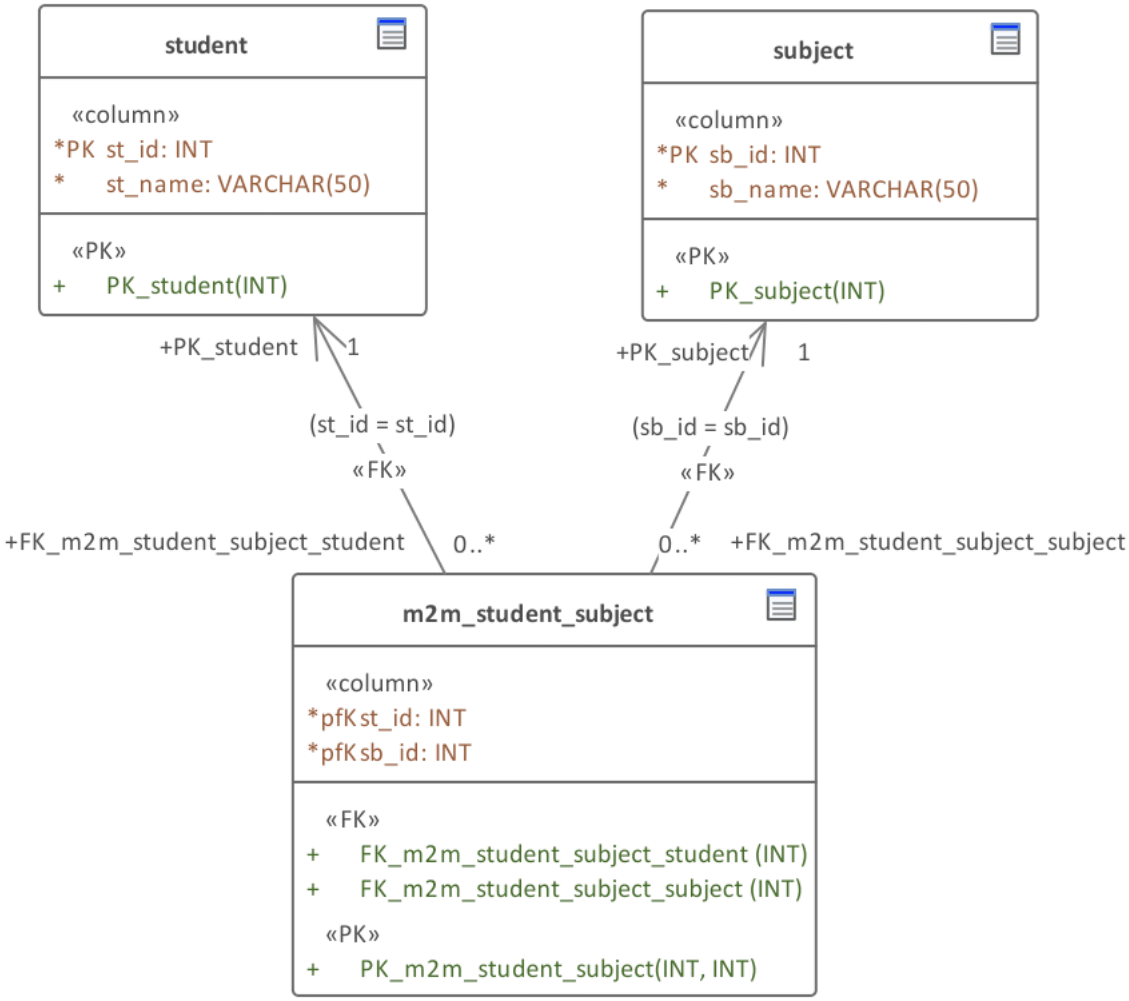


Figure 2.3.e — Schema of “many to many” relationship at the physical level

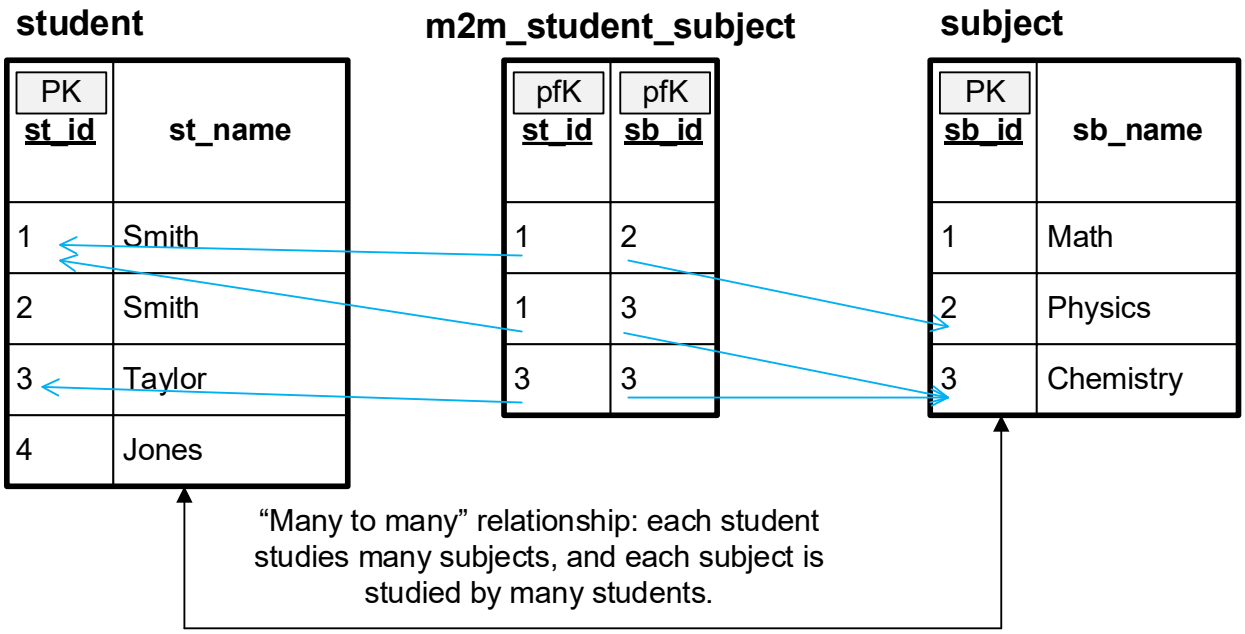


Figure 2.3.f — Tables representing “many to many” relationship at the physical level

In the case of “many to many” relationship, the so-called “relationship properties” are specified in the association relation (table). E.g., at the end of a course a student should receive a grade. But the grade is objectively neither a property of the student (he has many grades, not one), nor a property of the subject (it is illogical to give grades to a subject at all) — it is precisely a property of the relationship between a particular student and a particular subject. The corresponding example is shown in figure 2.3.g.

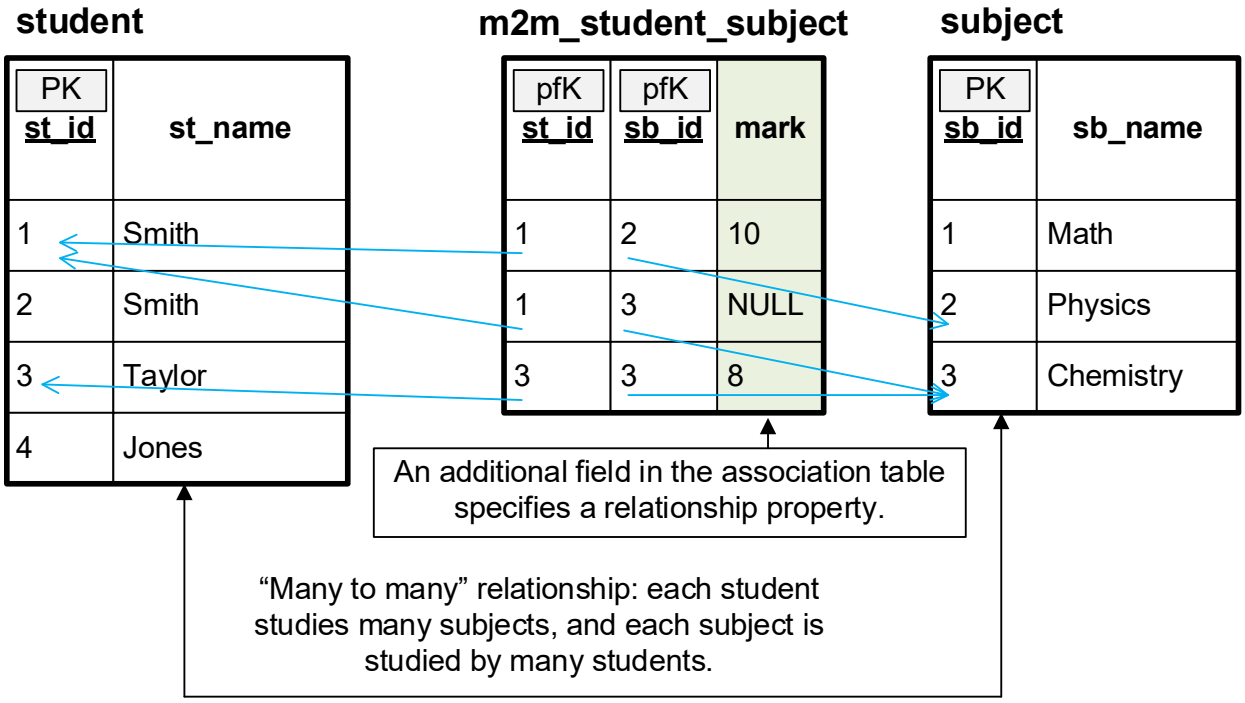


Figure 2.3.g — The way of “many to many” relationship properties specification

And it remains to consider the last classic version of relationships.



One to one relationship[49] — an association that connects two relations in such a way that one tuple of the parent relation can correspond to no more than one tuple of the child relation.

Simplified: one record in table A can correspond to no more than one record in table B.

It follows from the definition that this relationship is a special case of the “one to many” relationship, where an additional restriction is imposed on the child relation — the foreign key in such a relation must have the property of uniqueness of its values (sometimes this is achieved by making the foreign key also the primary key, but there are other technical ways, such as using a unique index{106}).

The reason we consider this relationship in the last place is that it has a very narrow scope.



If a “one to one” relationship appears in your database design, either you have a very strong reason for its existence, or... you have a design error. These relationships rarely appear “by themselves”, because nothing prevents you from simply putting all the relevant attributes into one relation (instead of two relations connected with “one to one” relationship).

[49] **One to one correspondence** — a rule pairing two sets s1 and s2 (not necessarily distinct) such that each element of s1 corresponds to exactly one element of s2 and each element of s2 corresponds to exactly one element of s1; equivalently, that pairing itself. (“The New Relational Database Dictionary”, C.J. Date)

So, “one to one” relationships are used if:

- there are entities of different types in the subject area that are united by this relationship (for example, “driver’s license” and “traffic violations registration blank”, although even in this situation it is very likely that all the information will be contained in one relation);
- when describing some entity with a huge number of properties, we have reached the DBMS’ limitations on the number of fields in one table or on the maximum record size — then we can “continue” in the next table, combining it with the previous one with this relationship (but immediately there is doubt that the schema is built correctly, because in reality this situation is highly unlikely);
- in order to optimize performance, we want to split into two separate tables those data that need access very often and those that need access very rarely, which will allow the DBMS to process smaller amounts of data during frequent operations (and, again, there are other ways to optimize performance for such situations);
- there are many entities of different types in the subject area being described, each of which nevertheless has the same set of matching properties (this is probably the only case where a “one to one” relationship is really useful; it will be discussed later{86}).

In the schema (see figure 2.3.h), the “one to one” relationship looks almost like a “one to many” relationship (the difference is in the numbers indicating the relationship cardinality{53}, and in the fact that in the child table the primary key is also a foreign key).

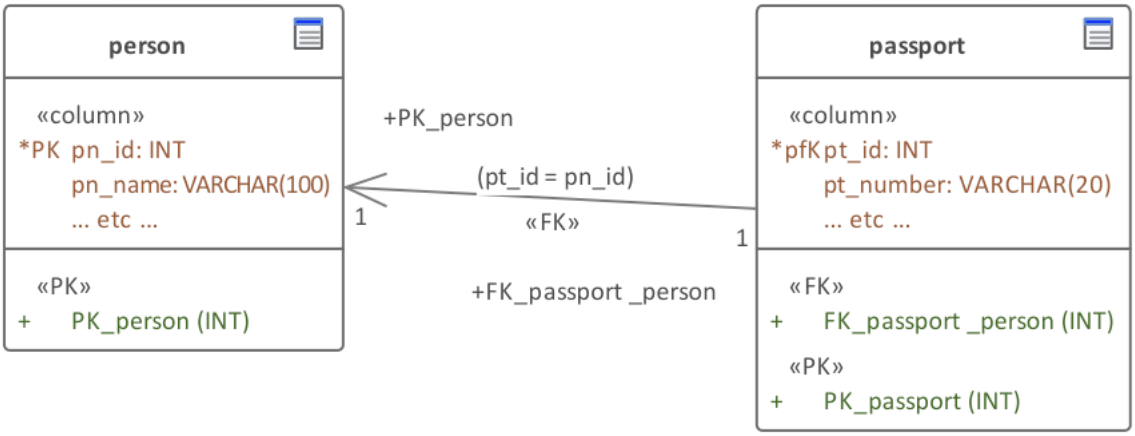


Figure 2.3.h — Schema of “one to one” relationship

More detailed examples of how to work with “one to one” relationships will be shown in the corresponding section{86}.

But relationship cardinality can also take on “non-standard” values, i.e., in addition to “one to many”, “many to many”, and “one to one” relationships, there can also be “X to Y” relationships, where X and Y are arbitrary integers or ranges, such as “1 to 3”, “1 to 0…5”, “0…2 to 1”, etc.

All corresponding options are determined solely by the requirements of the subject area that the database describes. For example:

- “a system should have at least one (but no more than thee) administrators” → “1 to 1…3”;
- “an employee may not have a special pass, but if they do, there may not be more than five such special passes” → “1 to 0…5”;
- “a bicycle may not be rented by anyone, still each person may rent no more than ten bicycles” → “0…1 to 0…10”;

- “one to three system administrators may be responsible for each server, but each administrator may be responsible for no more than twenty servers” → “1…3 to 0…20”.

Technically[50] the control of compliance with the relationship cardinality constraint in this case is implemented with the help of triggers{341}. However, no less often such constraints are implemented not in the database, for convenience reasons, but in the applications working with it. This is especially true if the corresponding constraints are different for different applications, or if these constraints may change over time (e.g., “the main system administrator may arbitrarily change the maximum number of servers that his subordinates are responsible for”).

We have just considered cases where relationship cardinality is expressed by a range of values (e.g., “1 to 0…5”). A similar situation is also typical for “classical” relationship cardinalities, which can be represented as:

- “one to many” relationship: “1 to 1…M”, “1 to 0…M”, “0…1 to 1…M”, “0…1 to 0…M”, etc.
- “many to many” relationship: “1…M to 1…N”, “0…M to 1…N”, “1…M to 0…N”, “0…M to 0…N”, etc.
- “one to one” relationship: “1 to 1”, “1 to 0…1”, “0…1 to 1”, “0…1 to 0…1”.

It follows from this phenomenon that relationships for which the concept of “parent” and “child” tables is relevant (i.e., “one to many” and “one to one” relationships) have another property: they can be identifying and non-identifying.



Identifying relationship[51] — an association that connects two relations in such a way that any tuple of the child relation is always matched with a tuple of the parent relation, and the tuple of the child relation can be fully defined only in conjunction with the value of the primary key of the corresponding tuple of the parent relation.

Simplified: a record in a child table cannot exist without a corresponding record in the parent table (to guarantee this property, the foreign key is made part of the compound primary key of the child table); in the case of a sequential relationship, any child relation stores data about all parent relations.



Non-identifying relationship[52] — an association that connects two relations in such a way that a tuple of the child relation can correspond to one or zero tuple of the parent relation, and the tuple of the child relation can be fully defined without using the value of the primary key of the corresponding tuple of the parent relation.

Simplified: a record in a child table can exist without a corresponding record in the parent table; in the case of a sequential relationship, any child relation stores data about only one (closest) parent relation.



See “Fundamentals of Database Systems” (by Ramez Elmasri, Shamkant Navathe) book, 6th edition, chapter 7.5 “Weak entity types” for a detailed description of the logic of origin and application of such relationships.

[50] “Using MySQL, MS SQL Server and Oracle by examples” (Svyatoslav Kulikov), example 33 [https://svyatoslav.biz/database_book/]

[51] **Identifying relationship** — the relationship type that relates a weak entity type to its owner. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

[52] **Non-identifying relationship** — a regular association (relationship) between two independent classes. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

As an example of the use of an identifying relationship, consider the situation of a training course in a training center. Each course can be conducted many times (i.e., can have many instances), so in the database schema this part of the domain can be expressed by “one to many” relationship, and since each fact (instance) of conducting a course must be strictly linked to the corresponding “parent” course and can be defined only through this course, the relationship will be identifying.

The corresponding schema and data example are shown in figures 2.3.i and 2.3.j.

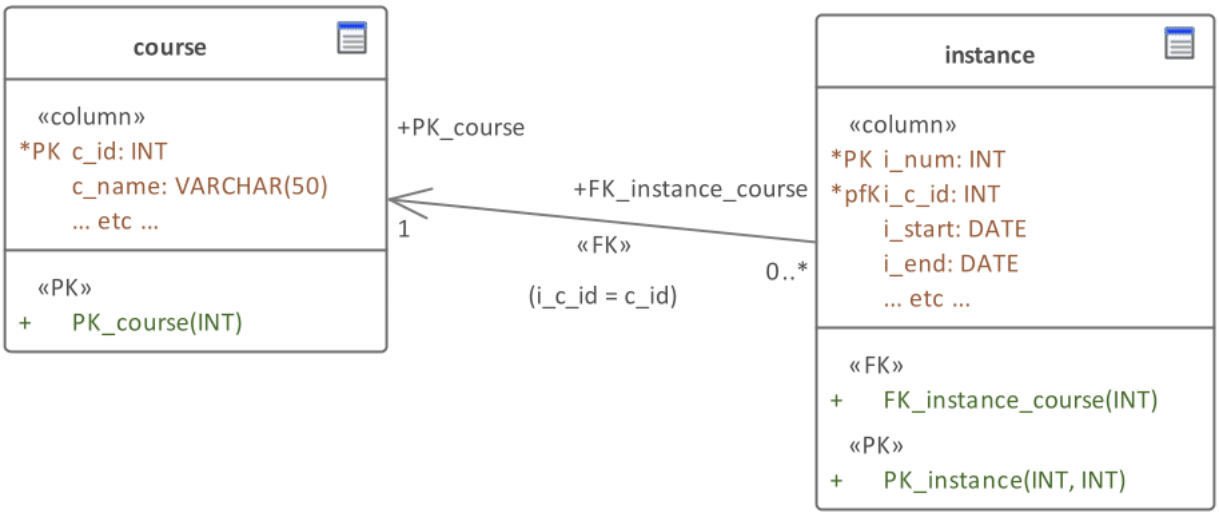


Figure 2.3.i — Schema of the identifying relationship

course

PK c_id	c_name	...
1	Java	...
2	.NET	...
3	C++	...
4	Python	...

instance

PK i_num	PK i_c_id FK	i_start	i_end	...
1	1	2017-01-10	2017-05-10	...
2	1	2017-07-20	2017-10-18	...
3	1	2018-03-05	2018-06-09	...
1	2	2017-01-02	2017-03-04	...
2	2	2017-08-17	2017-12-31	...
1	3	2017-01-01	2030-12-31	...

Figure 2.3.j — Data sample for the identifying relationship

So, to create an identifying relationship it is necessary to make the foreign key of the child relation a part of its primary key, i.e., the primary key of the child relation in case of identifying relationship will (almost[53]) always be compound one (and will include the entire primary key of the parent relation, even if the latter in turn is also compound).

[53] Except for “one to one” relationship, where a foreign key consisting of a single field can itself be the primary key.

There is an important technical feature to consider when forming identifying relationships: since in their classical implementation the foreign key is a part of the primary key, we get the familiar situation of a compound key{36}, in which the sequence of the fields{47} is important. It is worth putting the foreign key first, which will avoid the need to create a separate index on it.

The need to add the primary key of the parent relation to the primary key of the child relation in the case of several consecutive identifying “one to many” relationships leads to the fact that for relations at the “end of this chain” primary keys may consist of a large number of fields (see figure 2.3.k). In terms of performance, efficiency of data storage and usability of such a database, this situation raises a lot of questions. That is why “classical” identifying relationships are rarely used in practice.

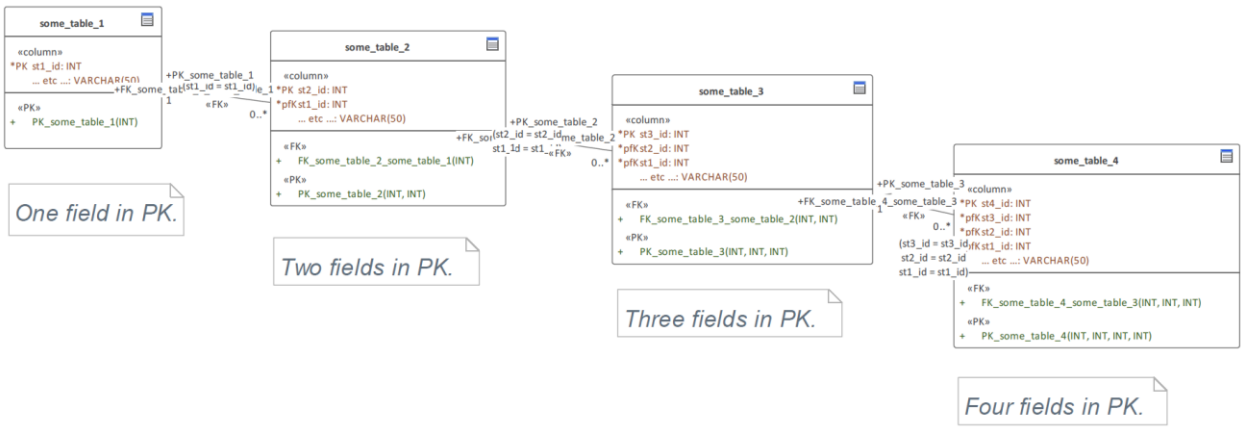


Figure 2.3.k — The problem of increasing the size of primary keys when using identifying relationships

However, such “key accumulation” has a positive side: at any moment we can “move along the chain” in any direction for any number of steps without intermediate operations. Let’s consider this with an example (figure 2.3.l).

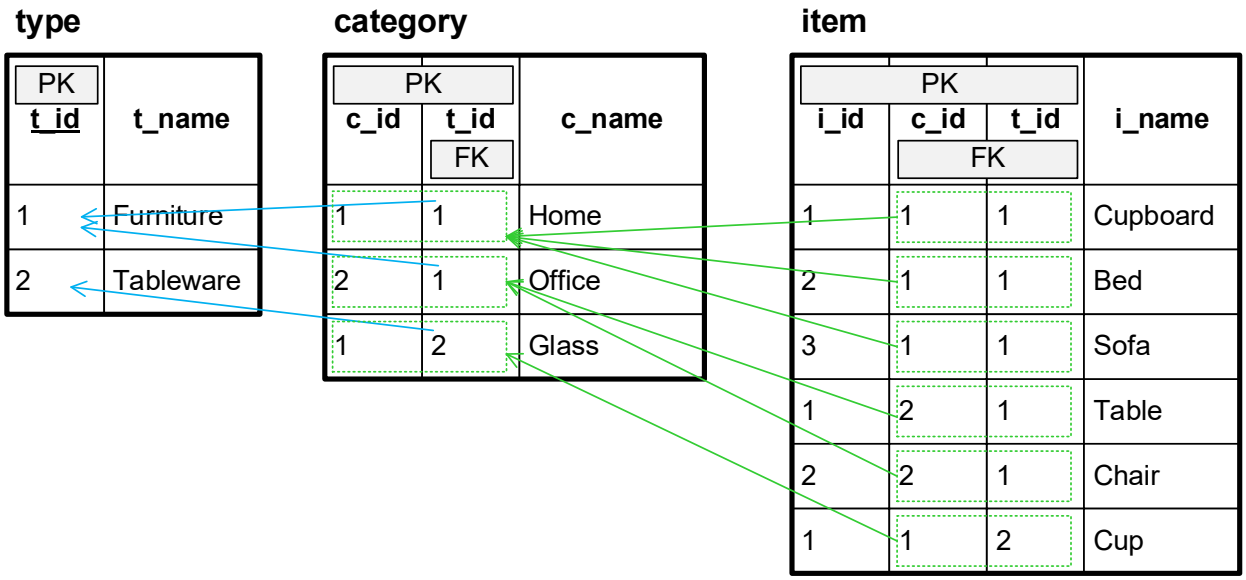


Figure 2.3.l — Data sample for the case of using several consecutive identifying relationships

To find out, for example, how many products belong to the first type (“Furniture”), we can immediately query the `item` table, using the value of its `t_id` field to determine the fact that some product belongs to the first type. Note that the intermediate table (`category`) is not needed here at all.

Similarly, if we need to find out what type the “Cup” product belongs to, we use the value of the `t_id` field of the `item` table and immediately query the `type` table based on it, without using the intermediate `category` table in any way.

Now let’s consider a non-identifying relationship. When it is organized, the primary key of the parent relation becomes “just a foreign key” and is not a part of the primary key of the child relation. The corresponding schema and data example are shown in figures 2.3.m and 2.3.n.

In this case, computers and rooms are quite independent entities in their own right (unlike instances of training of courses, which have no meaning in isolation from the courses themselves{61}), and the relationship shows only the fact that such-and-such computers are located in such-and-such rooms.

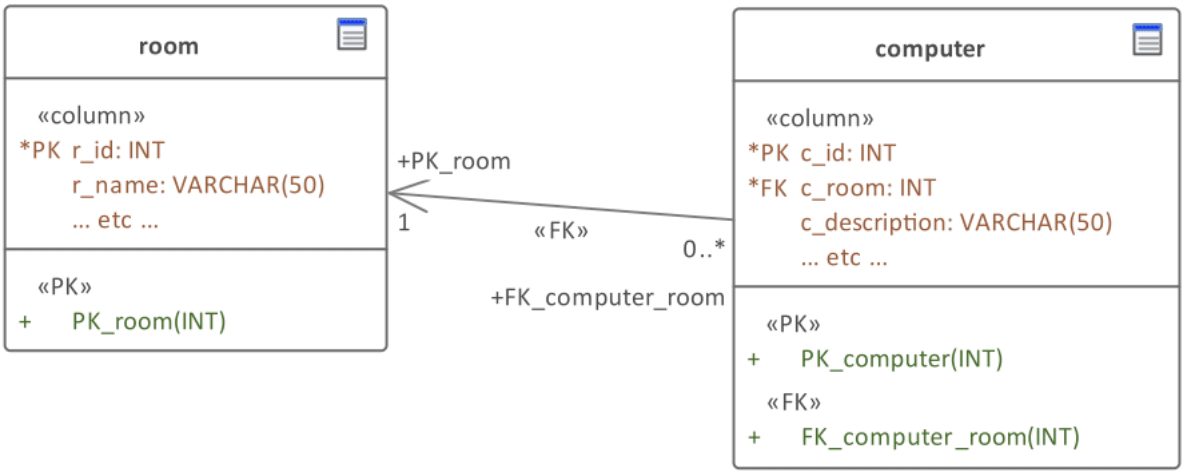


Figure 2.3.m — Schema of the non-identifying relationship

room

PK r_id	r_name	...
1	Room 1	...
2	Room 6	...
3	Room 7	...
4	Lab 12	...

computer

PK c_id	FK c_room	c_description	...
1	1	Computer-1	...
2	1	Computer-2	...
3	1	Computer-3	...
4	2	Computer-4	...
5	*NULL*	Computer-5	...

Figure 2.3.n — Data sample for the non-identifying relationship

As we don't need to make the foreign key of the child relation a part of its primary key to create a non-identifying relationship, we can painlessly create simple surrogate primary keys in child relations even in cases where several relations form a chain of relationships (see figure 2.3.o).

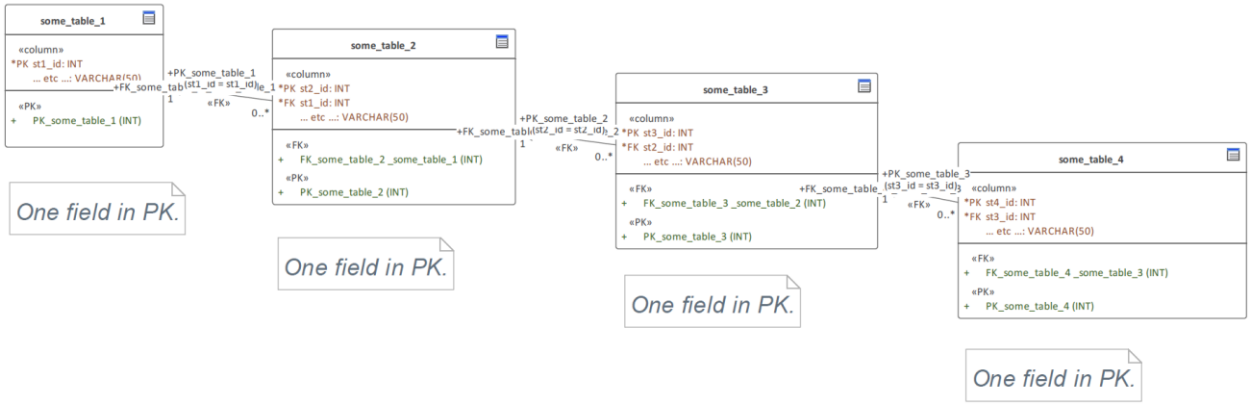


Figure 2.3.o — Absence of the problem of increasing the size of primary keys when using non-identifying relationships

It is easy to guess that the positive side of the "key accumulation" characteristic of identifying relationships becomes negative here: we can "move along the chain" only one step in either direction. If, however, there is an "intermediate link" (or several links) between the relations of interest, we are obliged to use such intermediate relations to obtain the information of interest. Let's consider this with an example (figure 2.3.p).

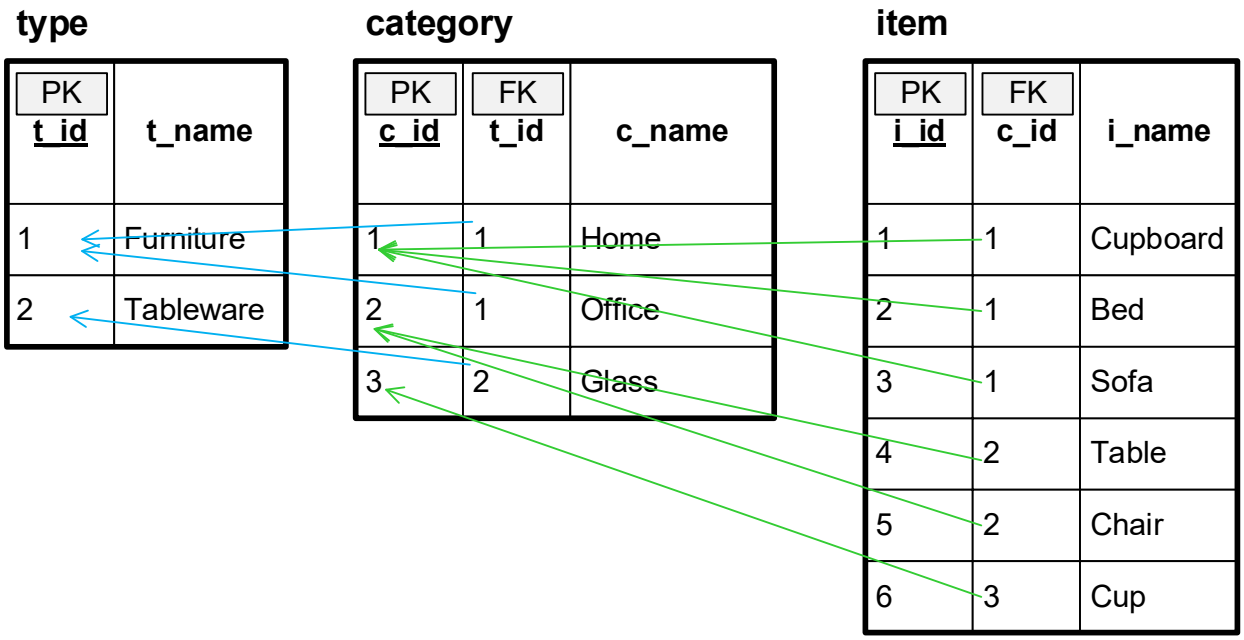


Figure 2.3.p — Data sample for the case of using several consecutive non-identifying relationships

To find out, for example, how many products belong to the first type ("Furniture"), we cannot immediately query the `item` table, because it has no information about product types (it only has information about product categories). We must first determine all product categories that belong to the type we are interested in, then for each received category determine the number of products that belong to it, and only after that, having summed up the received amounts of products, we will know the answer to the original question.

Similarly, if we need to find out what type of product “Cup” refers to, we must first determine the category of this product, and then (based on information about what type of goods the category found out belongs to) we can get information about the type of goods we are interested in.

Another key difference between the identifying and non-identifying relationship is that, in the second case, a record in the child table can be “not linked” to any record in the parent table.

In the case of the identifying relationship, this situation is completely excluded, because the use of `NULL` values in the fields that are part of the primary key is prohibited. Therefore, any record in the child table in the foreign key (which is part of its primary key) will necessarily store some specific value of the primary key of the parent table.

In the case of a non-identifying relationship, there is no such limitation, and we can store `NULL` values in the foreign key as a sign that the “parent” of the record does not exist.

Let’s refine the example shown in figure 2.3.n: let’s imagine that computers “Computer-2” and “Computer-5” are not yet put in any room (for example, they are just stored in the warehouse for now). We get the situation presented in figure 2.3.q:

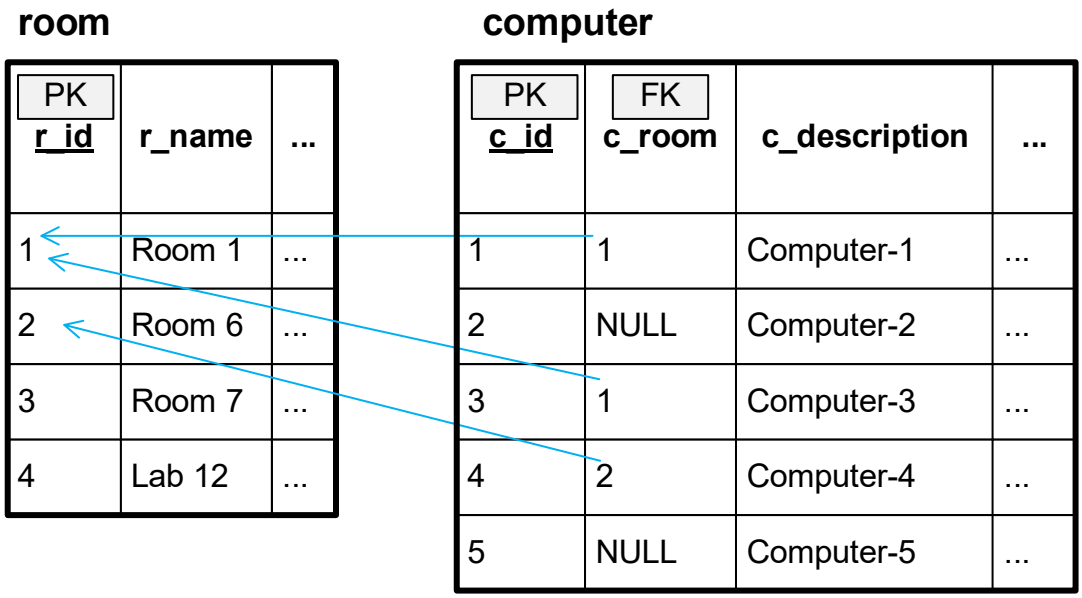


Figure 2.3.q — Example of using `NULL` values for non-identifying relationships

In some cases, on the contrary, we need to exclude the possibility of placing records in the child table without explicitly specifying the corresponding records from the parent table (say, by law, computers located in the warehouse must be clearly assigned to the room named “Warehouse”). Then on the foreign key of the child table `NOT NULL` constraint is activated, prescribing the DBMS to prohibit attempts to insert `NULL` values there.



In practice, it is the creation of non-identifying relationships with permission or prohibition of insertion of `NULL` values in foreign keys that is used most often: even when such a solution seems questionable in terms of relational theory. It’s all about ease of data management and the absence of duplication of fields (which on large amounts of data can lead to a tangible increase in the volume of the database).

In summary, if we compare identifying and non-identifying relationships, we get the following picture:

Identifying relationship	Non-identifying relationship
Used when the child relation is “incomplete” without the parent relation.	Used when it is necessary to logically combine self-sufficient relations.
The primary key of the parent relation becomes the foreign key of the child relation and is included in its primary key.	The primary key of the parent relation becomes the foreign key of the child relation but is **not a part** of its primary key.
A record in the child relation cannot exist without a corresponding record in the parent relation.	A record in the child relation can exist without a corresponding record in the parent relation. If this situation is undesirable (or unacceptable), it can be eliminated by defining a `NOT NULL` constraint on the foreign key.
In the case of a “chain of relationships”, the primary key becomes larger and larger in each successive child relation.	In the case of a “chain of relationships”, in each subsequent child relation the primary key does not become larger.
In the case of a “chain of relationships”, for any tuple of any parent relation, we can immediately get a list of tuples from any child relation. And for any tuple of any child relation, we can immediately get information about the parent tuple of any level.	In the case of a “chain of relationships”, to determine the parent element or a list of children, you will most often have to perform an intermediate `JOIN` operation.
In those graphical notations that support this distinction, it is represented by a solid line.	In those graphical notations that support this distinction, it is represented by a dotted line.

This concludes the theoretical part about relationships as such, and now we consider another topic that deserves its own chapter, because it would not be an exaggeration to say that (mostly, yet not only) for the sake of this feature relational databases were invented.



Task 2.3.a: using any database design tool, implement 5-10 fragments of schemas using each “one to many” and “many to many relationships”, and think of at least one example for a “one to one” relationship.



Task 2.3.b: are all relationships in the “Bank{395}” database correct? Are there any necessary relationships missing? If you think that this database schema needs improvement, make the appropriate changes.



Task 2.3.c: is it possible to replace some of “many to many” relationships in the “Bank{395}” database with “one to many” relationships? If you think “yes”, please edit the schema accordingly.

2.3.2. Referential Integrity and Database Consistency



Referential integrity[54] — a property of a relational database that consists in strict adherence to the rule: if the foreign key of a child relation contains some value, this value must necessarily be present in the primary key of the parent relation.

Simplified: a record in the child table cannot refer to a nonexisting record in the parent table.



Database consistency[55] — a property of a relational database that consists of strict adherence, at any given time, to all constraints set implicitly by the relational model or explicitly by a specific database schema.

Simplified: the DBMS controls data types, referential integrity, `CHECK`{338} constraints, triggers{341} execution, queries correctness (validity), etc., etc. at any given time.

Let's start with referential integrity.

Once a relationship is explicitly established between the two relations, the DBMS checks adherence to following rule when modifying data in the corresponding tables: if the foreign key of the child relation contains some value, this value must be present in the primary key of the parent relation.

I.e., the DBMS will not allow to:

- add to the child table a record with the value of the foreign key, which is not among the values of the primary key of the parent table;
- add to the child table a record with the value of the foreign key equal to `NULL`, if this foreign key constraint is set to `NOT NULL`;
- change the foreign key value of a record in the child table to a value that is not among the values of the primary key of the parent table;
- change the value of the foreign key in the child table to `NULL`, if the `NOT NULL` constraint is set on this foreign key;
- delete from the parent table a record whose primary key value is among the values of the child table's foreign key (if the corresponding cascade operation{68} is enabled);
- change the primary key value of a record in the parent table if that value is among the values of the child table's foreign key (if the corresponding cascade operation{68} is enabled);
- violate the constraint on relationship cardinality, implemented through triggers[56].

Let's schematically depict the main idea in figure 2.3.r.

[54] **Referential integrity** — the rule that no referencing tuple is allowed to exist if the corresponding referenced tuple doesn't also exist. ("The New Relational Database Dictionary", C.J. Date)

[55] **Database consistency** — … a database is in a state of consistency if and only if it conforms to all declared integrity constraints (database constraints and type constraints). ("The New Relational Database Dictionary", C.J. Date)

[56] "Using MySQL, MS SQL Server and Oracle by examples" (Svyatoslav Kulikov), example 33 [https://svyatoslav.biz/database_book/]

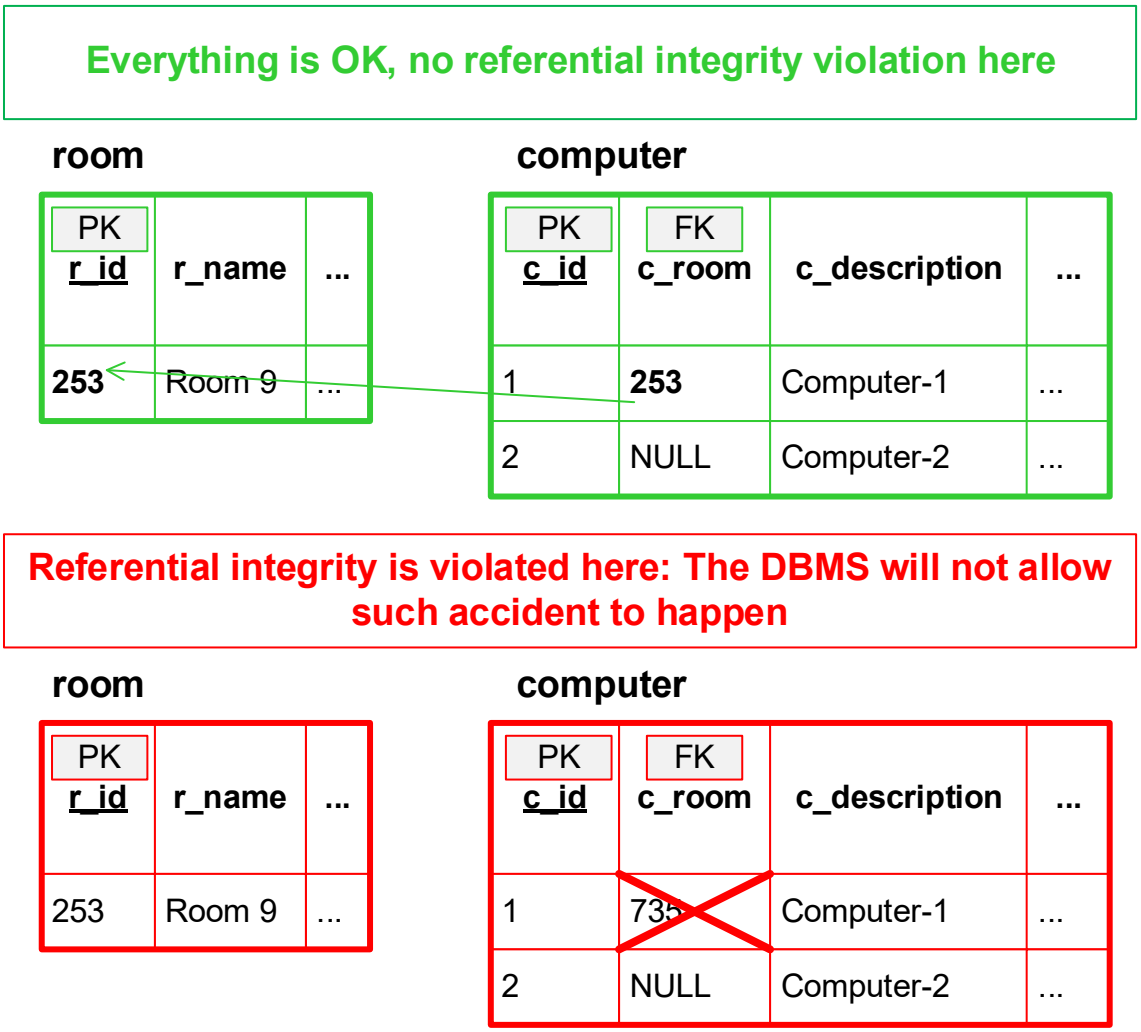


Figure 2.3.r — The main idea of referential integrity



It is critical to understand that DBMS has no “telepathy” and cannot “guess” which values should be compared if the relationships between tables are not **explicitly** made.

Sometimes, for compatibility purposes with different DBMSes, or for other reasons, relationships between tables may not exist (although they are needed), and the appropriate checks are assigned to the application working with the database. Although such a solution is justified in some rare cases, it is extremely risky.

So far, we have considered cases of data modification in a child table — here everything is simple enough, because the DBMS only needs to compare the value of the child table’s foreign key with the values of the parent table’s primary key. Even though this operation can sometimes be quite costly, its algorithm is trivial.

But what happens if a modification operation is executed on the parent table? Then the DBMS implements one or another cascade operation (specified when configuring the relationship).

There are the following types of cascade operations (let’s represent them all at once in one figure to simplify perception and memorization — see figure 2.3.s).

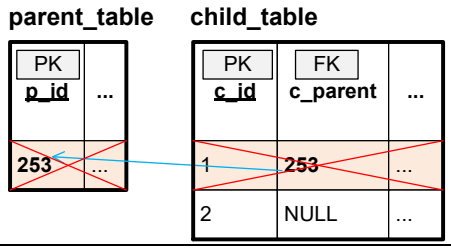
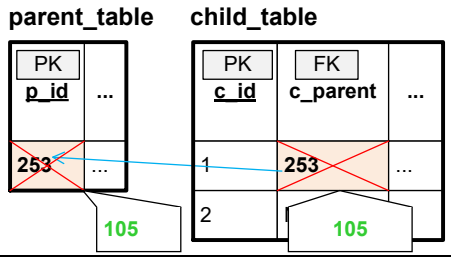
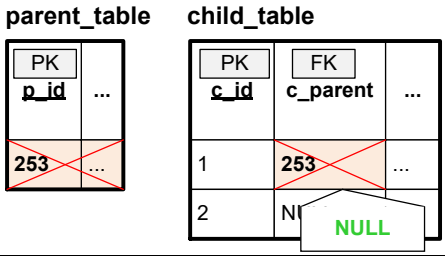
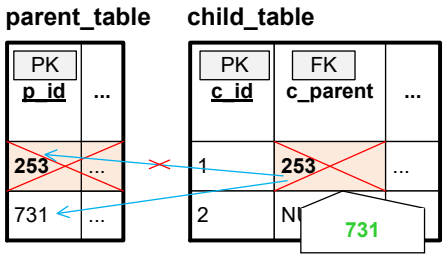
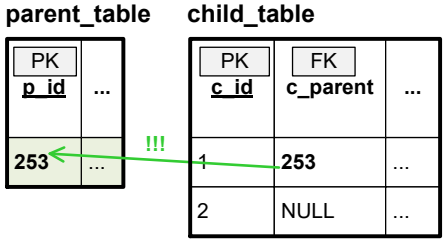
Name and essence of the operation	Schematic explanation	Subject area sample
Cascade deletion. When we delete a record from the parent table, all related records from the child table are automatically deleted.		It is a rule on some news site to delete all related news when a news channel (section) is deleted.
Cascade update. When we change the primary key value of a record in the parent table, the old value automatically changes to the new value in the foreign key of all corresponding records of the child table.		The owner of the payment is identified by their passport id. If the user changes their passport, in all of their payments the old passport id will have to be changed to the new passport id.
Setting empty keys. When we delete a record from the parent table, the old value in the foreign key of all corresponding records of the child table automatically changes to `NULL`.		When a lease contract is terminated (lessee information is deleted), the room becomes vacant (no lessee is specified, i.e., `NULL`).
Setting a default value. When we delete a record from the parent table, the old value in the foreign key of all corresponding records of the child table automatically changes to a new value (set beforehand or calculated directly during the operation).		If a call center employee is dismissed, all telephone numbers assigned to them are transferred to the head of the unit.
Prohibition of cascade operation. There can be a cascade deletion prohibition and a cascade update prohibition. The DBMS will not allow to delete a record from the parent table (or change its primary key value) whose primary key value is contained in the foreign key of at least one entry of the child table.		It is not possible to delete a product category in some online store if at least one product is assigned to it.

Figure 2.3.s — All types of cascade operations

Since there are so many misunderstandings associated with cascade operations, let's consider a few important points at once.

1. Any cascade operation is activated only when we modify data in the **parent** table. Never any data changes in the child table can trigger a cascade operation.
2. Any cascade operation is activated only if data modifications in the parent table affect the primary key. The primary key is guaranteed to be affected only by a record deletion operation (because it is impossible to delete "a part of a record"). If other fields that are not part of the primary key are modified, the cascade operation is not triggered.
3. Data insertion (into any table, no matter parent or child) and data selection (also no matter where from) never activates any cascade operations.
4. Not all DBMSes "know" how to perform all kinds of cascade operations "on their own". If the DBMS you are using does not support one or another operation, you can achieve the necessary functionality by using triggers{341}.
5. Almost all cascade operations are mutually exclusive (except "cascade update", which is "compatible" with all other operations), and therefore several different cascade operations cannot be enabled on some single relationship (for example, **both** "cascade delete" **and** "setting a default value" simultaneously[57])



Very often one may hear a question of the following kind: "Which cascade operation is the right one to perform here?" (And the person asking just draws a schema of two relations.) The only honest and correct answer is — any. The choice of cascade operation depends **only** on the subject area.

Let's demonstrate it by an example of a news site that has news sections (parent table) and the news themselves (child table)):

- if the rules of this site require deleting all relevant news upon a section deletion, we need to enable "cascade delete";
- if the rules of this site require that upon news section deletion all the relevant news should be moved to the "no section" section, we need to enable "setting empty keys";
- if the rules of this site require that upon news section deletion all relevant news should be moved to a predefined section (such as "Miscellaneous"), we must enable "setting a default value";
- if the rules of this site prohibit the deletion of a section in which there is at least one news, we need to enable "prohibition of cascade deletion".

In other words: the DBMS does not care at all what to do with your data. It is equally ready to perform any permissible operation. But it is up to you, as the creator of the database, to decide what operation to perform.

At the beginning of this chapter, we mentioned two definitions: in addition to referential integrity, the DBMS is also concerned with ensuring database consistency{67}.

In fact, maintaining referential integrity is one of the many components of database consistency (some other components are given in the definition). But since this chapter is about relationships and all the benefits that come automatically from their creation, it is very important to emphasize again that the DBMS can only control those constraints that come from the very nature of relational databases (for example, you cannot insert data into a table by specifying incorrect field names) and those that you as the creator of a particular database have **explicitly** specified: relationships, triggers, etc.

What a DBMS cannot do is "guess" your thoughts. Let's explain with an example of a very common error — data coherence violation.

[57] It is theoretically possible to create such a trigger{351} that will "choose" which cascade operation to perform at the moment based on some data (and it will look as if several different mutually exclusive cascade operations are implemented on the same relationship). But without using triggers, none of the DBMSes allow implementing such behavior.

Suppose we have a database of some news site, where there are a lot of news sections, and in each section a lot of news itself. To display a list of sections on the site, the customer wants to display the number of news items in each section and the date-time of publication of the most recent news.

There are two solutions to this task:

- each time we form a list of sections we count the amount of news in each of them, as well as we determine the date-time of the most recent news;
- create two additional fields in the table that stores information about sections (to store the amount of news in the section and the date-time of the most recent news publication); and when forming the list of sections, we immediately take ready-made data from these fields — see figure 2.3.t.

The first option is much easier to implement and does not require any modifications on the database schema, but it leads to a very large loss of performance[58]. The second option will work much faster but requires some tuning of the database.

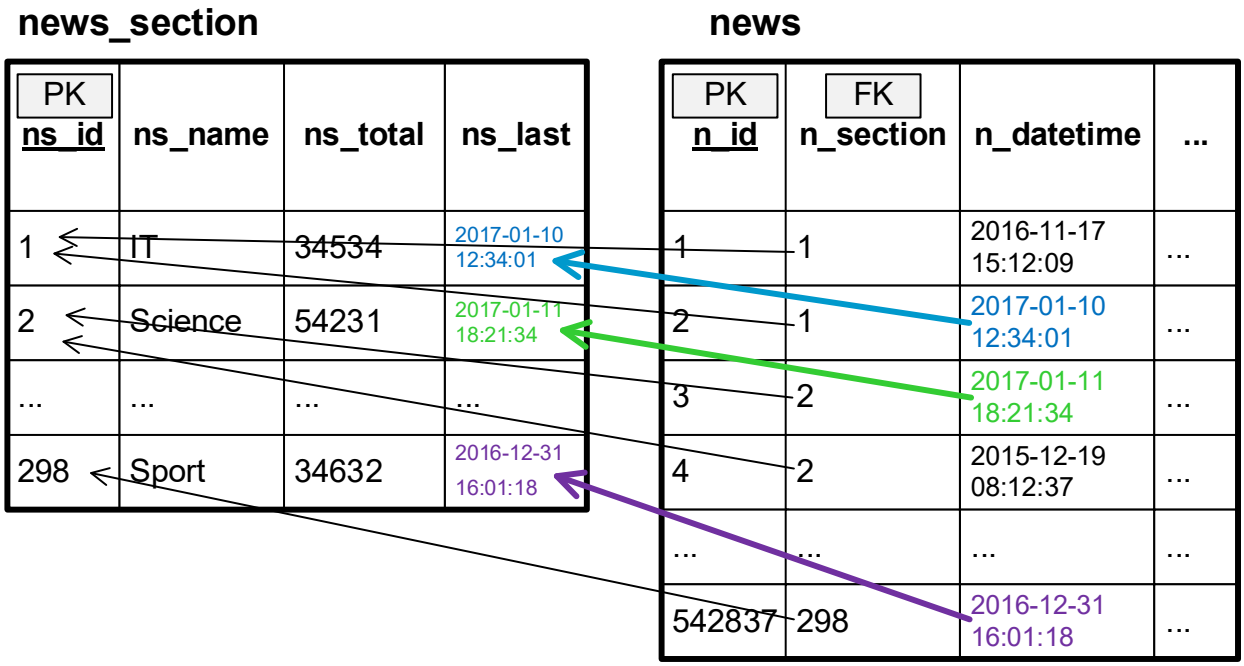


Figure 2.3.t — Database fragment with aggregating and caching fields

The `ns_total` field contains the results of counting the number of news items for each section — such fields are often called aggregating ones, i.e., containing the results of some kind of data processing (generalization, aggregation).

The `ns_last` field contains information about the date-time of publication of the latest news in each section — such fields are often called caching ones (as opposed to aggregating, they simply contain a copy of the data, which will take much less time to access than if you extract the same data from their primary storage).

The complexity of this solution is that DBMS (without additional database modification) does not “understand” that values of `ns_total` and `ns_last` fields of `news_section` table are somehow related to data stored in `news` table. And therefore, nothing prevents the values of these fields to be incorrect (i.e., not corresponding to the actual amount of news in each section and the date-time of last news publication per section).

[58] The difference in performance between the first and second option can be as much as a million times or more.

To eliminate the possibility of such coherence violation, we need to create several triggers{341} on the `news` table. Those triggers will be automatically executed during data insertion, update and deletion and will update the corresponding values in the `news_section` table.

A simplified example of such a situation will be discussed later{96}. A detailed solution for a similar problem can also be found in the book on the practical use of databases[59].

So, that's it for the theory — now let's look at the relationships in practice.



Task 2.3.d: for all the schemas you created in task 2.3.a{66} investigate the possibility and necessity of applying aggregation and caching fields, and make appropriate edits to the schemas.



Task 2.3.e: how can the performance of the "Bank{395}" database be improved by using aggregation and caching fields? Make appropriate edits to the schema.



Task 2.3.f: which cascade operations should be set on the relationships in the "Bank{395}" database? Make appropriate edits to the schema.

[59] "Using MySQL, MS SQL Server and Oracle by examples" (Svyatoslav Kulikov), example 32 [https://svyatoslav.biz/database_book/]

2.3.3. Creating and Using the Relationships

As we have said many times before, the primary key{36} of a parent table and the foreign key{43} of a child table, the implementation of which we discussed in the corresponding section{46} are used to organize the relationships.

Let's begin our consideration of the technical implementation of "one to many" relationships with the example shown earlier in figures 2.3.a{54} and 2.3.b{54}.

The code for generating the corresponding fragment of the database schema looks as follows (we provide it for three different DBMS).

MySQL Example of code for figures 2.3.a and 2.3.b

```
CREATE TABLE `payment`
(
 `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `person` INT UNSIGNED NOT NULL,
 `money` DECIMAL(10,2) NOT NULL,
 CONSTRAINT `PK_payment` PRIMARY KEY (`id`)
);

CREATE TABLE `employee`
(
 `id` INT UNSIGNED NOT NULL AUTO_INCREMENT,
 `passport` CHAR(9) NOT NULL,
 `name` VARCHAR(50) NOT NULL,
 CONSTRAINT `PK_employee` PRIMARY KEY (`id`)
);

ALTER TABLE `employee`
 ADD CONSTRAINT `UQ_passport` UNIQUE (`passport`);

ALTER TABLE `payment` ADD CONSTRAINT `FK_payment_employee`
  FOREIGN KEY (`person`) REFERENCES `employee` (`id`)
   ON DELETE CASCADE ON UPDATE RESTRICT;
```

MS SQL Example of code for figures 2.3.a and 2.3.b

```
CREATE TABLE [payment]
(
 [id] INT NOT NULL IDENTITY (1, 1),
 [person] INT NOT NULL,
 [money] MONEY NOT NULL
);

CREATE TABLE [employee]
(
 [id] INT NOT NULL IDENTITY (1, 1),
 [passport] CHAR(9) NOT NULL,
 [name] NVARCHAR(50) NOT NULL
);

ALTER TABLE [payment]
 ADD CONSTRAINT [PK_payment]
  PRIMARY KEY CLUSTERED ([id]);

ALTER TABLE [employee]
 ADD CONSTRAINT [PK_employee]
  PRIMARY KEY CLUSTERED ([id]);

ALTER TABLE [employee]
 ADD CONSTRAINT [UQ_passport] UNIQUE NONCLUSTERED ([passport]);

ALTER TABLE [payment] ADD CONSTRAINT [FK_payment_employee]
 FOREIGN KEY ([person]) REFERENCES [employee] ([id]) ON DELETE CASCADE;
```

Oracle | Example of code for figures 2.3.a and 2.3.b

```
CREATE TABLE "payment"
(
 "id" NUMBER(38) NOT NULL,
 "person" NUMBER(38) NOT NULL,
 "money" NUMBER(15,4) NOT NULL
);

CREATE TABLE "employee"
(
 "id" NUMBER(38) NOT NULL,
 "passport" CHAR(9) NOT NULL,
 "name" NVARCHAR2(50) NOT NULL
);

CREATE SEQUENCE "SEQ_payment_id"
 INCREMENT BY 1
 START WITH 1
 NOMAXVALUE
 MINVALUE 1;

CREATE OR REPLACE TRIGGER "TRG_payment_id"
 BEFORE INSERT
 ON "payment"
 FOR EACH ROW
 BEGIN
  SELECT "SEQ_payment_id".NEXTVAL
  INTO :NEW."id"
  FROM DUAL;
 END;

CREATE SEQUENCE "SEQ_employee_id"
 INCREMENT BY 1
 START WITH 1
 NOMAXVALUE
 MINVALUE 1;

CREATE OR REPLACE TRIGGER "TRG_employee_id"
 BEFORE INSERT
 ON "employee"
 FOR EACH ROW
 BEGIN
  SELECT "SEQ_employee_id".NEXTVAL
  INTO :NEW."id"
  FROM DUAL;
 END;

ALTER TABLE "payment"
 ADD CONSTRAINT "PK_payment"
  PRIMARY KEY ("id") USING INDEX;

ALTER TABLE "employee"
 ADD CONSTRAINT "PK_employee"
  PRIMARY KEY ("id") USING INDEX;

ALTER TABLE "employee"
 ADD CONSTRAINT "UQ_passport" UNIQUE ("passport") USING INDEX;

ALTER TABLE "payment"
 ADD CONSTRAINT "FK_payment_employee"
 FOREIGN KEY ("person") REFERENCES "employee" ("id") ON DELETE CASCADE;
```

In the future, we will try to keep examples in MySQL syntax (as the most compact), but now let's take a closer look at the code, because it contains parts that are equally relevant to the topics of creating tables, keys, and relationships.

Action	MySQL	MS SQL Server	Oracle
Creating tables	Lines 1–7 and 9–15	Lines 1–6 and 8–13	Lines 1–6 and 8–13
Specifying primary keys	Lines 3, 6, 11, 14 — combined with table creation	Lines 15–17 and 19–21	Lines 47–49 and 51–53
Specifying the fact that primary keys are auto-incrementable	Lines 3 and 11 — combined with table creation	Lines 3 and 10 — in MS SQL Server the primary key has to be an identity-field to achieve the desired effect	Lines 15–45 — in Oracle to achieve the desired effect one must create a sequence and use its values with a trigger to produce values for the primary key
Specifying the fact that "passport" field values must be unique	Lines 17–18	Lines 23–24	Lines 55–56
Creating a "one to many" relationship	Lines 20–22	Lines 26–27	Lines 58–60

Note that only MySQL syntax requires prohibition of cascade updates (the `ON UPDATE RESTRICT` relationship property) to be defined explicitly. MS SQL Server has this behavior (called `NO ACTION`) enabled by default and does not require it to be defined explicitly[60], and Oracle does not support cascade updates at all (yet they can be emulated by triggers).

Now let's see what we've got by creating the relationship.

First, let's fill the `employee` table with test data:

id	passport	name
1	AA1231234	Smith
2	BB1231234	Taylor
3	CC1231234	Jones

MySQL — Example of code for test data insertion into the `employee` table

```
1    INSERT INTO `employee`
2                (`passport`, `name`)
3    VALUES      ('AA1231234', 'Smith'),
4                ('BB1231234', 'Taylor'),
5                ('CC1231234', 'Jones')
```

MS SQL — Example of code for test data insertion into the `employee` table

```
1    INSERT INTO [employee]
2                ([passport], [name])
3    VALUES      ('AA1231234', N'Smith'),
4                ('BB1231234', N'Taylor'),
5                ('CC1231234', N'Jones')
```

[60] Strictly speaking, MySQL also uses NO ACTION by default, but historically it has been customary to write RESTRICT explicitly.

Oracle — Example of code for test data insertion into the `employee` table

```
INSERT ALL
 INTO "employee" ("passport", "name") VALUES ('AA1231234', N'Smith')
 INTO "employee" ("passport", "name") VALUES ('BB1231234', N'Taylor')
 INTO "employee" ("passport", "name") VALUES ('CC1231234', N'Jones')
SELECT 1 FROM "DUAL"
```

Now let's fill the `payment` table with test data:

id	person	money
1	1	100.00
2	2	400.00
3	1	150.00
4	2	800.00
5	2	900.00

MySQL — Example of code for test data insertion into the `payment` table

```
INSERT INTO `payment`
            (`person`, `money`)
VALUES      (1, 100),
            (2, 400),
            (1, 150),
            (2, 800),
            (2, 900)
```

MS SQL — Example of code for test data insertion into the `payment` table

```
INSERT INTO [payment]
            ([person], [money])
VALUES      (1, 100),
            (2, 400),
            (1, 150),
            (2, 800),
            (2, 900)
```

Oracle — Example of code for test data insertion into the `payment` table

```
INSERT ALL
 INTO "payment" ("person", "money") VALUES (1, 100)
 INTO "payment" ("person", "money") VALUES (2, 400)
 INTO "payment" ("person", "money") VALUES (1, 150)
 INTO "payment" ("person", "money") VALUES (2, 800)
 INTO "payment" ("person", "money") VALUES (2, 900)
SELECT 1 FROM "DUAL"
```

So far, all queries have been successful. But what happens if we try to insert data that violates referential integrity{67}? Let's test this by attempting to insert "4" value in the `person` field (no such employee exists) and `NULL` value (it's also forbidden because this field has a `NOT NULL` constraint).

The code below has a comment after each query; that comment contains an error message returned by the DBMS in response to our actions.

MySQL — Referential integrity violation attempt

```
INSERT INTO `payment`
            (`person`, `money`)
VALUES      (4, 999)
-- Error Code: 1452. Cannot add or update a child row: a foreign key
-- constraint fails

INSERT INTO `payment`
            (`person`, `money`)
VALUES      (NULL, 999)
-- Error Code: 1048. Column 'person' cannot be null
```

MS SQL — Referential integrity violation attempt

```
INSERT INTO [payment]
            ([person], [money])
VALUES      (4, 999)
-- Msg 547, Level 16, State 0, Line 1
-- The INSERT statement conflicted with the FOREIGN KEY constraint
-- "FK_payment_employee".
-- The statement has been terminated.

INSERT INTO [payment]
            ([person], [money])
VALUES      (NULL, 100)
-- Msg 515, Level 16, State 2, Line 1
-- Cannot insert the value NULL into column 'person', table 'payment';
-- column does not allow nulls. INSERT fails.
-- The statement has been terminated.
```

Oracle — Referential integrity violation attempt

```
INSERT INTO "payment"
            ("person", "money")
VALUES      (4, 999)
-- Error starting at line : 92 in command ...
-- SQL Error: ORA-02291: integrity constraint (FK_payment_employee)
-- violated - parent key not found
-- *Cause:    A foreign key value has no matching primary key value.
-- *Action:   Delete the foreign key or add a matching primary key.

INSERT INTO "payment"
            ("person", "money")
VALUES      (NULL, 999)
-- Error starting at line : 96 in command ...
-- SQL Error: ORA-01400: cannot insert NULL into ("payment"."person")
-- *Cause:    An attempt was made to insert NULL into previously
-- listed objects.
-- *Action:   These objects cannot accept NULL values.
```

So, all three DBMS have successfully prevented an attempt to perform operations that violate the constraints of the database schema.

We have dealt with the insertion. Now let's try to "break" the database by updating and deleting data.

First let's try to change the value of the primary key for the record in parent table with `id`=1 — this operation should be forbidden (cascade update is restricted in relationship properties).

Then let's try to delete the record with `id`=2 — this operation should be successful, and at that all payments of the employee with `id`=2 should be automatically deleted from `payment` table.

For comparison, let's perform the same operations with the record with `id`=3 (this employee has no payments, so the update of his `id` should be successful, and deleting his record will not lead to any additional changes in the database).

MySQL — Demonstration of cascade operations

```mysql
UPDATE `employee`
SET    `id` = 7
WHERE  `id` = 1
-- Error Code: 1451. Cannot delete or update a parent row: a foreign key
-- constraint fails

DELETE FROM `employee`
WHERE  `id` = 2
-- 1 row(s) affected

UPDATE `employee`
SET    `id` = 9
WHERE  `id` = 3
-- 1 row(s) affected

DELETE FROM `employee`
WHERE  `id` = 9
-- 1 row(s) affected
```

MS SQL — Demonstration of cascade operations

```mssql
UPDATE [employee]
SET    [id] = 7
WHERE  [id] = 1
-- As [id] is an identity-field, the
-- error message will be as follows:
-- Msg 8102, Level 16, State 1, Line 1
-- Cannot update identity column 'id'.

-- If we remove the identity property from the [id] field, the
-- error message will be as follows:
-- The UPDATE statement conflicted with the REFERENCE constraint
-- "FK_payment_employee".
-- The statement has been terminated.

DELETE FROM [employee]
WHERE  [id] = 2
-- (1 row(s) affected)

UPDATE [employee]
SET    [id] = 9
WHERE  [id] = 3
-- (1 row(s) affected)

DELETE FROM [employee]
WHERE  [id] = 9
-- (1 row(s) affected)
```

Oracle — Demonstration of cascade operations

```
1     UPDATE "employee"
2     SET    "id" = 7
3     WHERE  "id" = 1
4     -- Error starting at line : 108 in command ...
5     -- SQL Error: ORA-02292: integrity constraint (FK_payment_employee)
6     -- violated - child record found
7     -- *Cause: attempted to delete a parent key value that had a foreign
8                dependency.
9     -- *Action:   delete dependencies first then parent or disable constraint.
10
11    DELETE FROM "employee"
12    WHERE  "id" = 2
13    -- 1 rows deleted.
14
15    UPDATE "employee"
16    SET    "id" = 9
17    WHERE  "id" = 3
18    -- 1 rows updated.
19
20    DELETE FROM "employee"
21    WHERE  "id" = 9
22    -- 1 rows deleted.
```

So, all three DBMSes behave almost identically (and quite logically, in full accordance with theoretical expectations). Only MS SQL Server has a feature related to how it implements autoincrementable primary keys: it is forbidden to update `identity`-fields, so we had to disable corresponding property of `id` field to make sure cascade update is forbidden. Note that in reality this problem almost never occurs, because the `identity`-field is artificial (it doesn't make any sense for the subject matter area), so there's no need to update it.

After performing all the operations shown above, some of the data has been deleted and our database is left with only:

Data in the `employee` table

id	passport	Name
1	AA1231234	Smith

Data in the `payment` table

id	person	Money
1	1	100.00
3	1	150.00

Let's restore the original data sets and see how sample queries work when the values we are interested in are in different tables.

Data in the `employee` table

id	passport	name
1	AA1231234	Smith
2	BB1231234	Taylor
3	CC1231234	Jones

Data in the `payment` table

id	person	money
1	1	100.00
2	2	400.00
3	1	150.00
4	2	800.00
5	2	900.00

In this situation, the most commonly used queries are **JOIN** queries or queries with subqueries. Let's demonstrate both options[61].

Suppose we want to get the list of employees and the sum of payments for each of them. The expected result will look like this:

name	sum
Smith	250.00
Taylor	2100.00
Jones	NULL

The **NULL** value for Jones (to whom not a single payment applies) here is perfectly correct and even more advantageous than "0" (although zero is not hard to get) — this way we emphasize the fact that there were no payments, not that their sum is zero.

So, the SQL queries code to get this result will look like this:

MySQL — Example of a JOIN SQL query for two tables

```mysql
SELECT `name`,
       SUM(`money`) AS `sum`
FROM   `employee`
       LEFT OUTER JOIN `payment`
                    ON `employee`.`id` = `payment`.`person`
GROUP  BY `employee`.`id`
```

MS SQL — Example of a JOIN SQL query for two tables

```mssql
SELECT [name],
       SUM([money]) AS [sum]
FROM   [employee]
       LEFT OUTER JOIN [payment]
                    ON [employee].[id] = [payment].[person]
GROUP  BY [employee].[id],
          [employee].[name]
```

Oracle — Example of a JOIN SQL query for two tables

```oracle
SELECT "name",
       SUM("money") AS "sum"
FROM   "employee"
       LEFT OUTER JOIN "payment"
                    ON "employee"."id" = "payment"."person"
GROUP  BY "employee"."id",
          "employee"."name"
```

Figure 2.3.u schematically shows the DBMS logic for such queries.

If we describe it in words, it turns out:

- The DBMS takes all records from the **employee** table and searches for correspondences from the **payment** table based on the equality of the **id** (from the **employee** table) and **person** (from the **payment** table) fields.
- If no match is found for any of the entries in the **employee** table, **NULL** values are substituted for the data from the **payment** table.
- Having received a set of data matched from two tables (in figure 2.3.u the corresponding information is presented in the "LEFT OUTER JOIN result" block), the DBMS performs summation of payment values within groups of records with the same **id** field value.
- After performing this operation, we get the final result (in figure 2.3.u the relevant information is presented in the "SUM and GROUP BY result" block).

[61] The SQL language is beyond the scope of this book, but you can read a huge number of detailed examples in the "Using MySQL, MS SQL Server and Oracle by Examples" book (Svyatoslav Kulikov). [https://svyatoslav.biz/database_book/]

Initial (source) data

employee

PK **id**	passport	name
1	AA1231234	Smith
2	BB1231234	Taylor
3	CC1231234	Jones

payment

PK **id**	FK **person**	money
1	1	100
2	2	400
3	1	150
4	2	800
5	2	900

LEFT OUTER JOIN result

id	person	name	money
1	1	Smith	100
2	2	Taylor	400
1	1	Smith	150
2	2	Taylor	800
2	2	Taylor	900
3	NULL	Jones	NULL

SUM and GROUP BY result

id	person	name	money
1	1	Smith	250
2	2	Taylor	2100
3	NULL	Jones	NULL

Figure 2.3.u — Simplified logic of querying two tables

Now let's look at another example based on the same data set. Suppose we want to get the list of employees who have not had any payments. This problem can be solved either by using a `JOIN` query, or by using a subquery.

The expected result will look like this:

name
Jones

MySQL — An example of solving a problem using a `JOIN` query and a subquery

```mysql
-- Option 1 (JOIN):
SELECT `name`
FROM   `employee`
       LEFT OUTER JOIN `payment`
                    ON `employee`.`id` = `payment`.`person`
WHERE  `payment`.`person` IS NULL

-- Option 2 (subquery):
SELECT `name`
FROM   `employee`
WHERE  `id` NOT IN (SELECT `person`
                    FROM   `payment`)
```

MS SQL — An example of solving a problem using a `JOIN` query and a subquery

```mssql
-- Option 1 (JOIN):
SELECT [name]
FROM   [employee]
       LEFT OUTER JOIN [payment]
                    ON [employee].[id] = [payment].[person]
WHERE  [payment].[person] IS NULL

-- Option 2 (subquery):
SELECT [name]
FROM   [employee]
WHERE  [id] NOT IN (SELECT [person]
                    FROM   [payment])
```

Oracle — An example of solving a problem using a `JOIN` query and a subquery

```oracle
-- Option 1 (JOIN):
SELECT "name"
FROM   "employee"
       LEFT OUTER JOIN "payment"
                    ON "employee"."id" = "payment"."person"
WHERE  "payment"."person" IS NULL

-- Option 2 (subquery):
SELECT "name"
FROM   "employee"
WHERE  "id" NOT IN (SELECT "person"
                    FROM   "payment")
```

The solutions for all three DBMSes are exactly the same and work as follows.

Option 1:

- By executing the **`LEFT OUTER JOIN`**, the DBMS receives the same data set that was considered in the previous example (in figure 2.3.u the corresponding information is presented in the “LEFT OUTER JOIN result” block).
- In contrast to the previous example, where this operation was followed by summing up by groups of records, in this case the DBMS only needs to select the records that have not found a “pair” in the **`payment`** table — the **`NULL`** value of the **`person`** field is a sign of such a situation.

Option 2:

- First, the DBMS will execute the subquery (presented in lines 11–12 of the solutions for all three DBMSes) and get the list of employee identifiers that had payments.
- Now the DBMS will select from the **`employee`** table those records whose identifiers (**`id`**) are not in the set obtained in the previous step.

The presented solutions differ not only syntactically. There are many differences in the behavior of different DBMSes, depending on the data sets and the nuances of the queries themselves (unfortunately, it is beyond the scope of this book to discuss these details). Also, these variants may differ significantly in performance (in general, the `JOIN` option should be faster, but there may be exceptions)

This completes our consideration of "one to many" relationships and moves on to "many to many" relationships. Let's begin our consideration of their technical implementation with the example shown earlier in figures 2.3.e{57} and 2.3.f{57}.

The code for generation of the corresponding fragment of the database schema is as follows. As promised earlier, now we will consider the code for MySQL only.

MySQL Example of code for figures 2.3.e and 2.3.f

```
CREATE TABLE `m2m_student_subject`
(
 `st_id` INT NOT NULL,
 `sb_id` INT NOT NULL,
 CONSTRAINT `PK_m2m_student_subject` PRIMARY KEY (`st_id`,`sb_id`)
);

CREATE TABLE `subject`
(
 `sb_id` INT NOT NULL AUTO_INCREMENT,
 `sb_name` VARCHAR(50) NOT NULL,
 CONSTRAINT `PK_subject` PRIMARY KEY (`sb_id`)
);

CREATE TABLE `student`
(
 `st_id` INT NOT NULL AUTO_INCREMENT,
 `st_name` VARCHAR(50) NOT NULL,
 CONSTRAINT `PK_student` PRIMARY KEY (`st_id`)
);

ALTER TABLE `m2m_student_subject`
 ADD CONSTRAINT `FK_m2m_student_subject_student`
  FOREIGN KEY (`st_id`) REFERENCES `student` (`st_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `m2m_student_subject`
 ADD CONSTRAINT `FK_m2m_student_subject_subject`
  FOREIGN KEY (`sb_id`) REFERENCES `subject` (`sb_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;
```

There is nothing fundamentally new in this code as compared to what was discussed earlier (in the example devoted to "one to many" relationships). Except to reiterate: the "many to many" relationship as such does not exist at database level — it is formed from two "one to many" relationships drawn from linked tables (in our case, `student` and `subject`) to the so called "linking table" (in our case, `m2m_student_subject`).

Let's fill the `student` table with test data:

st_id	st_name
1	Smith
2	Taylor
3	Jones

Let's fill the **subject** table with test data:

sb_id	sb_name
1	Mathematics
2	Physics
3	Chemistry

And finally let's fill the **m2m_student_subject** table with test data:

st_id	sb_id
1	2
1	3
3	3

MySQL Example of code for test data insertion into the **student** table

```
INSERT INTO `student`
            (`st_name`)
VALUES      ('Smith'),
            ('Taylor'),
            ('Jones')
```

MySQL Example of code for test data insertion into the **subject** table

```
INSERT INTO `subject`
            (`sb_name`)
VALUES      ('Mathematics'),
            ('Physics'),
            ('Chemistry')
```

MySQL Example of code for test data insertion into the **m2m_student_subject** table

```
INSERT INTO `m2m_student_subject`
            (`st_id`, `sb_id`)
VALUES      (1, 2),
            (1, 3),
            (3, 3)
```

Earlier{56} it was promised to explain why in **m2m_student_subject** table both fields (**st_id** and **sb_id**), which are foreign keys, are simultaneously part of the compound natural primary key.

Let's explain this point with the example of solving the following problem. Suppose we need to find out how many students are studying each subject. The expected result looks like this.

sb_name	attendees
Physics	1
Chemistry	2

To get the result in this case it is enough to use only two tables (the logic of building a query is the same as in the examples devoted to "one to many" relationships).

MySQL An example code for solving this problem

```
SELECT `sb_name`,
       COUNT(`st_id`) AS `attendees`
FROM   `subject`
       JOIN `m2m_student_subject` USING(`sb_id`)
GROUP  BY `sb_id`
```

First, DBMS finds all pairs of records from **subject** and **m2m_student_subject** tables (based on equality of **sb_id** field values), then groups the obtained result by **sb_id** field values and counts the number of records in each such group. That's how the final result is obtained.

Now let's return to the compound primary key of `m2m_student_subject` table and pretend that this restriction does not exist, i.e., `st_id` and `sb_id` fields are not part of the compound primary key, and their values combination is not required to be unique. Then nothing prevents us from placing, for example, the following data into `m2m_student_subject` table:

st_id	sb_id
1	2
1	3
3	3
3	3
3	3
3	3

By executing the SQL query, we just considered on such a data set, we get:

sb_name	attendees
Physics	1
Chemistry	5

That is, there are five students studying chemistry. But if you look at the data in the `student` table, it is easy to see that we have only three students, i.e., there is no way there can be five students studying chemistry. To avoid such situations (which in real life can have serious economic and legal consequences), foreign keys in the "linking table" are introduced as part of the natural compound primary key.



Sometimes one may hear the objection that in other subject areas, on the contrary, it is necessary to repeatedly duplicate rows in the "linking table". Perhaps. But it is much more rational to have an additional field (a kind of counter) in this table and store the "number of repetitions" there, but still store the pair of primary key values of linked tables in the "linking table" exactly once. This solution is both more productive and more consistent with the concept of relational databases.

Let's continue this example by complicating the task: in addition to the number of students in each subject, we should also get a list of their names. That is, the result should look like this.

sb_name	attendees	names
Physics	1	Smith
Chemistry	2	Smith,Jones

The code for getting this result looks as follows.

MySQL — An example code for solving this problem

```
1   SELECT `sb_name`,
2          COUNT(`st_id`) AS `attendees`,
3          GROUP_CONCAT(`st_name`) AS `names`
4   FROM   `subject`
5          JOIN `m2m_student_subject` USING(`sb_id`)
6          JOIN `student` USING(`st_id`)
7   GROUP  BY `sb_id`
```

Here we can no longer do without querying only two tables, because students' names, subjects' names, and information about which students are studying which subjects are stored in three different tables. The general simplified logic for such queries is shown in figure 2.3.v.

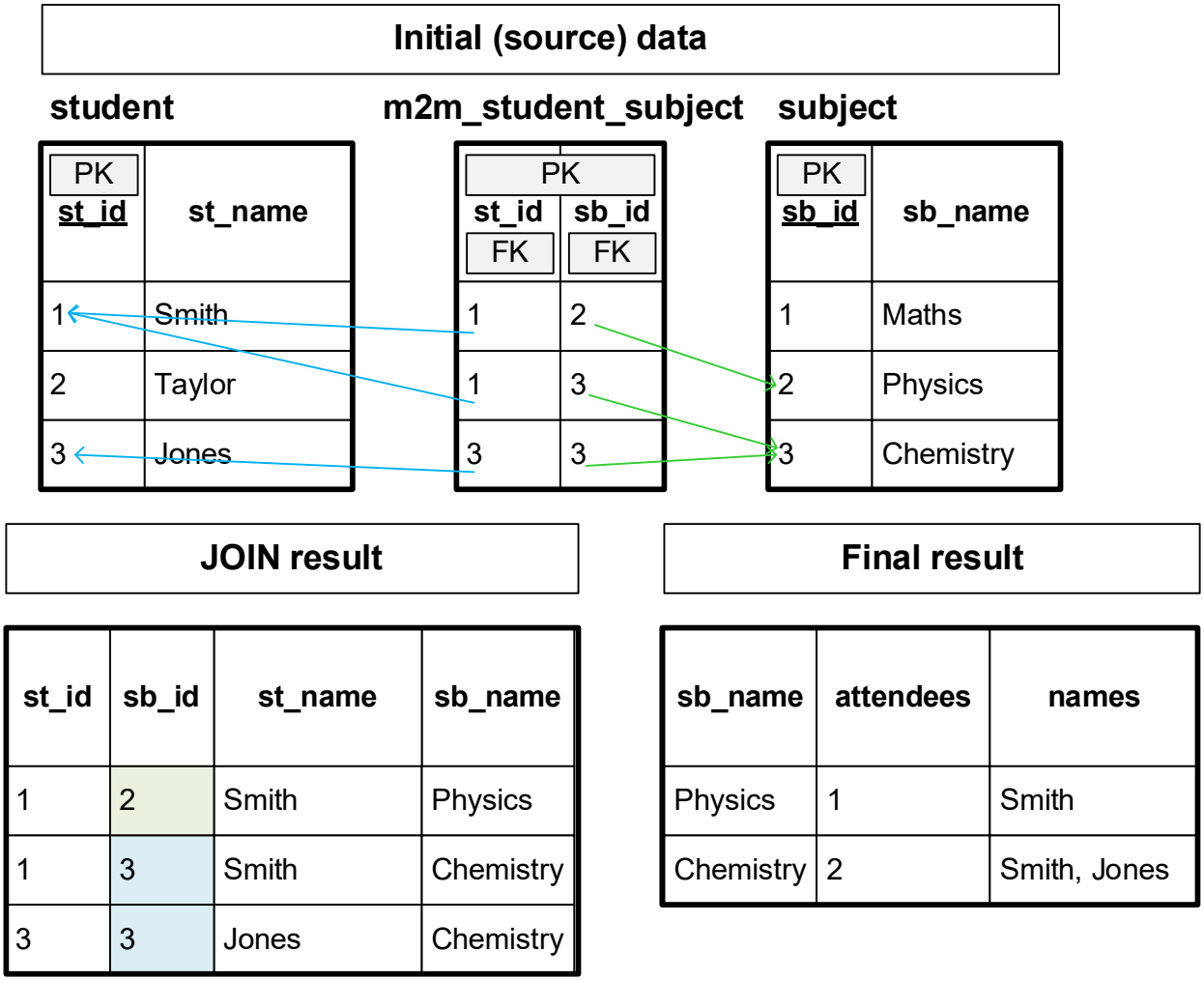


Figure 2.3.v — Simplified logic of querying three tables

Here we deliberately will not look at the internal logic of the different `JOINs`, groupings, etc. (as all these topics are covered in the book about using databases in practice[62]) — it is important now to understand the general principle of how different kinds of relationships are used, because this principle remains the same in any relational DBMS, but there are dozens of specific ways to implement it.

Now consider the most complex and atypical example, which will illustrate the use of "one to one" relationships and triggers{341} that provide data consistency.

Let's imagine that we are creating a database of an online store that sells computer parts. Obviously, products in such a store have both common properties (e.g., name and price) and unique properties that depend on the type of product (e.g., a monitor has screen diagonal length, but a processor has no such property, but there are others that do not exist for monitors).

[62] "Using MySQL, MS SQL Server and Oracle by Examples" (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

If we try to put information about all products in one table, we get two very unpleasant effects:

- such a table would have to contain thousands of fields (since we would have to list all possible properties of all kinds of products) — and this alone makes such a solution impossible, since any DBMS has a limit on the number of fields in a table, and it is usually measured in tens;
- even if we were able to create such a huge table, it would be almost entirely filled with **`NULL`**-values (because if some property does not apply to a certain type of product, the corresponding value should be set to **`NULL`**) — that is, data storage would not be organized optimally.

An alternative (and very well-proven) solution is to use a particular case[63] of a "star" schema with a set of "one to one" relationships.

To make the example easier to understand, we deliberately limit the number of product categories to two ("Printers", "CPUs"), and give only a couple of characteristics for each product category.

The resulting schema is shown in figure 2.3.w.

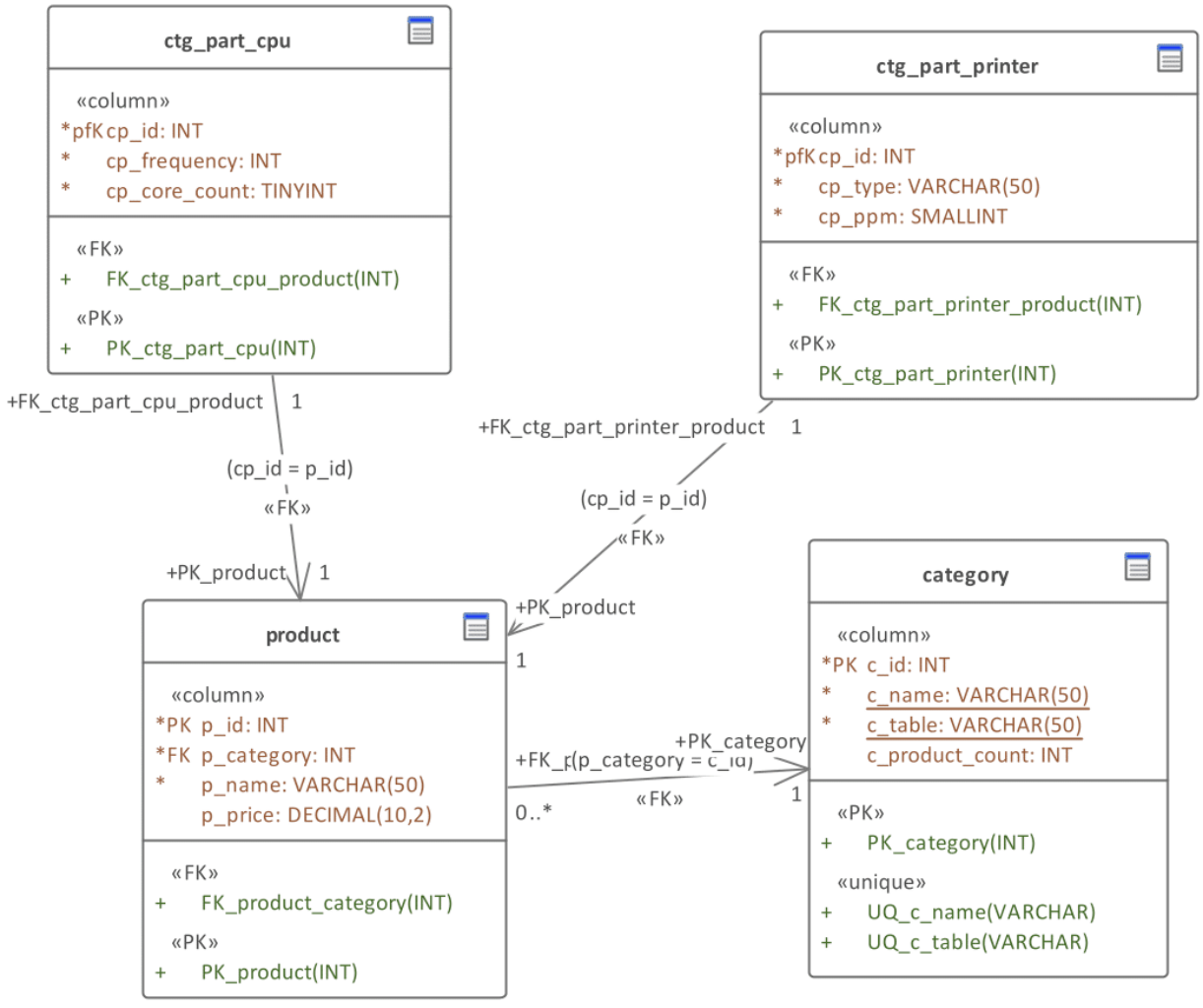


Figure 2.3.w — "One to one" relationships usage sample schema

[63] This is exactly a particular case, because in the classical "star" schema, the foreign keys are in the "central" table, and in this particular case — in the child tables.

So, “one to one” relationships are established here between the parent table (`product`) and its child tables (`ctg_part_cpu` and `ctg_part_printer`) and will be established to any new child table that describes products of any new category.

Relationship cardinality (“1 … 1”) is ensured by the fact that the foreign keys in the child tables are also primary keys. Also, there are two rather strict rules:

- primary keys of all child tables involved in “one to one” relationships must have the same name (in our case — `cp_id`), because otherwise the implementation of queries will become much more complicated;
- the names of all child tables involved in “one to one” relationships must begin with the `ctg_part_` prefix, because otherwise it would make writing code for applications working with such a database more complicated (see the explanation in the next paragraph).



This solution is quite optimal in terms of data storage, but it entails one very serious risk.

Let’s first note that there is a rather strict rule: an application working with a database can change any data, but under no circumstances should it change the database structure (at least, because there may be many other applications working with the database, and those applications do not expect such changes; and the very procedure of making changes to the database structure may cause irreversible damage to it).

But what to do in our situation if we need to add a new product category, delete or edit (change the set of product characteristics) an existing one? We have no other choice but to change the database structure (add a new table and a relationship, delete a table, change the set of fields in a table).

This is why the “two rather strict rules” were outlined above (in the context of the discussed risk, the second is particularly relevant) — they allow us to minimize the probability of problems with the structure of the database, albeit not excluded, but minimized.

Let’s create a database on the basis of the presented schema, fill it with test data and examine the logic of various operations.

The data will be as follows.

In `category` table:

c_id	c_name	c_table	c_product_count
1	Printer	ctg_part_printer	3
2	CPU	ctg_part_cpu	2

In `product` table:

p_id	p_category	p_name	p_price
1	1	Cheap printer	100.00
2	1	Expensive printer	1000.00
3	1	Just a printer	300.00
4	2	Cheap CPU	500.00
5	2	Expensive CPU	7000.00

In `ctg_part_cpu` table:

cp_id	cp_frequency	cp_core_count
4	1800	2
5	3500	6

In `ctg_part_printer` table:

cp_id	cp_type	cp_ppm
1	laser	10
2	laser	30
3	ink-jet	5

The code looks as follows.

MySQL Example code for figure 2.3.w (creating tables and relationships)

```
CREATE TABLE `product`
(
 `p_id` INT NOT NULL AUTO_INCREMENT,
 `p_category` INT NOT NULL,
 `p_name` VARCHAR(50) NOT NULL,
 `p_price` DECIMAL(10,2)  NULL,
 CONSTRAINT `PK_product` PRIMARY KEY (`p_id`)
);

CREATE TABLE `ctg_part_printer`
(
 `cp_id` INT NOT NULL,
 `cp_type` VARCHAR(50) NOT NULL,
 `cp_ppm` SMALLINT NOT NULL,
 CONSTRAINT `PK_ctg_part_printer` PRIMARY KEY (`cp_id`)
);

CREATE TABLE `ctg_part_cpu`
(
 `cp_id` INT NOT NULL,
 `cp_frequency` INT NOT NULL,
 `cp_core_count` TINYINT NOT NULL,
 CONSTRAINT `PK_ctg_part_cpu` PRIMARY KEY (`cp_id`)
);

CREATE TABLE `category`
(
 `c_id` INT NOT NULL AUTO_INCREMENT,
 `c_name` VARCHAR(50) NOT NULL,
 `c_table` VARCHAR(50) NOT NULL,
 `c_product_count` INT    NULL,
 CONSTRAINT `PK_category` PRIMARY KEY (`c_id`)
);

ALTER TABLE `category`
 ADD CONSTRAINT `UQ_c_name` UNIQUE (`c_name`);

ALTER TABLE `category`
 ADD CONSTRAINT `UQ_c_table` UNIQUE (`c_table`);

ALTER TABLE `product`
 ADD CONSTRAINT `FK_product_category`
  FOREIGN KEY (`p_category`) REFERENCES `category` (`c_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `ctg_part_printer`
 ADD CONSTRAINT `FK_ctg_part_printer_product`
  FOREIGN KEY (`cp_id`) REFERENCES `product` (`p_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;

ALTER TABLE `ctg_part_cpu`
 ADD CONSTRAINT `FK_ctg_part_cpu_product`
  FOREIGN KEY (`cp_id`) REFERENCES `product` (`p_id`)
   ON DELETE CASCADE ON UPDATE CASCADE;
```

MySQL Example code for figure 2.3.w (data insertion)

```
INSERT INTO `category`
            (`c_name`, `c_table`, `c_product_count`)
VALUES      ('Printer', 'ctg_part_printer', 3),
            ('CPU', 'ctg_part_cpu', 2);

INSERT INTO `product`
            (`p_category`, `p_name`, `p_price`)
VALUES      (1, 'Cheap printer', 100),
            (1, 'Expensive printer', 1000),
            (1, 'Just a printer', 300),
            (2, 'Cheap CPU', 500),
            (2, 'Expensive CPU', 7000);

INSERT INTO `ctg_part_printer`
            (`cp_id`, `cp_type`, `cp_ppm`)
VALUES      (1, 'laser', 10),
            (2, 'laser', 30),
            (3, 'ink-jet', 5);

INSERT INTO `ctg_part_cpu`
            (`cp_id`, `cp_frequency`, `cp_core_count`)
VALUES      (4, 1800, 2),
            (5, 3500, 6);
```

What's worth paying attention to here:

- primary keys in **`ctg_part_printer`** and **`ctg_part_cpu`** tables are not autoincrementable (their values are known in advance, since they are both primary and foreign keys);
- when inserting data, be sure to follow the sequence:
  1. insert a record to the **`product`** table;
  2. figure out the value of the autoincrementable primary key for the record we've just added;
  3. insert a record to the child table (with an additional product description) using that value of the autoincrementable primary key.
- this sequence of actions must be performed in one session (without closing the connection to DBMS), because otherwise you can "lose" the new primary key value of the **`product`** table and/or "attach" an additional product description to the wrong product;
- the **`category`** table has the special **`c_table`** field, which stores the name of the table where additional data about products of each category are located (the name of this table is mandatory to know in order to perform the vast majority of queries to the database built on this schema).

The final picture of what the database looks like if built according to the schema from figure 2.3.w, created and populated with data using the code just discussed, is shown in figure 2.3.x.

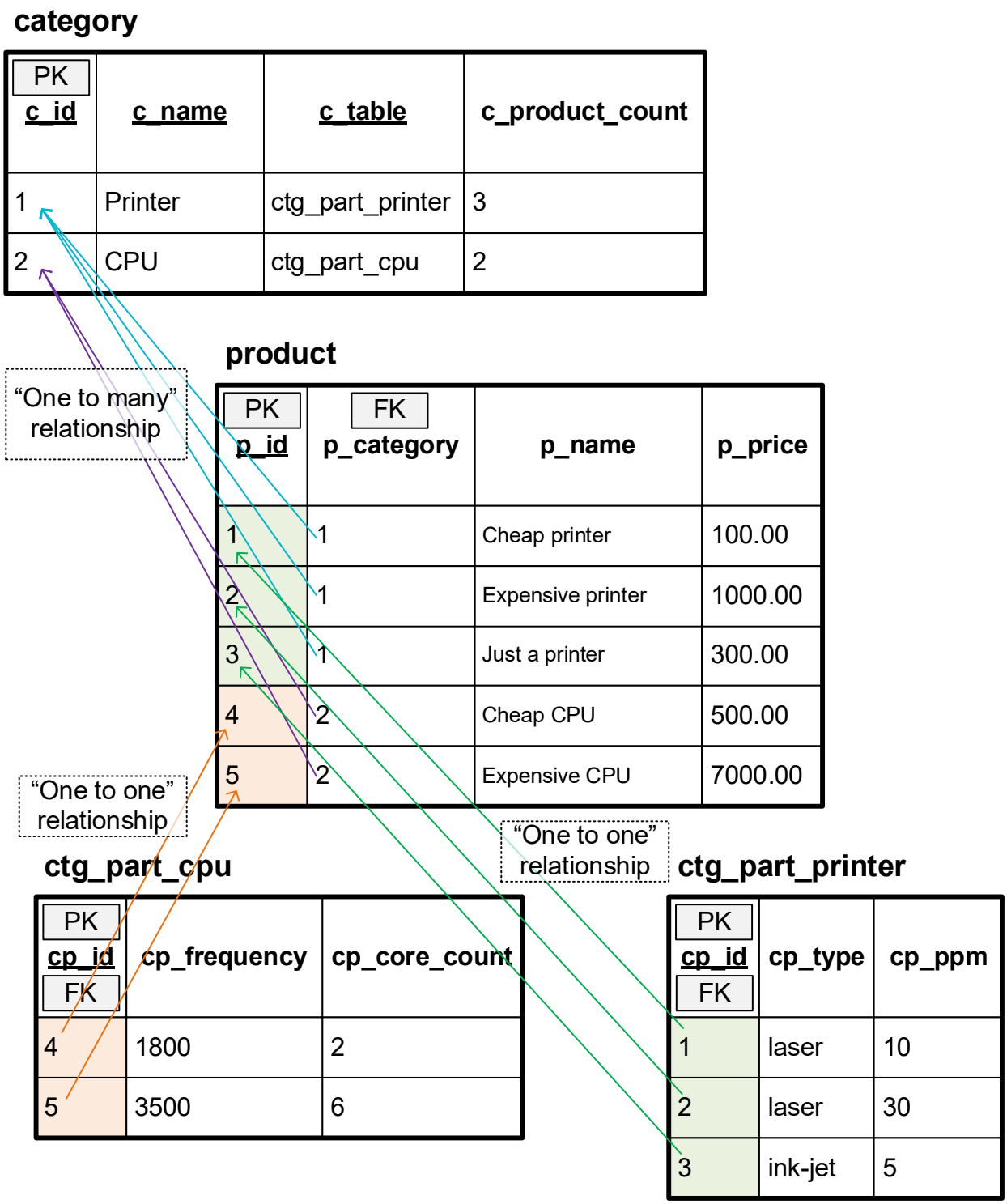


Figure 2.3.x — Final view of a database fragment built using "one to one" relationships

Before we proceed to operations with such a database, let's emphasize the point once again: the description of each product is divided into two parts, one of which (characteristic for any product) is stored in the `product` table, and the second (characteristic only for products of this category) is stored in its own separate table.

Data insertion can be done in two ways:

- perform a separate query to insert a record into the product table, then perform a separate query to find out a new value of the autoincrementable primary key, then perform a separate query to insert a record into the child table (substituting the primary key value obtained during the previous step) — each of these queries is usually performed separately at the database application level;
- create a stored procedure{353} that will receive a set of data for insertion and perform all necessary operations "internally" — this is a more reliable way, but such a stored procedure will have to be created separately for each product category (or we'll have to create a rather complex solution with transferring to the procedure and then analyzing a huge data array).

MySQL — Example code for figure 2.3.x (data insertion, option 1 — sequential operations)

```
1    -- a) Data insertion into "product" table:
2    INSERT INTO `product`
3                (`p_category`, `p_name`, `p_price`)
4    VALUES      (1, 'New printer', 150.43);
5
6    -- b) Getting the new value of the autoincrementable primary key:
7    SET @new_key = (SELECT LAST_INSERT_ID());
8
9    -- c) Data insertion into s table that stores specific product properties:
10   INSERT INTO `ctg_part_printer`
11               (`cp_id`, `cp_type`, `cp_ppm`)
12   VALUES      (@new_key, 'laser', 17);
```

MySQL — Example code for figure 2.3.x (data insertion, option 2 — creating a stored procedure)

```
1    DROP PROCEDURE ADD_PRINTER;
2    DELIMITER $$
3    CREATE PROCEDURE
4     ADD_PRINTER (IN printer_name  VARCHAR(50),
5                  IN printer_price DECIMAL(10, 2),
6                  IN printer_type  VARCHAR(50),
7                  IN printer_ppm   SMALLINT,
8                  OUT new_pk INT)
9    BEGIN
10    -- Category id detection. In reality, you don't have to do
11    -- this (the id unlikely to change), but here this operation is given to
12    -- demonstrate the complete set of actions.
13    SET @category_id = (SELECT `c_id`
14                        FROM   `category`
15                        WHERE  `c_table` = 'ctg_part_printer');
16
17    -- Data insertion into "product" table:
18    INSERT INTO `product`
19                (`p_category`, `p_name`, `p_price`)
20    VALUES      (@category_id, printer_name, printer_price);
21
22    -- Getting the new value of the autoincrementable primary key:
23    SET @new_key = (SELECT LAST_INSERT_ID());
24
25    -- Data insertion into s table that stores specific product properties:
26    INSERT INTO `ctg_part_printer`
27                (`cp_id`, `cp_type`, `cp_ppm`)
28    VALUES      (@new_key, printer_type, printer_ppm);
29
30    SET new_pk = @new_key;
31   END;
32   $$
33   DELIMITER ;
```

MySQL Example code for figure 2.3.x (data insertion, option 2 — using a stored procedure)

```
CALL ADD_PRINTER('Brand new printer', 199.23, 'photo', 2, @pk_assigned);
SELECT @pk_assigned;
```

After executing the stored procedure, as a “bonus” we get the value of the primary key of the record storing data about the product (this value is the same for product and `ctg_part_printer` tables, so there is no need to specify which table we are talking about).

The variable with which we get this value can also be used to transmit information about errors (e.g., using a range of negative integers as error codes). At the moment this has not been implemented in the considered stored procedure code in order to simplify it as much as possible.

Unlike data insertion, which requires such an unusual approach, update and deletion are quite trivial. When updating, we just need to remember that the product data is in two different tables (i.e., we need to update the proper one or both of them). And to perform deletion we need to delete a record from `product` table (which will activate cascade operation and automatically delete the corresponding record from the corresponding child table).

Here are the corresponding code examples:

MySQL Example code for figure 2.3.x (data update and deletion)

```
-- Update of the data part stored in the "product" table:
UPDATE `product`
   SET `p_price` = 299.45
 WHERE p_id = 7;

-- Update of the data part stored in the "ctg_part_printer" table:
UPDATE `ctg_part_printer`
   SET `cp_ppm` = 3
 WHERE cp_id = 7;

-- Data deletion:
DELETE FROM `product`
      WHERE p_id = 7;
```

We have considered all operations of data modification. It remains to demonstrate how data selection is performed using this schema.

Here the task is divided into two:

- in the simple case, the query will be executed on the `product` table and/or `product` and `category` tables (for example, to calculate the average price of all products or the average price of products within a particular product category) — here the logic is no different from the previously discussed examples of working with “one to many” relationships;
- in a more complicated case, the query will need to be executed on the `product` table and the corresponding child table (depending on the product category), which stores additional specific product parameters.

For the first case, let’s just consider a couple of example queries without explanations (see details of working with “one to many” relationships above{73} in this chapter, in case something in these queries seems complicated).

The queries below allow us to calculate the average price of all products (a query for only `product` table) and create a list of product categories with the average price of each product category (a query for both `product` and `category` tables).

MySQL — Example code for determining the average price of all products and products in each category

```
1     -- Determining the average price of all products:
2     SELECT AVG(`p_price`) AS `avg_price`
3     FROM   `product`
4
5     -- Determining the average price of products in each category:
6     SELECT `c_name`,
7            AVG(`p_price`) AS `avg_price`
8     FROM   `product`
9            JOIN `category`
10             ON `p_category` = `c_id`
11    GROUP  BY `p_category`
```

The result of the first query:

avg_price
1780.000000

The result of the second query:

c_name	avg_price
Printer	466.666667
CPU	3750.000000

If we want to get a complete list (with all parameters) of products from a category, we have to query the **`product`** table and the corresponding child table that stores data about the products of the specified category.



We should note that it is technologically difficult (and practically meaningless) to try to get a complete list (with all parameters) of all products of several (or all at once) categories simultaneously, because in the resulting table, the data in the columns describing the specific parameters will have different meaning for different product categories.

This is a very typical mistake that leads to violation of the fundamental principles of relational theory: by definition, the data in one table column (relation attribute) must have the same meaning, i.e., be treated equally for all table entries (relation tuples). In the case under consideration this rule is violated, and the result will look like this.

MySQL — A query with correct syntax yet without any sense from a human point of view

```
1     SELECT `p_id`,
2            `p_name`,
3            `p_price`,
4            `cp_frequency`  AS `param1`,
5            `cp_core_count` AS `param2`
6     FROM   `product`
7            JOIN `ctg_part_cpu`
8              ON `p_id` = `cp_id`
9     UNION
10    SELECT `p_id`,
11           `p_name`,
12           `p_price`,
13           `cp_type` AS `param1`,
14           `cp_ppm`  AS `param2`
15    FROM   `product`
16           JOIN `ctg_part_printer`
17             ON `p_id` = `cp_id`
```

The result of such a query will be a data set in which **`param1`** and **`param2`** fields have different meanings for different records (e.g., **`param1`** field for printers means type, but for CPUs it means frequency).

p_id	p_name	p_price	param1	param2
4	Cheap CPU	500.00	1800	2
5	Expensive CPU	7000.00	3500	6
1	Cheap printer	100.00	laser	10
2	Expensive printer	1000.00	laser	30
3	Just a printer	300.00	ink-jet	5

The counterargument can sometimes be heard, which is that almost any DBMS allows us to run queries with grouping and summarizing (`ROLLUP`, `CUBE`), where the field values in some rows take on a different special meaning. Yes, DBMS can technically perform many operations, potentially leading to many errors in applications. Whether to take advantage of these features, or to favor reliability is up to you.

Let's get back to the problem at hand. As in the other examples discussed above, it has several solutions.

The first (simple) is feasible if you know the name of the table storing the specific parameters of the specified product category on the side of the application working with the DBMS. Then the solution can be performed with one simple query:

MySQL — Example code for a case when the query is completely prepared on the application side

```
1    SELECT `product`.*,
2           `ctg_part_cpu`.*
3    FROM   `product`
4           JOIN `ctg_part_cpu`
5             ON `p_id` = `cp_id`
```

This query gives the following result (yes, the query can be improved by listing specific table fields of interest, but in this case, for the sake of clarity, the full set of data is chosen):

p_id	p_category	p_name	p_price	cp_id	cp_frequency	cp_core_count
4	2	Cheap CPU	500.00	4	1800	2
5	2	Expensive CPU	7000.00	5	3500	6

However, it would be nice to be able to freely select data on some product category just by specifying its identifier and not thinking about anything else. Since MySQL does not allow to perform dynamic queries inside stored functions{353} the solution is based on the stored procedure{353}.

In the code below, all those actions that an application would have to perform are implemented within a stored procedure:

- first, based on the category identifier, we get the name of the table in which the specific parameters of the products of the specified category are stored;
- then we have two ways of proceeding:
  - there is no such product category — empty result should be returned;
  - there is such a product category, we know the name of the table where the specific parameters of its products are stored, and we can form and execute a query to get this information.

MySQL — Creation of the stored procedure to retrieve all data about products of a given category

```
DELIMITER $$
CREATE PROCEDURE
 SHOW_PRODUCTS_FROM_CATEGORY (IN category_id INT)
BEGIN
 -- Detection of the name of the table in which the data about the specific
 -- parameters of the products of this category are stored:
 SET @ctg_table = (SELECT `c_table`
                   FROM   `category`
                   WHERE  `c_id` = category_id);

 IF (@ctg_table IS NULL)
  THEN
   -- If the category with the identifier of interest is
   -- not found, an empty result is returned.
   SELECT NULL;
  ELSE
   -- Query text generation:
   SET @query_text = CONCAT('SELECT `product`.*, `', @ctg_table,
                            '`.* FROM `product` JOIN `', @ctg_table,
                            '` ON `p_id` = `cp_id`');

   -- Preparing and executing the query:
   PREPARE query_stmt FROM @query_text;
   EXECUTE query_stmt;
  END IF;
END;
$$
DELIMITER ;
```

Now getting a full set of data about the products of any category comes down to this simple call of the stored procedure:

MySQL — Using the stored procedure to retrieve all data about products of a given category

```
CALL SHOW_PRODUCTS_FROM_CATEGORY(2);
```

The last thing left for us to consider in this chapter is the implementation of data consistency with the help of triggers (which will be described in detail in the corresponding section{341}, so here we will only give an example code with a minimum of explanations)

This example doesn't apply to any type of relationship in isolation and can be applied to any relationship except "one to one", where it becomes meaningless.

You may have noticed that the category table has a `c_product_count` field, the value of which for each record equals the number of products in the corresponding category. Of course, it can be calculated at any time by a separate query, but for the case of a very large number of products and their categories, the execution of such a query can take considerable time, and therefore we would like to be able to get the value we are interested in immediately — that's why we store it explicitly.

When filling the tables with test data, we specified this value in the insertion request itself, because we knew it beforehand, but in real database operation it should be calculated automatically, which is a classic variant of database consistency{67}.

The value of this field should change in three cases:

- a new product was added to a category (data insertion);
- a product was deleted from a category (data deletion);
- a product was moved from one category to another (data update).

In other words, the DBMS must automatically update the value of this field in all three cases listed. And we have a corresponding mechanism to fully automate this operation — triggers{341}.

As you may have already guessed, this problem also has several solutions (and if we consider the features of different DBMSes, there will be even more options). In general, we can distinguish two approaches:

- reliable, but slow — we can recalculate the number of items in each category when we modify the list of products;
- fast, yet less reliable — we can change the number of items in each category by the number of items that were added/removed/moved (here we risk getting an incorrect value if the original value did not correspond to reality).

Let's implement both options and start with the slow (and reliable) one. The basic code for the **`INSERT`** and **`DELETE`** triggers could also be implemented through an **`UPDATE`** query with **`JOIN`**, but for the sake of variety there is an alternative solution.

MySQL — Creation of triggers (the first option — slower, but more reliable)

```
DELIMITER $$
CREATE TRIGGER `prod_count_ins`
AFTER INSERT ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = (SELECT COUNT(`p_id`)
                               FROM   `product`
                               WHERE  `p_category` = NEW.`p_category`)
   WHERE  `c_id` = NEW.`p_category`;
  END;
$$

CREATE TRIGGER `prod_count_upd`
AFTER UPDATE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
    JOIN (SELECT `p_category`,
                 COUNT(`p_id`) AS `new_count`
          FROM   `product`
          GROUP  BY `p_category`
          HAVING `p_category` IN (OLD.`p_category`, NEW.`p_category`)
         ) AS `prepared_data`
         ON `c_id` = `p_category`
   SET  `c_product_count` = `new_count`
   WHERE `c_id` IN (OLD.`p_category`, NEW.`p_category`);
  END;
$$

CREATE TRIGGER `prod_count_del`
AFTER DELETE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = (SELECT COUNT(`p_id`)
                               FROM   `product`
                               WHERE  `p_category` = OLD.`p_category`)
   WHERE  `c_id` = OLD.`p_category`;
  END;
$$
DELIMITER ;
```

A detailed explanation of the logic of such triggers is given in the corresponding book[64], but if we briefly summarize, it turns out:

- in the first trigger, based on the value of the category identifier of the added product `NEW.`p_category`` in the subquery, a new number of products in this category is determined, and this value is used to update the `c_product_count` field;
- in the second trigger, the value of the `c_product_count` field must be updated for two categories (whose identifiers are in `OLD.`p_category`` and `NEW.`p_category`)` — to avoid running two separate queries, here we used an update query with `JOIN`;
- in the third trigger, based on the value of the category identifier of the deleted item `OLD.`p_category`` in the subquery, the remaining number of items in the category is determined, and this value is used to update the `c_product_count` field.

Pay attention to the fact that all triggers are exactly `AFTER`-triggers, i.e., they must be activated **after** modification operation to process new actual data.

Let's go to the second variant of this solution — fast, but which does not guarantee correctness of the obtained result (in case if the initial value of `c_product_count` field was incorrect).

MySQL — Creation of triggers (the first option — slower, but less reliable)

```
DELIMITER $$
CREATE TRIGGER `prod_count_ins_fast`
AFTER INSERT ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` + 1
   WHERE  `c_id` = NEW.`p_category`;
  END;
$$

CREATE TRIGGER `prod_count_upd_fast`
AFTER UPDATE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` + 1
   WHERE  `c_id` = NEW.`p_category`;
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` - 1
   WHERE  `c_id` = OLD.`p_category`;
  END;
$$

CREATE TRIGGER `prod_count_del_fast`
AFTER DELETE ON `product`
 FOR EACH ROW
  BEGIN
   UPDATE `category`
   SET    `c_product_count` = `c_product_count` - 1
   WHERE  `c_id` = OLD.`p_category`;
  END;
$$
DELIMITER ;
```

[64] "Using MySQL, MS SQL Server and Oracle by examples" (Svyatoslav Kulikov), example 32 [https://svyatoslav.biz/database_book/]

The code of these triggers is much simpler than in the first variant of the solution. The whole point of the operation is to find out the identifiers of `NEW.`p_category`` (the product was added to such category) and `OLD.`p_category`` (the product was deleted from such category), and then the value of `c_product_count` field of the corresponding category is increased or decreased by one (depending on whether the product was added or deleted).

It remains to check how the considered triggers work (obviously, it is necessary to create a set of triggers **either** according to the first option, **or** according to the second option, but **not simultaneously**).

MySQL — Test of `INSERT`-trigger

```
1    INSERT INTO `product`
2                (`p_category`, `p_name`, `p_price`)
3    VALUES      (1, 'Some printer 1', 200.5),
4                (1, 'Some printer 2', 300.7)
```

			Before	After
c_id	c_name	c_table	c_product_count	c_product_count
1	Printer	ctg_part_printer	3	5
2	CPU	ctg_part_cpu	2	2

MySQL — Test of `UPDATE`-trigger

```
1    UPDATE `product`
2    SET    `p_category` = 2
3    WHERE  `p_id` = 1
```

			Before	After
c_id	c_name	c_table	c_product_count	c_product_count
1	Printer	ctg_part_printer	5	4
2	CPU	ctg_part_cpu	2	3

MySQL — Test of `DELETE`-trigger

```
1    DELETE FROM `product`
2    WHERE  `p_id` > 5
```

			Before	After
c_id	c_name	c_table	c_product_count	c_product_count
1	Printer	ctg_part_printer	4	2
2	CPU	ctg_part_cpu	3	3

It is worth mentioning that the approach we just considered has an alternative — the “entity-attribute-value” model[65]. In this alternative, we return to “one to many” relationships, but we achieve the same goal: we store information about a set of properties of entities that have very few overlapping attributes.

Figure 2.3.y shows a schema of this solution, and figure 2.3.z shows an example of tables with data.

The key idea of this approach is to store all the “different properties” of all entities in the same table, but not in separate columns (as the relational data model dictates), but in just two, the first of which (`pr_name`) will store the name of the property, and the second (`pr_value`) will store its value.

[65] “Entity-attribute-value” model [https://en.wikipedia.org/wiki/Entity%E2%80%93attribute%E2%80%93value_model]

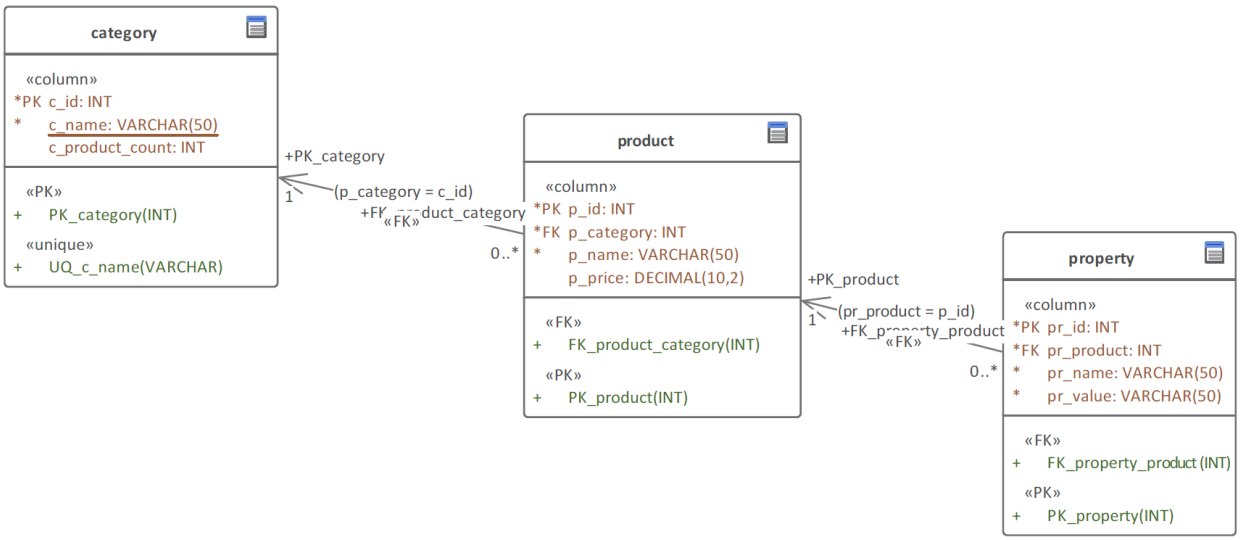


Figure 2.3.y — Entity-attribute-value schema sample

The advantages of this solution are as follows:

- there is no need to make changes to the database schema when changing the set of described entities and/or the set of entity attributes;
- the names of all tables and columns are known in advance and are fixed;
- instances of the same entity may have different sets of attributes.

But this solution also has a serious set of drawbacks:

- in fact, the principle of relational databases is violated here, because now, within the scope of the subject area, the values inside the `pr_name` and `pr_value` columns have a different meaning (e.g., the “capacity” for HDD/SSD, and the “capacity” for “ink cartridge”, even having same name, still have different meanings);
- operations of comparison, summation, getting average value, etc. for the `pr_value` column now make sense only as a part of the additional operation of determining whether a certain set of values in interest belong to the same properties of products of the same type (i.e., it makes no sense to compare, for example, the capacity of an SSD with the data transfer rate of an SSD or the capacity of an SSD with the capacity of an ink cartridge)
- there is no possibility to strictly specify the data type for the `pr_value` column, because depending on the described property of the product, there can be integers, doubles, strings, and — generally, any data type;
- as a consequence of the previous point, it is extremely difficult to control data consistency (correctness of type, value, format, the need for presence (or the possibility of absence), etc.);
- all of the above leads to one of two negative consequences — either it is necessary to implement within the DBMS several extremely complex mechanisms for manipulating such data and controlling their consistency, or all the corresponding operations must be implemented on the application (applications) side, which could potentially become an even bigger problem.

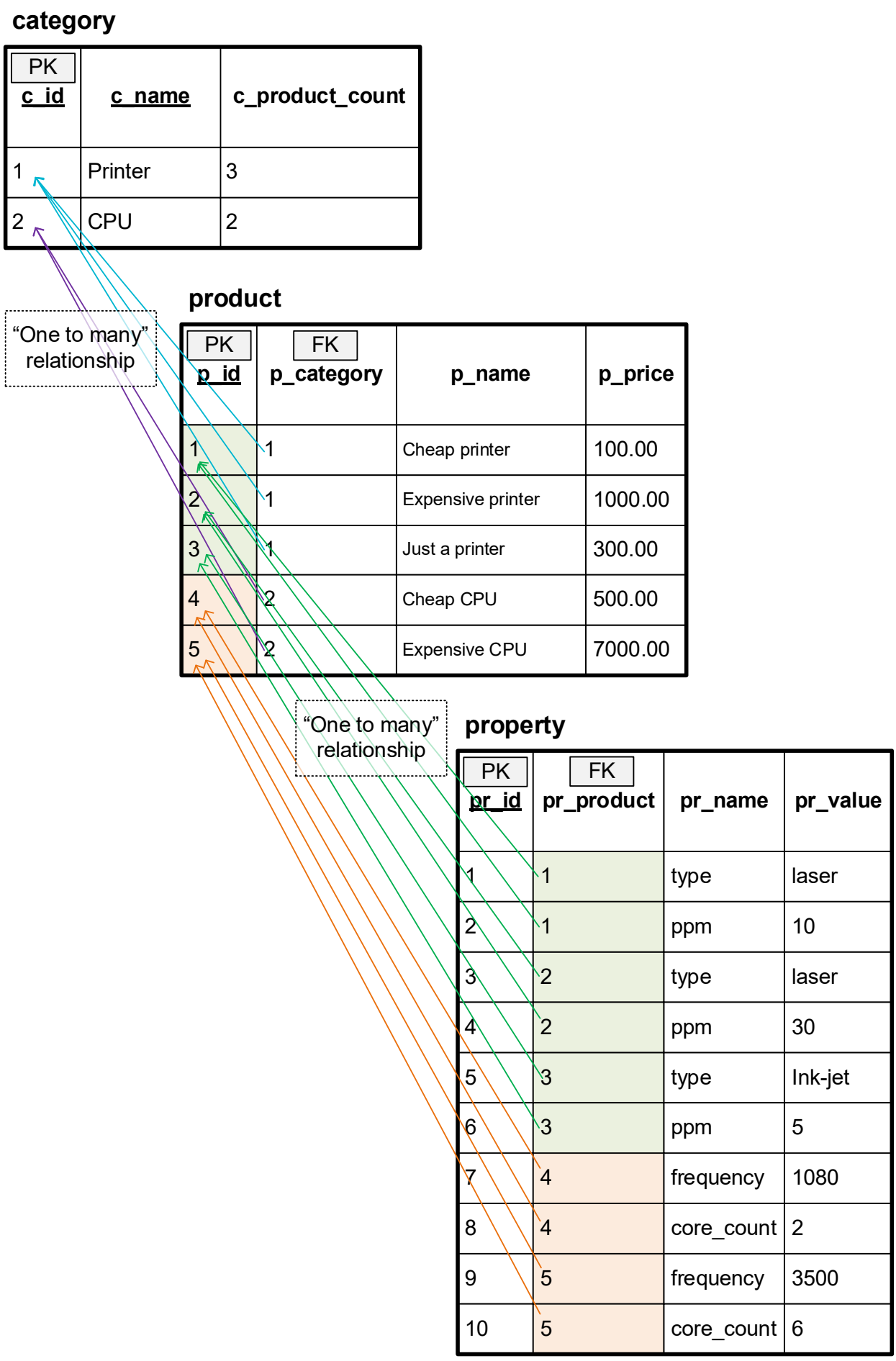


Figure 2.3.z — Entity-attribute-value schema implementation sample

© EPAM Systems, 2020–2025

Does such a wide range of shortcomings mean that the entity-attribute-value approach has no right to exist?

No, it doesn't. In some situations, this solution can be more efficient (and even more reliable) than the previously considered one (using "one to one" relationships and several tables).

But it is quite obvious that much more attention and effort are required from database developers to implement "manually" many DBMS mechanisms that work imperceptibly (immediately and by themselves) in alternative solutions.

At this point, the basic operations associated with the practical use of relationships can be considered reviewed. Although, of course, in the following sections we will return to them more than once, because it is difficult to imagine a more fundamental idea of relational databases.



Task 2.3.g: put into practice the idea described above in note{85} about cases where the "linking table" needs to account for the number of relationships set. Use triggers to control the counter. Examples of subject areas for which such a schema is relevant (but better yet, make up your own):

- users and music files (the counter stores the number of listens);
- discount card holders and stores (the counter stores the number of visits);
- students and subjects (the counter stores the number of attended classes).



Task 2.3.h: imagine that some application needs to track and record the number of messages users have sent to each other (any user can send any number of messages to any other user (but not to himself)). Create a database schema fragment to store the information and implement a restriction that doesn't allow one to enter into the database the fact that users sent a message to themselves.



Task 2.3.i: in the "Bank{395}" database should we implement the "relationship counter" solution described above{85} in any tables that implement "many to many" relationships? If you think "yes", please modify the schema accordingly.

2.4. Indexes

2.4.1. General Information on the Indexes



In its entirety, the material in this chapter would take several thousand pages. For this reason, please note that only the most basic information necessary to form an idea of how indexes work will be presented here.

It should also be noted that indexes lie almost entirely in the realm of physical{305} database design, and relational theory has little to do with them.

In general terms, the definition of “index” can be formulated as follows:



Index[66] — a special database structure used to speed up record search (or retrieval) and physical access to records.

Simplified: a mechanism that greatly speeds up the search for necessary information in the database (by analogy: for people, a city map is such an “index” that allows them to quickly find the right building).

Indexes are widely used in database design because of the following advantages:

- the size of the indexes is much smaller than the size of the indexed data, which allows them to be placed in RAM for faster access;
- the index structure is specifically optimized for search operations;
- as a consequence of the previous two points, indexes significantly (sometimes by several orders of magnitude) speed up data search or retrieval.

However, indexes have disadvantages, which must be taken into account, so as not to create indexes where they are not needed:

- when there are a lot of indexes, they occupy a tangible amount of RAM;
- the presence of indexes significantly slows down data modification operations (insertion, deletion, update), because when the DBMS changes the data, it is necessary to update the indexes by adjusting them to the new values of the indexed data.

Hence the logical question: how do we know that this or that index is needed? There are several indications that speak in favor of creating an index:

- operations of reading from the table are performed much more often than operations of data modification;
- a field or a set of fields often appear in queries in the `WHERE` section;
- research has shown that the presence of an index improves query performance;
- it is necessary to provide uniqueness of values of a field (or a set of fields), which is not a primary key (in this case the so-called unique index{106} is created);
- a field (or a set of fields) is a foreign key — in this case indexes can significantly speed up `JOIN`-queries.

[66] **Index** — a specific kind of physical access path (an implementation construct, intended to improve the speed of access to data as physically stored). (“The New Relational Database Dictionary”, C.J. Date)

There are many different kinds of indexes, and now we will look at the main examples (see figure 2.4.a). Also note that depending on the DBMS and the chosen storage engine{30} the list of supported indexes and features of their implementation may be very different.

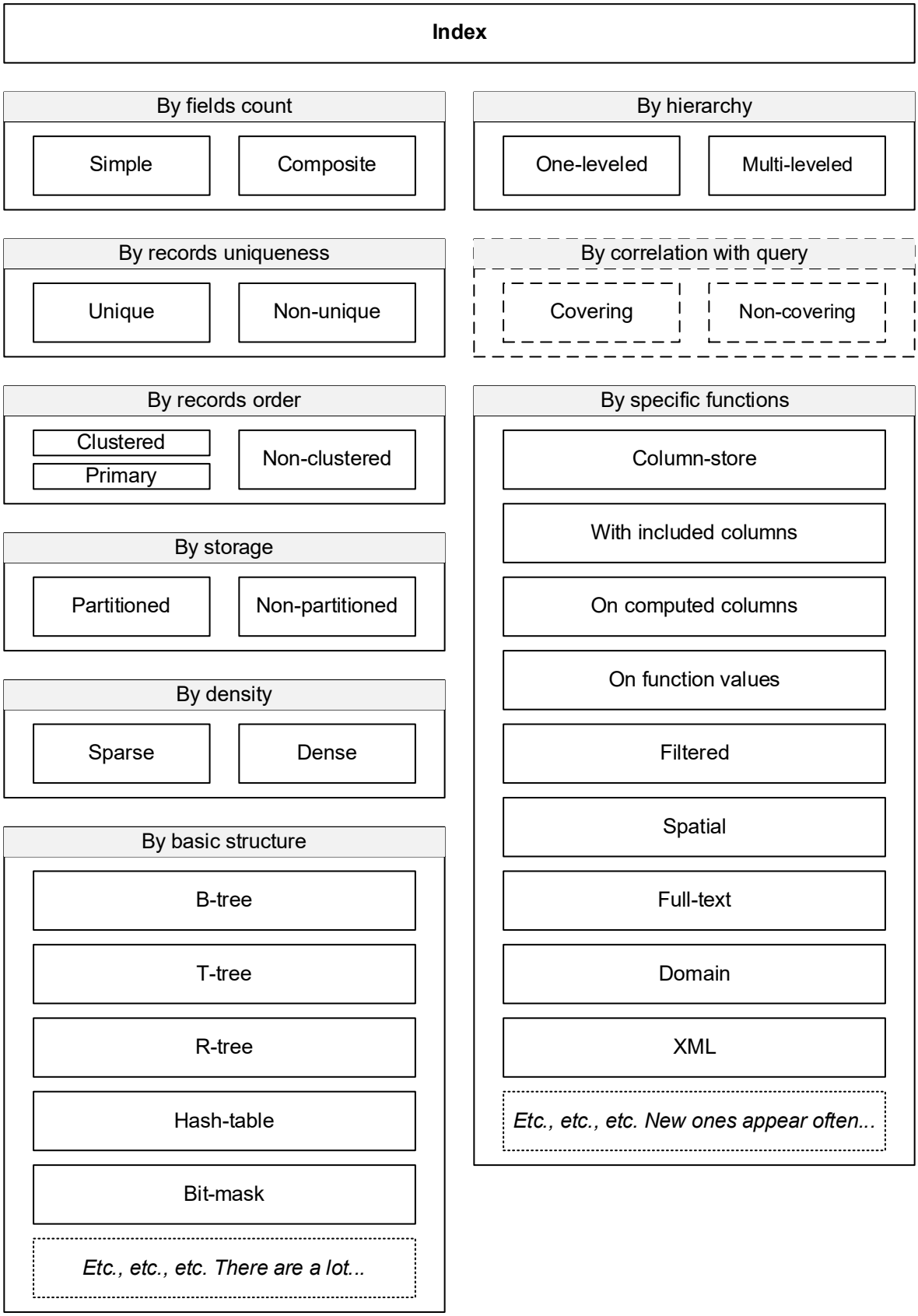


Figure 2.4.a — Index types and their interconnection

By fields count indexes are classified quite similarly to keys{32}, namely:



Simple index (single-column index[67]) — index built on a single field of a table.

Simplified: an index that includes information about the contents of only one field of a table.



Composite index[68] — index built on two or more fields of a table.

Simplified: an index that includes information about the contents of two or more fields of a table.

In some cases (with some assumptions) the terms "simple key{36}"/“simple index” and “composite key{36}”/“composite index” can be used as synonyms[69].

Both simple and composite indexes can be used to ensure uniqueness of alternate key values{35} (in this case it will be a unique index{106}) or to speed up the search of records by the value of indexed fields (such index can be both unique{106}, and non-unique{106}).

For composite indexes, the field sequence problem discussed in detail earlier{47} is fully characteristic: the field that will often be searched for separately from the other fields should be the first.

Any modern DBMS supports the use of both simple and composite indexes.

Since the question of how a composite index is physically organized is often asked, let’s explain this point by comparing a simple and a composite index based on a balanced tree{114}. Figure 2.4.b presents the structure of such a tree in a simplified way, and a more detailed explanation is given in figure 2.4.i below{114}.

In general, two strategies are used to index the second (third, etc.) field:

- combined keys usage, when values of additional fields are stored next to values of main fields — such approach is technically easier, but can lead to a noticeable increase in memory costs, if in the stored data there is a situation when one value of the first indexed field corresponds to many different values of subsequent fields;
- nested trees usage (when each node of the main tree is the root of an additional nested tree) — this approach is technically more complicated but saves memory in the case where one value of the first indexed field corresponds to many different values of subsequent fields.

[67] **Single-column index** — an index that is created based on only one table column (“SQL Indexes”). [https://www.tutorialspoint.com/sql/sql-indexes.htm]

[68] **Composite index** — an index on two or more columns of a table (“SQL Indexes”). [https://www.tutorialspoint.com/sql/sql-indexes.htm]

[69] Strictly speaking, “key” implies uniqueness of values in a column, while “index” can be “non-unique”, i.e., it does not require following this rule.

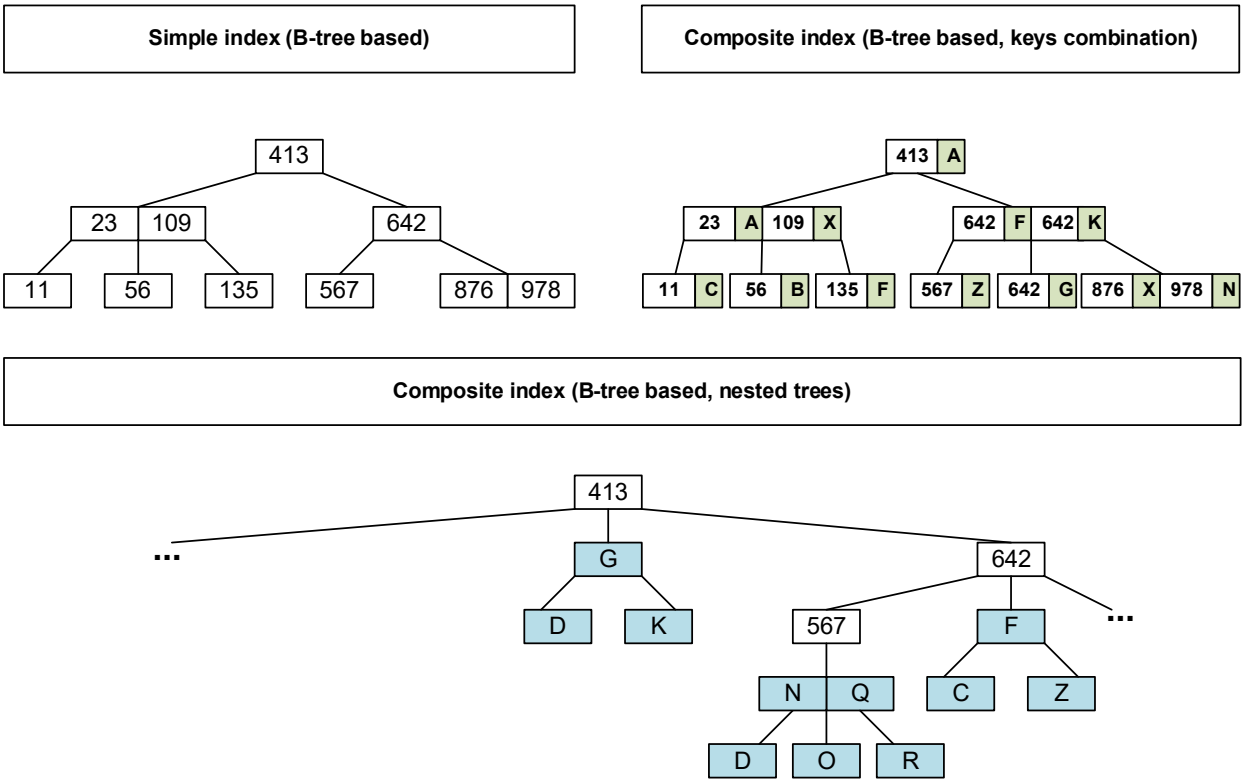


Figure 2.4.b — B-tree based simple and composite indexes structure

By records uniqueness indexes can also be compared to keys{32} in the sense that the concepts of “unique index” and “key” even at the level of SQL syntax are considered by some DBMSes as synonyms. At the same time, it is worth remembering that the relational theory operates with the notion of “key”, while “index” appears at the level of database implementation in a particular DBMS.

!!! **Unique index**[70] — an index built on the table field(s) containing unique values.

Simplified: an index built on a field that is the primary or alternate key of a table.

!!! **Non-unique index**[71] — an index built on the table field(s) that **do not** contain unique values.

Simplified: “just an index” (for non-unique index in SQL syntax does not even have a dedicated keyword, we just write `INDEX`*).*

Both unique and non-unique indexes can be simple{105} or composite{105}.

The purpose of the non-unique index is only to speed up search operations on the indexed fields, while the unique index is associated with the control of uniqueness of the values of the indexed field(s).

Different DBMS in this context offer different possibilities:

- in MySQL, the only way to ensure the uniqueness of the values of the field(s), which is not the primary key, is to create on this field(s) a unique index;
- in MS SQL Server, enabling of field uniqueness property leads to creation of unique index on this field (and vice versa — creation of unique index on a field leads to enabling of field uniqueness property);

[70] **Unique index** — an index on the basis of some superkey.

[71] **Non-unique index, index** — see index{99}.

- in Oracle, the uniqueness of field values can be provided both with and without a unique index, but for performance reasons, it is recommended to build a unique index on the field with unique values.

Any modern DBMS supports both unique and non-unique indexes.

By records order indexes are divided into clustered indexes (and primary indexes as their subtype) and non-clustered indexes.



Clustered index[72] — an index built on the field (possibly with non-unique values), by which the physical ordering of the data in the file is done.

Simplified: the table data is physically ordered on disk by the value of the indexed field; field values can be repeated.



Primary index[73] — an index, built on the field with unique values, by which the physical ordering of the data in the file is done.

*Simplified: the table data is physically ordered on disk by the value of the indexed field; field values **cannot** be repeated.*



Non-clustered index[74] — an index built on a field for which **no** physical ordering of the data in the file has been done.

Simplified: "just an index", the ordering principles of which have nothing to do with the physical location of the data on disk.

First, let's explain how "primary key{36}" and "primary index{107}" are related.
In theory, "primary key" refers to relationships{53}, referential integrity{67} and normalization{157} — that is, the theoretical foundations of relational databases, and "primary index" refers to the way data is organized on disk, and storage engines{30} — that is, to the physical design level and the way a particular DBMS is implemented.

In practice, the situation in different DBMS is as follows:
- MySQL always builds the primary index on the primary key;
- MS SQL Server allows us to build a "clustered unique" (actually — primary) index on the field(s), which is not the primary key;
- Oracle allows us to use even non-unique indexes for indexing the primary key, and not to order table data by primary key (exactly in this formulation, i.e., Oracle does not operate with the concept of "clustered index", but clearly talks about "index-organized tables").

In short: most often "primary key" and "primary index" will be synonymous, but exceptions are possible (and quite common).

Now let's see what this looks like. Let's start with the clustered and primary indexes (figure 2.4.c).

[72] **Clustered index** — an index that is used when ordering field is not a key field — that is, if numerous records in the file can have the same value for the ordering field. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)
[73] **Primary index** — an index that is specified on the ordering key field of an ordered file of records. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)
[74] **Non-clustered index** — see index{99}.

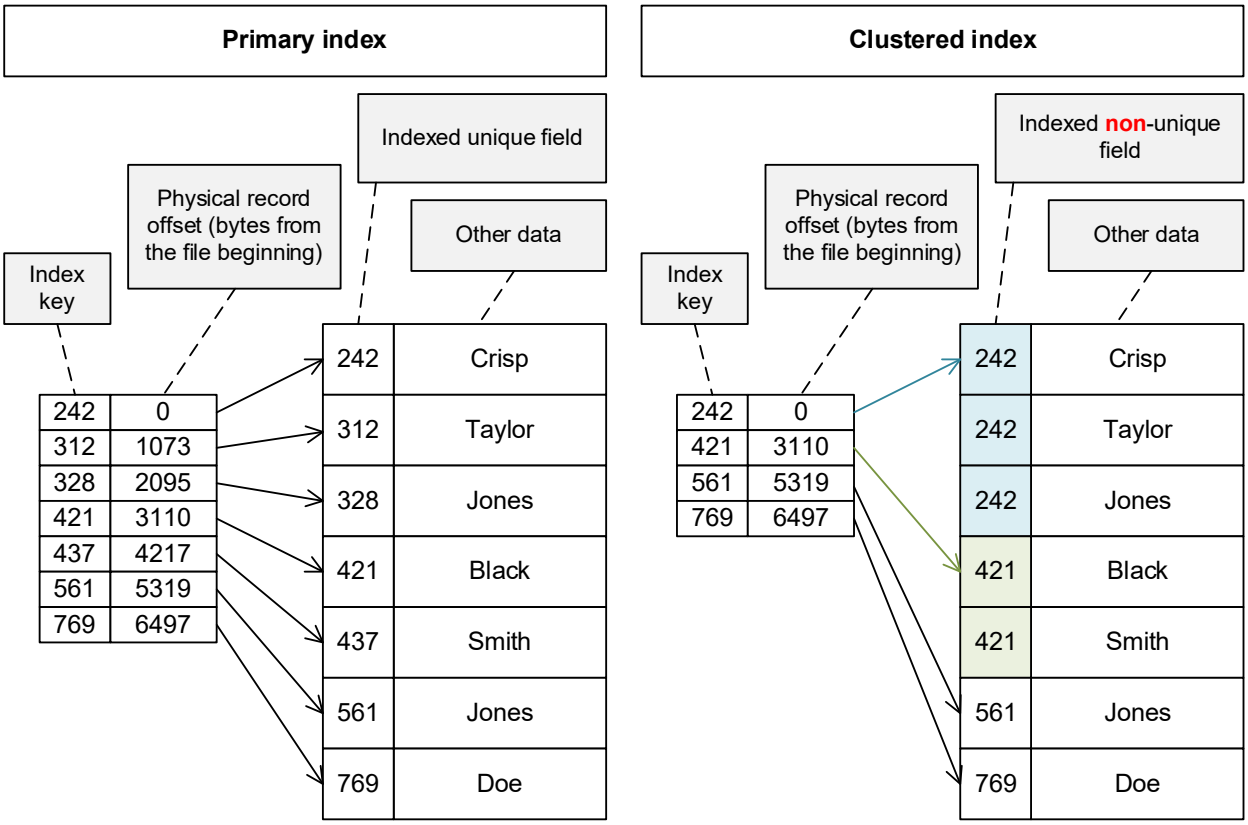


Figure 2.4.c — Schematic view of primary and clustered indexes

The main difference is that the primary index contains data about the location of each record **with a unique indexed field value**, and the clustered index contains data about the beginning of a block of records **with the same indexed field values**. If not to go into technical peculiarities of implementation of algorithms for building and using such indexes — in other respects, they are identical.



The primary index may also contain data about the location of a block of records rather than each record, see “sparse index {112}” further.

Non-clustered indexes differ from clustered indexes in that the sequence of records in the index and in the physical file does not coincide. As a rule, this leads to complication of index structure itself due to necessity of storing several different addresses of records with matching values of indexed field.

A schematic representation of the non-clustered index is shown in figure 2.4.d.

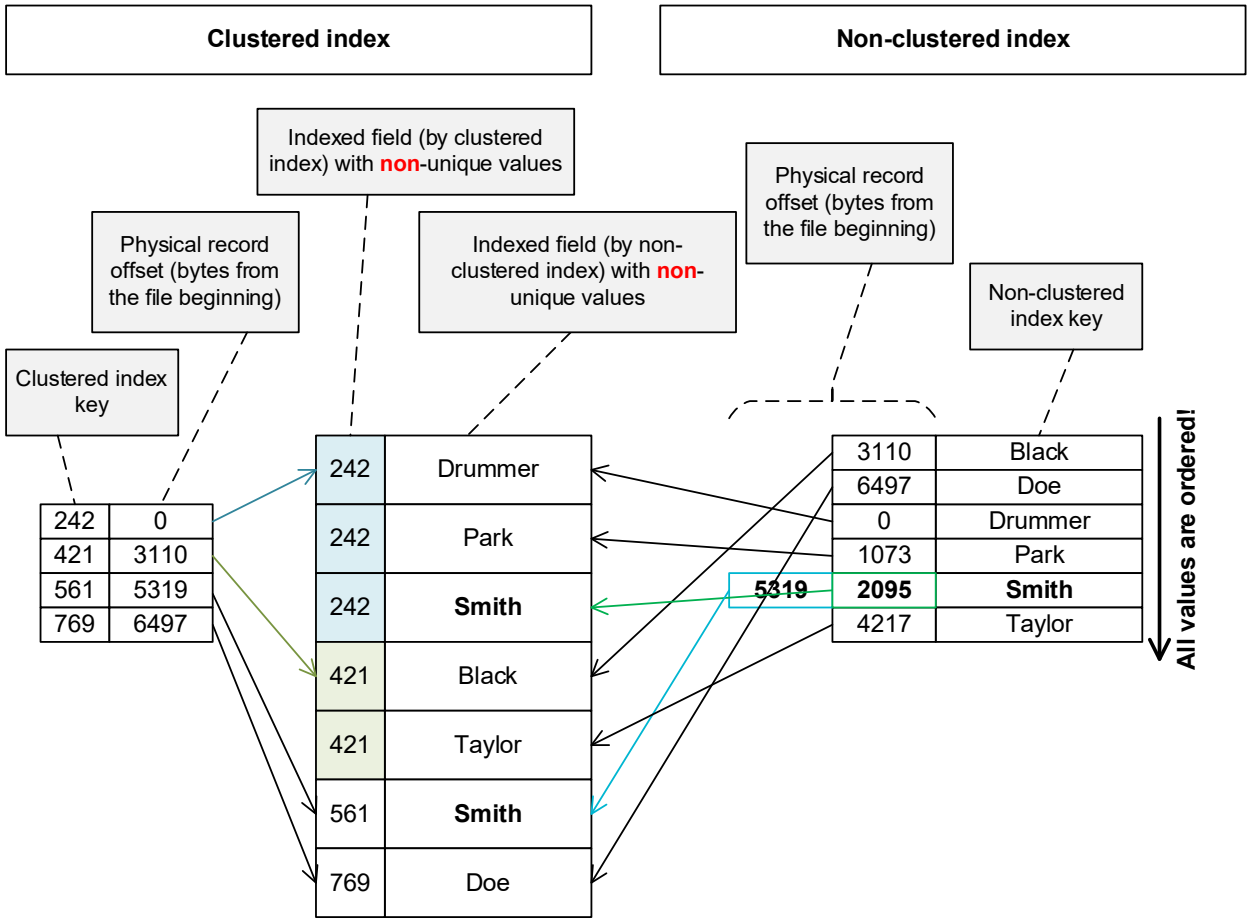


Figure 2.4.d — Schematic view of non-clustered index (in comparison to clustered one)

As you can see in figure 2.4.d, the keys of non-clustered index are ordered (in this case alphabetically) by the value of the field being indexed, but the data itself in the table remains ordered by another field (on which the clustered index is built).

In the case of non-uniqueness of the indexed field values (in our case such value is “Smith” in the non-clustered index it is necessary to build a block (a chain) of offsets for each record in the table indexed field of which contains the corresponding value.

Any modern DBMS supports both clustered and non-clustered indexes, but in some cases (for example, in MySQL) clustered index is built automatically (on the primary key, or the first unique index or specially added field[75]), i.e., it is not always possible to create “a clustered index we want”.



See a detailed description of primary, clustered and secondary (non-clustered) indexes in “Fundamentals of Database Systems” (by Ramez Elmasri, Shamkant Navathe), 6th edition, chapter 18.1 “Types of Single-Level Ordered Indexes”.

[75] “Clustered and Secondary Indexes” [https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html]

By storage structure indexes (like tables in many storage engines{30}) can be partitioned and non-partitioned:



Partitioned index[76] — an index stored and processed as separate parts (sections, fragments) in order to improve performance.

Simplified: an index divided into several fragments.



Non-partitioned index[77]) — an index stored and processed as a single data structure.

Simplified: "just an index" (without using special commands by default all indexes are created as non-partitioned).

The idea of index partitioning is very close to the idea of table partitioning — a way of storing table data in several parts (which allows us to place them on separate physical drives and process them in parallel).

Depending on the DBMS, there may be the following relationship between table and index partitioning:

- MySQL allows us to partition tables and indexes only simultaneously (i.e., we cannot build a non-partitioned index on a partitioned table or a partitioned index on a non-partitioned table[78]);
- MS SQL Server allows us to use any combination of index and table partitioning (i.e., to build partitioned and non-partitioned indexes on both partitioned and non-partitioned tables[79]);
- Oracle allows us to use any combination of index and table partitioning (i.e., to build partitioned and non-partitioned indexes on both partitioned and non-partitioned tables[80]).



In the vast majority of cases (of DBMSes and storage engines{30}) the partitioning of cluster indexes{107} and/or clustered tables are not supported.

Since the implementation features of partitioned indexes depend very much on the chosen DBMS (and even its version), we will limit ourselves to a schematic representation of several options for partitioning indexes and tables, using the example of how it is implemented in Oracle (figure 2.4.e).



Be sure to carefully read the documentation for your DBMS and/or chosen storage engine. The unreasonable use of partitioned indexes and/or tables may significantly reduce DBMS performance and lead to other undesirable consequences.

[76] **Partitioned index** — an index broken into multiple pieces. ("Understanding Partitioned Indexes in Oracle 11g", Richard Niemiec). [https://logicalread.com/oracle-11g-partitioned-indexes-mc02/]

[77] **Non-partitioned index, index** — see index{99}.

[78] "Overview of Partitioning in MySQL" [https://dev.mysql.com/doc/refman/8.0/en/partitioning-overview.html]

[79] "Create Partitioned Tables and Indexes" [https://technet.microsoft.com/en-us/library/ms188730(v=sql.130).aspx]

[80] "Partitioning Concepts" [https://docs.oracle.com/cd/E11882_01/server.112/e25523/partition.htm]

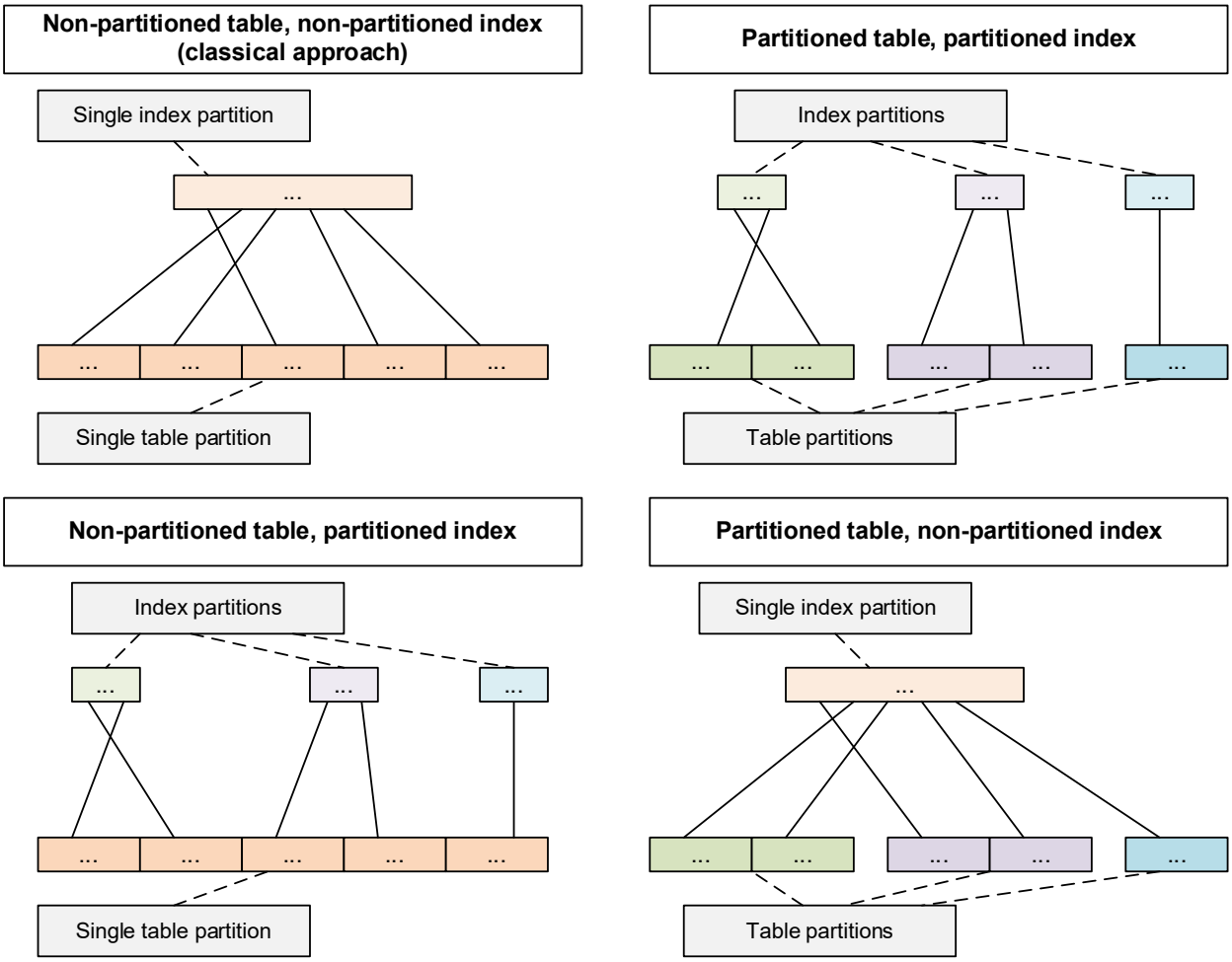


Figure 2.4.e — Schematic view of partitioned indexes and tables options

And to conclude the discussion of partitioned indexes, let's schematically present the purpose of creating both partitioned indexes and partitioned tables themselves (figure 2.4.f), which is to enable distribution of data storage across different devices and parallel processing of such partitioned data.

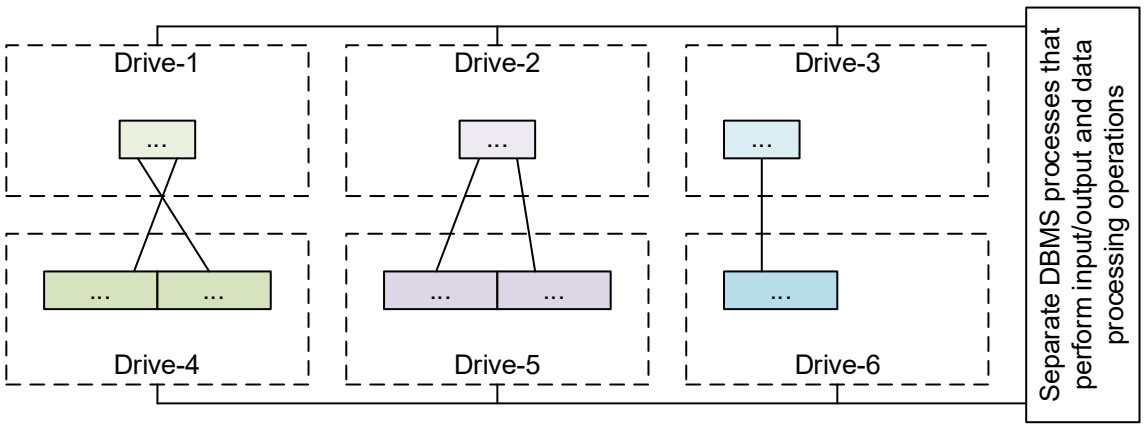


Figure 2.4.f — Schematic view of partitioned indexes and tables creation purpose

Most modern DBMSes support both partitioned and non-partitioned indexes, but the features of their implementation and available features are very different in different DBMSes and storage engines[30].

By density indexes can be dense and sparse:

!!! **Dense index**[81] — an index containing a pointer to the location of the record for each value of the field being indexed.

Simplified: the index stores the addresses of each table record.

!!! **Sparse index**[82] — an index that contains a pointer to the location of a block of records for each value (if non-unique) or group of values (if unique) of the indexed field.

Simplified: the index stores addresses of blocks (groups) of records.

The difference between dense and sparse indexes is easiest to explain in the context of the previously discussed primary{107}, clustered{107} and non-clustered{107} indexes.

Despite the fact that in theory primary indexes can be sparse (this will be discussed in a moment), in practice almost all DBMSes have dense primary indexes.

Let's clarify the situation with dense and sparse primary indexes. By definition{107} such indexes are built on table fields containing unique values. But the index itself (as shown in figure 2.4.g) may contain pointers both to each individual record and to a block of records (in this case specific values of the indexed field will be in the range "from the current value in the index inclusive to the next value in the index exclusive").

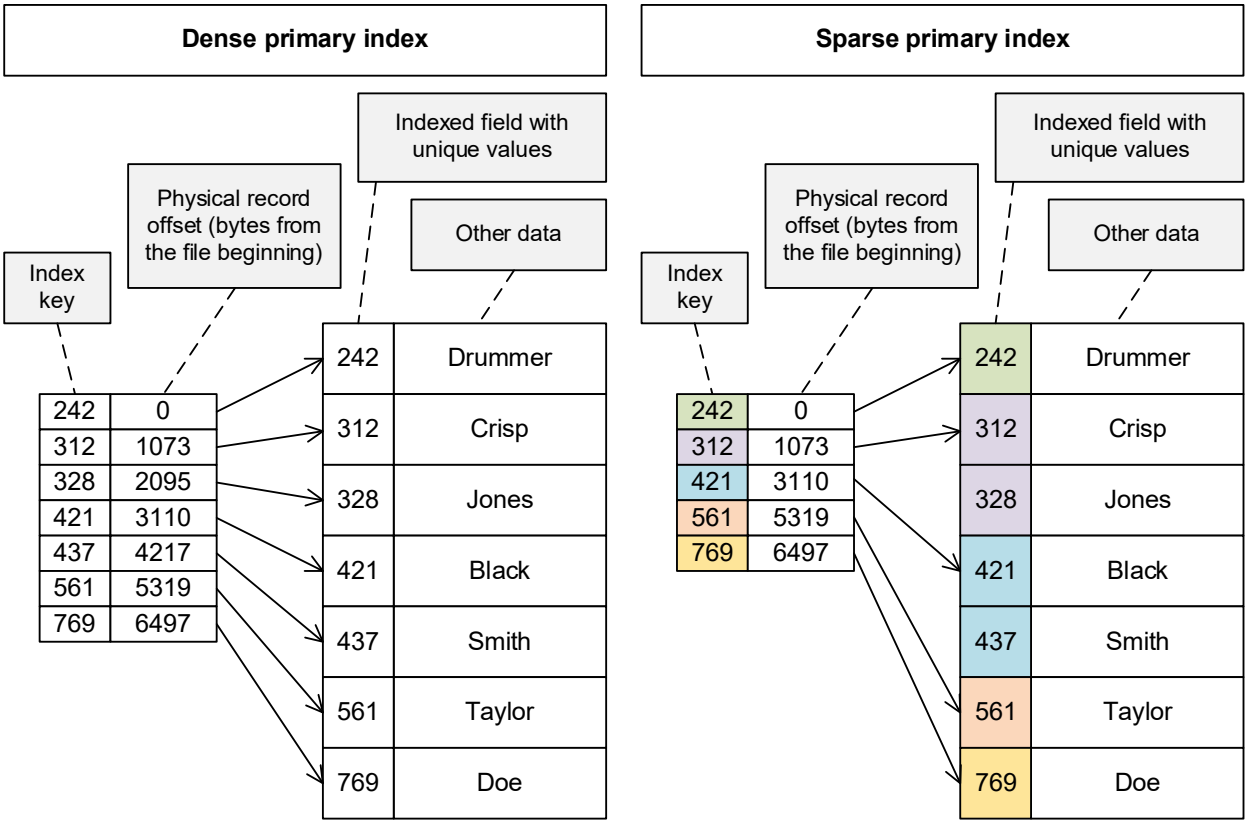


Figure 2.4.g — Dense and sparse primary indexes

[81] **Dense index** — an index that has an entry for every search key value (and hence every record) in the data file. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

[82] **Sparse index** — an index that has entries for only some of the search values. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

The advantages of sparse indexes are their smaller size and the possibility of rarer updates. But this advantage pales in comparison to the disadvantage of having to refer to a table to search for a specific record whose indexed field value is included in a particular block. That is why in practice DBMS developers prefer to use dense primary indexes.

It is worth noting that clustered (non-primary) indexes do not suffer from the just mentioned disadvantage: since the indexed field in the corresponding record block will have the same values, for further execution of some operations (for example, extraction of other data not covered by the index) the DBMS will always have to access the table file, and for other operations (for example, counting the number of unique values) the information stored in the index is sufficient.

It should be obvious from the above reasoning that non-clustered indexes must be dense, because the idea of “block of records with some ordering or some common feature” is irrelevant for them.

Thus, clustered indexes can be considered a classic case of sparse indexes, and non-clustered indexes can be considered a classic dense indexes case (figure 2.4.h).

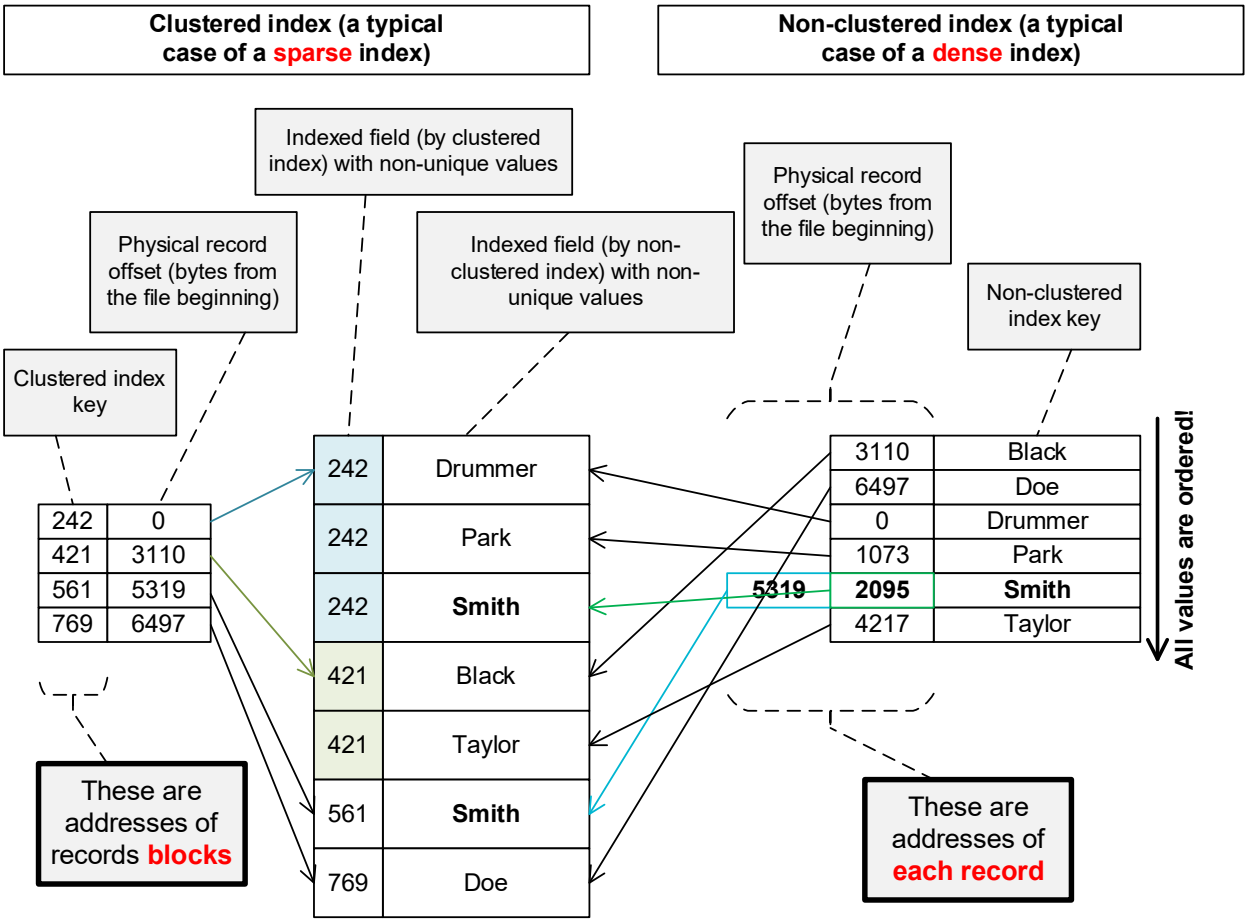


Figure 2.4.h — Clustered and non-clustered indexes as classical cases of sparse and dense indexes

Most modern DBMSes support both dense and sparse indexes, but in the vast majority of cases dense indexes are preferred, and the available choices are very limited.

By basic structure indexes can be represented by a very large number of varieties, the main ones being:



B-tree[83] based index — an index organized using a B-tree (balanced tree), optimized for range-based searches and for operations with large blocks of data. Allows storing part of the index in external memory.

Simplified: one of the main forms of index organization in most DBMSes and storage engines. (The letter "B" in the index name comes from the word "balanced".)



T-tree[84] based index — an index organized using a T-tree (a kind of balanced tree that stores pointers to data addresses in memory instead of the data itself), optimized for operations when both the index and the data are entirely in RAM.

Simplified: an index for DBMSes and storage engines that assume storage of all processed data in RAM. (The letter "T" in the name of the index comes from the graphical representation of the T-tree nodes in the article in which it was first introduced.)



R-tree[85] based index — index organized using R-tree (a special form of geographic and geometric data representation), optimized to perform operations with specific data types storing geographic coordinates or information on geometric shapes.

Simplified: an index to speed up geographic and geometric data processing (the letter "R" in the index name comes from the word "rectangle").



Hash-table[86] based index — an index organized using a hash-table (a special structure for storing key-value pairs), optimized to perform searches based on strict comparison and handling relatively rarely changed data.

Simplified: one of the main forms of index organization in most DBMSes and storage engines (along with B-tree based indexes).



Bitmap[87] based index — an index organized using a bitmap (a special structure for storing information about the presence of a particular value in a field), optimized to work with columns with relatively few different values.

Simplified: the index stores in a very compact form an indication of the presence in some cell of a column of one of the values present in the whole column.

The description of the data structures and algorithms for their processing is far beyond the scope of this book, but let's give a schematic representation of each of these indexes (figures 2.4.i, 2.4.j, 2.4.k, 2.4.l, 2.4.m), and then move on to the areas of their application.

---

[83] **B-tree** — a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. ("Wikipedia") [https://en.wikipedia.org/wiki/B-tree]

[84] **T-tree** — a balanced index tree data structure optimized for cases where both the index and the actual data are fully kept in memory, just as a B-tree is an index structure optimized for storage on block oriented secondary storage devices like hard disks. ("Wikipedia") [https://en.wikipedia.org/wiki/T-tree]

[85] **R-tree** — a tree data structure used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons. "Wikipedia" [https://en.wikipedia.org/wiki/R-tree]

[86] **Hash-table** — a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. "Wikipedia" [https://en.wikipedia.org/wiki/Hash_table]

[87] **Bitmap** — an array data structure that compactly stores bits. It can be used to implement a simple set data structure. A bitmap is effective at exploiting bit-level parallelism in hardware to perform operations quickly. "Wikipedia" [https://en.wikipedia.org/wiki/Bit_array]

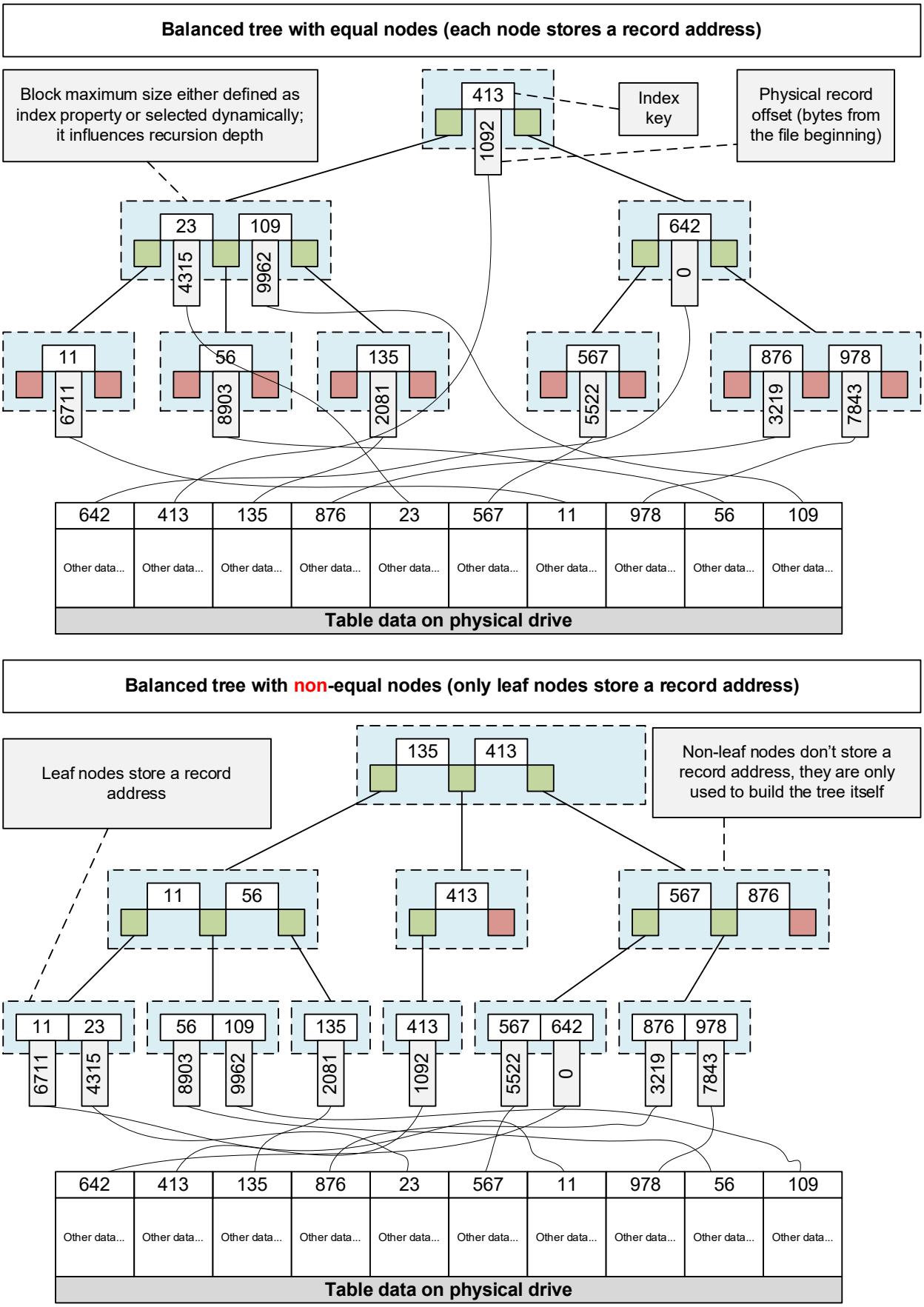


Figure 2.4.i — Schematic view of B-tree based index

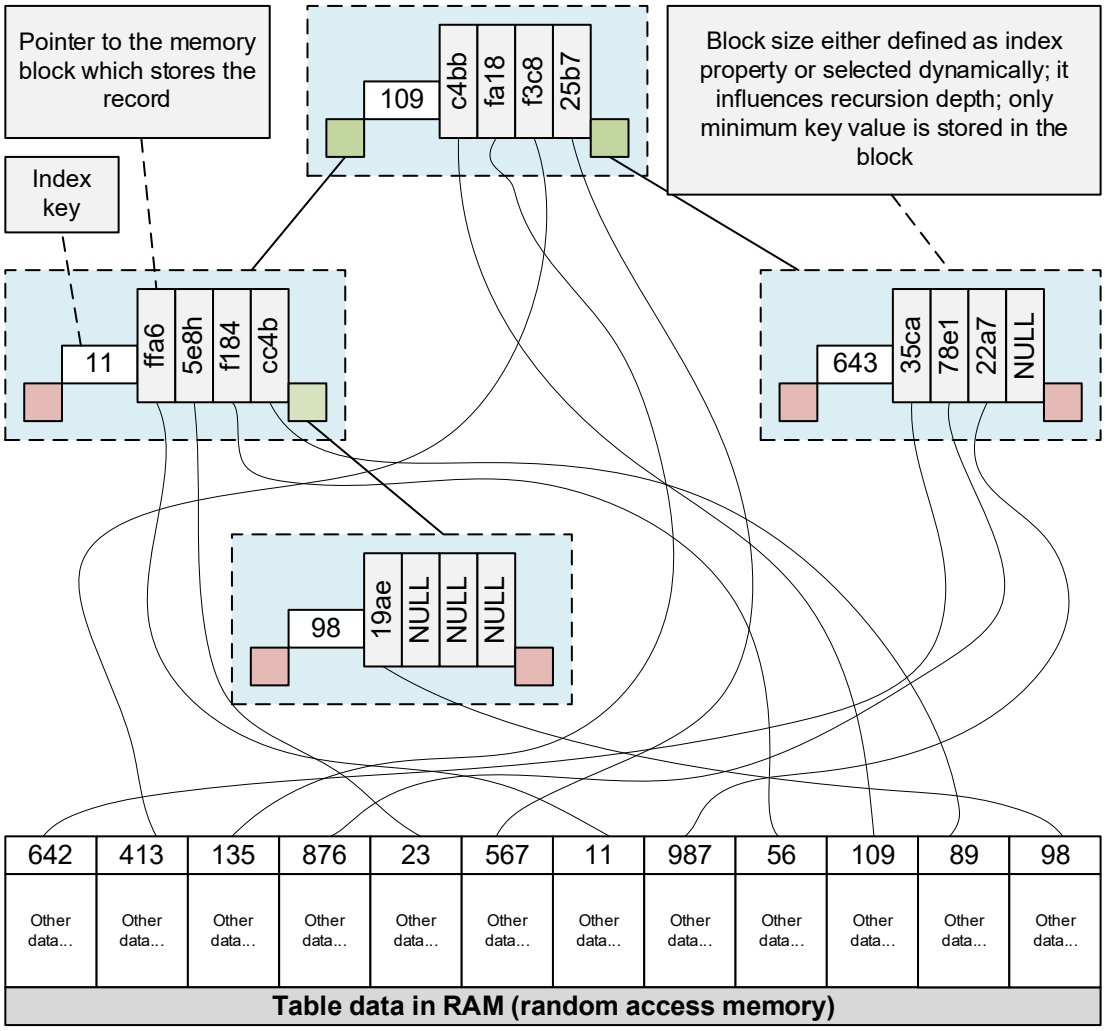


Figure 2.4.j — Schematic view of T-tree based index

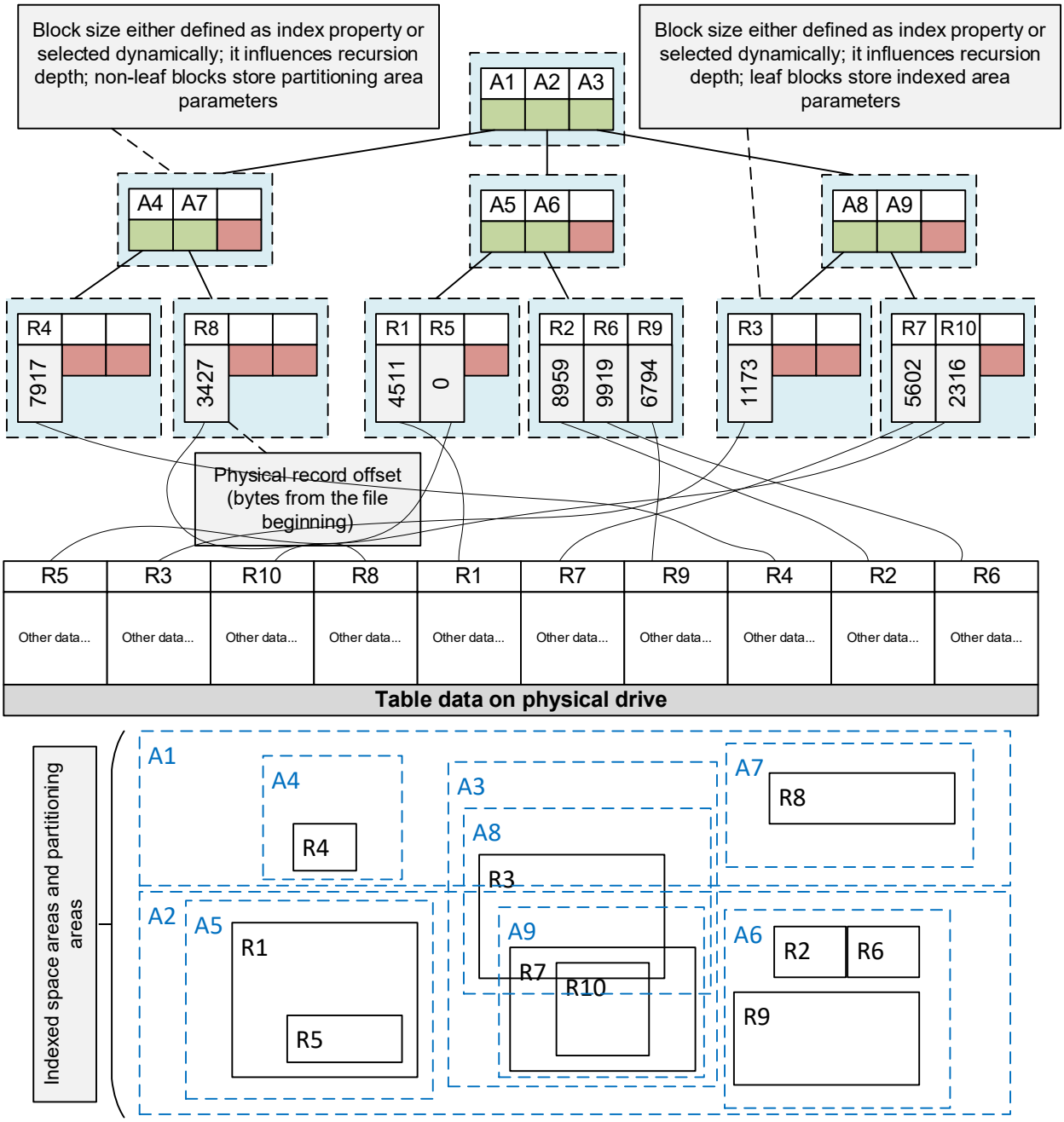


Figure 2.4.k — Schematic view of R-tree based index

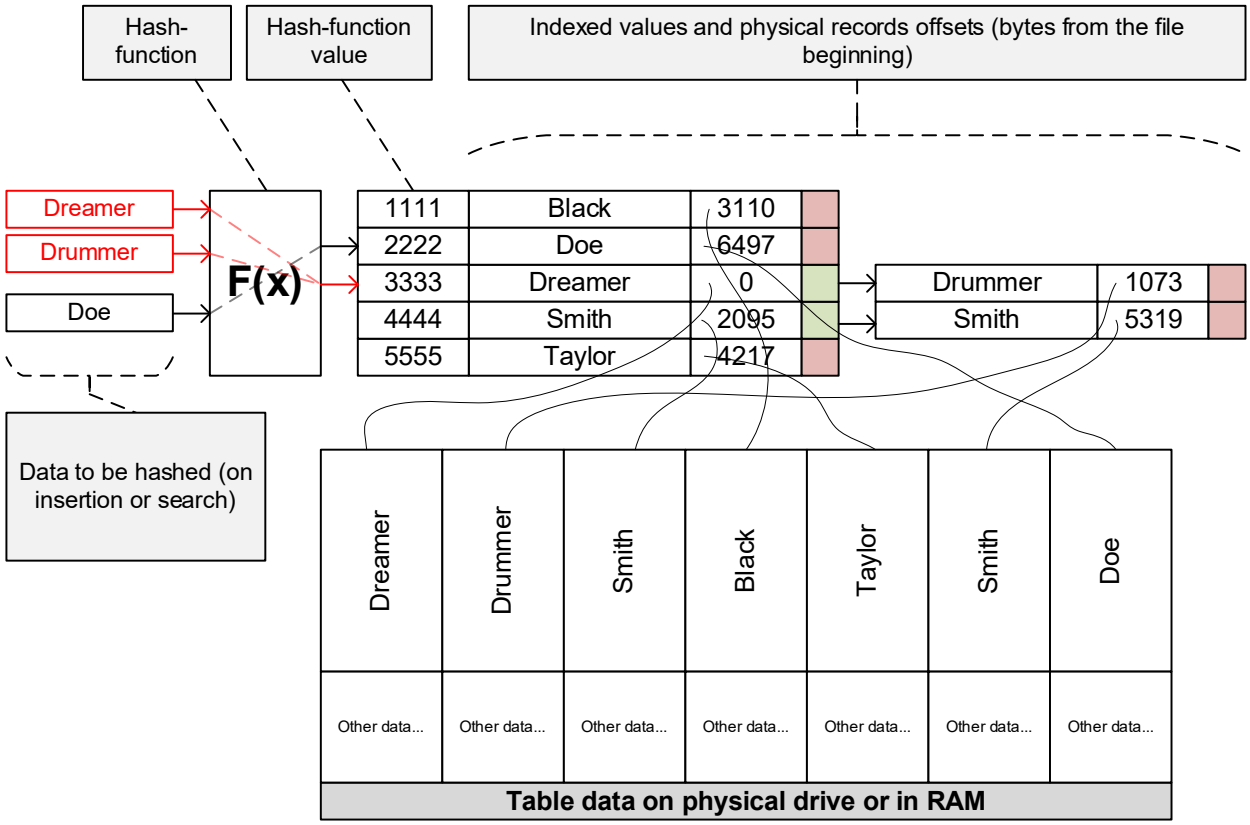


Figure 2.4.l — Schematic view of hash-table based index

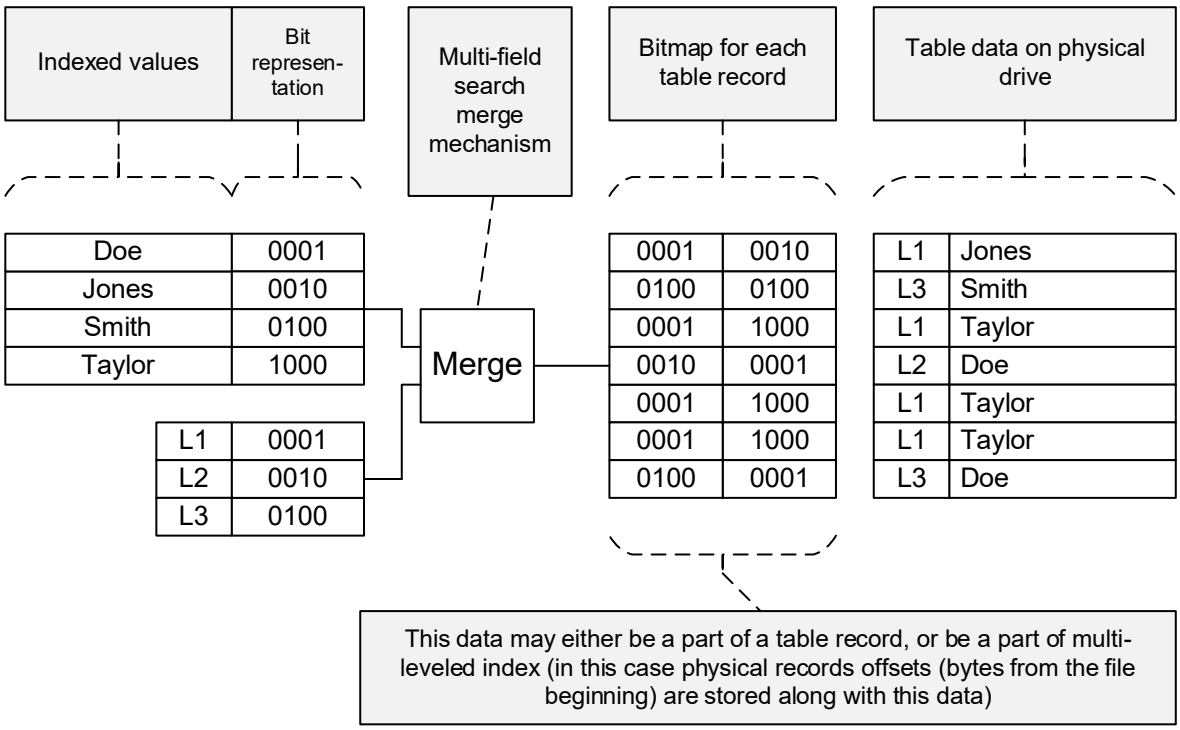


Figure 2.4.m — Schematic view of bitmap based index

Now let's look at the main areas of application for each type of index presented in this classification. Note: we are not talking about advantages and disadvantages here, because they depend very much on the specific situation.

Index based on...		**B-tree**	**T-tree**	**R-tree**	**Hash-table**	**Bitmap**
The main area of application...		Typical form of implementation of most indexes, effective for =, >, ≥, <, ≤ operations, can be loaded into memory in parts	Mainly for so-called "in-memory" databases (when all data is held in RAM)	Spatial and geometric data	The second most frequently used (after B-trees) form of index implementation, effective for the = and != operations	Low power columns (small number of different values)
Can be...	Simple	Yes	Yes	Yes	Yes	Yes
	Composite	Yes	Yes	Theoretically	Yes	Yes
	Unique	Yes	Yes	Theoretically	Yes	Yes
	Non-Unique	Yes	Yes	Yes	Yes	Yes
	Clustered	Yes	Theoretically	Theoretically	Theoretically	Yes
	Nonclustered	Yes	Yes	Yes	Yes	Yes
	Partitioned	Yes	Theoretically	Theoretically	Theoretically	Theoretically
	Unpartitioned	Yes	Yes	Yes	Yes	Yes
	Dense	Yes	Yes	Yes	Yes	Yes
	Sparse	Yes	Theoretically	Theoretically	Theoretically	Yes
	Single-Level	Extremely rarely	Extremely rarely	Theoretically	Yes	Yes
	Multi-level	Yes	Yes	Yes	Theoretically	Theoretically
	Covering	Yes	Yes	Yes	Yes	Yes
	Non-covering	Yes	Yes	Yes	Yes	Yes

Here the options "extremely rare" and "theoretically" mean that in principle it is possible to create such an index, but in practice it is not used or is used in very rare cases either due to low efficiency or high complexity of implementation.

In the context of different DBMSes, the situation is as follows: the vast majority of DBMSes support indexes based on balanced trees and hash-tables, many support indexes based on R-trees, but indexes based on T-trees and bitmaps are least common.

By hierarchy indexes are divided into single-level and multi-level, and both of these varieties have already been met in this chapter, even though without explicit emphasis.



Single-level index[88] — an index, the structure of which is flat, i.e., contains exactly one level.

Simplified: an index without hierarchy.



Multi-level index[89] — an index the structure of which is hierarchical, i.e., contains two or more levels; such levels may contain the same type of information or differ in their purpose (as a rule, leaf nodes will differ from non-leaf nodes, while all non-leaf nodes regardless of their level perform the same role).

Simplified: an index with a hierarchy (usually tree based).

Typical representatives of single-level indexes are:

- primary index{107};
- clustered index{107};
- hash-table based index{114};
- bitmap based index{114}.

Typical representatives of multi-level indexes are:

- B-tree based index{114};
- T-tree based index{114};
- R-tree based index{114};
- XML-index{132}.

It is worth noting that theoretically any multi-level index can be reduced to a single-level index and vice versa, but in practice indexes are of these two types because this organization is the most effective for solving the tasks assigned to an index.

Most modern DBMSes support both single-level (at least clustered and/or primary) and multi-level (usually B-tree based) indexes.

By correlation with SQL query indexes can be both covering and non-covering, and this division in the entire classification considered by us is the only conventional, i.e., the same index can be both covering and non-covering, because there is a dependency here not only on the index itself, but also on the SQL-query.



Covering index[90] — an index that explicitly contains information inside itself that is sufficient to execute an SQL query without accessing the data stored outside this index (in the table itself).

Simplified: an index in which there is enough information to execute an SQL query without accessing the data in the table.



Non-covering index[91] — an index, which only allows DBMS to speed up finding the information id needs, while the information it is looking for is not contained within the index or is not contained in full.

*Simplified: "just an index", which does **not** contain enough information to execute an SQL query without accessing the data in the table.*

[88] **Single-level index** — an index based on ordered file. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

[89] **Multi-level index** — an index based on tree data structure. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

[90] **Covering index** — an index that contains all information required to resolve the query is known as a "Covering Index"; it completely covers the query ("Using Covering Indexes to Improve Query Performance", Joe Webb). [https://www.simple-talk.com/sql/learn-sql-server/using-covering-indexes-to-improve-query-performance/]

[91] **Non-covering index** — see index{99}.

Let's demonstrate the essence of query index coverage with an example. Suppose we have the following `users` table:

Primary key, clustered index	Composite non-clustered index {u_email, u_status}		There are no indexes on these columns	
u_id	**u_email**	**u_status**	**u_login**	**u_name**
1	jones@mail.uk	Active	jones	Jone Jones
2	james@mail.uk	Active	james	Kane James
3	taylor@mail.uk	Locked	taylor	Tim Taylor
4	smith@gmail.com	Active	smith	Samuel Smith
5	dow@yahoo.com	Locked	dow	Daniel Dow

Let's start considering queries with a rather controversial example:

MySQL — The first example of an index-covered query

```
1    SELECT COUNT(`u_id`)
2    FROM   `users`
3    WHERE  `u_id` >= 2
4           AND `u_id` <= 10
```

On the one hand, yes — index on `u_id` field contains all necessary information for query execution, and data in table can be left untouched. I.e., this index is a covering one for this query.

On the other hand, in some DBMSes and storage engines[30] a clustered index is, physically, nothing more than the table itself (indicating the fact that it is ordered by some field). Then formally here index can be considered as not covering, although the speed of the query execution will still be higher than if there was no index.

Let's look at a more classic example:

MySQL — The second example of an index-covered query

```
1    SELECT `u_status`
2    FROM   `users`
3    WHERE  `u_email` = 'jones@mail.uk'
```

For this query the index `{u_email, u_status}` is a covering one, since it contains the values of all the fields used in the query (the DBMS can search for the first `u_email` field in a complex index as fast as for the whole index, and the retrieved `u_status` field value is also contained in the index).

Finally, here is an example of a non-covered query:

MySQL — An example of a non-covered query

```
1    SELECT `u_login`
2    FROM   `users`
3    WHERE  `u_email` = 'jones@mail.uk'
4       AND `u_status` = 'Active'
```

The `{u_email, u_status}` index is not a covering one for this query, because even though it will help the DBMS to find the right record very quickly, the `u_login` field value will have to be extracted from the table data itself.

Obviously, the question of which DBMS supports covering and non-covering indexes is meaningless: any index can be both covering and non-covering, depending on the SQL query being executed.

By specific functions the indexes are not so much divided into groups as they get their specific names in different DBMS. Also note that more and more specific indexes appear every year, so the list below is probably incomplete.



Columnstore index[92] — a technology for storing and processing data in such a way that the basic unit is not a row (as in the classical case), but a column.

Simplified: an index for fast processing of individual columns.

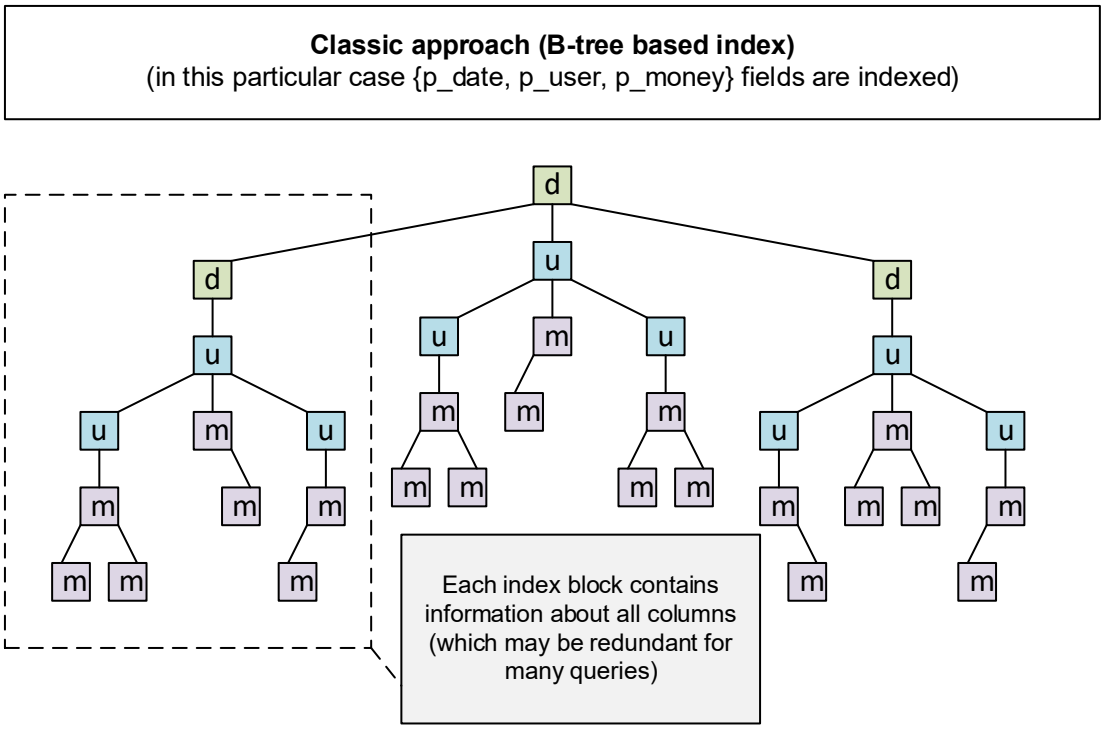
This type of index was introduced in MS SQL Server 2012, came there from the field of data warehouses and is called “a technology” for a reason. In fact, it is not only and not so much an index as a way to organize data in a table (if this index is a clustered index) or a way to access individual table columns very quickly.

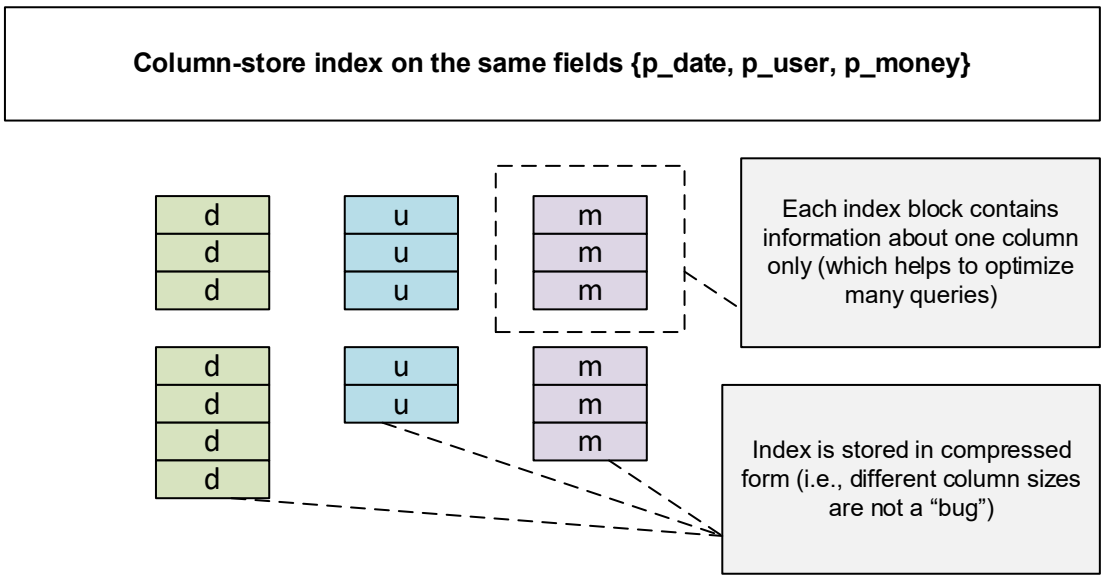
Let’s demonstrate the structure of such an index graphically in comparison with an ordinary index (figure 2.4.n).

It is worth realizing that a columnstore index is not “magic” and does not allow us to improve each query (moreover, in a “typical database”, where search queries prevail over analytical ones, such an index would be more harmful than useful).

But in the example in figure 2.4.n, queries like “show the number of payments by dates”, “show the average and median of payments”, “show the minimum and maximum payments” and the like will work faster with a columnstore index.

[92] **Columnstore index** — a technology for storing, retrieving and managing data by using a columnar data format, called a columnstore (“Columnstore Indexes Guide”). [https://msdn.microsoft.com/en-us/library/gg492088(v=sql.130).aspx]





Typical data sample. For such data column-store index may be efficient during a lot of analytical queries.

p_id	p_user	p_money	p_date	...
1	234	34556	2016-02-12	...
2	89	565	2016-03-18	...
3	34	341235	2015-09-02	...
4	2342	24234	2017-02-14	...
...	...	...	...	...
34526256	34235	21321	2016-12-19	...

Figure 2.4.n — Schematic view of column-store index

!!! **Index with included columns**[93] — a non-clustered{107} index that contains in its leaf nodes information from an additional field that is not used when constructing the index itself.

Simplified: an index that allows us to cover {120} more queries by containing more data.

Let's explain the difference between a composite{105} index and an index with included columns. In a composite index all fields are used to build the index, but in an index with included columns the data from these included columns are just stored and do not affect the index logic (figure 2.4.o).

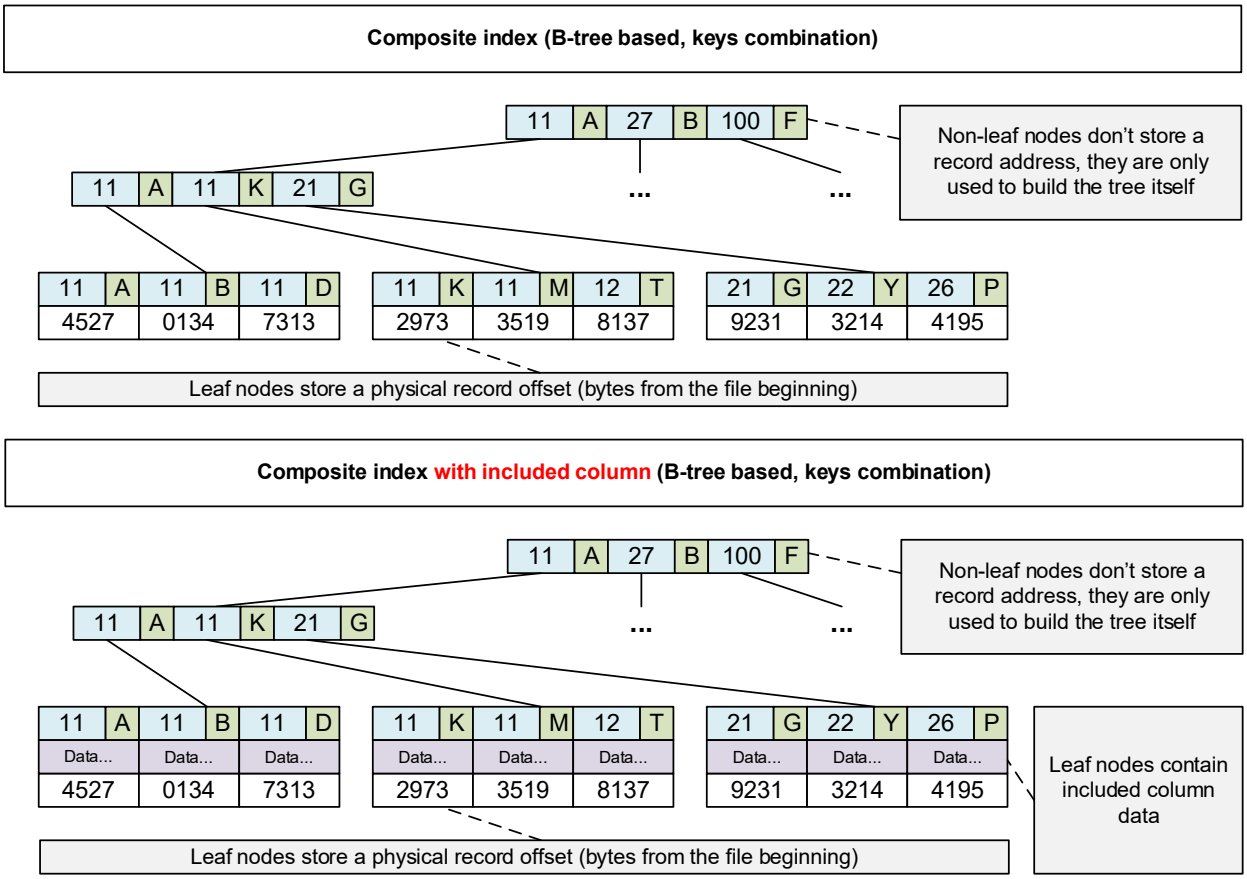


Figure 2.4.o — Schematic structure of composite index and composite index with included column

The presence of data from the included column in the leaf nodes allows the BDMS not to refer to the table while executing the query in which we are interested in that data from the included column; that allows such index to cover queries that would not be covered by "normal index", and at the same time to avoid memory overflow (due to the fact that data from the included column is not considered during index construction and is not stored in the non-leaf nodes).

Obviously, such an index cannot be used for searching through the values of an included column, it just allows to quickly retrieve these values, but the search itself must be performed through the values of the main index fields.

This index is used in MS SQL Server, but it is not very popular in other DBMSes.

[93] **Index with included columns** — a nonclustered index that includes nonkey columns to cover more queries ("Create Indexes with Included Columns"). [https://msdn.microsoft.com/en-us/library/ms190806.aspx]



Index on computed columns[94] — an index built on the values of virtual columns (whose data often is not physically stored in the database).

Simplified: an index on the column whose value is calculated by the DBMS itself.



Function-based index[95] — an index based on the results of applying a function (built-in or user-defined) to the stored data.

Simplified: an index on the results of applying a function to the data (the value of the function is calculated by the DBMS itself).

These two types of indexes are considered together for a reason, because they are essentially the solution to the same problem, but the first (on computed columns) is used in MS SQL Server, and the second (function-based) is used in Oracle.



Since version 11, Oracle also supports computed columns, but despite the same name, the implementation (and capabilities) of this mechanism in MS SQL Server and Oracle have serious differences. Be sure to read the relevant sections of the documentation carefully.

The problem solved by using these indexes can be formulated as speeding up access to data whose values are calculated at the DBMS level, rather than transferred and stored in the database explicitly (as happens in the vast majority of cases). It is impossible (or extremely difficult and inefficient) to build “classical” indexes on such data, which is why some DBMSes offer the solution considered here.

Let’s explain it graphically (figure 2.4.p). Suppose that (for MS SQL Server) in some operations we need to search people by their initials very quickly (for this purpose we create a computed column `u_initials`), and (for Oracle) that for other operations we need to search by full name in upper case, where data are in the order of first name, middle name, last name (for this purpose we create a function, whose values are indexed).

Technically, such indexes can be implemented in almost any of the previously discussed ways, but most often they will be dense{112} non-clustered{107} B-trees{114} based indexes.

[94] **Index on computed columns** — an index that is built on virtual column that is not physically stored in the table, unless the column is marked persisted; a computed column expression can use data from other columns to calculate a value for the column to which it belongs (“Indexes on Computed Columns”) [https://msdn.microsoft.com/en-us/library/ms189292.aspx]

[95] **Function-based index** — rather than indexing a column, you index the function on that column, storing the product of the function, not the original column data; when a query is passed to the server that could benefit from that index, the query is rewritten to allow the index to be used (“Oracle Function-Based Indexes”). [https://oracle-base.com/articles/8i/function-based-indexes]

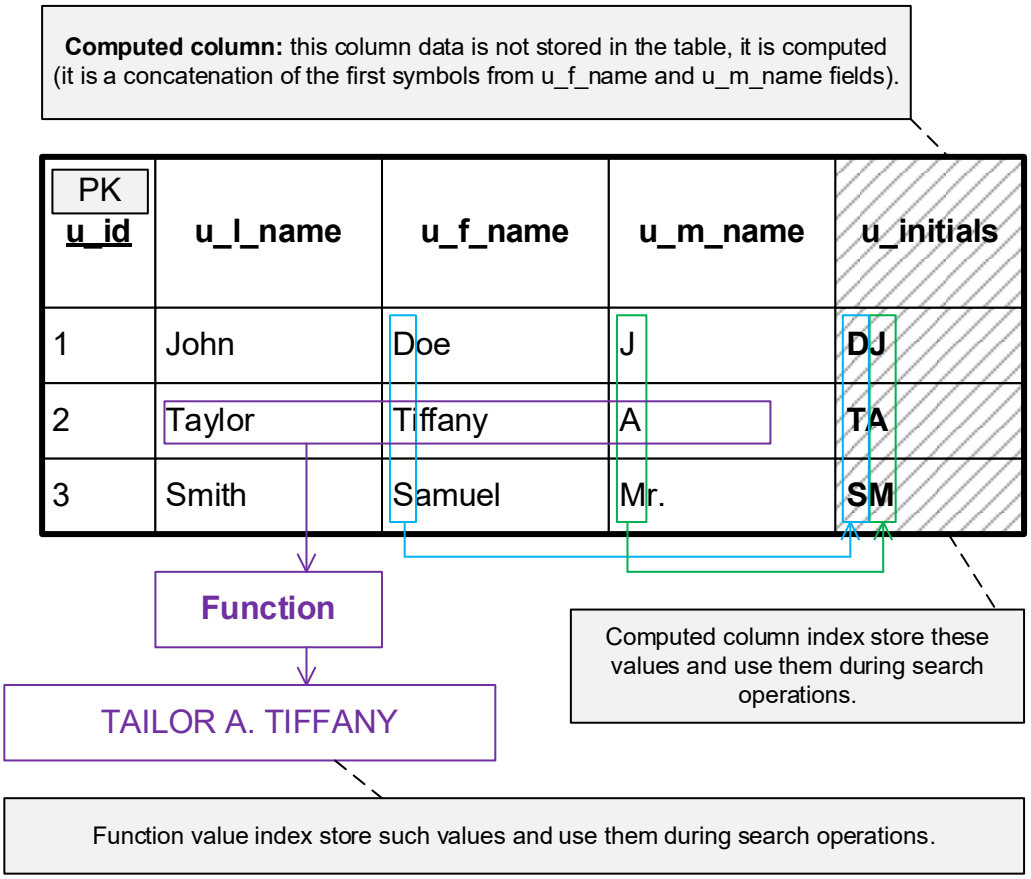


Figure 2.4.p — Indexes on computed columns and indexes on function values

As it was mentioned above, indexes of these two types are relevant primarily for MS SQL Server and Oracle, but there is a reason to believe that other DBMSes will introduce similar solutions in the foreseeable future.



Filtered index[96] — an index that takes into account only a (small) fraction of the entire set of values in the indexed column that satisfies the condition specified when the index was created.

Simplified: an index on parts of the column values.

In practice, there is often a need to perform operations that affect only a certain (usually small) part of data that meets some predefined criteria (e.g.: sending messages to active users only, displaying a list of only unfinished tasks, manual check only payments over a certain amount, etc.)

In such situations, it is effective to use filtered indexes, which contain information only about records that satisfy the specified criterion. Such indexes not only allow us to search faster, but also take less memory and are updated faster.

Schematically, the principle of the filtered indexes is shown in figure 2.4.q (let's assume that for some operations with the list of users we constantly need only those named "Taylor" or "Smith", and we don't need any others).

[96] **Filtered index** — an optimized nonclustered index especially suited to cover queries that select from a well-defined subset of data ("Filtered Indexes"). [https://docs.microsoft.com/en-us/sql/relational-databases/indexes/create-filtered-indexes?view=sql-server-ver15]

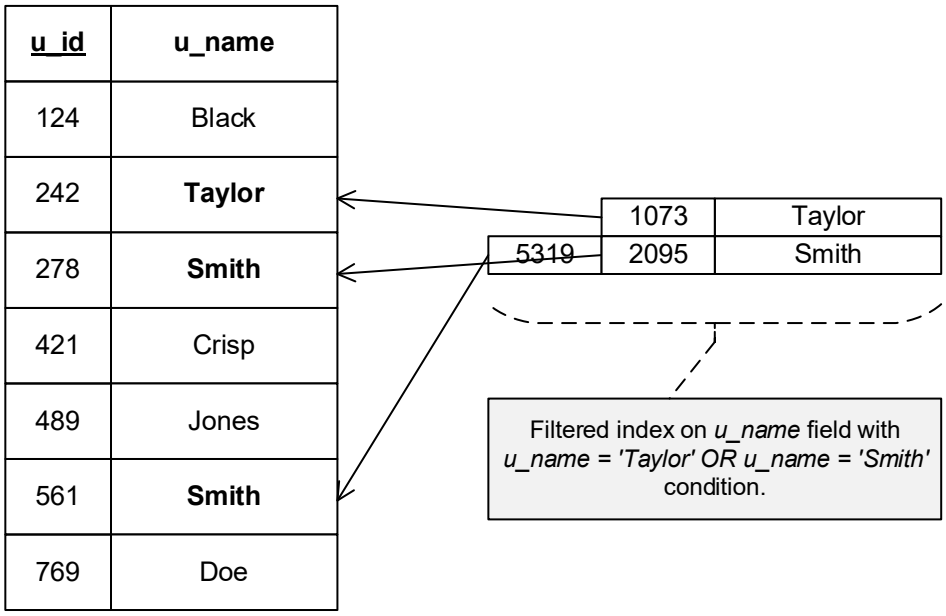


Figure 2.4.q — Filtered index basic idea

Filtered indexes are explicitly supported in MS SQL Sever, and in some other DBMSes can be emulated via conditional constructs[97].

!!! **Spatial** index[98] — an index optimized to speed up value retrieval operations in columns storing spatial or geometric data.

Simplified: an index on columns with spatial or geometric data type.

Such indexes can be constructed using R-trees{114} or based on B-trees{114} and multi-level partitioning of the space into homogeneous nested areas (such a solution is used in MS SQL Server and is shown in figure 2.4.r — the number of levels may vary in different implementations, but in MS SQL Server it is four).

The domain of application of spatial indices is limited (as follows from their definition) to spatial and geometric data, but here they allow to optimize the performance of many operations very effectively.

[97] Filtered indexes emulation in Oracle: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:4092298900346548432

[98] **Spatial index** — a type of extended index that allows you to index a spatial column (a table column that contains data of a spatial data type, such as geometry or geography) (“Spatial Indexes Overview”). [https://msdn.microsoft.com/en-us/library/bb895265.aspx]

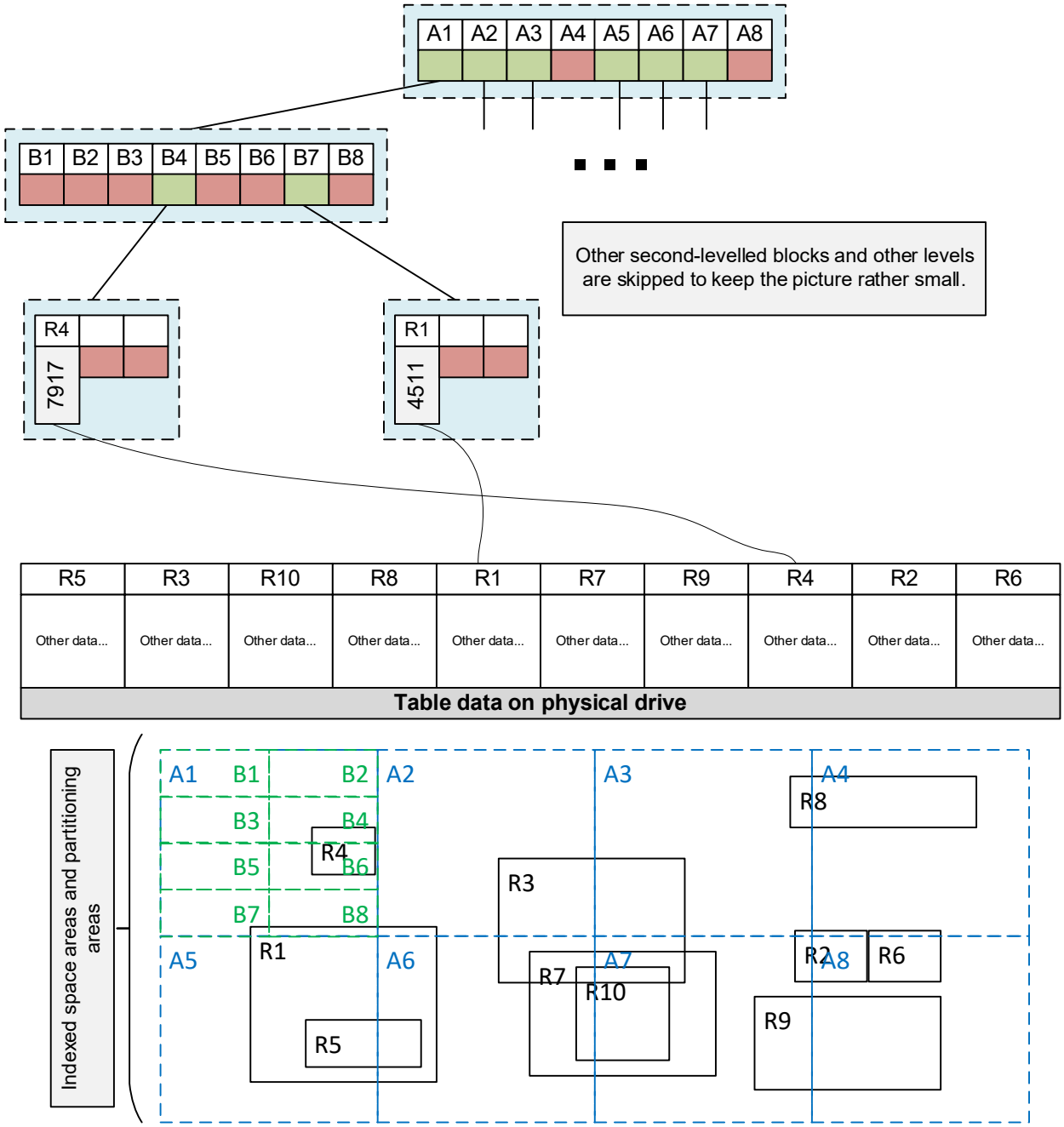


Figure 2.4.r — One of the options for implementing a spatial index (another option based on the R-tree was considered earlier{114})

Currently, spatial indexes are supported by MySQL, MS SQL Server, Oracle and other DBMSes that support spatial and geometric data types.



Full text index[99] — an index optimized to speed up the search operations for substring occurrences in text fields.

Simplified: an index to search for a text in a text.

Earlier considered in this chapter "classical" indexes are aimed at strict comparison of searched and stored values or at checking nesting and intersection of areas (typical for spatial{127} indexes). All these approaches turn out to be useless in solving one specific but very demanded problem: searching a text in a text.

The most frequent case of this task you are definitely familiar with is the search for information on the Internet with the help of search engines. Of course, the algorithms used there and the way they are implemented are different from the one we are going to discuss now, but the concept remains the same: there are many texts from which you need to select those that contain the words or phrases you are looking for.

In the case of implementing your own search engine (for a blog, online store, forum, news site), one possible solution could be the use of full text indexes.

Despite the enormous differences in the implementation of full text indexes in different DBMSes, the general principle of their operation (illustrated in figure 2.4.s) is similar: the analyzed text is divided into words, and then for each word a list of records containing text with this word in the indexed field is stored. Often the position(s) of occurrence of this word in the text is stored in addition, to be able to perform a search taking into account the distance between words.



Despite all the beauty and power of full text indexes, you should understand that they have a wide range of limitations, and their use requires a precise understanding of what you are doing (to understand the scale of complexity it is recommended to read the MS SQL Server documentation, where full text indexes are described in 28 **sections**). In other words: you should not expect miracles, but some queries can be noticeably accelerated.

So, let's move on to a graphical representation of the full text indexes (figure 2.4.s). Suppose we have a table storing the message history in some corporate messenger. To speed up the search for messages, we built a full text index on the corresponding field (and specified "when", "where", "and", "will", "the", "for", "I", "shall", "you", "thanks" as stop words).

Physically, a full text index can be implemented based on B-tree{114}, T-tree{114}, hash-table {114} or other algorithm. For simplicity we assume that in our case hash-table{114} is used.

[99] **Full text index** — index that is created on text-based columns to help speed up queries on data contained within those columns, omitting any words that are defined as stopwords ("InnoDB FULLTEXT Indexes"). [https://dev.mysql.com/doc/refman/8.0/en/innodb-fulltext-index.html]

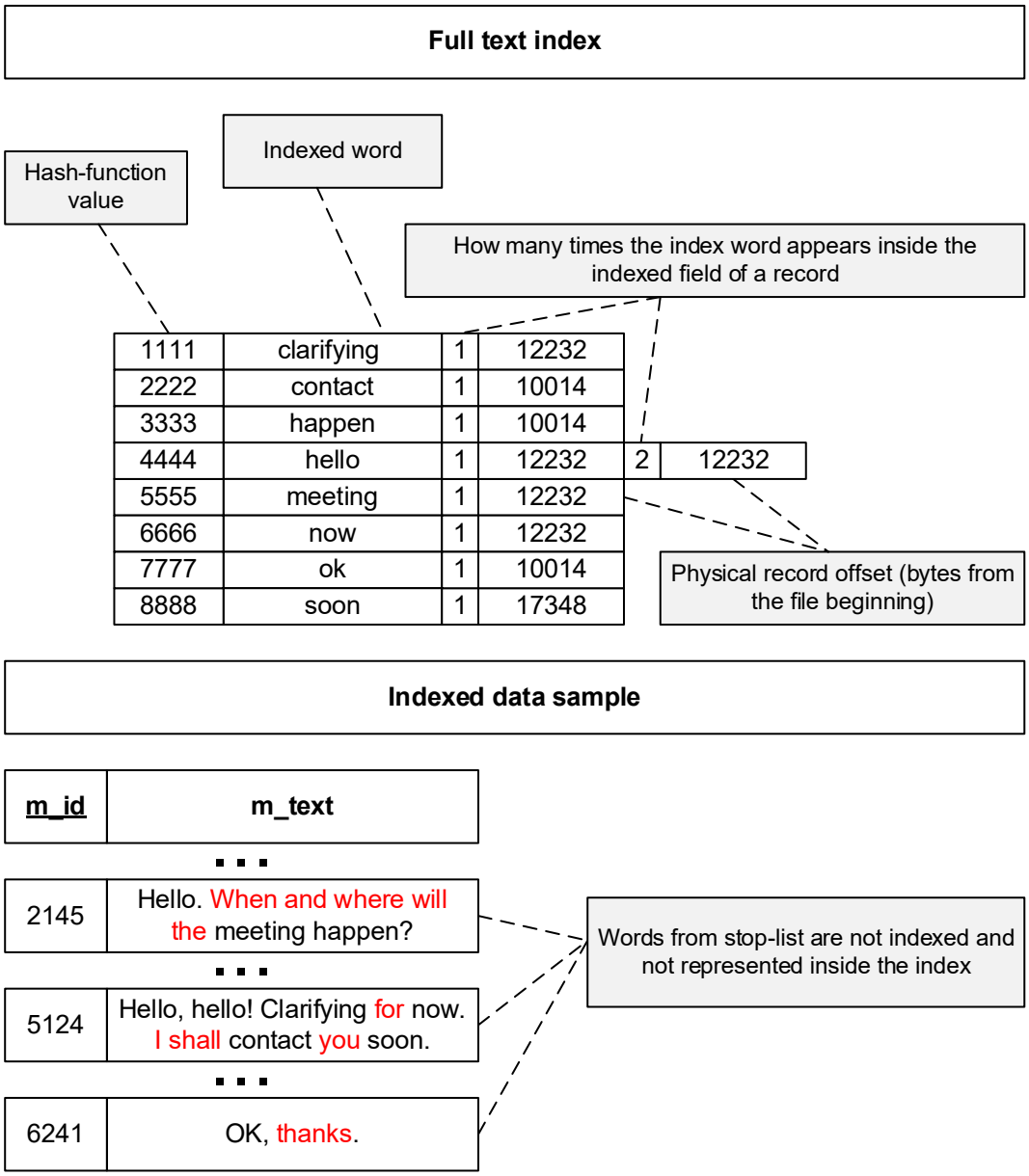


Figure 2.4.s — Schematic view of full text index

Full text indexes are implemented in almost all modern DBMSes, but their implementation mechanisms, capabilities and limitations vary from one DBMS to another, so it remains to repeat the obvious advice: read the documentation thoroughly.



Domain index[100] — an index used to work with specific data in a specific subject area; interaction with such an index is implemented via user subprograms (as opposed to “normal” indexes, whose management is already implemented in the DBMS itself).

Simplified: an index, whose logic is under your own control.

This is a very specific type of indexes, which is used in Oracle. To begin with, it is worth noting that Oracle itself implements spatial and full text indexes using the domain index mechanism. Awareness of this fact allows you to better understand the part of the definition that says “working with domain-specific data” — agree that searching for nested and overlapping geographic areas is quite suitable for such specificity.

Technically, such an index can be implemented based on a bitmap{114}, a hash-table{114} or a B-tree{114}. And its special behavior should be described by you as the authors of the index (and implemented using the `ODCIIndex` interface).

An example code for such an implementation can be found in the documentation[101], but we will limit ourselves to enumerating a few tasks that can be solved with the help of the domain index:

- searching in the text, taking into account the different forms of each of the words that make up the text;
- processing of time intervals, taking into account their duration and overlap;
- indexing serialized data;
- indexing documents in some non-text format (for example, when storing PDF documents inside a database);
- etc.

Traditionally, let’s look at the approximate structure of the index discussed (figure 2.4.t), but stress once again that the structure here is completely classical, and the whole point of the index is how (using your code) the corresponding indexed values are calculated.

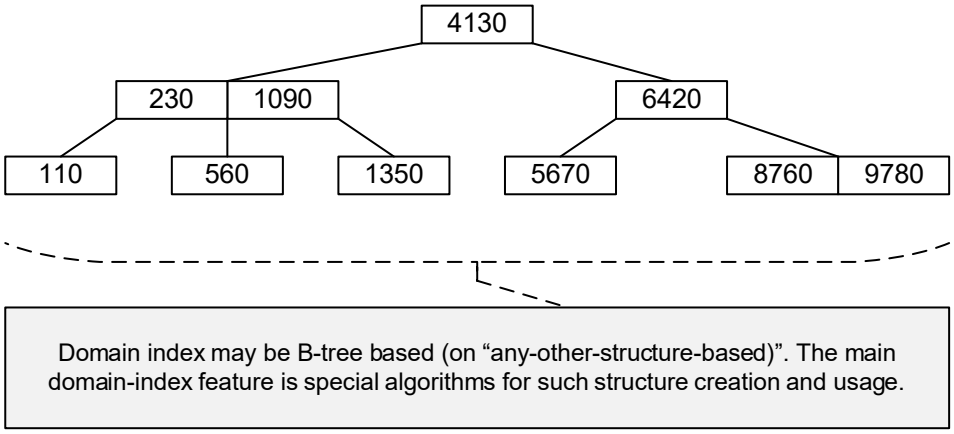


Figure 2.4.t — Schematic view of domain index

[100] **Domain index** — an application-specific index that is used for indexing data in application-specific domains; a domain index is an instance of an index which is created, managed, and accessed by routines supplied by an `indextype`; this is in contrast to B-tree indexes maintained by Oracle internally (“Building Domain Indexes”). [https://docs.oracle.com/cd/B10501_01/appdev.920/a96595/dci07idx.htm]

[101] A code example to implement a domain index: https://docs.oracle.com/cd/B28359_01/server.111/b28286/ap_examples001.htm#SQLRF55430

Currently, only Oracle[102] supports domain indexes in this form, but (considering the DBMS development trend) there is a reason to expect that other DBMSes will support similar solutions in the near future.



You can read a lot of useful information about domain (and other) indexes in Oracle in the “Expert Indexing in Oracle Database 11g: Maximum Performance for Your Database” book (by Bill Padfield, Sam R. Alapati, Darl Kuhn).



XML index[103] — an index optimized to process XML data and speed up search for paths and values within XML documents.

Simplified: an index to speed up XML handling.

Despite the fact that initially it was not intended to store and process hierarchical data structures using relational DBMSes, in recent years there are almost no relational DBMS without XML and/or JSON[104] support. The emergence of these new data types led to the need for their effective indexing. Thus, XML indexes appeared.

Further reasoning here is based on the example of MS SQL Server. In this DBMS XML indexing is based on two types of XML indexes:

- A primary XML index is the result of parsing the stored XML documents (which allows us to avoid such parsing every time we access the table data). Structurally, such an index is nothing but a table (so-called `node table`) and implies storing a large number of records for each document (approximately equal to the number of elements in XML-document).
- Secondary XML indexes are much more similar to “classical indexes” and allow us to speed up a search on specified XPath and XML document elements values.

An approximate structure of how these indexes (and the primary clustered table index) are interlinked is shown in figure 2.4.u.

[102] Similar, but not identical features have recently appeared in PostgreSQL. [https://www.postgresql.org]

[103] **XML index** — index that can be created on xml data type columns (it indexes all tags, values and paths over the XML instances in the column and benefits query performance) (“XML Indexes”). [https://msdn.microsoft.com/en-us/library/ms191497.aspx]

[104] **JSON** — an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value). (“Wikipedia”) [https://en.wikipedia.org/wiki/JSON]

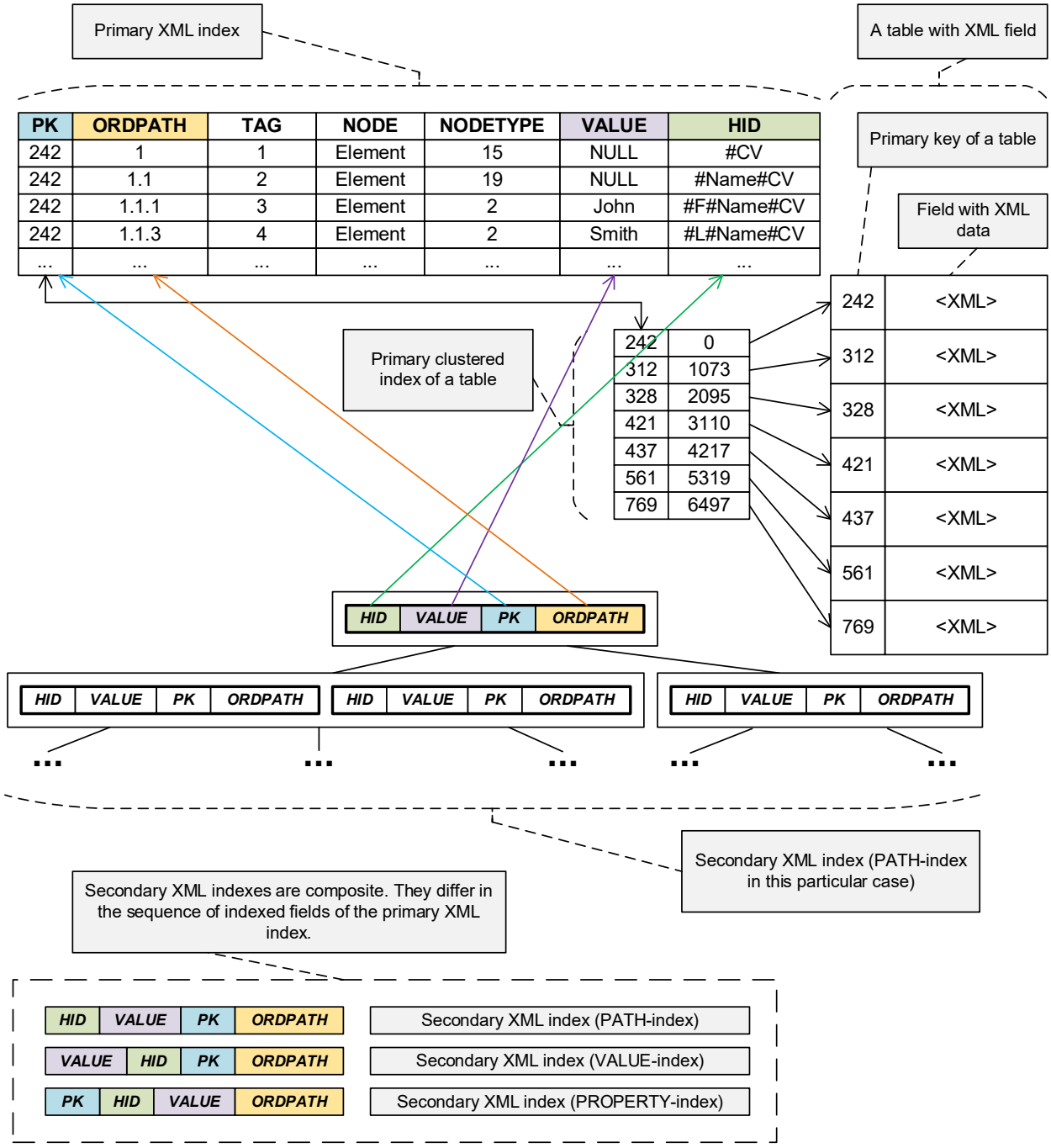


Figure 2.4.u — Schematic view of XML index

XML indices are explicitly implemented in MS SQL Server (figure 2.4.u is created just for this DBMS case), but can be emulated in Oracle through indexes on function values{125}. According to rumors, in the near future it will also be possible to emulate XML indexes in MySQL via computed column{125} indexes (when support for the corresponding technology appears in this DBMS).



You can read a lot of useful information about XML indices in MS SQL Server in the “Pro SQL Server 2008 XML” book (by Michael Coles).



Task 2.4.a: which indexes should be built on the “Bank{395}” database’s relationships? Make appropriate edits to the schema.



Task 2.4.b: should we use composite indexes anywhere in the “Bank{395}” database? If you think “yes”, please edit the schema accordingly.



Task 2.4.c: should we use clustered indexes anywhere in the “Bank{395}” database? If you think “yes”, please edit the schema accordingly.

2.4.2. Creating and Using the Indexes

The following basic operations can be performed with indexes at the database level and while execution SQL queries:

- index creation (deletion);
- existing index enabling (disabling);
- analysis of index usage during query execution;
- hinting the DBMS on index usage.

Let's consider these operations one by one.

Index creation (and deletion)

It should be obvious from the material of the previous chapter that with such a variety of indexes (including those specific to some DBMS) there is no single universal solution for their creation. Sometimes this problem even has to be solved in several stages, pre-creating additional database structures and/or index parts (e.g., it is actual for full text{129} indexes and XML indexes{132}).

So, here we consider an example of creating "just an index", which can speed up information retrieval when performing certain queries.

First, let's create the **books**[105] table with the **b_id** primary key in three DBMSes:

MySQL — Creation of **books** table

```
1    CREATE TABLE `books`
2    (
3     `b_id` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
4     `b_name` VARCHAR(150) NOT NULL,
5     `b_year` SMALLINT UNSIGNED NOT NULL,
6     `b_quantity` SMALLINT UNSIGNED NOT NULL,
7     CONSTRAINT `PK_books` PRIMARY KEY (`b_id`)
8    )
```

MS SQL — Creation of **books** table

```
1    CREATE TABLE [books]
2    (
3     [b_id] [int] IDENTITY(1,1) NOT NULL,
4     [b_name] [nvarchar](150) NOT NULL,
5     [b_year] [smallint] NOT NULL,
6     [b_quantity] [smallint] NOT NULL,
7     CONSTRAINT [PK_books] PRIMARY KEY CLUSTERED ([b_id])
8    )
9    ON [PRIMARY]
```

Oracle — Creation of **books** table

```
1    CREATE TABLE "books"
2    (
3     "b_id"       NUMBER(10) NOT NULL,
4     "b_name"     NVARCHAR2(150) NOT NULL,
5     "b_year"     NUMBER(5) NOT NULL,
6     "b_quantity" NUMBER(5) NOT NULL,
7     PRIMARY KEY ("b_id")
8    )
9    ORGANIZATION INDEX
```

[105] See the "Library" database in "Using MySQL, MS SQL Server and Oracle by Examples" book (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

Already at this stage, you may have noticed that in all cases, the primary key is a clustered index{107} (in MySQL, with InnoDB storage engine{30} the primary key automatically becomes a clustered index, in MS SQL Server we explicitly created the primary key as a clustered index, and in Oracle we achieved the same effect by creating the table as an index-organized one (with the **ORGANIZATION INDEX** option)).

But now we will not talk about the advantages or disadvantages of clustered indexes or index-organized tables (alas, here it is very difficult to give universal recipes); we will move on to "just indexes" themselves.

Let's create three indexes, one of which will be objectively redundant:

- on the {**b_name**, **b_year**} fields (composite index);
- on the **b_quantity** field;
- on the **b_name** field (this index makes no sense because the **b_name** field is the first in the {**b_name**, **b_year**} index, and we considered earlier{47} that the DBMS can search data using the first field in a composite index as fast as using all fields in that index).

MySQL — Creating indexes on the **books** table

```
1    CREATE INDEX `idx_b_name_b_year`
2    ON `books` (`b_name`, `b_year`);
3
4    CREATE INDEX `idx_b_quantity`
5    ON `books` (`b_quantity`);
6
7    CREATE INDEX `idx_b_name`
8    ON `books` (`b_name`);
```

MS SQL — Creating indexes on the **books** table

```
1    CREATE INDEX [idx_b_name_b_year]
2    ON [books] ([b_name], [b_year]);
3
4    CREATE INDEX [idx_b_quantity]
5    ON [books] ([b_quantity]);
6
7    CREATE INDEX [idx_b_name]
8    ON [books] ([b_name]);
```

Oracle — Creating indexes on the **books** table

```
1    CREATE INDEX "idx_b_name_b_year"
2    ON "books" ("b_name", "b_year");
3
4    CREATE INDEX "idx_b_quantity"
5    ON "books" ("b_quantity");
6
7    CREATE INDEX "idx_b_name"
8    ON "books" ("b_name");
```

It is obvious from the SQL code that in case of "just indexes" (until we start managing their types and parameters) in all three DBMSes the syntax of this command is completely identical.

Deleting indexes (in the same simple situation, if we don't deal with primary keys or complex-organized indexes that rely on additional data structures) is just as trivial — just execute the **DROP INDEX** command:

MySQL Deleting the index

```
1    DROP INDEX `idx_b_name` ON `books`;
```

MS SQL Deleting the index

```
1    DROP INDEX [idx_b_name] ON [books];
```

Oracle Deleting the index

```
1    DROP INDEX "idx_b_name";
```

Note: in Oracle, index names are global for the whole schema (which is why we don't need to specify what table the index belongs to), while in MySQL and MS SQL Server index names are local within the table (yes, we can create indexes with the same name on different tables, but we should avoid this, because this approach complicates further work with the database).

Existing index enabling (and disabling)

By default, an index is immediately in the enabled state after creation, but for performance evaluation (or other reasons) it may be necessary to temporarily switch an index to the disabled state.



Disabling and re-enabling an index makes sense exactly from the viewpoint of studying the DBMS behavior (and one can often get by with "hints{151}" on using the index in an SQL query) — one **does not need** to disable the index before large-scale data modification operations: most DBMSes have special mechanisms for performing such operations without disabling the indexes (see below{155} and also read your DBMS documentation).

A disabled index automatically becomes outdated (the information in it does not correspond to the actual table data set) and requires rebuilding after being enabled, i.e., a rather laborious DBMS operation.

So, if we still decide to disable or enable an index, we can do it with the following queries.

As follows from MySQL query syntax, we cannot disable or enable a separate index in this DBMS. Also in case of MySQL `DISABLE KEYS` and `ENABLE KEYS` commands are not applicable to the currently most popular InnoDB storage engine{30}: if we execute these commands on a table that uses this storage engine, we will get a "Table storage engine for 'table_name' doesn't have this option" message. If we use MyISAM storage engine, the commands to disable and enable indexes will work (the primary key and unique indexes will not be disabled anyway).

MySQL Indexes disabling and enabling

```
1    -- Disabling indexes:
2    ALTER TABLE `books` DISABLE KEYS;
3
4    -- Enabling indexes:
5    ALTER TABLE `books` ENABLE KEYS;
```

MS SQL Indexes disabling and enabling

```
-- Disabling index:
ALTER INDEX [idx_b_name] ON [books] DISABLE;

-- Enabling index:
ALTER INDEX [idx_b_name] ON [books] REBUILD;
```

Oracle Indexes disabling and enabling

```
-- Without this command, data modification operations will become
-- unavailable for a table that has a disabled index:
ALTER SESSION SET skip_unusable_indexes = true;

-- Disabling index:
ALTER INDEX "idx_b_name" UNUSABLE;

-- Enabling index:
ALTER INDEX "idx_b_name" REBUILD;
```

Despite the fact that MS SQL Server and Oracle provide enough flexibility to disable and re-enable some indexes, the above remark still applies to these DBMSes: there are more technically efficient ways to get the desired DBMS behavior, so consider disabling indexes as a last resort.

Analysis of index usage during query execution

By creating indexes, we assume that with them the DBMS will be able to execute certain queries more efficiently. But it is always worth checking whether our hopes are justified.

One of the simplest and at the same time the most rational way to check this is to analyze the query execution plan, in which the DBMS will provide us with information about the indexes used (among other things).

The way of obtaining a query execution plan and its contents vary greatly in different DBMSes, so we will consider several options in sequence.

First, we will delete all indexes (except the primary keys) from the previously created `books` table and add ten million records to this table. Then we will check how adding indexes will affect the following queries:

- counting books published in the specified range of years;
- counting books with a specified title;
- counting books with a specified year of publication and a specified word in the title;
- searching for books with a maximum number of copies;
- making a list of ten books with the least number of copies.

In MySQL, we can get a query plan with the `EXPLAIN` command, which simply needs to be added to the beginning of a query. Let's look at examples of queries and query plans provided by MySQL.

MySQL Query execution plan retrieval sample

```
1    -- 1) Counting books published in the specified range of years:
2    EXPLAIN
3    SELECT COUNT(*)
4    FROM   `books`
5    WHERE  `b_year` BETWEEN 1990 AND 1995;
6
7    -- 2) Counting books with a specified title:
8    EXPLAIN
9    SELECT COUNT(*)
10   FROM   `books`
11   WHERE  `b_name` = 'Book 0 1';
12
13   -- 3) Counting books with a specified year of publication and a specified
14   --    word in the title:
15   EXPLAIN
16   SELECT COUNT(*)
17   FROM   `books`
18   WHERE  `b_year` = 1995
19      AND `b_name` LIKE '%ok 0 5%';
20
21   -- 4) Searching for books with a maximum number of copies:
22   EXPLAIN
23   SELECT *
24   FROM   `books`
25   WHERE  `b_quantity` = (SELECT MAX(`b_quantity`)
26                          FROM   `books`);
27
28   -- 5) Making a list of ten books with the least number of copies:
29   EXPLAIN
30   SELECT *
31   FROM   `books`
32   ORDER  BY `b_quantity` ASC
33   LIMIT  10;
```

For queries 1–3 the execution plan is the same and looks like this:

id	select_type	Table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

The execution plan for query 4 is as follows:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where
2	SUBQUERY	books	ALL	NULL	NULL	NULL	NULL	9730420	NULL

The execution plan for query 5 is as follows:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using filesort

Of all the information presented here, we will be interested in the fields **`possible_keys`** (potentially applicable indexes) and **`key`** (the index that was used when executing the query). So far, we see that in all cases these fields have **`NULL`**, which is quite logical, because apart from primary keys there are no other indexes.

Let's get query execution plans in MS SQL Server. This DBMS has the ability to show the query execution plan as a text view[106] and as a graphical view.

Since the graphical representation is more visual, we'll use the text representation for the first of the following queries only.

[106] See documentation for details: "Displaying Execution Plans by Using the Showplan SET Options (Transact-SQL)". [https://technet.microsoft.com/en-us/library/ms180765.aspx]

MS SQL Query execution plan retrieval sample

```
1    -- Enabling query execution plan display:
2    SET SHOWPLAN_TEXT ON
3
4    -- 1) Counting books published in the specified range of years:
5    SELECT COUNT(*)
6    FROM   [books]
7    WHERE  [b_year] BETWEEN 1990 AND 1995;
8
9    -- 2) Counting books with a specified title:
10   SELECT COUNT(*)
11   FROM   [books]
12   WHERE  [b_name] = 'Book 0 1';
13
14   -- 3) Counting books with a specified year of publication and a specified
15   --    word in the title:
16   SELECT COUNT(*)
17   FROM   [books]
18   WHERE  [b_year] = 1995
19      AND [b_name] LIKE '%ok 0 5%';
20
21   -- 4) Searching for books with a maximum number of copies:
22   SELECT *
23   FROM   [books]
24   WHERE  [b_quantity] = (SELECT MAX([b_quantity])
25                          FROM   [books]);
26
27   -- 5) Making a list of ten books with the least number of copies:
28   SELECT TOP 10 *
29   FROM   [books]
30   ORDER  BY [b_quantity] ASC;
```

So, for the first query, the text representation of the execution plan looks like this:

```
Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1006],0)))
 |--Stream Aggregate(DEFINE:([Expr1006]=Count(*)))
  |--Clustered Index Scan(OBJECT:([book_theory].[dbo].[books].[PK_books]),
    WHERE:([book_theory].[dbo].[books].[b_year]>=[@1] AND
           [book_theory].[dbo].[books].[b_year]<=[@2]))
```

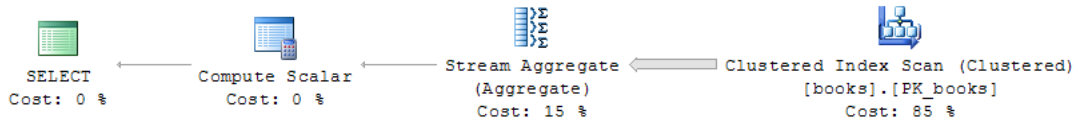
If you execute a query in MS SQL Server Management Studio using the Ctrl+L key combination, you will not only get a nice visual representation of the query execution plan (see figure 2.4.v), but also some additional statistical information on the actual operations performed by the DBMS, as well as tips from the DBMS on what indexes could speed up query execution.

Here is an example of such a hint for the first query:

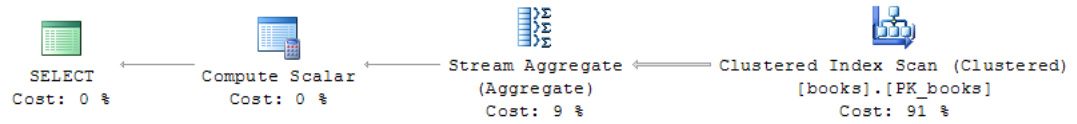
```
The Query Processor estimates that implementing the following index could improve the query cost
by 94.2711%.
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [books] ([b_year])
```

I.e., the DBMS not only determines the necessity of an index and the potential benefit of creating it, but also forms a sample query to create such an index.

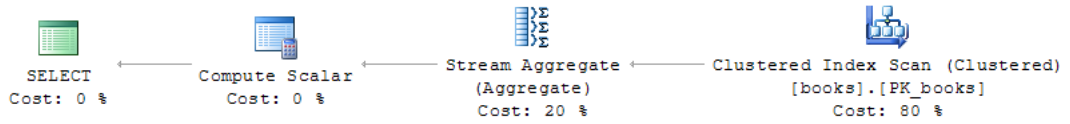
Query 1. Counting books published in the specified range of years:



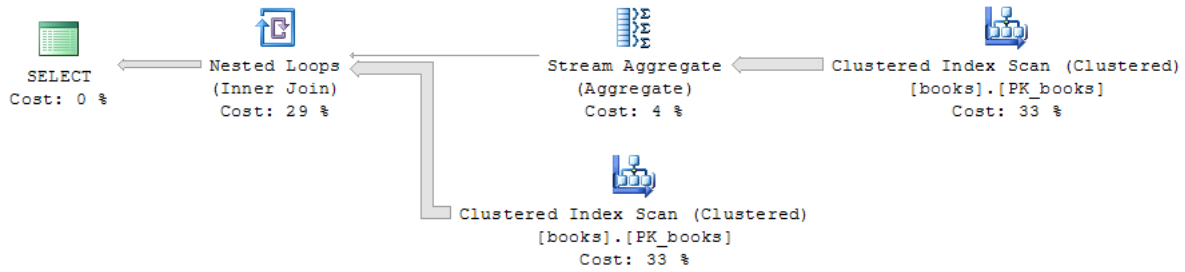
Query 2. Counting books with a specified title:



Query 3. Counting books with a specified year of publication and a specified word in the title:



Query 4. Searching for books with a maximum number of copies:



Query 5. Making a list of ten books with the least number of copies:

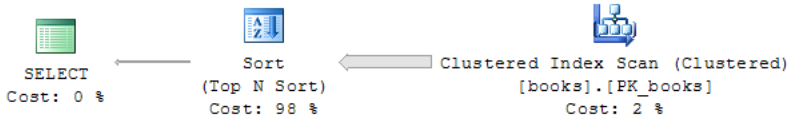


Figure 2.4.v — Visual representation of query execution plans in MS SQL Server Management Studio

So, in all five plans we see only `Clustered Index Scan` (actually — the operation of table data reading) and operations on the resulting data. And this is a quite correct result, which demonstrates that so far, the DBMS has nothing to use to speed up query execution.

To get the query execution plan in Oracle, we must first add the `EXPLAIN PLAN FOR` construct to the beginning of the query, and then use one of several options[107] to view the resulting plan, such as this:

Oracle — One of the options for viewing the resulting query execution plan

```
1    SELECT PLAN_TABLE_OUTPUT
2    FROM TABLE(DBMS_XPLAN.DISPLAY())
```

[107] See the options for viewing the resulting query execution plan here: https://docs.oracle.com/cd/E11882_01/server.112/e41573/ex_plan.htm#PFGRF94681

Let's see the execution plans for the following queries:

Oracle | Query execution plan retrieval sample

```
1    -- 1) Counting books published in the specified range of years:
2    EXPLAIN PLAN FOR
3    SELECT COUNT(*)
4    FROM   "books"
5    WHERE  "b_year" BETWEEN 1990 AND 1995;
6
7    SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
8
9    -- 2) Counting books with a specified title:
10   EXPLAIN PLAN FOR
11   SELECT COUNT(*)
12   FROM   "books"
13   WHERE  "b_name" = 'Book 0 1';
14
15   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
16
17   -- 3) Counting books with a specified year of publication and a specified
18   --    word in the title:
19   EXPLAIN PLAN FOR
20   SELECT COUNT(*)
21   FROM   "books"
22   WHERE  "b_year" = 1995
23      AND "b_name" LIKE '%ok 0 5%';
24
25   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
26
27   -- 4) Searching for books with a maximum number of copies:
28   EXPLAIN PLAN FOR
29   SELECT *
30   FROM   "books"
31   WHERE  "b_quantity" = (SELECT MAX("b_quantity")
32                          FROM   "books");
33
34   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
35
36   -- 5) Making a list of ten books with the least number of copies:
37   EXPLAIN PLAN FOR
38   SELECT *
39   FROM
40    (
41     SELECT "books".*,
42            ROW_NUMBER() OVER (ORDER  BY "b_quantity" ASC) AS "rn"
43     FROM   "books"
44    ) "tmp"
45   WHERE "rn" <= 10;
46
47   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

The resulting execution plans look like this (see figure 2.4.w). Note the fact that these "tables with results" are really just text, albeit broken into separate rows of the `PLAN_TABLE_OUTPUT` column).

Query 1. Counting books published in the specified range of years:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |     4 | 12197   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |     4 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |   492K|  1925K| 12197   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

Query 2. Counting books with a specified title:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |    26 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |    26 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |     2 |    52 | 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_name=U'Book 0 1')
```

Query 3. Counting books with a specified year of publication and a specified word in the title:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |    30 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |    30 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  3499 |   102K| 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year=1995 AND b_name LIKE U'%ok 0 5%')
```

Query 4. Searching for books with a maximum number of copies:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                    | 40590 |  1545K| 24367   (1)| 00:04:53 |
|*  1 |  INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 | 40590 |  1545K| 12193   (1)| 00:02:27 |
|   2 |   SORT AGGREGATE       |                    |     1 |     4 |            |          |
|   3 |    INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  8118K|    30M| 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(b_quantity= (SELECT MAX(b_quantity) FROM books books))
```

Query 5. Making a list of ten books with the least number of copies:

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation               | Name               | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  1 |  VIEW                   |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  2 |   WINDOW SORT PUSHED RANK|                   |  8118K|   301M|   403M| 93828   (1)| 00:18:46 |
|   3 |    INDEX FAST FULL SCAN | SYS_IOT_TOP_101259 |  8118K|   301M|       | 12174   (1)| 00:02:27 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(rn<=10)
2 - filter(ROW_NUMBER() OVER ( ORDER BY b_quantity)<=10)
```

Figure 2.4.w — Query execution plans in Oracle

As with MySQL and MS SQL Server, Oracle has no choice but to scan the entire table (**`INDEX FAST FULL SCAN`** on an index-organized table).

Now it's time to create indexes and check how their presence affects query execution plans. Also, to make it clear, let's compare the time medians (after 100 iterations) of query execution before and after the creation of indexes.

Query	Median execution time before indexes creation, s			Median execution time after indexes creation, s		
	MySQL	MS SQL	Oracle	MySQL	MS SQL	Oracle
Counting books published in the specified range of years	6.228	0.963	3.082	0.074	0.012	0.177
Counting books with a specified title	8.576	2.619	3.000	0.003	0.001	0.005
Counting books with a specified year of publication and a specified word in the title	7.817	1.590	3.096	0.128	0.271	0.213
Searching for books with a maximum number of copies	7.023	2.873	2.871	0.063	0.002	0.002
Making a list of ten books with the least number of copies	10.464	6.450	5.653	0.003	0.001	5.573

To carry out this research, the following indexes were created (the syntax for creating them was discussed earlier{135}):

- on the **`b_name`** field;
- on the {**`b_year`**, **`b_name`**} fields (composite index, and the fields in it are exactly in this order, because there is already a separate index on **`b_name`** field);
- on the **`b_quantity`** field.

Even a quick glance at the results of the experiment shows that the use of indexes justified itself. Only in one case (the 5th query in Oracle) the result "with indexes" remained almost unchanged. It's worth mentioning that for smaller data sets (up to a million records), the results "with indexes" can in some cases be even worse than "without indexes". But we'll come back to this question a little later{150}.

Another interesting fact is that in Oracle's query plans the estimated execution time of a query is much longer than the actual execution time. And this is considered correct behavior of the query optimizer[108].

We now turn to the comparison of query execution plans.

For MySQL, the creation of indexes led to the following change in such plans (see the code of the queries themselves above{139}).

[108] See this discussion at AskTOM: https://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:1434984500346471382

Query 1. Counting books published in the specified range of years:

Before:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

After:

id	select_type	table	type	possible_keys
1	SIMPLE	books	range	idx_b_year_b_name

key	key_len	ref	rows	Extra
idx_b_year_b_name	2	NULL	971120	Using where; Using index

Query 2. Counting books with a specified title:

Before:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

After:

id	select_type	table	type	possible_keys
1	SIMPLE	books	ref	idx_b_name

key	key_len	ref	rows	Extra
idx_b_name	452	const	5	Using where; Using index

Query 3. Counting books with a specified year of publication and a specified word in the title:

Before:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

After:

id	select_type	table	type	possible_keys
1	SIMPLE	books	ref	idx_b_year_b_name

key	key_len	ref	rows	Extra
idx_b_year_b_name	2	const	174654	Using where; Using index

Query 4. Searching for books with a maximum number of copies:

Before:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where
2	SUBQUERY	books	ALL	NULL	NULL	NULL	NULL	9730420	NULL

After:

id	select_type	table	type	possible_keys
1	PRIMARY	books	ref	idx_b_quantity
2	SUBQUERY	NULL	NULL	NULL

key	key_len	ref	rows	Extra
idx_b_quantity	2	const	103390	Using where
NULL	NULL	NULL	NULL	Select tables optimized away

Query 5. Making a list of ten books with the least number of copies:

Before:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using filesort

After:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	index	NULL	idx_b_quantity	2	NULL	10	NULL

Changes in query execution plans allow us to draw the following conclusions:

- All the indexes we created worked (i.e., the DBMS was able to apply them to data search or retrieval).
- The amount of data the DBMS had to process has decreased quite substantially (see the values of the `rows` parameter — even though it is predictive, the changes are still too significant to ignore).
- The previously discussed idea{47} that the DBMS can use the first field separately in a composite index to perform a search operation as efficiently as the whole index was confirmed: in the first query, the composite index `idx_b_year_b_name` on the {`b_year`, `b_name`} fields was used by MySQL to perform a search on the `b_year` field.
- The presence of an index allows the DBMS to optimize even queries with subqueries very effectively. Note the value of the `Extra` parameter in the fourth query: the “Select tables optimized away” value in this case means that the DBMS was able to get all the information needed to execute the subquery without accessing the table data.
- An index can be useful even if it is not built on the field being searched. For example, in the fifth query we see that the DBMS has not found any index suitable for optimizing the search (the `possible_keys` parameter is `NULL`), but it still used the `idx_b_quantity` index, since this field is used for data ordering. The same DBMS behavior (using an index that is not in the “suitable for query execution” list) is typical for so called “covering indexes{120}”.

For MS SQL Server, the creation of indexes led to the following change in query execution plans (see the code of the queries themselves above{140}) — see figure 2.4.x.

Unlike MySQL, the changes here are not so striking at first. Everything seems to be the same for queries 1-3. But if you pay attention to the labels next to the operation icons, you will notice that MS SQL Server switched to Index Seek (search using index) instead of Clustered Index Scan (i.e., looking through all table data), which is a much faster operation and, in many cases, reduces query execution time by tens or hundreds of times.

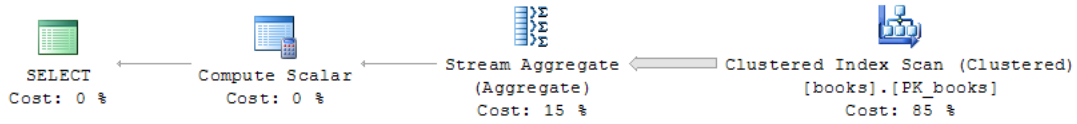
For the fourth query, the changes are even more substantial: instead of two separate table lookups, the DBMS now determines a set of keys for the sought-for records based on an index, then chooses the records themselves based on these keys (which also happens much faster than a full table lookup), and generates the final result (almost instantly, as it follows from the fact that the remaining three operations took 0 % of the total query execution time).

Finally, in the fifth query the execution plan has changed completely: instead of using ordering (an objectively very long operation), the DBMS now follows the same path as in the fourth query: obtains a list of sought-for records based on the index, finds their keys, searches the table for the remaining data using the keys, and then forms the final result.

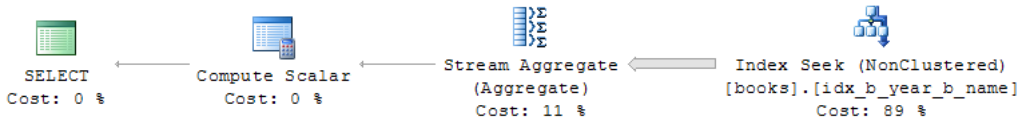
The conclusions for this database are generally similar to those for MySQL: all indexes worked, and in each case the database was able to benefit from their use (even in the fifth query, where using an index might not seem obvious, its presence allowed a very nontrivial restructuring of the query execution plan and led to a tangible performance gain).

Query 1. Counting books published in the specified range of years:

Before:

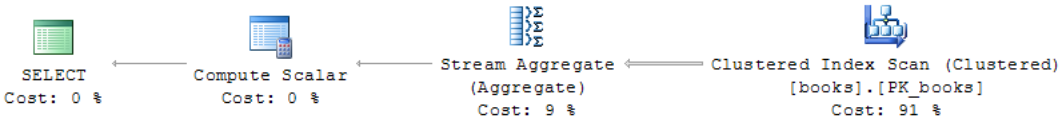


After:

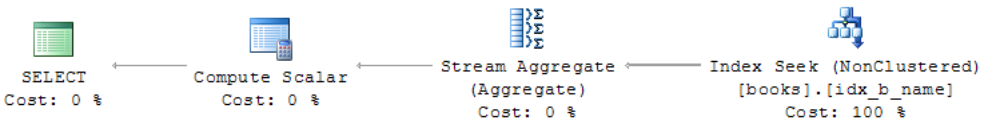


Query 2. Counting books with a specified title:

Before:

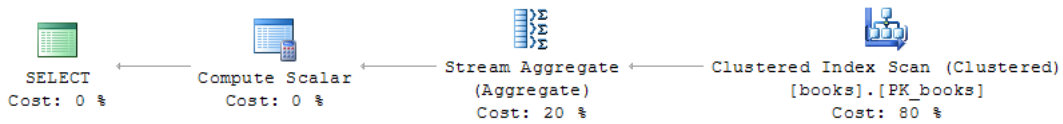


After:

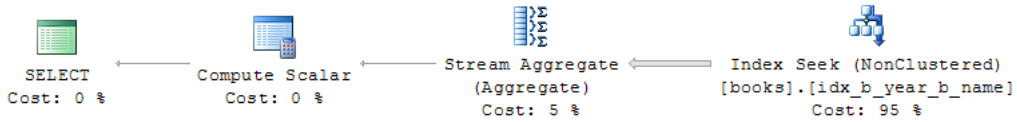


Query 3. Counting books with a specified year of publication and a specified word in the title:

Before:

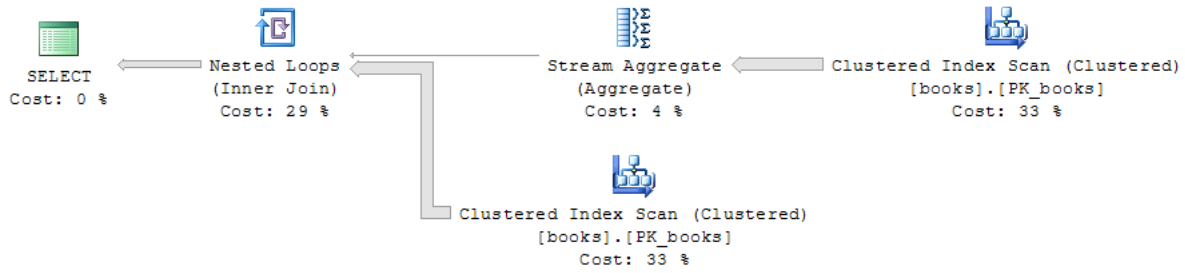


After:



Query 4. Searching for books with a maximum number of copies:

Before:



After:

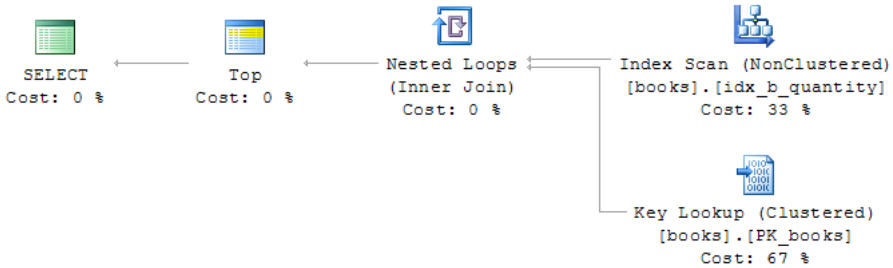
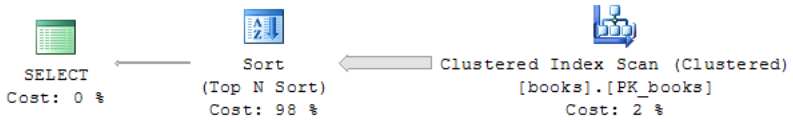


Figure 2.4.x (part 1) — Visual representation of query execution plans in MS SQL Server Management Studio before and after creating indexes

Query 5. Making a list of ten books with the least number of copies:

Before:



After:

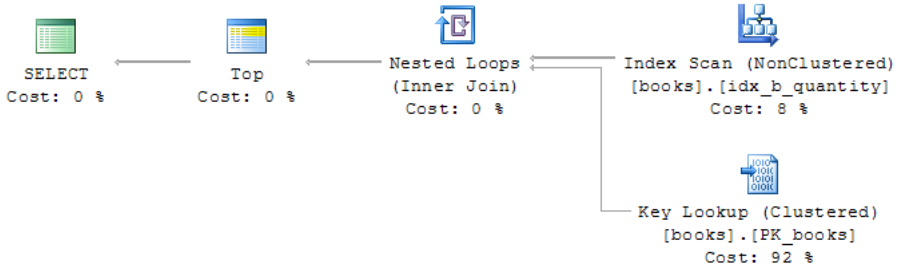


Figure 2.4.x (part 2) — Visual representation of query execution plans in MS SQL Server Management Studio before and after creating indexes



It is time to emphasize two important facts:
1) The logic of using indexes depends very much on DBMS, storage engine{30}, type of index, dataset, query, and other specific factors. Don't expect that what is shown in the examples at hand will always and everywhere work that way.
2) Remember that indexes are not "free" for the DBMS: they consume RAM, reduce performance of data modification operations, and cause other overhead. Create indexes thoughtfully — if you really need them.

Let's move on to a comparison of query execution plans for Oracle. First, take another close look at the results of the query performance experiment{143}: Oracle is the only DBMS in which the median value of the fifth query execution time was almost unchanged.

Let's compare the query execution plans presented in figure 2.4.y before and after creating the indexes (see the code of the queries themselves above{142}) to understand why this is the case.

From figure 2.4.y you can see:

- For queries 1–4 instead of "INDEX FAST FULL SCAN" (which for index-organized tables actually means reading all data without taking into account the index structure; here the word "index" means the index-organized **`books`** table itself) the DBMS applied "INDEX RANGE SCAN" (index analysis with its structure in order to get the record IDs; here the word "index" means **`idx_b_year_b_name`** in the first query and **`idx_b_name`** in the second one). Analyzing an index of one or two fields, moreover, considering its (in our case) tree structure and subsequent selection of necessary records is faster than a full reading of table data.
- And only in the case of the fifth query we see that its plan has not changed. That is, in contrast to MySQL and MS SQL Server, this DBMS could not use any of the available indexes to optimize the execution of such a query. The main reason for this is a fundamental difference of "first N rows" retrieval mechanisms in Oracle.

Query 1. Counting books published in the specified range of years:

Before:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |     4 | 12197   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |     4 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |   492K|  1925K| 12197   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

After:

```
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |     4 |  4072   (1)| 00:00:49 |
|   1 |  SORT AGGREGATE   |                   |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name |   492K|  1925K|  4072   (1)| 00:00:49 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year>=1990 AND b_year<=1995)
```

Query 2. Counting books with a specified title:

Before:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |    26 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |    26 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |     2 |    52 | 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_name=U'Book 0 1')
```

After:

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |            |     1 |    26 |     4   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE   |            |     1 |    26 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_name |     2 |    52 |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_name=U'Book 0 1')
```

Query 3. Counting books with a specified year of publication and a specified word in the title:

Before:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |    30 | 12193   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |    30 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  3499 |   102K| 12193   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year=1995 AND b_name LIKE U'%ok 0 5%')
```

After:

```
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |    30 |   581   (1)| 00:00:07 |
|   1 |  SORT AGGREGATE   |                   |     1 |    30 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name |  3499 |   102K|   581   (1)| 00:00:07 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year=1995)
filter(b_name LIKE U'%ok 0 5%')
```

Figure 2.4.y (part 1) — Query execution plans in Oracle before and after creating indexes

Query 4. Searching for books with a maximum number of copies:

Before:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                     | 40590 |  1545K| 24367   (1)| 00:04:53 |
|*  1 |  INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 | 40590 |  1545K| 12193   (1)| 00:02:27 |
|   2 |   SORT AGGREGATE       |                     |     1 |     4 |            |          |
|   3 |    INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |  8118K|    30M| 12174   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(b_quantity= (SELECT MAX(b_quantity) FROM books books))
```

After:

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                   | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                    | 40590 |  1545K| 12196   (1)| 00:02:27 |
|*  1 |  INDEX FAST FULL SCAN       | SYS_IOT_TOP_101259 | 40590 |  1545K| 12193   (1)| 00:02:27 |
|   2 |   SORT AGGREGATE            |                    |     1 |     4 |            |          |
|   3 |    INDEX FULL SCAN (MIN/MAX)| idx_b_quantity     |     1 |     4 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(b_quantity= (SELECT MAX(b_quantity) FROM books books))
```

Query 5. Making a list of ten books with the least number of copies:

Before:

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                | Name               | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  1 |  VIEW                    |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  2 |   WINDOW SORT PUSHED RANK|                    |  8118K|   301M|   403M| 93828   (1)| 00:18:46 |
|   3 |    INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 |  8118K|   301M|       | 12174   (1)| 00:02:27 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
1 - filter(rn<=10)
2 - filter(ROW_NUMBER() OVER ( ORDER BY b_quantity)<=10)
```

After:

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                | Name               | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  1 |  VIEW                    |                    |  8118K|  1579M|       | 93828   (1)| 00:18:46 |
|*  2 |   WINDOW SORT PUSHED RANK|                    |  8118K|   301M|   403M| 93828   (1)| 00:18:46 |
|   3 |    INDEX FAST FULL SCAN  | SYS_IOT_TOP_101259 |  8118K|   301M|       | 12174   (1)| 00:02:27 |
-----------------------------------------------------------------------------------------------------
1 - filter(rn<=10)
2 - filter(ROW_NUMBER() OVER ( ORDER BY b_quantity)<=10)
```

Figure 2.4.y (part 2) — Query execution plans in Oracle before and after creating indexes

Let's return to the question{144} of the reason why, on small amounts of data, creating indexes can have much less positive (and sometimes even negative) effect on query execution speed.

To recount in a nutshell several hundred pages of documentation and notes from AskTOM[109]: on small amounts of data, both external factors (operating system, third-party applications, hardware, network, etc.) and internal (not reflected in the query plan) features of the DBMS behavior have too much influence on the experiment results.

If this experiment is repeated many times{143} on a small amount of data, we can obtain a short-term increase in the median execution time for almost all the queries. However, the tendency to its decrease is clearly visible as the data in the table becomes larger and larger.

[109] "Ask The Oracle Masters (AskTOM)" [https://asktom.oracle.com]



Hence another very important conclusion: the test data used for experimental evaluation of DBMS performance when executing certain operations under certain conditions should be as close to the real data as possible. Since not only the realistic results depend on the data set, but even the DBMS behavior itself (the algorithms it uses and the sequence of their application), so a solution that is effective in one situation may be detrimental in another.

Hinting the DBMS on index usage



Use index hints as well as query hints only if you know exactly what you are doing (or for research). Most likely, the DBMS will build a better query execution plan without your help.

Earlier{137} we already said that disabling and re-enabling indexes in everyday work is a bad idea, instead of which we can use special solutions (see below {155}) and/or "hints", with which we can influence the DBMS use of indexes.

Traditionally, let's start with MySQL. In this DBMS we have three options for using index hints. We can:

- list the indexes the DBMS must limit itself to when executing the query (`USE INDEX`);
- list the indexes that are not allowed to be used when executing the query (`IGNORE INDEX`);
- specify the indexes that must be used as an alternative to an unacceptably long table lookup (`FORCE INDEX`).

We can also specify for which kind of operation (`JOIN`, `ORDER BY`, `GROUP BY`) we want to use the index[110].

Let's demonstrate the change in the plan for "Counting books published in the specified range of years" query (see the previous section{138}) using the `IGNORE INDEX` hint.

MySQL — Prohibiting the use of index during query execution

```
1    -- Counting books published in the specified range of years
2    -- (without index):
3    EXPLAIN
4    SELECT COUNT(*)
5    FROM   `books`
6    IGNORE INDEX (`idx_b_year_b_name`)
7    WHERE  `b_year` BETWEEN 1990 AND 1995
```

Before we abandoned the index, the query execution plan looked like this:

id	select_type	table	type	possible_keys
1	SIMPLE	books	range	idx_b_year_b_name

key	key_len	ref	rows	Extra
idx_b_year_b_name	2	NULL	971120	Using where; Using index

After abandoning the index, the query execution plan now looks like this (as if there were no indexes available for this query):

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

[110] See details in the documentation: "Index Hints" [https://dev.mysql.com/doc/refman/8.0/en/index-hints.html]

By simply changing the query this way in MySQL, we can quickly check whether the existence of an index still speeds up the query, or whether the situation has changed so much that the index is no longer effective and can be removed.

If we try to force (**`FORCE INDEX`**) the DBMS to use an index which is not suitable for query execution, the situation will be similar.

MySQL — Forced use of an index during query execution

```
-- Counting books published in the specified range of years
-- (forcing the use of an unsuitable index):
EXPLAIN
SELECT COUNT(*)
FROM   `books`
FORCE  INDEX (`idx_b_quantity`)
WHERE  `b_year` BETWEEN 1990 AND 1995
```

Before forcing the use of an unsuitable index, the query execution plan looked like this:

id	select_type	table	type	possible_keys
1	SIMPLE	books	range	idx_b_year_b_name

key	key_len	ref	rows	Extra
idx_b_year_b_name	2	NULL	971120	Using where; Using index

After forcing the use of an unsuitable index, the query execution plan looks like this:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	books	ALL	NULL	NULL	NULL	NULL	9730420	Using where

In this case, despite all our efforts, the DBMS still did not give in to the provocation and used the only available query execution method — searching the whole table. But in more complex situations, using this hint may lead to an even more suboptimal plan.

This situation just confirms the warning given at the beginning of this section: we may damage the DBMS operation by using incorrect hints, which may be even good in research tasks, but is unacceptable in real-world applications.

In MS SQL Server index hints are not a separate part of the syntax and are blurred in the general structure of query hints[111].

As an example, let's implement a behavior similar to MySQL using this DBMS.

Let's forbid MS SQL Server to use an index when running the "Counting books published in the specified range of years" query, (see the previous section{138}).

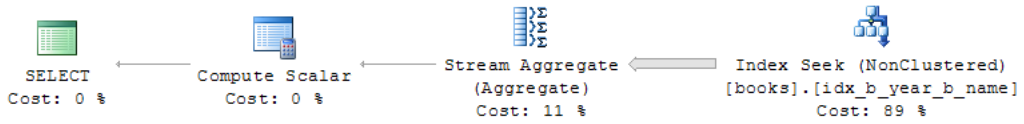
[111] See details in the documentation: "Hints (Transact-SQL)" [https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql]

MS SQL — Prohibiting the use of index during query execution

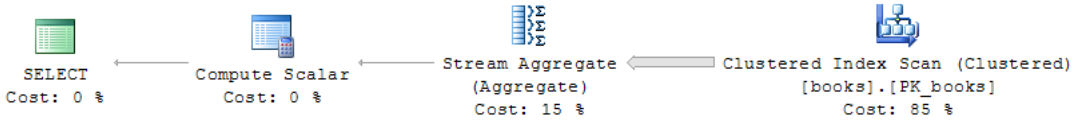
```
1    -- Counting books published in the specified range of years
2    -- (without index):
3    SELECT COUNT(*)
4    FROM   [books]
5    WITH   (INDEX(0))
6    WHERE  [b_year] BETWEEN 1990 AND 1995
```

As with MySQL, the situations before and after the abandonment of the index are an exact copy of the “index present” and “no index present” situations.

Query execution plan before abandoning the index:



Query execution plan after abandoning the index:

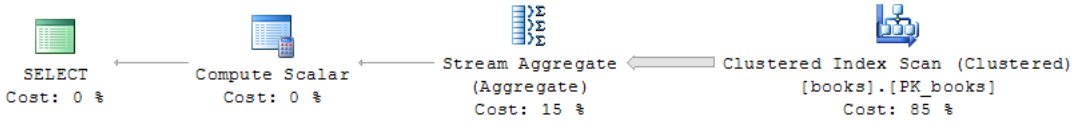


Let’s try to force the DBMS to use an index that is useless for this query.

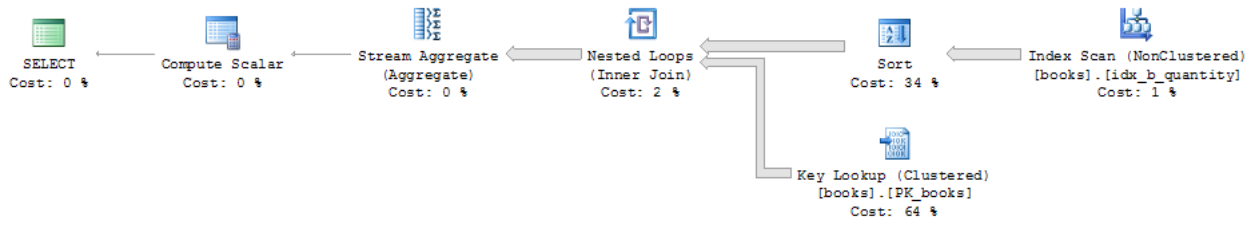
MS SQL — Forced use of an index during query execution

```
1    -- Counting books published in the specified range of years
2    -- (forcing the use of an unsuitable index):
3    SELECT COUNT(*)
4    FROM   [books]
5    WITH   (INDEX([idx_b_quantity]))
6    WHERE  [b_year] BETWEEN 1990 AND 1995
```

Before forcing the use of an unsuitable index, the query execution plan looked like this:



After forcing the use of an unsuitable index, the query execution plan looked like this:



Here (unlike MySQL) we just got a situation of noticeable deterioration of the query execution plan, when MS SQL Server tried to execute our hint, but ended up just making some meaningless unnecessary operations.

In Oracle the situation with hints for using indexes is completely similar to that of MS SQL Server in the sense that these hints are part of the general syntax of hints to the query execution optimizer[112].

As an example, let's implement behavior similar to MySQL and MS SQL Server using this DBMS.

Let's forbid Oracle to use an index when executing the "Counting books published in the specified range of years" query (see the previous section{138}).

Oracle — Prohibiting the use of index during query execution

```
1    -- Counting books published in the specified range of years
2    -- (without index):
3    EXPLAIN PLAN FOR
4    SELECT /*+ NO_INDEX("books" "idx_b_year_b_name") */
5           COUNT(*)
6    FROM   "books"
7    WHERE  "b_year" BETWEEN 1990 AND 1995;
8
9    SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

And this is the third time we've seen a pattern in which the execution plans for queries "with index" and "without index" have swapped.

Before we abandoned the index, the query execution plan looked like this:

```
-------------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |     4 |  4072   (1)| 00:00:49 |
|   1 |  SORT AGGREGATE   |                   |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name |   492K|  1925K|  4072   (1)| 00:00:49 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year>=1990 AND b_year<=1995)
```

After the query was abandoned, the query execution plan now looks like this (as if there were no indexes available for this query):

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                    |     1 |     4 | 12197   (1)| 00:02:27 |
|   1 |  SORT AGGREGATE       |                    |     1 |     4 |            |          |
|*  2 |   INDEX FAST FULL SCAN| SYS_IOT_TOP_101259 |   492K|  1925K| 12197   (1)| 00:02:27 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

Now let's try to force the DBMS to use an index that is useless for the current query.

Oracle — Forced use of an index during query execution

```
1    -- Counting books published in the specified range of years
2    -- (forcing the use of an unsuitable index):
3    EXPLAIN PLAN FOR
4    SELECT /*+ INDEX("books" "idx_b_quantity") */
5           COUNT(*)
6    FROM   "books"
7    WHERE  "b_year" BETWEEN 1990 AND 1995;
8
9    SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

[112] See details in the documentation: "Using Optimizer Hints": https://docs.oracle.com/cd/E11882_01/server.112/e41573/hintsref.htm.

Before forcing the use of an unsuitable index, the query execution plan looked like this:

```
-----------------------------------------------------------------------------------------
| Id  | Operation         | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |                   |     1 |     4 |  4072   (1)| 00:00:49 |
|   1 |  SORT AGGREGATE   |                   |     1 |     4 |            |          |
|*  2 |   INDEX RANGE SCAN| idx_b_year_b_name |   492K|  1925K|  4072   (1)| 00:00:49 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - access(b_year>=1990 AND b_year<=1995)
```

After forcing the use of an unsuitable index, the query execution plan looked like this:

```
-----------------------------------------------------------------------------------------
| Id  | Operation          | Name               | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                    |     1 |     4 |  6748K  (1)| 22:29:43 |
|   1 |  SORT AGGREGATE    |                    |     1 |     4 |            |          |
|*  2 |   INDEX UNIQUE SCAN| SYS_IOT_TOP_101259 |   492K|  1925K|  6748K  (1)| 22:29:43 |
|   3 |    INDEX FULL SCAN | idx_b_quantity     |  8118K|       | 29911   (1)| 00:05:59 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
2 - filter(b_year>=1990 AND b_year<=1995)
```

As in the case of MS SQL Server, here again we have a situation of noticeable deterioration of the query execution plan, when the DBMS performed several meaningless unnecessary operations. Pay attention to the predicted execution time of the top-level operations: it was 49 seconds, now it is 22 and a half hours. Worth thinking about.

After looking at these examples, you may think that using DBMS hints for indexes can only do harm (in general, this is a correct opinion and will prevent you from a lot of problems).

In fact, in really complex situations, if you have enough experience and a deep understanding of the database structure, the peculiarities of a particular query, and many other facts, using DBMS hints for indexes can significantly improve the query execution plan. But such complex and yet realistic tasks are far beyond the scope of this book.

And at the end of this section, let's explain how to deal with the problem of data modification performance degradation in the presence of a large number of indexes. Most often this question arises in the context of an insert operation, although many solutions will be useful for updating and deleting data as well.

For each of the following tips, a caution is relevant: in the context of the task you are dealing with, this or that approach may be inapplicable, technically impossible, or incompatible with the DBMS, storage engine{30}, database schema, etc.

Nevertheless.

- Make sure there are no redundant indexes on your table (remember{47} that the DBMS can search the first field of a composite index as fast as the whole composite index, so there is no need to create a separate index on this field).
- Perform several modification operations within a single transaction{364}. In most cases the DBMS will not update the indexes until the transaction is complete.
- Make sure that you have optimally chosen the clustered index{107} of the table being modified and perform the modification operations so that the values of the corresponding field in the data being modified are in the same sequence as the values of the same field in the clustered index.
- Consider using temporary tables as an intermediate buffer for storing processed information, as much as possible transfer to the side of the DBMS any data modification related operations.

- If the modified table contains foreign keys{43}, make sure that the corresponding fields (or sets of fields) of the parent table(s) are indexed (although this condition is almost always met automatically.
- If your DBMS allows it, temporarily disable foreign keys{43} and unique indexes{106}. This is a **very dangerous** operation, so think twice before doing it.
- Study the query execution plan and the relevant recommendations for your DBMS. There is usually a lot of interesting stuff in there[113, 114, 115].

This concludes the section on the basic structures of relational databases. In the next section, based on this knowledge, we will consider ways to build database schemas that are resistant to a number of unobvious errors when performing data reading and modification operations.



Task 2.4.d: which indexes in the “Bank{395}” database should be disabled when importing a large amount of data? How much would this improve performance? Perform a corresponding experiment.



Task 2.4.e: examine several execution plans for typical queries to the “Bank{395}” database. Is it possible to speed up the execution of these queries by “hinting the DBMS on index usage{151}”? Perform a corresponding experiment.



Task 2.4.f: is it possible to use alternative code to create indexes when creating the “Bank{395}” database? If you think “yes”, write an appropriate solution and check if it works.

[113] For MySQL: https://dev.mysql.com/doc/refman/8.0/en/optimizing-innodb-bulk-data-loading.html
[114] For MS SQL Server: https://technet.microsoft.com/en-us/library/ms190421(v=sql.105).aspx
[115] For Oracle: http://www.dba-oracle.com/t_insert_tuning.htm

Chapter 3: Normalization and Normal Forms

3.1. Data Operation Anomalies

3.1.1. Theoretical Review of Data Operations Anomalies

Historically, the situation with the formulation of a strict universal definition of data operation anomalies has not been the best: various authors clearly describe certain types of anomalies, leaving for a general definition only streamlined phrases such as “some difficulties[116]”, “additional problems[117]”, etc. Therefore, let’s generalize the description of individual anomalies (to be discussed below) and formulate a general definition.



Data operation anomalies — an incorrect execution of data operations or the occurrence of side effects of data operations, resulting from a violation of the database adequacy to the subject area{11}.

Simplified: while performing a correct (from the user’s point of view) operation with a correct SQL query we get a wrong result or an undesirable side effect.

The following three types of data anomalies are traditionally distinguished.



Insertion anomaly[118] — a violation of the data insertion logic, expressed in the need to use fictitious values or `NULL`-values for part of the attributes.

Simplified: we don’t know the values of some fields in an entry and have to either make them up or replace them with `NULL`-values.



Deletion anomaly[119] — a violation of the data deletion logic, expressed in the loss of information not intended for deletion, the last copy of which was stored in the attributes of the record being deleted.

Simplified: we wanted to delete some data, but along with them we lost others, which we did not intend to delete.



Modification anomaly[120] — a violation of data update logic, expressed in the need to update the values of some attribute(s) in several record(s) in addition to the one with which the update operation is currently performed, in order to maintain database consistency.

Simplified: by updating one record, we have to update several others, so as not to compromise the integrity of the database.

Also note that even though there is no such term in the scientific literature, we can also talk about selection anomaly, because it very often accompanies the modification anomaly.

---

116 “… variety of… difficulties…” (C.J. Date, “An Introduction to Database Systems”, 8th edition, p. 359).

117 “… additional problem… “ (Ramez Elmasri, Shamkant Navathe, “Fundamentals of Database Systems”, 6th edition, p. 507).

118 **Insertion anomalies** can be differentiated into two types: a) To insert a new tuple into table, we must include either the attribute values for all columns or NULLs (if there is no data for some columns); b) To insert a new tuple into table, we must include NULL value into PK/FK fields, and that violates the entity integrity. (Ramez Elmasri, Shamkant Navathe, “Fundamentals of Database Systems”, 6th edition, [with some text rephrasing])

119 **Deletion anomalies** are related to the next situation: if we delete from a tuple that happens to represent the last copy for a particular attribute value, the information concerning that value is lost from the database. (Ramez Elmasri, Shamkant Navathe, “Fundamentals of Database Systems”, 6th edition, [with some text rephrasing])

120 **Modification anomalies** are related to the next situation: if we change the value of one of the attributes of a particular tuple, we must update the tuples of all corresponding tuples; otherwise, the database will become inconsistent. (Ramez Elmasri, Shamkant Navathe, “Fundamentals of Database Systems”, 6th edition, [with some text rephrasing])



Selection anomaly — a violation of the logic of data selection, which manifests itself in the distortion of the results of the query.

Simplified: we select some data, but either we do not get the full set of data, or we get the wrong data at all.



It is important to understand that data operation anomalies have nothing to do with DBMS failures or SQL query execution errors. When an anomaly occurs, the DBMS works perfectly correctly, and the executed SQL query is also correct and meets the intended purpose.

This is where the danger of anomalies lies: everything works correctly, but the results are wrong.

Let's demonstrate each of the anomalies considered with an example, for which we use the `work` relation shown in figure 3.1.a.

work

PK			
w_employee	**w_project**	**w_role**	**w_salary**
Smith	"Oak" Project	Manager	500
Taylor	"Ash" Project	Consultant	150
Jones	"Maple" Project	Manager	900
Jones	"Oak" Project	Developer	300

Figure 3.1.a — The `work` relation used to demonstrate data operation anomalies

The **insertion anomaly** when working with such a relation can manifest itself in the following forms.

When hiring a new employee (see figure 3.1.b, case 1), we must immediately specify which project, in which role, and at what salary that employee works. If we do not have at least some of this information, we have to make it up or replace it with `NULL`. But since the `w_project` field is part of the primary key, at least in this field, the insertion of `NULL`-values is not allowed.

When a new project appears (see figure 3.1.b, case 2), we must immediately assign at least one employee to it (and also specify the salary).

When a new role appears (see figure 3.1.b, case 3), we must immediately assign a project and an employee with a salary to it.

Thus, we are constantly forced to use some information when inserting, which objectively we may not have at the time of performing the operation. In rare cases such information may be contained in records already existing in the table, but in such a situation it is necessary to read this information beforehand, using an additional (and usually nontrivial and unreliable) algorithm.

work

PK			
w_employee	**w_project**	**w_role**	**w_salary**
Smith	“Oak” Project	Manager	500
Taylor	“Ash” Project	Consultant	150
Jones	“Maple” Project	Manager	900
Jones	“Oak” Project	Developer	300
Jackson	?	?	?
?	“Linden” Project	?	?
?	?	Supervisor	?

1
2
3

Figure 3.1.b — Insertion anomalies example

The **deletion anomaly** when working with this relation can manifest itself in the following forms.

If “Taylor” employee quits (see figure 3.1.c, case 1), and the corresponding record is deleted, the information that there was the “Ash” project and there was a “Consultant” role in the firm will be lost (this information does not appear anywhere but in this record).

work

PK			
w_employee	**w_project**	**w_role**	**w_salary**
Smith	“Oak” Project	Manager	500
Taylor	“Ash” Project	Consultant	150
Jones	“Maple” Project	Manager	900
Jones	“Oak” Project	Developer	300

We lose
We delete
We lose
We delete
We lose
We delete
1
3
2

Figure 3.1.c — Deletion anomalies example

If the firm abolishes the “Manager” role (see figure 3.1.c, case 2) and deletes the corresponding records, information about “Smith” employee will be lost (and if he was the only one working on the “Oak” project, information about this project would also be lost).

If the “Ash” project is closed (see figure 3.1.c, case 3) and the corresponding records are deleted, the information about “Taylor” employee and the “Consultant” role will be lost.

Thus, when we delete one part of the data, we suddenly lose another part of the data that we did not intend to delete.

The **modification anomaly** when working with such a relation can manifest itself in the following forms.

If we decide to rename the “Manager” role to “Coordinator” role (see figure 3.1.d, case 1), this information will have to be updated in several records.

If “Jones” employee decides to change the surname to “Joness” (see figure 3.1.d, case 2), we will have to somehow figure out which records are about the same “Jones” (which is not at all obvious in this database structure) and make edits to these records. It is important to be sure to distinguish this particular “Jones” from his full namesakes, so as not to accidentally rename them as well.

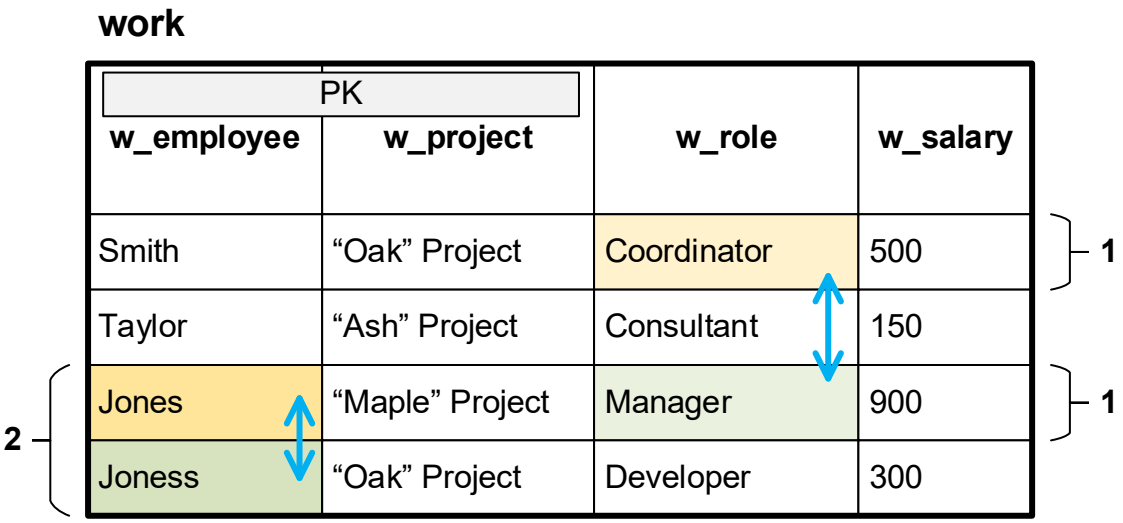


Figure 3.1.d — Modification anomalies example

Thus, by updating data in one record, we are forced to update the same data in other records, at the risk of accidentally updating something that should not be updated or missing something that should be updated.

The **selection anomaly**, as a rule, is a consequence of the modification anomaly.

Let’s imagine that during the process just considered changing the employee’s surname from “Jones” to “Joness” we did not update all the necessary records — this is the situation shown in figure 3.1.d (case 2). Now, for example, by getting the list of projects and the total salary of this employee (“Jones”), we will get a result that will be missing information about the “Oak” project, and the total salary will be 300 less than it is in reality.



There is another unfortunate fact associated with modification and selection anomalies: their presence indicates that (most likely) in the software that uses such a database, the values of interest (projects, employees, etc.) are not selected from a list, but are entered manually every time. This approach drastically increases the probability of typos, i.e., the entered value will not match the already existing ones, and therefore will not get into the selection in the future and will not be updated or deleted during the corresponding operations.

To summarize, if we try to use the `work` relation “as is” in real life, nothing will work correctly: neither insertion, nor deletion, nor modification, nor even data selection.

Obviously, it's necessary to make some changes in the schema, and that's what we'll talk about next when discussing normalization.

For now, we will briefly look at how to understand in real life that the existing schema is subject to one or another data operations anomaly.



Task 3.1.a: what data insertion anomalies are possible in the "Bank{395}" database? Refine the schema so as to eliminate the possibility of their occurrence.



Task 3.1.b: what data modification anomalies are possible in the "Bank{395}" database? Refine the schema so as to eliminate the possibility of their occurrence.



Task 3.1.c: what data deletion anomalies are possible in the "Bank{395}" database? Refine the schema so as to eliminate the possibility of their occurrence.

3.1.2. Ways to Detect Data Operation Anomalies

The clearest sign that data operations anomalies will occur when working with a particular database schema is the violation of normal forms. But in this way, we risk getting a vicious circle (because one of the signs of violation of normal forms is the presence of data operations anomalies). Therefore, in this chapter, we will consider other, not affecting normalization, ways of detecting anomalies and signs of their presence.

Signs of the presence of data operation anomalies are best detected in the process of database design.

In top-down design{10} it is worth constantly checking whether a situation has arisen in which information about several independent real-world entities ends up in the same relation. This task is a bit complicated by the fact that some real-world entities may be properties of other entities. This description may sound confusing, so let's explain it with examples.

First, let's consider the case where the conclusion is obvious. Suppose we create a database of the human resources department, describe the **`employee`** relation and see the following picture.

```
Employee:
   • First name
   • Middle name
   • Last name
   • Child name
```

Here, quite objectively, information about the child (the question immediately arises: why "child", but not "children") is misplaced in the attributes of the **`employee`** entity. Such a schema would suffer from at least deletion{157} and modification{157} anomalies.

If we revise this schema as follows, this problem will no longer exist (in the process of further design of the database a "many to many{56}" relationship will be established between the **`employee`** relation and the **`child`** relation).

```
Employee:
   • First name
   • Middle name
   • Last name

Child:
   • Child name
```

Now consider the case where the conclusion is not so obvious. Let's continue discussing the **`employee`** relation, which now has a **`phone number`** attribute.

```
Employee:
   • First name
   • Middle name
   • Last name
   • Phone number
```

Is the telephone number an independent entity of the real world, or can it be considered an "employee property"? The answer to this question depends solely on the requirements of the subject area: if only one (no more than one) phone number is assigned to each employee by the rules — everything is fine, no anomalies; if several phone numbers can be assigned to each employee and/or one number can be used by several employees, it is worth creating a separate relation as follows (and establishing a "one to many{53}" or "many to many{56}" relationship between them during further database design).

```
Employee:
   • First name
   • Middle name
   • Last name

Phone:
   • Phone number
```

It's easy to notice that a good clue here is the number of occurrences: if "something" can occur 0 or 1 times — this "something" can be considered a property (e.g., date of birth, driving license number, work experience), but if "something" can occur more than 1 times — this "something" should be moved to a separate relation (e.g., list of children, list of cars, list of professional skills).



Be sure to study the requirements and limitations of the subject area! Don't rely on intuitive notions. E.g., it is "intuitive" that a person can only have one date of birth, one passport, etc., but now imagine that we are designing a database of wanted criminals, each of whom may be hiding under different names, have several fake passports, etc.

And there is another important exception to the rule prescribing not to put information about several independent real-world entities in the same relation: the "star[121]" and the "snowflake[122]" schemas. A detailed description of such schemas belongs to the domain of data warehouses and is beyond the scope of this book, but we will consider a small example.

First, let's start with a graphic explanation of the origin of the schemas' names (see figure 3.1.e).

The central table (the so-called "fact table") is linked to the surrounding tables containing descriptive information about the subject area. In the "snowflake" schema, due to additional normalization, such tables may in turn refer to other tables, those to other tables, and so on.

The "fact table" will collect a large number of foreign keys{43} from the surrounding tables, and this situation may seem suspicious, but in practice there is not only no problem here, but even the opposite — the "fact table" can be very well normalized.

[121] See "The Star Schema" article (by Emil Drkušić) for details [http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-the-star-schema]

[122] See "The Snowflake Schema" article (by Emil Drkušić) for details [http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-the-snowflake-schema]

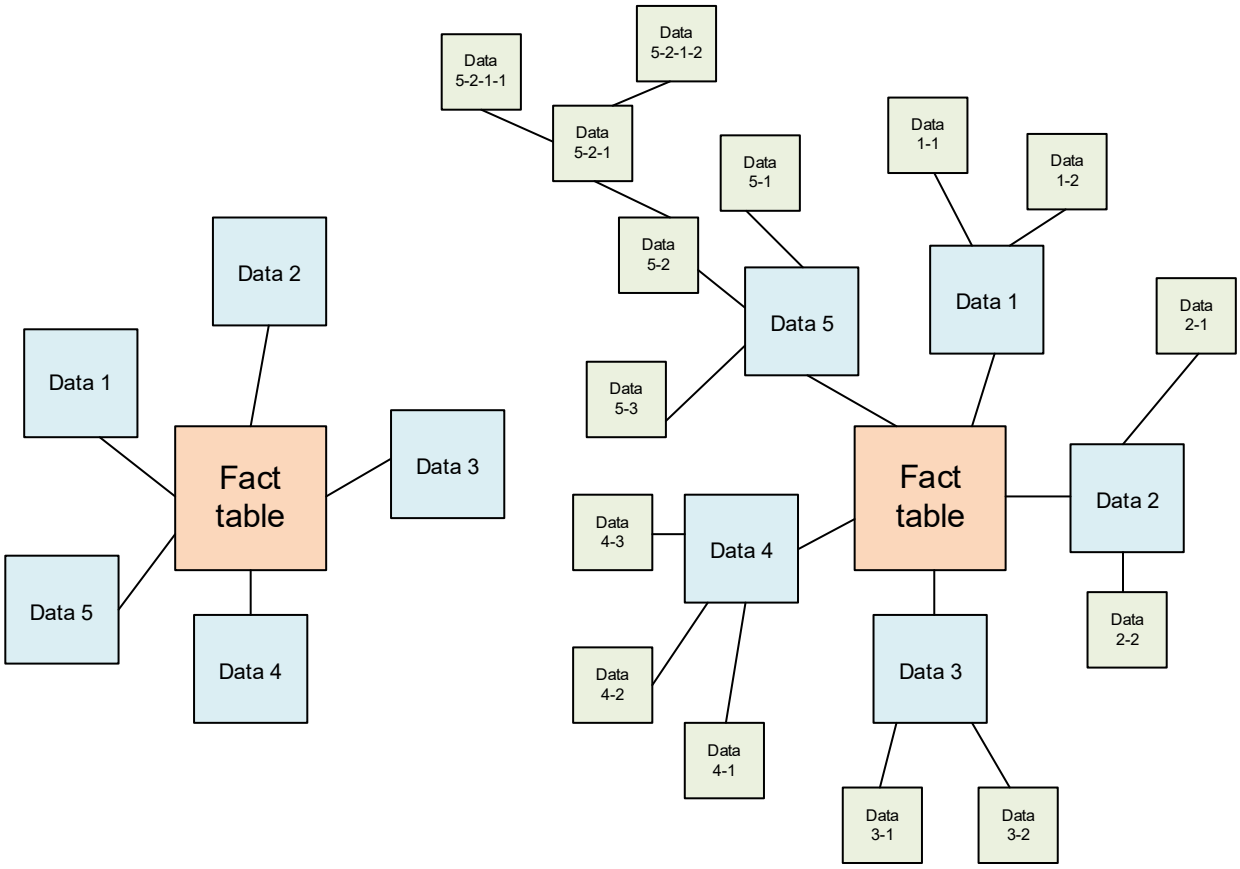


Figure 3.1.e — “Star” (left) and “snowflake” (right) schemas

For the refinement of the understanding of the star schema, consider another (classic) example (see figure 3.1.f).

The data tables here are `student`, `tutor`, `subject`, `classes_type`, and the fact table is `education_process`, which collects information about the “facts of classes”.

Although there may be many students, tutors, subjects, and types of classes, each “fact” (i.e., one entry in the `education_process` table) reflects only the information that such-and-such tutor taught such-and-such classes in such a subject to such-and-such student and gave such-and-such a grade in the end. Therefore, data modification anomalies are not typical for this table.

By the way, this schema is the solution to the previously discussed{11} problem of violation of such a property of the database as the adequacy to the subject area{11}.



Figure 3.1.f — “Star” schema example

Now let’s move on to the bottom-up design{10} in which we get an excellent opportunity to think through the reaction of the database to the execution of a particular SQL-query.

Two basic techniques can be used here.

The essence of the first of them is to consistently check each query for the generation of the corresponding anomaly{157}. I.e., we should consciously look for such variants of queries and input data, in which an anomaly is possible. If the search was successful — great, we found a flaw in our database and can rework the schema.

If we do not yet have a ready-made set of queries (and even if we do), a great helper and generator of ideas here is the representative of the customer for whom the application using our database is being developed. They know like no one else what operations will have to be performed in the course of their daily work, which is especially valuable if the subject area is exotic, and the project team does not have people who are well versed in it.

The essence of the second technique can be expressed in one word: testing. In the broadest sense of the word. Exactly in the process of applying different testing techniques at different levels of database design, many flaws are revealed, among which will be data operations anomalies.

About the testing itself you can read in the corresponding book[123], the database testing will be discussed later{376}, but now we will return to the first technique and consider an example of a query investigation for anomalies.

Suppose we have the `activity` relation shown in figure 3.1.g. Its attributes have the following meaning:

- `a_id` — record identifier (primary key), has no “physical meaning” and is used for internal purposes for guaranteed record identification;
- `a_employee` — employee identifier (foreign key);
- `a_project` — project name;
- `a_role` — identifier of an employee role on a project (foreign key);
- `a_start_date` — the date the employee started working on a project;
- `a_end_date` — the date the employee completes work on a project.

activity

PK a_id	FK a_employee	a_project	FK a_role	Not NULL a_start_date	a_end_date
115	234	“Oak” Project	11	2017.01.09	2017.11.01
116	12	“Ash” Project	3	2012.12.21	NULL
119	3134	“Maple” Project	6	2015.03.01	2016.04.01
121	12	“Oak” Project	5	2015.03.15	2016.05.20
...					

Figure 3.1.g — `activity` relation for the exploration

There are a lot of questions about this relation, but the key ones can be formulated by thinking about typical queries.

Suppose we are going to add or change an entry:

- How can we guarantee the insertion of a correct (existing) project name, if the `a_project` field is not a foreign key?
- If we have to assign an employee to a project when the start date of their work on the project is unknown, what should we put in the `a_start_date` field (it is declared as `NOT NULL`, so we have to specify the date explicitly)?
- What happens if the completion date is earlier than the start date?
- What happens if the start and/or completion date falls outside of the project date range?
- Is it acceptable for the same employee to be mentioned two or more times in the same role on the same project in overlapping date ranges?

[123] “Software Testing. Base Course.” (Svyatoslav Kulikov) [https://svyatoslav.biz/software_testing_book/]

Suppose we are going to delete an entry:

- If an employee quits, do we understand correctly that we will have to delete all records of their participation in projects where the completion date is past the exit date? Or will we have to change the completion date of the project to the exit date?
- What should we do if the project is closed? Delete all the records mentioning it? Or change the completion dates for all employees on this project to the date when the project is closed?

And so on. Such questions can be asked for a long time, and only the answers from the customer representative will help us to redesign the existing database schema in the most optimal (and correct) way.

But even without such answers, it is clear that in its current form, the relation is prone to many anomalies and potentially dangerous situations, some of which cannot be solved even by reworking the database schema: for example, the issue of “the dates of an employee working on a project” are outside “the dates of the project itself” must be solved with the help of triggers{341}.

Once again, the most important thing is that at any stage of design, you should always ask questions like “what if…” and “do I correctly understand that…” Sometimes the very appearance of such a question will allow you to notice deficiencies in the database schema, and a full-fledged answer can even show a completely different and more efficient way of storing and processing data in the current situation.

And now we move on to a more formal way of obtaining relations that are not subject to data modification anomalies: normalization. In most books, the material of the next few chapters is presented in a mixed way (e.g., the description of the normalization process is interrupted to talk about this or that dependency or normal form), but here, to create a clearer picture, each topic will be treated holistically and continuously, and its relationship to related topics will be shown by references.



Task 3.1.d. This is a description of the `file` relation used in the database of a file-sharing service. Examine this relation for data operations anomalies and suggest ways to modify the relation to eliminate those anomalies.

Field name	Data type	Field properties	Description
f_uid	BIGINT UNSIGNED	Primary key.	Global file identifier.
f_fc_uid	BIGINT UNSIGNED	Foreign key, NULL values allowed.	File category identifier (FK to the “category” table).
f_size	BIGINT UNSIGNED	NULL values prohibited.	File size (bytes).
f_upload_dt	INTEGER	NULL values prohibited.	Date-time of the file upload to the server (Unixtime).
f_save_dt	INTEGER	NULL values prohibited.	Date-time till which the file should be stored on the server (Unixtime).
f_src_name	VARCHAR(255)	NULL values prohibited.	The original name of the file (how the user named the file on their computer). Without extension, because it is stored separately in the “f_src_ext” field.
f_src_ext	VARCHAR(255)	NULL values allowed.	Original extension of the file (what it was on the user’s computer).
f_name	CHAR(200)	NULL values prohibited.	Name of the file on the server. Five SHA1 hashes.
f_sha1_cs	CHAR(40)	NULL values prohibited.	Checksum of the file, SHA1 hash.
f_ar_uid	BIGINT UNSIGNED	Foreign key, NULL values allowed.	File access permissions (FK to the “permissions” table).
f_downloaded	BIGINT UNSIGNED	NULL values allowed.	File download counter.
f_al_uid	BIGINT UNSIGNED	Foreign key, NULL values allowed.	Age restrictions for access to the file (FK to “age_restriction” table). If age restrictions are specified both for the file and for the category to which the file belongs, stricter restrictions are applied (i.e., the maximum age is taken, e.g.: “16+” and “18+” — “18+” is taken).
f_del_link_hash	CHAR(200)	NULL values prohibited.	Link for the file deletion (five SHA1 hashes) if the file was uploaded by unregistered user. Registered users can delete the file in “My Files” section.



Task 3.1.e: should the “star” and “snowflake” schemas{163} be used in the “Bank{395}” database? Create an alternative version of the schema and check which data operation anomalies{157} appeared and which disappeared (compared to the original schema).



Task 3.1.f: are there relations in the “Bank{395}” database in which information about several independent real-world entities{162} falls into one relation? If you think “yes”, refine the schema to eliminate this drawback.

3.2. General Ideas about Normalization

3.2.1. Dependency Theory

If you are sufficiently familiar with the mathematical foundations of normalization and just want to refresh your memory on the normal forms themselves, you can go directly to the appropriate section{235} or skip this section and go to the next one{207}, which deals with the normalization requirements.

The content of this section will be one of the most theoretical and abstract, and at the same time it is absolutely necessary for understanding the essence of normal forms (the definition of most of which will include a reference to one dependency or another).

And normal forms and normalization themselves, as has been emphasized earlier{162}, are one of the best ways to eliminate data operation anomalies{157}.

Therefore, even though the following material may seem “out of touch” at first, you will understand its value more and more as you gain practical experience in database design.



It is crucial to understand the following fact now: any of the dependencies shown below should be considered strictly in the context of the subject area and the corresponding constraints, because when they change, the dependency itself changes as well.

A typical mistake of novice database developers is to try to represent dependencies as some general-purpose abstraction that is valid for all cases of its application.

This desire is understandable, but it greatly complicates the perception of dependencies concept in a situation when the same relation variable appears to be in some normal form or violates it (because it has one or another dependency or does not have it) when the constraints of the subject area change.

Therefore, for each dependency, a detailed explanation of the cases of its presence and absence will be given below.

Basic set theory terminology



Set[124] — an unordered set of unique elements of the same type, for which there is an operation to determine the presence of any of the elements in this set.

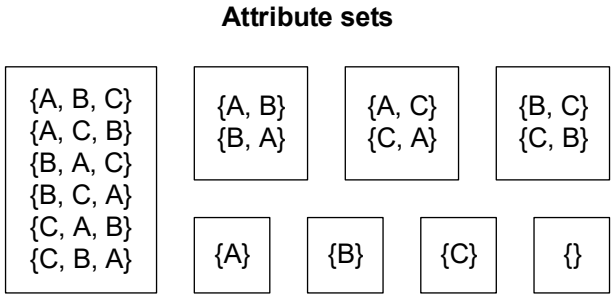
Simplified: a set of elements of the same type; these elements are not repeated (no duplicates); it is always possible to find out whether a particular element is in the set or not.

Next, we will often talk about a set of relation attributes, so let’s look at the corresponding example (see figure 3.2.a).

[124] **Set** — a collection of objects, called elements, with the property that given an arbitrary object x, it can be determined whether or not x appears in the collection (set membership). (“The New Relational Database Dictionary”, C.J. Date)

sample

A	B	C
1731	43	10
1731	42	10
1414	43	10
3443	42	10



Sets that are within the same rectangle are equivalent to each other (i.e., the order of the elements does not matter).

Figure 3.2.a — Relation and attribute sets

In mathematics, there are a number of generally accepted symbols relating to sets and operations on them. Key symbols of these are shown in the following table.

In mathematics it is customary to denote set elements by lowercase letters, but in database theory (as will be shown later) this tradition does not exist, i.e., both the sets themselves and their elements will be denoted by uppercase letters.

Symbol	Meaning	Description	Example
Brackets { and }	Set	List of elements	X = {a, b, c, d, e} Y = {c, e, f, g} Z = {b, c, a, d, e} K = {a, b, c, d} L = {h, i, j} * These values are used in the remaining examples below.
∅ or {}	Empty set	A set that contains no elements	M = {} N = ∅
∈	Element	The element is in the set	a ∈ X
∉	NOT element	The element is **not** in the set	a ∉ Y
| and | or # symbol	Cardinality	Number of elements in the set	|X| = 5 #Y = 4
=	Equality	Sets contain the same collection of elements	X = Z
∩	Intersection	A collection of only such elements that are **in both one** set **and the other** set	X ∩ Y = {c, e}
∪	Union	The collection of elements of both sets	X ∪ Y = {a, b, c, d, e, f, g}
⊆	Subset	One set is part of the other one	K ⊆ X Z ⊆ X
⊂	Strict subset	One set is part of the other, but they are not equal to each other	K ⊂ X
⊄	NOT subset	One set is **not** part of the other one	L ⊄ X
⊇	Superset	One set contains the other one	X ⊇ K X ⊇ Z
⊃	Strict superset	One set contains the other, but they are not equal to each other	X ⊃ K

⊅	NOT superset	One set does not contain the other one	X ⊅ L
\ or -	Complement (difference)	A collection of only those elements that exist **only** in one set (i.e., they do **not** exist in the other set)	X - Y = {a, b, d}
×	Cartesian product	The set of all possible combinations of elements of one and another set	Let P = {a, b} and Q = {c, d, e}, therefore P × Q = {{a, c}, {a, d), {a, e}, {b, c}, {b, d}, {b, e}}

Since set theory is incomparably more extensive and complex than is shown in this brief reference material, it is advisable to read "The Joy of Sets (2nd ed.)" (by Keith Devlin), which presents all the necessary information in a very simple and clear form.

Basic terminology related to all types of dependencies

There are several basic terms, both underlying the normalization process and used in the definitions of the dependencies and normal forms discussed below. Let's consider these terms.

Projection[125] — a subset of attributes of the relation and their corresponding tuples such that all combinations of values of the selected attributes in the projection tuples are also found in the original relation.

Simplified: some of the attributes (the rest are ignored) and all tuples are selected from the original relation; duplicates of all tuples in the resulting new relation are removed.

The essence of the projection is easiest to explain on an example from geometry, familiar from school (see figure 3.2.b). Let's assume that in three-dimensional space the positions of points are given by three coordinates (x, y, z).

The projection of any point onto each of the three planes will be given by only two coordinates (e.g., (x, y)) — this effect is analogous to selecting a subset of attributes of a relation.

Since several points in space may have some coordinates coincide, on the projection they will become one point — this effect is analogous to the elimination of duplicate tuples.

So, in figure 3.2.b we have points A(x_1, y_1, z_1) and B(x_2, y_1, z_1). Since their coordinates (*y_1 for both points*) and (*z_1 for both points*) coincide, the projections of both points on the YZ plane become one point (coincide).

[125] Let relation r have attributes called A1, A2, …, An (and possibly others). Then (and only then) the expression r{A1,A2,...,An} denotes the **projection** of r on {A1, A2, …, An}, and it returns the relation with heading {A1,A2,...,An} and body consisting of all tuples t such that there exists a tuple in r that has the same value for attributes A1, A2, ..., An as t does.

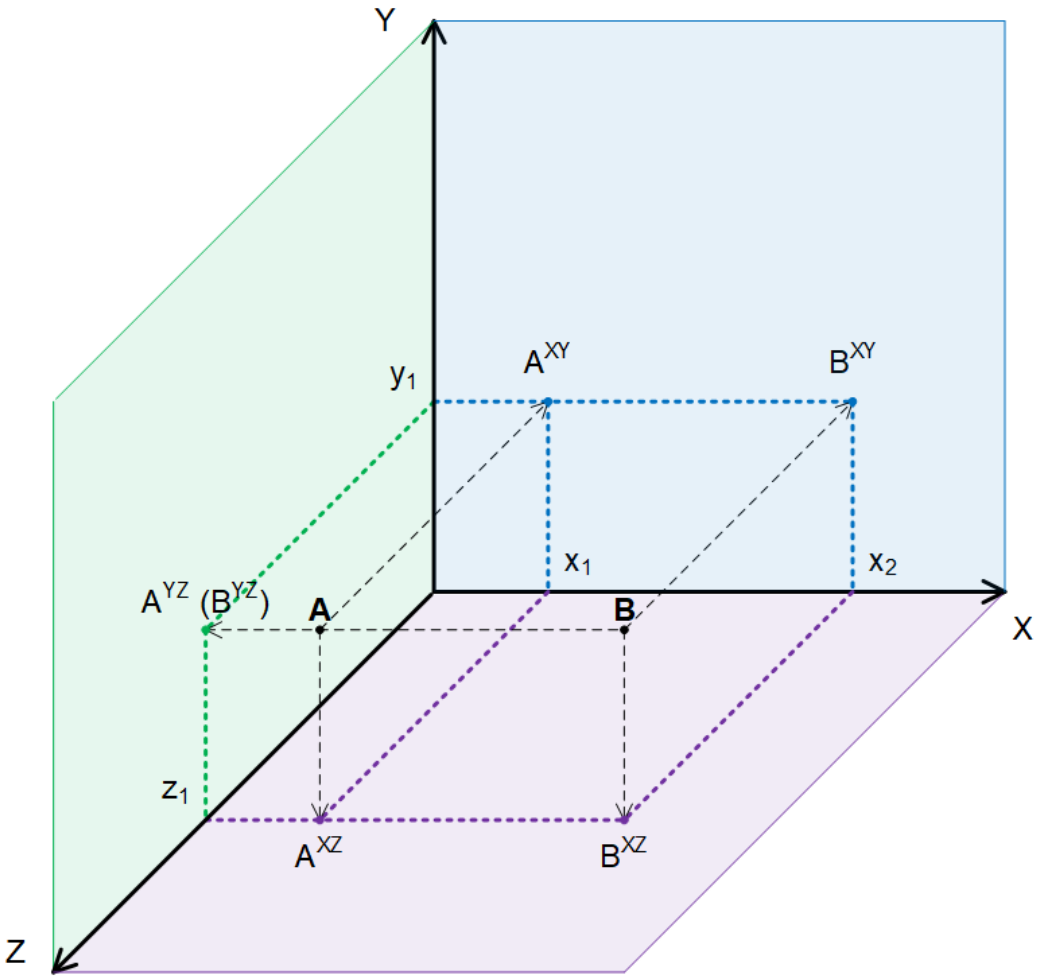


Figure 3.2.b — Projections in geometry

Now, from the geometrical analogy, let's return to databases and consider the projections of the **`sample`** relation shown in figure 3.2.a.

According to the above definition{172} we select some set of attributes from the original relation and then eliminate duplicates in the resulting new relation.

For the original **`sample`** relation with attributes **`A`**, **`B`**, **`C`** we can obtain the following projections: **`AB`**, **`BC`**, **`AC`** (shown in figure 3.2.a), as well as the non-practical projections **`A`**, **`B`**, **`C`**.

To put it strictly, it is also possible to get a projection with zero attributes (i.e., a relation with nothing in it). But it is easy to understand that in practice we will not be interested in it either.

It is important to understand that any relation can be projected on any set of its attributes (and the more attributes, the more possible variants are obtained), but now we will show that far from any projection is applicable in practice — because later it will be necessary to recreate exactly the original relation from the projections we have.

In the example shown in figure 3.2.c the projection **`AB`** and **`BC`**, as well as **`AB`** and **`AC`** are applicable.

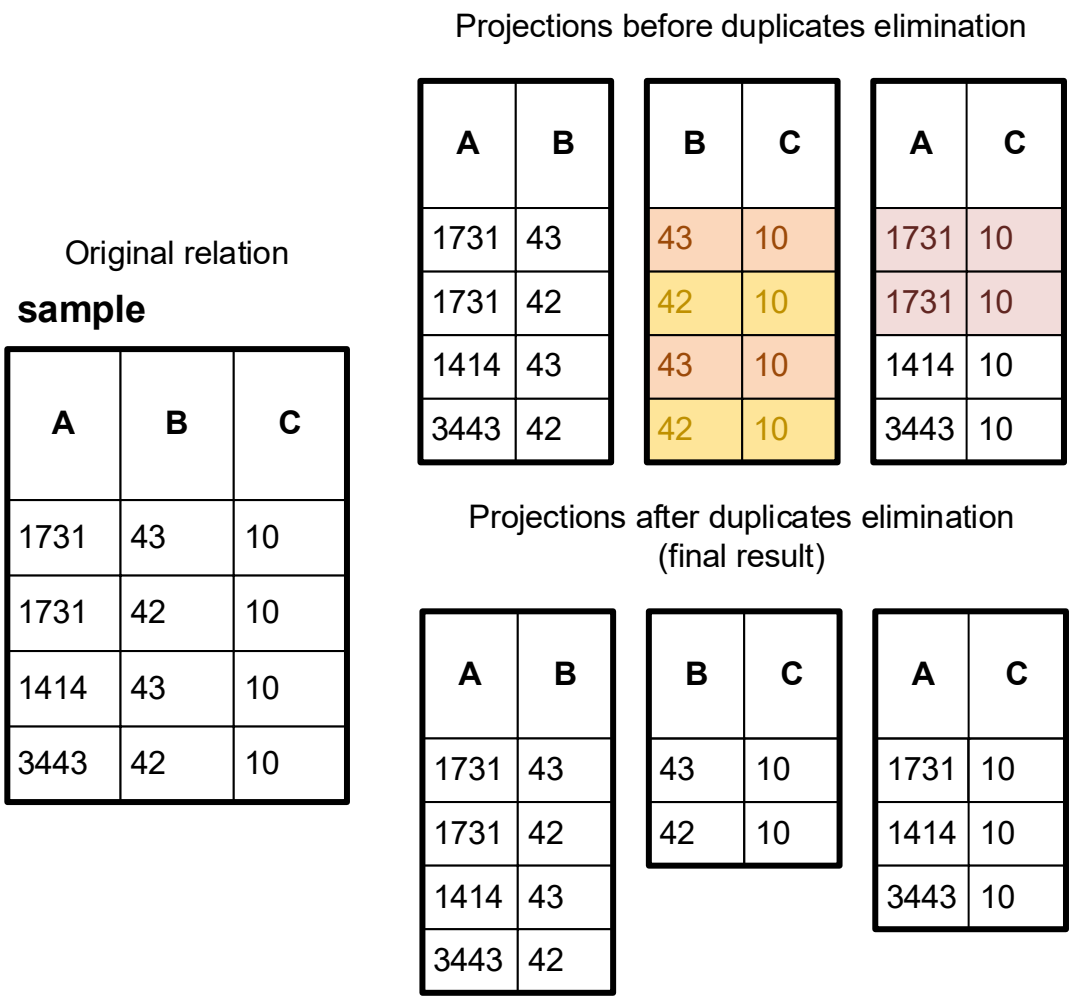


Figure 3.2.c — Projections in databases

If we consider the projections `BC` and `AC`, when we try to reconstruct the original relation, we get two extra tuples that did not exist before (see figure 3.2.d). Going ahead, we will say that there is a fifth normal form to solve this problem{262}.

Now that we have considered in detail the concept of relation projections, it will be easy to move on to the next term — lossless decomposition.



In various sources one can find such terms as “lossless decomposition”, “non-loss decomposition”, “lossless-join decomposition”, “nonadditive decomposition” — all these terms mean the same thing (and, as a rule, can be replaced by simply “decomposition”, if it is not clear from the context that “lossless decomposition” is opposed to “just a decomposition” (which allows distortion of information when reconstructing a relation from its projections)).

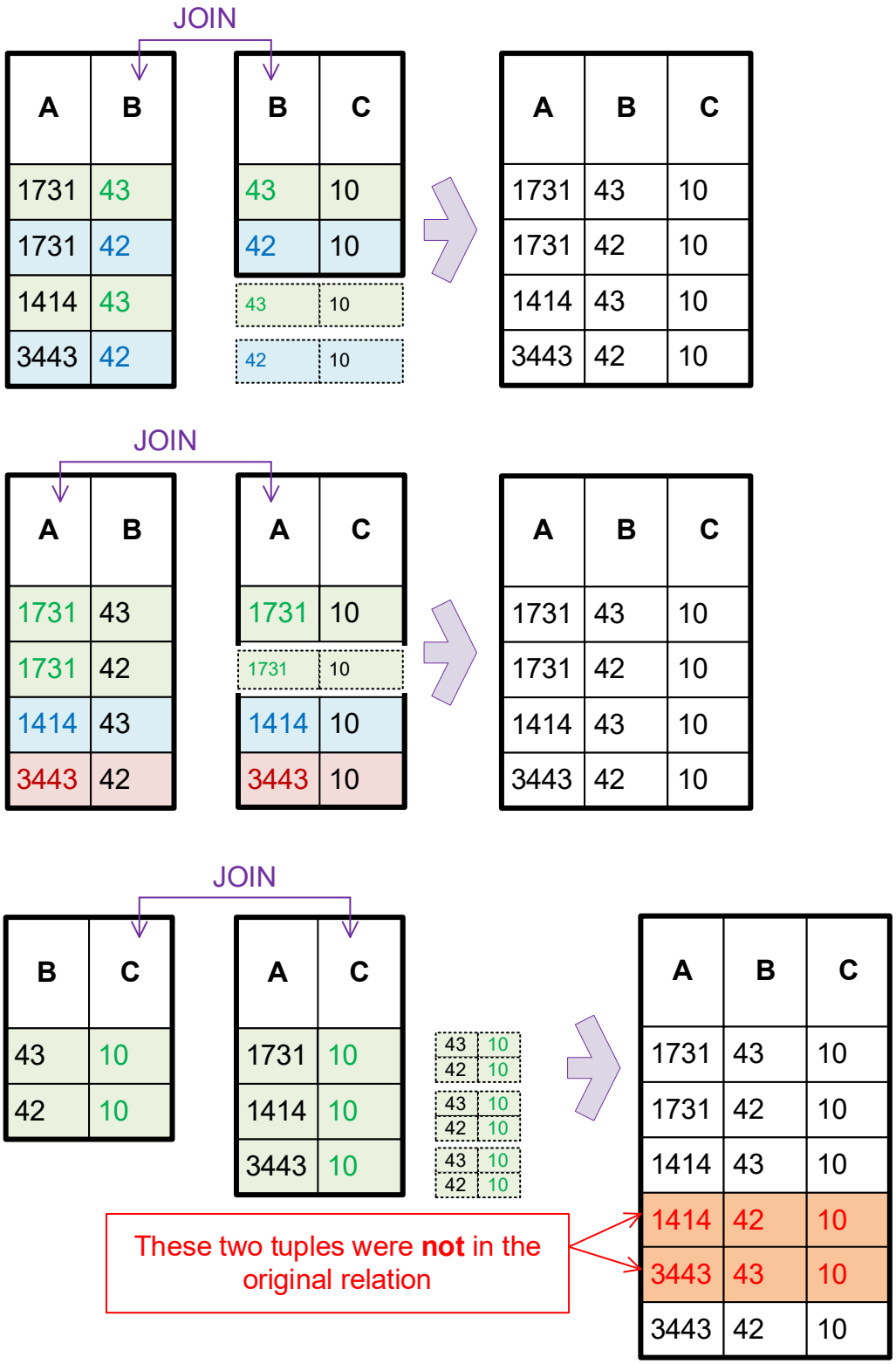


Figure 3.2.d — Recreating the original relation from its projections



Nonloss[126] **decomposition**[127] (lossless decomposition, lossless-join decomposition, nonadditive decomposition) — a replacement of a relation by a set of its projections, for which the following requirements are met:

- natural join[128] of the projections is guaranteed to result in the exact original relation (a mandatory requirement);
- all such projections are necessary to obtain the original relation (an optional requirement);
- at least one of the projections is in a higher normal form than the original relation (an optional requirement).

Simplified: the original relation is broken down into two or more projections, and it is guaranteed that the original relation can be derived from them, there are no extra (unnecessary) projections among them, at least one of them is in a higher normal form.

Figure 3.2.c shows the bad case, i.e., decomposition of the relation into such projections, among which there are redundant ones, as well as not allowing to get the original relation.

Figure 3.2.d shows two good cases, i.e., decomposition of the relation into projections **AB** + **BC**, **AB** + **AC**, which fully satisfy all three conditions of the definition: they can be used to obtain the original relation, there are no extra projections among them, they are in higher normal forms (at least the projection **BC** is in the 5th normal form{262}).

Now, based on the basic concepts just discussed, we will examine the dependencies on which certain normal forms are based.

Dependencies on which the second normal form is based

Yes, there is nothing missing here: the first normal form is not related to dependency theory, so we start with those on which the second normal form is based{241}.



Functional dependency[129] between two sets of attributes{20} X and Y, which are subsets of attributes of relation{19} R, means the *constraint* imposed on tuples{20} of the value r of relation{22} R, which means that whenever any two tuples t_1 and t_2 fulfill the equality $t_1[X] = t_2[X]$, the equality $t_1[Y] = t_2[Y]$ must also be fulfilled. The functional dependency is denoted by $X \rightarrow Y$ with X as the determinant and Y as the dependent part[130].

Simplified: if there is a functional dependency between the sets of attributes X and Y, there is a rule that if two rows in the table have the same X value, then the Y value must also be the same.

[126] **Nonloss decomposition** — replacing a relvar R by certain of its projections R_1, R_2, ..., R_n, such that (a) the join of R_1, R_2, ..., R_n is guaranteed to be equal to R, and usually also such that (b) each of R_1, R_2, ..., R_n is needed in order to provide that guarantee (i.e.none of those projections is redundant in the join), and usually also such that (c) at least one of R_1, R_2, ..., R_n is at a higher level of normalization than R is. (“The New Relational Database Dictionary”, C.J. Date)

[127] The “decomposition” term has no special definition in the database theory, because it is a commonly used word (its synonyms: “partitioning”, “division”, “distribution”, etc.)

[128] What is meant here is the NATURAL JOIN operation, i.e., the JOIN of tables by all their columns of the same name.

[129] A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

[130] Often the “dependent part” is simply called a “function” and it’s said “Y is a function of X”.

For a very quick understanding of the essence of functional dependency we can recall the school mathematics course, where the expression “Y is a function of X” was often heard (see figure 3.2.e). Indeed, if “X set” is a candidate key of a table, this figure graphically illustrates the notion of functional dependency to the fullest extent and without any assumptions.



Important! Many years of training experience show that many people try to call functional dependency a “direct dependency” or “a forward dependency”. This is incorrect. Moreover — these terms generally belong to different fields of science. Do not make this mistake!

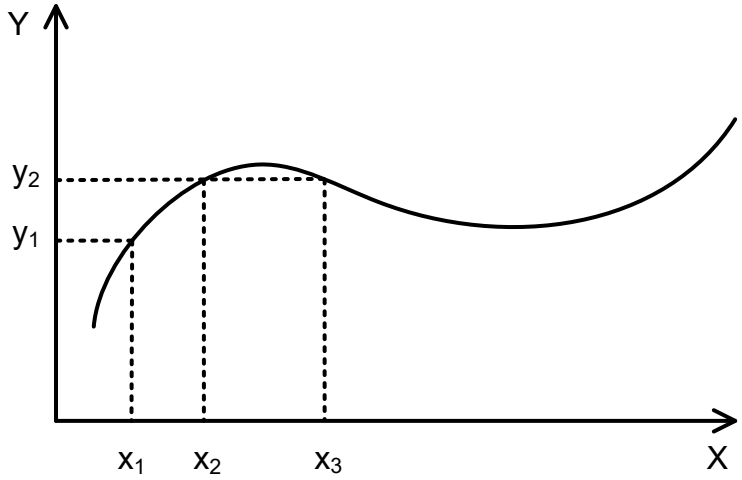


Figure 3.2.e — Graphical representation of the functional dependency

The definition{176} of functional dependency implies that one value of X set must correspond strictly to one value of Y set (indeed, if several different values of Y set were allowed, we could have a situation in which two or more tuples contain the same values of X, but they correspond to different values of Y, which contradicts the definition).

This important consequence brings us right up to examples from the field of databases.

First, let’s consider examples in which the functional dependency is present:

- {Passport id} → {Full name};
- {Personal employee id} → {Time of service in the company};
- {Student id, Course id} → {Final grade}.

Now let’s consider examples where there is no functional dependency:

- {Passport id} → {Full name};
- {Personal employee id} → {Time of service in the company};
- {Student id, Course id} → {Final grade}.

If you think there is some mistake, it is worth re-reading the cautionary note{170} at the beginning of this chapter: as long as we do not specify the constraints of the subject area, we cannot guarantee the presence or absence of functional dependency.

If the subject area says that one passport id corresponds to one full name, then there is a functional dependency {Passport id} → {Full name}. If it is said that one passport id can correspond to several names (e.g., in different languages or considering maiden name), there is no functional dependency here.

If the subject area specifies that one personal employee id corresponds to strictly one value of time of service in the company, the functional dependency {Personal employee id} → {Time of service in the company} is present. If it is said that one employee's personal id can correspond to several values of time of service in the company (e.g., a separate value for each period of work is considered for cases when an employee is fired and returned), there is no functional dependency here.

If the subject area says that there is only one final grade for each student in each course, there is a functional dependency {Student id, Course id} → {Final grade}. If it says that each student can have several final grades for each course (e.g., including retakes), there is no functional dependency.



Once again, let's emphasize: the presence or absence of functional dependency between some sets of attributes is determined solely by the constraints of the subject area.



A very detailed description of functional dependency and related terms can be found in the following books:

- "Fundamentals of Database Systems (6th edition)" (by Ramez Elmasri, Shamkant Navathe) — chapter 15.
- "An Introduction to Database Systems (8th edition)" (by C.J. Date) — chapter 11.

To conclude the discussion of functional dependency, here is another very revealing example, which often confuses novice database developers. The previously considered implication{177} says that if there is a functional dependency X → Y, one value of X corresponds to one value of Y, and this is true. But the reverse is not true, i.e., one value of Y does not necessarily correspond to one value of X.

Let's illustrate this with the example already discussed (see figure 3.2.f) using the `result` relation.

result

PK		
r_student_id FK	**r_subject_id** FK	**r_mark**
1731	43	10
1731	42	10
1414	43	10
3443	42	10

{r_student_id, r_subject_id} → {r_mark}

~~{r_mark} → {r_student_id, r_subject_id}~~

Figure 3.2.f — Correct understanding of the value correspondence within the functional dependency

Suppose the subject area says that there is only one final grade for each student in each course, i.e., there is a functional dependency {Student id, Course id} → {Final grade}. But this does not prevent several students from getting the same grades in the same course, and there is no violation of the functional dependency here.

There are several other important terms associated with functional dependency which are necessary for understanding the second{241} normal form.

!!! **Full functional dependency**[131] — a functional dependency X → Y in which removing any subset of attributes A from the X set destroys the functional dependency, i.e., for any A ε X, the statement (X - A) → Y will be false.

Simplified: if we remove at least one element from the X set, the remaining set is no longer sufficient to fulfill the "one value of X defines one value of Y" statement.

!!! **Partial functional dependency**[132] — a functional dependency X → Y in which removing some subset of attributes A from the X set does not destroy the functional dependency, i.e., for some A ε X, the statement (X - A) → Y will be true.

Simplified: there is an element which, if removed from the X set, the remaining set of elements will still be sufficient to fulfill the "one value of X defines one value of Y" statement.

[131] A functional dependency X → Y is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute A ε X, (X - {A}) does not functionally determine Y. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

[132] A functional dependency X→Y is a **partial functional dependency** if some attribute A ε X can be removed from X and the dependency still holds; that is, for some A ε X, (X - {A}) → Y. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

It is not for nothing that these two definitions are considered together, because they describe two opposite situations, which are easiest to explain using the example of the composite{36} primary key{36}.

Here we are talking about a composite primary key, because by the definition of both dependencies considered here the X set must allow at least one element to be removed, while a simple{36} primary key initially contains only one element, the removal of which makes no sense (since, in fact, we would remove the key itself by doing so).

An example of a full functional dependency can be found in the **`result`** relation already discussed earlier (see figures 3.2.f, 3.2.g).

Indeed, removing any of the fields from the composite primary key {**`r_student_id`**, **`r_subject_id`**} will cause the grade in the **`r_mark`** field to refer either simply to the subject without taking the student into account, or simply to the student without taking the subject into account. That is, we will no longer be able to answer the question "what grade did student so-and-so get in subject so-and-so", because we will have no information about either the student or the subject.

Moreover, each of the **`r_student_id`** and **`r_subject_id`** fields separately cannot act as a primary key, because in this relation the values of these fields should be objectively duplicated (each student gets grades in several subjects, hence the student identifier should be duplicated; for each subject several students get grades, hence the subject identifier should be duplicated).

Speaking of partial functional dependency, it is worth emphasizing that there can be both several full and several partial functional dependencies within one relation at the same time.

Let's demonstrate this with the **`result`** relation (we will have one full and one partial functional dependency, but obviously with the addition of new attributes we can get several cases of both dependencies).

Let's add the **`r_payment_id`** attribute to the **`result`** relation, showing the student's form of education payment (1 = "State-funded", 2 = "Study-for-fee") — see figure 3.2.h. If you wondered if this solution is wrong, you are absolutely right — this way we get a violation of the second{241} normal form, but for an example of partial functional dependency this relation fits very well.

Obviously, the value of the student's form of education payment identifier depends only on the student identifier value and does not depend on the course identifier value. Therefore, removing **`r_subject_id`** from {**`r_student_id`**, **`r_subject_id`**} set does not destroy the {**`r_student_id`**, **`r_subject_id`**} → {**`r_payment_id`**} functional dependency.

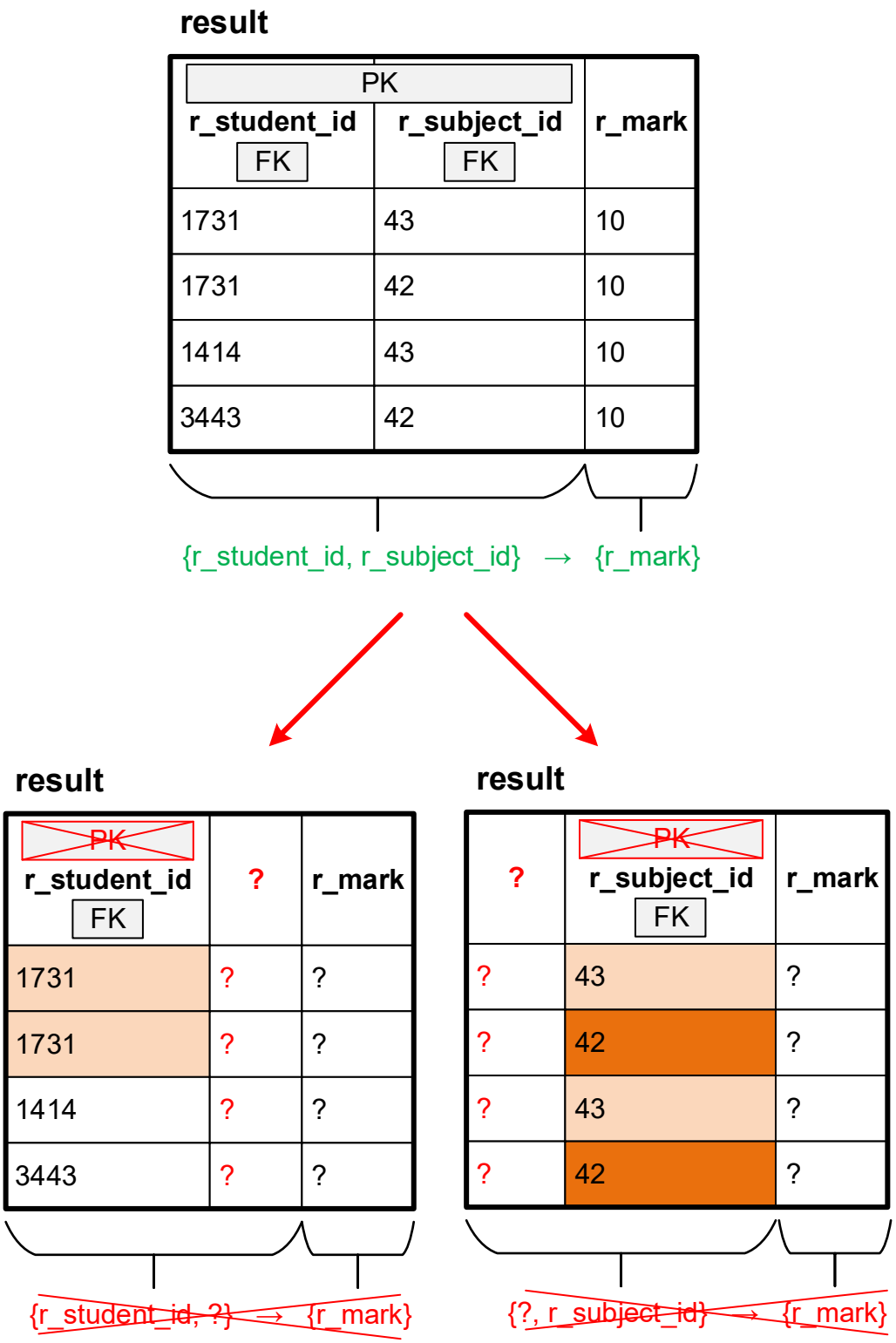


Figure 3.2.g — Full functional dependency

result

PK			
r_student_id FK	**r_subject_id** FK	**r_mark**	**r_payment_id**
1731	43	10	1
1731	42	10	1
1414	43	10	2
3443	42	10	1

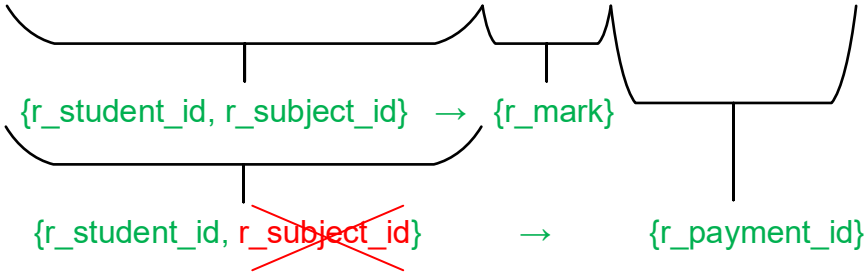


Figure 3.2.h — Partial functional dependency

For better memorization, let's again briefly show the essence of the just considered dependencies (see figure 3.2.i):

FD – for one X value there is only one Y value	$X\{A_1, A_2, \ldots A_n\} \rightarrow Y$
Full FD – not a single element can be removed from the X set, so as not to lose the way to find out the value of Y	$X\{\mathbf{A_1, A_2, \ldots A_n}\} \rightarrow Y$
Partial FD – It is possible to remove some elements from the X set and not lose the way to find out the value of Y	$X\{\mathbf{A_1}, \cancel{\mathbf{A_2}}, \mathbf{\ldots A_n}\} \rightarrow Y$

Figure 3.2.i — Functional dependency, full functional dependency, partial functional dependency

To conclude this part of the section, let's note that functional dependency can be trivial{187} and nontrivial{187}, which will be discussed below.

Dependencies on which the third normal form and the Boyce-Codd normal form are based

The following varieties of dependencies (related to the third{247} normal form{253} and the Boyce-Codd normal form{253}) should also be considered together, since they are closely related at the level of definitions and scope of application.

!!! **Transitive dependency**[133] — a functional dependency X → Z in R relation in which the conditions X → Y and Y → Z are satisfied, where Y is also a subset of the attributes of R relation, but is not a candidate key{34} or a subset of any of keys{32} of R relation.

Simplified: a "chain of functional dependencies" like X → Y → Z, in which one value of X defines one value of Y, and one value of Y defines one value of Z.

No formal definitions are given for the following three concepts in trustworthy official primary sources, but nevertheless we will formulate them.

!!! **Redundant dependency** — dependency between X and Y attribute sets of R relation, which can be derived from other dependencies within R relation.

Simplified: a "superfluous" dependency that provides no additional information compared to other dependencies.

In general, redundant dependency can be of the following two kinds:

!!! **Redundant transitive dependency** — a transitive dependency between the X and Z attribute sets of R relation (determined by the existence of dependencies X → Y and Y → Z), in which the dependency X → Z also exists.

Simplified: a "chain of functional dependencies" like X → Y → Z, in which there is a dependency X → Z ("directly", without Y).

!!! **P**seudotransitive dependency — a redundant functional dependency that appears in the following case: if for X, Y, W, Z attribute sets of R relation there are functional dependencies X → Y, {Y, W} → Z and {X, W} → Z, then the {X, W} → Z dependency is redundant.

Simplified: if the determinant of some functional dependency can be derived from other functional dependencies, there is no need for another functional dependency whose determinant is "collected" from the determinants of the first two dependencies.

Now let's show all these cases in sequence by examples starting with the transitive dependency (see figure 3.2.j):

[133] A functional dependency X → Y in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R and both X → Z and Z → Y hold. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

current_result

PK cr_student_id	cr_average_mark	cr_current_status
1731	8.34	Good
2352	9.99	The Best!
5632	1.23	Oh, my God… ☹
4534	6.45	Normal

{cr_student_id} → {cr_average_mark} → {cr_current_ststus}

Figure 3.2.j — Transitive dependency

Here the value of average grade (`cr_average_mark` attribute) depends on the value of student id, and the status value (`cr_current_status` attribute) depends on the value of average grade (but does not depend on the value of student id), so we get a "chain" like `student id` → `average grade` → `status`, in which each "step" depends on the previous one — this is a transitive dependency.

Obviously, such a "chain" can be longer than three elements, although in practice such cases are rare.

Redundant dependency, as was noted earlier, in practice has two main forms of manifestation — redundant transitive dependency (see figure 3.2.k) and pseudotransitive dependency (see figure 3.2.l).

For `access_control` relation the pass id value (`ac_pass_id` attribute) depends on the student id value (`ac_student_id` attribute), and the student name value (`ac_student_name` attribute) depends on the pass id value (`ac_pass_id` attribute). Thus, there is a transitive dependency `student id` → `pass id` → `student name`.

But the student's name depends not only on the pass id, but also on the identifier of the student themselves, i.e., the condition `student id` → `student name` is met. Thus, the "middle link" in this "chain" is "superfluous" and can be eliminated without losing the ability to find out the student's name from the student's id.

Sure, if we simply delete the `ac_pass_id` from `access_control` relation, we will lose information about students' passes, but in this case (as well as in others) we are not talking about deleting attributes of the relation — we can do a much more efficient thing by decomposing{176} the original relation into several new ones, which we will talk about in detail when examining the normalization process{218} and the normal forms{235} themselves.

access_control

PK ac_student_id	ac_pass_id	ac_student_name
1731	34523	Samuel Smith
2352	46362	Jane Johnes
5632	45346	Tom Taylor
4534	56745	Tim Taylor

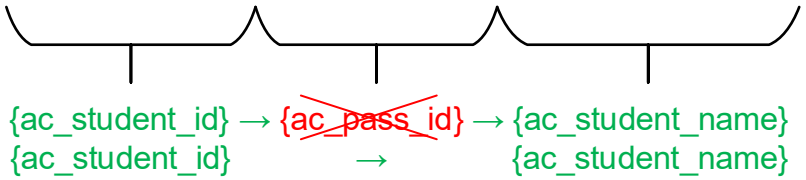


Figure 3.2.k — Redundant transitive dependency

Before examining the pseudotransitive dependency, let's remind ourselves once again: all the reasoning given here is relevant only in the context of the stipulated constraints of the subject area.

Here we slightly rework the `access_control` relation (see figure 3.2.l) and agree that:

- the pass *may* be deactivated (the `ac_is_pass_active` attribute value will be "No") for some objective reason that does not depend on the pass status (`ac_pass_status` attribute), but only on the student id (`ac_student_id` attribute);
- the pass **must** be deactivated (`ac_is_pass_active` attribute value will be "No") if the pass status (`ac_pass_status` attribute) is "Not issued" or "Lost".

access_control

PK ac_student_id	ac_pass_id	ac_pass_status	ac_is_pass_active
1731	34523	Issued	No
2352	46362	Issued	Yes
5632	45346	Not issued	No
4534	56745	Lost	No

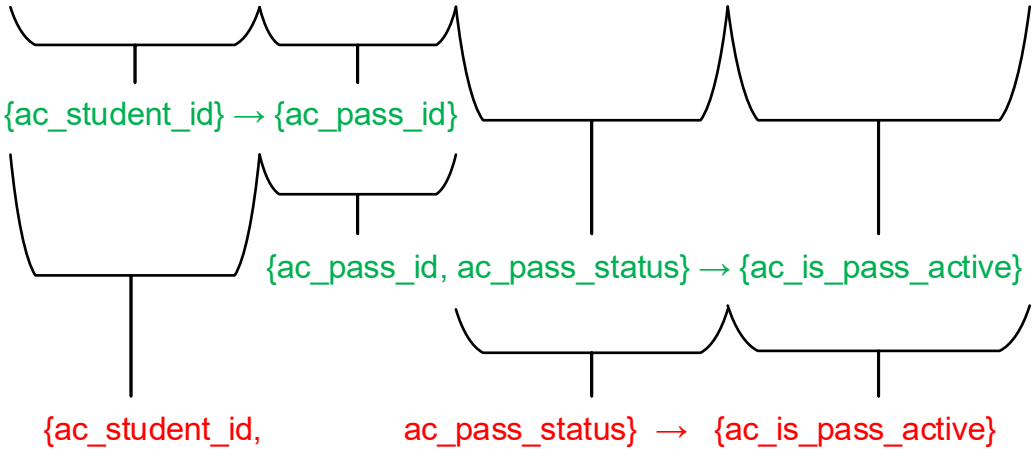


Figure 3.2.l — Pseudotransitive dependency

Thus, we obtain the following functional dependencies:

1. {**ac_student_id**} → {**ac_pass_id**}
2. {**ac_pass_id**, **ac_pass_status**} → {**ac_is_pass_active**}
3. {**ac_student_id**, **ac_pass_status**} → {**ac_is_pass_active**}

Why is the third dependency redundant? Because substituting the first dependency for {**ac_student_id**} will give us the second dependency. This is easier to explain graphically — see figure 3.2.m.

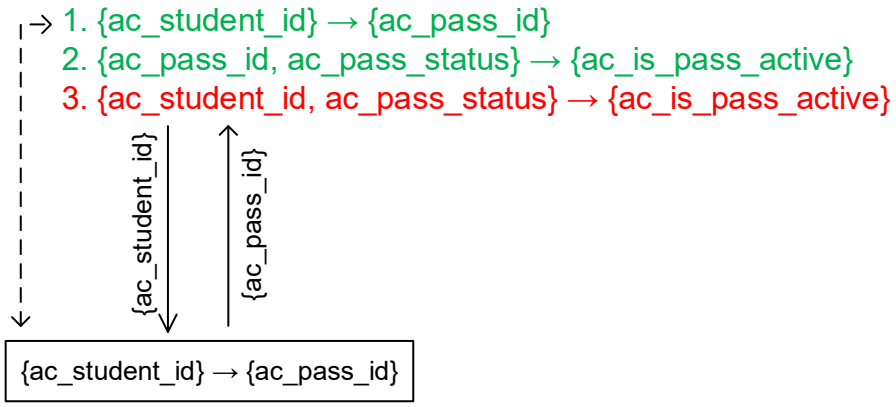


Figure 3.2.m — Explanation of the idea of pseudotransitive dependency

To understand the essence of the third{247} normal form, we need to consider two more definitions of functional dependencies.



Trivial functional dependency[134] — a functional dependency that cannot be violated, which is fulfilled under the following condition: $X \rightarrow Y$, $X \supseteq Y$.

Simplified: the dependency of a part of a set of attributes on the whole set, by definition, cannot be violated (i.e., always exists).



Nontrivial functional dependency[135] — a functional dependency that can be violated, which is fulfilled under the following condition: $X \rightarrow Y$, $X \not\supset Y$.

Simplified: the dependency of one set of attributes on another set may or may not hold (i.e., it may be violated).

Let's consider these dependencies on a graphical example (figure 3.2.n).

The essence of trivial functional dependency is that if we know the whole set of attribute values, then we also know exactly the value of each individual attribute (or the value of the combination of attributes) in this set.

So, if we know some value pair of the set {**`r_student_id`**, **`r_subject_id`**} (for example, {1731, 43}), hence we know exactly that in this pair the first attribute (**`r_student_id`**) is 1731, and the second attribute (**`r_subject_id`**) is 43. Here the following trivial functional dependencies are fulfilled:

- {**`r_student_id`**, **`r_subject_id`**} → {**`r_student_id`**}
- {**`r_student_id`**, **`r_subject_id`**} → {**`r_subject_id`**}
- {**`r_student_id`**, **`r_subject_id`**} → {**`r_student_id`**, **`r_subject_id`**}

Another example, even a little simpler: if we know that a person's full name {**`last name`**, **`given name`**, **`patronymic`**} equals {Smith, Smit, Smithson}, then we objectively know that his **`last name`** is Smith, **`given name`** is Smit, **`patronymic`** is Smithson. Here the following trivial functional dependencies are fulfilled:

- {**`last name`**, **`given name`**, **`patronymic`**} → {**`last name`**}
- {**`last name`**, **`given name`**, **`patronymic`**} → {**`given name`**}
- {**`last name`**, **`given name`**, **`patronymic`**} → {**`patronymic`**}
- {**`last name`**, **`given name`**, **`patronymic`**} → {**`last name`**, **`given name`**}
- {**`last name`**, **`given name`**, **`patronymic`**} → {**`given name`**, **`patronymic`**}
- {**`last name`**, **`given name`**, **`patronymic`**} → {**`last name`**, **`patronymic`**}
- {**`last name`**, **`given name`**, **`patronymic`**} → {**`last name`**, **`given name`**, **`patronymic`**}

And even a simpler example (for a set consisting of one element): if we know that a person's passport id {**`passport`**} is AA123456, then we know that their passport id is AA123456. Here the only trivial functional dependency is:

- {**`passport`**} → {**`passport`**}

[134] **Trivial functional dependency** — an FD that can't possibly be violated. The FD $X \rightarrow Y$ is trivial if and only if $X \supseteq Y$. ("The New Relational Database Dictionary", C.J. Date)

[135] **Nontrivial functional dependency** — a functional dependency $X \rightarrow Y$ is trivial if $X \supseteq Y$; otherwise, it is **nontrivial**. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

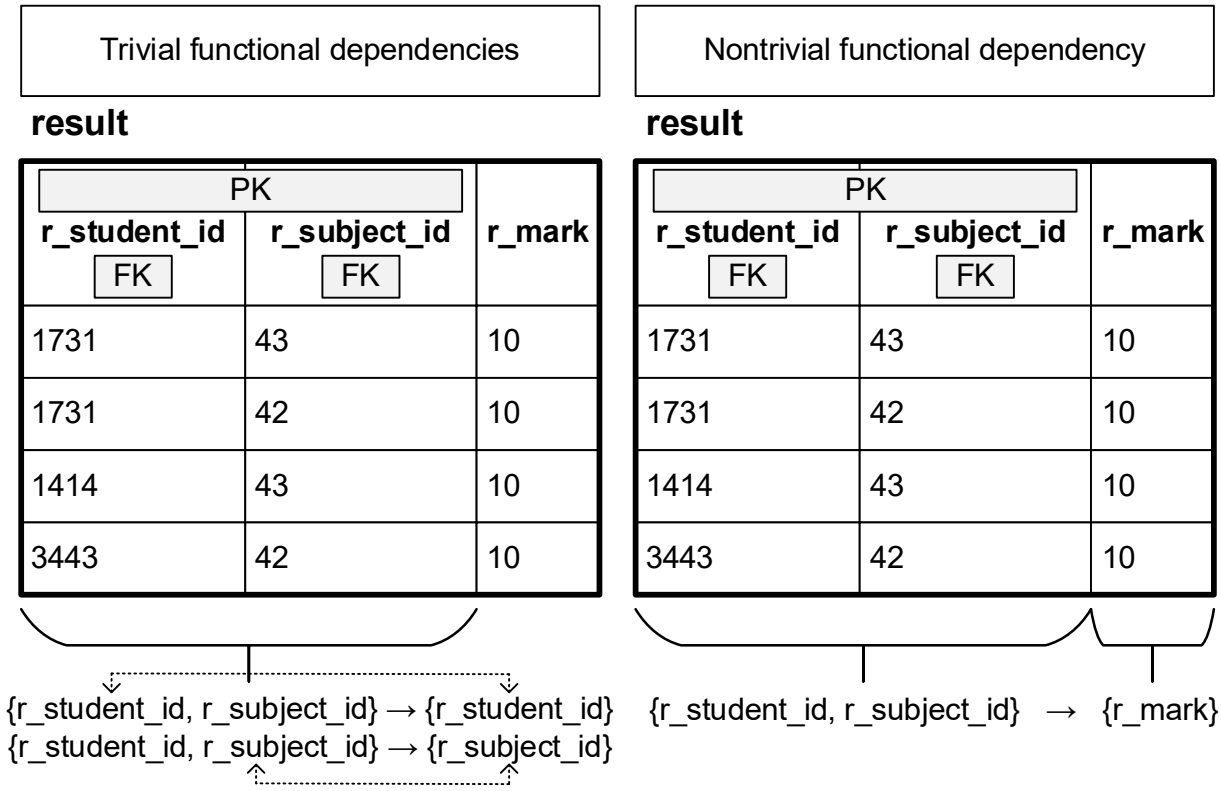


Figure 3.2.n — Trivial and nontrivial functional dependencies

Nontrivial functional dependency is described even in the definition as the opposite of trivial, i.e., here the functionally dependent part is not a part of the function determinant (e.g., in figure 3.2.n the **`r_mark`** attribute, which depends on the {**`r_student_id`**, **`r_subject_id`**} set, is not a part of this set).

The full{179} and partial{179} functional dependencies discussed earlier are just examples of nontrivial functional dependencies.

So, by now we have considered all the dependencies necessary to understand the third{247} normal form and the Boyce-Codd{253} normal form.

Dependencies on which the fourth normal form is based

The following dependency is a prime example of how, in relational theory, some statements and conclusions affect others: the so-called “multivalued dependency” is a consequence of the first normal form{236} that forbids creating multivalued attributes in relations.

!!!

Multivalued dependency[136] — dependency in R relation{19} (denoted by $X \twoheadrightarrow Y$ or $X \twoheadrightarrow Y|Z$), where X, Y and Z = (R — (X ∪ Y) — are subsets of R attributes{20}) defining the following constraint. If there are two tuples{20} t_1 and t_2 such that $t_1[X] = t_2[X]$, then tuples t_3 and t_4 must also exist and the following conditions have to be met:
a) $t_3[X] = t_4[X] = t_1[X] = t_2[X]$;
b) $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$;
c) $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

Simplified: multivalued dependency requires that a relation containing two tuples that match one of the three attributes also contain two additional tuples “crosswise” containing combinations of the remaining two attributes (see figure 3.2.o).

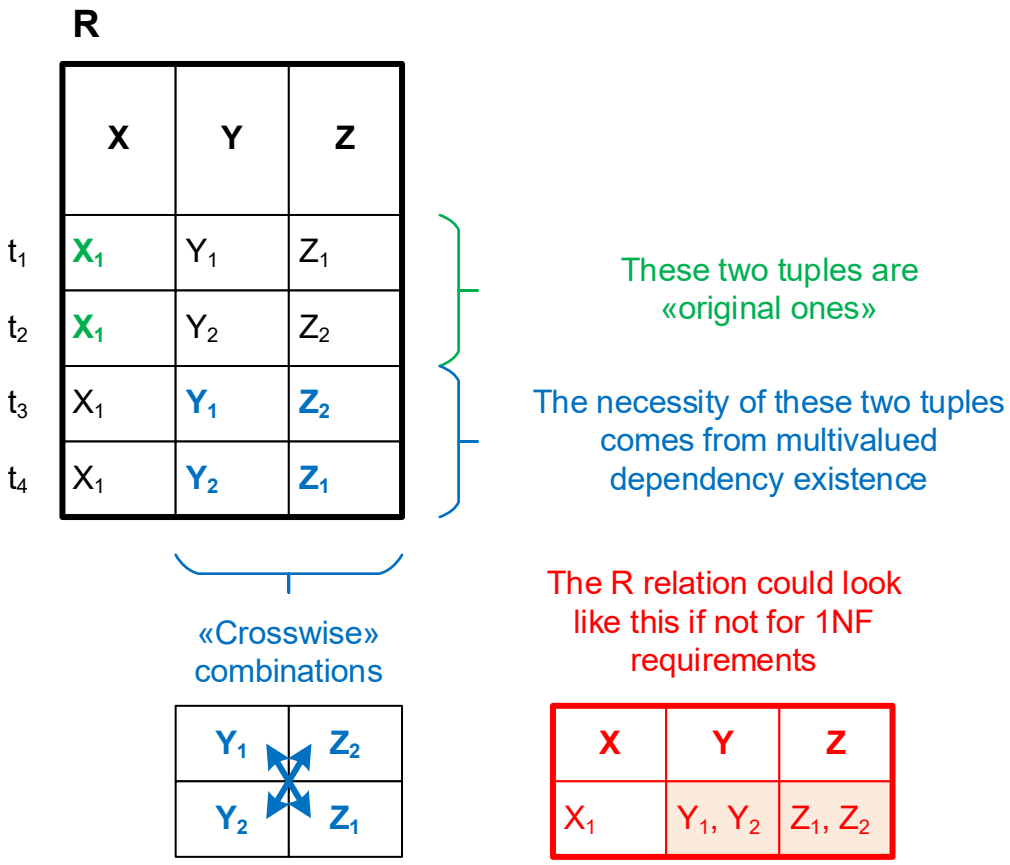


Figure 3.2.o — Graphic explanation of multivalued dependency

[136] A **multivalued dependency** $X \twoheadrightarrow Y$ specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote (R — (X ∪ Y)): a) $t_3[X] = t_4[X] = t_1[X] = t_2[X]$; b) $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$; c) $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

In layman's terms, in the presence of a multivalued relation "there are two one to many relationships", i.e., one value of X corresponds to many (two or more) values of Y, and also one value of X corresponds to many (two or more) values of Z.

The only way to reflect this situation without violating the 1NF is to add additional tuples, because without them it would not be possible to show the independence of the values of Y and Z from each other (it would seem that $Y \to Z$, i.e., the value of Y_1 strictly corresponds to Z_1, and Y_2 strictly corresponds to Z_2).

This reasoning leads us to the following two definitions.



Trivial multivalued dependency[137] — a multivalued dependency $X \twoheadrightarrow Y$ in which Y is a subset of X or $R = X \cup Y$.

Simplified: a multivalued dependency is trivial if the set of Y attributes is part of the set of X attributes, or the collection of the X and Y sets makes up the whole set of attributes of R relation (then by the basic definition{189} there are no attributes left for the Z set, i.e., it is empty).



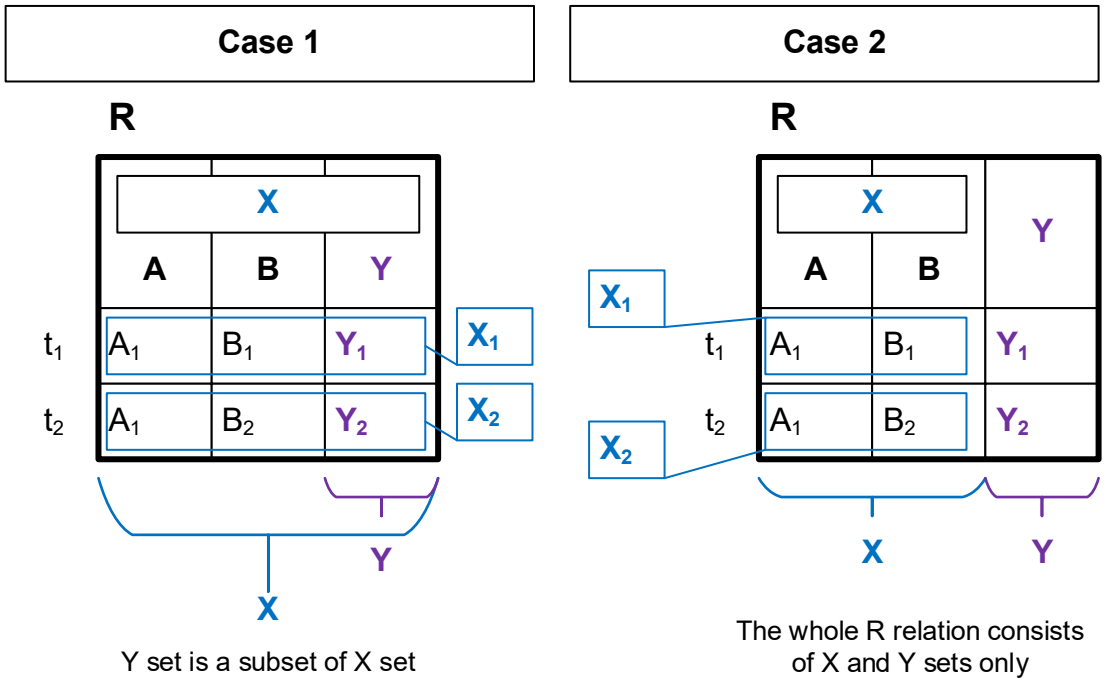
Nontrivial multivalued dependency[137] — a multivalued dependency $X \twoheadrightarrow Y$ in which Y is not a subset of X and $R \neq X \cup Y$.

Simplified: a multivalued dependency is nontrivial if the set of Y attributes is not part of the set of X attributes, and the collection of the X and Y sets does not constitute the whole set of attributes of R relation (so that by the basic definition{189} for Z set there remains some attributes that are not part of either X or Y sets).

It is easy to guess that when we consider the fourth normal form{258} we will be interested in the nontrivial multivalued dependency (this is what is shown in figure 3.2.o). And the two cases where the multivalued dependency is trivial are shown in figure 3.2.p.

There is also another informal indication that multivalued dependency is trivial: if multivalued dependency exists for X, Y, Z attributes, and at the same time any "internal" functional dependency{176} ($X \to Y$, $X \to Z$, $Y \to Z$, $Y \to X$ etc., i.e. any) exists — in such situation the multivalued dependency will degrade to one of two cases shown in figure 3.2.p.

[137] An MVD $X \twoheadrightarrow Y$ in R is called a **trivial MVD** if (a) Y is a subset of X, or (b) $X \cup Y = R$. An MVD that satisfies neither (a) nor (b) is called a **nontrivial MVD**. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)



Multivalued dependency degradation to trivial functional dependency

R

	X	Y	Z
t_1	X_1	Y_1	Z_1
t_2	X_1	Y_2	Z_2
t_3	X_1	Y_1	Z_2
t_4	X_1	Y_2	Z_1

If $X \rightarrow Z$ or $Y \rightarrow Z$ dependency exists, t_3 and t_4 tuples can not exist (otherwise "one argument value defines one function value" rule would be violated).

Figure 3.2.p — Both cases of obtaining a trivial multivalued dependency

So, now you can switch temporarily to the study of the fourth normal form{258}, since we have just considered the dependencies underlying it.

Dependencies on which the fifth normal form is based

The fifth normal form{262} is based on the so-called “join dependency”, which is a generalized case of the multivalued dependency{189} (which, in turn, is a generalization of the functional dependency[138]).



Join dependency[139] — a dependency in the R relation denoted by JD(R_1, R_2, ..., R_n) and producing the following restriction: for any data set, the R relation must allow lossless decomposition{176} into R_1, R_2, ..., R_n projections{172}.

Simplified: the mutual dependency between several (more than two) attributes of a relation, obliging the relation to contain only such data, in which decomposition of the relation into projections from groups of these attributes will be a lossless decomposition (i.e., allowing to recover exactly the original relation).



Trivial join dependency[140] — such a join dependency JD(R_1, R_2, ..., R_n), in which at least one R_i component contains the full set of attributes of the R relation.

Simplified: a join dependency that always holds (due to the fact that a relation can always be exactly reconstructed “from itself” (this effect is achieved if at least one of the projections contains the full set of attributes of the relation, i.e., is, in fact, the original relation itself)).



Nontrivial join dependency[141] — such a join dependency JD(R_1, R_2, ..., R_n), in which no R_i component contains the full set of attributes of the R relation.

Simplified: a join dependency that may be violated, i.e., there may be a situation in which a relation cannot be reconstructed without distortion from its projections onto the groups of attributes in the dependency.

Few are ready to argue that the definition of join dependency sounds rather confusing, so let’s go straight to examples, but let’s add a little informal clarification beforehand: join dependencies often (but not always!) appear where a relation cannot be decomposed into two projections without losses, yet the decomposition into three or more projections is possible.

Let’s start with an almost childish example demonstrating join dependency in a relation in which all non-key attributes{21} functionally depend{176} on the primary key{36} and yet do not depend on each other — such an `elective` relation (describing optional classes) is shown in figure 3.2.q.

[138] This issue is covered in detail in section 15.7 of the “Fundamentals of Database Systems, 6th edition” book (by Ramez Elmasri, Shamkant Navathe).

[139] A **join dependency** (JD), denoted by JD(R_1, R_2, ..., R_n), specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a nonadditive join decomposition into R_1, R_2, …, R_n. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

[140] A join dependency JD(R_1, R_2, ..., R_n), specified on relation schema R, is a **trivial JD** if one of the relation schemas R_i in JD(R_1, R_2, ..., R_n) is equal to R. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

[141] This definition is formulated as an inverse case of trivial join dependency, therefore — see footnote 140.

Initial relation

elective

PK **e_id**	**e_group**	**e_tutor**	**e_topic**
157	1	33	Stop using ORMs!
248	8	24	MySQL vs ...
411	1	78	Oracle 999 overview
489	7	33	Arduino in free space

Functional dependencies:
{e_id} → {e_group}
{e_id} → {e_tutor}
{e_id} → {e_topic}

Join dependency:
JD((e_id, e_group),
(e_id, e_tutor),
(e_id, e_topic))

Projections

P_1 (e_id, e_group)

PK **e_id**	**e_group**
157	1
248	8
411	1
489	7

P_2 (e_id, e_tutor)

PK **e_id**	**e_tutor**
157	33
248	24
411	78
489	33

P_3 (e_id, e_topic)

PK **e_id**	**e_topic**
157	Stop using ORMs!
248	MySQL vs ...
411	Oracle 999 overview
489	Arduino in free space

Figure 3.2.q — The simplest example of join dependency

This example is really trivial, because it is easy to understand that the original relation is guaranteed to be recoverable if each of its projections contains the primary key of the original relation. Nevertheless, even this case is an illustration of the idea of join dependency.

Let's move on to a more complex example. Consider a `workload` relation that shows the workload of faculty members in different departments (see figure 3.2.r). For clarity, let's leave text labels in the relation, although in a real database instead of text there would be numeric values, and the fields themselves would be foreign keys.

Obviously, for such a relation only the three projections presented in figure 3.2.r make sense (the remaining options are three more projections on one attribute each (which makes no sense) and a projection on all the original attributes, i.e., the relation itself).

Initial relation

workload

	PK		
	w_tutor	**w_subject**	**w_faculty**
(1)	Smith	Mathematics	Exact sciences
(2)	Smith	Informatics	Natural sciences
(3)	Taylor	Mathematics	Natural sciences
(4)	Jones	Informatics	Cybernetics
(5)	Taylor	Physics	Exact sciences
(6)	Taylor	Mathematics	Exact sciences
(7)	Smith	Mathematics	Natural sciences

Functional dependencies:
none.

Join dependency:
JD((w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty))

The tuples marked with a background appear only due to the presence of the rule: “If a tutor teaches some **S** subject, and this **S** subject is taught at some **F** faculty, and the tutor teaches at least one other subject at this **F** faculty, then he is obliged to teach that **S** subject at this **F** faculty“.

a) There is “Mathematics” at “Exact sciences” faculty (1).
b) “Taylor” tutor teaches at “Exact sciences” faculty (5).
c) “Taylor” tutor teaches “Mathematics” (3).
So: “Taylor” tutor should teach “Mathematics” at “Exact sciences” faculty (6).

a) There is “Mathematics” at “Natural sciences” faculty (3).
b) “Smith” tutor teaches at “Natural sciences” faculty (2).
c) “Smith” tutor teaches “Mathematics” (1).
So: “Smith” tutor should teach “Mathematics” at “Natural sciences” faculty (7).

Projections

P_1 (w_tutor, w_subject)

PK	
w_tutor	**w_subject**
Smith	Mathematics
Smith	Informatics
Taylor	Mathematics
Jones	Informatics
Taylor	Physics

P_2 (w_subject, w_faculty)

PK	
w_subject	**w_faculty**
Mathematics	Exact sciences
Informatics	Natural sciences
Mathematics	Natural sciences
Informatics	Cybernetics
Physics	Exact sciences

P_3 (w_tutor, w_faculty)

PK	
w_tutor	**w_faculty**
Smith	Exact sciences
Smith	Natural sciences
Taylor	Natural sciences
Jones	Cybernetics
Taylor	Exact sciences

No two of these projections are enough to get the original relation (extra tuples that weren’t there before appear). While it is easy to obtain the original relation with all three projections.

Figure 3.2.r — Example of join dependency

As noted in figure 3.2.r, the described join dependency is relevant because of the subject area rule (constraint): “If a tutor teaches some **S** subject, and this **S** subject is taught at some **F** faculty, and the tutor teaches at least one other subject at this **F** faculty, then he is obliged to teach that **S** subject at this **F** faculty”.

You can see for yourself (using any database that supports `NATURAL JOIN` operation) that combining any two of the projections shown in figure 3.2.r generates “extra data”, and only combining all three projections allows us to get exactly the original relation.

Just in case, here is the SQL code for that operation (in MySQL syntax).

MySQL — Reconstructing the original relation using three projections

```
1    SELECT *
2    FROM   (SELECT *
3            FROM   `p1`
4                   NATURAL JOIN `p2`) AS `p12`
5           NATURAL JOIN `p3`
```

But what happens if we “abandon” this complicated and confusing rule about tutors, subjects, faculties, and their interrelationships? We get the result shown in figure 3.2.s.

Note that the projections P_1, P_2, P_3 in both figures (3.2.r and 3.2.s) are exactly the same, while the relations shown in these figures are different. Just above is the SQL code, the execution of which clearly demonstrates that combining the obtained projections results in the relation shown in figure 3.2.r, i.e., the relation shown in figure 3.2.s is not obtained.

Thus, the elimination of the rule producing join dependency has resulted in a relation that cannot be losslessly decomposed.

Initial relation

workload

PK		
w_tutor	**w_subject**	**w_faculty**
Smith	Mathematics	Exact sciences
Smith	Informatics	Natural sciences
Taylor	Mathematics	Natural sciences
Jones	Informatics	Cybernetics
Taylor	Physics	Exact sciences

Functional dependencies: none.

Jion dependencies: none.

Projections

P_1 (w_tutor, w_subject)

PK	
w_tutor	**w_subject**
Smith	Mathematics
Smith	Informatics
Taylor	Mathematics
Jones	Informatics
Taylor	Physics

P_2 (w_subject, w_faculty)

PK	
w_subject	**w_faculty**
Mathematics	Exact sciences
Informatics	Natural sciences
Mathematics	Natural sciences
Informatics	Cybernetics
Physics	Exact sciences

P_3 (w_tutor, w_faculty)

PK	
w_tutor	**w_faculty**
Smith	Exact sciences
Smith	Natural sciences
Taylor	Natural sciences
Jones	Cybernetics
Taylor	Exact sciences

No combinations of these projections allow us to get the original relation.

Figure 3.2.s — Example of join dependency absence

So, now you can switch temporarily to the study of the fifth normal form{262}, since we have just considered the dependencies underlying it, and even almost formulated its definition.

Dependencies on which the domain-key normal form is based

The dependencies presented below are in no way related to those discussed earlier. They were specifically formulated by Ronald Fagin (the author of the domain-key normal form) and are in general much easier to understand than many “classical” dependencies.



Domain dependency[142] — a dependency with respect to R relation, denoted by IN(A, S) and producing the following restriction: any value of the A attribute must be from the S set.

Simplified: let’s imagine the “day of week” set (MON, TUE, WED, THU, FRI, SAT, SUN) and an attribute of some R relation, named `day_of_week` *and referring to the domain “day of week”; the restriction of the resulting domain dependency is fulfilled only if there are no values other than MON, TUE, WED, THU, FRI, SAT, SUN in any table record in the* `day_of_week` *field.*

Even more simplified: the value of any table field must match the list (or range) of allowed values.



Key dependency[143] — dependency with respect to R relation, denoted by KEY(K) and producing the following restriction: no two tuples of the R relation can have the same values of the K set.

Simplified: if some set of K table fields is a key{32}, then in each row of the table the set of attribute values included in K is unique.

Even more simplified: the key{32} values of a table should not be duplicated.

Let’s start our explanation with the key dependency, because it is quite self-evident: indeed, any modern DBMS, by definition, guarantees the uniqueness of primary keys{36} and unique indexes{106}. However, Fagin emphasizes[144] that his definition is more about a superkey{33}, i.e., we are not interested in the key minimality property here.

This clarification might seem unimportant if it did not shift our focus from a specific implementation of the relation schema in the form of a DBMS table to the relation schema itself, reflecting the features of the subject area.

With this remark in mind, an “even more simplified” formulation of the key dependency can be formulated as follows: if according to the rules of the subject area, a set of values of some fields of a relation must be unique for any record, then the key dependency is present in the relation if and only if this restriction is fulfilled (regardless of whether a primary key, a unique index, or any other technical means of implementing this restriction are used).

The graphical explanation of the key dependency (see figure 3.2.t) looks no less trivial. If we assume that the `e_passport` attribute is the key of the `employee` relation, the key dependency is present in the first case (no duplicate values in this field) and is absent in the second case (duplicate values are in this field).

[142] The **domain dependency** IN(A,S), where A is an attribute and S is a set, means that the A entry in each tuple must be a member of the set S. For example, let A be the attribute SALARY, and let S be the set of all integers between 10'000 and 100'000. If A is one of the attributes of relation R, then R obeys IN(A, S) if and only if the SALARY entry of every tuple of R is an integer between 10'000 and 100'000. (“A Normal Form for Relational Databases That Is Based on Domains and Keys”, Ronald Fagin)

[143] The **key dependency** KEY(K), where K is a set of attributes, says that K is a key, that is, that no two tuples have the same K entries. (“A Normal Form for Relational Databases That Is Based on Domains and Keys”, Ronald Fagin)

[144] We remark that keys are usually defined at the schema level, not at the instance level. At the schema level, for K to be a key, it is usually assumed that K is minimal, that is, that no proper subset of K is a key. However, this minimality assumption is not useful at the instance level. Thus, we wish a relation in which a proper subset of K happens to uniquely identify tuples to be a possible instance of a schema in which K is a key. So, our definition, in which minimality is not assumed, is more convenient for our purposes. (“A Normal Form for Relational Databases That Is Based on Domains and Keys”, Ronald Fagin)

There is the key dependency

employee

...	e_passport	e_salary	...
...	AA123456	10 000	...
...	BB654321	12 000	...
...	CC112233	11 000	...
...	DD332211	10 500	...

There is NO key dependency

employee

...	e_passport	e_salary	...
...	**AA123456**	10 000	...
...	**AA123456**	10 000	...
...	CC112233	11 000	...
...	DD332211	10 500	...

Figure 3.2.t — Graphic explanation of the key dependency

Speaking of domain dependency, it is worth clarifying again that our focus should not be on a particular implementation (most DBMS provide very flexible mechanisms for limiting ranges and sets of field values), but on the requirements of the subject area.

Thus, this dependency is present in the relation (see figure 3.2.u) if and only if all values of some attribute of the relation are representatives of the corresponding domain{20} (assume, that in the `event` relation the `e_day_of_week` attribute can take one of seven values: MON, TUE, WED, THU, FRI, SAT, SUN).

There is the domain dependency

event

...	e_date	e_day_of week	...
...	2018-05-01	TUE	...
...	2018-05-01	TUE	...
...	2018-05-02	WED	...
...	2018-05-02	WED	...

There is NO domain dependency

event

...	e_date	e_day_of week	...
...	2018-05-01	ABC	...
...	2018-05-01	TUE	...
...	2018-05-02	WED	...
...	2018-05-02	WED	...

Figure 3.2.u — Graphic explanation of the domain dependency

So, now you can temporarily switch to the study of the domain-key normal form{267}.

Dependencies on which the sixth normal form is based

The join dependency considered earlier{192} is relevant for databases that do not contain the so-called “temporal data[145]” (i.e., information about moments and intervals of time and associated values of some attributes of the relation).

If the database is designed to store and process such data, generalized relational operations[146] and generalized dependencies become relevant for it, in particular — generalized join dependencies.

!!! **Generalized join dependency** (U_join dependency[147]) — a dependency in R relation (which H header contains a set of ACL attributes of interval type), denoted by USING (ACL): ¤ {X_1, X_2, ..., X_n} and producing the following constraint: the union (according to set theory) of X_1, X_2, ..., X_n projections must produce a relation with the same H header.

Simplified: the mutual dependency between several (more than two) attributes of a relation (with interval attributes), obliging the relation to contain only such data, in which decom-position of the relation into projections from groups of these attributes will be a lossless decomposition (i.e., allowing to recover exactly the original relation).

In fact, in a simplified formulation of this definition, only the specification “with interval attributes” was added; otherwise, it is equivalent to the simplified formulation of the “usual” join dependency{192} definition.

Technical note: in his book[148] C. J. Date explains why this dependency is tied to the header of the relation and not to the relation itself, but to simplify our material a bit, we will leave this question to your own study of the primary source.

By analogy with the corresponding forms of “usual” join dependency, let’s formulate definitions of trivial and nontrivial generalized join dependency:

!!! **Trivial generalized join dependency** (trivial U_join dependency[149]) — a join dependency USING(ACL): ¤ {X_1, X_2, ..., X_n}, in which at least one X_i component contains the full set of attributes of R relation (i.e., its entire H header).

Simplified: a generalized join dependency that always holds (due to the fact that a relation can always be exactly reconstructed “from itself” (this effect is achieved if at least one of the projections contains the full set of attributes of the relation, i.e., is, in fact, the original relation itself)).

[145] See chapter 23 (“Temporal Databases”) in the “An Introduction to Database Systems (8th edition)” book (by C.J. Date), where the author devotes about 50 pages to this subject with very detailed descriptions and examples.

[146] Once again see chapter 23 (“Temporal Databases”) in the “An Introduction to Database Systems (8th edition)” (by C.J. Date). Since temporal databases are partially outside the scope of our topic, we will limit ourselves to a brief review of the key definitions.

[147] Let H be a heading, and let ACL be a commalist of attribute names in which every attribute mentioned (a) is one of the attributes in H and (b) is of some interval type. Then a U_join dependency (U_JD) with respect to ACL and H is an expression of the form USING (ACL) : ¤ {X_1,X_2,...,X_n}, such that the set theory union of X_1, X_2, ..., X_n is equal to H. (“The New Relational Database Dictionary”, C.J. Date)

[148] “The New Relational Database Dictionary”, C.J. Date, page 423.

[149] This definition is formulated by analogy with the definition of the trivial join dependency{192}.



Nontrivial generalized join dependency (nontrivial U_join dependency[150]) — a join dependency (ACL): ¤ {X_1, X_2, ..., X_n}, in which no X_i component contains the complete set of attributes of the R relation.

Simplified: a generalized join dependency that may be violated, i.e., there may be a situation in which a relation cannot be reconstructed without distortion from its projections onto the groups of interval attributes in the dependency.

And before we look at the example, here are two more definitions relevant to temporal databases.



Important: outside the context of temporal databases, the following two terms may have a different meaning.



Horizontal decomposition[151] — a decomposition of a relation containing temporal data into relations containing explicit intervals ("from… to…") and relations containing implicit intervals ("from… to the current moment", "from the current moment to…").

Simplified: information about intervals of the "from… to…" kind (i.e., having an initial and a final moment in time) is placed in one relation, and information about intervals of the "from… to the current moment" and "from the current moment to…" kind (i.e., having only one fixed moment in time — initial or final) is placed in other relations.



Vertical decomposition[152] — a decomposition of a temporal interval relation which is not in the 6NF{271} into a set of temporal interval relations which are in the 6NF{271}.

Simplified: the original set of attributes of a temporal relation that is not in the 6NF{271} is processed so as to obtain new sets of attributes (possibly not coinciding with the original ones) of new temporal relations that are in the 6NF{271}. The situation presented in figure 3.2.v is a clear example of vertical decomposition.

Now let's turn to the example (see figure 3.2.v). In the original `education_path` relation there is a nontrivial generalized join dependency, which is clearly shown by creating two P_1 and P_2 projections of this relation, which allows us to obtain exactly the original relation using the generalized join operation (`U_join`[153]).

[150] This definition is formulated by analogy with the definition of the nontrivial join dependency{192}.

[151] **Horizontal decomposition** — informal term used to refer to the decomposition of a temporal relvar into a combination of *since* relvars and *during* relvars. ("The New Relational Database Dictionary", C.J. Date)

[152] **Vertical decomposition** — informal term used to refer to the decomposition (via U_projection) of a *during* relvar that's not in sixth normal form into a set of *during* relvars that are. ("The New Relational Database Dictionary", C.J. Date)

[153] See section 23.5 ("Generalizing the relational operators") in the "An Introduction to Database Systems (8th edition)" book (by C.J. Date).

Note: unlike with “usual” databases and related decomposition operations, here the values of `ep_period` attribute in P_1 and P_2 projections are obtained not by copying values from the original relation, but by computing new values (in this case — based on summation of time intervals).

Initial relation

education_path

PK			
ep_student	**ep_period**	**ep_faculty**	**ep_group**
13452	01.01.2018-28.02.2018	Chemistry	1
13452	01.03.2018-31.05.2018	Chemistry	5
13452	01.06.2018-01.12.2018	Physics	5

Projections

P_1 (ep_student, ep_period, ep_faculty)

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Chemistry
13452	01.06.2018-01.12.2018	Physics

P_2 (ep_student, ep_period, ep_group)

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-28.02.2018	1
13452	01.03.2018-01.12.2018	5

Figure 3.2.v — Example of a generalized join dependency

The P_1 and P_2 projections cannot be vertically decomposed further (one of the resulting projections will coincide exactly with the original relation), so it turns out that in the P_1 and P_2 projections there is a trivial generalized join dependency (or, to remember more easily: there is no nontrivial generalized join dependency).

Finally, let's consider a small example that explains the essence of horizontal decomposition (this term has not previously been used in the discussion of generalized join dependency, but it is quite important).

Initial relation

education_path

PK			
ep_student	**ep_period**	**ep_faculty**	**ep_group**
13452	01.01.2018-28.02.2018	Chemistry	1
13452	01.03.2018-31.05.2018	Chemistry	5
13452	01.06.2018-01.12.2018	Physics	5

New relations after horizontal decomposition

P_1 (ep_student, ep_period, ep_faculty)

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Chemistry
13452	01.06.2018-01.12.2018	Physics

P_2 (ep_student, ep_period, ep_group)

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-28.02.2018	1
13452	01.03.2018-01.12.2018	5

Figure 3.2.w — Example of horizontal decomposition

Let's imagine that it is necessary to add to the P_1 and P_2 relations the information that from December 02, 2018 to the present moment the student with identifier 13452 is studying at the faculty of Biology in group 10. Figure 3.2.w shows both situations at once, before and after the horizontal decomposition.

Why should intervals of the form "from… to…" and the form "from… to the current moment" not be stored in the same relations? C.J. Date gives rather detailed explanation[154], but in brief: there is a huge set of questions about even the simplest operations (comparison, addition, subtraction) for "now" concept; for cases where "now" is the beginning of an interval, there is a need for additional data management operations (the simplest example — sooner or later "now" becomes larger than "end of interval"). In short, the benefit is minimal, but the potential problems are plentiful.

Yes, in any modern DBMS it is possible to get the current date and time value and compare it to any datetime value. Yes, many DBMSes allow us to automatically update datetime value when a table record is accessed. Yes, we can use "now" value when constructing almost any SQL query. All this is true, and yet it does not cancel the problems outlined by C.J. Date (most of which have only limited solutions, and some not solved at all).

The conclusion is simple: keep the information about closed intervals (like "from… to…") and open intervals (like "from… to the current moment" and "from the current moment to…") in different relations.

So, now you can temporarily switch to studying the sixth normal form{271}.

A final overview of all main dependencies

We have just considered only the "most necessary" dependencies, the study of which is crucial for understanding normal forms.



In the context of relational theory, it is common to consider many more dependencies (and their properties) used in formal proofs of normalization algorithms. If this material is of interest to you, please refer to the following literature:

- "Fundamentals of Database Systems, 6th edition" (by Ramez Elmasri, Shamkant Navathe) — chapters 15–16.
- "An Introduction to Database Systems, 8th edition" (C.J. Date) — chapters 11–13.

In order to consolidate the material, we will once again give a list of the dependencies just discussed and present their definitions in an even more concise and simplified form than was done earlier.

A similar short list of all normal forms will be presented below{274}.



Please use the definitions below only as a way of quickly remembering the nature of a particular dependency. Full, rigorous definitions can be found quickly by following the link provided next to the original version of each term.

[154] See chapter 23.6 ("Database design", subsection "The moving point *now*") in the "An Introduction to Database Systems (8th edition)" book (by C.J. Date).

Functional dependency{176} — a dependency in which one value of the determinant defines one value of the dependent part.

Full functional dependency{179} — a dependency, in which the value of the dependent part depends on all parts of the determinant.

Partial functional dependency{179} — a dependency in which the value of the dependent part depends only on some parts of the determinant.

Trivial functional dependency{187} — a functional dependency that cannot be violated (due to the fact that the dependent part is a part of the determinant).

Nontrivial functional dependency{187} — a functional dependency that can be violated (due to the fact that the dependent part is **not** a part of the determinant).

Transitive dependency{183} — a “chain” of dependencies (two or more functional dependencies in which the dependent part of the previous dependency is the determinant of the next dependency).

Redundant dependency{183} — a dependency that can be calculated from other dependencies.

Redundant transitive dependency{183} — a “chain” of dependencies in which there is a functional dependency between the determinant of the first dependency and the dependent part of the last dependency.

Pseudotransitive dependency{183} — a “chain” of dependencies in which the determinant of the last dependency can be calculated from the dependent parts of several previous dependencies.

Multivalued dependency{189} — a dependency, which requires a relation containing two tuples matching one of the three attributes to also contain two additional tuples “crosswise” containing combinations of the remaining two attributes.

Trivial multivalued dependency{190} — a multivalued dependency in which for a group of three X, Y, Z attributes either Y is a part of X or Z is absent.

Nontrivial multivalued dependency{190} — a multivalued dependency in which for a group of three X, Y, Z attributes attribute Y is not a part of X, and attribute Z is present.

Join dependency{192} — a dependency, which requires the relation to explicitly contain all the tuples that will be obtained after reconstructing the relation from its projections onto the groups of attributes that are part of this dependency.

Trivial join dependency{192} — a join dependency in which at least one set of attributes that make up the set of projections on which the given dependency is built is a complete set of attributes of the relation.

Nontrivial join dependency{192} — a join dependency in which none of the sets of attributes that make up the set of projections on which this dependency is built represents the complete set of attributes of the relation.

Domain dependency{197} — a dependency, which binds all values of any attribute of a relation to belong to the set of allowed values for the given attribute.

Key dependency{197} — a dependency that binds all values of the relationship key to be unique.

Generalized join dependency{200} — a dependency that requires a temporal relation to explicitly contain all attributes and tuples that will be obtained after reconstructing the relation from its projections to groups of attributes that are part of this dependency and that contain at least one interval attribute.

Trivial generalized join dependency (trivial U_join dependency{200}) — a join dependency in which at least one set of attributes that make up the set of projections on which the given dependency is built is the complete set of attributes of the initial temporal relation.

Nontrivial generalized join dependency (nontrivial U_join dependency{201}) — a join dependency in which **none** of the sets of attributes that make up the set of projections on which this dependency is built represents the complete set of attributes of the initial temporal relation.

So, we are almost ready to consider normal forms. All that remains is to give some important considerations about the process of their application, and this will be the focus of the next chapter.



Task 3.2.a: write down (in any notation you like) all functional dependencies present in the “Bank{395}” database.



Task 3.2.b: are there relations in the “Bank{395}” database that contain multivalued dependencies? If you think “yes,” refine the schema to eliminate such dependencies.



Task 3.2.c: make a list of dependencies discussed in this chapter, but absent from the “Bank{395}” database.

3.2.2. Normalization Requirements in the Database Design Context

Let's start with the main definition.



Normalization[155] — a process of decomposing relation variable R into a set of projections R_1, R_2, ..., R_n such that:
a) the join of R_1, R_2, ..., R_n projections allows to obtain the initial relation variable R;
b) each of R_1, R_2, ..., R_n projections is necessary to fulfill the condition "a";
c) at least one of R_1, R_2, ..., R_n projections is in a higher normal form than the original relation variable R.

Simplified: a relation variable is split into several new ones that are in higher normal forms and allow us to get the original relation variable by using `JOIN` *operation.*

Before we start talking about the normalization process, it's worth considering what to strive for, what to focus on, and what to avoid when performing normalization — this can all be summarized in a single list of normalization requirements[156], which are not a strict and exhaustive list, but still provide enough guidance.

The normalization process is an integral part of database design, and therefore allows us to identify and correct many flaws in the designed schema — both directly related to violations of normal forms, and many other, but no less dangerous.

Since normalization is not possible without the redesign of the database schema, the corresponding actions will necessarily affect the design documentation, and therefore part of the following requirements refers to the process of documenting the design.

Also note that the following requirements in some cases may contradict each other, and this is perfectly normal — there is no goal to formally achieve compliance with all of them, there is only a need to consider those requirements that best meet the key quality indicators of the database being designed.

First, however, let's consider the basic idea expressed by several normalization principles[157]:

- A relation variable that is not in the essential tuple normal form (ETNF{274}) must be decomposed into a set of projections that are in ETNF or even higher normal forms.
- The original relation variable must be reconstructable through the union of the resulting projections.
- The decomposition of the original relation variable into projections must preserve the dependencies that existed in the original relation variable.
- In the decomposition process, each resulting projection must be necessary to reconstruct the original relation variable.

---

[155] **Normalization** is a replacing a relvar R by certain of its projections R_1, R_2, ..., R_n, such that (a) the join of R_1, R_2, ..., R_n is guaranteed to be equal to R, and usually also such that (b) each of R_1, R_2, ..., R_n is needed in order to provide that guarantee (i.e., none of those projections is redundant in the join), and usually also such that (c) at least one of R_1, R_2, ..., R_n is at a higher level of normalization than R is. The usual objective of normalization is to reduce redundancy and thereby to eliminate certain update anomalies that might otherwise occur. ("The New Relational Database Dictionary", C.J. Date)

[156] In the traditional primary sources, these requirements are presented in fragments in the early works of Codd and Date, and in the form of a single short list first formulated in the "Fundamentals of Relational Database Design" course (in Russian) [http://www.intuit.ru/studies/courses/1095/191/info], after which they were repeatedly copied into many national-language materials. The problem is that a short enumeration is not enough for understanding by novice database developers; that is why such explanations are given here, and the list itself has been refined.

[157] **Normalization principles** is a set of principles used to guide the practical process of normalization. The principles in question are as follows: (a) A relvar not in ETNF should be decomposed into a set of projections that are in ETNF (and possibly some higher normal form, such as 5NF or 6NF); (b) the original relvar should be reconstructable by joining those projections back together again; (c) the decomposition process should preserve dependencies; (d) every projection should be needed in the reconstruction process. ("The New Relational Database Dictionary", C.J. Date)



Although this list refers to the so called “non-canonical” normal form of essential tuples, from the scientific point of view everything is correct here. From the practical point of view, however, very rarely a relation variable that is in 4NF{258} will not be in ETNF{274}, so in the normalization process we focus exactly on 4NF{258} (and sometimes on 3NF{247} or BCNF{253}, if this degree of normalization is sufficient to eliminate all the data operation anomalies{157}).

And now let’s move on to more extensive, but no less important requirements, which should be considered when performing normalization.

Maintaining subject area constraints

In the process of normalization, special attention should be paid to the results of decomposition of the original relation variable into projections{172}. Such results should:

- meet the lossless decomposition{176} requirements;
- save explicitly or allow to restore all pre-existing dependencies and constraints when performing JOIN operations.

Violation of this requirement is highly undesirable, because it will require the creation of additional (usually — very nontrivial) mechanisms to ensure the database consistency{67}.

Compliance with the general requirements for a database schema

All the general universal database requirements{10} were discussed at the very beginning of this book. To recap them briefly: adequacy to the subject area, usability, performance, data safety. These requirements are so fundamental that they should be considered at any level of database design and any manipulation of the database schema, because if any part of these requirements is violated, we risk having a database that is not suitable for further use.

Primary keys minimality

It would seem that so much has already been said about primary keys{36} minimality{36} that we don’t need to repeat it. Alas, in practice there are problems with this requirement.

In general, we can ask ourselves three questions about a primary key of some table:

- Does the table need a primary key at all?
- Is the key small enough?
- Is the key big enough?

Does the table need a primary key at all? Intuitively, it seems obvious that a primary key is always needed, because otherwise we cannot reliably distinguish table rows, we cannot make relationships, and we get a lot of other negative consequences. This is true, but not always. There are exceptions, when the presence of a primary key is not only unnecessary, but on the contrary — it is drastically wrong. An example of such a case will be discussed very soon{211}. For now, let’s agree that a primary key is most often necessary, and the only question is “how should it look like”.

Is the chosen primary key small enough? First of all, it is worth choosing between natural{38} and surrogate{38} primary keys (which often entails choosing between composite{36} and simple{36} primary keys).

Without diving into long philosophical arguments, we agree that the composite natural primary key also has the right to exist, but in the vast majority of cases, a fair choice would be in favor of a simple surrogate one, where we only have to choose the most appropriate data type to ensure the key minimality.

If for some reason the integer type does not suit us, it is worthwhile learning how the particular DBMS stores and handles the data type we choose. Sometimes we can get very unpleasant surprises in the form of "aligning" strings in UTF8 encoding on the boundary of "four bytes per character", augmenting the stored data with preset values (usually spaces or zeros) to a certain fixed length, etc. Only a thoughtful study of the documentation will help here.

But let's imagine a good situation: the integer type suits us, and we have chosen even one of the "shortest" types — for simplicity of illustration let's assume that we work with MySQL and we have chosen the `TINYINT` (actually a byte) data type.

Is the chosen primary key big enough? This is the easiest question of the three, but for some reason it is the one that most people rarely think about. To get the right answer, we need to understand the amount records we'll store in the table, and how that amount will change over time.

In our example we have chosen a data type (`TINYINT`) that can store 256 values. This will be perfectly sufficient to assign unique identifiers to all records if there are few of them, and they are not added over time (or added only in exceptional cases). E.g., such a primary key should be enough for tables with the list of countries, user statuses, payment types, etc.

But this data type is definitely not enough to store a list of employees, a list of articles, a list of bank transactions, etc. The situation is exacerbated by the fact that such a primary key is often made auto-incrementable, so even removing obsolete records from the table does not release some of the previously used values of such a primary key[158] — it gradually and inevitably approaches the maximum limit, at which the insertion of new records will not be possible.

So, here's a universal advice. A couple of bytes gained per record will save you a fraction of a percent of the size of a typical database, and may increase performance by an insubstantial amount, but it will potentially lead to a crash at the worst possible moment. This is by no means a call to make primary keys huge, just leave some reasonable margin.

Data nonredundancy requirement

The modification{157} and deletion{157} data operations anomalies discussed earlier flourish when the same data is stored in multiple rows of the same table. The situation is very similar when the same data is stored in different tables.

Let's imagine that in some database we have the following situation (see figure 3.2.x): two relations (`contract` and `finance`) contain information about contracts and their corresponding amounts. And we see that for one contract the amounts do not coincide. Provided that according to the subject area requirements it should be the same value — which data to believe? "We are saved" if there are additional sources of information containing "the truth" (assuming that we noticed the error, but we might not have noticed), but what if there are no such sources? It's a dead end: some of the data is definitely wrong, and we can't figure out which one.

Such a situation would be avoided if the amount was kept strictly in one place.

[158] In fact, there is a solution, but it is not trivial at all. See example 38 in the "Using MySQL, MS SQL Server and Oracle by examples" book (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]



Sometimes one may hear the objection that by storing data in several places we increase the chances of not losing it in case of failures and storage damage. This is fundamentally wrong: the task of creating backups, mirroring and other data safety measures should never be confused with the task of designing a database. You can't use tools to solve one of these tasks to solve the other.

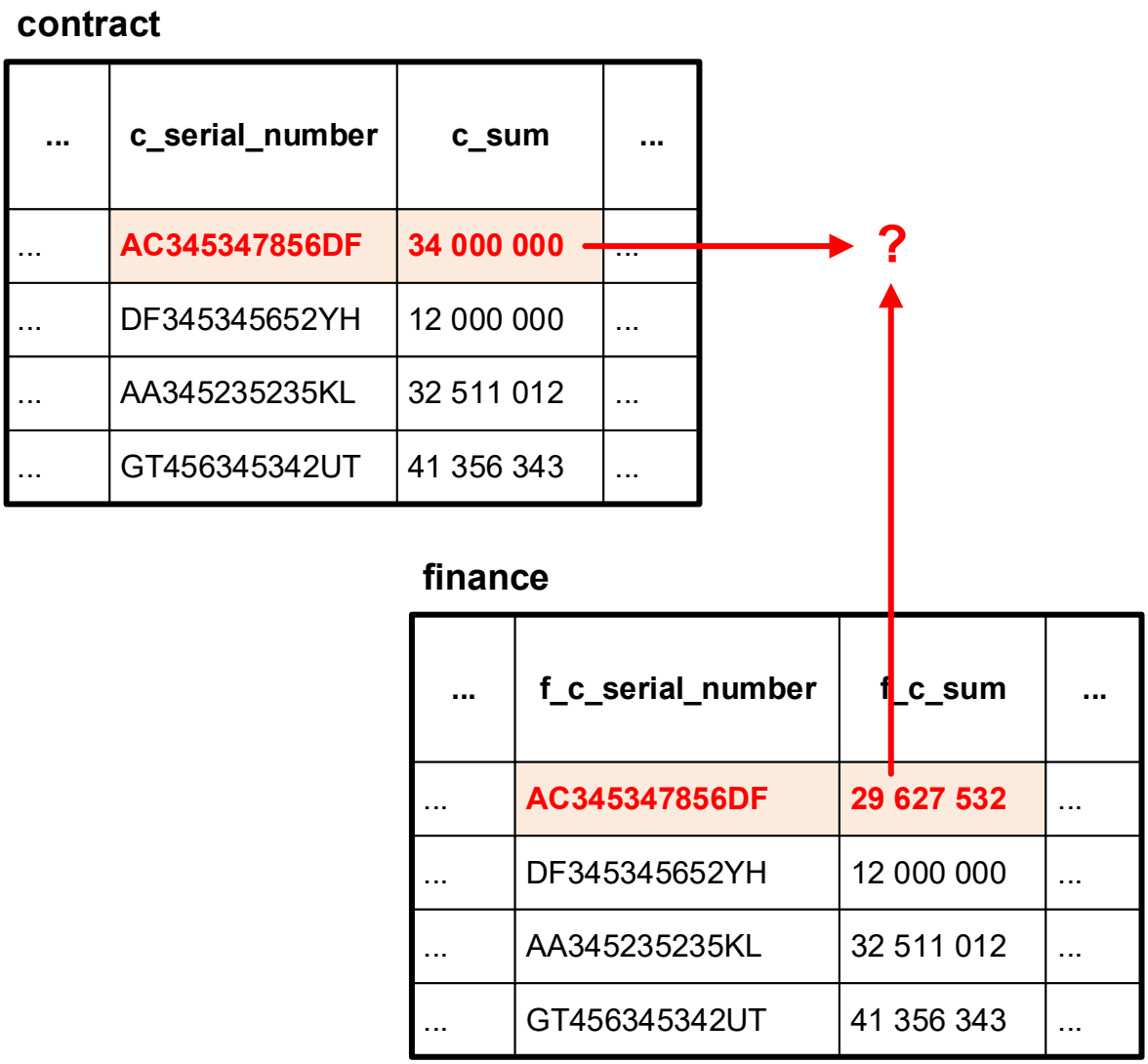


Figure 3.2.x — Data nonredundancy requirement violation

System performance requirement

We've discussed database performance earlier{14} a little bit, and now we will approach this question from a different perspective. What characteristics of a database schema that affect performance can we easily manage at the design stage? Let's emphasize — *at the design stage*, i.e., here we do not yet touch the DBMS settings, hardware parameters, network infrastructure, etc.

We can think about primary keys minimality{208} (and indexes minimality, because in this context everything said about keys also applies to indexes), evaluate the necessity of creating certain indexes, think about the target schema normalization depth[159] and the possibility of using denormalization{228} as a solution for performance optimization.

[159] This is not a rigorous term, but in general it means the maximum normal form to which we are going to normalize each individual relation variable.

Let's consider a simple example of optimization in the context of performance of a database schema consisting of three tables (see figure 3.2.y) — let's assume that we have a database of some news site.

Suppose that based on similar products study the following was found:

- the **`news`** table is accessed on average 200–300 thousand times per day for reading, 5–10 times for writing, and 2–3 times for modification (or deletion);
- the **`log`** table is accessed 500–700 thousand times a day for data insertion, never accessed for modification (or deletion), and read requests are performed once or twice a week, but we need to get their results very quickly.

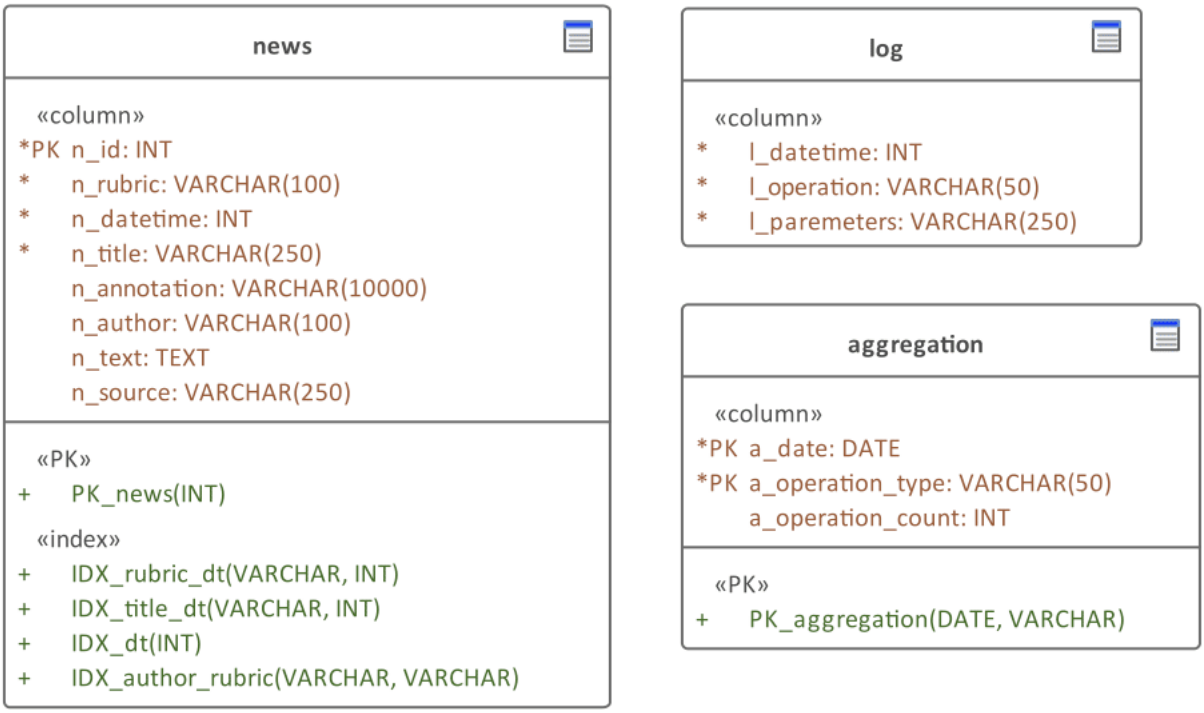


Figure 3.2.y — Ensuring maximum database performance at the design stage

The **`news`** table is a good example of the "read optimization" case, i.e., we can initially create several indexes without worrying that their presence will slow down the execution of data modification operations.

The **`log`** table is a good example of the "write optimization" case, so we completely remove any indexes from it, including the primary key (this very case is the previously{208} promised example of a situation where a table does not need a primary key). As a result, we have sacrificed reading speed and greatly benefited in writing speed.

But we are left with one unsolved problem: it is said that reading data from the **`log`** table must be very fast, and we have just abandoned everything on fast reading from this table. But there is still a solution: we need to figure out what read operations will be performed and how up-to-date the data will be, and then prepare the results in advance.

Suppose we found out that the customer needs to know how many times a day a particular operation was performed on the site, and the relevance of the information should be "within a day". Then we'll create the **`aggregation`** table, which will contain the results of corresponding queries, and the queries themselves will be executed once a day at the least busy time for the site (e.g., at night).

Yes, we have optimized each particular table to perform only one type of operation (either reading or modifying data), but in general, this schema responds as quickly as possible to all the most typical data operations, which is what we wanted to achieve.

It is easy to see that the **`news`** table in figure 3.2.y is a subject to a number of data operation anomalies{157} that can (and should!) be prevented by normalization.

Data consistency requirement

This requirement is both one of the simplest and one of the most difficult.

The simplicity here is in the fact that, actually, we are talking about data consistency{67} and that the database schema should not create prerequisites for its violation.

The difficulty, on the other hand, is divided into two parts.

First, it is difficult to comply with all consistency constraints at once — there are a lot of them, and it is highly probable that some of them will not be known at all (due to mistakes on the subject area analysis stage), some will be forgotten to implement using the corresponding DB/DBMS mechanisms, and some will not be implemented correctly (or left outdated due to changes in database schema or subject area requirements). More details about how this problem is solved are described in the corresponding section{278}.

Second, there is a great temptation to move the implementation of some restrictions to the level of applications working with the database, or refuse to implement some restrictions at all, based on the assumption that “everything will work fine as it is”.



It is largely for this reason that this fundamental requirement for any relational database is repeated many times in a variety of contexts: the vast majority of catastrophic database problems occur precisely because someone once considered this or that fatal situation to be unlikely or not possible at all. But it did occur causing irreparable damage. The conclusion is very simple: if you understand that some constraint is necessary, you should implement it within the DB/DBMS, without hoping that someone somewhere will do it for you.

And another manifestation of the complexity under consideration is worthy of special mention: the full implementation of all necessary constraints may conflict with the requirements of database performance{210} and database structure flexibility{212}.

Unfortunately, there is no general solution.

If sufficient resources are available, it is certainly worth giving preference to the consistency, and the issues of database performance and database structure flexibility should be solved, respectively, by adding hardware resources and careful documentation (also keeping in mind that in the future redesigning such a schema will require more effort).

In practice, however, you have to look for compromises, many of which may look very questionable from the standpoint of database theory, but still effectively solve the business problem.

Flexibility of the database structure requirement

When designing a database, it should be understood that the created schema is not finite, that it will evolve as the project develops, and that changes can occur both today and over a long period of time.

This requirement of normalization calls to be mindful of the coming changes and prepare for them now.

We have just seen{212} that this requirement has a “natural enemy” in the form of a huge variety of constraints objectively necessary to ensure data consistency{67}, and that this problem requires particular solutions in the context of the business problem at hand.

But there are also a number of other problems that can make the evolutionary development of a database a living hell. These problems are trivial, have simple and straightforward solutions, but, unfortunately, are common in the works of novice database specialists. They all boil down to a violation of one or more of the following rules.

The names of all structures must be mnemonic and follow the same naming convention. We have already partially touched on this rule{25} in the section on relations. In a nutshell: use names with coherent form and with meanings that make it clear what the named entity is and what its purpose is.

We have just stressed{212} the difficulty of thinking through all the necessary consistency constraints, yet with the problem of database structures naming, this task becomes almost impossible — instead of thinking through some complex rule, the engineer spends all their energy trying to simply understand what their predecessor (or even themselves) had in mind.

Compare:

	Bad	Good
Tables' names	data	registered_user
	connection	m2m_user_role
	t1	news_section
Fields' names	pp	u_primary_phone
	ok	o_is_order_confirmed
	some_text_data	a_biography
Stored functions' names	process	get_initials
	get_date	unixtime_to_datetime
	fnc	upcase_first_letters
Triggers' names	no_admins	t_user_protect_last_admin_del
	update_all	t_aggregate_user_stats_upd
	bad_dates	t_date_start_le_end_ins_upd
Indexes' names	qsrch	idx_section_author_date
	singlerecord	unq_login
	tr	idx_author_btree

In fact, most naming rules from classical programming can be applied to database structure names. Yes, in some DBMS there are quite strict restrictions that do not allow to create a very long and detailed name, but even in this situation it is possible to preserve the main essence by replacing only the universal and/or not the most important parts of the name with abbreviations.

Separately, we would like to say a few words about using prefixes in names: whether to name indexes with `idx` prefix (and unique indexes with `unq` prefix), whether to begin table field names with an abbreviation from the table name itself, etc. This is a very "religious" question, in which each of the answers has many supporters and opponents.

The choice is up to you. The main thing is to maintain consistency in the choices you make, i.e., for example, if you decide to start table field names with an abbreviation from the table name, this rule must be followed *for all fields in all tables*. Without exception.

The database schema should be provided with comments. This rule is also fully consistent with the programming classics. Any modern design tool allows you to write comments on fields, tables, code of triggers and stored routines, etc. And such comments are stored in the database itself and will be available even if the accompanying documentation is not available for some reason.

Compare the following two fragments of code (in MySQL syntax):

MySQL — Code without comments

```mysql
CREATE TABLE `file`
(
 `f_uid`              BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
 `f_fc_uid`           BIGINT UNSIGNED NULL,
 `f_size`             BIGINT UNSIGNED NOT NULL,
 `f_upload_datetime`  INTEGER NOT NULL,
 `f_save_datetime`    INTEGER NOT NULL,
 `f_src_name`         VARCHAR(255) NOT NULL,
 `f_src_ext`          VARCHAR(255) NULL,
 `f_name`             CHAR(200) NOT NULL,
 `f_sha1_checksum`    CHAR(40) NOT NULL,
 `f_ar_uid`           BIGINT UNSIGNED NOT NULL,
 `f_downloaded`       BIGINT UNSIGNED NULL DEFAULT 0,
 `f_al_uid`           BIGINT UNSIGNED NULL,
 `f_del_link_hash`    CHAR(200) NOT NULL
)
-- ...
```

MySQL — Code with comments

```mysql
CREATE TABLE `file`
(
 `f_uid`              BIGINT UNSIGNED NOT NULL AUTO_INCREMENT
                      COMMENT 'Global file identifier.',
 `f_fc_uid`           BIGINT UNSIGNED NULL
                      COMMENT 'File category identifier
                               (FK to "category" table).',
 `f_size`             BIGINT UNSIGNED NOT NULL
                      COMMENT 'File size (bytes).',
 `f_upload_datetime`  INTEGER NOT NULL
                      COMMENT 'Datetime of the file upload (Unixtime).',
 `f_save_datetime`    INTEGER NOT NULL
                      COMMENT 'Datetime of the file expiration (Unixtime).',
 `f_src_name`         VARCHAR(255) NOT NULL
                      COMMENT 'Initial file name (as it was on a
                               user PC). Without the extension as
                               It is stored separately in the
                               "f_src_ext" field!',
 `f_src_ext`          VARCHAR(255) NULL
                      COMMENT 'Initial file extension (as it was on a
                               user PC).',
 `f_name`             CHAR(200) NOT NULL
                      COMMENT 'Server-side file name. Five SHA1 hashes.',
 `f_sha1_checksum`    CHAR(40) NOT NULL
                      COMMENT 'File checksum, SHA1 hash.',
 `f_ar_uid`           BIGINT UNSIGNED NOT NULL
                      COMMENT 'File access permissions (FK to
                               "access_permission" table).',
 `f_downloaded`       BIGINT UNSIGNED NULL DEFAULT 0
                      COMMENT 'File downloads counter.',
 `f_al_uid`           BIGINT UNSIGNED NULL
                      COMMENT 'Age restrictions for access to the file
                               (FK to "age_restriction" table). If age
                               restrictions are specified both for the
                               file and for the category to which the file
                               belongs, stricter restrictions are applied
                              (i.e., the maximum age is taken, e.g.:
                               "16+" and "18+" — "18+" is taken).',
 `f_del_link_hash`    CHAR(200) NOT NULL
                      COMMENT 'Link for the file deletion (five SHA1 hashes)
                              if the file was uploaded by unregistered
                              user. Registered users can delete the
                              file in "My Files" section.'
)
-- ...
```

Of course, code without comments looks much shorter and more compact, and this can even be an advantage if you wrote it a couple of days ago and are actively working on it. But if it's code you didn't write yourself, or if you wrote it months (years?) ago, the version with comments is much more informative.

The database design process should be documented. This requirement is a logical continuation and extension of the previous one: comments are fine, but comments alone are often not enough. Especially since comments will only refer to the final state of the schema, and we are now talking about documenting the whole process — from the analysis of the subject area to DBMS settings and the hardware and software environment configuration.

The more complex the designed database is, the more contradictory requirements it has, the more nontrivial nuances the subject area has — the more useful the full documentation is, as it not only captures certain decisions, but also explains why this decision was made, what were the alternatives, what were the arguments that proved decisive.

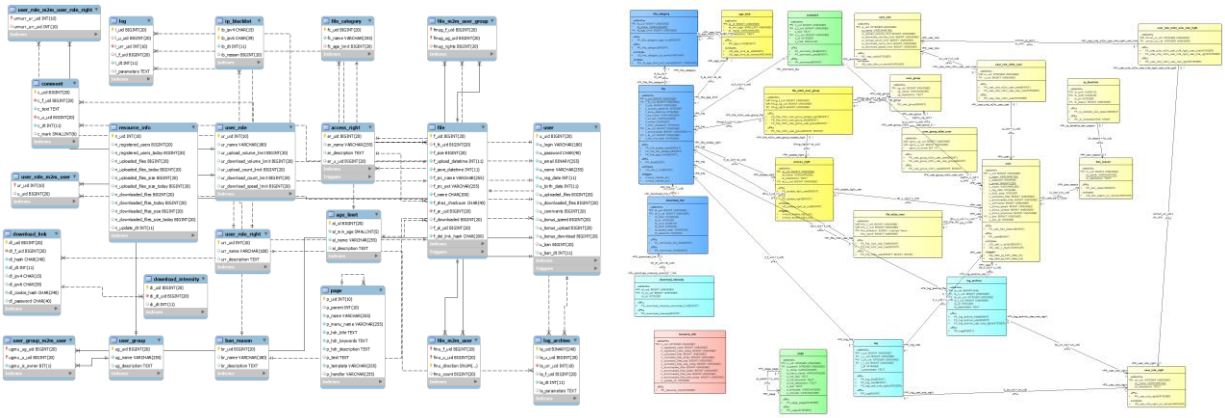
If there is no such documentation, the "refinement" of the database will be much more like walking on a minefield, where any careless action can lead to disastrous consequences, or… it will be easier to develop a new database from scratch than adapt the existing one to the new requirements.

The documentation should present the database schema in a commonly used graphical notation. This rule could be combined with the previous one, but there are too many objections that the graphical representation of the schema can be obtained in any database development environment through reverse engineering{323}.

It is possible. But compare the following two cases (see figure 3.2.z).

This is a microscopic (by industrial standards) database of two dozen tables, but even at this size it is clear that the logically grouped, color-coded, and commented tables are much easier to understand than just a bunch of piled objects.

When we are talking about databases of hundreds of tables, this difference becomes not only significant, but almost insurmountable: attempts to make "human-usable" an automatically generated schema can take days and weeks.



Automatically generated schema Manually created schema

Figure 3.2.z — Automatically generated and manually created schemas of the same database

The schema must be free of unreasonable rigid restrictions and/or atypical solutions. Perhaps nothing undermines the process of database re-development as much as the presence of “strange” and “illogical” solutions that are not reflected in the documentation or insufficiently explained.

In the extreme case, the presence of such artifacts can mean one of two things: either this decision was made “out of desperation” as almost the only way to enforce some kind of technological or business rule, or… it is just someone’s stupidity, that is, a decision made thoughtlessly.

In the second case, of course, it is worth redoing the schema immediately, getting rid of its obvious flaw. In the first case, such a redesign can lead to irreversible consequences.

Therefore, any atypical solution should be documented and explained in as much detail as possible. But even better, such solutions should be avoided altogether.

Data up-to-date state requirement

Like many others, this requirement can be very simple or very complex to fulfill, depending on the data schema, the requirements of the subject area, and a myriad of other factors.

In the simplest case it can be formulated as follows: avoid creating caching and aggregating tables[160], and if you still have to create them, devise a mechanism for timely updates.

Earlier{210} figure 3.2.y gave an example of just such a situation: the `aggregation` relation is an aggregating relation, i.e., it stores the processed (aggregated) data from the `log` relation. In that example, we have agreed that it is allowed to use aggregated data that is a day old relative to data being aggregated — this is an almost utopian convention, used for simplicity in a learning context. In reality, this “allowable lag” can be measured in minutes, seconds, fractions of seconds.

There is not only the problem of updating the aggregated data so frequently, but also the problem of ensuring that it is up-to-date — there should never be a situation where an application working with our database receives outdated data, being sure that it is working with up-to-date data.

Without going into technical details[161] of cache invalidation mechanisms implementation, the most frequent solution (provided it is technically feasible and acceptable) would be to update the aggregated/cached data so often that the “age” of the updated copy is always no greater than the maximum allowed “lag period”.

So, it’s clear that aggregating/caching tables can be both useful and troublesome. What if we try to do without them?

Then we move on to the more complex case of this requirement implementation. On the one hand, we already run the risk of violating the system performance requirement{210}, because some queries may take considerable time to execute. On the other hand, we still have the question: is the data accessed by the application up-to-date at any given moment?

What happens, for example, if in the process of updating several million records the DBMS managed to do only part of this work, and at exactly the same moment the application executes a request to read data from this table? What data will it get? Only updated data? Only outdated data? The original set as it was before the update began?

[160] See example 31 in the “Using MySQL, MS SQL Server and Oracle by examples” book (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

[161] If, however, to touch upon the technical part a little: there is a huge (in fact, infinite) number of algorithms for cache invalidation and ways of technical implementation of accompanying operations. The higher the system performance requirements are, the more nontrivial decisions can be made, up to developing your own hardware and software systems.

Anything different? Or will the application have to wait for the DBMS to finish updating all the data?

Here we come to the phenomenon of transactions, which we will devote a separate section of this book{364} to. For now, let's note that the database schema (and related structures such as views{331}, checks{338}, triggers{341}, stored routines{364}) depend, among other things, on the available ways for applications to interact with the database.

The overall conclusion of this chapter: normalization is not only a way to make the database schema immune to anomalies, it is also a very responsible part of the database design process, requiring increased attention and following the list of rules just discussed.



Task 3.2.d: which of the normalization requirements discussed in this chapter are violated in the "Bank{395}" database schema? Refine the schema to eliminate the relevant deficiencies.



Task 3.2.e: using handouts{4} refine the "Bank{395}" database schema, adding any necessary comments.



Task 3.2.f: what performance problems{210} of the "Bank{395}" database can be predicted from its schema? Refine the schema so as to eliminate the relevant deficiencies.

3.2.3. Normalization Process from a Practical Point of View

Normalization can be considered as a process of analyzing the existing database schema for dependencies{170} and constraints, as well as existing data operation anomalies{157} and redundancy.

In fact, normalization is the analysis of the database schema in order to identify and eliminate its deficiencies.

Relation schemas with identified disadvantages will be reprocessed (usually decomposed{176}) into new schemas that are free of these disadvantages.

Thus, normalization provides:

- a logical framework for analyzing relation schemas;
- a set of normal forms as signs of the depth of normalization and tools of the normalization process.

When considering any relation schema, it is possible to subject it to a series of checks to see if it is in some normal form or another. If the required depth of normalization is not achieved, the schema can be decomposed{176}.

It is important to understand that the mere correspondence of some relation schema to some normal form is not in itself a sign of a well-executed design — it is only one of the conditions. Both the general requirements for any database{10} and many other factors, depending on the subject area, features of the chosen technological platform, etc., should be taken into account.

It is also worth noting that in practice normalization is often stopped at the 3NF{247} level (sometimes at the BCNF{253} level), since normal forms of higher orders often do not bring additional benefits to the database schema but only complicate it. Sometimes normalization can also be stopped at lower levels (e.g., at the 2NF{241} level) — usually in order to provide the necessary performance of operations.

In the following example, we will go through the whole process of normalization from 0NF to 6NF, considering formal proofs of finding (or not finding) the schema of each relation in this or that normal form.



Critically important! As it has already been said repeatedly{70}, {170}, all reasoning about properties and features of relation schemas, relation variables, relations themselves can be carried out only in the context of the subject area. Therefore, we will also argue for the presence or absence of this or that dependency by the requirements of the subject area, which we will explicitly specify.

Yes, the normal forms themselves will be discussed later{235}, but practice has shown that for understanding and memorization it is better to consider the normalization process first, and then the normal forms themselves.

So…

0NF, “terrible schema”

Let’s imagine that due to some unbelievable coincidences we got such a relation schema (see figure 3.2.3.a) — yes, no one in his right mind would make such a schema, but nevertheless…

It is even difficult to make oneself say that this relation schema is in any normal form whatsoever. But formally it is in the so-called “zero normal form{274}”, which makes no requirements at all for the relation schema.

According to the requirements of the subject area we will need to store such data about the employee as:

- full name;
- position;
- department;
- telephone numbers of the department;
- availability of a corporate car;
- names and dates of birth of all children of the employee.

At the moment, apparently, it is supposed to store it all in an arbitrary form in the `data` column.



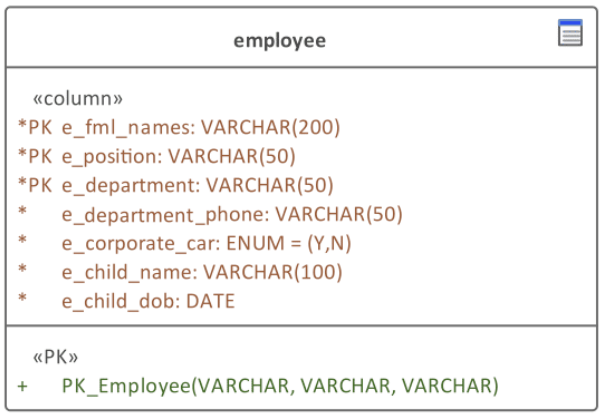
Figure 3.2.3.a — Relation schema in 0NF

Obviously, this situation does not suit us, and therefore we will begin to normalize this schema by bringing it to 1HF{236}.

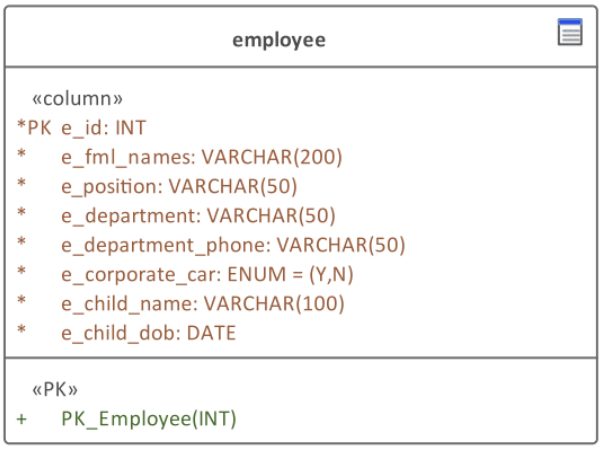
1NF, “atomized attributes”

From this point onwards we will consider two variants of developments: with natural{38} and surrogate{38} primary keys.

The first result of this relation schema transformation of to 1NF{236} is shown in figure 3.2.3.b.



Variant with natural primary key



Variant with surrogate primary key

Figure 3.2.3.b — Relation schema in 1NF, step 1

For the variant with the natural primary key, we have to introduce an additional restriction of the subject area: there cannot be two people with the same full name in the same department in the same position.

For the variant with the surrogate primary key this restriction can be omitted. But if this restriction exists objectively (e.g., there is a rule in the firm that forbids people with the same full name to work in the same department in the same position), we have to create a unique index on the **`e_fml_names`**, **`e_position`**, **`e_department`** fields.

For most fields their atomicity is not questionable, but there may be doubts about the **`e_fml_names`** field, which stores the full name.

As shown in the description of 1NF{236}, there is no “magic formula” to determine whether a field is atomic or not. Therefore, here again we will resort to the requirements of the subject area and decide that in the process of using this database never, under any circumstances, we will not need to use the surname, first name, and patronymic of an employee separately[162], that is, the **`e_fml_names`** field will be considered atomic.

However, the **`e_department_phone`** field is not atomic. According to the subject area requirement, the department can have several phones, i.e., here we get a classical violation of 1NF{236} expressed in the fact that there is a multivalued attribute in the relation schema (the **`e_department_phone`** field has to store many phone numbers).

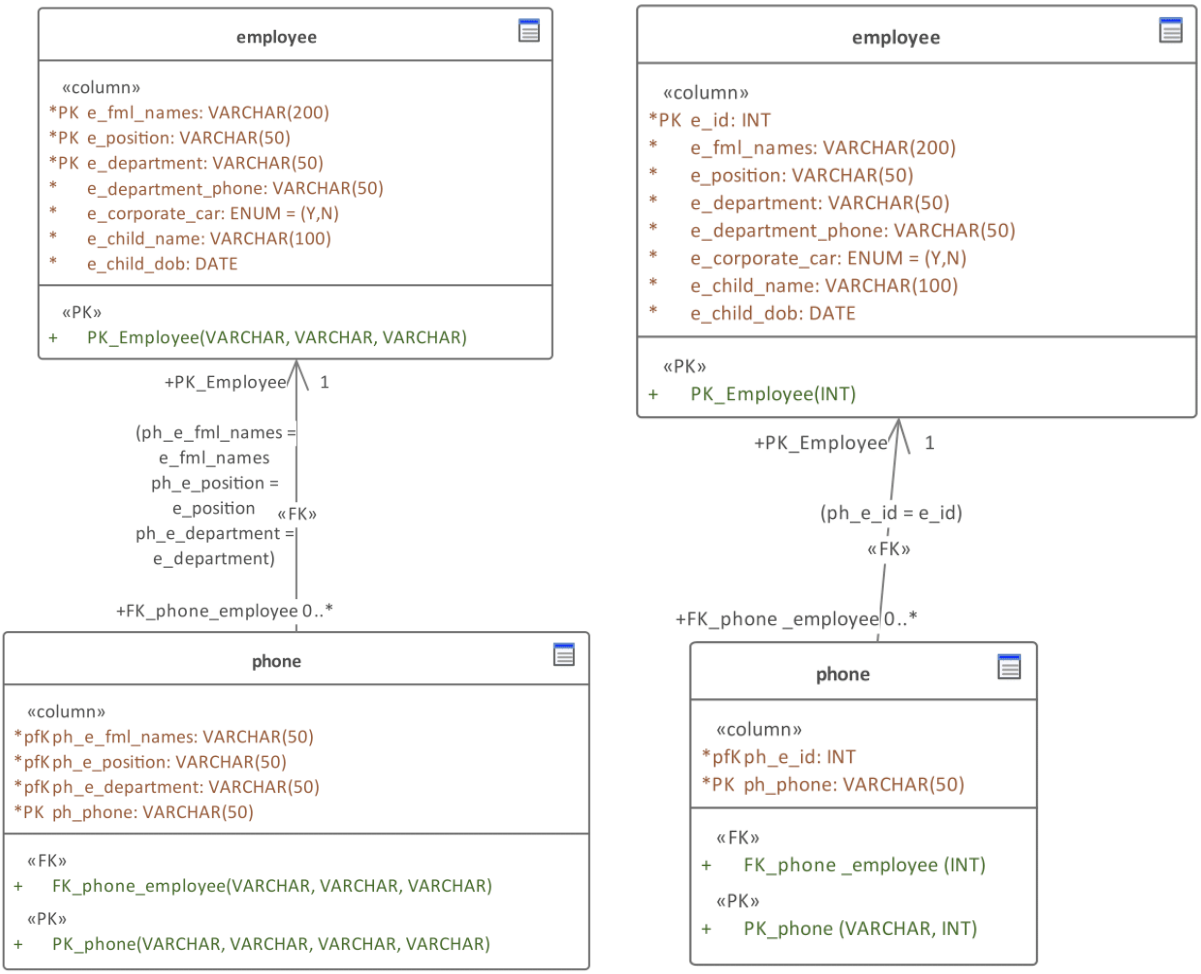
To fix this problem, we decompose the employee schema into two (see figure 3.2.3.c).

The result looks very suspicious in terms of common sense, but at this stage we have no choice — we must link the phone number to the primary key of the parent table (we can’t just link it to a department as its name is not a primary key).

Here we see the problem right away: we cannot rule out the possibility of assigning the same phone number to different departments (triggers{341} can solve this problem, but even there we have to invent an algorithm for deciding whether or not to allow insertion and modification operations on phones, which is not completely obvious).

Yes, we could temporarily “forget” about this problem with the non-atomicity of the **`e_department_phone`** field and return to it after the partial dependencies{179} are eliminated, but in a learning context this would be out of sequence, so let’s just be nervous about this strange solution for now.

[162] This is a very crude assumption, almost never fulfilled in real life. But here it is introduced to simplify further reasoning.



Variant with natural primary key — Variant with surrogate primary key

Figure 3.2.3.c — Relation schema in 1NF, step 2 (final)

So, all requirements of the 1NF are fulfilled: all fields are atomic, there are primary keys (the second requirement is not widespread in the formulation of the 1NF, but since we immediately met it — it certainly did not get worse).

2NF, “elimination of partial dependencies”

Here we need a strong formulation of the 2NF{241} because if we use the weak formulation it appears that the variant with the surrogate key is already in the 2NF.

The strong 2NF formulation says that the relation variable must be in 1NF (we have already provided this) and that every non-key attribute{21} of the relation variable must functionally fully{179} depend on any candidate key{34}.

What candidate keys do we have? This is already mentioned “three attributes” (**`e_fml_names`**, **`e_position`**, **`e_department`**), which in the variant with a natural primary key became the primary key itself, and in the variant with an surrogate primary key remained an alternate key{35}.

It’s immediately apparent that the department’s phones depend solely on the department, but not on the name and position of an employee. Yes, we put the phones in a separate relation, but this did not solve the problem — now we simply have an error in the schema: the phones are “linked” to the employee, not to the department.

And to complete the picture, let’s add another requirement of the subject area: corporate cars are provided only to employees holding certain positions.

So, we have two partial dependencies:

- the **ph_phone** field depends only on the **e_department** field;
- the **e_corporate_car** field depends only on the **e_position** field.

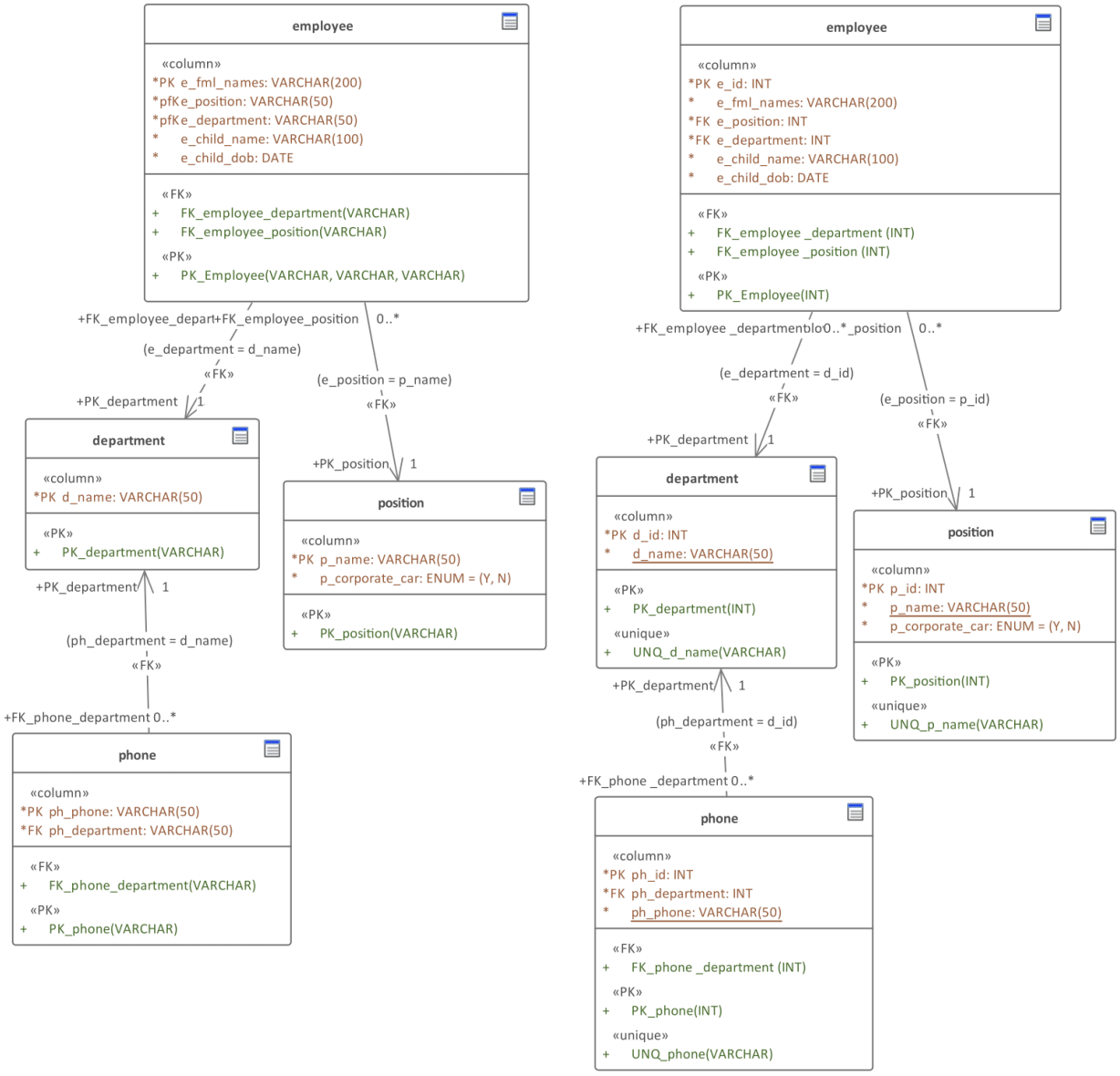
Let's eliminate these dependencies by decomposing the relation schema into several projections (see figure 3.2.3.d).

In both variants (with a natural and a surrogate primary key) the fields that depend on a part of the candidate key have been put into separate relation schemas, and thus the **employee** relation schema has been brought to the 2NF.

The obtained schemas of **position**, **department**, **phone** relations are also in the 2NF: their attributes are atomic (we did not add any new fields, except for surrogate primary keys), and since they have no composite candidate keys, partial dependency of a non-key attribute on a candidate key cannot exist here by definition (for them to exist, a candidate key must have parts, i.e., it must be composite).

Despite the fact that the variant with a surrogate primary key seems more cumbersome, with a large number of employees and a small number of departments and positions it can win in performance and the volume of stored data due to more compact foreign keys in **employee** relation.

Yes, to ensure the uniqueness of department and job titles, as well as to ensure the uniqueness of phone numbers, we had to make unique indexes{106} here, but objectively these relations will change very rarely, and therefore there will be no loss in performance.



Variant with natural primary key Variant with surrogate primary key

Figure 3.2.3.d — Relation schema in 2NF

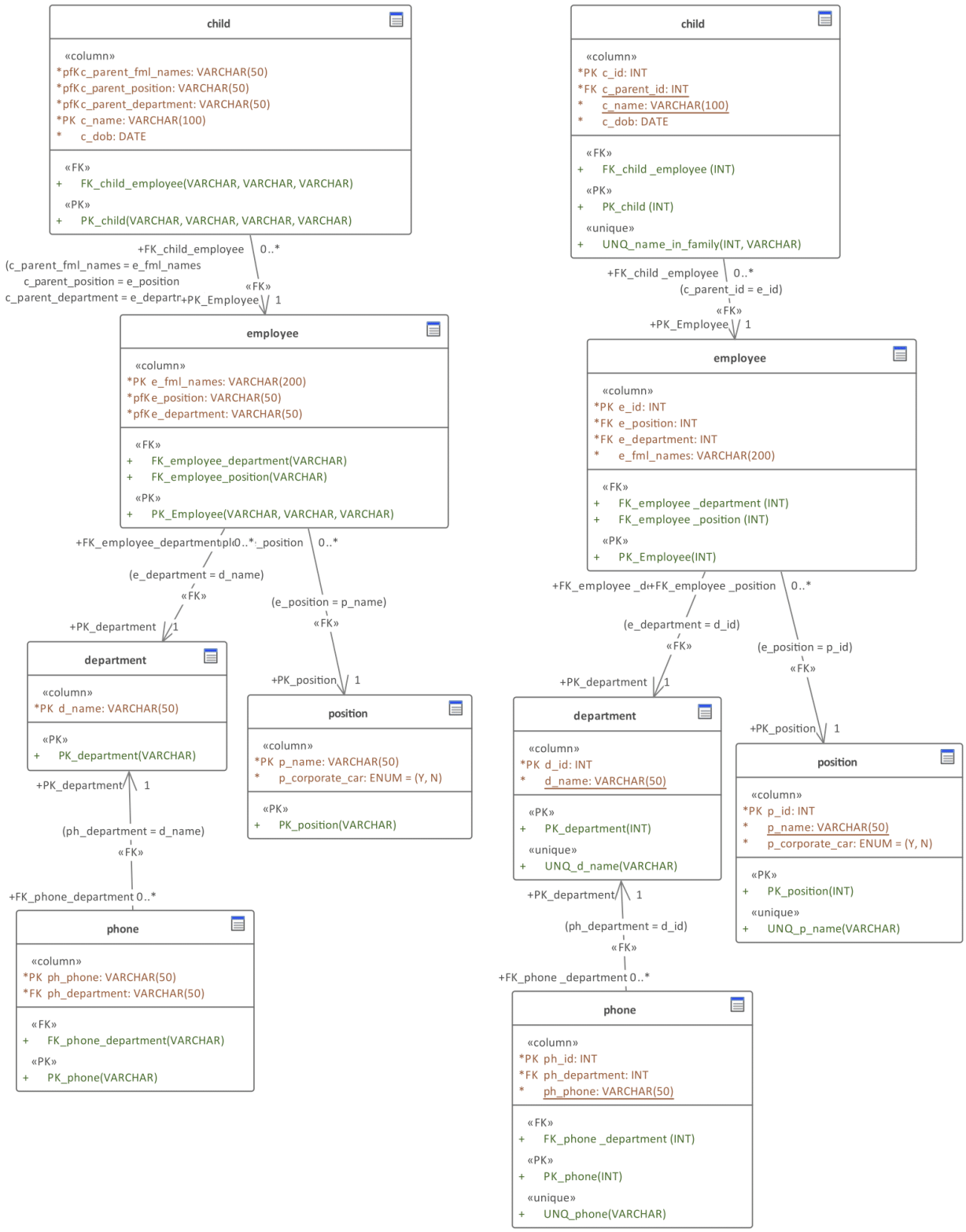
Why did we leave out the `e_child_name` and `e_child_dob` fields? Because they only depend on the whole candidate key (i.e., there is no partial dependency here). After all, in order to answer the question “whose child is this?” we need to be guaranteed to “identify the parent” as the position, or the department, or the full name alone is not enough.

So, for all the available schemas of relations all the requirements of the 2NF are fulfilled: the schemas of relations are in the 1NF, and no non-key attribute depends partially on any candidate key.

3NF, “elimination of transitive dependencies”

The following problem remains to be solved: the child’s birthday depends on the child, not the parent, i.e., there is a transitive dependency `{primary key}` → `e_child_name` → `e_child_dob`.

So, we decompose the employee relation schema once again (see figure 3.2.3.e).



Variant with natural primary key Variant with surrogate primary key

Figure 3.2.3.e — Relation schema in 3NF

The resulting schema of the **child** relation is in the 3NF{247} as:

- 1NF: all its attributes are atomic;
- 2NF: there are no partial dependencies of any non-key attribute on the primary key:
  - in the variant with a natural primary key there is only one non-key attribute **c_dob**, which is defined by the whole primary key (to know the date of birth of a child, we need to know both whose child it is and who the child is);
  - in the variant with the surrogate primary key, this key consists of one field, and the alternate key {**c_parent**, **c_name**} is also completely necessary to determine the date of birth of the child.
- 3NF: since there are no other attributes besides the primary key (and the alternate key in the surrogate primary key variant) and the only non-key attribute in the relation, a transitive dependency cannot exist by definition (as a third, “intermediate” attribute is required for its existence).

Similar reasoning is fully applicable to the rest of the relation schemas in the resulting database schema. Thus, the whole schema is at least in 3NF.

BCNF, 4NF, etc.

The resulting database schema (compare the result in figure 3.2.3.e with the initial situation in figure 3.2.3.a) is already quite usable. But nothing prevents us from checking the “maximum depth” of its normalization.

BCNF{253} requires that in every nontrivial{187} functional dependency {X} → A the {X} set is a superkey{33}. Earlier in the transformation to the 3NF we already dealt with all the non-key attributes, so now let’s look at the key ones.

In the variant with natural primary key this rule could theoretically be violated in **child** and **employee** relations (they have sets consisting of several elements and being determinants of functional dependencies), but in fact there is no violation, because dependent attributes are also parts of the set defining them, i.e., the dependency is trivial.

In the variant with a surrogate primary key such a question is relevant only for the **child** relation, but even there either the dependency is trivial (as the set with **c_parent_id** and **c_name** is defining its components), or the BCNF requirement that the determinant of dependency must be a superkey is satisfied (for {**c_parent_id**, **c_name**} → c_id dependency).

So, all relations of the obtained database schema are in BCNF.

4NF{258} requires that for any nontrivial multivalued{189} dependency X ↠ Y existing in a relation the X set must be a superkey.

But there are no multi-valued dependencies in any of the relation variables in our database, i.e., the 4NF requirement cannot, by definition, be violated.

So, all relations of the resulting database schema are in 4NF.

5NF{262} requires that for every nontrivial join dependency{192} JD(R_1, R_2, …, R_n) existing in a relation each R_i set of attributes is a superkey{33} of the original relation variable.

We cannot perform lossless decomposition{176} of any of the available relation schemas without including primary and/or candidate keys in each of the projections. And a set of attributes containing a primary and/or candidate key is, by definition, a superkey.

So, all relations of the resulting database schema are in 5NF.

DKNF{267} requires that all constraints and dependencies that exist in the relation variable must be the consequence of domains{20} and keys{32} constraints, and that there are no “hidden” rules derived from anything else.

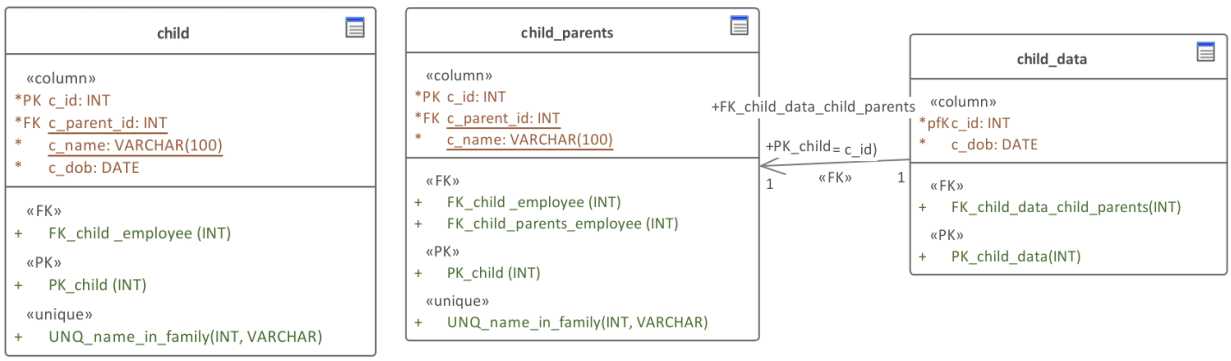
Here the proof will be a bit strange, but nevertheless: let’s assume that we have fulfilled all the requirements of the subject area and reflected them in the database schema structure (by the way, the created unique indexes in the variant with surrogate primary keys following exactly this goal: we guarantee the uniqueness of alternate key values). Then the requirements of the DKNF can be considered satisfied.

So, all relations of the obtained database schema are in DKNF.

6NF{271} requires that the relation variable must not allow in principle any lossless decomposition{176}, i.e., any join{192}, {200} dependencies present in it must be trivial{192}, {200}.

For the variant with natural primary key, this requirement is satisfied, because each existing relation schema has only one non-key field (as a result, at least one projection must contain a primary key and this non-key field, i.e., the projection is equivalent to the original schema, i.e., the join dependency is trivial).

For the variant with surrogate primary key, further decomposition of some relations is possible, but… meaningless. Let’s demonstrate this with the example of the `child` relation schema (see figure 3.2.3.f).



Before decomposition After decomposition

Figure 3.2.3.f — Meaningless decomposition of the `child` relation schema

Such decomposition is technically possible but will bring nothing but the need to perform additional actions in any data operations.

So, all relations of the obtained database schema for the variant with natural primary key are in the 6NF, and `child`, `position`, and `phone` relations for the variant with surrogate primary key are not in the 6NF (but as just shown, it is not necessary).

At the end of this chapter, note that as you gain some experience, you will begin to form database schemas that are immediately in at least 3NF — at some point it just happens instinctively.

As for formally analyzing some schema in order to find out if it is normalized enough, it is much easier and more logical to do it through thinking over queries: if you find a situation where “something goes wrong” (most likely — a data operation anomaly{157} or just awkward, illogical, complex queries are obtained), then it is worth considering options for reworking the existing relation schemas.

Sometimes this rework will involve normalization (or denormalization), but not less often you will simply find errors in the database schema, inadequacy to subject area, and other flaws that can be corrected without involving the normalization process.

At this point we pause with normalization and consider the reverse process — denormalization, to which the next chapter will be devoted.



Task 3.2.g: what is the normal form of each “Bank{395}” database relation schemas? Is it worth changing this normal form? If you think “yes”, make appropriate changes to the schema.



Task 3.2.h: are there any relation schemas in the “Bank{395}” database that cannot be normalized further? Justify your opinion.



Task 3.2.i: are there any relation schemas in the “Bank{395}” database further normalization of which is possible, but for some reason is undesirable? Justify your opinion.

3.2.4. Denormalization

As noted earlier{218}, the normalization process often stops at the 3NF{247} or BCNF{253} level, since deeper normalization is either unnecessary or even harmful (primarily leading to query complexity and performance degradation).

Such an "early stop" of normalization can also to some extent be considered denormalization, but more strictly speaking, the definition of denormalization is as follows.



Denormalization[163] — a process of bringing relation schemas to a lower normal form by joining them.

Simplified: the result of JOIN of two or more tables is saved as one new table.

In general, four types of denormalization can be distinguished:

- joining relations' schemas;
- creating caching relations schemas or individual caching attributes;
- creating aggregating relations schemas or individual aggregating attributes;
- creating persistent (materialized) views{331}.

Although only the first option belongs to "classical denormalization," we will now consider them all one by one.

Joining relations' schemas

Let's imagine that some fictional organization has a huge number of departments, each with a huge number of phones. We first decided to use the database schema shown in figure 3.2.3.e{224}, but very quickly ran into the fact that performing a `JOIN` of the `phone` and `department` tables takes an unacceptably long time from the customer's point of view.

One solution to this problem would be to join these tables into one, as shown in figure 3.2.4.a.

Now all phone numbers of the department are stored in the same table (even in the same record) where the information about the department is stored, and therefore are available within the query to the same table, which solved the problem of "unacceptably long running `JOIN`".

But this solution caused a new problem: now the `department` relation schema is not even in 1NF{236} because it has a multivalued attribute), and we will have to implement (at the DBMS level or at the level of the application working with the database) a special algorithm to at least update the `d_phones` field.

Note also that finding the department that owns a certain specified phone now also becomes a more difficult and longer task.

This example already shows that there is no general perfect solution: by winning at one thing, we sacrifice something else.

[163] **Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

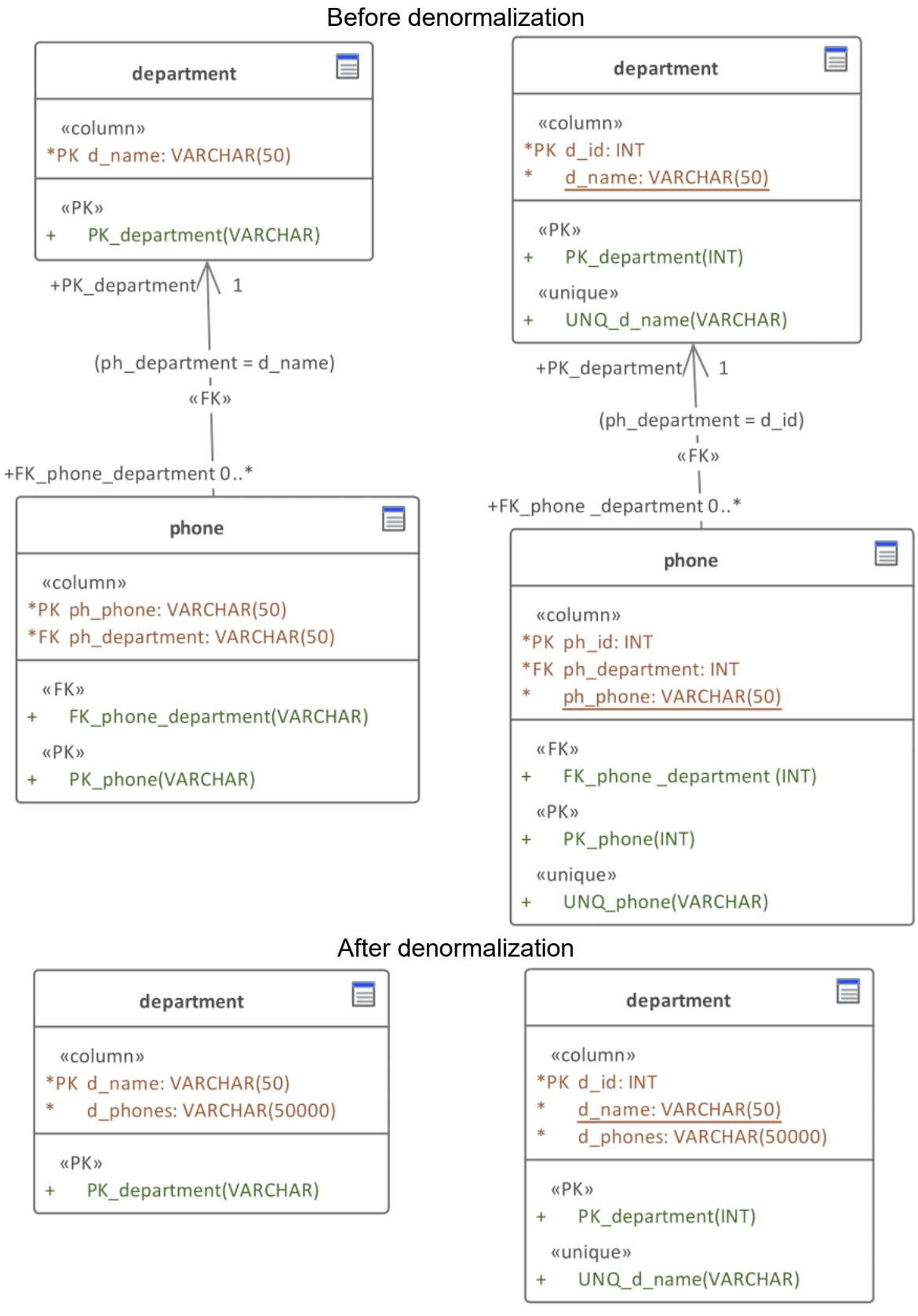


Figure 3.2.4.a — Denormalization by joining relations' schemas

Creating caching relations schemas or individual caching attributes

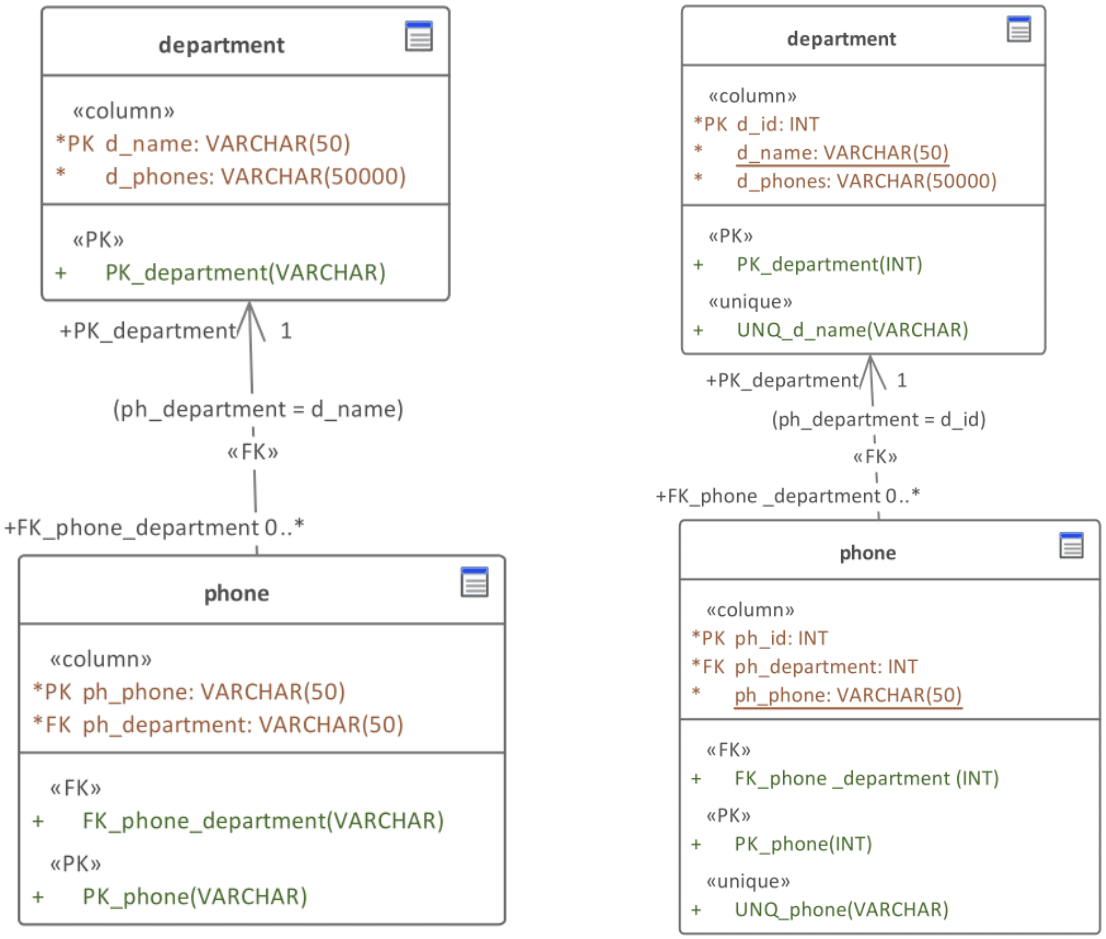
The just noted problem with nontrivial updating of **`d_phones`** field can be solved quite elegantly: the data in it can be viewed only as a temporary fast storage (cache) of data, and the long-term storage can be performed (as before) in a separate table.

Such a database schema is shown in figure 3.2.4.b.

With this approach, all operations on insertion, modification, or deletion of phone numbers are still performed using the **`phone`** relation. As well as a quick search for the department that owns the specified phone number can also be performed using the **`phone`** relation.

And the **`d_phones`** field of the **`department`** relation should be automatically updated "collecting" all department's phone numbers when the **`department`** relation changes. Such automatic updating can be implemented using triggers{341} (see also practical examples in the corresponding book[164]).

If we go one step further, we move from caching attributes to caching relations. Suppose for some reason we need to get a list of employees' names with a list of each employee's children immediately. We can create a separate **`family`** relation, as shown in figure 3.2.4.c.



Variant with natural primary key — Variant with surrogate primary key

Figure 3.2.4.b — Example of denormalization by creating a caching attribute

[164] See the "Using MySQL, MS SQL Server and Oracle by examples" book (Svyatoslav Kulikov), chapter 4.1 [https://svyatoslav.biz/database_book/]



Variant with natural primary key | Variant with surrogate primary key

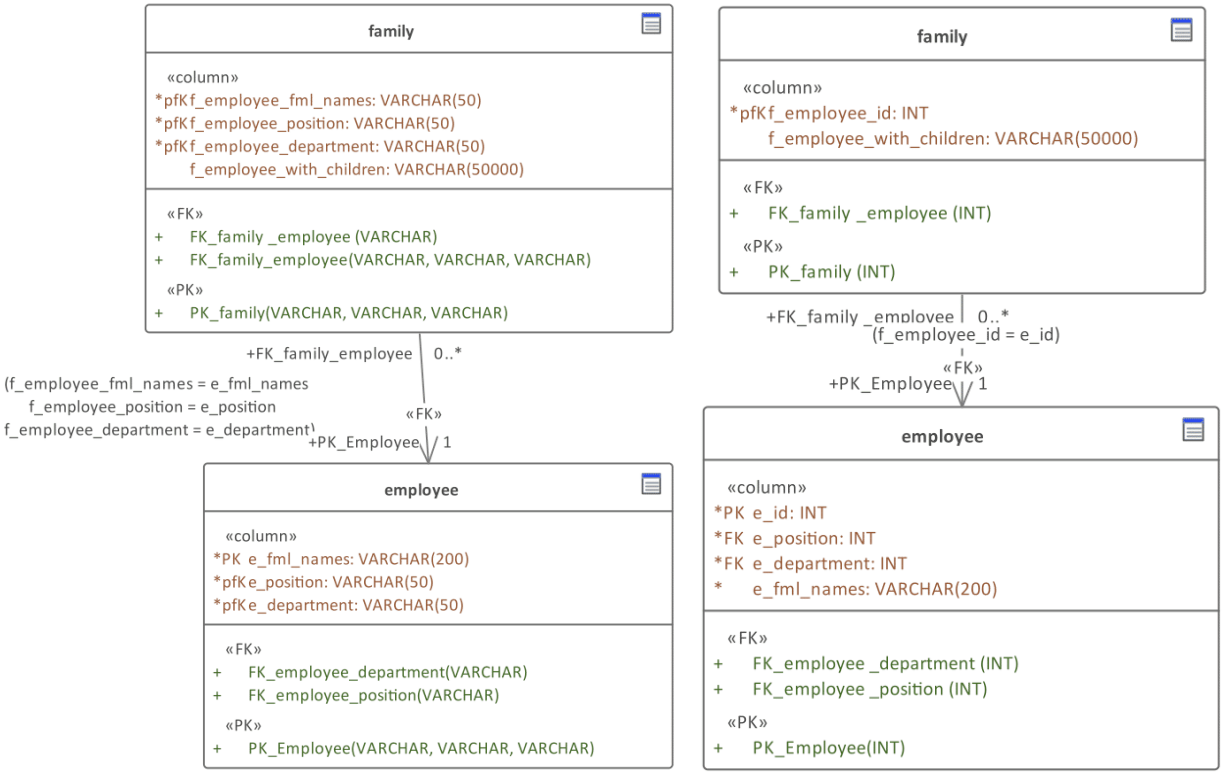
Figure 3.2.4.c — The first example of denormalization by creating a caching relation

It is easy to see that in this case there is no difference in the variants with natural and surrogate primary keys — we just create a table, each row of which will store information about one employee and all of their children. There is no even a primary key.

This relation is in the 0NF{274}, i.e., it violates all imaginable and unimaginable rules of database design, but it well serves the task assigned to it.

In the solution shown in figure 3.2.4.c, we will have to update the entire data in the **`family`** table using triggers every time a change is made to the **`employee`** or **`child`** table, because information about “where is which employee” and “where is whose child” is not stored anywhere.

But we can slightly reduce the potential workload of the DBMS. Let’s add a foreign key from the **`family`** table to the **`employee`** table (see figure 3.2.4.d).



Variant with natural primary key | Variant with surrogate primary key

Figure 3.2.4.d — The second example of denormalization by creating a caching relation

Now when we change the `employee` and/or `child` table, we can find out for which employee we need to update the data in the `family` table, i.e., we can update only the individual record and not the whole table.

Such a solution raises two more important nuances:

- note how much more compact and logical the surrogate primary key variant looks — this is another important argument in favor of using such primary keys;
- whether it is necessary to create a primary key (or at least just an index) on the employee identifier in `family` table is an open question: it depends on the specific DBMS, the specific storage engine{30} and many other parameters, i.e., there is no definite answer (although we intuitively want to create a primary key or unique index to at least speed up the “search for the employee whose data should be updated” operation).

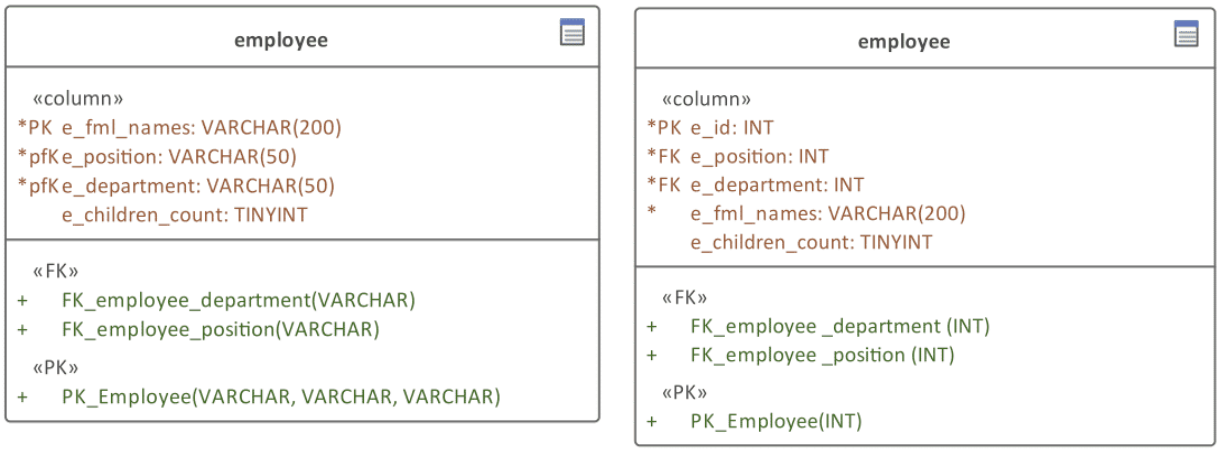
It would seem that there is no further way to denormalize the database schema, we have already “hit rock bottom”, having reached 0NF{274}. But… we could go even further.

Creating aggregating relations schemas or individual aggregating attributes

The difference between caching and aggregation[165] is that based on the cache (even if not always, even theoretically) there is a chance to recover the original data (yes, in reality it is not required, but it is possible), and based on the aggregated data it is impossible to recover the original information in principle. Nevertheless, aggregated data is very useful.

Let’s continue with the example of employees and their children. Suppose we need to find out the number of children of any employee immediately (i.e., a `JOIN` with `COUNT()` is an unacceptably long operation).

We can pre-calculate the values we are looking for (yes, again with triggers{341}) and store them in the `employee` table in the `e_children_count` field, as shown in figure 3.2.4.e.



Variant with natural primary key Variant with surrogate primary key

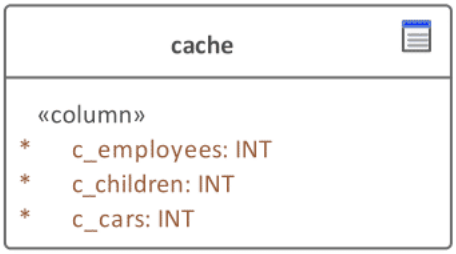
Figure 3.2.4.e — Example of denormalization by creating an aggregating attribute

[165] Since the aggregation result is actually stored in a caching field or relation, this terminological difference is very often neglected. It is, in fact, insignificant. In both cases, "some data is somehow collected and stored in a convenient place from which it can be retrieved very quickly in ready form”.

Aggregating relations are usually used to store a large amount of “miscellaneous” information. E.g., for a corporate portal we need to instantly give some widget the data on how many employees we have, how many employees have children, how many employees are entitled to corporate cars.

Objectively, this information cannot be placed in any of the existing relations, i.e., a new one must be created. And it can be created in two ways, which we will conventionally call “horizontal” and “vertical”.

“Horizontal” method is used when we need to aggregate a small unchanging set of data — see figure 3.2.4.f. There will always be exactly one row in such a table, each column of which contains the data we are looking for.



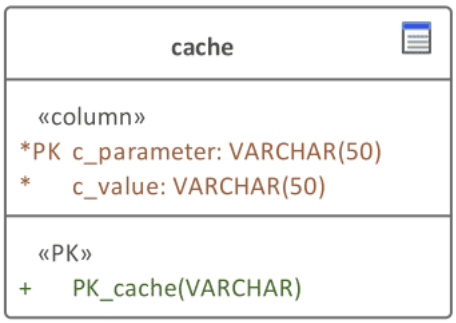
Relation schema

c_employees	c_children	c_cars
257	342	125

Data sample

Figure 3.2.4.f — “Horizontal” aggregating relation

“Vertical” method is used when we need to aggregate a large and/or changing set of data — see figure 3.2.4.g. Such a table will always have exactly two columns (parameter name, parameter value), and each of its rows will store one “key-value” pair.



Relation schema

c_parameter	c_value
employees	257
children	342
cars	125

Data sample

Figure 3.2.4.g — “Vertical” aggregating relation

Despite the seemingly great universality, the “vertical” method has one serious drawback: we cannot specify different data types for different parameters. Therefore, as a rule, `string` data type is chosen as the most universal.

Creating persistent (materialized) views

This solution is a particular case of creating caching and aggregating relations for DBMSes that support so-called persistent (materialized) views{331}.

We will describe the mechanism in more detail in the corresponding chapter{331}, yet for now let’s note the essence: the data that is collected by the view are automatically saved by the DBMS as a table, and are just as automatically updated according to the specified algorithm.

This approach avoids creating triggers{341}, but in many cases it leads either to an increased load on the DBMS (if the data in the persistent view is updated when no update is needed for real) or to cache obsolescence (if the data in the persistent view was not yet modified, although the real data has already changed).

In this respect, triggers and classical caching and aggregating views give us more leeway, but nothing prevents combining these approaches if the DBMS we choose allows it.

We conclude this chapter with two important facts:

- it is worth distinguishing a denormalized database schema (which was first normalized and then denormalized) from just an insufficiently normalized schema: the former is good, and the latter is just an example of poor design;
- denormalization, while bringing its benefits, still has a price: we need to create and maintain additional database structures, sacrificing the volume of stored data and the performance of some operations.

Now we need only consider the normal forms themselves. Let's begin…



Task 3.2.j: are there any relation schemas in the "Bank{395}" database whose denormalization would result in improved performance while not resulting in data operation anomalies{157}? Justify your opinion.



Task 3.2.k: should we add any aggregating relation schemas to the "Bank{395}" database? If you think "yes", please make appropriate changes to the schema.



Task 3.2.l: should we add any materialized views to the "Bank{395}" database? If you think "yes", please make appropriate changes to the schema.

3.3. Normal Forms

If for some reason you have skipped the section on dependency theory{170} and now doubt that you are sufficiently familiar with the mathematical foundations of normal forms, it is advisable to refresh your memory on the relevant material{170}.



In various literature you may encounter formulations of the definition of normal forms that begin with the words “the relation schema is...”, “the relation variable is...”, “the relation is...” (for more details on the difference between these notions, see here{22}).

To be perfectly scientifically correct, it is worth using the term “relation variable” in the definition of normal forms (see the original source[166]).

In the following, however, we will deliberately use the term “relation” instead of “relation variable” in some cases in order to avoid additional nontrivial explanations and to simplify the presentation of the material. All the more so because all the examples give particular relations, even if they are the values of a relation variable.

Before considering specific normal forms, let’s give a general definition.



Normal form (NF[167]) — a certain property of the relation variable, which depends on the type of the normal form and is provided by normalization{207}.

Simplified: one of the normal forms listed below (yes, this definition looks somewhat “recursive,” but the vast majority of original sources do not provide a pure definition of “normal form” at all).

Now let’s get to the specifics.

[166] C.J. Date, “An Introduction to Database Systems”, 8th edition. Chapter 3.3 (“Relations and relvars”).

[167] **Normal form** of a relvar: see first normal form; second normal form; etc. (“The New Relational Database Dictionary”, C.J. Date)

3.3.1. First Normal Form

Historically, the first normal form was introduced to prohibit the creation of multi-valued and compound attributes and their combinations (so called “tables within tables”). Since no relational DBMS today allows creating a “table within a table”, the emphasis in the definition of the first normal form shifts to a deeper interpretation of the concept of “atomicity”.



A relation variable is in the **first normal form** (1NF[168]) if and only if each attribute of the relation contains strictly one atomic value[169].

Simplified: every relation attribute is atomic (i.e., the DBMS must not operate on any separate part of the attribute).

Consider examples of violation of the first normal form (see figure 3.3.a).

As just noted, no modern relational DBMS allows to specify “table” as the table field type, i.e., to create a “table within a table”. Even though object-relational DBMSes allow using complex structures (actually, objects) as attribute values, we will return to classical relational databases and take a closer look at cases of multivalued and compound attributes.

In figure 3.3.a the multi-valued attribute `s_mark` is a list of grades. Obviously, when performing such typical operations as searching for minimum, maximum and average grades the DBMS will have to analyze the contents of this attribute in every record — in other words, the attribute is not atomic.

[168] A relvar is in **1NF** if and only if in every legal value of that relvar every tuple contains exactly one value for each attribute. (C.J. Date, “An Introduction to Database Systems”, 8th edition)

[169] Some authors explicitly add to this definition the necessity of having a primary key in the relation (in fact, that is why 1NF, 2NF, 3NF, BCNF are called “normal forms based on primary keys”), but in the vast majority of original sources there is no such requirement. In any case, such peculiarities are of scientific rather than practical interest.

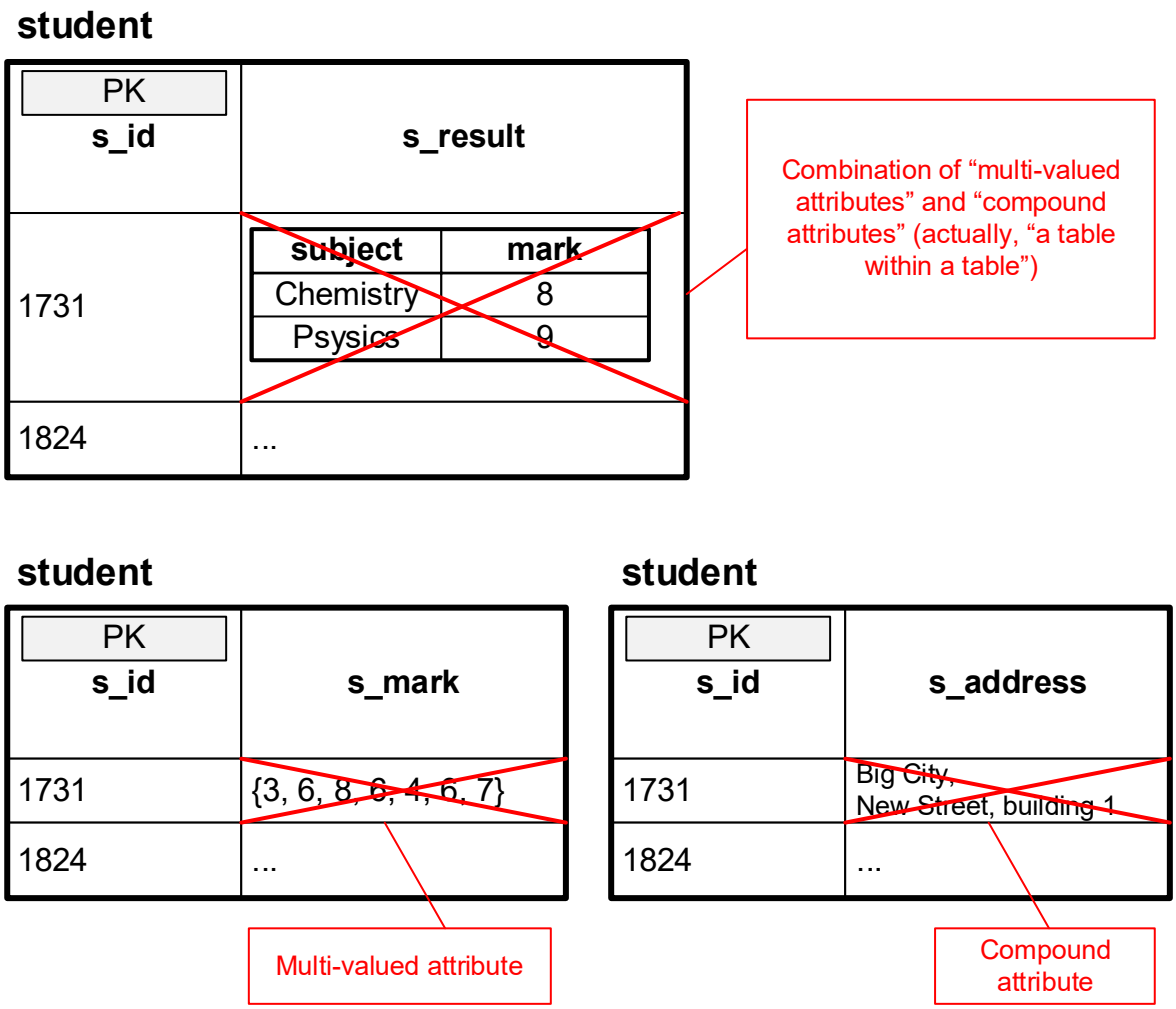


Figure 3.3.a — Violations of the first normal form

To bring a relation with multivalued attributes to the 1NF, it is necessary to remove each such attribute into a separate relation and connect it with the original relation by a “one to many{53}” relationship. The result of such an operation is shown in figure 3.3.b.

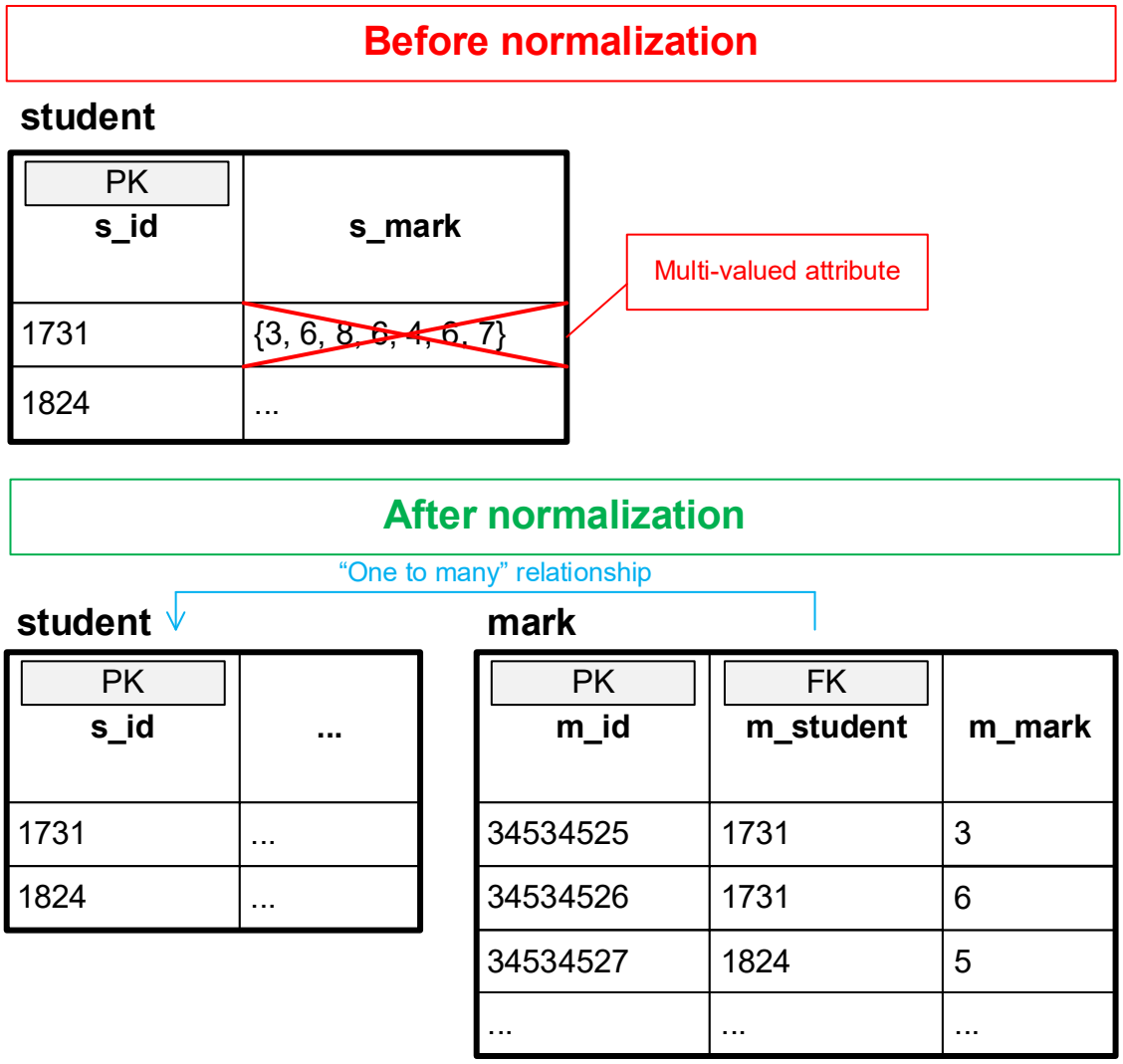


Figure 3.3.b — Bringing a relation with multivalued attributes to the 1NF

If it's simple and obvious with multi-valued attributes, with compound attributes the question becomes almost philosophical: how can we understand that an attribute is already "atomic enough"? In the example in figure 3.3.a everything is quite trivial: yes, you can put city, street, and house number in separate fields — the result of such operation is shown in figure 3.3.c.

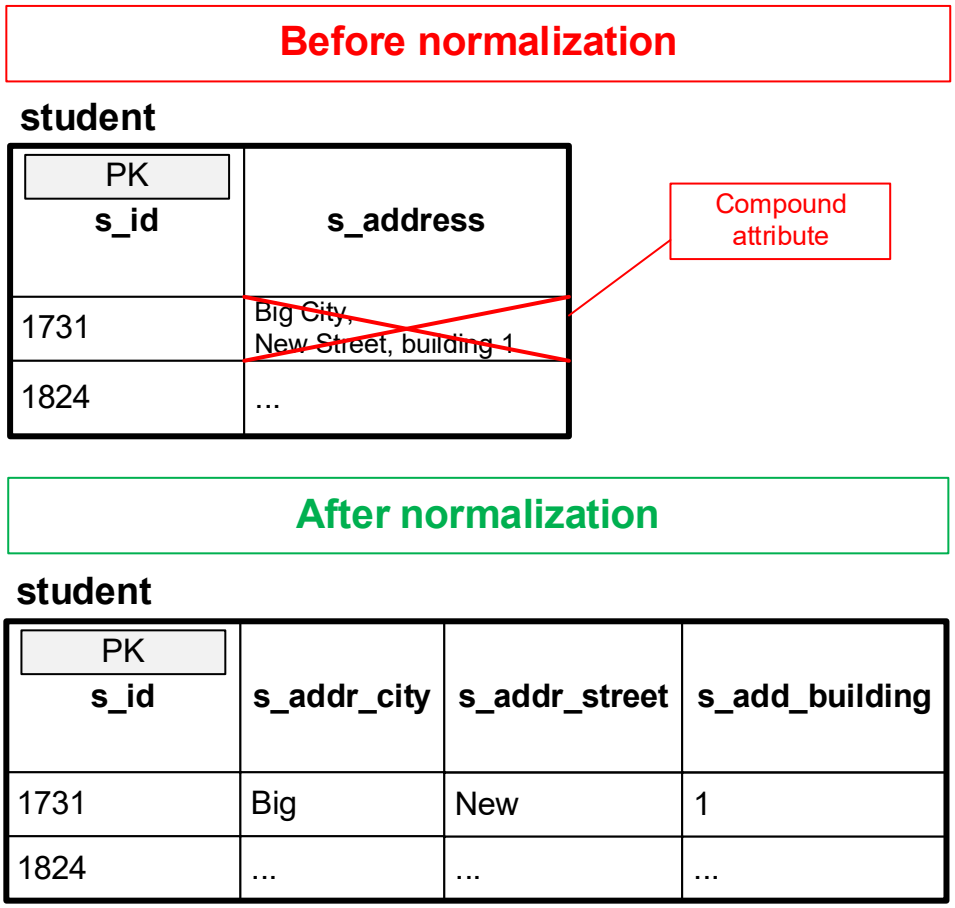


Figure 3.3.c — Bringing a relation with compound attributes to the 1NF

And now an unexpected question: was it necessary to perform this operation? Was the `s_address` really not atomic? As a rule, most people here believe that it was not, and therefore the operation of splitting it into three separate attributes is justified.

What about the following cases:

- Should the operator code and phone number be stored in separate fields?
- Should the username and domain be stored in separate fields for the e-mail address?
- Should the year, month, and day be stored in separate fields? If so, should we keep the day of the week as well? And the number of the week in the year? And an indication of whether it is a holiday?
- And for the time, should the hours, minutes, and seconds (as well as their fractional part) be stored in separate fields?
- Should the passport series and number be stored in separate fields?

We can think of many such questions, i.e., examples from subject areas. They all have one thing in common: doubts about the usefulness of splitting the original value into parts. After all, if we “don’t need” separate parts (we don’t make the DBMS analyze them) such a division is at least meaningless, and at most harmful (we have to constantly merge these fragments into a single whole when performing data operations).

If we approach this situation thoughtlessly, we’ll take it to the point of absurdity and start storing every bit of data in a separate table field — now it definitely can’t be more “atomic”.

But from a practical point of view, it is not so bad: we have to estimate the number and complexity of operations in which we will analyze part of the attribute value. Of course, ideally the number of such operations is zero, and then our relation is fully consistent with the definition of the 1NF. But in reality, for the sake of a small percentage of extremely rare and fast operations we will not "cut" an attribute and thus complicate and slow down all other operations which operate on the value of this attribute as a whole.

So, for example, the date may well be stored in a field of `DATE` type as a whole, if we do not have a constant need to search for records according to "all events that occurred in April and August of any year" or "all events that occurred on the first day of any month".

A brief conclusion on the first normal form:

- the DBMS will not allow us to make a "table within a table";
- each multivalued attribute must be removed by putting it into a new relation;
- each compound attribute is worth scrutinizing to see if it is indeed compound (not atomic) and, if so, we have to split it into several new separate attributes.



Task 3.3.a: are there any relation schemas in the "Bank{395}" database that are not in the first normal form? If you think "yes", make appropriate changes in the schema to bring such relation schemas to the appropriate normal form.

3.3.2. Second Normal Form

If for some reason you have skipped the section on dependency theory{170} and now doubt that you are sufficiently familiar with the mathematical foundations of normal forms, it is still advisable to refresh your memory on the relevant material (it is especially worth re-reading the definitions of full{179} and partial{179} functional dependencies).



{Here is the so-called “weak formulation” of the definition of the 2NF. It is often used in various books, but below it will be shown why the 2NF should be considered in the “strong formulation” of its definition.}

A relation variable is in the **second normal form** (2NF[170]) if and only if it is in the first normal form{236} and each of its non-primary attributes{21} is functionally fully{179} dependent on the primary key{36}.

Simplified: no attribute that is not a part of the primary key must not functionally depend on a part of the primary key.

It follows from this formulation of the 2NF definition that if the primary key of a relation is simple{36} and the relation is in the 1NF, then it is automatically in the 2NF too. It seems very good and simple (it is enough to add a simple surrogate{38} primary key to a relation in the 1NF), but there is a more complete definition of the 2NF.



{Here is the so-called “strong formulation” of the 2NF definition.}

A relation variable is in the **second normal form** (2NF[171]) if and only if it is in the first normal form{236} and each of its non-key attributes{21} is functionally fully{179} dependent on any candidate key{34}.

Simplified: no attribute that is not part of a candidate key should be functionally dependent on any part of any of the candidate keys.

To begin, consider an example of a violation of the second normal form in the weak formulation (see figure 3.3.d).

The `group` relation describes a study group, and its attributes mean the following:

- `g_number` — the sequence number of the group within the year of admission;
- `g_start_year` — the year of admission (the year when the group began its studies);
- `g_years` — years to study (how many years after the start of study the group should graduate);
- `g_head` — the identifier of the group head (foreign key).

Since several groups can be formed in each admission year, this relation has only one obvious candidate key — the {`g_number`, `g_start_year`} combination, which was chosen as the primary key.

The `g_years` attribute cannot be part of the candidate key (we see that its combinations with `g_number` and `g_start_year` attributes are duplicated.

[170] A relation schema R is in **2NF** if every nonprime attribute A in R is fully functionally dependent on the primary key of R. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

[171] A relation schema R is in **2NF** if every nonprime attribute A in R is not partially dependent on any key of R. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

Why can't we consider `g_head` attribute (the same student cannot be the head of several groups, i.e., the value of this field will be unique) as another candidate key? Because then we would have to include a `NOT NULL` property to this field and we would get an insertion anomaly right away{157} — when forming the group, we would have to assign the head right away, which objectively would not be true in most cases.

Let's consider the dependencies available shown in figure 3.3.d:

- The full functional dependency {`g_number`, `g_start_year`} → {`g_head`} shows that the identifier of the head depends on the primary key entirely (indeed — to know the head of a group, we must be guaranteed to identify the group).
- The partial functional dependency {~~`g_number`~~, `g_start_year`} → {`g_years`} shows that the duration of study depends only on the year of admission but does not depend on the group number within the year of admission. Here we see that previously it took five years of study (for applicants up to and including 1999), and then it began to take four years (for applicants from 2000 onwards).

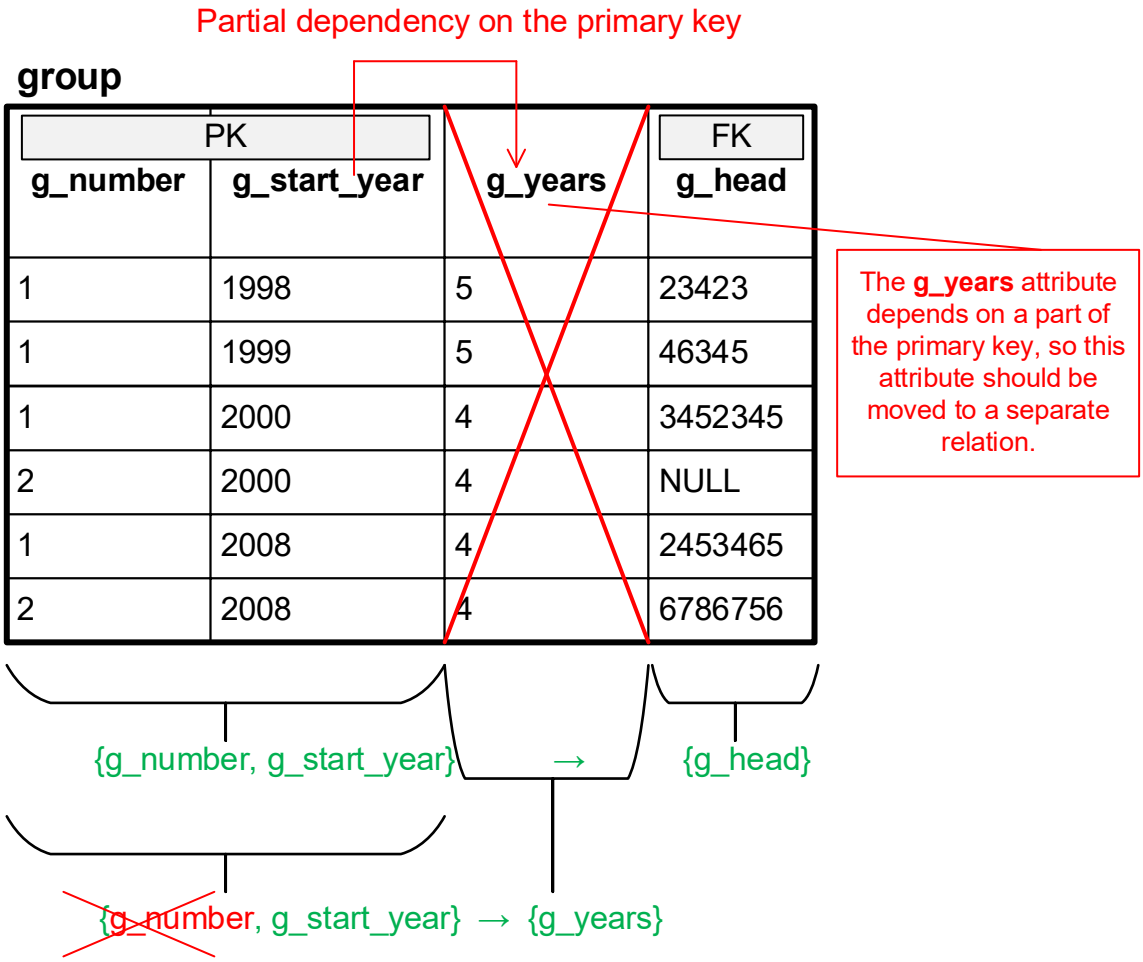


Figure 3.3.d — Violation of the second normal form (in the "weak formulation")

So, we see that there is a non-key attribute (`g_years`) that depends on a part of the primary key, i.e., the “weak formulation” of the definition of the second normal form is violated, and the `group` relation is not in the second normal form.



The “weak formulation” of the 2NF definition is widespread because relations with composite alternate keys do not often occur in real life. That is, if there are no such keys in the relation, one can be guided by the “weak formulation” when normalizing to the 2NF. But if there are such keys (as will be shown below) — it is necessary to be guided by the “strong formulation”.

Before discussing the consequences of violation of the second normal form and ways to bring relation variables to it, let’s consider violation of the second normal form in its “strong formulation”. To do this, let’s add the `g_id` surrogate primary key to the `group` relation (see figure 3.3.e).

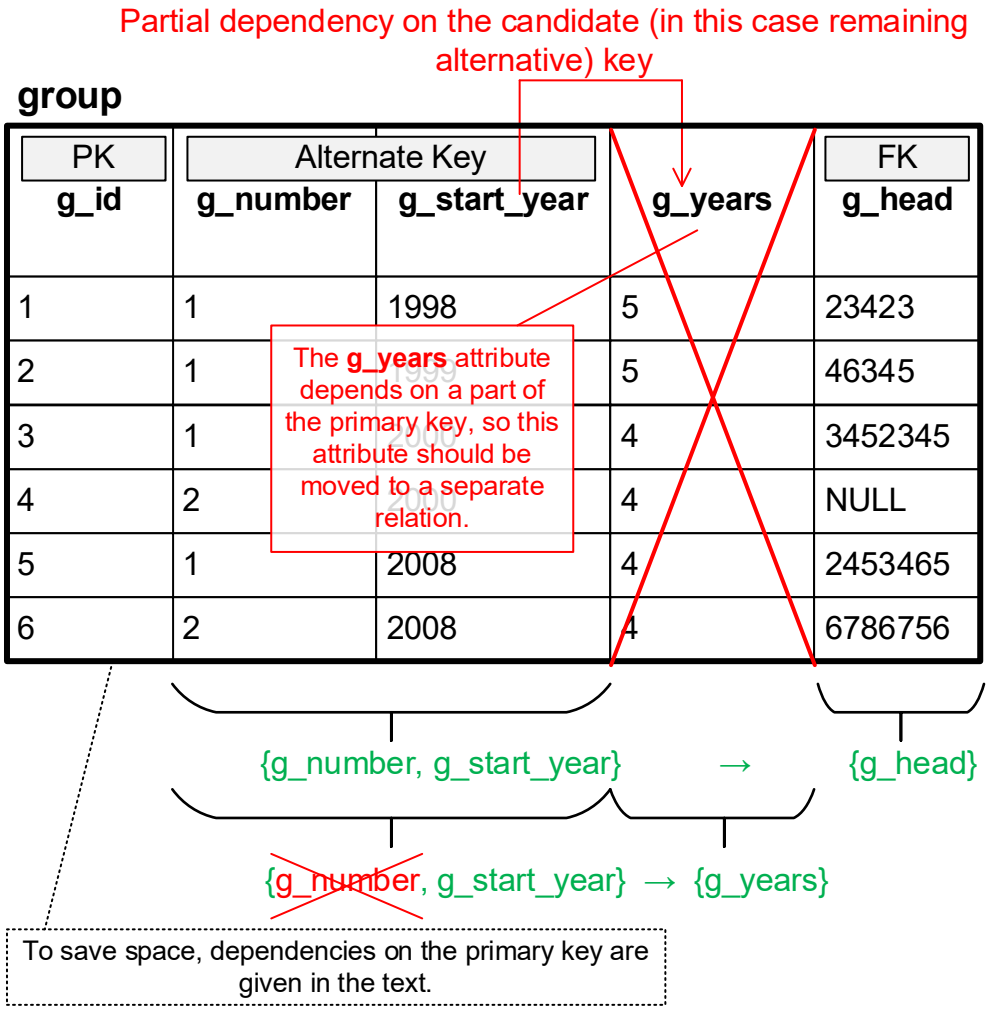


Figure 3.3.e — Violation of the second normal form (in the “strong formulation”)

Let's consider the dependencies in figure 3.3.e:

- The full functional dependency {**g_number**, **g_start_year**} → {**g_head**} has not changed compared to the situation in figure 3.3.d.
- The partial functional dependency {~~**g_number**~~, **g_start_year**} → {**g_years**} has not changed compared to the situation in figure 3.3.d.
- There are four new functional dependencies:
  - {**g_id**} → {**g_number**} as the unique identifier of the group unambiguously defines its number;
  - {**g_id**} → {**g_start_year**} as the unique identifier of the group unambiguously defines the year of the beginning of the study;
  - {**g_id**} → {**g_years**} as the unique identifier of the group unambiguously defines the education length;
  - {**g_id**} → {**g_head**} as the unique identifier of the group unambiguously defines its head.

So, all non-key attributes functionally fully depend on the primary key, i.e., "weak formulation" of the definition of the second normal form is fulfilled, but "strong formulation" is violated, because there is the **g_years** attribute dependent on a part of the {~~**g_number**~~, **g_start_year**} candidate key.

And now it is time to show the consequences of violating the second normal form. Suppose there is a rule in the subject area (it is fully implemented in figures 3.3.d and 3.3.e): "for applicants up to and including 1999 it takes five years to study, and for applicants in 2000 and later it takes four years to study.".

But what prevents us from breaking this rule by adding new or changing the current data so that (for example) for the group that started in 1998, the study period is 4 years (or 3, or 8, or any other number of years)? In the current state of the database schema, nothing.

An intuitive desire to include control of this rule through a trigger{341} can lead to the opposite, but no less dangerous situation: what happens if the requirements change in the future? For example, for applicants "from 2015 onward", the learning period should again become five years.

The only option that ensures both that the relationship between the year of enrollment and the education length at present and that this rule can be adjusted in the future is to create a separate relation, as shown in figure 3.3.f.

After normalization, the only non-key **g_head** attribute of the **group** relation is functionally fully dependent on all candidate keys:

- {**g_id**} → {**g_head**} i.e., the group identifier unambiguously identifies the head of the group;
- {**g_number**, **g_start_year**} → {**g_head**} i.e., the alternative way to guarantee the identification of the group (alternate key{35}) unambiguously defines its head.

In **education_length** relation the only non-key attribute **el_years** is functionally fully dependent on the only candidate key (which, for lack of alternatives, is chosen as primary):

- {**el_start**} → {**el_years**} i.e., the year of the beginning of education unambiguously identifies the length of education.

Thus in both obtained relations there are no non-key attributes{21} partially dependent{179} on candidate keys{34}, which fully meets the requirements of the "strong formulation" of the definition of the second normal form.

And now, by specifying the year of the beginning of education for a new group, we are guaranteed to correctly specify its length of education.

Before normalization

Partial dependency on the candidate (in this case remaining alternative) key

group

PK	Alternate Key			FK
g_id	**g_number**	**g_start_year**	**g_years**	**g_head**
1	1	1998	5	23423
2	1	1999	5	46345
3	1	2000	4	3452345
4	2	2000	4	NULL
5	1	2008	4	2453465
6	2	2008	4	6786756

The **g_years** attribute depends on a part of the primary key, so this attribute should be moved to a separate relation.

After normalization

"One to many relationship"

group

PK	Alternate Key		FK
g_id	**g_number**	**g_start_year** FK	**g_head**
1	1	1998	23423
2	1	1999	46345
3	1	2000	3452345
4	2	2000	NULL
5	1	2008	2453465
6	2	2008	6786756

education_length

PK	
el_start	**el_years**
1998	5
1999	5
2000	4
2008	4

The **el_years** (formerly **g_years**) attribute is functionally fully dependent on a single candidate key.

Figure 3.3.f — Bringing the relation to the second normal form

Summary of the second normal form:

- we should be guided by a “strong formulation” of the definition of the 2NF and look for partial dependencies of non-key attributes on candidate keys, not only on the primary key;
- each attribute that partially depends on a candidate key should be moved to a separate relation, making the “part of the candidate key” on which the transferred attribute depended before the normalization as the primary key of this new relation.



Task 3.3.b. in addition to the possibility of mistakenly specifying the length of education in the relations shown in figures 3.3.d and 3.3.e, there is a second problem there — data modification anomalies. Look closely at these relations and determine what anomalies they are subject to.



Task 3.3.c: are there any relation schemas in the “Bank{395}” database that are not in the second normal form? If you think “yes”, make appropriate changes in the schema to bring such relation schemas to the appropriate normal form.

3.3.3. Third Normal Form

Before reading the material in this section, it is worth recalling the definitions of transitive{183} and nontrivial{187} functional dependencies.



{Here is the so-called “simplified formulation” of the 3NF definition.}

A relation variable is in the **third normal form** (3NF[172]) if and only if it is in the 2NF{241} and each of its non-key attributes{21} is non-transitively{183} dependent on the primary key{36}.

Simplified: there should not be in the relation any attributes that are not part of the primary key and are transitively dependent on the primary key.



{Here is the so-called “canonical formulation” of the 3NF definition.}

A relation variable R is in the **third normal form** (3NF[173]) if and only if it is in the 2NF{241} and for every nontrivial{187} functional dependency {X} → A in this relation variable at least one of two conditions is satisfied:
a) {X} is a superkey{33} of the relation variable R;
b) A is a key attribute{21} of the relation variable R.

Simplified: if any attribute functionally depends on a set of other attributes, then at least one of two conditions must be satisfied:
a) this set of attributes uniquely identifies any record;
b) this dependent attribute is a part of the candidate key.

In contrast to the “weak” and “strong” formulations of the 2NF{241} definition, these definitions only show two views of the same situation, with the “simplified” formulation following from the “canonical” one. Let’s show that.

- Since (by definition{33}) any super-key uniquely identifies any table entry and (by definition{36}) a primary key uniquely identifies any table entry, one value of the primary key always corresponds to strictly one value of any super-key, i.e., (by definition{176}) there is a functional dependency {primary key} → {any super-key}. We can also recall that the primary key itself is a special case of a superkey.
- Since the dependency {primary key} → {super key} → A is redundant, the reasoning in the previous paragraph allows to understand the condition “a” as “{X} is the primary key” in the “canonical” formulation (which corresponds to the “simplified” formulation).
- Condition “b” in the “canonical” formulation can be expressed through negation, i.e., instead of “A is a key attribute” say “A is not a non-key attribute” (which also corresponds to the “simplified formulation”).

[172] A relation schema R is in **3NF** if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

[173] A relation schema R is in **3NF** if, whenever a nontrivial functional dependency X → A holds in R, either (a) X is a superkey of R, or (b) A is a prime attribute of R. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

Let's consider an example of violation of the third normal form (see figure 3.3.g). Here we see two cases corresponding to both formulations of the 3NF definitions:

- according to "simplified" definition it turns out that there is a transitive dependency {**g_id**} → {**g_faculty**} → {**g_dean**} → {**g_dean_dob**}, i.e., non-key attribute **g_faculty** (the faculty the group belongs to) depends on the primary key **g_id**, and the dean (**g_dean**) and all information related to the dean (in this case we limited ourselves to one field **g_dean_dob**, which stores the dean's birth date) depends on the faculty;
- according to the "canonical" definition, there is a transitive dependency {**g_number**, **g_start_year**} → {**g_faculty**} → {**g_dean**} → {**g_dean_dob**}, i.e., non-key attribute **g_faculty** (the faculty the group belongs to) depends on the alternate key {**g_number**, **g_start_year**}, and the dean (**g_dean**) and all information related to the dean (in this case we limited ourselves to one field **g_dean_dob**, which stores the date of birth of the dean) depends on the faculty.

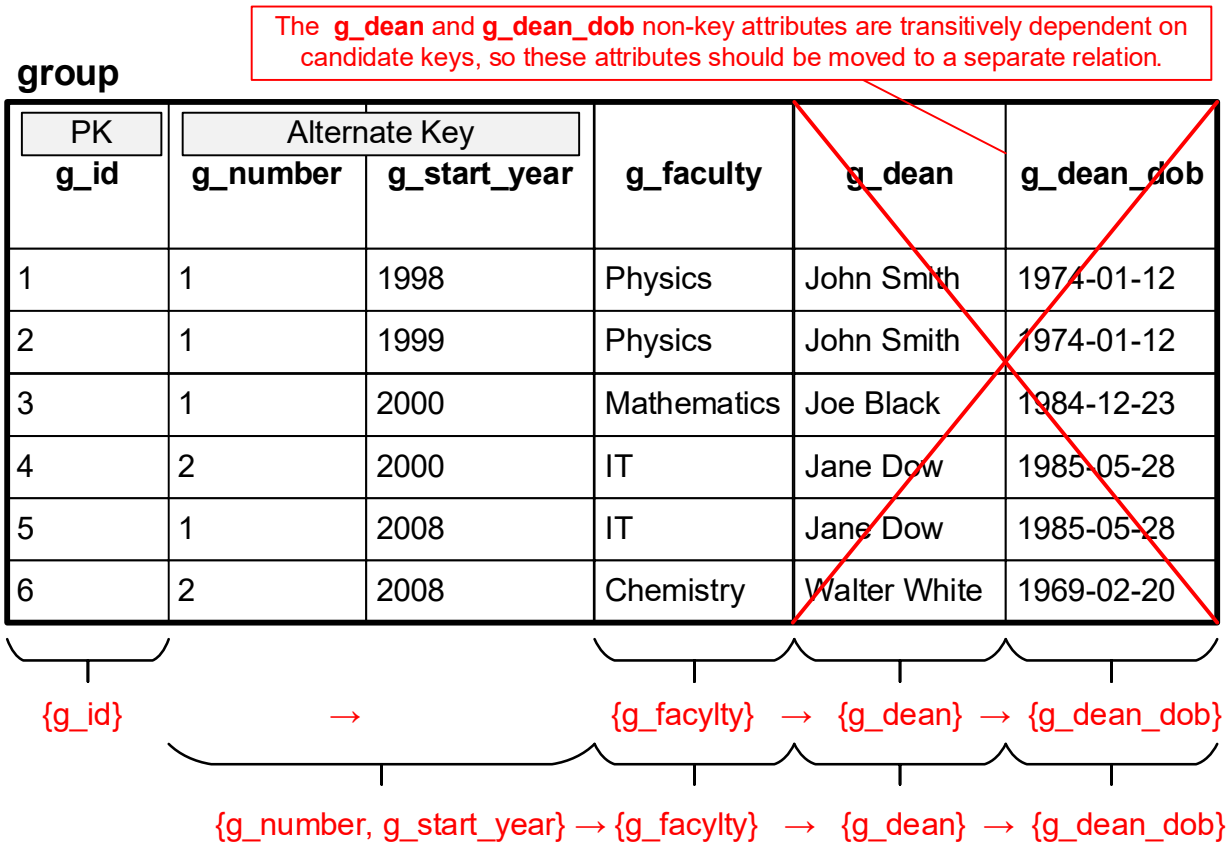


Figure 3.3.g — Violation of the third normal form

Violation of the third normal form is undesirable because it leads to meaningless duplication of stored information. Imagine that each faculty has 100 groups: then we will store the name of the dean and his date of birth 100 times for each faculty (and, in a very bad case, several dozen more data values characterizing the dean).

Besides the senseless memory cost, we also get a modification anomaly{157} because if any information about the dean is changed, it will have to be updated in the records of all the groups belonging to that faculty.



It is often asked what the fundamental difference between a violation of the 2NF (when discussing figure 3.3.e{243} we said that one can mistakenly assign the wrong length of education to a group) and a violation of the 3NF (nothing prevents one from assigning the wrong faculty to a group, for example) is.

The difference is that a violation of the 2NF allows us to deviate from some global rule of the subject area (i.e., "**any** group that started in such and such a year has been studying for so many years") and violation of the 3NF only allows us to mistakenly specify some individual data value without violating any global rule (yes, we can mistakenly assign some group to the wrong faculty, but we will not violate any rule like "any group meeting such and such conditions is studying in such faculty", because there is no such rule).

Let's consider bringing a relation to the third normal form (see figure 3.3.h).

We saw earlier that `g_dean` and `g_dean_dob` attributes are "superfluous" in the `group` relation and should be moved to a separate relation, but as we see in figure 3.3.h, two new relations were created instead of one — `faculty` and `staff`.

There are two reasons for this solution.

The first is trivial and very practical: it is logical to assume that the database will store information not only about faculty deans, but also about other university employees, only some of whom are deans. We would have nowhere to put information about "non-deans" if we did not separate descriptions of faculty and staff.

The second is semi-intuitive, but still worth mentioning. If we left the information about the deans with respect to faculty, we would again get a violation of the third normal form, because there would be a transitive dependency of the non-key attribute on the candidate key: {Faculty} → {Dean} → {Dean's date of birth}. But there is no such dependency with respect to `staff` because the employee's date of birth does not depend on their name.

In other words, the attribute containing the name of the dean plays a different role depending on which relation it is in. In `group` relation before normalization, the value of this attribute answered the question "Who is the dean of this faculty, who is this staff member?" and defined all information about the staff member (including date of birth). Regarding the `staff` the `s_id` attribute (primary key) answers the question "Who is this employee?" and it is this attribute that defines the date of birth and other parameters of the employee. And the `s_name` attribute answers only the question "What is the name of this employee?" and does not affect the date of birth: that is why there is no transitive dependency {`s_id`} → {`s_name`} → {`s_dob`}, but there are only two functional dependencies {`s_id`} → {`s_name`} and {`s_id`} → {`s_dob`}, whose existence does not contradict the 3NF definition.

Since this idea may not be obvious, but understanding it is important for proper bringing relations to the 3NF, it is shown graphically in figure 3.3.i.

Before normalization

group

PK	Alternate Key				
g_id	**g_number**	**g_start_year**	**g_faculty**	**g_dean**	**g_dean_dob**
1	1	1998	Physics	John Smith	1974-01-12
2	1	1999	Physics	John Smith	1974-01-12
3	1	2000	Mathematics	Joe Black	1984-12-23
4	2	2000	IT	Jane Dow	1985-05-28
5	1	2008	IT	Jane Dow	1985-05-28
6	2	2008	Chemistry	Walter White	1969-02-20

The **g_dean** and **g_dean_dob** non-key attributes are transitively dependent on candidate keys, so these attributes should be moved to a separate relation.

After normalization

group

PK	Alternate Key		FK
g_id	**g_number**	**g_start_year**	**g_faculty**
1	1	1998	1
2	1	1999	1
3	1	2000	2
4	2	2000	3
5	1	2008	3
6	2	2008	4

"1-M" relationship

Now, none of the relations presented here has a transitive dependency of non-key attributes on candidate keys.

faculty

PK	Alt. Key	FK
f_id	**f_name**	**f_dean**
1	Physics	127
2	Mathematics	216
3	IT	678
4	Chemistry	993

"1-M" relationship

staff

PK		
s_id	**s_name**	**s_dob**
127	John Smith	1974-01-12
216	Joe Black	1984-12-23
678	Jane Dow	1985-05-28
993	Walter White	1969-02-20

Figure 3.3.h — Bringing the relation to the third normal form

© EPAM Systems, 2020–2025

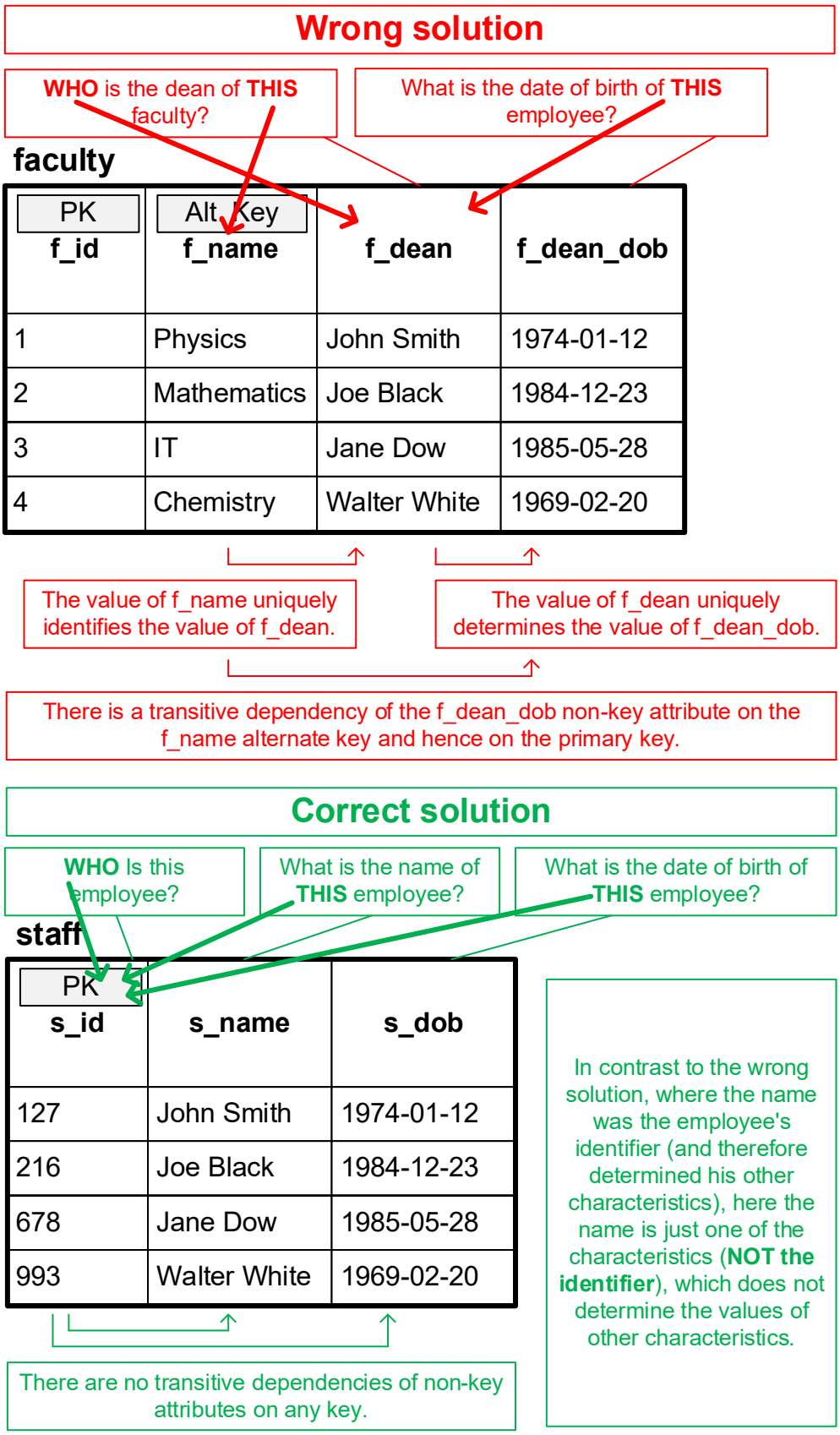


Figure 3.3.i — Changing the essence of an attribute depending on the relationship to which it belongs

A brief conclusion on the third normal form:

- the “simplified” and “canonical” formulations of the 3NF definition are equivalent, and either of them can be followed;
- the 3NF does not protect a relation from the possibility of mistakenly entering some local data (e.g., mistakenly entering a person’s passport number), but since the relation is already in the 2NF, “global rules” that apply to many records will be respected;
- a clear sign of violation of the 3NF is meaningless duplication of the same data in multiple rows of a table (then the attributes whose values are meaninglessly duplicated are the first candidates for moving to a new separate relation).



Task 3.3.d: are there any relation schemas in the “Bank{395}” database that are not in the third normal form? If you think “yes”, make appropriate changes in the schema to bring such relation schemas to the appropriate normal form.

3.3.4. Boyce-Codd Normal Form

Before reading the material in this section, it is worth recalling the definitions of transitive{183} and nontrivial{187} functional dependencies).



{Here is the so-called “simplified formulation” of the definition of BCNF.}

A relation variable is in the **Boyce-Codd normal form** (BCNF) if and only if it is in the 2NF{241} and each of its attributes is non-transitively{183} dependent on the primary key{36}.

Simplified: there must be no attributes in the relation that are transitively dependent on the primary key.



{Here is the so-called “canonical formulation” of the definition of BCNF.}

A relation variable R is in the **Boyce-Codd normal form** (BCNF[174]) if and only if it is in the 2NF{241} and for every nontrivial{187} functional dependency {X} → A in this relation variable {X} is a superkey{33}.

Simplified: if some attribute functionally depends on a set of other attributes, this set of attributes uniquely defines any record.

The difference between the BCNF and the 3NF is that the 3NF allows transitive dependency of key attributes on the superkey, while BCNF states that no such dependency should exist for any attribute (including the key attribute).

This difference can be clearly seen by placing the definitions side by side:

	3NF	BCNF
Simplified formulation	A relation variable is in the third normal form if and only if it is in the 2NF and each of its non-key attributes is non-transitively dependent on the primary key.	A relation variable is in the Boyce-Codd normal form if and only if it is in the 2NF and each of its ~~non-key~~ attributes is non-transitively dependent on the primary key.
Canonical formulation	A relation variable R is in the third normal form if and only if it is in the 2NF and for every nontrivial functional dependency {X} → A in this relation variable at least one of two conditions is satisfied: a) {X} is a superkey of the relation variable R; b) A is a key attribute of the relation variable R.	A relation variable R is in the Boyce-Codd normal form if and only if it is in the 2NF and for every nontrivial functional dependency {X} → A in this relation variable ~~at least one of two~~ condition~~s~~ is satisfied: a) {X} is a superkey of the relation variable R; ~~b) A is a key attribute of the relation variable R.~~

[174] A relation schema R is in **BCNF** if whenever a nontrivial functional dependency X → A holds in R, then X is a superkey of R. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

There are two other formulations of the BCNF definition that may be easier to understand and remember.



A relation variable is in the **Boyce-Codd normal form** (BCNF[175]) if and only if it is in the 3NF{247} and contains no overlapping candidate keys.

Simplified: the relation must be in the 3NF, and there must not be any candidate keys{34} that have common attributes.

A relation variable is in the **Boyce-Codd normal form** (BCNF[176]) if and only if the determinants{176} of all its functional dependencies are candidate keys.

Simplified: in any functional dependency {X} → A the set {X} must be a candidate key{34}.

Consider an example of the BCNF violation (see figure 3.3.j). The `gift` relation describes gift sets. In this case (within the subject area) there are conventions:

- no set may contain two or more objects of the same type;
- no set may contain two or more objects of the same name;
- only one variant of an item is allowed for each type of item.

The first convention allows us to consider a combination of {`g_set`, `g_type`} fields values as unique and declare a corresponding composite primary key.

The second convention allows us also to consider the combination of {`g_set`, `g_name`} fields values as unique, but since the primary key is already chosen, this combination will remain an alternate key. Yet, the `g_name` attribute is a key one (so the `gift` relation does not violate the 2NF{241}: yes, the `g_name` attribute depends on a part of the primary key, but it is a key attribute itself).

The third convention gives us the functional dependency {`g_type`} → {`g_name`}, but we see (figure 3.3.j) that the `g_type` attribute has no uniqueness property, i.e., the determinant of the functional dependency is not a candidate key, and therefore the `gift` relation is not in the Boyce-Codd normal form.

Violation of the BCNF is dangerous with the same set of anomalies as violation of the 2NF: for example, we can mix up the names of objects for the “Set 2” gift set and write “toy — marmalade, sweets — teddy bear”, and then in “Set 1” we also mix up something and write “sweets — oranges”. Besides mixing up the data here, we also get a violation of the just mentioned third convention, according to which for each type of item only one variant of the item is allowed, and we ended up with “sweets” being both “teddy bear” and “oranges”.

[175] A 3NF table that does not have multiple overlapping candidate keys is guaranteed to be in BCNF. “A Note on Relation Schemas Which Are in 3NF But Not in BCNF” (Vincent, M.W. and B. Srinivasan)

[176] Relvar R is in BCNF if and only if every FD that holds in R is implied by some superkey. (“The New Relational Database Dictionary”, C.J. Date)

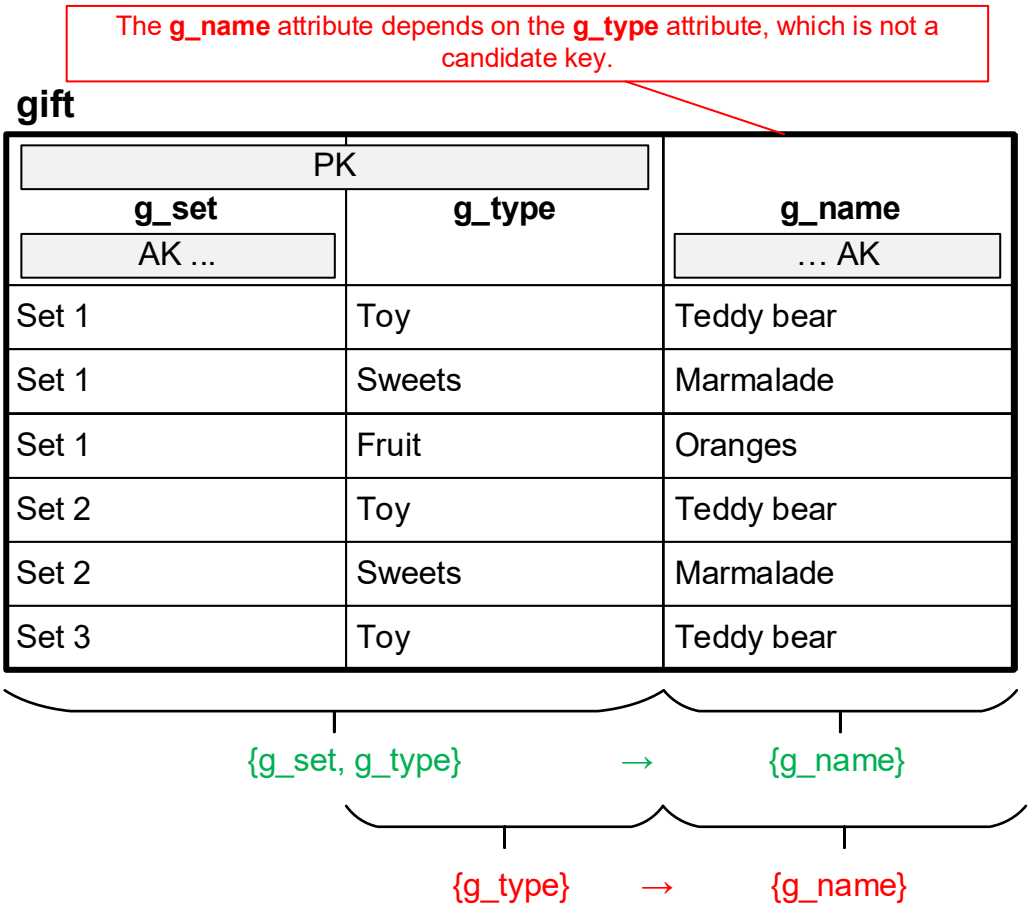


Figure 3.3.j — Violation of the Boyce-Codd normal form

To bring this relation to the BCNF, its attributes must be distributed into new relations.

Note that there can be complexity here, because not every distribution will then allow to get the original relation through the `JOIN` operation[177], but in the general case is enough to “cut” the original relation by the “problem” field (which is the determinant of the functional dependency and is not a candidate key; in our case it’s the `g_type` field).

The result of this transformation is shown in figure 3.3.k.

[177] See chapter 15.5 of the “Fundamentals of Database Systems” book (by Ramez Elmasri and Shamkant Navathe) for details.

Before normalization

gift

PK		
g_set AK ...	**g_type**	**g_name** … AK
Set 1	Toy	Teddy bear
Set 1	Sweets	Marmalade
Set 1	Fruit	Oranges
Set 2	Toy	Teddy bear
Set 2	Sweets	Marmalade
Set 3	Toy	Teddy bear

The **g_name** attribute depends on the **g_type** attribute, which is not a candidate key.

After normalization

set

PK	
s_name	**s_o_type** FK
Set 1	Toy
Set 1	Sweets
Set 1	Fruit
Set 2	Toy
Set 2	Sweets
Set 3	Toy

"1-M" relationship

object

PK	
o_type	**o_name**
Toy	Teddy bear
Sweets	Marmalade
Fruit	Oranges

Figure 3.3.k — Bringing the relation to the Boyce-Codd normal form

A brief conclusion on the Boyce-Codd normal form:

- all formulations of the BCNF definition are equivalent, and any of them can be followed;
- like the 3NF, the BCNF does not protect the relation from the possibility of specifying some local data mistakenly (for example, one can mistakenly write the name of a toy), but it allows us to reduce data duplication and protects the relation from violation of "global rules" that apply to many records and several attributes;
- a clear sign of the BCNF violation is the presence of overlapping candidate keys.



Task 3.3.e: are there any relation schemas in the “Bank{395}” database that are not in the Boyce-Codd normal form? If you think “yes,” make appropriate changes to the schema to bring such relation schemas to the appropriate normal form.

3.3.5. Fourth Normal Form

Before reading the material in this section, it is worth recalling the definitions of multivalued{189} dependency and its subspecies, trivial{190} and nontrivial{190}.



A relation variable R is in the **fourth normal form** (4NF[178]) if and only if for any of its nontrivial multivalued dependencies X ↠ Y the X set is a superkey{33} of R.

Simplified: the relation must be in BCNF {253} and it must not have nontrivial {190} multivalued dependencies.

A good way to explain the 4NF is the logic of deriving a simplified definition from a strict one.

Why should the relation be in the BCNF (see the simplified formulation of the definition)?

Let's start with how functional dependency{176} and multivalued dependency{189} are related. In fact, a functional dependency is a multivalued dependency in which the number of values of the function is limited to one, i.e., A ↠ B is equivalent to A → B when the cardinality of B is equal to one[179].

Taking into account the fact that (see the strict formulation of the definition) the multivalued dependency X ↠ Y must be nontrivial, the functional dependency X → Y must also be nontrivial{187}, i.e., Y cannot be a subset of X.

This is fully consistent with the definition of the BCNF{253}, which for this case can be formulated as follows: "a relation R is in the BCNF if and only if it is in the 2NF, and for every nontrivial functional dependency X → A in this relation X is a superkey of the relation R".

Why should there be no nontrivial multivalued dependencies in the relation (see the simplified formulation of the definition)?

Because (see the strict formulation of the definition) in the dependency X ↠ Y the X set must be a superkey, i.e., the relation cannot contain two or more tuples with the same value of the totality of X set attributes. And since a value of X can occur only once, it can be matched to at most one value of Y.

And this is just a case of degeneration of a nontrivial multivalued dependency into a trivial dependency (see figure 3.2.p{191}), i.e., nontrivial multivalued dependencies cannot exist within the 4NF (which is said in the simplified formulation of the 4NF definition).

The only thing left is to explore graphical examples.

Figure 3.3.I shows a case of violation of the fourth normal form. Thus, if the subject area allows each applicant to apply to more than one faculty, and each faculty has more than one admission exam, then there are dependencies {**ue_applicant**} ↠ {**ue_faculty**} and {**ue_faculty**} ↠ {**ue_exam**}, which can be written as {**ue_applicant**} ↠ {**ue_faculty**} | {**ue_exam**}.

[178] Relvar R is in **fourth normal form**, **4NF**, if and only if every MVD that holds in R is implied by some superkey of R — equivalently, if and only if for every nontrivial MVD X ↠ Y that holds in R, X is a superkey for R (in which case the MVD X ↠ Y effectively degenerates to the FD X → Y). ("The New Relational Database Dictionary", C.J. Date)

[179] See detailed explanation in chapter 13.2 of the "An Introduction to Database Systems (8th edition)" book (by C.J. Date).

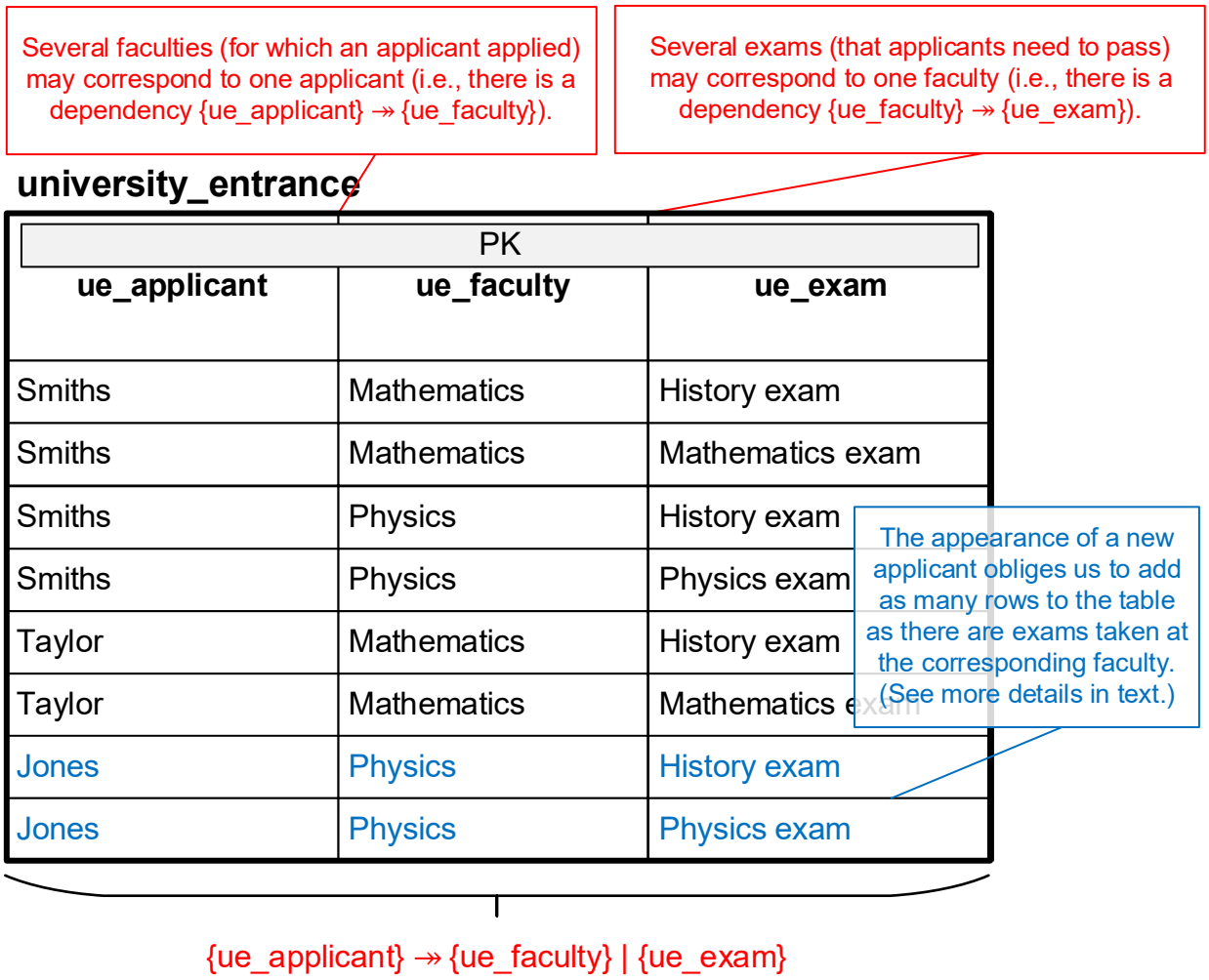


Figure 3.3.l — Violation of the fourth normal form

The `university_entrance` relation is in the BCNF (it has no non-key attributes or overlapping candidate keys), but it is not in the 4NF.

Violation of the 4NF is potentially fraught with a series of data operation anomalies[157]:

- when a new applicant appears, the record of them must be duplicated as many times as there will be a total number of exams in all the faculties for which they have applied;
- when a new exam appears in a certain faculty, we will have to add the corresponding entry as many times as there are applicants who want to enter that faculty;
- when adding a new faculty... we cannot add it, because no one has applied for it yet, and the `ue_applicant` is a part of the primary key;
- similar problems will occur when modifying and deleting data, i.e., the relation presented in figure 3.3.l is an excellent example of almost any data operation anomaly occurrence.

The logic of bringing the `university_entrance` relation to the 4NF is shown in figure 3.3.m.

Before normalization

university_entrance

PK		
ue_applicant	**ue_faculty**	**ue_exam**
Smiths	Mathematics	History exam
Smiths	Mathematics	Mathematics exam
Smiths	Physics	History exam
Smiths	Physics	Physics exam
Taylor	Mathematics	History exam
Taylor	Mathematics	Mathematics exam
Jones	Physics	History exam
Jones	Physics	Physics exam

Several faculties (for which an applicant applied) may correspond to one applicant (i.e., there is a dependency {ue_applicant} ↠ {ue_faculty}).

Several exams (that applicants need to pass) may correspond to one faculty (i.e., there is a dependency {ue_faculty} ↠ {ue_exam}).

After normalization

application

PK	
a_applicant	**a_faculty**
Smiths	Mathematics
Smiths	Physics
Taylor	Mathematics
Jones	Physics

exam

PK	
e_faculty	**e_exam**
Mathematics	History exam
Mathematics	Mathematics exam
Physics	History exam
Physics	Physics exam

Figure 3.3.m — Bringing the relation to the fourth normal form

The **`application`** and **`exam`** relations are in the 4NF because they have no non-key attributes, no overlapping candidate keys, and no “extra” attributes that could make up the Z set from the definition of a nontrivial multivalued dependency{190}.

However, this result of bringing the relation to the 4NF has a very unpleasant disadvantage: we cannot make any relationship between the **`application`** and **`exam`** relations because making a relationship would cause the primary key of the parent relation to migrate to the child one, which would give us the original **`university_entrance`** relation.

Therefore, from the practical implementation point of view, the final schema shown in figure 3.3.n is more advantageous.

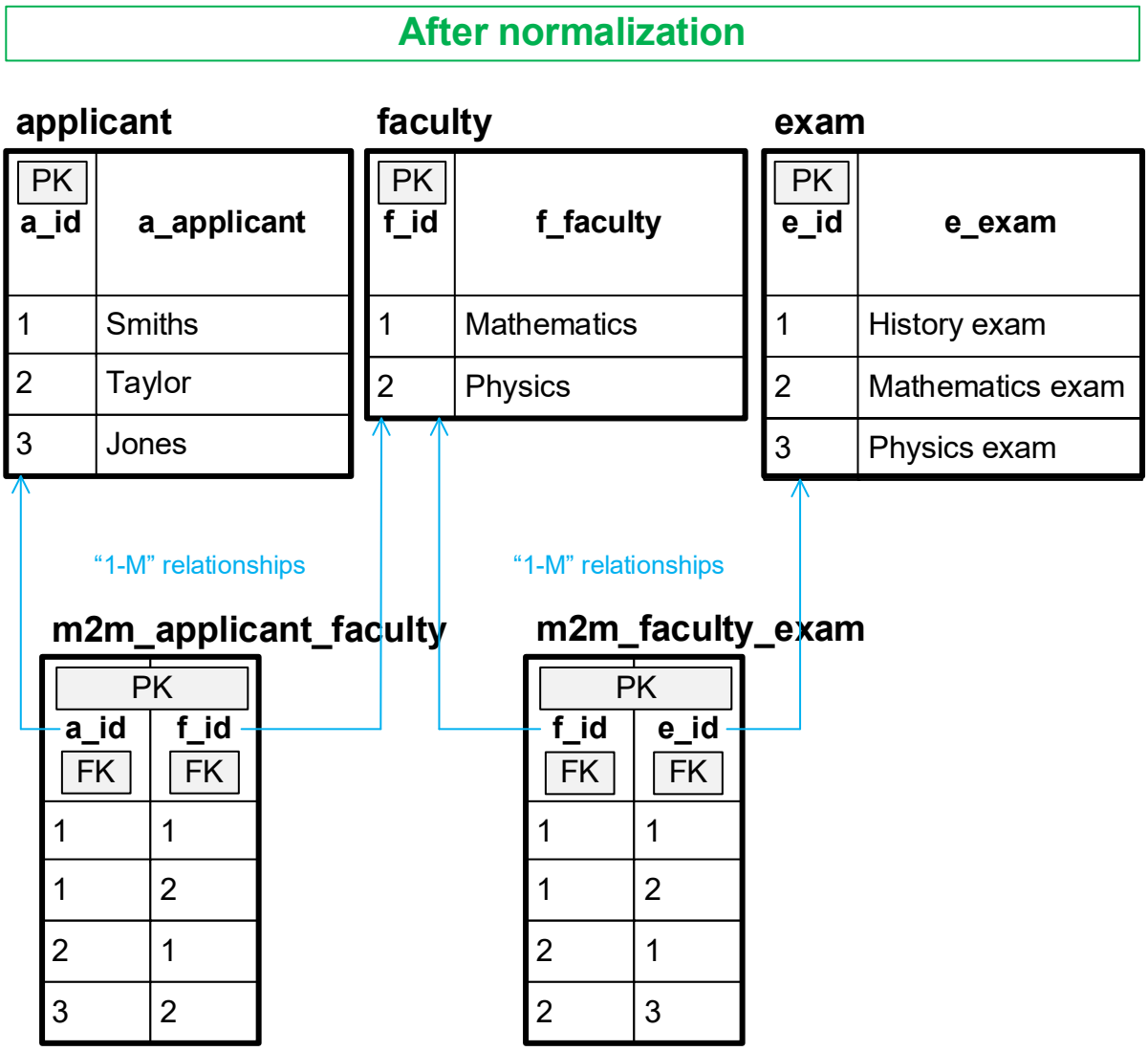


Figure 3.3.n — A more effective approach to bring the relation to the fourth normal form

Although the variant shown in figure 3.3.n contains considerably more relations (i.e., the DBMS will have to operate on more objects when performing `JOIN` operations), it still allows us to use the referential integrity {67} mechanism and therefore may be considered more preferable than the one shown in figure 3.3.m.

Brief conclusion on the fourth normal form:

- relations that are in the BCNF but are not in the 4NF are very rare case (in practice they almost never appear);
- like the BCNF, the 4NF does not prevent a relation from the possibility of entering some local data mistakenly (e.g., we can mistakenly write the name of the exam or assign it to the "wrong" faculty), but it reduces data duplication and prevents violations of "global rules" that apply to many records and several attributes;
- a clear sign of violation of the 4NF is the presence of a group of three or more attributes, in which attributes in pairs depend on each other, but at the same time do not depend on the values of other attributes.



Task 3.3.f: are there any relation schemas in the "Bank{395}" database that are not in the fourth normal form? If you think "yes", make appropriate changes to the schema to bring such relation schemas to the appropriate normal form.

3.3.6. Fifth Normal Form

Before reading the material in this section, it is worth recalling the definition of join dependency{192}.



A relation variable is in the **fifth normal form** (5NF[180, 181]) if and only if it is in the 4NF {258} and for each nontrivial join dependency{192} JD(R_1, R_2, …, R_n) each R_i attribute set is a superkey of the original relation variable.

Simplified: we need to know all the candidate keys of the relation and all the join dependencies that exist in it, and make sure that every projection that defines any of the join dependencies contains a candidate key.

If we return to the examples of the presence and absence of join dependency (see figures 3.2.r{194} and 3.2.s{196}), it is easy to notice that in real life there are very rarely subject area limitations such as the mentioned earlier: “If a tutor teaches some **S** subject, and this **S** subject is taught at some **F** faculty, and the tutor teaches at least one other subject at this **F** faculty, then he is obliged to teach that **S** subject at this **F** faculty”.

It is because of the rarity and difficulty of detecting such cumbersome and nontrivial rules in the subject area that normalization to the 5NF is said to almost never be performed.

And yet, let’s consider the signs of violation of the 5NF and the way to achieve it. The 5NF is violated if (all conditions must be fulfilled simultaneously):

- there are join dependencies{192} in the relation;
- for at least one such dependency there is a projection that does not include the candidate key of the relation.

Let’s continue with the relation presented earlier in figure 3.2.r{194}. It is in the 4NF since there are no multi-valued dependencies{189} here, but taking into account the rule just quoted (about the correlation of tutors, faculties and subjects) we can say that there is a JD((`w_tutor`, `w_subject`), (`w_subject`, `w_faculty`), (`w_tutor`, `w_faculty`)).

Thus, the first condition of violation of the 5NF is fulfilled — there are join dependencies in the relation.

The fulfillment of the second condition of the 5NF violation is even more obvious: none of the projections (`w_tutor`, `w_subject`), (`w_subject`, `w_faculty`), (`w_tutor`, `w_faculty`) contains candidate keys (which is easily checked by the data presented in figure 3.3.o).

[180] The 5NF is also known as the “project-join normal form” (PJNF).

[181] A relation schema R is in fifth normal form (5NF) (or project-join normal form (PJNF)) with respect to a set F of functional, multi-valued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, ..., R_n) in F^+ (that is, implied by F), every R_i is a superkey of R. (“Fundamentals of Database Systems”, Ramez Elmasri, Shamkant Navathe)

workload

PK		
w_tutor	**w_subject**	**w_faculty**
Smith	Mathematics	Exact sciences
Smith	Informatics	Natural sciences
Taylor	Mathematics	Natural sciences
Jones	Informatics	Cybernetics
Taylor	Physics	Exact sciences
Taylor	Mathematics	Exact sciences
Smith	Mathematics	Natural sciences

There is a join dependency:
JD((w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty))

Non of (w_tutor, w_subject), (w_subject, w_faculty), (w_tutor, w_faculty) projections contains a candidate key.

Figure 3.3.o — Violation of the fifth normal form

The violation of the 5NF, the existence of which we have just proved, indicates that this relation can be decomposed{176} losslessly into several separate relations. The result of such decomposition is shown in figure 3.3.p.

Before normalization

workload

PK		
w_tutor	**w_subject**	**w_faculty**
Smith	Mathematics	Exact sciences
Smith	Informatics	Natural sciences
Taylor	Mathematics	Natural sciences
Jones	Informatics	Cybernetics
Taylor	Physics	Exact sciences
Taylor	Mathematics	Exact sciences
Smith	Mathematics	Natural sciences

There is a join dependency:
JD((w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty))

Non of (w_tutor, w_subject),
(w_subject, w_faculty),
(w_tutor, w_faculty)
projections contains a
candidate key.

After normalization

workload_ts

PK	
w_tutor	**w_subject**
Smith	Mathematics
Smith	Informatics
Taylor	Mathematics
Jones	Informatics
Taylor	Physics

workload_sf

PK	
w_subject	**w_faculty**
Mathematics	Exact sciences
Informatics	Natural sciences
Mathematics	Natural sciences
Informatics	Cybernetics
Physics	Exact sciences

workload_tf

PK	
w_tutor	**w_faculty**
Smith	Exact sciences
Smith	Natural sciences
Taylor	Natural sciences
Jones	Cybernetics
Taylor	Exact sciences

Figure 3.3.p — Bringing the relation to the fifth normal form

As in the case of the 4NF{258} we got a variant that is correct from the relational theory point of view, but is extremely inconvenient in practice: we cannot make any relationship between the relations obtained during normalization, because making a relationship would lead to the migration of the primary key of the parent relation to the child one, which would give us the original `workload` relation.

Therefore, from the practical implementation point of view, the final schema shown in figure 3.3.q is more advantageous.

After normalizarion

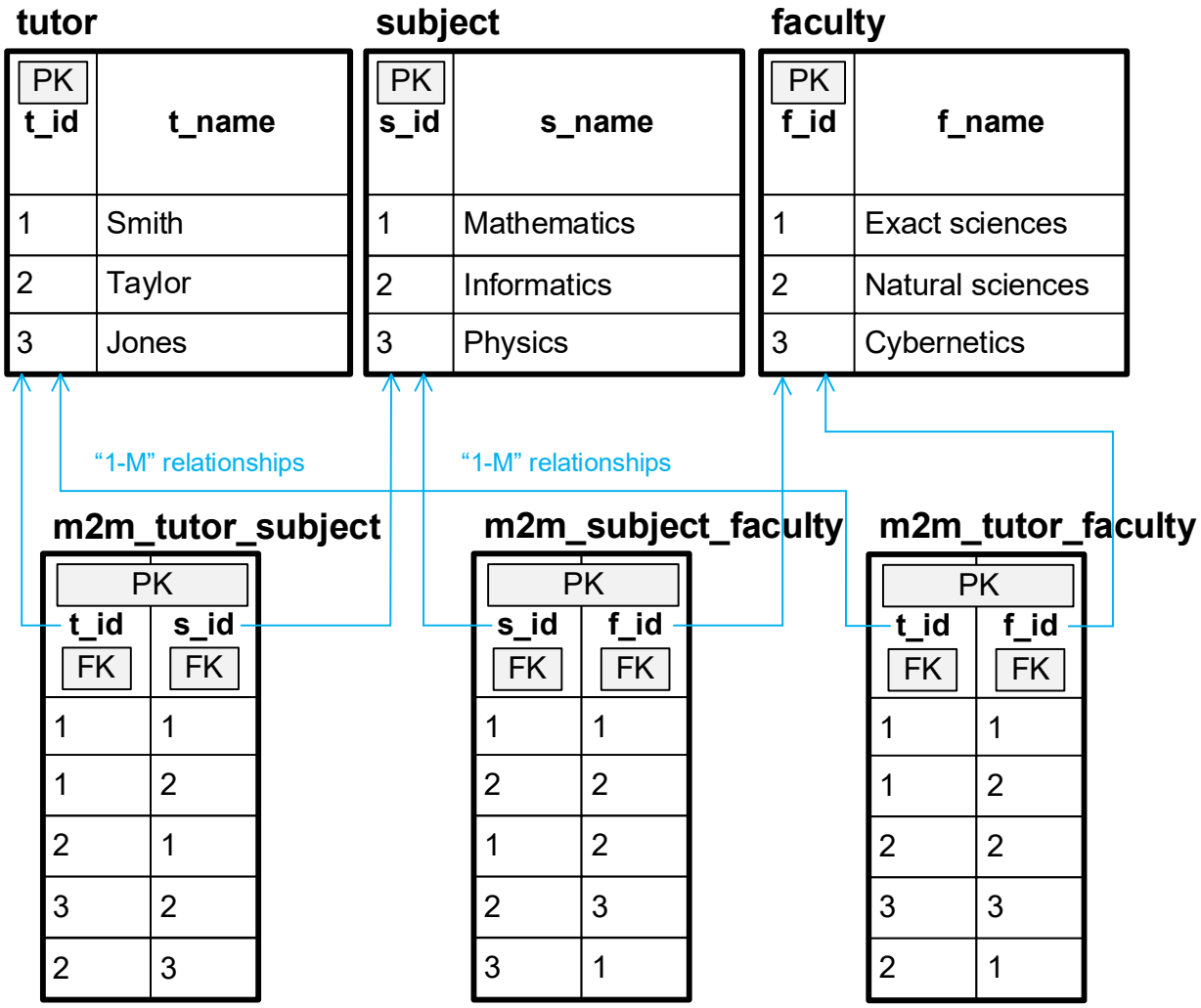


Figure 3.3.q — A more effective approach to bring the relation to the fifth normal form

The variant shown in figure 3.3.q contains considerably more relations (i.e., the DBMS will have to operate on more objects when performing `JOIN` operations), but it allows us to use the referential integrity{67} mechanism, and therefore may be considered preferable to the one shown in figure 3.3.p. Also, this option allows more effective use of triggers{341} to enforce business rules about the correlation between tutors, subjects, and faculties.

We conclude this chapter by noting that if the subject area did not have a rule about the correlation between tutors, subjects, and faculties, there would be no join dependency in `workload` relation and it would have been in the 5NF from the beginning.

A brief conclusion on the fifth normal form:

- relations that are in the 4NF but are not in the 5NF are very rare (in practice they almost never occur);
- like the 4NF, the 5NF does not prevent the relation from the possibility of entering some local data mistakenly (e.g., we can mistakenly write the name of the department or assign it to the “wrong” subject), but it reduces data duplication and protects the relation from violations of “global rules” that apply to many records and several attributes;
- a sign of violation of the 5NF is the presence of a group of three or more attributes, in which all of the attributes included in this group are dependent on each other;
- the 5NF is the “final” normal form, i.e., a relation in the 5NF can no longer be decomposed losslessly (with the exception of “temporal relations” — see 6NF{267}).



Task 3.3.g: are there any relation schemas in the “Bank{395}” database that are not in the fifth normal form? If you think “yes”, make appropriate changes to the schema to bring such relation schemas to the appropriate normal form.

3.3.7. Domain-Key Normal Form

Before reading the material in this section, it is worth recalling the definitions of the domain{197} and key{197} dependencies.



A relation variable is in **domain-key normal form** (DKNF[182, 183]) if and only if all constraints and dependencies existing in this relation variable are consequences of domain{20} and key{32} constraints.

Simplified: all rules that apply to a relation must follow directly from the properties of its fields (and groups of fields), and there must not be any "hidden" rules that follow from anything else.

The DKNF is both one of the simplest and most complex normal forms. Its simplicity follows directly from the definition (and, as shown earlier, from the definitions of the domain{197} and key{197} dependencies).

Indeed, as it seems, there is nothing easier than to ensure the uniqueness of the relation keys and to control that any value of any attribute belongs to the appropriate domain — modern DBMS provide the widest range of technical possibilities for that, but…

But it only seems to be.

Remember that the definition of key dependency{197} refers to a subject area, but **not** to a technical implementation. Thus, for any table with a surrogate{38} primary key, the question of finding candidate keys and guaranteeing the uniqueness of their values for each record remains open (which on the one hand casts doubt on the usage of a surrogate key, but on the other hand is not easy in real situations where candidate keys{34} may not always be evident).

As for the domain dependency{197}, it is also simple only in theory. Yes, it is easy to define a range of values, e.g., for days of the week or a range of values for a person's height. But try solving the same problems for domains like "name", "address", "product description" and the like — we will always be forced to compromise between "create a restriction too strong, not allowing some valid values" and "create a restriction too weak, allowing some invalid values".

And this is just the "top of the iceberg". By the definition of the DKNF, there should be no "hidden rules" in a relation variable. But what if there are, e.g., such rules in the subject area:

- the new employee's salary cannot be less than the old one;
- a vacation should not start on Friday;
- green birds should get one-third more food than yellow birds;
- it is forbidden to give a discount on the sale of white cars;
- there must be at least three users with administrator permissions.

All these rules (and thousands like them) do not follow neither from the uniqueness of the keys, nor from the set of values of any field at all.

[182] A relation schema is said to be in **DKNF** if all constraints and dependencies that should hold on the valid relation states can be enforced simply by enforcing the domain constraints and key constraints on the relation. The idea behind **DKNF** is to specify (theoretically, at least) the *ultimate normal form* that takes into account all possible types of dependencies and constraints. ("Fundamentals of Database Systems", Ramez Elmasri, Shamkant Navathe)

[183] A 1NF relation schema is in **DK/NF** if every constraint can be inferred by simply knowing the DDs (domain dependencies) and the KDs (key dependencies). Putting it another way: a 1NF relation schema is in **DK/NF** if by enforcing the DDs and KDs, every constraint of the schema is automatically enforced. ("A Normal Form for Relational Databases That Is Based on Domains and Keys", Ronald Fagin)

That is why the DKNF on the one hand represents a very neat mathematical solution[184], and indeed the relation variable that is in the DKNF is also automatically in all the normal forms preceding{274} it (assuming that the domain cardinality is infinite).

But on the other hand, the DKNF is so difficult to achieve in practice that its existence is of more scientific than practical interest — and the problem is not in the DKNF itself, but in the complexity of subject areas and their rules, which by the very fact of their existence contradict the definition of the DKNF.

However, if the relation is brought to the DKNF "forcibly" (i.e., with normalization for the sake of normalization), most likely, you will have to say goodbye to the opportunity to fulfill the key database requirements{10}.

There is another interesting fact: the DKNF is the only normal form that is not a part of the hierarchy of normal forms, but more about this will be discussed in the corresponding section{274}.

And yet, in spite of all the strangeness of the DKNF, let's consider a small example[185]. The **`employee`** relation (see figure 3.3.r) describes employees and contains two fields — **`name`** and **`role`**. The **`role`** field can store data both about the role in the sense of "manager", "assistant manager", etc. (actually, the employee's position), and about a role in the sense of "programmer", "architect", "tester" etc. (actually, the employee's function), and we know that an employee can have only one value of "position" property, but there could be any number of values of "function" property.



If just now you thought that this is some kind of madness, and that in reality "position" and "function" would be in different fields (and even different relations), you are absolutely right. But let's emphasize that this is just a clear example of bringing the relation to the DKNF, and from a purely formal point of view there are no inconsistencies here.

This relation is in the 5NF{262} (all its possible projections will contain the superkey of the original relation, though it is even simpler here: all projections of this relation will be the original relation), but the DKNF is violated here: there is a restriction in the subject area ("an employee can only have one position") that does not follow directly from either the domain{197} or the key{197} dependencies.

By the way, formally, nothing prevents us from inserting a record about the same employee, but with a different value of the "position" property, i.e., from violating a subject area requirement.

To bring this relation to the DKNF, we need to break it into two separate relations, the first of which will store information about the position of an employee, and the second — about their functions.

Interesting fact: we are not even talking here about projections of the original relation — the new relations have their own separate attributes that were not present in the original one.

[184] Ronald Fagin devotes about twenty pages to formulas and proofs in his article "A Normal Form for Relational Databases That Is Based on Domains and Keys".

[185] This example is borrowed from the "A Normal Form for Relational Databases That Is Based on Domains and Keys" article (by Ronald Fagin).

Before normalization

employee

PK	
e_name	**e_role**
Smith	**Coordinator**
Smith	Programmer
Smith	Analyst
Taylor	Manager
Taylor	Programmer
Taylor	Architect
Smith	**Manager**

The possibility to insert such data violates the subject area requirement “each employee should have only one position”, and this rule itself does not follow in any way from the restrictions of keys and / or domains.

After normalization

employee_position

PK	
ep_name	**ep_position**
Smith	Coordinator
Taylor	Manager

employee_function

PK	
ef_name FK	**ef_function**
Smith	Programmer
Smith	Analyst
Taylor	Programmer
Taylor	Architect

Now the subject area requirement “each employee should have only one position” follows from the key constraint.

Figure 3.3.r — Bringing the relation to the domain-key normal form

A brief conclusion on the domain-key normal form:

- not every relation can be brought to the DKNF;
- if the relation is in the DKNF, it is also in the 5NF;
- a sign of violation of the DKNF (or even the impossibility of bringing the relationship to the DKNF) is the presence in the subject area of such rules, which do not follow directly from the sets of attribute values of the relation and / or the property of uniqueness of the keys;
- bringing a relation to the DKNF is usually performed not by decomposing the relation, but by creating new relations with new attributes (such attributes that did not exist in the original relation).



Task 3.3.h: are there any relation schemas in the “Bank{395}” database that are not in the domain-key normal form? If you think “yes”, make appropriate changes to the schema to bring such relation schemas to the appropriate normal form.

3.3.8. Sixth Normal Form

Before reading the material in this section, it is worth recalling the definition of generalized join dependency{200}.



A relation variable is in the **sixth normal form** (6NF[186]) if and only if it admits in principle no lossless decomposition{176}, i.e., any join dependencies{192}, {200} present in it are trivial{192}, {200}.

Simplified: when decomposing such a relation variable, at least one of the resulting projections will always be equivalent to the original relation variable.

Figure 3.3.s shows the same situation that was an illustration of the generalized join dependency{200}, but now we will look at the relations presented from a slightly different perspective.

The original `education_path` relation contains information about which faculty and group a student was in at what period of time. However, it is obvious[187] that the faculty and group number can change independently of each other.

This leads to very nontrivial operations to keep the relation in a consistent state. Try to determine by yourself how the relation will change if the student with identifier 13452:

- was studying in group 3 in February 2018 (but we just forgot to specify it and now we want to correct the situation);
- went on academic leave for a month in August 2018, and therefore continued to be listed at the Faculty of Chemistry, but was not in any group;
- has changed groups every month since 2019, remaining in the same faculty.

Some of these operations are relatively simple and “just” will lead to meaningless duplication of data. But there are operations here that will require a very complicated change of relation.

The situation would be much simpler if we could store information about a student’s time periods in some faculty and in some group independently.

This is exactly what is achieved by decomposing the `education_path` relation into P_1 and P_2 projections, and no information is lost as a result of this decomposition: we can still find out, for any point in time, at which faculty and in which group the student with ID 13452 was studying.

Further decomposition of P_1 and P_2 projections is impossible, because we cannot “tear apart” their primary keys (we will lose the information “which student at what period of time is the point of discussion”) or “tear off” the non-key field from the primary key (the faculty name or group number themselves without reference to the student and time period are meaningless), so for both projections we have only this decomposition option: separately save the relation with the primary key values and separately... save the original relation.

Thus, the (generalized) join dependencies available in the P_1 and P_2 projections are trivial, i.e., both the P_1 and P_2 relations contain no nontrivial (generalized) join dependencies, i.e., they cannot be decomposed in any lossless way, i.e., they are in the 6NF.

[186] Relvar R is in sixth normal form, 6NF, if and only if it can’t be nonloss decomposed at all, other than trivially — i.e., if and only if the only JDs to which it’s subject are trivial ones. Equivalently, relvar R is in 6NF if and only if it’s in 5NF, is of degree n, and has no key of degree less than n-1. (“The New Relational Database Dictionary”, C.J. Date)

[187] Yes, such words are often infuriating ☺, but... I assure you, it REALLY is obvious.

Before normalization

education_path

PK			
ep_student	**ep_period**	**ep_faculty**	**ep_group**
13452	01.01.2018-28.02.2018	Chemistry	1
13452	01.03.2018-31.05.2018	Chemistry	5
13452	01.06.2018-01.12.2018	Physics	5

The faculty and the group number can independently change within any time interval, which obviously leads to a simple idea: we can store the time intervals for the faculty and the group number separately, i.e., we can decompose the relation into projections that are in a higher normal form.

After normalization

P_1 (ep_student, ep_period, ep_faculty)

PK		
ep_student	**ep_period**	**ep_faculty**
13452	01.01.2018-31.05.2018	Chemistry
13452	01.06.2018-01.12.2018	Physics

P_2 (ep_student, ep_period, ep_group)

PK		
ep_student	**ep_period**	**ep_group**
13452	01.01.2018-28.02.2018	1
13452	01.03.2018-01.12.2018	5

Now both obtained relations store (each separately) information about the period of time a student studied at which faculty, and in what period of time a student was in which group.

Figure 3.3.s — Bringing the relation to the sixth normal form

Pay attention to the second part of the definition[186], which says that a relation variable of arity *n* is in the 6NF if it is in the 5NF and does not contain keys of arity less than *n-1*.

Let's illustrate this with an example of the same relations shown in figure 3.3.s.

The `education_path` relation has an arity of 4 (it has 4 attributes) and is in the 5NF, since all of its valid projections contain its own primary key.

However, the primary key in that relation has arity of 2, i.e., 2 < (4-1), and by the definition of 6NF there should be no such keys in the relation.

In turn, the P_1 and P_2 projections have the arity of 3, and their keys have the arity of 2. So, the condition 2 < (3-1) is not fulfilled, i.e., these relations have no keys with arity less than “arity of relation -1”. According to this feature, they are in the 6NF.

A brief conclusion on the sixth normal form:

- the 6NF is relevant only for temporal relations, for all others the final normal form is the 5NF{262};
- the 6NF allows us to greatly simplify operations with the relation caused by the need to separately manage the values of attributes associated with time intervals, but not related to each other;
- a sign of violation of the 6NF is the presence of two or more attributes, unrelated to each other, but related to the same temporal attribute;
- the 6NF is the “completely final” normal form, i.e., any relation in the 6NF can no longer be decomposed further losslessly.



Task 3.3.i: are there any relation schemas in the “Bank{395}” database that are not in the sixth normal form? If you think “yes”, make appropriate changes to the schema to bring such relation schemas to the appropriate normal form.

3.3.9. Normal Forms Hierarchy and Non-Canonical Normal Forms

By analogy with the way a brief list of all basic dependencies{204} was presented earlier, let's give the same list of all basic normal forms.



Please use the definitions below only as a way of quickly remembering the essence of a particular normal form. Full, rigorous definitions can be found quickly by following the link provided next to each term.

A relation variable is in the **zero normal form** (0NF{219}) if no special requirements are imposed on it, and any violation of any normal forms is allowed.

A relation variable is in the **first normal form** (1NF{236}) if each of its attributes is atomic (i.e., the DBMS does not have to operate on any individual part of any attribute).

A relation variable is in the **second normal form** (2NF{241}) if it is in the 1NF, and no attribute that is not a part of the primary key is functionally dependent on any part of the primary key.

A relation variable is in the **third normal form** (3NF{247}) if it is in the 2NF and does not contain non-key attributes that are transitively dependent on the primary key.

A relation variable is in the **Boyce-Codd normal form** (BCNF{253}) if it is in the 2NF and does not contain attributes that are transitively dependent on the primary key.

To remember the difference between the 3NF and the BCNF:

3NF	BCNF
A relation variable is in the 3NF if it is in the 2NF and does not contain non-key attributes that are transitively dependent on the primary key.	A relation variable is in the BCNF if it is in the 2NF and does not contain ~~non-key~~ attributes that are transitively dependent on the primary key.

A relation variable is in the **fourth normal form** (4NF{258}) if it is in the BCNF and does not contain any nontrivial multivalued dependencies.

A relation variable is in the **fifth normal form** (5NF{262}) if it is in the 4NF and each of its projections defining any of the join dependencies contains a candidate key.

A relation variable is in the **domain-key normal form** (DKNF{267}) if all the rules that are applicable to it follow directly from the properties of its fields (and / or groups of fields), and there are no "hidden" rules derived from anything else.

A relation variable is in the **sixth normal form** (6NF{271}) if it allows in principle no lossless decomposition, i.e., when decomposing such a relation variable into projections at least one of the resulting projections will always be equivalent to the original relation variable.

Schematically, the whole hierarchy of normal forms can be represented by a kind of pyramid (see figure 3.3.t), in which the normal forms we discussed earlier are marked in **bold**.

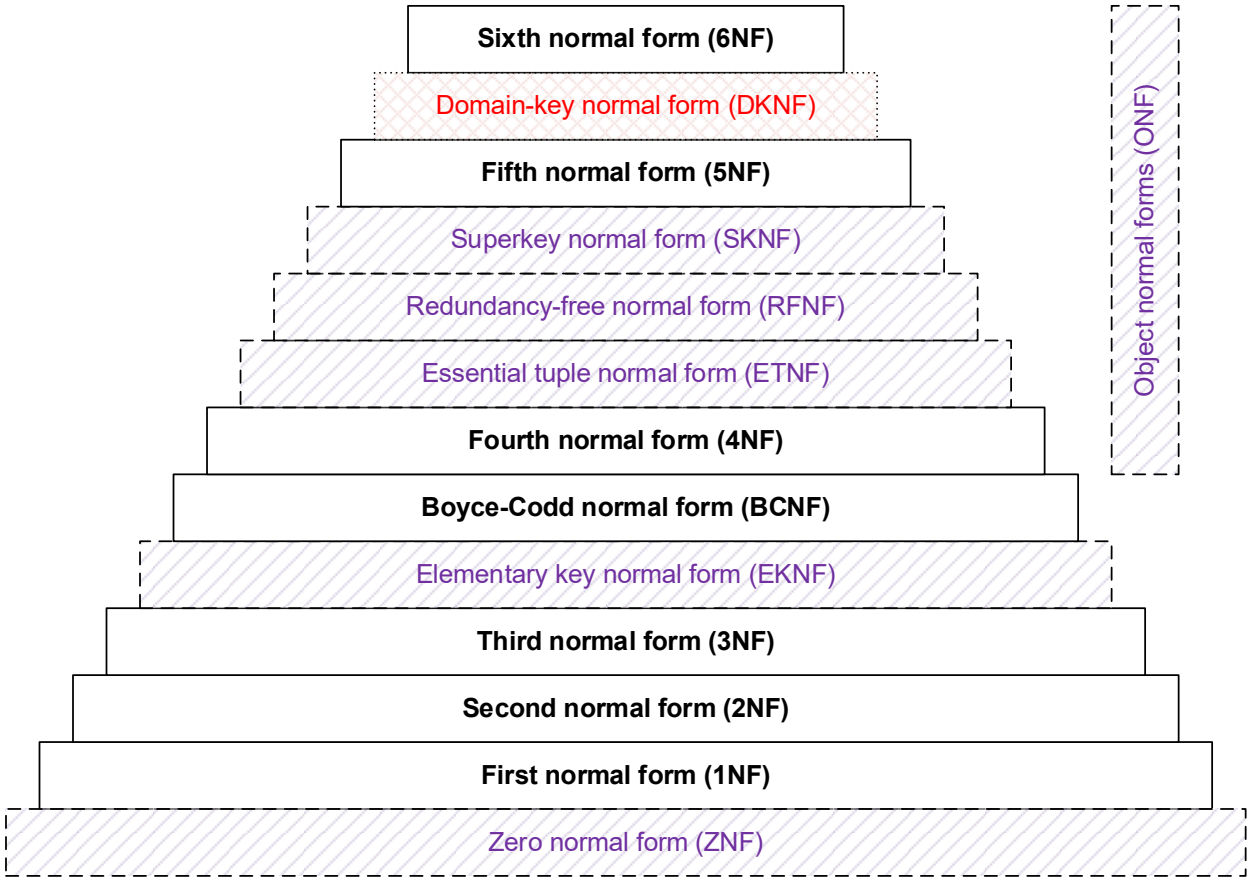


Figure 3.3.t — Hierarchy of normal forms

For the other normal forms, we will limit ourselves to very brief definitions (without explanations) and references to primary sources.

There are several answers to the legitimate question of why all these “non-canonical” normal forms are needed at all:

- some of them were historically formulated earlier than the canonical normal forms that later “replaced” them;
- some of them can be regarded as links between neighboring canonical normal forms;
- some of them show certain aspects of relational theory from a different angle, allowing a better and deeper understanding of the issues at hand.

So…

The zero normal form (ZNF) describes “a relation variable that is not even in 1NF”, i.e., in such a relation variable all imaginable and unimaginable violations of any rules and any common sense are allowed.

Elementary key normal form (EKNF) (in contrast to 3NF{247}, see “canonical formulation”) requires the “attribute A” appearing in the definition to be not just a key but an elementary key, i.e., it should be an irreducible determinant of its non-trivial functional dependency.

Essential tuple normal form (ETNF) requires any allowable value of a relation variable to contain only such tuples that deleting at least one of them would make it impossible to obtain the same information from the remaining tuples as it had before deletion.

Redundancy-free normal form (RFNF) requires any allowable value of a relation variable to follow the rule: all possible projections of this relation variable contain the superkeys of the original relation, and the set of these superkeys constitutes the header of the original relation.

Superkey normal form (SKNF) requires any component of any irreducible join dependency to be the superkey of the original relation variable.

The domain-key normal form deserves special attention as this form, strictly speaking, is not a part of the general hierarchy. Yes, logically it is convenient to place it between the 5NF and the 6NF (as is done in figure 3.3.t), but unlike all other normal forms that "are based on previous ones", the DKNF is not "based" on any form (nor does the 6NF follows from the DKNF).

Moreover, the author of the DKNF shows[188] that normal forms from 1 to 5 can themselves be a derivative of the DKNF.

And absolutely separate mention should be made of object normal forms (relevant to object-relational databases). There are the 4ONF (4th object normal form), etc. (that is why in figure 3.3.t the corresponding block starts exactly at the 4NF level), but all of them are beyond the scope of this book.



If you still want to learn more about non-canonical normal forms, you should first consult the following sources:

- "The New Relational Database Dictionary" (by C.J. Date) (this is where the general concept of the hierarchy of normal forms and most of the definitions are taken from);
- "Database Normalization Complete" (by M. Jason) (an excellent concise guide to all imaginable and unimaginable normal forms).

As another way to remember the hierarchy of normal forms, here is the following table[189]:

[188] "A Normal Form for Relational Databases That Is Based on Domains and Keys", Ronald Fagin.

[189] The original idea is taken from here: https://en.wikipedia.org/wiki/Database_normalization#Normal_forms

	0NF	1NF	2NF	3NF	EKNF	BCNF	4NF	ETNF	5NF	DKNF	6NF
There is a primary key	?	+	+	+	+	+	+	+	+	+	+
There are no multi-valued attributes	?	+	+	+	+	+	+	+	+	+	+
Each attribute is atomic	-	+	+	+	+	+	+	+	+	+	+
Any non-key attribute is functionally fully dependent on the primary key	-	-	+	+	+	+	+	+	+	+	+
Any non-key attribute is non-transitively dependent on the primary key	-	-	-	+	+	+	+	+	+	+	+
Any dependent attribute is part of an irreducible key	-	-	-	-	+	+	+	+	+	+	+
Any attribute is nontransitively dependent on the primary key	-	-	-	-	-	+	+	+	+	+	+
There are no nontrivial multi-valued dependencies	-	-	-	-	-	-	+	+	+	+	+
Any join dependency is built on a superkey	-	-	-	-	-	-	-	+	+	+	+
Any nontrivial join dependency is built on a candidate key	-	-	-	-	-	-	-	-	+	+	+
Any constraint is generated by domain and key constraints	-	-	-	-	-	-	-	-	-	+	+
Any join dependency is trivial	-	-	-	-	-	-	-	-	-	-	+

We can say that at this point we have considered all the theory necessary to perform the database design, to which the next section will be devoted.



Task 3.3.j: which of the normalization operations you performed in tasks 3.3.a–3.3.i were really necessary, and which were merely educational and would have been more harmful to the database than useful? Justify your opinion.

Chapter 4: Database Design

4.1. Design at the Infological (Conceptual) Level

4.1.1. Design Goals and Objectives at the Infological (Conceptual) Level

Before reading the material in this chapter, it is worth reviewing the basics of database modeling{7}.

As stated in the definition of infological (conceptual{8}) modeling level, its main purpose is to create a schema that reflects the entities of the subject area, their attributes, and relationships (perhaps not all yet) between entities.

I.e., it is necessary to investigate the subject area as deeply as possible and to express its concepts in the form of entities, attributes, relationships.

The study of the subject area is just the main complexity, which is divided into the following local problems:

- information extraction;
- research depth;
- research boundaries.

Information extraction is a general problem for any project (even outside the context of databases). As a rule[190], the customer does not have a complete technical description of the project, so it is necessary to organize the collection of requirements for the project and to identify those requirements that directly or indirectly relate to the database.

There are many techniques for identifying requirements (interviews, focus groups, questionnaires, workshops and brainstorming, observation, prototyping, document analysis, processes and interactions modeling, and so on, and so forth.[191])

Regardless of the chosen approach, the process will be long, iterative, and focused on the main task: to collect the most correct information in its entirety. And this leads us to the following local problem.

Research depth is the biggest challenge for novice planners, which is especially evident in the example of students' works. It is not enough to identify only *some* entities and *some* of their attributes. It is necessary to identify *all* of them.

Figuratively speaking, a real library database cannot consist of two tables like "books" (with only "title" and "year of publication" fields) and "authors" (with only "full name" and "year of birth" fields) — such a database will have dozens of tables with dozens of fields.

The problem is becoming more complicated by the fact that, in general, it is not the customer's task to think through all the possible details, nuances, and aspects. It is the task of the database designer. And the task is very serious, because the failure in its solution is almost guaranteed to lead to inadequacy of the database to the subject area{11}.

On the other hand, there is a danger of overstepping the boundaries of the project and encountering the following problem.

[190] In fact, always ☺.

[191] See the chapter "2.2.3. Ways of requirements gathering" in the "Software Testing. Base Course" book (Svyatoslav Kulikov) [https://svyatoslav.biz/software_testing_book/]

Research boundaries are often a managerial rather than a technical problem, but this does not make it any easier: at the insistence of the customer or for technical reasons, the database may begin to include entities that exist in reality and are related to the subject area but are not relevant to the project being implemented.

So, continuing the example of the library, we can go from the library database itself to logistics (books somehow get to the library), storage management (the books are stored somewhere), accounting (because financial issues must be addressed somehow), personnel management (because the library employs staff), etc.

To avoid this effect (called “boundary blurring”), special techniques outside the scope of this book are used, but it is always worth keeping in mind what database we are designing and raise the question of whether we have exceeded the boundaries of the project, if there is any doubt that certain entities of the real world are relevant to the database we are designing.

The list of tasks of this level of design can also be safely added by ensuring that the database meets key requirements{10}: and although they cannot be achieved by design at the infological (conceptual) level alone, it is here that the foundation is laid for many decisions that will be taken at subsequent levels and will create a truly effective database.



Task 4.1.a: formulate a list of questions, the answers to which would help you improve the “Bank{395}” database schema.

4.1.2. Design Tools and Techniques at the Infological (Conceptual) Level

After reading the previous section, one might get the impression that such non-trivial tasks require some kind of super complex tools. Not at all.

Yes, the techniques themselves are not easy — primarily the techniques for requirements elicitation{278}, but even their complexity is mostly in the amount of effort involved.

As for the tools, they are quite simple and one way or another come down to recording in an easy-to-understand form the information gathered. And it is very important to understand that it should not only be convenient for the database designer, but also a representative of the customer, with whom the results of the design will be repeatedly discussed.

In terms of such convenience, the comprehensible and familiar multi-level lists, which can be compiled in any text editor, are unsurpassed. The first level will be entities, the second — their attributes (see the example in figure 4.1.a).

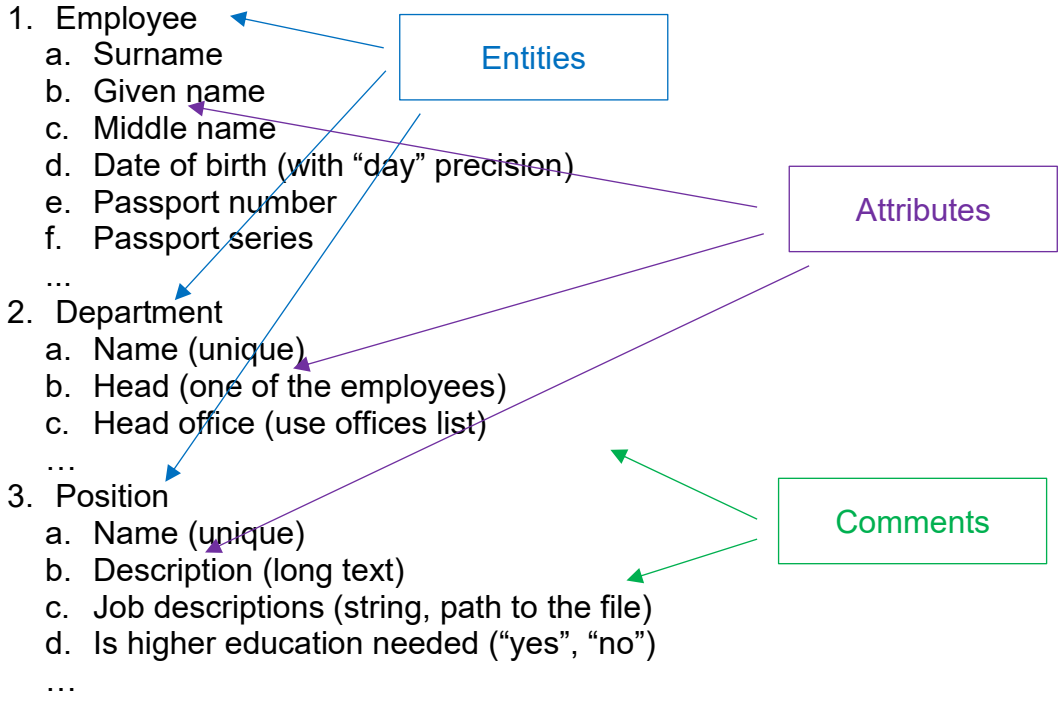


Figure 4.1.a — Example of textual representation of the infological (conceptual) model

The text representation is convenient not only for its familiarity, but also for its practicality:

- everyone has the necessary software on their computer to view and edit text documents;
- creating and editing such documents is very fast;
- notes and comments can be very conveniently placed (and visible immediately without further action);
- the result can be printed out instantly;
- etc.

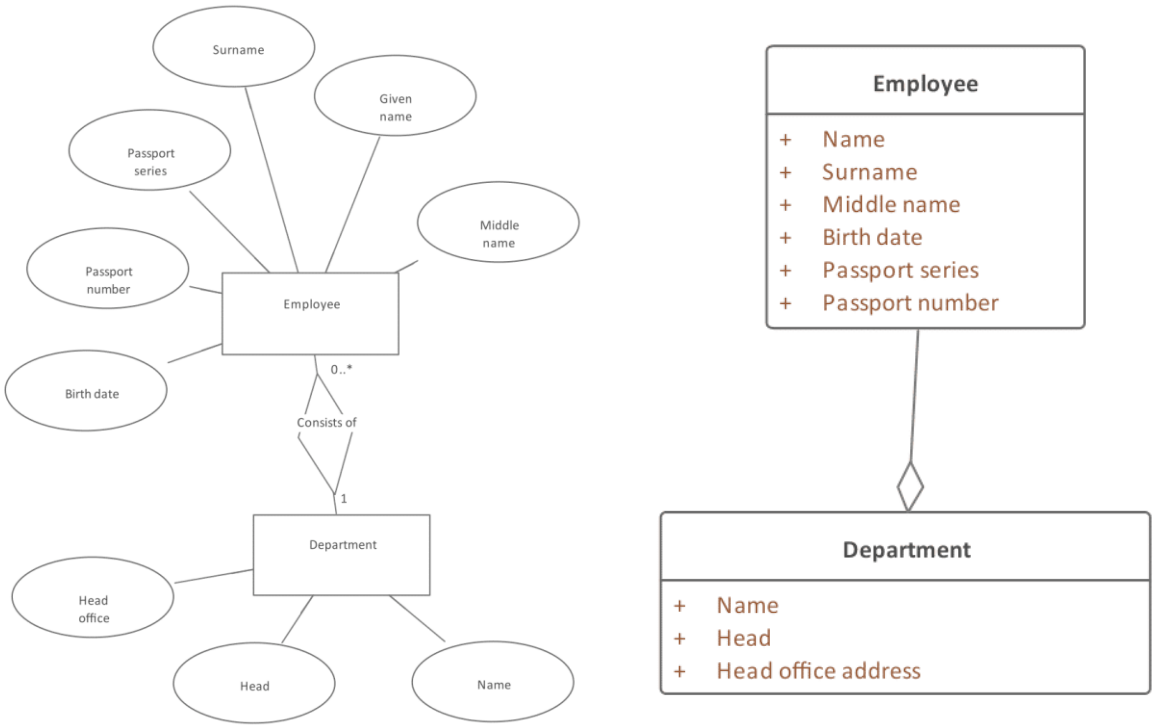
And since everything is so beautiful, it seems that other tools simply do not exist because they are unnecessary. But this is not the case.

The text representation has a number of serious disadvantages:

- it is inconvenient for representing relationships between entities (in fact, it can only be done with comments);
- it is not compact (it can occupy several dozens of pages where other forms of representation would take a couple of screens);
- it allows for discrepancies in the technical aspects of the database implementation (and most often these aspects are omitted at all);
- attempts to eliminate the above disadvantages make textual representation overloaded with information and gradually negate its advantages.

Alternatives to textual representation are graphical forms — in the form of semantic models, graph models, and UML diagrams[192].

Before we consider their examples, let's briefly talk about ER (entity-relation) diagrams mentioned in a great number of sources. Alas, despite their theoretical beauty, they are extremely inconvenient. Compare (see figure 4.1.b) the representation of a small fragment of a database schema in the form of an ER diagram[193] (in Cheng's notation) and in the form of a UML diagram.



ER diagram UML diagram

Figure 4.1.b — Comparison of ER diagram and UML diagram

[192] Yes, there are special languages for describing conceptual models. E.g., CML (conceptual modeling language), but they harmoniously combine the disadvantages of the classical "list" approach and the complexity of graphical models. Such languages have their own fields of application, they have the right to exist, but we cannot call them widespread, so we also do not consider them in this book.

[193] Even such a powerful design tool as Sparx Enterprise Architect is still "unable" to correctly align labels inside elements of ER diagrams, which also indicates the "being in demand" of this form of model representation.

Now back to what really applies in practice.

Semantic and graph models are not as widespread as UML diagrams, but they are very useful in describing complex relationships in subject areas.

It is difficult to use them “directly” to design databases (they do not project directly onto a relational model), but as a constantly available “cheat sheet” they are almost indispensable.

Suppose, for example, we need to represent a complex relationship between employees’ roles in a database (in order to create some control triggers{341}). In order not to keep such a correlation in mind, we can express it with a semantic schema — see figure 4.1.c.

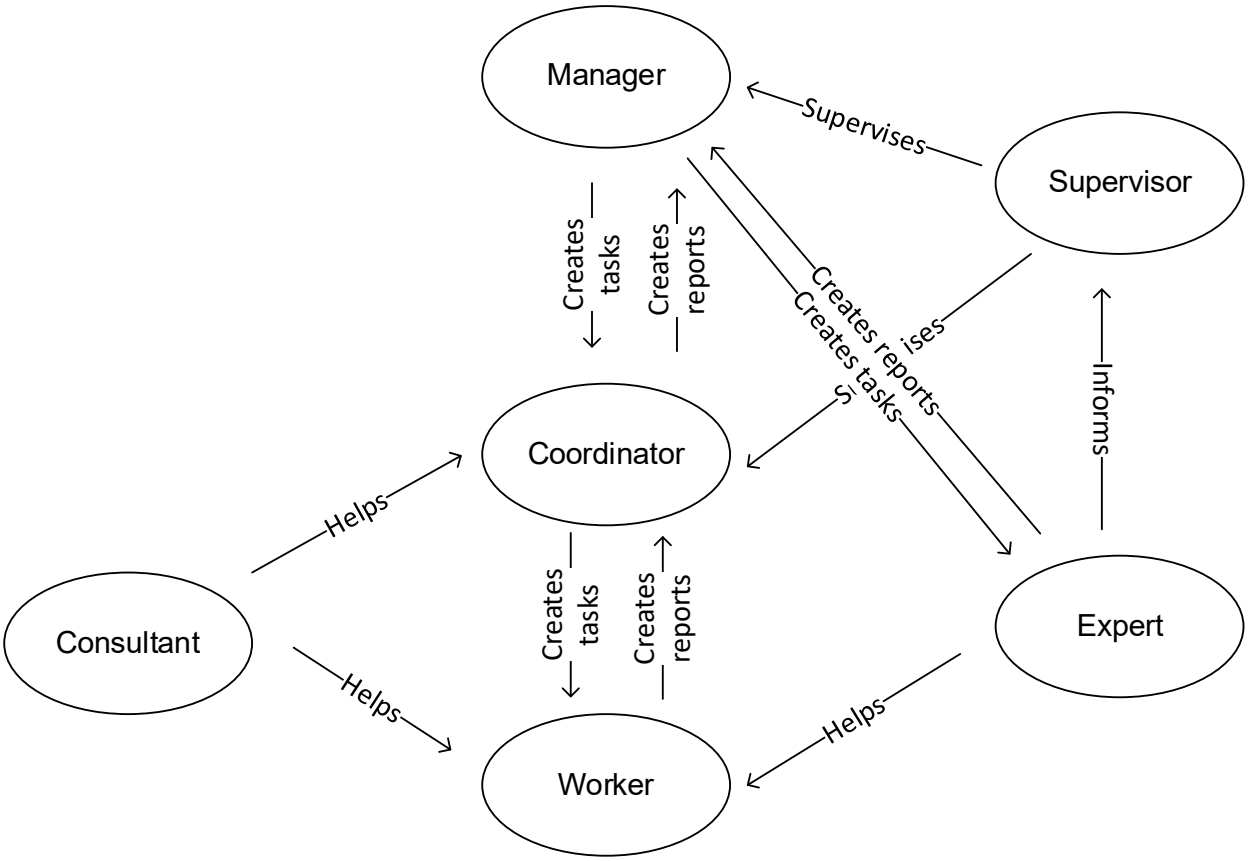


Figure 4.1.c — Semantic schema sample

Graph models are very convenient for describing states and transitions between them. Again, it is much easier to draw a diagram (see figure 4.1.d) than to keep it in mind all the time.

Semantic and graph models can be useful not only to describe complex and unusual subject areas, but also those in which everything seems familiar and obvious, but in this particular project has some atypical features (for example, figure 4.1.c shows that the consultant does not help the manager, although intuitively it seems that he should help; in figure 4.1.d some task cannot go between the states “In progress” and “Declined”, although intuitively there are no objective prohibitions).

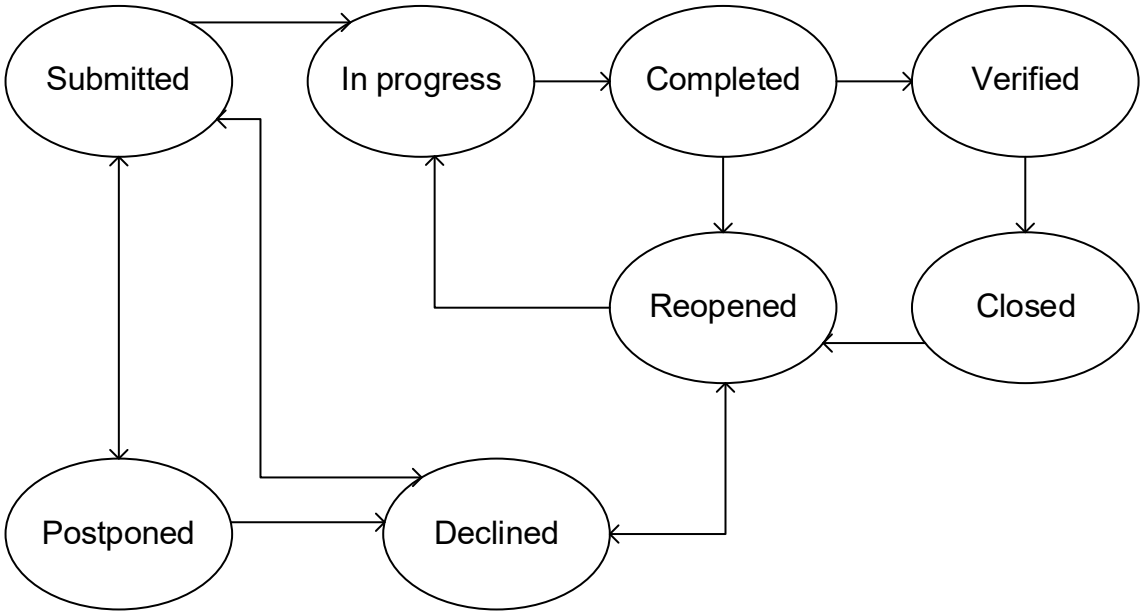


Figure 4.1.d — Graph schema sample

Finally, let's move on to UML and start with a brief reminder of the relationship types[194] (their general list is shown in figure 4.1.e). Yes, the possibilities of UML are much wider, but for database modeling in general it will be enough to remember what a "relationship", "class", "attribute", "method" are.

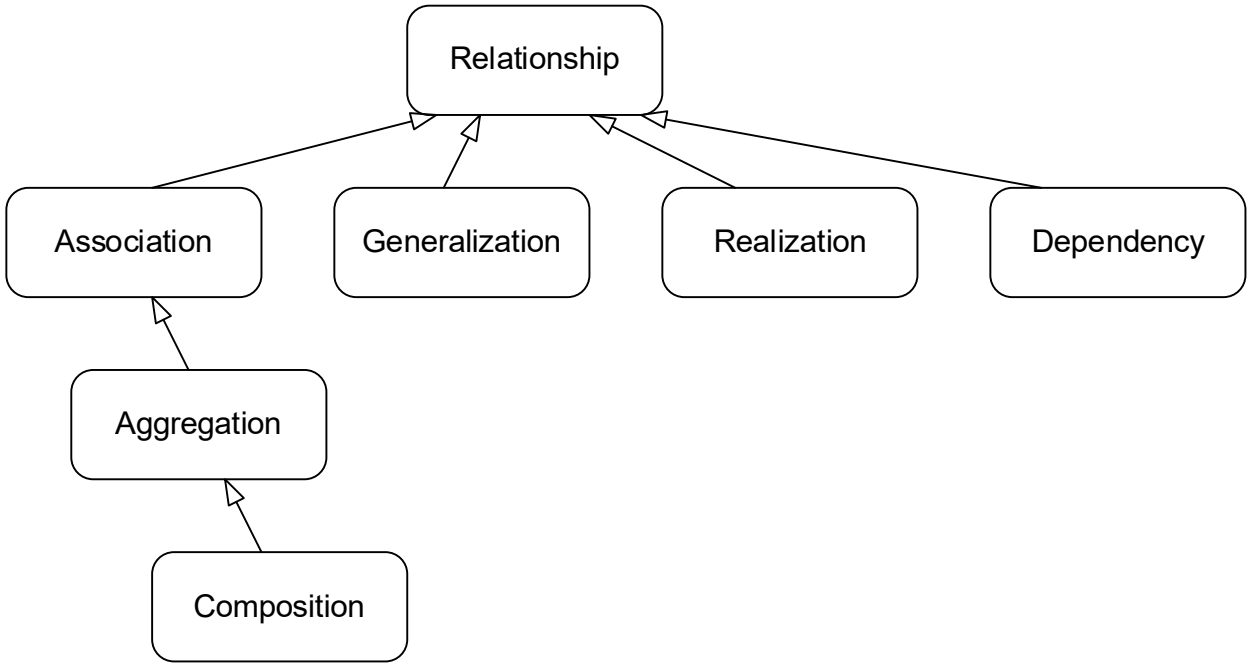


Figure 4.1.e — UML relationships hierarchy

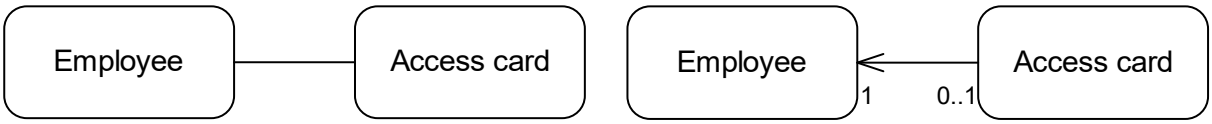
[194] A full discussion of this technology is beyond the scope of this book, but there is a large amount of good documentation on the Internet. You can start from here: https://www.tutorialspoint.com/uml/

Let's take a closer look at all types of UML relationships and their application in database modeling.

Association is the most general, universal and "non-binding" option: it simply reflects the fact that some entities are in a relationship (the type and features of this relationship are not specified, although we can specify some parameters if we wish, e.g., relationship cardinality and its direction).

The example below (see figure 4.1.f) shows that the employee and the access card are linked. The example with refined parameters shows that:

- the access card belongs to the employee (and not vice versa);
- the access card must belong to exactly one employee;
- there may be employees without an access card;
- the employee can have no more than one access card.



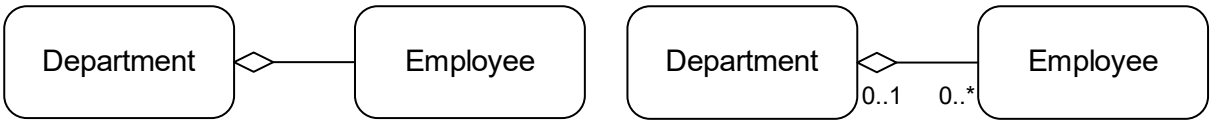
The most general view Clarification of some parameters

Figure 4.1.f — Association sample

Aggregation is a special case of association, showing that the child elements are part of the parent element (but can also exist separately).

The example below (see figure 4.1.g) shows that the department consists of employees ("aggregates" the employees). The example with refined parameters shows that:

- there can be anywhere from zero to infinity employees in a department;
- an employee may belong to no department;
- an employee may belong to no more than one department.



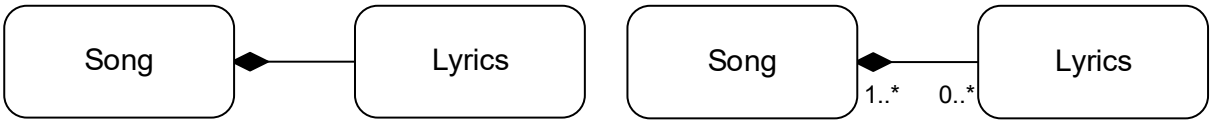
The most general view Clarification of some parameters

Figure 4.1.g — Aggregation sample

Composition is another special case of association, showing that the child elements are part of the parent element, but, unlike aggregation, here the child elements cannot exist by themselves (without the parent element).

The example below (see figure 4.1.h) shows that the lyrics are part of a song (and does not exist by itself, i.e., it is "song lyrics"). The example with refined parameters shows that:

- each lyrics must refer to at least one song;
- a lyrics can refer to more than one song;
- each song may include from zero to an infinity of lyrics.



The most general view Clarification of some parameters

Figure 4.1.h — Composition sample

Generalization is a type of relationship that shows that a child element is a special case of a parent element.

Generalization is very often used in programming (a child class is a special case of a parent class) and in database design is usually immediately expressed in the form of some relationship. But in the initial stages of designing the infological (conceptual) level generalizations have the right to exist (they will be replaced by something a little bit later), because they can reflect the real situation in the subject area.

The example below (see figure 4.1.i) shows that a contractor is a special case of an employee. It is also the generalization used in figure 4.1.e to illustrate the hierarchy of UML relationships. Note that generalization has no cardinality, because expressions like "an employee is N contractors" make no practical sense: here is a very hierarchy, which by definition has no such property as "relationship cardinality".

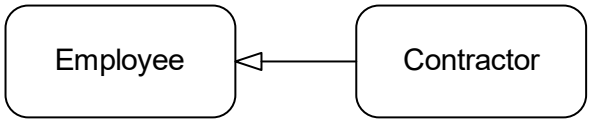


Figure 4.1.i — Generalization sample

Realization is a type of relationship that shows that a child element implements the behavior specified by a parent element.

Like generalization, realization is widely used in programming (classes realize (implement) interfaces), and in database design it is expressed in the form of some relationship, but, again, in the initial stages of infological (conceptual) level design realization can be applied in the UML diagram to reflect the features of the subject area.

The example below (see figure 4.1.j) shows that a technical solution is subordinate to a standard, because it must "behave" as the standard states.

Although many tools allow to expose the "realization relationship cardinality", the classic version of UML diagrams does not represent it.

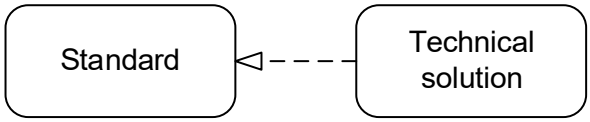
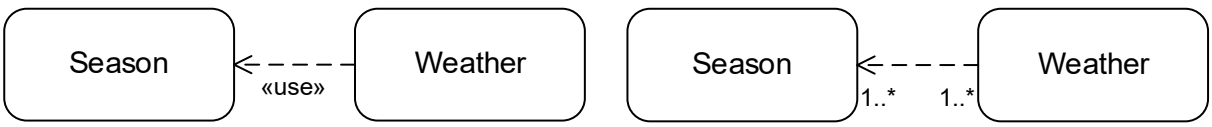


Figure 4.1.j — Realization sample

Dependency is a type of relationship that shows that changes in the parent element necessarily lead to changes in the child element (but not vice versa).

The example below (see figure 4.1.k) shows that a change in season must result in a change in weather (the reverse is not true). The example with refined parameters shows that:

- each season has at least one type of weather (but there can be more, up to infinity);
- each type of weather must refer to at least one season (but can refer to more seasons, up to infinity).



The most general view — Clarification of some parameters

Figure 4.1.k — Dependency sample

It remains to consider how the UML concepts of “class”, “attribute” and “method” are related to the database design.

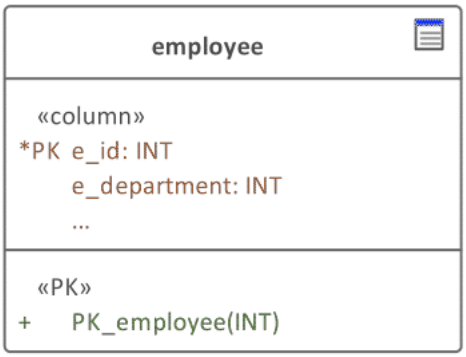


Figure 4.1.l — Connection of UML concepts “class”, “attribute” and “method” to database design

Here one figure 4.1.l will suffice to illustrate all the ideas. The rectangle denotes a class, the upper block shows the class name, the middle block shows class attributes, and the lower block shows class methods.

The easiest thing to deal with is the “class” itself (in database design this would be “relation schema” or just “relation” depending on the context) and the “attribute” (in database design this would be “relation attribute”).

The situation looks a bit more unusual with methods, but it perfectly reflects the DBMS logic: “relation methods” will include primary keys, indexes, and other *operations*.

And this is not a typo — these are exactly operations. From a human point of view, e.g., a primary key is just a field or several fields with certain properties, but from the DBMS point of view it is also the need to *perform* value uniqueness checks. Similarly, an index is the need to *perform* a set of actions for its construction and updating. Therefore, these are exactly the *operations*, and they are reflected in the methods block (i.e., what the relation “can do”).

It is fair to say that this brief UML review will be sufficient to understand all the examples presented in this book.

There are many tools used to design database schemas in UML notation, they are developing intensively, and a description of working with them can almost always be found in the accompanying user manual.

We should only note that one of the most popular is Sparx Enterprise Architect, in which all UML diagrams presented in this book were created.

Let’s move on to look at the big example.



Task 4.1.b: create an appropriate semantic schema based on the “Bank{395}” database infological (conceptual) schema.

4.1.3. Example of Design at the Infological (Conceptual) Level

So far, simple unrelated examples have been used to illustrate certain ideas, most often involving one or two relation schemas.

But from now on, we're going to look at a "long-playing" and very voluminous example: we're going to design a full-fledged database — albeit a training one, but as close to reality as possible.

So, let's imagine that we have to create a file sharing service, i.e., a web application where users can place files for other users to download.

Let's consciously simplify to the limit[195] the list and form of presentation of requirements for the application, obtaining at first approximation this result:

1. The application may contain several pages (the quantity, the hierarchy, and the contents may vary).
2. Application users may create groups and join such groups.
3. Each user may have several roles with a set of permissions for each role.
4. Uses may upload and download files, share files with specific users, groups of users, and the whole world.
5. Users may comment files.
6. Each file has a rating.
7. There may be replies to comments (and other replies) — up to 10 levels of nesting depth.
8. Each file must belong to a category, which determines the set of permissions and limitations.
9. The application shall log all actions of all users.
10. There must be possibility to ban users, groups of users, and non-registered users (by ip address).
11. The application shall display (with minimum time delay) the following statistics: total users, total uploaded files quantity and volume, total downloaded files quantity and volume.

Consider the list of entities, the need for which arises from the presence of such requirements:

#	Requirement text	List of entities
1	The application may contain several pages (the quantity, the hierarchy, and the contents may vary).	Page.
2	Application users may create groups and join such groups.	User, user group.
3	Each user may have several roles with a set of permissions for each role.	Role, permission.
4	Uses may upload and download files, share files with specific users, groups of users, and the whole world.	File.
5	Users may comment files.	Comment.
6	Each file has a rating.	Mark.
7	There may be replies to comments (and other replies) – up to 10 levels of nesting depth.	Comment.
8	Each file must belong to a category, which determine the set of permissions and limitations.	File category.
9	The application shall log all actions of all users.	Log.
10	There must be possibility to ban users, groups of users, and non-registered users (by ip address).	Ban reason.
11	The application shall display (with minimum time delay) the following statistics: total users, total uploaded files quantity and volume, total downloaded files quantity and volume.	Statistics.

[195] See the chapter "2.2. Documentation and requirements testing" in the "Software Testing. Base Course" book (Svyatoslav Kulikov) [https://svyatoslav.biz/software_testing_book/]

So, we have the following list of entities:

1. Page.
2. User.
3. User group.
4. Role.
5. Permission.
6. File.
7. Comment.
8. Mark.
9. File category.
10. Log.
11. Ban reason.
12. Statistics.

This does not yet consider “technical relations” (for “many to many” relationships). Other entities that have not yet been taken into account may also appear here.

Unfortunately, it is impossible to convey the process of communication with the customer in a book format (in fact, most often it is a regular conversation in a question-answer format), but let’s assume that after communication and clarification of details we got the following picture (attributes of entities were added, there are more entities themselves):

1. Page:
    a. Parent page.
    b. Name (to display on the page itself).
    c. Name in menu.
    d. Title (TITLE HTML-tag).
    e. Keywords (meta … keywords HTML-tag).
    f. Description (meta … description HTML-tag).
    g. Text.
2. User:
    a. Login.
    b. Password.
    c. E-mail.
    d. Registration datetime.
    e. Birth date.
    f. Bonus for uploads.
3. User group:
    a. Group owner (creator).
    b. Name.
    c. Description.
4. Role:
    a. Name.
    b. Upload volume limit.
    c. Download volume limit.
    d. Upload quantity limit.
    e. Download quantity limit.
    f. Download speed limit.
5. Permission:
    a. Name.
    b. Description.

6. File:
    a. Size (bytes).
    b. Creation datetime.
    c. Expiration datetime.
    d. Original name (without extension).
    e. Original extension.
    f. Name on server.
    g. Checksum.
    h. Deletion link (for unregistered users).
7. Public access link:
    a. File.
    b. Hash.
    c. Expiration datetime.
    d. Access password (if any).
8. File access permission:
    a. File.
    b. Action.
    c. Who can perform this action.
9. Mark:
    a. File.
    b. Who gave this mark.
    c. Mark value.
10. Comment:
    a. To which file.
    b. To which comment.
    c. Text.
    d. Who made this comment.
    e. Creation datetime.
11. File category:
    a. Name.
    b. Age restriction (if any).
12. Age restriction:
    a. Minimal age (years).
    b. Name.
    c. Description.
13. Log:
    a. User.
    b. Ip-address.
    c. Operation.
    d. File (if applicable).
    e. Datetime.
    f. Parameters.
14. Log archive (for records older than one month):
    a. User.
    b. Ip-address.
    c. Operation.
    d. File (if applicable).
    e. Datetime.
    f. Parameters.

 © EPAM Systems, 2020–2025

15. Ban reason:
    a. Name.
    b. Description.
16. Ip blacklist:
    a. IPv4-address.
    b. IPv6-address.
    c. Expiration datetime.
    d. Ban reason.
17. Statistics:
    a. Registered users.
    b. Registered users today.
    c. Files uploaded.
    d. Files uploaded today.
    e. Uploaded volume.
    f. Uploaded volume today.
    g. Downloaded files.
    h. Downloaded files today.
    i. Downloaded volume.
    j. Downloaded volume today.

Once again, let's assume that after thinking about and discussing the result with our colleagues, we returned after a couple of days to discuss it with the customer, in the process of which we gradually began to write out the technical details.

We got the following:

1. Page:
    a. Id.
    b. Parent page (rFK).
    c. Name (to display on the page itself).
    d. Name in menu (unique within one menu level).
    e. Title (TITLE HTML-tag).
    f. Keywords (meta … keywords HTML-tag).
    g. Description (meta … description HTML-tag).
    h. Text.
2. User:
    a. Id.
    b. Login (unique).
    c. Password (sha256-hash).
    d. E-mail (unique).
    e. Registration datetime (up to seconds).
    f. Birth date (up to days).
    g. Speed bonus for uploads (FK).
    h. Speed bonus for uploads expiration datetime (up to seconds).
    i. Volume bonus for uploads (FK).
    j. Volume bonus for uploads expiration datetime (up to seconds).
3. Bonus:
    a. Id.
    b. Quantity of uploaded files needed to achieve the bonus.
    c. Volume of uploaded files needed to achieve the bonus.
    d. Speed addition.
    e. Volume addition.

4. User group:
    a. Id.
    b. Owner (FK).
    c. Name (unique).
    d. Description.
5. Role:
    a. Id.
    b. Name (unique).
    c. Upload volume limit (bytes).
    d. Download volume limit (bytes).
    e. Upload quantity limit.
    f. Download quantity limit.
    g. Download speed limit (bytes per second).
6. Permission:
    a. Id.
    b. Name (unique).
    c. Description.
7. File:
    a. Id.
    b. Size (bytes).
    c. Creation datetime (up to seconds).
    d. Expiration datetime (up to seconds).
    e. Original name (without extension).
    f. Original extension.
    g. Name on server (sha256-hash).
    h. Checksum (sha256-hash).
    i. Deletion link (for unregistered users, sha256-hash).
8. Public access link:
    a. Id.
    b. File (FK).
    c. Hash (sha256-hash).
    d. Expiration datetime (up to seconds).
    e. Access password (may be NULL, sha256-hash).
9. File access permission:
    a. Id.
    b. File (FK).
    c. Action (FK).
    d. User (FK).
    e. Group (FK).
    f. “Allowed for any registered user” flag.
    g. “Allowed for anybody” flag.
10. Mark:
    a. Id.
    b. File (FK).
    c. User (FK).
    d. Value (1 to 10).

 © EPAM Systems, 2020–2025

11. Comment:
    a. Id.
    b. File (FK).
    c. Comment (FK).
    d. Text.
    e. User (FK).
    f. Creation datetime (up to seconds).
12. File category:
    a. Id.
    b. Name (unique).
    c. Age restriction (may be NULL, FK).
13. Age restriction:
    a. Id.
    b. Minimal age (years).
    c. Name (unique).
    d. Description.
14. Log:
    a. User (FK, NULL for non-registered users).
    b. Ip-address.
    c. Operation (FK).
    d. File (if applicable, FK).
    e. Datetime (up to seconds).
    f. Parameters (if applicable, text).
15. Operation:
    a. Id.
    b. Name (unique).
16. Log archive (for records older than one month):
    a. User (FK, NULL for non-registered users).
    b. Ip-address.
    c. Operation (FK).
    d. File (if applicable, FK).
    e. Datetime (up to seconds).
    f. Parameters (if applicable, text).
17. Ban reason:
    a. Id.
    b. Name (unique).
    c. Description.
18. Ip blacklist:
    a. IPv4-address (unique).
    b. IPv6-address (unique).
    c. Expiration datetime (up to seconds).
    d. Ban reason (FK).
19. Statistics:
    a. Registered users.
    b. Registered users today.
    c. Files uploaded.
    d. Files uploaded today.
    e. Uploaded volume (bytes).
    f. Uploaded volume today (bytes).
    g. Downloaded files.
    h. Downloaded files today.
    i. Downloaded volume (bytes).
    j. Downloaded volume today (bytes).

 © EPAM Systems, 2020–2025

As you can easily see, the textual description already takes almost three pages, and it does not yet reflect the relationships and intermediate relations for “many to many” relationships. We are not going to abandon the textual representation yet, but we will supplement it with a graphical one to demonstrate how much more compact and clearer it is (see figure 4.1.m).

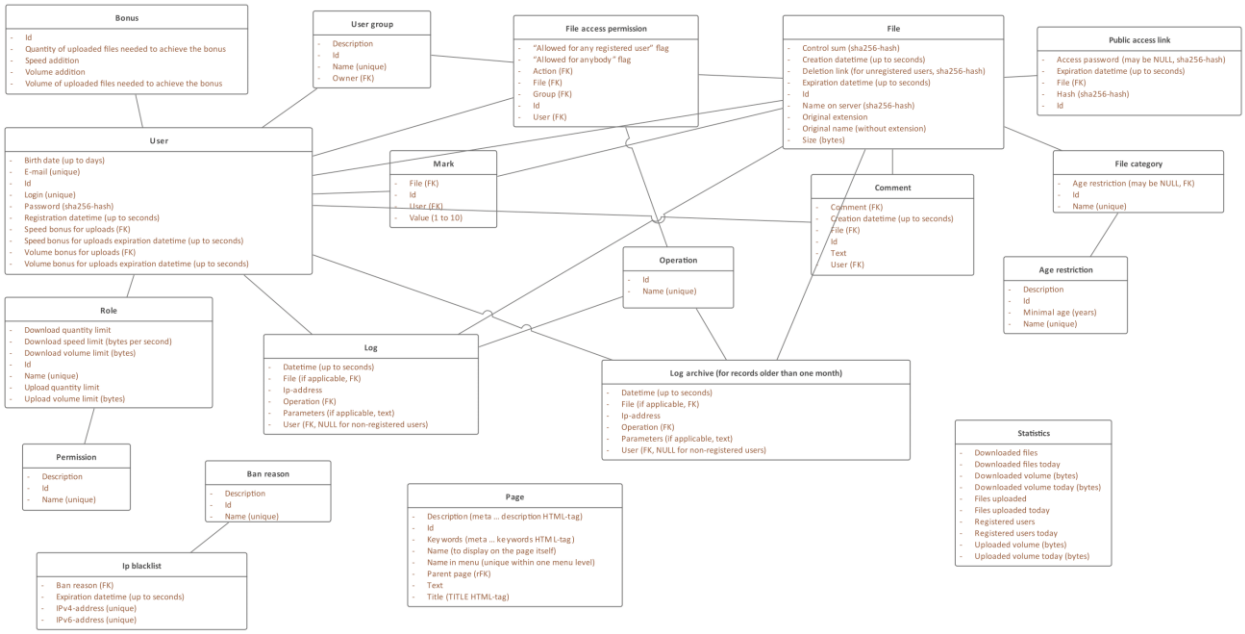


Figure 4.1.m — Graphical representation of the results of the design at the infological (conceptual) level[196]

Instead of three pages of text, we got one picture (which fits on one screen) which not only shows all the same information, but also adds relationships between entities, which gives us additional information and increases clarity.

In this particular example, a small “trick” has been applied: even though we are now on the infological (conceptual) level, we have already added surrogate primary keys{38} (the “Id” field) and foreign keys{43} to both the text description and the graphical representation — which, strictly speaking, refers to the datalogical modelling level, but is already very helpful from a technical point of view.

This “trick” clearly illustrates that the division into levels of modeling is largely conventional in nature, and the boundaries between levels are very blurred{8}.

Also, this “trick” gives us another advantage: if you carefully read the wording of entity attribute descriptions, you will notice that they are almost ready comments to the table fields that we will design very soon.

The more complex the project is, the more complex its database is, and the more complex the subject area is, the more time should be spent in communication with the customer in order to fully identify all the entities, attributes, relationships and features of business processes that will affect the database schema (if this is not done, it is very likely to violate such a key feature of the database as the adequacy to the subject area{11}).

However, it is still worth remembering that it is not possible to take into account all the nuances at the infological (conceptual) modelling level (and that is why we have to go through the whole top-down {10} and bottom-up{10} design process several times).

[196] You can find a full-size picture in the handouts{4} or at this link: https://svyatoslav.biz/relational_databases_book_download/pics/conceptual_level_graphical_representation_en.png.

So, now we can safely move on to the next stage — design at the datalogical level.



Task 4.1.c: refine the infological (conceptual) schema of the “Bank{395}” database so as to eliminate all the deficiencies found in it (as far as possible without involving a “hypothetical customer” to get answers to emerging issues).

4.2. Design at the Datalogical (Logical) Level

4.2.1. Design Goals and Objectives at the Datalogical (Logical) Level

Before reading the material in this chapter, it is worth recalling the basics of database modeling{7}.

As mentioned in the definition of the datalogical{9} modeling level, its main purpose is to detail the conceptual model and turn it into a schema in which the previously identified entities, attributes, and relationships are designed according to the rules of modeling for the chosen type of database (often even taking into account a particular DBMS).

Here we do not stop analyzing the subject area (most likely there will be new questions, the answers to which will force us to modify the infological (conceptual) model), but the special attention at this level should be paid to the previously discussed requirements for any database{10}.

Since the future structure of the database is formed exactly at the datalogical level, all decisions made here (and, alas, committed errors) will have a very strong impact on the degree of database adequacy to the subject area, on the database usability, on its performance, and data safety.

In fact, all of the information presented earlier in Chapters 2{19} and 3{157} of this book applies in full to the specifics of design at the datalogical modelling level.

What tables{19} should we create? Why exactly these kinds of tables? Maybe there is a better option? What fields{19} will these tables have (and what data types and other properties will these fields have)? What keys{32} should we use? What relationships{53} to make and how exactly to make them? Maybe some of the indexes{103} and views{331} are already “visible”? Is our database schema sufficiently normalized{207}? Is it not subject to some anomalies{157}? All these questions and many others will have to be answered in the process of database design at the datalogical modelling level.

If we reduce the above to literally two phrases, we get:

- the purpose of design at the datalogical modelling level is to create the structure of the future database;
- the task of design at the datalogical modelling level is to make the created structure of the highest quality, free of issues detected at this stage.

It is impossible to foresee and eliminate all issues in advance, and that is perfectly normal. But we can at least get rid of the most obvious, typical mistakes.

The so-called bottom-up design{10} helps a lot to achieve this: we already see the database structure and can already analyze what results the execution of certain queries on such a structure will lead to. And before “things get too far”, we can easily change the database structure if we see that it has this or that disadvantage.

And in addition to the general information presented in Chapters 2{19} and 3{157} of this book, we can apply various specific techniques and tools that can greatly simplify our work.

Let’s take a look at them.



Task 4.2.a: make a list of questions, the answers to which would help you improve the “Bank{395}” database schema.

4.2.2. Design Tools and Techniques at the Datalogical (Logical) Level

Before we proceed directly to the tools and techniques, let's take a step back and emphasize what you need to know and be able to do to successfully design databases at the datalogical modelling level.

Ideally	At least
Have a deep understanding of relational theory.	Understand relational theory.
Know "on instinct" the normalization theory.	Understand at least 1-3 normal forms.
Have a deep knowledge of SQL.	Know the basics of SQL.
Have a good knowledge of the target DBMS at a level that allows for fine-grained administration.	Be able to install and configure the target DBMS.
Be able to use design tools.	Have a lot of patience to learn design tools.

The relational and normalization theory are sufficiently introduced in the previous sections of this book, for a deeper dive into SQL you can refer to the "Using MySQL, MS SQL Server and Oracle by Examples[197]" book.

The hardest part will be the last two points — both DBMS and design tools are developing too rapidly for there to be some "fundamental" books on them (they will be obsolete by the time of publishing), so the most reliable solution is to read the official documentation for a particular version of a particular DBMS and a particular design tool that you use. The official documentation may not be written in an extremely simplified language, but it will at least be up to date — and this advantage in this case outweighs all the disadvantages.

Back to the subject of this chapter.

Design at the infological (conceptual) level implied an intensive discussion with the customer on the schema being formed, and therefore we tried to use tools that would be accessible and understandable to a "non-technical" person (who, e.g., may be the deepest expert in international logistics or another subject area, but does not have to understand the technical details of databases).

Starting from the datalogical level, we are already much more focused on technical specialists, so the solutions used here may not be clear to the customer — and this is normal, because we will make the necessary changes to the infological model and continue to speak to the customer "in his language", not overloading them with technical details.

And the first thing technicians should do is arrive at a few key agreements (conventions), which can generally be divided into two groups:

- DBMS conventions;
- database conventions.

The absence or insufficient elaboration of such conventions greatly reduces such a property of the database as usability {12}.

[197] "Using MySQL, MS SQL Server and Oracle by Examples" (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

DBMS conventions may include, for example, the following items (specific examples will be in the next section{299}):

- DBMS type;
- particular DBMS;
- minimum supported version of the selected specific DBMS;
- infrastructural features of the selected specific DBMS.

In this list, the items are listed in descending order of their "impact" (so, for example, the transition in the middle of the project from a relational database to a hierarchical one will actually mean starting the project from scratch; but the transfer of several servers into the cloud can go almost painlessly).

Since the choice of a particular DBMS and its version greatly affects further technical decisions, it is also worth dealing with these issues before much work is done at the datalogical level. If the schema has to be redesigned for another version of the selected DBMS or even for another DBMS, it is guaranteed to require a lot of routine work, and possibly even intellectual work, if new technological solutions have to be sought to replace those already created.

Database conventions are easier to make, because they are barely dependent on external factors, and therefore, as a rule, almost do not change. What, however, does not reduce their importance. Such decisions include (specific examples will be in the next section{299}):

- structures naming conventions;
- SQL code conventions;
- comments conventions;
- other project-specific conventions (e.g., API[198] conventions in the form of views{331} and stored routines{353}).

Of course, no one is immune to changes in the project situation that will force us to revise any of the conventions just outlined, but we can think through these conventions so well that in the future they will not have to be changed on our own initiative.

Things are ambiguous with tools at the datalogical level.

For the infological (conceptual) level, in general, it would be enough for us to have any text editor (or any convenient for us and the customer's representatives specialized UML (or other graphic language) editor — there are a lot of such tools, and most specialists know how to use them well).

For the datalogical level, we need specialized tools that allow us to optimally form the database structure, to revise and analyze it repeatedly, and eventually export it to a full-fledged SQL code for import into the DBMS.

Each DBMS developer traditionally provides its own tools, e.g.:

- MySQL Workbench[199] for MySQL;
- SQL Server Management Studio[200] for MS SQL Server;
- SQL Developer Data Modeler[201] for Oracle;
- and so on, there are a lot of these tools.

These are certainly very powerful tools, designed with all the features of the target DBMS in mind, and they are definitely worth considering. Except for one "but": they are more useful for the next, physical {305} design level.

---

[198] **API** (application programming interface) — a computing interface which defines interactions between multiple software intermediaries. ("Wikipedia") [https://en.wikipedia.org/wiki/Application_programming_interface]

[199] "MySQL Workbench" [https://www.mysql.com/products/workbench/]

[200] "SQL Server Management Studio" [https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms]

[201] "SQL Developer Data Modeler" [https://www.oracle.com/database/technologies/appdev/datamodeler.html]

Each of these tools has a long history of development, their creators set different goals in the first place, and as a result, each such tool is good in some way, but can hardly claim to be a “reference solution”.

Tools originally created to design “databases in principle” might be eligible for this title, e.g.:

- Sparx Systems Enterprise Architect[202] (most of the illustrations with database schemas in this book are made using this tool);
- DbSchema[203];
- DbDiagram.io[204];
- and so on, there are a lot of these tools.

Why do these “universal” tools win? Here are just a few of the most obvious advantages:

- None of them work directly with the physical database (which reduces to zero your chances of one day destroying an already working customer database).
- These tools support the syntax and features of many DBMSes, which allows you to compare solutions for different DBMSes, to switch (convert schemas) from one DBMS to another, or even to work in such unusual situations where part of the database runs on one DBMS and part runs on another DBMS.
- The creators of these tools considered many shortcomings of “highly specialized” analogues and tried to correct them.
- Such tools tend to have a much more user-friendly interface, more advanced collaboration (teamwork) tools, and many other competitive advantages.

Naturally, you have a full right to choose both between these groups of tools and between some specific products. And at the training stage it is recommended to try to work with at least 3–5 fundamentally different tools in order to understand their differences from personal experience and form your own preferences.



Task 4.2.b: create a datalogical schema of the “Bank{395}” using MySQL Workbench.

[202] “Sparx Systems Enterprise Architect” [https://sparxsystems.com/products/ea/index.html]
[203] “DbSchema” [https://dbschema.com]
[204] “DbDiagram.io” [https://dbdiagram.io]

4.2.3. Example of Design at the Datalogical (Logical) Level

As noted earlier, when starting to design the database at the datalogical level, it is worth making a number of agreements (conventions){296}. Let's do this.

DBMS conventions:

- DBMS type: relational.
- particular DBMS: MySQL (community edition);
- minimal DBMS version: 8.0 and newer (as earlier versions have some technical limitations);
- DBMS infrastructure details: standalone server[205].

Database conventions:

- Structures naming:
  - all table and field names must be lowercase only;
  - word separator in tables' and fields' names must be "_" only;
  - nouns in tables' names must be in singular form only (e.g., "file", NOT "files"), nouns in fields' names may be in plural form (still it's not recommended);
  - fields' names must have prefixes composed with beginning tables' name letters;
  - all unique constraints' names must have "UNQ_" prefix and must contain all corresponding fields' names;
  - all triggers' names must have "TRG_" prefix and must contain the tables' names and triggering events' names.
- SQL code formatting:
  - all SQL keywords must be in uppercase;
  - all structures' names must be enclosed in "`" symbols.
- Comments principles:
  - all database structures must have a comment.
- Other specifics (like API, and so on):
  - all datetime fields must be of `INTEGER` type and store UNIXTIME-values;
  - all date fields must be of `DATE` type;
  - all primary keys must be surrogate, auto-increment, unsigned.

As the main design tool we will use Sparx Enterprise Architect{298}, but we emphasize that in specially complex cases and/or when designing a large database, there may be an intermediate stage, in which we can still rely on the text description of the schema.

In the case of our particular educational project, there is no obvious need for such an intermediate step, but to complete the picture, let's give an example. As a rule, tabular representation will be used here, because it is more convenient for perception of structured information than lists.

During the formation of this example, we will also make some edits to the schema, clarifying what we "forgot" at this level (see task 4.2.d{304}).

[205] In reality, most likely, we would immediately focus on a cluster of servers, because such a project involves a fairly high load, with which a standalone server is unlikely to cope. But for the sake of simplifying the learning material we will stop at a more trivial solution.

Database table	Database table field	Data type	Comments
Page	Id	SMALLINT	Primary key
	Parent page (rFK)	SMALLINT	
	Name (to display on the page itself)	VARCHAR(500)	
	Name in menu (unique within one menu level)	VARCHAR(100)	Make a trigger for this!
	Title (TITLE HTML-tag)	VARCHAR(500)	
	Keywords (meta … keywords HTML-tag)	VARCHAR(500)	
	Description (meta … description HTML-tag)	VARCHAR(500)	
	Text	TEXT	
User	Id	BIGINT	Primary key
	Login (unique)	VARCHAR(100)	Unique value
	Password (sha256-hash)	CHAR(64)	
	E-mail (unique)	VARCHAR(150)	Unique value
	Registration datetime (up to seconds)	INT	
	Birth date (up to days)	DATE	
	Speed bonus for uploads (FK)	SMALLINT	
	Speed bonus for uploads expiration datetime (up to seconds)	INT	
	Volume bonus for uploads (FK)	SMALLINT	
	Volume bonus for uploads expiration datetime (up to seconds)	INT	
	Uploaded files count	BIGINT	Aggregation field, updates by trigger
	Downloaded files count	BIGINT	Aggregation field, updates by trigger
	Comments count	BIGINT	Aggregation field, updates by trigger
	Marks count	BIGINT	Aggregation field, updates by trigger
	Ban reason (FK)	SMALLINT	
	Ban expiration datetime (up to seconds)	INT	
Bonus	Id	SMALLINT	Primary key
	Quantity of uploaded files needed to achieve the bonus	BIGINT	
	Volume of uploaded files needed to achieve the bonus	BIGINT	
	Speed addition (bytes per second)	BIGINT	
	Volume addition (bytes)	BIGINT	
User group	Id	SMALLINT	Primary key
	Owner (FK)	BIGINT	
	Name (unique)	VARCHAR(100)	Unique value
	Description	TEXT	

Database table	Database table field	Data type	Comments
Role	Id	SMALLINT	Primary key
	Name (unique)	VARCHAR(100)	Unique value
	Upload volume limit (bytes)	BIGINT	NULL, if no limit
	Download volume limit (bytes)	BIGINT	NULL, if no limit
	Upload quantity limit	BIGINT	NULL, if no limit
	Download quantity limit	BIGINT	NULL, if no limit
	Download speed limit (bytes per second)	BIGINT	NULL, if no limit
	Download speed limit (bytes per second)	BIGINT	NULL, if no limit
Permission	Id	SMALLINT	Primary key
	Name (unique)	VARCHAR(100)	Unique value
	Description	TEXT	
File	Id	BIGINT	Primary key
	Owner	BIGINT	
	Size (bytes)	BIGINT	
	Creation datetime (up to seconds)	INT	
	Expiration datetime (up to seconds)	INT	NULL, if no expiration datetime
	Original name (without extension)	VARCHAR(1000)	
	Original extension	VARCHAR(1000)	
	Name on server (sha256-hash)	CHAR(64)	Unique value
	Checksum (sha256-hash)	CHAR(64)	
	Deletion link (for unregistered users, sha256-hash)	CHAR(64)	Unique value
Public access link	Id	BIGINT	Primary key
	File (FK)	BIGINT	
	Hash (sha256-hash)	CHAR(64)	Unique value
	Expiration datetime (up to seconds)	INT	NULL, if no expiration datetime
	Access password (may be NULL, sha256-hash)	CHAR(64)	NULL, if not set
File access permission	Id	BIGINT	Primary key
	File (FK)	BIGINT	
	Action (FK)	SMALLINT	
	User (FK)	BIGINT	For these two fields one must be NULL, the other must be not NULL
	Group (FK)	SMALLINT	
	“Allowed for any registered user” flag	BIT(1)	
	“Allowed for anybody” flag	BIT(1)	
Mark	Id	BIGINT	Primary key
	File (FK)	BIGINT	
	User (FK)	BIGINT	
	Value (1 to 10)	TINYINT	

 © EPAM Systems, 2020–2025

Database table	Database table field	Data type	Comments
Comment	Id	BIGINT	Primary key
	File (FK)	BIGINT	For these two fields one must be NULL, the other must be not NULL
	Comment (FK)	BIGINT	
	Text	TEXT	
	User (FK)	BIGINT	
	Creation datetime (up to seconds)	INT	
	Mark (FK)	BIGINT	When commenting file, it is possible to add both comment text and also rate the file, rates are possible in top-level comments only
File category	Id	SMALLINT	Primary key
	Name (unique)	VARCHAR(100)	Unique value
	Age restriction (may be NULL, FK)	SMALLINT	
Age restriction	Id	SMALLINT	Primary key
	Minimal age (years)	TINYINT	
	Name (unique)	VARCHAR(100)	Unique value
	Description	TEXT	
Log	User (FK, NULL for non-registered users)	BIGINT	
	Ip-address	VARCHAR(45)	
	Operation (FK)	SMALLINT	
	File (if applicable, FK)	BIGINT	NULL, if no files involved
	Datetime (up to seconds)	INT	
	Parameters (if applicable)	TEXT	Serialized array
Operation	Id	SMALLINT	Primary key
	Name (unique)	VARCHAR(100)	Unique value
Log archive (for records older than one month)	User (FK, NULL for non-registered users)	BIGINT	
	Ip-address	VARCHAR(45)	
	Operation (FK)	SMALLINT	
	File (if applicable, FK)	BIGINT	NULL, if no files involved
	Datetime (up to seconds)	INT	
	Parameters (if applicable)	TEXT	Serialized array
Ban reason	Id	SMALLINT	Primary key
	Name (unique)	VARCHAR(100)	Unique value
	Description	TEXT	
Ip blacklist	IPv4-address (unique)	CHAR(15)	For these two fields one must be NULL, the other must be not NULL
	IPv6-address (unique)	CHAR(45)	
	Expiration datetime (up to seconds)	INT	
	Ban reason (FK)	SMALLINT	
Statistics	Registered users	BIGINT	All fields of this table are aggregating and are updated by triggers on the corresponding tables
	Registered users today	BIGINT	
	Files uploaded	BIGINT	
	Files uploaded today	BIGINT	
	Uploaded volume (bytes)	BIGINT	
	Uploaded volume today (bytes)	BIGINT	
	Downloaded files	BIGINT	
	Downloaded files today	BIGINT	
	Downloaded volume (bytes)	BIGINT	
	Downloaded volume today (bytes)	BIGINT	

Although the above table does not differ much from the list of entities and attributes{290} discussed at the infological (conceptual) level, it still contains a number of important additional details and can be a good addition to the project documentation.

Now let's present the resulting schema graphically (see figure 4.2.a).

There are already more than two dozen tables and a large number of relationships between them, yes — the schema is quite voluminous, and therefore in "one picture" it is difficult to read, but it still looks much more compact in comparison with four pages of text descriptions. Especially if you open the source file (see the book handouts{4}) in Sparx Enterprise Architect and select the scale of display convenient for you personally.

You may notice that there are more tables in figure 4.2.a than were presented earlier in the text description{300}. This is due to the fact that during the formation of the schema the tables for "many to many{56}" relationships were added, as well as other small edits (including adaptation to the requirement of database usability{12} were made).

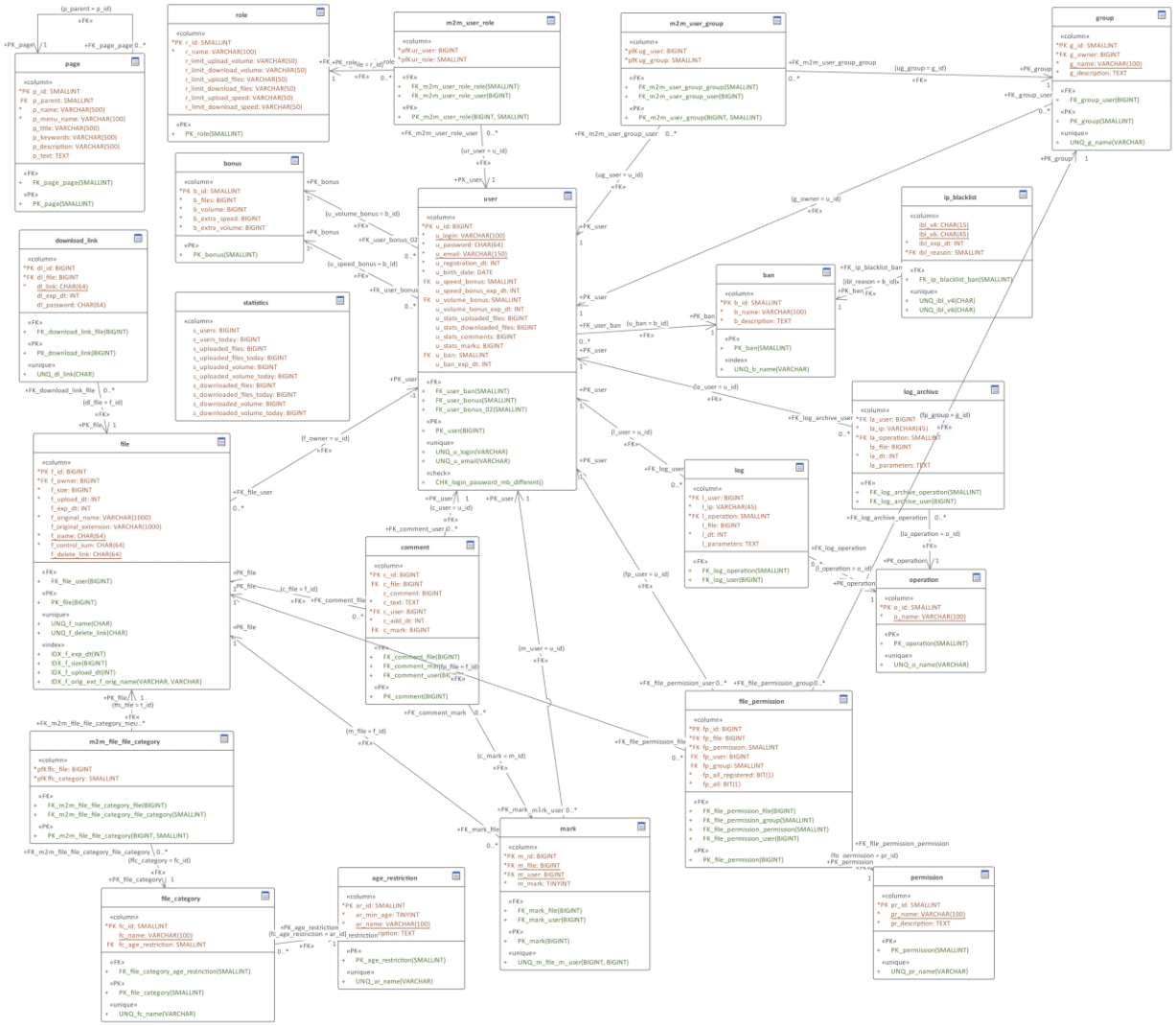


Figure 4.2.a — Graphical representation of the results of the design at the datalogical (logical) level[206]

[206] You can find a full-size picture in the handouts{4} or at this link: https://svyatoslav.biz/relational_databases_book_download/pics/logical_level_graphical_representation_en.png.

Also, in the process of creating this schema, almost all necessary uniqueness constraints{106} were added, relationships between tables were made, and all comments were "written" into the schema (so that in the future, when generating SQL code, they will be transferred to the real database).

Already from this point on, we can (and even should) periodically generate SQL code and import the result into the real database to avoid various annoying errors that everyone is prone to make (if something is "wrong" with the schema, the import fails, and the DBMS reports the cause in error messages in detail). It is much easier to fix such defects now, while the schema is still quite "raw", and its revision will not require much effort.

For example, when preparing material for this book, the first attempt to transfer the schema to the DBMS resulted in the following error:

MySQL — DBMS SQL script execution error message

```
1   ALTER TABLE `page` ADD CONSTRAINT `fk_page_page` FOREIGN KEY (`p_parent`)
2   REFERENCES `page` (`p_id`)
3   ON DELETE RESTRICT
4   ON UPDATE RESTRICT
5   Error Code: 1825. Failed to add the foreign key constraint on table 'page'.
6   Incorrect options in FOREIGN KEY constraint 'FK_page_page'.
```

Indeed, the primary key of the `page` table was an unsigned integer, while the recursive foreign key of the same table was a signed integer (capable of taking negative values).

After fixing this (and a couple of similar) errors, the import to the DBMS was successful.

And yet, there is one more very intense stage left before the design is completed — the creation of the database schema at the physical modelling level.



Task 4.2.c: refine the datalogical schema of the "Bank{395}" database so as to eliminate all of the deficiencies found in it (as far as possible without involving "hypothetical customer" to get answers to emerging issues).



Task 4.2.d: earlier{299} in this chapter it was noted that during the intermediate stage of design (in which we still relied on the textual description of the schema), we made some edits that we "forgot" at the infological (conceptual) level. What edits did we make? Make a complete list.



Task 4.2.e: what other data would you add to the summary table{300} reflecting the result of the datalogical design of this filesharing service database? Make appropriate changes.

4.3. Design at the Physical Level

4.3.1. Design Goals and Objectives at the Physical Level

Before reading the material in this chapter, it is worth recalling the basics of database modeling[{7}].

Starting from this point, we enter an area in which there are very few general solutions, because, as follows from the definition of the physical level of modeling[{10}], we will be forced to maximize respect for the technical features and capabilities of a particular DBMS.

And yet some common goals and objectives can be highlighted even here, because of particular interest to us are:

- access permissions;
- encodings;
- storage engines;
- indexes;
- DBMS settings.

Access permissions

In relatively small and simple projects, it is assumed that interaction with the DBMS is carried out on by a single user — the application working with the DBMS is always authorized using the same “login-password” pair and controls the permissions of its users itself (see figure 4.3.a).

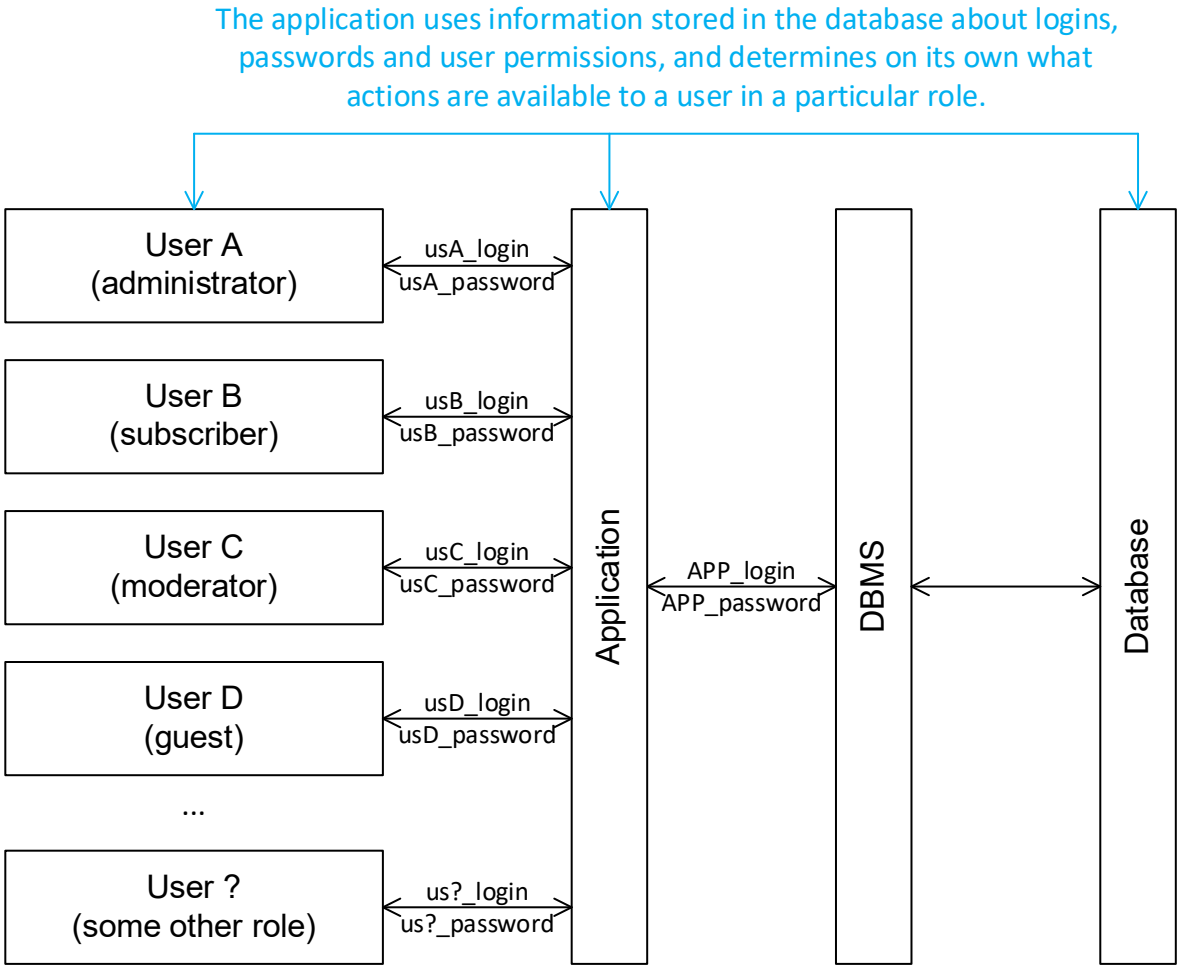


Figure 4.3.a — Working with the DBMS via single credentials

This option is widespread because of its simplicity and speed of implementation. But if we consider a situation where many different applications can work with the same database, and there are higher security requirements, it makes sense to shift the authentication and permission control to the DBMS (see figure 4.3.b).

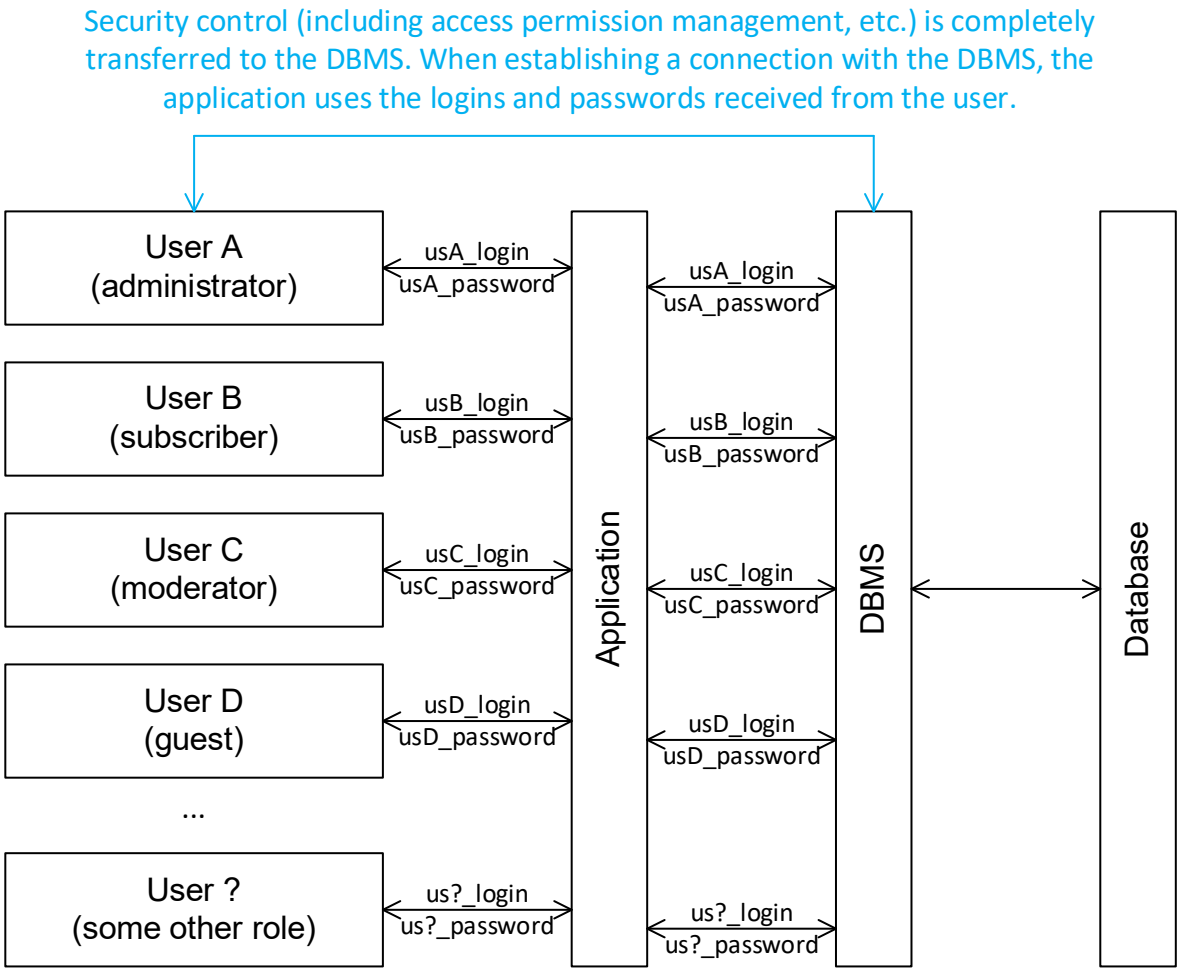


Figure 4.3.b — Working with the DBMS via multiple credentials

Yes, this approach is much more difficult to implement, it requires not only ongoing DBMS administration (at least in terms of managing the existing users and their permissions), but also requires serious modifications in the database itself (e.g., an additional level of abstraction can be created for data access in the form of views{331} and stored procedures{353} — such approach allows a much more flexible management of permissions and data protection).

But despite its noticeably greater complexity, this approach allows to:

- ensure a significantly higher level of security;
- eliminate the need to duplicate the security model across multiple individual applications (the model is implemented once at the database/DBMS level and applies to all clients);
- avoid deliberate or accidental “bypassing” of the security system (e.g., when working directly with the database during maintenance or troubleshooting).

Encodings

One of the most common mistakes beginners make when designing databases is not paying enough attention to character encodings.

The situation is greatly exacerbated by the fact that developers whose native language is different from English often use localized software[207] (including the operating system), which leads to automatic configuration of the DBMS so that on the developer's computer "everything works". And when transferred to a production server, it stops.

If the encodings for the database, tables (and even individual table fields in some DBMSes) are not specified explicitly when designing the database, all encodings will be set to "default", i.e., taken from the settings of the DBMS into which the database is imported, when transferring the database schema to the DBMS.

Let's demonstrate the consequences of this with a simple example.

Suppose we use MySQL, and the default encoding on the developer's computer is `utf8`. At the same time, on one of the servers on which the created database will run, the default encoding is `latin1`.

To make it clear, let's simplify the example and create just one table with one text field:

MySQL — Incorrect table creation (no encoding specified)

```
1    CREATE TABLE `words`
2    (
3      `word` VARCHAR(255) NULL
4    )
```

Let's fill this table with data (or rather, let's try unsuccessfully; we'll use "Pear", "Apple", "Orange" in Russian to keep UTF8 characters' representation):

MySQL — Error when adding UTF8 text

```
1    INSERT INTO `words`
2                (`word`)
3    VALUES      ('Груша'),
4                ('Яблоко'),
5                ('Апельсин')
```

At this very stage we already get an error message:

MySQL — Error message

```
1    Error Code: 1366. Incorrect string value:
2    '\xD0\x93\xD1\x80\xD1\x83...' for column 'word' at row 1
```

In this case, it is safe to say that we are still very lucky. Other circumstances could have led to a situation where data insertion was successful, but ordering and searching "didn't work", i.e., gave wrong results.

And all you had to do was not be lazy and specify the character encoding in your design tool, so that the query to create a table would look like this (note lines 5 and 6 of the query):

MySQL — Correct table creation (encoding is specified)

```
1    CREATE TABLE `words`
2    (
3      `word` VARCHAR(255) NULL
4    )
5    DEFAULT CHARACTER SET = utf8
6    COLLATE = utf8_general_ci
```

[207] Please never do this. If you are developing software and/or databases, all software on your computer must be in English. No exceptions.

Now queries for adding, ordering, searching, etc. will result in the correct expected results.

Yes, this problem is very easy to spot. But if you're developing an application whose many copies will be used by a large number of your customers (and you can't know their DBMS settings in advance in each case), get ready for massive loads of angry messages of "nothing works". Or adjust encodings wherever it's possible.



I appeal to those who say, "it always worked without it". No. It didn't work, it "just luckily happened". That is, until now you were just lucky. Now you know how to improve the reliability of the solutions you develop, but how you proceed is up to you, of course.

Storage engines

Storage engines have already been mentioned before{30}, and here we will only point out that all the problems and recommendations just described on the example of encodings fully apply to them.

If to rely on default settings, it is easy to find oneself in a situation where the DBMS behavior is not what you expected at all. There may be problems with performance, referential integrity control (yes, there are still storage engines that do not support such control[208]), transactions, table locks on modification operations, replication, scaling, etc., etc.

The storage engine completely defines the behavior of the database at the lowest level: the level of the files which store the data. Therefore, it would be very unwise to rely on some defaults. On the contrary, you should always specify the storage engine explicitly, if your chosen DBMS allows it.

For example, in MySQL, the `ENGINE` parameter in the table creation syntax is responsible for this. You just have to explicitly specify it using the design tool of your choice so that this information will be included in the resulting SQL query that will be used to create your database (see line 5 in the example query below):

MySQL — Correct table creation (storage engine is specified)

```
1  CREATE TABLE `words`
2  (
3    `word` VARCHAR(255) NULL
4  )
5  ENGINE = InnoDB
6  DEFAULT CHARACTER SET = utf8
7  COLLATE = utf8_general_ci
```

Indexes

An extensive section of this book has already been devoted to indexes{103}. From the information presented there it follows that there is a wide range both of the varieties of indexes and of their parameters.

Many indexes (e.g., unique ones{106} or on fields which obviously will be used for search operations very often), as a rule, are already "visible" at the datalogical level of modeling — and are created there as well. But that's why the physical level exists, not only to check once again if some of the "obvious" indexes are not forgotten, but also to perform a series of additional actions:

- fill the database with test data, as close to real data as possible (both in volume and content);
- perform load testing on the basis of already available information about typical queries{384};
- identify the bottlenecks{384} in database performance that can be eliminated by using indexes;

[208] E.g., MyISAM storage engine in MySQL.

- create the necessary indexes;
- make a note for the future of the need to periodically check the effectiveness of the created indexes, both during the development of the database and applications working with it, and in the real work of the created project.

The creation and deletion of indexes (with the exception of unique{106} and primary{107} ones) affects not the database structure and logic, but its performance. Therefore, you can safely make the appropriate changes at any time (although it is assumed that with proper design this moment occurs before the real users start working with the database).

The main difficulty lies in the fact that when fine-tuning indexes, it is necessary not only to conduct a lot of experiments, but also to carefully study the technical documentation for a particular version of a particular DBMS, because the solutions you know from one situation often behave quite differently in another one.

DBMS settings

Here we have reached the limit in our ability to formulate any general recommendations. The question “how to set up a DBMS?” without specifying hundreds of additional details sounds as abstract and ridiculous as, for example, asking “how to cook food?” or “how to draw a picture?”.

The most honest advice that can be given to novice database developers is this: do not change any DBMS settings unless you clearly understand exactly what you are doing, why, what for, and how. In most cases, the default DBMS settings are “optimized” for a wide range of typical solutions, which is likely to include your database.

If the product you are creating goes beyond “typical”, then you should consider its features when studying the documentation and experimenting with the DBMS settings.

And be sure to keep two things in mind:

- create backups before each change;
- test everything repeatedly in a test environment before making changes to the settings of a DBMS with which real users are already working.



Task 4.3.a: formulate a list of questions, the answers to which would help you improve the physical level of the “Bank{399}” database modeling.

4.3.2. Design Tools and Techniques at the Physical Level

It was noted in the previous chapter that at this level of modeling we are interested in access permissions, encodings, storage engines, indexes, and DBMS settings.

As for techniques and approaches to making necessary decisions, nothing fundamentally new is added here — we still have to collect information from:

- the customer — to fully understand the specifics of the project, the boundaries of the budget (which may significantly affect the choice of available DBMS configurations), the typical scenarios of work with the database, etc.;
- developers of the application interacting with the database (or applications, if there are several of them) — to more accurately predict the typical form of the load on the database, to see the specifics of the queries, to evaluate the security requirements, etc.;
- technical specialists from the data center (or cloud service) where the DBMS with our database will be located — in order to know in advance all the key limitations and options available to us to configure the DBMS and infrastructure.

Since the information we are interested in is very diverse in composition and presentation, we can use any of the tools discussed earlier in this section for its organization and integration into the database schema.

At a minimum, the management of encodings, storage engines and indexes is available in almost any modeling tool at the datalogical level{296}. And those tools that were marked as too narrowly specialized at the datalogical level, on the contrary, may be the most useful here.

As for managing access permissions and DBMS settings, the most beneficial tool here would be one that allows you to automatically configure these parameters every time you generate a database and/or deploy it to the DBMS. That is, such a tool must be able to automatically execute a set of SQL queries at the right time, modify text files, make changes to the Windows registry, etc.

DevOps[209] tools have all of these features (and many others). As their number is huge and the speed of their development and change is enormous, the list of specific recommended tools will be outdated literally every week, but nothing prevents you from asking more experienced colleagues for recommendations or simply “googling” the best tool for your specific situation at any time.

A few illustrative examples will be presented in the next chapter, and this one will conclude with a small “cheat sheet” that summarizes the material of the last two chapters in a brief form and also contains general advice (which will be most useful for beginners).

[209] **DevOps** — a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development lifecycle and provide continuous delivery with high software quality. (“Wikipedia”) [https://en.wikipedia.org/wiki/DevOps]

What	When to start considering	When to finish implementing	Tools	General advice
Access permissions	At the infological (conceptual) level	By the moment of DB commissioning	SQL scripts + DevOps tools	Unless you are creating a really complex and large-scale project, take the approach shown in figure 4.3.a{305} and save time and effort
Encodings	At the infological (conceptual) level	By the moment of acceptance testing	Design tool used at the datalogical level	Make sure you set encodings explicitly wherever your DBMS allows
Storage engines	At the datalogical level	By the moment of acceptance testing	Design tool used at the datalogical level	Make sure you set storage engines explicitly wherever your DBMS allows
Indexes	At the datalogical level	Can be modified even during DBMS operation	Design tool used at the datalogical level + SQL scripts + DevOps tools + load testing tools	Be very sure to carry out some research (including load testing), do not assume that some decisions will just seem logical — there can be many surprises
DBMS settings	At the datalogical level	Can be modified even during DBMS operation	SQL scripts + DevOps tools	Do not change the DBMS settings unless you know exactly what you are doing, why, and how

So, let's move on to examples.



Task 4.3.b: using Sparx Enterprise Architect, finalize the "Bank{395}" database schema so that it reflects all the necessary nuances of the physical design level (see handouts{4}).

4.3.3. Example of Design at the Physical Level

At the physical modelling level, it is quite difficult to visualize both the design process and its result. Especially since our educational database will often not need any special refinements. But we will try to cover the whole process as fully as we can in text and pictures.

Access permissions

Yes, in our particular case it is much more rational to connect to the DBMS with a single credentials (see figure 4.3.a{305}), but if we decided to go the more complicated way (see figure 4.3.b{306}), we could implement it as follows.

First, it is necessary to define some global user roles and a list of their access permissions regarding database objects.

For the sake of brevity, we will use the common abbreviations derived from the famous acronym CRUD[210]: C — access permission to create a record, R — access permission to read a record, U — access permission to update a record, D — access permission to delete a record.

	Application	Guest	User	Moderator	Administrator
age_restriction	R	R	R	CRUD	CRUD
ban	R	R	R	R	CRUD
bonus	R	R	R	R	CRUD
comment	R	R	R	CRUD	CRUD
download_link	R	R	R	CRUD	CRUD
file	R	R	CRUD	CRUD	CRUD
file_category	R	R	R	CRUD	CRUD
file_permission	R	R	CRUD	CRUD	CRUD
group	R	R	R	CRUD	CRUD
ip_blacklist	CRUD	-	-	-	CRUD
log	C	-	-	-	CRUD
log_archive	C	-	-	-	CRUD
m2m_file_file_category	R	R	CRUD	CRUD	CRUD
m2m_user_group	R	R	R	CRUD	CRUD
m2m_user_role	R	R	R	CRUD	CRUD
mark	R	R	CRUD	CRUD	CRUD
operation	R	-	-	-	CRUD
page	R	R	R	CRUD	CRUD
permission	R	R	R	R	CRUD
role	R	R	R	R	CRUD
statistics	R	R	R	R	CRUD
user	CRUD	R	CRUD	CRUD	CRUD

Obviously, some operations (e.g., logging) must be performed by the application anyway — regardless of which user is currently working with it — so, we have to create a separate role (and a separate user) just for the application itself. On behalf of this user, the registration of other users will be performed.

Now it is necessary to create the appropriate SQL code. For the sake of brevity, here is the code for the roles “Application” and “Administrator” (let’s leave the creation of similar code for the roles “Guest”, “User”, and “Moderator” to the task 4.3.d{322} for your own study).

[210] CRUD — Create, Read, Update, Delete.

MySQL — Creating users and managing their access permissions

```
1    -- 1) User for the "Application" role:
2    DROP USER IF EXISTS 'feapp'@'localhost';
3    CREATE USER 'feapp'@'localhost' IDENTIFIED BY '<strong password>';
4
5    GRANT SELECT ON `age_restriction` TO 'feapp'@'localhost';
6    GRANT SELECT ON `ban` TO 'feapp'@'localhost';
7    GRANT SELECT ON `bonus` TO 'feapp'@'localhost';
8    GRANT SELECT ON `comment` TO 'feapp'@'localhost';
9    GRANT SELECT ON `download_link` TO 'feapp'@'localhost';
10   GRANT SELECT ON `file` TO 'feapp'@'localhost';
11   GRANT SELECT ON `file_category` TO 'feapp'@'localhost';
12   GRANT SELECT ON `file_permission` TO 'feapp'@'localhost';
13   GRANT SELECT ON `group` TO 'feapp'@'localhost';
14   GRANT INSERT, SELECT, UPDATE, DELETE ON `ip_blacklist`
15                                       TO 'feapp'@'localhost';
16   GRANT INSERT ON `log` TO 'feapp'@'localhost';
17   GRANT INSERT ON `log_archive` TO 'feapp'@'localhost';
18   GRANT SELECT ON `m2m_file_file_category` TO 'feapp'@'localhost';
19   GRANT SELECT ON `m2m_user_group` TO 'feapp'@'localhost';
20   GRANT SELECT ON `m2m_user_role` TO 'feapp'@'localhost';
21   GRANT SELECT ON `mark` TO 'feapp'@'localhost';
22   GRANT SELECT ON `operation` TO 'feapp'@'localhost';
23   GRANT SELECT ON `page` TO 'feapp'@'localhost';
24   GRANT SELECT ON `permission` TO 'feapp'@'localhost';
25   GRANT SELECT ON `role` TO 'feapp'@'localhost';
26   GRANT SELECT ON `statistics` TO 'feapp'@'localhost';
27   GRANT INSERT, SELECT, UPDATE, DELETE ON `user` TO 'feapp'@'localhost';
28   GRANT CREATE USER ON *.* TO 'feapp'@'localhost' WITH GRANT OPTION;
29
30   -- 2) Users for the roles of "Guest", "User", "Moderator"
31   --    will be managed in the same way.
32
33   -- 3) User for the "Administrator" role:
34   DROP USER IF EXISTS 'feadmin'@'localhost';
35   CREATE USER 'feadmin'@'localhost' IDENTIFIED BY '<strong password>';
36
37   GRANT ALL PRIVILEGES ON * TO 'feadmin'@'localhost';
```

If we want to use an even more sophisticated access permission control system, we can create views{331} and stored procedures{353}, give appropriate access permissions only to these objects, and deny direct access to the tables to anyone at all.

Encodings

The logic of encoding management depends very much on the capabilities of your particular design tool and on whether the encodings will be different for different database objects.

In the extreme case, you will have to specify the encodings manually for each table. If you are using Sparx Enterprise Architect, you can do this through the so-called “tagged values” of the tables. A sample sequence of steps is shown in figure 4.3.c.

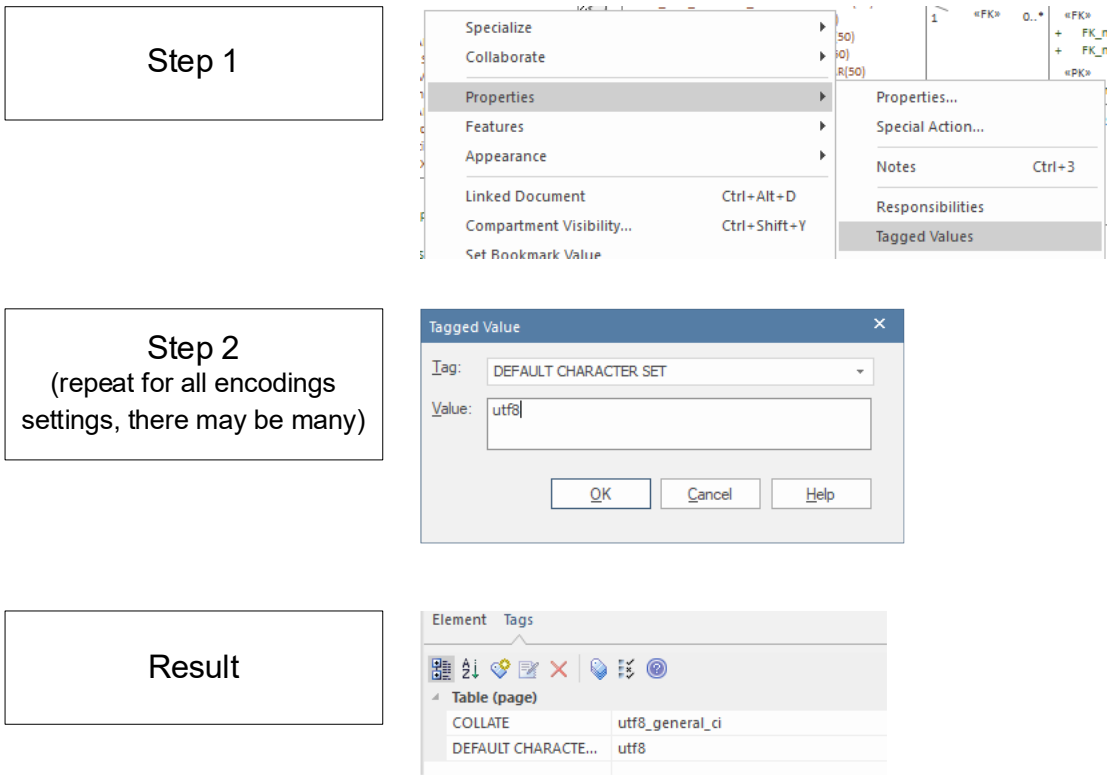


Figure 4.3.c — Setting encodings using tagged values

If all database objects have the same encoding, perhaps the most reliable way is to create a stored procedure that controls the encoding of all objects of interest (usually just tables) and run it during database creation.

The code to create and call such a procedure for MySQL may look like this[211].

[211] Details on how such stored procedures work can be found in section 5.2 “Using stored procedures” of the “Using MySQL, MS SQL Server and Oracle by Examples” book (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

MySQL — Stored procedure for managing the encodings of all tables in the database

```
1   DROP PROCEDURE IF EXISTS SET_ENCODING_TO_ALL_TABLES;
2
3   DELIMITER $$
4   CREATE PROCEDURE SET_ENCODING_TO_ALL_TABLES
5      (IN default_charset_name VARCHAR(150), IN collation_name VARCHAR(150))
6   BEGIN
7    DECLARE done INT DEFAULT 0;
8    DECLARE tbl_name VARCHAR(200) DEFAULT '';
9    DECLARE all_tables_cursor CURSOR FOR
10    SELECT `table_name`
11      FROM `information_schema`.`tables`
12      WHERE `table_schema` = DATABASE()
13        AND `table_type` = 'BASE TABLE';
14   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
15
16   OPEN all_tables_cursor;
17   tables_loop: LOOP
18    FETCH all_tables_cursor INTO tbl_name;
19    IF done
20     THEN LEAVE tables_loop;
21    END IF;
22
23    SET @alter_table_query = CONCAT('ALTER TABLE `', tbl_name,
24        '` CONVERT TO CHARACTER SET \'', default_charset_name,
25        '\' COLLATE \'', collation_name, '\'');
26
27    PREPARE alter_table_stmt FROM @alter_table_query;
28    EXECUTE alter_table_stmt;
29    DEALLOCATE PREPARE alter_table_stmt;
30   END LOOP tables_loop;
31   CLOSE all_tables_cursor;
32  END;
33  $$
34  DELIMITER ;
35
36  CALL SET_ENCODING_TO_ALL_TABLES('utf8', 'utf8_general_ci');
```

In this particular case, we could do without the stored procedure (see task 4.3.e{322} for self-study), but this way we get a good template for the future, which can be used with minimal modifications and to solve more complex problems.

Storage engines

Managing storage engines is exactly the same as managing encodings. We can either configure them in advance in the design tool (see figure 4.3.d for a brief tutorial on how to do this in Sparx Enterprise Architect) or use SQL code.

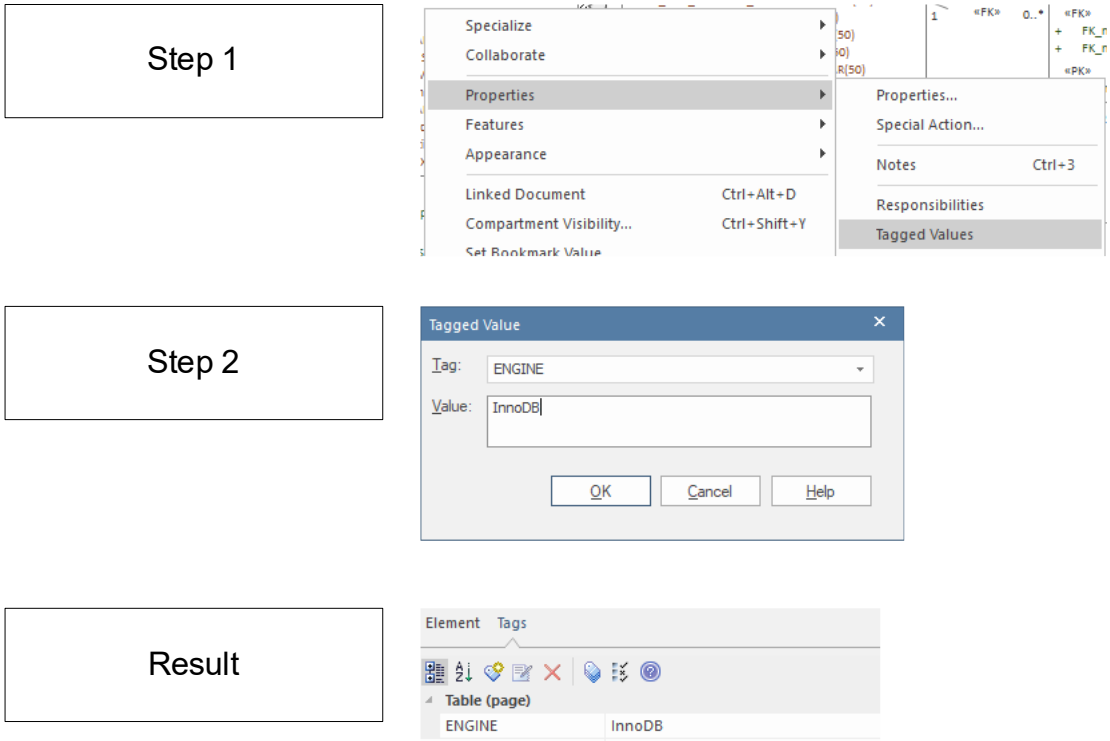


Figure 4.3.d — Choosing a storage engine using tagged values

Since we already have a nearly finished stored procedure, let's make some small changes to it so that it sets not only the encoding of all the tables, but also the storage engines.

MySQL — Improved stored procedure for managing encodings and storage engines of all tables

```
DROP PROCEDURE IF EXISTS SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES;

DELIMITER $$
CREATE PROCEDURE SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES
          (IN default_charset_name VARCHAR(150),
           IN collation_name VARCHAR(150),
           IN storage_engine VARCHAR(150))
BEGIN
 DECLARE done INT DEFAULT 0;
 DECLARE tbl_name VARCHAR(200) DEFAULT '';
 DECLARE all_tables_cursor CURSOR FOR
  SELECT `table_name`
    FROM `information_schema`.`tables`
    WHERE `table_schema` = DATABASE()
      AND `table_type` = 'BASE TABLE';
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

 OPEN all_tables_cursor;
 tables_loop: LOOP
  FETCH all_tables_cursor INTO tbl_name;
  IF done
   THEN LEAVE tables_loop;
  END IF;

  SET @alter_table_encoding_query = CONCAT('ALTER TABLE `', tbl_name,
    '` CONVERT TO CHARACTER SET \'', default_charset_name,
    '\' COLLATE \'', collation_name, '\'');
  SET @alter_table_engine_query = CONCAT('ALTER TABLE `', tbl_name,
    '` ENGINE = \'', storage_engine, '\'');

  PREPARE alter_table_encoding_stmt FROM @alter_table_encoding_query;
  PREPARE alter_table_engine_stmt FROM @alter_table_engine_query;

  EXECUTE alter_table_encoding_stmt;
  EXECUTE alter_table_engine_stmt;

  DEALLOCATE PREPARE alter_table_encoding_stmt;
  DEALLOCATE PREPARE alter_table_engine_stmt;
 END LOOP tables_loop;
 CLOSE all_tables_cursor;
END;
$$
DELIMITER ;

CALL SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES('utf8',
                                                   'utf8_general_ci', 'InnoDB');
```

Obviously, configuring the storage engine is not just a matter of selecting the storage engine itself — there are a huge number of additional parameters which need to be considered in each and every situation. Here the only right way is to carefully read the documentation[212], do some research, experiment.

[212] For example, the documentation on configuring InnoDB storage engine for MySQL 8.0 can be found here: https://dev.mysql.com/doc/refman/8.0/en/innodb-configuration.html.

Indexes

As the relevant section of this book[103] shows, there are a huge number of indexes, differing in purpose and objectives, the logic of operation, physical organization, and a variety of other parameters.

We have already created at the datalogical modelling level the most obvious indexes, which are responsible for the uniqueness of the values of some fields and accelerate the JOIN operation (they are automatically created on the foreign keys).

Since the creation and deletion of indexes (with the exception of unique[106] and primary[107] ones) does not affect the structure and logic of the database, the necessary changes can be made at any time; but in the vast majority of cases it requires not only a thorough analysis of the situation, but also the study of query execution plans and a series of experiments.

Let's illustrate this logic with one example (in other cases the actions will be the same, only the collected data will differ).

Let's take a look at the `file` table. It is both one of the largest in our database and one of the most loaded (in terms of the number of accesses to it).

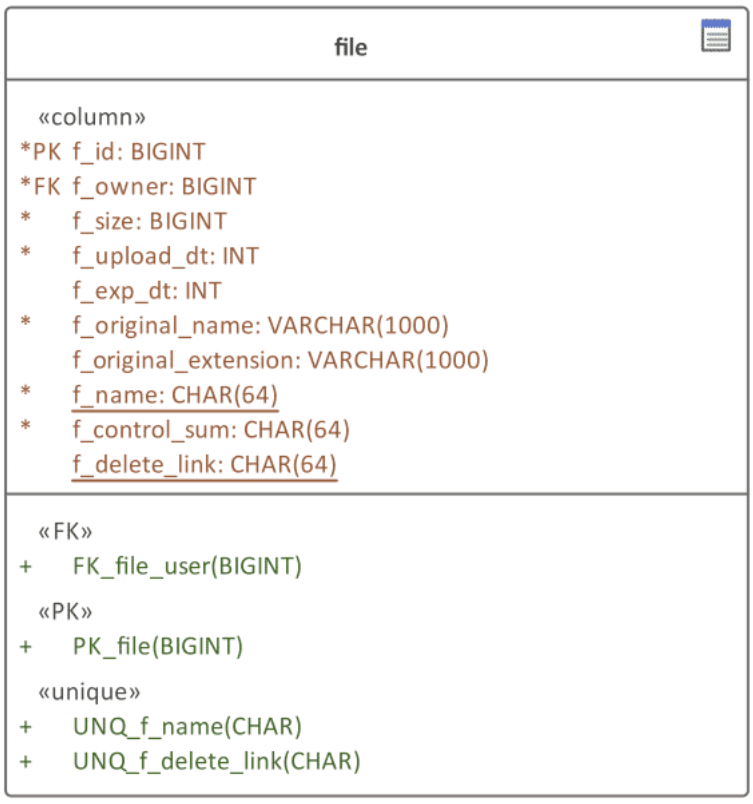


Figure 4.3.e — The `file` table before creating additional indexes

Assume that the most frequent operations with this table will be:

- ~~search for files of a specific user (this index is created automatically);~~
- deleting expired files;
- searching for files by name and/or extension;
- ordering the list of files by size;
- ordering the list of files by the upload datetime.

Although the result is completely self-evident, let's check on the example of deleting expired files to see what the query's execution plan will look like.

MySQL — Obtaining the query execution plan

```
1    EXPLAIN DELETE FROM `file` WHERE `f_exp_dt` <= UNIX_TIMESTAMP()
```

Quite expectedly, the plan shows that the DBMS will have to scan the whole table to find the necessary records (the values of the “possible_keys” and “key” parameters are **`NULL`**):

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
DELETE	file	ALL	NULL	NULL	NULL	NULL	1000000	100.00	Using where

Let’s create an index on the **`f_exp_dt`** field using Sparx Enterprise Architect tool (SQL code will be presented later). Now the query plan looks different:

select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
DELETE	file	ALL	IDX_f_exp_dt	IDX_f_exp_dt	4	NULL	1000000	50.00	Using where

And it remains to conduct an experiment. Let’s preliminarily delete the index on the **`f_exp_dt`** field in a database with a million records in the table, and determine the median execution time (on 100 attempts) of a query to find all expired files.

Then let’s create an index on the **`f_exp_dt`** field again and determine this median time again. We are not interested in absolute values (because they strongly depend on the hardware), but the final result cannot but please us: the speed of the query execution has increased by 121 times.

No index, s	With index, s	Difference, times
7.00951E-03	5.79357E-05	120.98775021274

Completely similar operations should be performed for the **`f_size`** and **`f_upoload_dt`** fields (to speed up the ordering of the list of files by size and by the time of uploading, respectively).

In the case of searching files by name and/or extension, we should assume that searching by extension (type) will be performed more often, and if we do not want to create several indexes, it will be more profitable to put the **`f_original_extension`** field at the first place in the index, and the **`f_original_name`** field at the second place. Earlier{47} it was explained in detail why the order of fields in composite indexes is critical.

After performing all the necessary procedures in Sparx Enterprise Architect, the **`file`** table looks like the following (see figure 4.3.f).

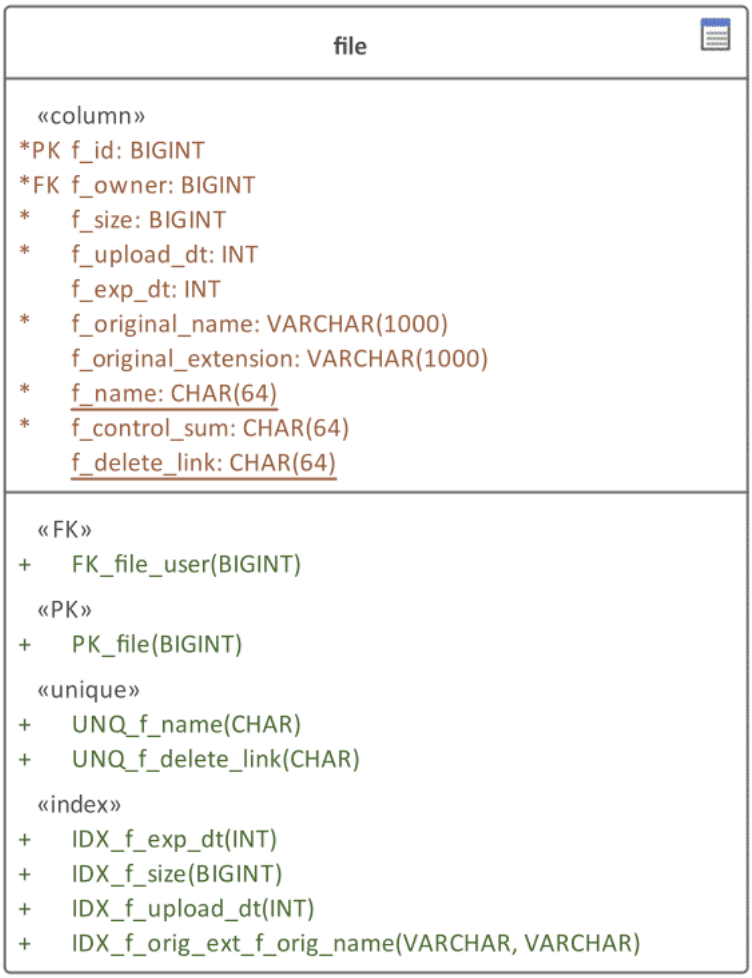


Figure 4.3.f — The `file` table after creating additional indexes

If you prefer to create indexes using SQL code, you can do it as follows:

MySQL — SQL code to create the just discussed indexes

```
1   ALTER TABLE `file`
2    ADD INDEX `IDX_f_exp_dt` (`f_exp_dt` ASC);
3
4   ALTER TABLE `file`
5    ADD INDEX `IDX_f_size` (`f_size` ASC);
6
7   ALTER TABLE `file`
8    ADD INDEX `IDX_f_upload_dt` (`f_upload_dt` ASC);
9
10  ALTER TABLE `file`
11   ADD INDEX `IDX_f_orig_ext_f_orig_name`
12           (`f_original_extension` ASC,`f_original_name` ASC);
```

Now it remains to repeat the same considerations and research for the rest of the database tables (which is one of the tasks for self-study, see task 4.3.f{322}).

It is easy to notice that technically there is nothing difficult in creating indexes, the difficulty here is the need to do a lot of considerations and experiments, as well as to carefully study the technical documentation for a particular version of a particular DBMS.

DBMS settings

In our example we use MySQL, and we can safely say that we are very lucky — MySQL as a whole is configured through SQL queries, environment variables and special configuration files.

In the case of other DBMSes, we would have to configure several separate applications, using not only SQL queries, environment variables and configuration files (in different formats), but also the Windows registry and even separate external services.

But even in MySQL there are a lot of options[213]. For illustration, here is a list of configuration files that this DBMS uses.

File	Purpose
%WINDIR%\my.ini, %WINDIR%\my.cnf	Global settings
C:\my.ini, C:\my.cnf	Global settings
BASEDIR\my.ini, BASEDIR\my.cnf	Global settings
defaults-extra-file	Used when running with --defaults-extra-file parameter
DATADIR\mysqld-auto.cnf	Persistent system variables for SET PERSIST or SET PERSIST_ONLY

If you run the `SHOW VARIABLES` SQL query, you will see over 500 different settings that MySQL uses in its work. Sometimes, the `SHOW STATUS` SQL query, which shows over 300 parameters of MySQL's current state, can be useful to analyze them and understand them better.

To give an example of changing settings, let's assume we have the whole DBMS at our disposal (i.e., we won't destroy the logic of other users' work) and switch the default encodings.

This can be done using SQL queries in three ways:

MySQL — SQL-code for temporary switching of default encodings

```
1    -- Approach 1:
2    SET character_set_server = utf8mb4
3    SET collation_server = utf8mb4_general_ci
4
5    -- Approach 2:
6    SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
7
8    -- Approach 3 (settings are saved even if the server is restarted):
9    SET PERSIST character_set_server = utf8mb4
10   SET PERSIST collation_server = utf8mb4_general_ci
```

You can also edit the `my.ini` file and add the following lines to the `[mysqld]` options group:

```
character_set_server = utf8mb4
collation_server = utf8mb4_general_ci
```

Then restart the MySQL service.

It remains to remind you of universal advice: do not change any DBMS settings unless you clearly understand what, why, what for, and how you are doing, because in most cases the default DBMS settings are "optimized" for a wide range of typical solutions, which most likely includes your database.

[213] It's a good idea to start by reading these sections of the documentation: https://dev.mysql.com/doc/refman/8.0/en/program-options.html and https://dev.mysql.com/doc/refman/8.0/en/server-administration.html.



Task 4.3.c: refine the “Bank{395}” database schema so as to eliminate all design flaws found in it at the physical modelling level (as far as possible without involving a “hypothetical customer” to get answers to the questions that arise).



Task 4.3.d: earlier{312} in this section we discussed an example of SQL code for creating and managing the access permissions of the “Application” and “Administrator” roles. Write a similar SQL code for the “Guest”, “User”, and “Moderator” roles.



Task 4.3.e: earlier{314} in this section we discussed the creation of a stored procedure that controls the encodings of all database objects of interest. It was also noted{315} that the same effect could be achieved without a stored procedure. Write SQL code that performs the same actions but is not part of a stored procedure.



Task 4.3.f: earlier{318} in this chapter we conducted a series of studies and experiments on working with indexes on the file table. Perform similar studies and experiments for the other tables in the filesharing service database.

4.4. Reverse engineering

4.4.1. Purpose and Objectives of Reverse Engineering

As the term itself suggests, reverse engineering helps us to get information about structure, settings, logic, etc. on the basis of the existing and working database.

Why is this information needed? There could be several reasons:

- The customer has given us the project, previously implemented by another contractor, for support and revision.
- It is necessary to make changes to the existing project, but for various reasons the documentation does not give us answers to our questions.
- We study the old project in order to transfer some of its functionality into the new one.
- And other rarer cases.

In direct modeling we always have to go through all three levels (infological (conceptual), datalogical, physical) and more than once.

The situation is different with reverse engineering.

First, we have to “go back” to the previous level much less often, because with the right work organization and the right tools we can collect all the necessary information at once.

Second, we quite often don’t need to get models of all levels: for example, it makes no sense to build a schema of an existing database if we just want to solve a couple of performance problems and to adjust indexes and/or storage engines.

If we reduce the above to a couple of brief formulations, we get:

- the purpose of reverse engineering is to obtain information about the existing database, sufficient to perform the upcoming work;
- the objective of reverse engineering is to collect the necessary information in its entirety and to present it in a form that is convenient for perception and further use.

A little beyond the scope of this topic (but still worthy of mentioning) is the analysis of the database and DBMS settings to eliminate some technical problems and optimize performance.

Technically, such an analysis can also be attributed to the reverse engineering of a database, but in practice it is not considered as such because in the process of such an analysis rather narrow, specific technical problems are considered.

And to solve these problems, on the one hand we need very specific information (which may be completely insufficient for understanding the overall structure of the database, its relationship to the subject area, etc.), and on the other hand we may need a lot and lot of such information (much more than would be needed if we were engaged in “classic” reverse engineering).

Anyway, if you investigate a working system (and the database as part of it), you collect information.

The type, volume, form of representation, and ways of retrieving this information can be very different, so the question of finding the most appropriate tools arises.



Task 4.4.a: formulate a list of questions that came up most often in your analysis of the “Bank{395}” database.

4.4.2. Tools and Techniques of Reverse Engineering

Since we are talking specifically about reverse engineering here, let's start at the physical modelling level.

As a rule, we are talking about a variety of diagnostic tools that allow us to get information about the DBMS settings, collect statistics on its work, evaluate performance parameters, see the occurring errors and determine their root cause.

Such tasks are best handled by highly specialized tools developed by DBMS manufacturers or special (usually very expensive) commercial tools. However, a huge amount of data can be collected by regular SQL queries. The next chapter will present a corresponding example.

The situation looks a little easier with the datalogical level, because most tools for database design also support reverse engineering: the tool connects to the DBMS and retrieves information about tables, relationships, indexes, stored procedures, etc.

The main difficulty here is the convenience of information presentation. This problem is not so acute at the physical modelling level, because there we (as a rule) extract a small amount of data, which will be used for a relatively short time, so the inconvenience of the presentation of these data can be "tolerated".

But if we have gone up to the datalogical modelling level, then (most likely) we need a lot of information, and we plan to use it for quite a long time. All the more, some of the data from the physical modelling level will also be put into our sampling, because some database structures cannot be described correctly if we ignore the low-level technical details.

And here we are in for a big disappointment, because neither narrowly specialized{297}, nor universal{298} tools are still able to present the collected information as conveniently for human perception as the originally created and designed human database schema could look.

The situation is more or less "bearable" if we're talking about a database with a couple of dozen tables, but when their number is measured in hundreds, at first you get a completely unreadable mishmash of chaotically arranged tables and links. And you will have to spend a lot of time on bringing this picture into a convenient form for further work.

Figure 4.4.a shows the result of the reverse engineering of a relatively small database (about 100 tables) done with DBeaver[214]. Yes, the tool did its best to somehow arrange the data in a digestible way. Although it turned out to be "usable", unfortunately, it is not convenient. That's why there is a lot of manual work to be done here in any case.

Special mention should be made of the situation where, after performing reverse engineering and making necessary edits to the database schema, we would like to automatically project these changes onto the actual working database.

While many tools support such functionality, it all depends on the nature of the change. So, for example, it's not hard to add a field to a table, or remove a field, or rename a view.

But how can we move a field from one table to another (without losing data)? How can we merge several fields into one or split one into several (again — without losing data)? As a rule, no tool provides ready-made solutions, because the algorithm for performing such an operation in each case will be unique and very specific.

So, again: there is a lot of manual work to be done (usually to write the appropriate SQL code, possibly with the creation of some temporary intermediate database states).

[214] "DBeaver" [https://dbeaver.io]

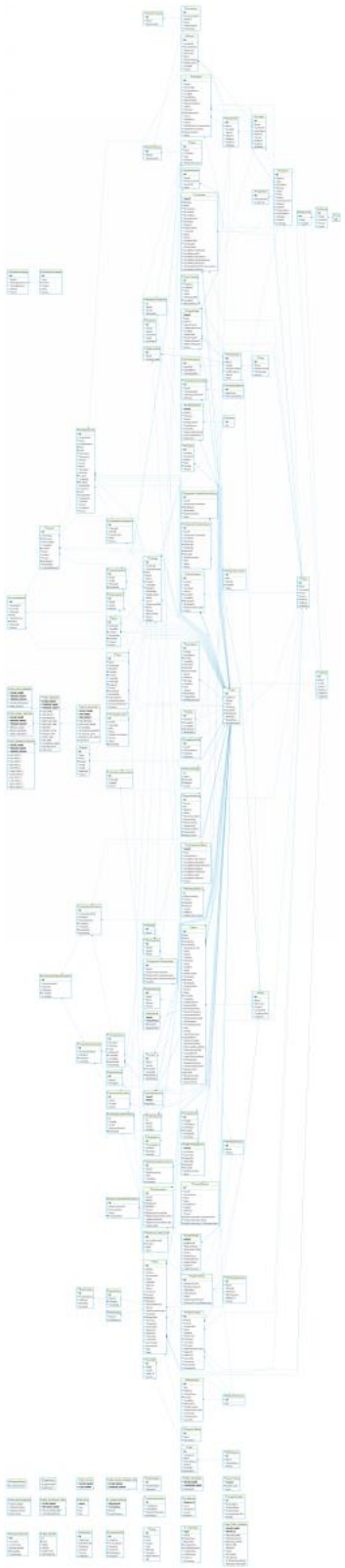


Figure 4.4.a — The result of reverse engineering

And finally, literally, a couple of words about the infological (conceptual) modelling level. If we are forced to go so high in the reverse engineering process, it is likely that we will either have to do some global redesign of the existing project or develop a new one altogether.

Therefore, it is safe to say that here we are, in fact, returning to “forward design”, only instead of the customer, we get information from a tool. We arrange the information in an easy-to-understand form and start discussing it with the customer, just as we would do in a classical “forward design”.



Task 4.4.b: using the ready-to-use SQL code for the “Bank{399}” database creation perform the same operation in MS SQL Server and Oracle, and then perform reverse engineering of the resulting database using any tool you like.

4.4.3. Example of Reverse Engineering

Since the operations presented here will be completely identical for all database objects, let's examine them using two illustrative examples: a stored procedure study and a table study.

As mentioned in the previous chapter, a lot of information can be obtained by executing SQL queries in any tool that allows it (even a trivial console client, although this is the worst possible choice). Since we use MySQL, let's perform this operation in the MySQL Workbench.

We can get the list of stored procedures and the list of tables in the database by the following SQL queries:

MySQL — Retrieving the list of stored procedures and the list of tables in the database

```
1    -- Retrieving the list of stored procedures:
2    SHOW PROCEDURE STATUS WHERE `db` = DATABASE()
3
4    -- Retrieving the list of tables:
5    SELECT `table_name`
6      FROM `information_schema`.`tables`
7     WHERE `table_schema` = DATABASE()
8       AND `table_type` = 'BASE TABLE'
```

When we have found the names of the stored procedure (`SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES`) and the table (`file`) of interest, we can retrieve the procedure code and a lot of information about the table.

MySQL — Retrieving the stored procedure code and table information

```
1    -- Retrieving the source code for creating a stored procedure:
2    SHOW CREATE PROCEDURE `SET_ENCODING_AND_STORAGE_ENGINE_TO_ALL_TABLES`
3
4    -- Retrieving the source code for creating a table:
5    SHOW CREATE TABLE `file`
6
7    -- Retrieving the list of table indexes:
8    SHOW INDEX FROM `file`
```

The source code for creating a stored procedure will look exactly as we wrote it to create the procedure — it was discussed earlier {317}.

The source code for creating the table will look like this. Note that all the comments that we carefully prescribed in the process of creating the database are still preserved here. And they can save a lot of time for someone who sees the structure of this table for the first time.

MySQL — Source code for creating the `file` table

```
CREATE TABLE `file` (
  `f_id` bigint(20) unsigned NOT NULL AUTO_INCREMENT
    COMMENT 'Global file identifier.',
  `f_owner` bigint(20) unsigned NOT NULL
    COMMENT 'File owner (FK).',
  `f_size` bigint(20) unsigned NOT NULL
    COMMENT 'File size (bytes).',
  `f_upload_dt` int(11) NOT NULL
    COMMENT 'Datetime of the file upload (Unixtime), up to seconds.',
  `f_exp_dt` int(11) DEFAULT NULL
    COMMENT 'Datetime of the file expiration (Unixtime), up to seconds.
             NULL to store forever.',
  `f_original_name` varchar(1000) NOT NULL
    COMMENT 'Original file name (without extension).',
  `f_original_extension` varchar(1000) DEFAULT NULL
    COMMENT 'Original file extension.',
  `f_name` char(64) NOT NULL
    COMMENT 'Server-side file name. SHA1 hash.',
  `f_control_sum` char(64) NOT NULL
    COMMENT 'File checksum. SHA256 hash.',
  `f_delete_link` char(64) DEFAULT NULL
    COMMENT 'Link for the file deletion (if the file was uploaded
             by unregistered user). SHA256 hash.',
  PRIMARY KEY (`f_id`),
  UNIQUE KEY `UNQ_f_name` (`f_name`),
  UNIQUE KEY `UNQ_f_delete_link` (`f_delete_link`),
  KEY `IDX_f_exp_dt` (`f_exp_dt`),
  KEY `IDX_f_size` (`f_size`),
  KEY `IDX_f_upload_dt` (`f_upload_dt`),
  KEY `IDX_f_orig_ext_f_orig_name` (`f_original_extension`,
                                    `f_original_name`),
  KEY `FK_file_user` (`f_owner`),
  CONSTRAINT `FK_file_user` FOREIGN KEY (`f_owner`)
  REFERENCES `user` (`u_id`) ON DELETE NO ACTION ON UPDATE NO ACTION
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Stored file.'
```

As for the indexes, their “source code” (if possible) is already presented in the source code for the table creation. But executing a query to show the list of indexes can give us a bit more information. Its result is a table like this.

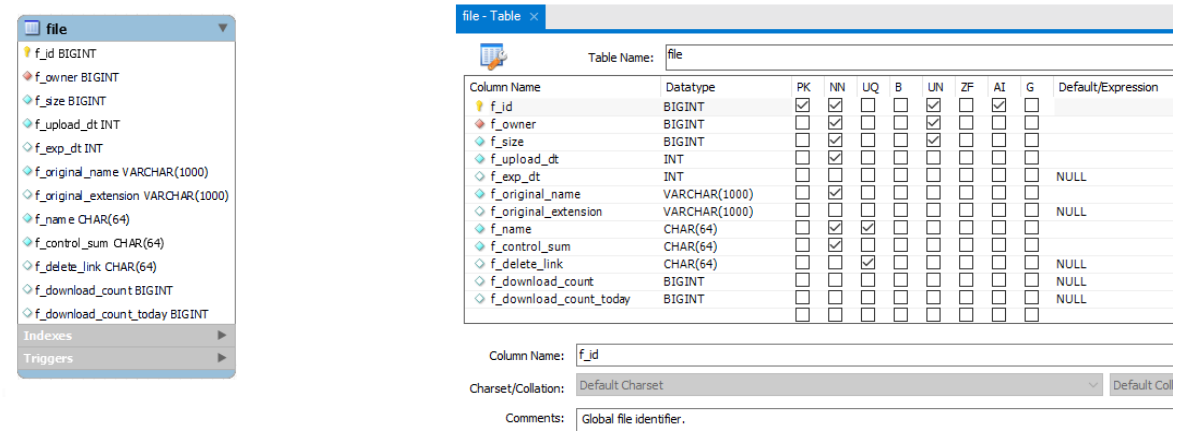
Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
0	PRIMARY	1	f_id	A	0	NULL	NULL		BTREE
0	UNQ_f_name	1	f_name	A	0	NULL	NULL		BTREE
0	UNQ_f_delete_link	1	f_delete_link	A	0	NULL	NULL	YES	BTREE
1	IDX_f_exp_dt	1	f_exp_dt	A	0	NULL	NULL	YES	BTREE
1	IDX_f_size	1	f_size	A	0	NULL	NULL		BTREE
1	IDX_f_upload_dt	1	f_upload_dt	A	0	NULL	NULL		BTREE
1	IDX_f_orig_ext_f_orig_name	1	f_original_extension	A	0	NULL	NULL	YES	BTREE
1	IDX_f_orig_ext_f_orig_name	2	f_original_name	A	0	NULL	NULL		BTREE
1	FK_file_user	1	f_owner	A	0	NULL	NULL		BTREE

Here we see information and sequence of fields in the index, encoding, information about indexed records and much, much more.

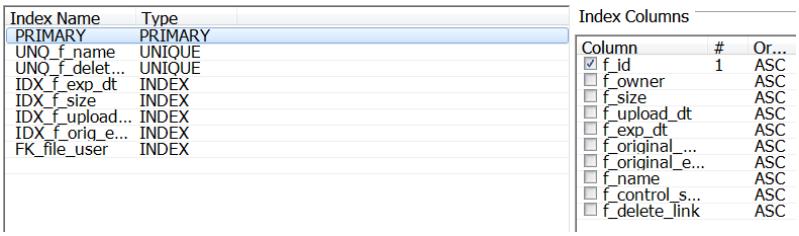
But the same information can be obtained by importing the entire database using the MySQL Workbench. In the menu select Database → Reverse Engineer and after performing the shown instructions we get the result shown in figure 4.4.b.

The `file` table

The `file` table detailed description



Indexes on the `file` table



Stored procedure

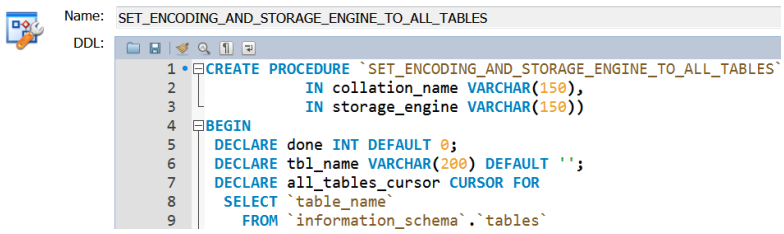


Figure 4.4.b — The result of a database import into MySQL Workbench

In fact, we are already at the stage of reverse engineering up to the datalogical level. Figure 4.4.c shows the overall picture (the whole database).

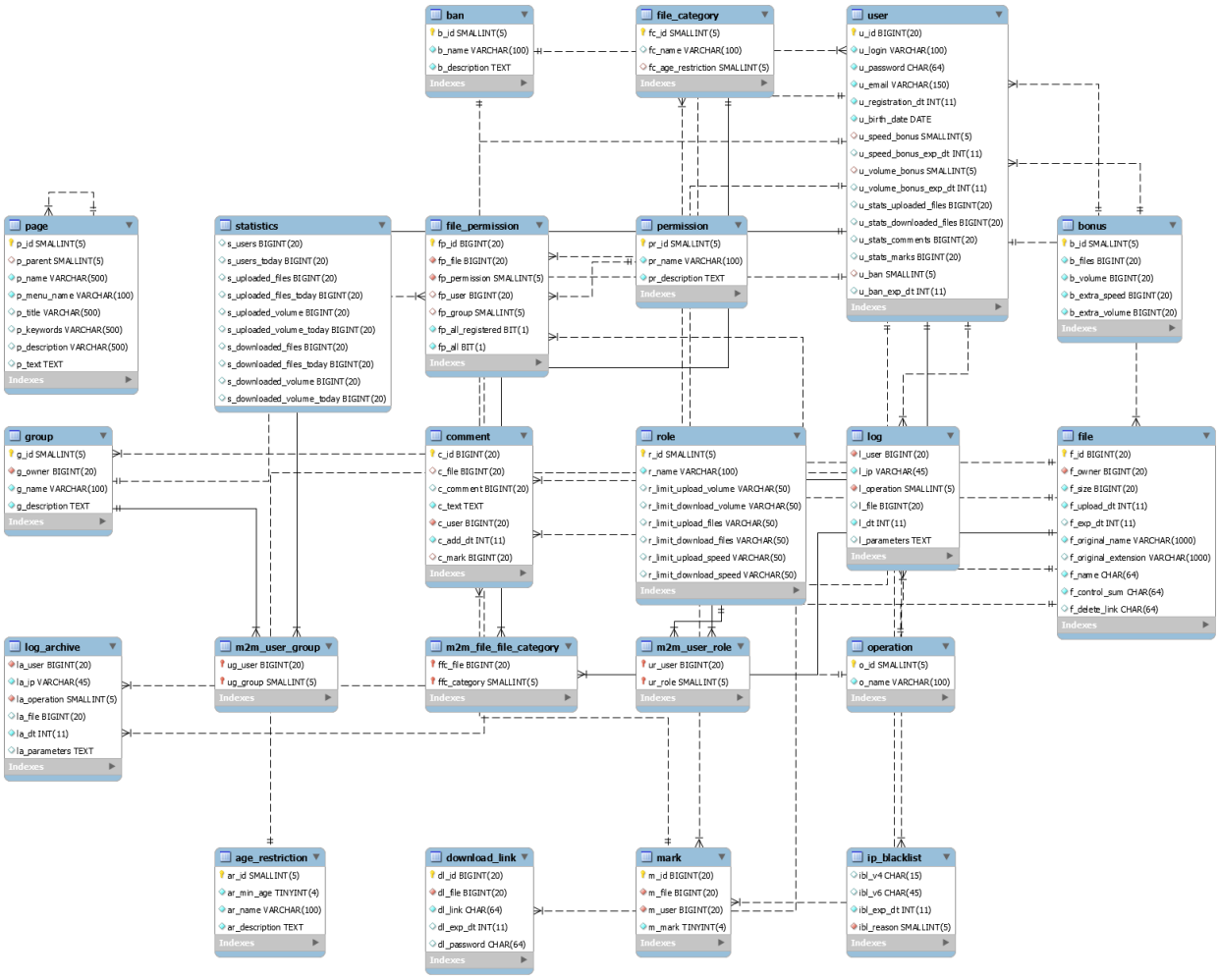


Figure 4.4.c — The result of importing the entire database into MySQL Workbench

By performing the last step (for brevity, for the `file` table only), we get the following textual description, familiar from the beginning of the design.

File:

- Id.
- Size (bytes).
- Creation datetime (up to seconds).
- Expiration datetime (up to seconds).
- Original name (without extension).
- Original extension.
- Name on server (sha256-hash).
- Checksum (sha256-hash).
- Deletion link (for unregistered users, sha256-hash).

It remains to get the same textual descriptions for the rest of the tables, and then the reverse engineering process will be completely finished.



Task 4.4.c: compare the initial “Bank{395}” database schema and the result of reverse engineering obtained in task 4.4.b{326}; make a list of differences, noting which version provides more information and is more readable.

Chapter 5: Additional Database Objects and Processes

5.1. Views

5.1.1. General Information on Views

So far, we have deliberately not used views so as not to complicate the examples considered, but views are ubiquitous in real databases, and their use provides a wide range of benefits.



View[215] — a virtual derived relation variable{22}, the value of which is the result of calculating the relational expression (query execution) specified when creating the view (such an expression must refer to at least one relation variable).

Simplified: an SQL query that can be executed by accessing it by a pre-specified name.

Let's at once consider another very important definition.



Materialized view[216] — a derived relation variable{22}, the value of which is the stored result of a pre-computed relational expression (query execution) specified when creating the materialized view (such an expression should refer to at least one relation variable). The recalculation and saving of the obtained result are done according to the rules determined during creation of the materialized view.

Simplified: an SQL query that can be executed by accessing it by a pre-specified name and the result of which is saved for later use.

Important! Both the term "materialized view" and the term "persistent view" are used to describe this concept with approximately equal frequency.

So, we found out that the views can be conditionally divided into two types:

- "ordinary" views, the result of calculation of which is not saved anywhere;
- materialized views, the result of calculation of which is saved and can be reused.

This difference is shown schematically in figure 5.1.a.

[215] **View** — a derived relvar that's virtual, not real (contrast snapshot). The value of a given view at a given time is the result of evaluating a certain relational expression — the view defining expression, specified when the view itself is defined — at the time in question. The view defining expression must mention at least one relvar, for otherwise the view wouldn't be, specifically, a variable as such. ("The New Relational Database Dictionary", C.J. Date)

[216] **Materialized view, snapshot** — a derived relvar that's real, not virtual (contrast view). The value of a given snapshot at a given time is the result of evaluating a certain relational expression — the snapshot defining expression, specified when the snapshot itself is defined — at some time prior to the time in question: to be precise, at the most recent "refresh time". The snapshot is "refreshed" (i.e., the snapshot defining expression is reevaluated and the result assigned as the new current value of the snapshot) on explicit user request or, more usually, when some prescribed event occurs, such as the passing of a certain interval of time. Note: The snapshot defining expression must mention at least one relvar, for otherwise the snapshot wouldn't be a variable as such. ("The New Relational Database Dictionary", C.J. Date)

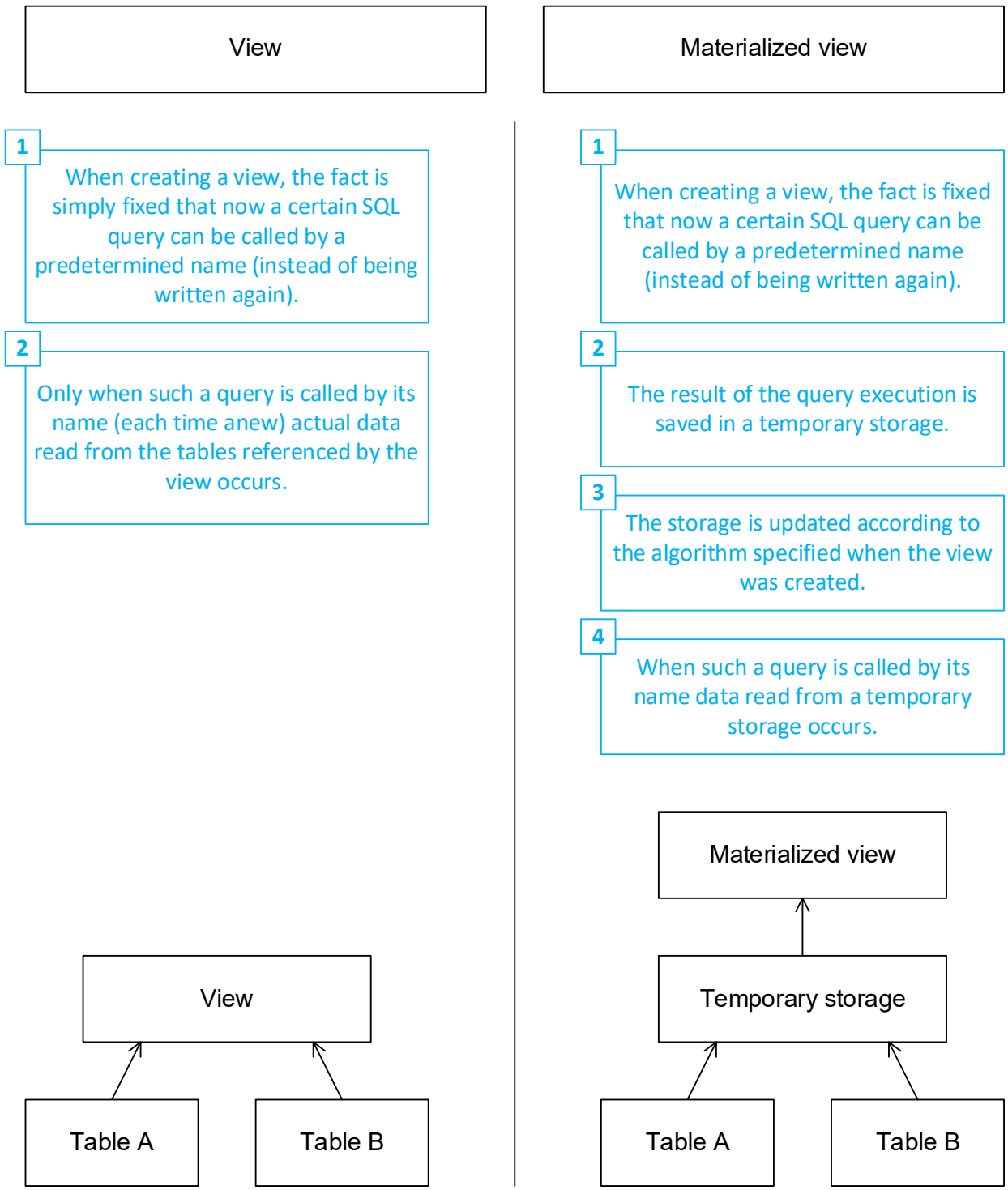


Figure 5.1.a — Comparison of views and materialized views

Since the views are built based on SQL queries, it should be made clear that:

- only a **`SELECT`**-queries can be the “basis” for building a view (i.e., you cannot build a view based on **`INSERT`**, **`UPDATE`**, **`DELETE`** queries);
- in some cases, the view may allow “bidirectionality”, i.e., it will be possible to use it not only to read data, but also to modify data.



So-called “bidirectional” (or “updatable”) views are still controversial. They are implemented in most DBMSes, but their capabilities are very limited, critically depend on the SQL query defining the view, and in general the use of such views may lead to an undefined DBMS behavior.

It is much more important to remember that if a security model is implemented using views, extra effort must be made to ensure that such views are not “bidirectional”, i.e., to block the possibility of modifying data while using them.

It would seem that if a view is (simplified) just a named SQL query, what’s the point of it at all? In fact, there are many advantages:

- Simplification of query execution. A view can be built on an SQL query of any complexity. And now, instead of working with a very complex query that can take tens or hundreds of lines, we just need to write something like `SELECT * FROM view`, to get the same result.
- Ability to build a simple and reliable API[217]. At the database design stage, we can provide a set of easy-to-use views that will “hide” complex queries and other non-trivial logic from the developers of applications that use the database, which greatly simplifies and accelerates the development of such applications and reduces the number of errors made during their development.
- Simplification of business logic. This point largely follows from the previous one, but even if we don’t create a full-fledged API[217], we can form a set of views for the most complex and popular cases (e.g., in some online store we need to show popular products, and the algorithm for defining popular products is very complex and non-trivial; it would make sense to create a `popular_items` view that would display a ready list of such products).
- Minimal overhead. A view (“normal”, not materialized) takes almost no space in the database, so even creation of hundreds and thousands of views does not significantly increase the size of the database.
- Performance. If we use materialized views, we get the possibility to repeatedly use the results of the SQL query execution, and the gain will be the more tangible, the longer the query execution takes and the less frequently we have to refresh the results of its execution.
- Security. Views are database objects; therefore, all DBMS mechanisms of access permissions control apply to them. Since views provide indirect access to the database tables, we can prohibit such access directly to the tables and allow it only through specially created and elaborated views. An additional level of security can be provided by the logic of the queries on which the views are based.

But if everything is so great, then why not create a huge number of views in every database? There are several reasons for this.

- Redundancy. Sometimes views are just not needed. E.g., the database is small, the business logic is simple, the security model is trivial — here, views will just be an extra layer of abstraction between the application and the database.
- Additional code. Views don’t appear and don’t exist by themselves. They need to be thought through, created, tested, and adjusted when the database or business logic changes. This requires extra work and increases the likelihood of errors.
- Limited capabilities. Although the use of views for data modification is implemented in many DBMS, it has a wide range of limitations, so we can argue that not every database operation can be performed using views.

[217] **API** (application programming interface) — a computing interface which defines interactions between multiple software intermediaries. (“Wikipedia”) [https://en.wikipedia.org/wiki/Application_programming_interface]

The bottom line is obvious: like any other tool, views can be very useful if used correctly, but thoughtlessly creating and using them is just as likely to cause more problems.



Task 5.1.a: make a list of views to add to the “Bank{395}” database. For each view, name which tasks can be solved with it.

5.1.2. Creating and Using Views

The vast majority of database design tools allow you to create views as database schema elements (which makes the design process more convenient and allows you to automate code generation). But since a view is actually a “named SQL query”, the main work (i.e., writing the query) still has to be done manually.

As a first example, consider the following situation: suppose for some reason our filesharing service needs to quickly and frequently get a list of files uploaded on the first weekend of each month (the first Saturday and Sunday of each month).

This is what an SQL query to solve this problem would look like.

MySQL — Getting a list of files uploaded on the first weekend of each month

```
SELECT *
FROM
        (
                SELECT *,
                CASE
                 WHEN WEEKDAY(FROM_UNIXTIME(`f_upload_dt`) -
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY) <=
                      WEEKDAY(FROM_UNIXTIME(`f_upload_dt`))
                 THEN WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) -
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5) + 1
                 ELSE WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) -
                INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5)
                END AS `W`,
                WEEKDAY(FROM_UNIXTIME(`f_upload_dt`)) + 1 AS `D`
        FROM `file`
        ) AS `prepared_data`
WHERE (`W` = 1)
  AND ((`D` = 6) OR (`D` = 7))
```

This query looks quite cumbersome, it is difficult to write and understand, and its improvement (provided that it is used in several places) is highly likely to lead to errors.

To avoid all these inconveniences, you can “wrap” this query into a view, which looks like this:

MySQL — Creating a view to get a list of files uploaded on the first weekend of each month

```
CREATE VIEW `files_on_first_days_off` AS
SELECT *
FROM    ( SELECT *,
                CASE
                 WHEN WEEKDAY(FROM_UNIXTIME(`f_upload_dt`) -
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY) <=
                      WEEKDAY(FROM_UNIXTIME(`f_upload_dt`))
                 THEN WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) -
                 INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5) + 1
                 ELSE WEEK(FROM_UNIXTIME(`f_upload_dt`), 5) -
                      WEEK(FROM_UNIXTIME(`f_upload_dt`) -
                INTERVAL DAY(FROM_UNIXTIME(`f_upload_dt`))-1 DAY, 5)
                END AS `W`,
                WEEKDAY(FROM_UNIXTIME(`f_upload_dt`)) + 1 AS `D`
        FROM `file`
        ) AS `prepared_data`
WHERE (`W` = 1)
  AND ((`D` = 6) OR (`D` = 7))
```

And now to get the same data it is enough to perform this primitive query:

MySQL — Getting a list of files uploaded on the first weekend of each month using the view

```
SELECT * FROM `files_on_first_days_off`
```

Assuming that such a list of files is needed for reporting and may contain slightly outdated information (say, with a day lag), it would be logical to create a materialized view with automatic updates once a day.

Unfortunately, MySQL (for now?) does not support materialized views, so in this database this problem can only be solved by using a so-called “caching table” and a stored procedure{353} that updates this table (see task 5.1.c{337}).

Support of materialized views is implemented in different DBMSes in very different ways. For example:

- in MS SQL Server it is impossible to control their updating logic (it happens automatically when data changes);
- in Oracle we can manage their updating logic in a very flexible way (this DBMS supports quite a complex syntax for this task);
- PostgreSQL has a separate command to update them.

In any case, it is necessary to be guided by the specific version of a particular DBMS in choosing the optimal solution.

As a second example, consider using a view to manage security.

Suppose that we wanted to create a separate role and restrict the file management capabilities of the users in that role — they are only allowed to work with files with the “jpg”, “jpeg”, “png”, “gif” extensions.

Then for such a role, we must deny direct access to the **`file`** table, and instead allow access to the special view:

MySQL — A view to provide restricted access to the `file` table

```
1    CREATE VIEW `files_with_jpg_jpeg_png_gif_extensions` AS
2    SELECT *
3      FROM `file`
4     WHERE LOWER(`f_original_extension`) IN ('jpg', 'jpeg', 'png', 'gif')
5    WITH CHECK OPTION
```

This view is “bidirectional” (updatable), i.e., we can use it not only to retrieve data, but also to add, update, and delete.

The **`WITH CHECK OPTION`** clause in the last query line instructs the DBMS to prohibit insertion and update operations that do not satisfy the **`WHERE`** clause, i.e., an attempt to add a file with a prohibited extension or change an existing file extension to a prohibited one will result in an error.

If we want to disable data update operations, the view can be rewritten as follows.

MySQL — A view to provide restricted access to the `file` table

```
1    CREATE VIEW `ro_files_with_jpg_jpeg_png_gif_extensions` AS
2    SELECT `f_id`,
3           `f_owner`,
4           `f_size` + 0,
5           `f_upload_dt`,
6           `f_exp_dt`,
7           `f_original_name`,
8           `f_original_extension`,
9           `f_name`,
10          `f_control_sum`,
11          `f_delete_link`
12     FROM `file`
13    WHERE LOWER(`f_original_extension`) IN ('jpg', 'jpeg', 'png', 'gif')
```

Note the 4th line of the query (the **`` `f_size` + 0 ``** clause): obviously, when fetching data, adding zero to a number does not change the number, i.e., data will be fetched correctly, but having such a clause in the query used to build the view prevents MySQL from using the view to modify data, which means that our goal is achieved.

Also note that in this case there is no need to add the `WITH CHECK OPTION` clause, since this view in principle does not allow any data modification operations.

Following the same logic, we can form any set of views that restrict access to individual columns of the table, to records with certain values or combinations of values, etc.

The general algorithm is very simple: we need to write a query to select data, then add the `CREATE VIEW `name_of_the_view` AS` structure before it — and that's it, we get the desired result.



A lot of additional practical examples are given in Section 3 "Using Views" of the "Using MySQL, MS SQL Server and Oracle by examples[218]" book.



Task 5.1.b: create the views that you listed in task 5.1.a{334}.



Task 5.1.c: earlier{336} in this section it was noted that MySQL does not support materialized views, so in this DBMS the task of updating the list of files uploaded on the first weekend of each month can only be solved by using a so-called "caching table" and a stored procedure that updates this table. Create an appropriate table and a stored procedure{353} to update it.

218 "Using MySQL, MS SQL Server and Oracle by examples" (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

5.2. Checks

5.2.1. General Information on Checks

We are already familiar with the simplest type of check — this is a restriction of `NOT NULL`, which is specified in the definition of the table field and prohibits the insertion of `NULL` values in the field.

However, the capabilities of the checks are much wider, and their use allows very flexible management of the list of allowable values of the table fields.



Check[219] — a rule that restricts all values of a certain table field to certain conditions.

Simplified: a condition that must be met by the value of a table field.

In general, checks can apply to the whole table (operate with the values of several fields) or to its individual fields (operate with the value of one field).

Using checks is a simpler and easier way to control database consistency{67} than using triggers{341} because checks (as a rule) are not only simpler to create, but also faster than triggers.

In any subject area, you can easily identify dozens of conditions, the control of which is convenient to assign to checks, for example:

- login and password must not be the same;
- date of employment must be greater than date of birth by at least 18 years;
- the discount amount must not exceed N% of the order amount;
- passport number must have the specified format;
- the maximum storage time of a file on the server must not exceed N days;
- etc.

The technical capabilities of the checks differ depending on the DBMS (from trivial “more / less / (not) equal” conditions to the use of regular expressions and stored functions{353}.

It is also recommended to keep in mind the performance: computationally complex checks are in fact no better than triggers in terms of their effect on the speed of insert and update operations.

And another (albeit small) disadvantage of checks is the fact that in the case of a violation of the check condition, we are very limited in our ability to generate an informative error message. While using triggers we can both form such messages, and (in some DBMSes) create exceptions and otherwise control the whole query execution process (up to correcting erroneous data automatically).



Task 5.2.a: make a list of checks that should be added to the “Bank{395}” database schema. For each check, specify which problems from the subject area are solved with it.

[219] **Check** — a rule that specifies the values allowed in one or more columns of every row of a table. (“The New Relational Database Dictionary”, C.J. Date)

5.2.2. Creating and Using Checks

Use of checks is automatic — the DBMS automatically controls the conditions described by the check and prohibits the data insert or update operation if they are violated.

As for the creation of checks, let's consider it using two examples. Suppose that the customer of our file-sharing service has specified the following two business rules:

- user login and password must not be the same;
- the maximum storage time of a file on the server must not exceed 500 days.

Most database design tools allow us to add checks to a database schema in the same way as we add indexes (including uniqueness constraints obtained using unique indexes[{106}]). But even if the design tool we choose does not have this capability, any DBMS allows us to add checks to a table by modifying that table (using the **`ALTER TABLE`** clause).

So, let's move on to our examples.



MySQL supports checks starting from version 8.0.16. If you are using an older version, all checks will be successfully created, but will not work (i.e., they will not be applied to data in the table — as if the checks do not exist at all).

In the first case, we need to compare the values of the **`u_login`** and **`u_password`** fields of the **`user`** table and make sure that they are different. The situation is complicated by the fact that the password already comes in the form of a SHA-256 hash, so we need to calculate this hash from the user name and compare the obtained value with the password value.

Technically it looks like this:

MySQL — Check that prohibits the values of the `u_login` and `u_password` fields to be the same

```
1    ALTER TABLE `user`
2     ADD CONSTRAINT `CHK_login_password_mb_different`
3      CHECK (SHA2(`u_login`, 256) != `u_password`)
```

Now if we try to add a user with the same login and password to the database, we will get an error message:

MySQL — Error message

```
1    Error Code: 3819.
2    Check constraint 'CHK_login_password_mb_different' is violated
```

In the second case, we need to make sure that the value of **`f_exp_dt`** field of the **`file`** table is within 500 days of the value of **`f_upload_dt`** field. Here we cannot use the value of current date because MySQL forbids using so called "nondeterministic" functions in checks (whose calling with the same parameters can give different results), and **`CURRENT_DATE()`** function which generates the current date, is objectively nondeterministic.

So, technically, the solution looks like this:

MySQL — Check that prohibits storing files for more than 500 days

```
1    ALTER TABLE `file`
2     ADD CONSTRAINT `CHK_max_500_days_store`
3      CHECK (DATEDIFF(FROM_UNIXTIME(`f_exp_dt`),
4                      FROM_UNIXTIME(`f_upload_dt`)) <= 500)
```

Now if we try to add a file to the database with a storage time of more than 500 days, we will get an error message:

MySQL — Error message

```
1    Error Code: 3819.
2    Check constraint 'CHK_max_500_days_store' is violated
```

Since we cannot generate our own error message when a check condition is violated, it is especially important to give the checks adequate meaningful names. This will allow you to diagnose the problem much faster when developing and debugging applications that work with your database.



A lot of more complicated practical examples are given in Section 4 “Using Triggers” of the “Using MySQL, MS SQL Server and Oracle by examples” book[220]. Yes, that section is about triggers, not checks, because triggers solve much more complicated problems.



Task 5.2.b: create the checks that you listed in Task 5.2.a{338}.

[220] “Using MySQL, MS SQL Server and Oracle by Examples” (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

5.3. Triggers

5.3.1. General Information on Triggers

Let's start with the definition.



Trigger[221] — a special database object that describes a list of actions that must be automatically performed when the specified event occurs.

Simplified: a description of an action to be performed automatically under certain conditions.

Triggers are implemented very differently in different DBMSes, and it is not only a question of syntax, but also what events can be associated with triggers, and how exactly they are executed (how the data associated with the triggered event is handled).

Let's start with events. The most classic use of triggers is to provide a reaction to data modification, i.e., to insert, update and delete operations.

Obviously, if some event occurs, the DBMS may execute a trigger before, after or instead of the corresponding operation. For clarity, let's summarize all types of triggers for several DBMSes in one table.

		DBMS		
Action	**Before, after, instead of**	**MySQL**	**MS SQL Server**	**Oracle**
Inserting data	Before	+	-	+
	After	+	+	+
	Instead of	-	+	+
Updating data	Before	+	-	+
	After	+	+	+
	Instead of	-	+	+
Deleting data	Before	+	-	+
	After	+	+	+
	Instead of	-	+	+
Creating a table	Before	-	-	See DBMS documentation
	After	-	+	
	Instead of	-	-	
Deleting a table	Before	-	-	See DBMS documentation
	After	-	+	
	Instead of	-	-	
User authentication	Before	-	-	See DBMS documentation
	After	-	+	
	Instead of	-	-	
Other operations with database structures	Before	-	-	See DBMS documentation
	After	-	-	
	Instead of	-	-	
Other situations	Before	-	-	See DBMS documentation
	After	-	-	
	Instead of	-	-	

[221] **Trigger,** triggered procedure — an action (the "triggered action") to be performed if a specified event (the "triggering event") occurs. A triggered procedure can be thought of as a stored procedure, except that stored procedures must be explicitly invoked, whereas a triggered procedure is invoked automatically whenever the triggering event occurs. ("The New Relational Database Dictionary", C.J. Date)

Most of all we will be interested in triggers associated with data modification operations, so let's present the logic of their work graphically (see figure 5.3.a).

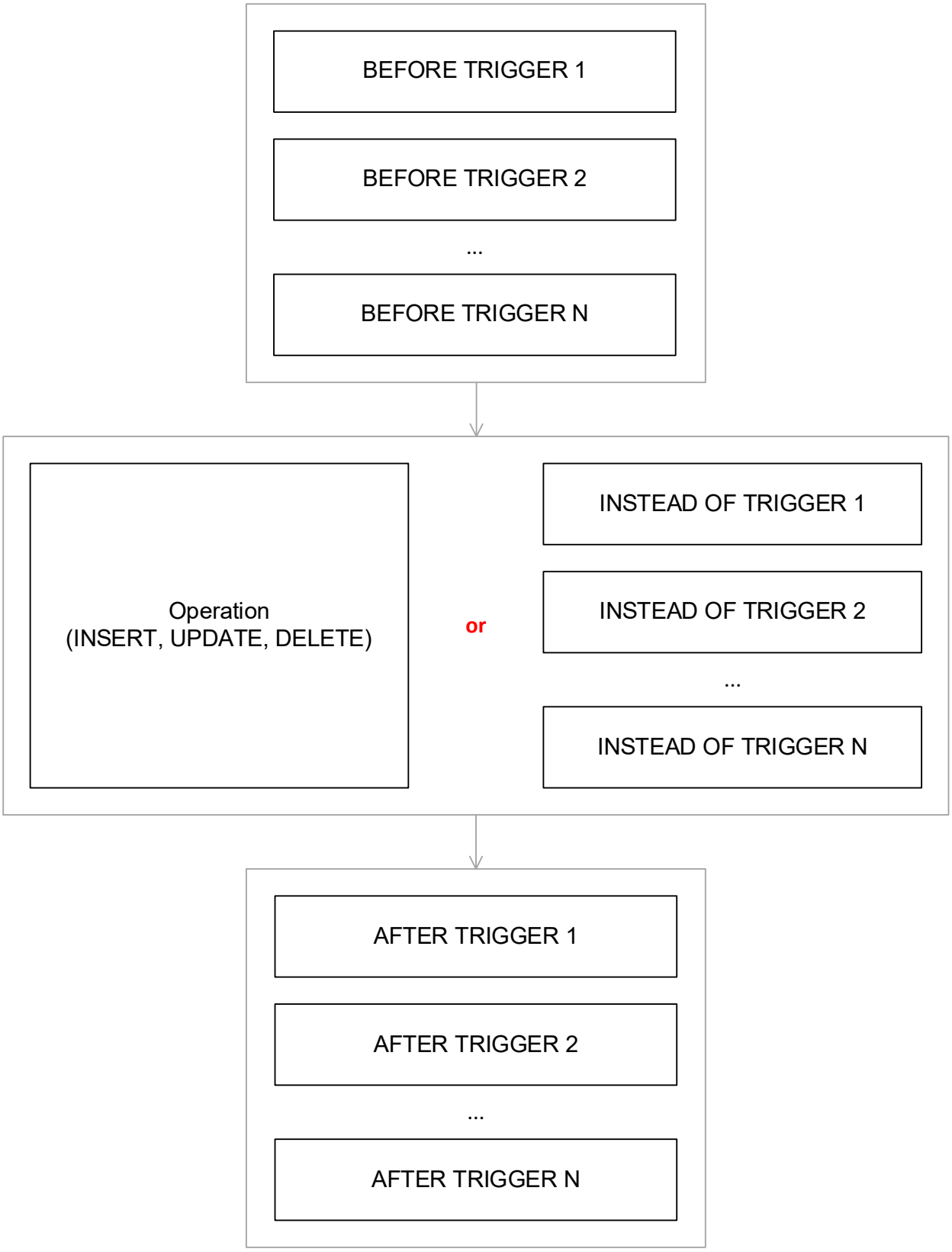


Figure 5.3.a — Triggers work logic

We should also emphasize that triggers executed instead of an operation (so-called **INSTEAD OF** triggers) are in fact executed *instead* of the corresponding operation, and therefore the operation itself should be implemented inside the trigger and executed when all checks and conditions are passed. This is especially true for MS SQL Server, which does not have **BEFORE** triggers and has to implement their logic in **INSTEAD OF** triggers.

Now let's look at exactly how triggers are executed and how data is processed. In this context, triggers are divided into two groups:

- row-level triggers run each time anew for each separate row (table record) affected by the SQL query;
- statement-level triggers run once for the whole SQL-query.

Let's explain this graphically (see figure 5.3.b).

Imagine that in some table we need to increase product price by 20 % for all goods cheaper than 50 (currency is not important, so we just leave the value). And let's assume that there are three records in the table that satisfy the query condition.

The query itself may look like this:

MySQL — 20 % increase in the price of goods, the price of which is now less than 50

```
1    UPDATE `item`
2       SET `i_price` = `i_price` + (`i_price` * 0.2)
3     WHERE `i_price` < 50
```

The row-level trigger will be activated three times (separately for each row). Using the special **OLD** and **NEW** keywords in most databases the trigger will have access to the old and new field values of the record for which it is called.

The statement-level trigger will be activated once, and it will have access to information about all records affected by the update. Since there are many records, here we can no longer use the approach with the **OLD** and **NEW** keywords (they allow working with only one record), but (by the example of MS SQL Server logic) we can use special "virtual" tables: **DELETED** (contains data before update) and **INSERTED** (contains data after update).



The order of records in **DELETED** and **INSERTED** tables may not match (as specifically shown in figure 5.3.b), so if the primary key value changes, we must rely on the values of other unique fields (if any) to match records in their old and new states. If there are no other unique fields in the table, but we need to match records in their old and new states, for DBMSes that do not support row-level triggers, this problem has no solution.

Additional features of the trigger implementation include the following:

- some DBMSes allow us to create no more than one "certain type" trigger per table (e.g., no more than one **BEFORE INSERT** trigger), while other DBMSes allow us to create many "same type" triggers and even control their execution sequence;
- in different DBMSes the "namespace" of triggers may be limited to a table (i.e., for two different tables we can create triggers with the same name) or the entire database (then the names of all triggers in the entire database must be different);
- capabilities and logic of "exotic" triggers (not related to data modification operations) are absolutely unique for every DBMS and require careful study of documentation.

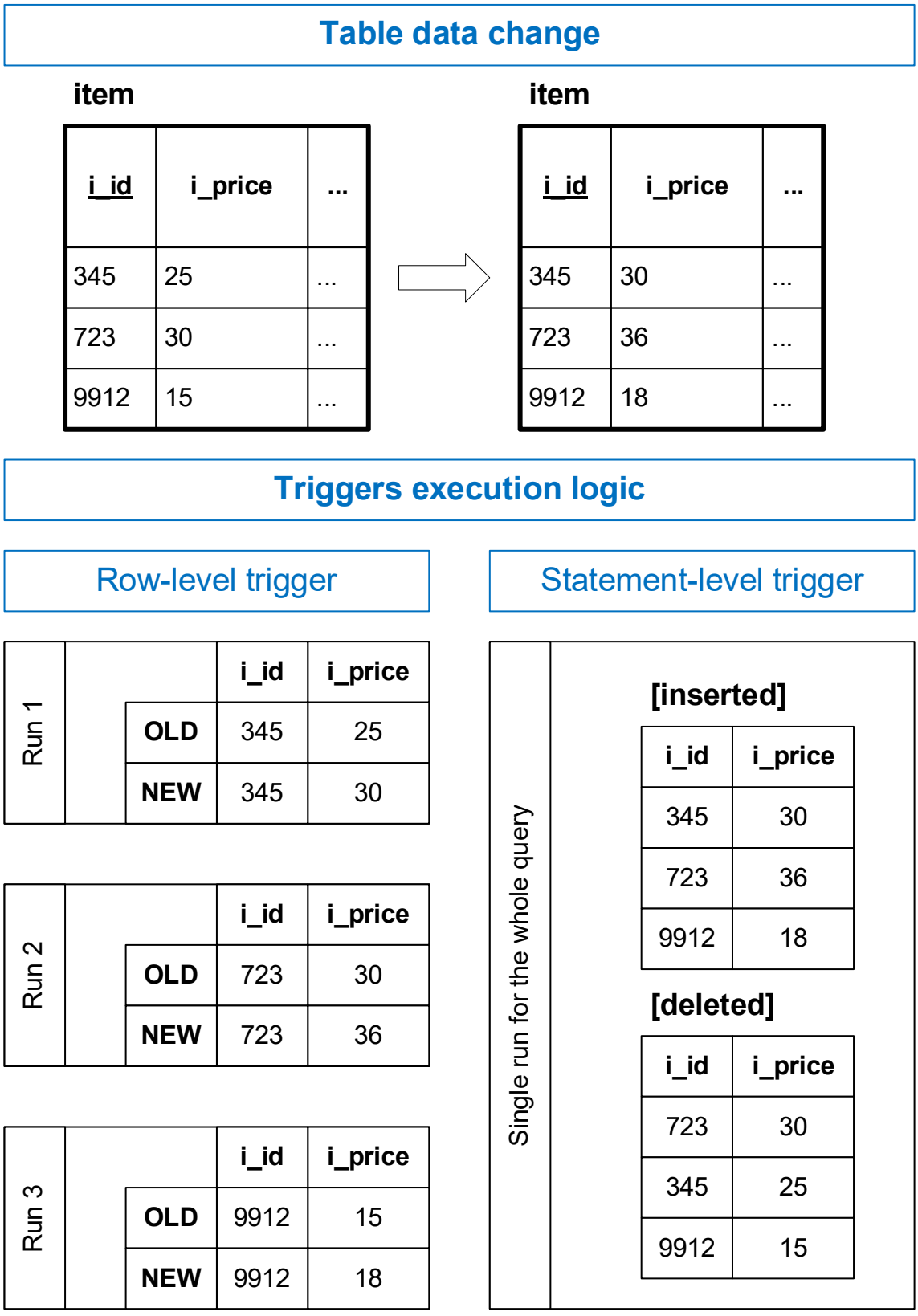


Figure 5.3.b — The difference between the operation of row-level and statement-level triggers

With the help of triggers in general the following range of tasks is solved:

- organizing cascade operations{68} (if the used storage engine{30} does not support them) or implementing a more complex logic of cascade operations than the DBMS provides;
- updating the data of caching (or aggregating) fields and tables (the example will be discussed later; it is one of the most frequent cases of using triggers);
- ensuring consistency{67}, i.e., controlling and changing such values of the fields that are in strict dependence on the values of other fields (or other conditions);
- controlling data modification operations to enforce business logic rules (e.g., prohibiting deletion of the last user with the “administrator” role or prohibiting deletion of such a role from such a user if business logic dictates that “there must always be at least one administrator in the system”);
- controlling relationship cardinality{53} when “standard” options (“one to one”, “one to many”, “many to many”) are not enough, yet it is necessary to provide more complex behavior;
- controlling the format and values of data if the DBMS does not support checks{338}, or if informative error messages must be generated, or if the check logic goes beyond what is provided by the standard DBMS capabilities;
- transparent correcting errors in data (in those rare cases when it is possible; e.g., bringing the full name into the format “Surname Name Patronymic”, i.e., all words are written in lower case with a capital letter, even if the data originally came in the form of, e.g., “john ADamS DoE”.

The disadvantages of triggers are their impact on performance. And it is the stronger the more complex operations are performed inside triggers (and such operations may include access to other tables, complex calculations, etc.)

And yet triggers are very common. They are used to solve a wide range of application tasks (especially in terms of data safety{15}).



Task 5.3.a: make a list of triggers that should be added to the “Bank{395}” database. For each trigger, specify which tasks are solved with it.

5.3.2. Creating and Using Triggers

Just like with checks{338}, triggers are used automatically — the DBMS starts them automatically at the appropriate moments (when the specified events occur), we just need to write the code that implements the required actions.

Most database design tools allow us to add the appropriate object to the schema, but since the triggers are defined by its SQL code, this code still has to be written manually.

As an example, consider solving two problems (both of which were mentioned in the previous chapter):

- updating the aggregating table data;
- ensuring that the business rule “there must always be at least one administrator in the system” is enforced.

The solution to the first problem is limited to updating the data in the `statistics` aggregating table (see figure 5.3.c).

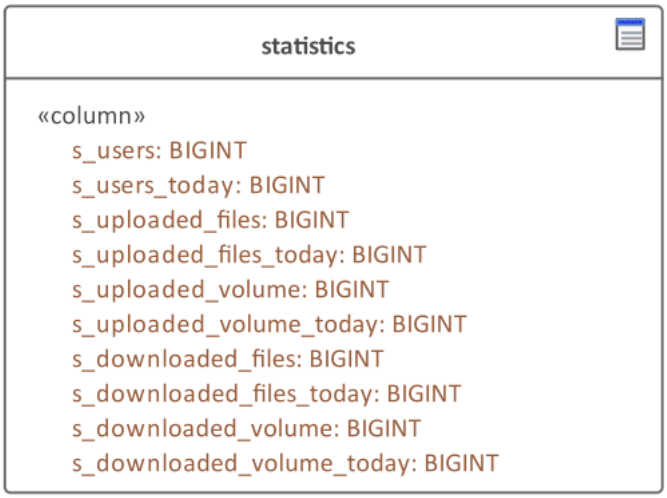


Figure 5.3.c — The `statistics` table

This table must always contain exactly one row with the corresponding data in each field.

In terms of reliability, it would be worth checking each time if there is exactly one row in this table (and if there are more rows, then delete them all and create one new one, or if there are no rows, then just create one new one).

But this will have a catastrophic effect on performance, so the operation to create a single row would have to rely on the database generation script, and then we have to assume that no one and nothing deletes the row or adds new ones.

The performance will also be greatly affected by the data generation strategy because we can:

- recalculate all the required values each time (this is very simple, but very slow, because we have to run several `COUNT()` queries on tables with lots of records);
- change the existing values by the corresponding amount (this is much faster but will require fine-tuning the table itself).

Let's take the second way and refine the table (see figure 5.3.d). The necessity of these changes is caused by the fact that part of the data should be counted "for today", and therefore it is necessary to understand when "today" turns into "yesterday".

Let's add `s_actual_date` field to the table: there we will store information about the estimated current date. Then each time the trigger will determine the real current date. And if its value is different from `s_actual_date` field value, it means that "it is a new day".

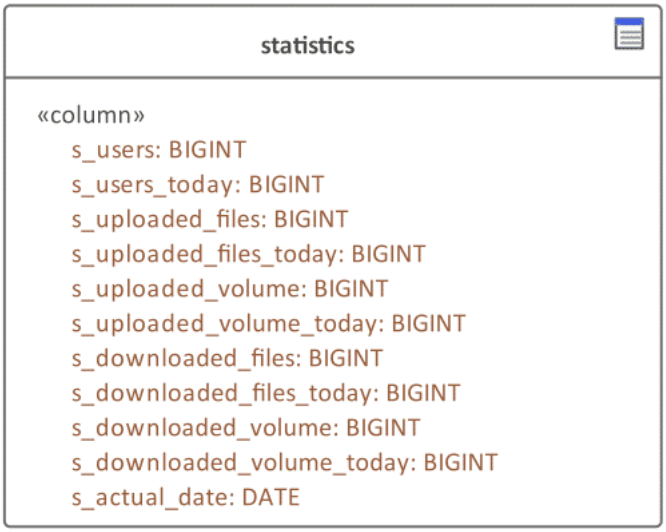


Figure 5.3.d — The `statistics` table after refinement

It also turns out (and very well illustrates the fact that errors in database design can be detected at any design level and at any moment) that there is a flaw in the `file` table — there is no field to store the number of downloads of a file. Let's refine this table as well (see figure 5.3.e) by adding `f_download_count` and `f_download_count_today` fields.

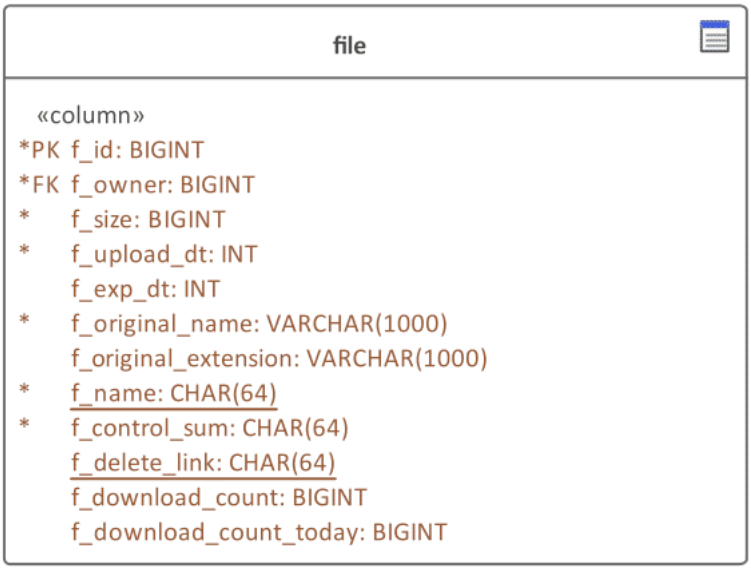


Figure 5.3.e — The `file` table after refinement

Now we need to create three triggers on the `file` table (yes, similar triggers need to be created on the `user` table as well, but this is a self-study task, see task 5.3.c{352}):

- `AFTER INSERT` — a new file has been added, we need to add information about it to the `statistics` table;
- `AFTER DELETE` — a file has been deleted, we need to remove information about it from the `statistics` table;
- `AFTER UPDATE` — information about a file has changed (perhaps the download count has increased, and we need to add this information to the `statistics` table).

All three triggers are `AFTER` ones, i.e., they will be called by the DBMS only after the data modification operation in `file` table is successfully completed. This prevents the DBMS from having to cancel all alterations performed by the triggers in case of an unsuccessful `file` table data modification operation.

Also, all three triggers will have to control the fact that the day has changed (when "today" turned into "yesterday") and perform some additional operations.

We design our database for MySQL, and this DBMS does not allow the same trigger to be associated with multiple events (i.e., to make the same trigger run when data is inserted, updated, or deleted), so we again have two options:

- duplicate part of the code in all triggers;
- transfer a part of the code to a stored procedure{353} and call it from triggers.

The second option is much more advantageous, because it allows us to use a stored procedure outside the triggers.

The next chapter{353} is devoted to stored procedures, but since we need this procedure now, we will start with its code.

MySQL — Stored procedure that clears the statistics when a new day begins

```
1   DROP PROCEDURE IF EXISTS NEW_DAY;
2   DELIMITER $$
3   CREATE PROCEDURE NEW_DAY ()
4   BEGIN
5    IF EXISTS (SELECT 1 FROM `statistics`
6                       WHERE `s_actual_date` != CURRENT_DATE())
7     THEN
8      UPDATE `statistics` SET
9       `s_users_today` = 0,
10      `s_uploaded_files_today` = 0,
11      `s_uploaded_volume_today` = 0,
12      `s_downloaded_files_today` = 0,
13      `s_downloaded_volume_today` = 0,
14      `s_actual_date` = CURRENT_DATE();
15     UPDATE `file` SET
16      `f_download_count_today` = 0;
17    END IF;
18  END;
19  $$ DELIMITER ;
```

The query in lines 5-6 checks if there is at least one line in the statistics table whose field `s_actual_date` does not contain today's date. If the condition is fulfilled, it means that it is a new day, and all "today's statistics" must be reset, which is done by queries in lines 8–14 and 15–16. Line 14 also changes the value of today's date stored in the `statistics` table.

We will call this procedure in each of the following three triggers.
Let's start with the trigger that is activated after inserting the data.

MySQL — Trigger activated by adding a file

```
1    DROP TRIGGER IF EXISTS `TRG_update_file_stats_after_ins`;
2    DELIMITER $$
3
4    CREATE TRIGGER `TRG_update_file_stats_after_ins` AFTER INSERT ON `file`
5     FOR EACH ROW BEGIN
6      CALL NEW_DAY();
7      UPDATE `statistics` SET
8        `s_uploaded_files` = `s_uploaded_files` + 1,
9        `s_uploaded_files_today` = `s_uploaded_files_today` + 1,
10       `s_uploaded_volume` = `s_uploaded_volume` + NEW.`f_size`,
11       `s_uploaded_volume_today` = `s_uploaded_volume_today` + NEW.`f_size`;
12    END;
13   $$
14   DELIMITER ;
```

After calling the stored procedure (line 6), which resets yesterday's statistics in case of a change of day, we increase the counters of uploaded files (lines 8–9) and increase the value of the volume of uploaded files (lines 10–11). And that's it.

The next trigger reacts to a data update. In the general case, we expect that the file's download count has changed. But (at least in theory) the file can also change size (and even the number of downloads arbitrarily). Therefore, the trigger code will be a bit more complicated.

MySQL — Trigger activated by updating a file

```
1    DROP TRIGGER IF EXISTS `TRG_update_file_stats_after_upd`;
2    DELIMITER $$
3
4    CREATE TRIGGER `TRG_update_file_stats_after_upd` AFTER UPDATE ON `file`
5     FOR EACH ROW BEGIN
6      IF (FROM_UNIXTIME(OLD.`f_upload_dt`) = CURRENT_DATE())
7       THEN
8        UPDATE `statistics` SET
9           `s_uploaded_volume_today` = `s_uploaded_volume_today` -
10                                      OLD.`f_size`;
11       UPDATE `statistics` SET
12          `s_uploaded_volume_today` = `s_uploaded_volume_today` +
13                                      NEW.`f_size`;
14       UPDATE `statistics` SET
15          `s_downloaded_files_today` = `s_downloaded_files_today` -
16                                       OLD.`f_download_count_today`;
17       UPDATE `statistics` SET
18          `s_downloaded_files_today` = `s_downloaded_files_today` +
19                                       NEW.`f_download_count_today`;
20       UPDATE `statistics` SET
21          `s_downloaded_volume_today` = `s_downloaded_volume_today` -
22                        OLD.`f_size` * OLD.`f_download_count_today`;
23       UPDATE `statistics` SET
24          `s_downloaded_volume_today` = `s_downloaded_volume_today` +
25                        NEW.`f_size` * NEW.`f_download_count_today`;
26      END IF;
27
28       CALL NEW_DAY();
29       UPDATE `statistics` SET
30          `s_uploaded_volume` = `s_uploaded_volume` - OLD.`f_size`;
31       UPDATE `statistics` SET
32          `s_uploaded_volume` = `s_uploaded_volume` + NEW.`f_size`;
33       UPDATE `statistics` SET
34          `s_downloaded_files` = `s_downloaded_files` -
35                                 OLD.`f_download_count`;
```

```
36        UPDATE `statistics` SET
37            `s_downloaded_files` = `s_downloaded_files` +
38                                   NEW.`f_download_count`;
39        UPDATE `statistics` SET
40            `s_downloaded_volume` = `s_downloaded_volume` -
41                    OLD.`f_size` * OLD.`f_download_count`;
42        UPDATE `statistics` SET
43            `s_downloaded_volume` = `s_downloaded_volume` +
44                    NEW.`f_size` * NEW.`f_download_count`;
45
46     END;
47    $$
48    DELIMITER ;
```

In this case, we need to check if the file was uploaded today (line 6) and, if so, adjust today's statistics (lines 8–25). Then we can call the stored procedure (line 28), which resets yesterday's statistics in case of day change, and correct the overall statistics (lines 29–44).

And finally, there is the trigger that reacts to the deletion of data. It is almost identical to the previous one, except that here the statistics will only be corrected downward.

MySQL — Trigger activated by deleting a file

```
1     DROP TRIGGER IF EXISTS `TRG_update_file_stats_after_del`;
2     DELIMITER $$
3
4     CREATE TRIGGER `TRG_update_file_stats_after_del` AFTER DELETE ON `file`
5      FOR EACH ROW BEGIN
6       IF (FROM_UNIXTIME(OLD.`f_upload_dt`) = CURRENT_DATE())
7        THEN
8        UPDATE `statistics` SET
9            `s_uploaded_volume_today` = `s_uploaded_volume_today` -
10                                       OLD.`f_size`;
11       UPDATE `statistics` SET
12           `s_uploaded_files_today` = `s_uploaded_files_today` - 1;
13       UPDATE `statistics` SET
14           `s_downloaded_files_today` = `s_downloaded_files_today` -
15                                        OLD.`f_download_count_today`;
16       UPDATE `statistics` SET
17           `s_downloaded_volume_today` = `s_downloaded_volume_today` -
18                           OLD.`f_size` * OLD.`f_download_count_today`;
19      END IF;
20
21       CALL NEW_DAY();
22       UPDATE `statistics` SET
23           `s_uploaded_volume` = `s_uploaded_volume` - OLD.`f_size`;
24       UPDATE `statistics` SET
25           `s_uploaded_files` = `s_uploaded_files` - 1;
26       UPDATE `statistics` SET
27           `s_downloaded_files` = `s_downloaded_files` -
28                                  OLD.`f_download_count`;
29       UPDATE `statistics` SET
30           `s_downloaded_volume` = `s_downloaded_volume` -
31                   OLD.`f_size` * OLD.`f_download_count`;
32
33     END;
34    $$
35    DELIMITER ;
```

Once again, in order for the **`statistics`** table to be updated completely, similar triggers must be created on the **`user`** table as well, but this is a task for self-study (see task 5.3.c{352}).

We move on to the second example — control of the business rule “there must always be at least one administrator in the system” with the help of triggers.

This rule can be violated in two cases:

- an attempt is made to delete the only user who is in the “administrators” group;
- an attempt is made to remove the only user from the “administrators” group.

In both situations we must perform the appropriate checks and prohibit the operation if it would result in a violation of the business rule.

Some difficulty is that there is no strict formal way to define the “administrators” group. We can rely only on its name or assume that its identifier is fixed, known to us and will not change in the future.

The second approach is more justified because in the `group` table the primary key is surrogate{38} and therefore will hardly change. And we can also, at the stage of database formation, add to the script the operation to create the appropriate user group with the specified identifier.

Thus, we will assume that the “administrators” group is described in the `group` table by a record with the value of the primary key equal to 1.

Let’s create a trigger that prohibits deleting the only user who is in the “administrators” group. This will be a `BEFORE DELETE` trigger on the `user` table.

MySQL — Trigger prohibiting the deletion of the last administrator

```
1    DROP TRIGGER IF EXISTS `TRG_preserve_last_admin_before_del`;
2    DELIMITER $$
3
4    CREATE TRIGGER `TRG_preserve_last_admin_before_del` BEFORE DELETE ON `user`
5     FOR EACH ROW BEGIN
6      IF (NOT EXISTS (SELECT 1
7                      FROM   `m2m_user_group`
8                      WHERE  (`ug_group` = 1)
9                      AND (`ug_user` != OLD.`u_id`)))
10      THEN
11       SIGNAL SQLSTATE '45001'
12       SET MESSAGE_TEXT = 'You can not delete the last user from
13                           Administrators (`g_id` = 1) group.',
14       MYSQL_ERRNO = 1001;
15      END IF;
16    END;
17   $$
18   DELIMITER ;
```

Lines 6–9 check if there is at least one user in the “administrators” group with an ID different from the user being deleted. If none is found, an exception is created in lines 11–14, which blocks the operation and cancels the current transaction.

Note that, unlike the checks{338}, we can generate an informative error message here. And when we try to remove the last user who was in the “administrators” group, it will be displayed:

MySQL — Error message

```
1    Error Code: 1001.
2    You can not delete the last user from Administrators (`g_id` = 1) group.
```

Now we need to create two triggers (`BEFORE DELETE` and `BEFORE UPDATE`) on the `m2m_user_group` table; that will prohibit removing the last user from the “administrators” group.

Their code would be completely identical, and if MySQL allowed us to associate a single trigger with several operations, we could do it all with a single trigger. But this is not yet possible in MySQL, so we have to create two separate triggers.

MySQL — Trigger that prohibits removing the last user from the “administrators” group

```
1    DROP TRIGGER IF EXISTS `TRG_preserve_last_user_in_admins_before_del`;
2    DELIMITER $$
3
4    CREATE TRIGGER `TRG_preserve_last_user_in_admins_before_del` BEFORE DELETE
5    ON `m2m_user_group`
6     FOR EACH ROW BEGIN
7      IF (NOT EXISTS (SELECT 1
8                      FROM   `m2m_user_group`
9                      WHERE  (`ug_group` = 1)
10                     AND (`ug_user` != OLD.`ug_user`)))
11      THEN
12       SIGNAL SQLSTATE '45001'
13       SET MESSAGE_TEXT = 'You can not remove the last user from
14                           Administrators (`g_id` = 1) group.',
15       MYSQL_ERRNO = 1001;
16      END IF;
17    END;
18   $$
19   DELIMITER ;
```

MySQL — Trigger that prohibits removing the last user from the “administrators” group

```
1    DROP TRIGGER IF EXISTS `TRG_preserve_last_user_in_admins_before_upd`;
2    DELIMITER $$
3
4    CREATE TRIGGER `TRG_preserve_last_user_in_admins_before_upd` BEFORE UPDATE
5    ON `m2m_user_group`
6     FOR EACH ROW BEGIN
7      IF (NOT EXISTS (SELECT 1
8                      FROM   `m2m_user_group`
9                      WHERE  (`ug_group` = 1)
10                     AND (`ug_user` != OLD.`ug_user`)))
11      THEN
12       SIGNAL SQLSTATE '45001'
13       SET MESSAGE_TEXT = 'You can not remove the last user from
14                           Administrators (`g_id` = 1) group.',
15       MYSQL_ERRNO = 1001;
16      END IF;
17    END;
18   $$
19   DELIMITER ;
```

Lines 7–10 of both queries check if there is at least one user in the “administrators” group with an ID that differs from the ID of the user being removed from the group. If none is found, an exception is created in lines 12–15 of both queries, which blocks the operation and cancels the current transaction.

And when an attempt is made to remove the last user from the “administrators” group, the error message will be displayed:

MySQL — Error message

```
1    Error Code: 1001.
2    You can not remove the last user from Administrators (`g_id` = 1) group.
```



A lot of more complicated practical examples are given in Section 4 “Using Triggers” of the “Using MySQL, MS SQL Server and Oracle by examples” book[222].



Task 5.3.b: create the triggers that you listed in task 5.3.a{345}.



Task 5.3.c: earlier{348} we discussed an example of how to create all the necessary triggers on the `file` table to update the statistics; and it was noted that similar triggers should be created on the `user` table as well. Create those triggers.

222 “Using MySQL, MS SQL Server and Oracle by Examples” (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

5.4. Stored Routines

5.4.1. General Information on Stored Routines

Like triggers{341}, stored routines are implemented very differently in different DBMSes, and again we are talking not only about syntax but also about fundamental differences and features (what operations stored routines can perform, how these routines can be used, etc.).

Let's start with definitions (this is especially important to understand the difference between stored procedures and stored functions (which are also sometimes called "user functions")).



Stored procedure[223] — a routine (possibly parameterized) designed to perform a number of data and database structure operations, stored on the database side and available both for calling from the code of other procedures and triggers, as well as for direct execution.

Simplified: a routine that is called directly or from other routines and performs some useful actions.



Stored function, user-defined function[224] — a routine (possibly parameterized) that extends the capabilities of the SQL language and works similarly to the built-in DBMS functions; it must return a value.

Simplified: a function that extends the capabilities of the DBMS and is designed to simplify frequently repeated operations.

So, the first and most important difference is that a function must return a value, a procedure has no such limitation (and often does not even have this capability). The second (typical for most DBMSes) difference is that a function can be used in any SQL query, while a procedure has its own specific syntax and some limitations.

Since this is not the end of the differences, let's give them in the form of a table (these statements are true for most DBMSes).

[223] **Stored procedure** — a subroutine, possibly parameterized; in other words, the implementation code for some operator. The term "stored procedure" has unfortunately come to mean something in practice that mixes model and implementation considerations. From the point of view of the model, a stored procedure is basically nothing more than an operator (or the implementation code for such an operator, rather). In practice, however, stored procedures have a number of properties that make them much more important than they would be if they were just operators as such (although the first two of the following properties will probably apply to operators in general, at least if the operators in question are system defined). First, they're compiled separately and can be shared by distinct applications. Second, their compiled code is, typically, physically stored at the site at which the data itself is physically stored, with obvious performance benefits. Third, they're often used to provide shared functionality that ought to be provided by the DBMS but isn't (integrity checking is a good example here, given the state of today's SQL implementations). ("The New Relational Database Dictionary", C.J. Date)

[224] **Stored function, user-defined function** — a way to extend SQL with a new function that works like a native (built-in) SQL function such as ABS() or CONCAT(). [https://dev.mysql.com/doc/refman/8.0/en/create-function-udf.html]

Stored procedure	Stored function
Can have several input and output parameters	Can have several input parameters, and is always obliged to return a value (in some DBMS this value can be a table)
Has its own syntax for calling and processing results	Can be used in any SQL query
Can call other stored procedures and functions	Can call only other stored functions, but not stored procedures
Can produce transactions	Cannot produce transactions
Has a full range of exceptions handling capabilities	Has a very narrow capability to handle exceptions (in some DBMSes, it cannot handle them at all)
Can perform any kind of data operations	Can perform read only operations (in most DBMSes with some exceptions like MySQL)
Can perform any kind of database object operations	Can only read information about database objects
Cannot be used in checks{338}	Can be used in checks{338}

If we reduce all the differences to one phrase, it turns out that we use functions when we need to get some value in an SQL query (by analogy with built-in functions like `SUM()`, `AVG()` etc.), and we use procedures when we need to perform some complicated actions (update data, create a new table, etc.).

Let's illustrate this graphically (see figure 5.4.a).

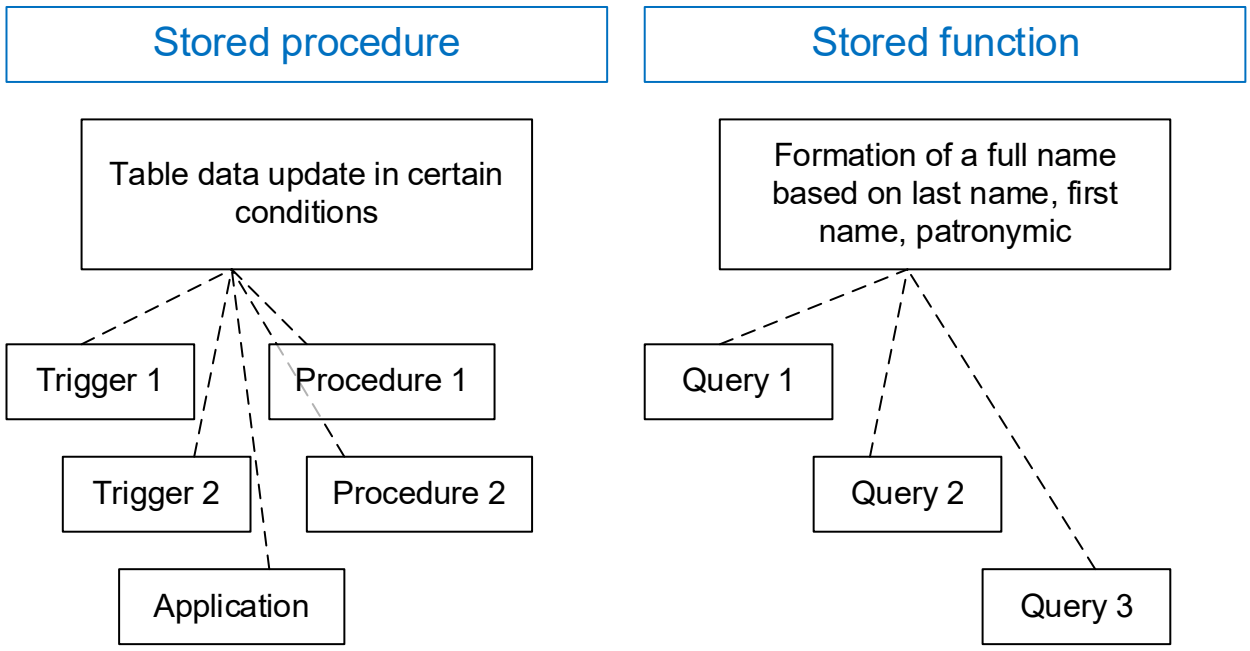


Figure 5.4.a — Typical usage of stored procedures and stored functions

In both cases, the following series of advantages are achieved:

- all necessary actions are described only once, i.e., it is not necessary to write the same code many times in different places;
- a routine call is a short construct, which makes code readability much better (compared to the situation where all relevant actions would be described explicitly);
- a routine runs on the DBMS side, i.e., no data is sent to the application or anywhere else, which improves performance and security.

The disadvantages of stored routines are not so obvious and depend very much on the particular DBMS, so the following statements are of a “possible” nature:

- in some cases, there may be performance degradation (compared to the direct execution of the code transferred to a stored routine);
- in some cases, there may be security problems (if wrong permissions are set for the stored routine);
- in some cases, the use of stored routines can complicate testing and debugging.

However, on the whole, the advantages of stored routines far outweigh their disadvantages.

Once again, let’s emphasize that syntax, implementation features, capabilities and limitations of stored routines are very different in different DBMSes (and even in different versions of the same DBMS), so be sure to study the documentation, experiment and thoroughly test the written code.



Task 5.4.a: make a list of stored routines that should be added to the “Bank{395}” database. For each routine, specify which tasks are solved by it.

5.4.2. Creating and Using Stored Routines

As an example, let's consider solving the following problems:

- correction of statistical data when the day changes;
- deleting the information with expired storage term;
- definition of user status ("novice", "experienced", "master") depending on the number and volume of downloaded and uploaded files;
- producing the information about a file size in a human-readable form with adjustment of the measurement unit (e.g., "10.35 MB", "1.24 GB", etc.).

The first two tasks will be implemented using stored procedures, the remaining ones using stored (user-defined) functions.

The vast majority of design tools allow us to add a separate object describing a stored procedure or function to the database schema. In the Sparx Enterprise Architect (tool that we use) these objects are called "Procedure" and "Function", respectively.

Earlier{348} we already looked at correcting the statistics when the day changes, but now we will look at this example in more detail. Let's start with the stored procedure code.

MySQL — Stored procedure that clears the statistics when a new day begins

```
1    DROP PROCEDURE IF EXISTS NEW_DAY;
2    DELIMITER $$
3    CREATE PROCEDURE NEW_DAY ()
4    BEGIN
5     IF EXISTS (SELECT 1 FROM `statistics`
6                         WHERE `s_actual_date` != CURRENT_DATE())
7      THEN
8       UPDATE `statistics` SET
9         `s_users_today` = 0,
10        `s_uploaded_files_today` = 0,
11        `s_uploaded_volume_today` = 0,
12        `s_downloaded_files_today` = 0,
13        `s_downloaded_volume_today` = 0,
14        `s_actual_date` = CURRENT_DATE();
15      UPDATE `file` SET
16        `f_download_count_today` = 0;
17    END IF;
18   END;
19   $$ DELIMITER ;
```

Recall that lines 5–6 of the query check if there is at least one row in the `statistics` table whose `s_actual_date` field does not contain current date. If the condition is true, then it is a new day, and all "today's statistics" must be reset, which is done by queries in lines 8–14 and 15–16. Line 14 also changes the value of current date stored in the `statistics` table.

We have used this stored procedure in several triggers, but there is a more elegant solution to eliminate the call of this procedure every time each trigger is activated, which would obviously improve the performance of these triggers.

Many DBMSes (MySQL among them) support the ability to execute specified actions on schedule. Such actions include the execution of a stored procedure. Unfortunately, many design tools do not allow us to add this information to the database schema, so we must manually place the appropriate SQL code in the database script.

Since by default the event scheduler (responsible for performing scheduled actions) in MySQL can be disabled, we need to add the following line to the `[mysqld]` option group in the `my.ini` configuration file:

```
event_scheduler = ON
```

Then restart the MySQL service.

Let's return to the essence of the problem. The change of the day objectively happens at midnight, so it is logical to perform all the corresponding actions at 0 hours 0 minutes 0 seconds of each day. To add the corresponding event to the schedule, just execute the following SQL query:

MySQL — Creation of an event to run the stored NEW_DAY procedure every midnight

```
1  CREATE EVENT `clear_statistics_at_midnight`
2   ON SCHEDULE
3    EVERY 1 DAY
4    STARTS '2000-01-01 00:00:00' ON COMPLETION PRESERVE ENABLE
5   DO
6    CALL NEW_DAY()
```

The **`STARTS`** clause in the 4th line of the query (it is better to put the date in the past) specifies the starting point (it is the time that matters, i.e., 00:00:00). Then this event will be repeated every day (3rd line of the query).

Now the corresponding stored procedure will automatically run every midnight and correct the statistical data. And this means that we can rewrite the previously shown{348} triggers to eliminate the need to call the stored procedure from their code (see task 5.4.c{363}).

Let's move on to the second task, i.e., deleting information with expired storage term.

Many tables in our database contain fields whose names end with **`exp_dt`**:

- user
  - u_speed_bonus_**exp_dt**
  - u_volume_bonus_**exp_dt**
  - u_ban_**exp_dt**
- ip_blacklist
  - ibl_**exp_dt**
- file
  - f_**exp_dt**
- download_link
  - dl_**exp_dt**

These fields contain data about the storage expiration datetime of the corresponding information. When this moment expires (on the specified date and time) it is necessary to delete or change the information in the database, namely:

- user table:
  - u_speed_bonus_exp_dt — remove information about the "speed bonus for uploaded files count";
  - u_volume_bonus_exp_dt — remove information about the "speed bonus for uploaded files volume";
  - u_ban_exp_dt — removed "banned" status;
- ip_blacklist table:
  - ibl_exp_dt — remove an ip address from the blacklist;
- file table:
  - f_exp_dt — delete file;
- download_link table:
  - dl_exp_dt — remove the download link.

And such changes to the database should happen completely automatically. It is easy to guess that we will use the same approach as in the previous example, i.e., we'll create a stored procedure and add its execution to the event scheduler. The only remaining question is how often this stored procedure will be called; the more often it will be called, the closer to real time the data will change, but the heavier will be the load on the database.

There is no “one-size-fits-all solution” here, we always have to start from the real situation, but let’s assume that we (and the customer) are satisfied with the option when such changes occur once every 30 minutes.

Let’s start with the stored procedure code:

MySQL — Stored procedure that deletes outdated information

```
1   DROP PROCEDURE IF EXISTS CLEAR_OUTDATED_OBJECTS;
2   DELIMITER $$
3   CREATE PROCEDURE CLEAR_OUTDATED_OBJECTS ()
4   BEGIN
5
6    UPDATE `user` SET `u_speed_bonus` = NULL
7     WHERE `u_speed_bonus_exp_dt` < UNIX_TIMESTAMP();
8
9    UPDATE `user` SET `u_volume_bonus` = NULL
10    WHERE `u_volume_bonus_exp_dt` < UNIX_TIMESTAMP();
11
12   UPDATE `user` SET `u_ban` = NULL
13    WHERE `u_ban_exp_dt` < UNIX_TIMESTAMP();
14
15   DELETE FROM `ip_blacklist`
16    WHERE `ibl_exp_dt` < UNIX_TIMESTAMP();
17
18   DELETE FROM `file`
19    WHERE `f_exp_dt` < UNIX_TIMESTAMP();
20
21   DELETE FROM `download_link`
22    WHERE `dl_exp_dt` < UNIX_TIMESTAMP();
23
24  END;
25  $$ DELIMITER ;
```

Each of the six queries sets to **`NULL`** the value of a foreign key or deletes a corresponding record if the specified information has expired.

Now we need to make the stored procedure run automatically once every 30 minutes. To do this let’s create an event:

MySQL — Creation of event to run the CLEAR_OUTDATED_OBJECTS stored procedure every 30 minutes

```
1   CREATE EVENT `clear_outdated_objects_every_30_minutes`
2    ON SCHEDULE
3     EVERY 30 MINUTE
4     STARTS '2000-01-01 00:01:00' ON COMPLETION PRESERVE ENABLE
5    DO
6     CALL CLEAR_OUTDATED_OBJECTS()
```

`STARTS` clause in line 4 of the query (we’d better put the date in the past) indicates the starting point (since the **`NEW_DAY`** procedure is already running exactly at midnight, we will put a small offset here, i.e., 00:01:00, so two procedures will not run in parallel and thus not decrease performance). Then this event will be repeated every 30 minutes (line 3 of the query).

Moving on to functions.

In the first task we need to define the user status (“novice”, “experienced”, “master”) depending on the number and volume of downloaded and uploaded files.

Let’s assume that we received the following information from the customer about how this status is determined (it is important that the “greater of achievements” is taken into account, i.e., if the user uploaded, e.g., only one file larger than 10 GB, he still gets the status of “master”):

Status	Uploaded files volume	Uploaded files count
Novice	< 1 GB	< 100
Experienced	1-10 GB	100-1000
Master	> 10 GB	> 1000

The algorithm for determining the status will be as follows:

- determine the status by the volume of uploaded files;
- determine the status by the count of uploaded files;
- choose the higher of the two statuses;

It would also be useful to add the ability to return the result as a number or as a string. Yes, the “number” here will also be a string, but it can always be converted to a “true number” later on.

So, here is the function code:

MySQL — Function that determines the user status

```
1   DROP FUNCTION IF EXISTS GET_USER_STATUS;
2   DELIMITER $$
3   CREATE FUNCTION GET_USER_STATUS(uploaded_volume BIGINT UNSIGNED,
4                                   uploaded_count BIGINT UNSIGNED,
5                                   return_mode VARCHAR(10))
6   RETURNS VARCHAR(150) DETERMINISTIC
7   BEGIN
8    DECLARE uploaded_volume_status INT;
9    DECLARE uploaded_count_status INT;
10   DECLARE final_status INT;
11
12   CASE
13    WHEN (uploaded_volume < 1073741824) THEN SET uploaded_volume_status = 1;
14    WHEN ((uploaded_volume >= 1073741824)
15      AND (uploaded_volume <= 10737418240)) THEN
16                                          SET uploaded_volume_status = 2;
17    WHEN (uploaded_volume > 10737418240) THEN SET uploaded_volume_status = 3;
18   END CASE;
19
20   CASE
21    WHEN (uploaded_count < 100) THEN SET uploaded_count_status = 1;
22    WHEN ((uploaded_count >= 100)
23      AND (uploaded_count <= 1000)) THEN SET uploaded_count_status = 2;
24    WHEN (uploaded_count > 1000) THEN SET uploaded_count_status = 3;
25   END CASE;
26
27
28   SET final_status = (SELECT GREATEST(uploaded_volume_status,
29                                       uploaded_count_status));
30
31   IF (return_mode = 'NUMBER')
32    THEN
33     RETURN CONCAT(final_status, '');
34    ELSE
35     CASE
36      WHEN (final_status = 1) THEN RETURN 'NOVICE';
37      WHEN (final_status = 2) THEN RETURN 'EXPERIENCED';
38      WHEN (final_status = 3) THEN RETURN 'MASTER';
39     END CASE;
40   END IF;
41
42  END;
43  $$
44
45  DELIMITER ;
```

In lines 12–18 the user status is determined by the volume of uploaded files, in lines 20–25 the user status is determined by the count of uploaded files, and in lines 28–29 the greater of these two statuses is chosen; finally, in lines 31–40 the format (number or string) of the result is determined, as well as the return of the result itself happened.

The `CONCAT(final_status, '')` expression in line 33 allows us to convert a number to a string representation, since this function must return `VARCHAR(150)` (as stated in line 6).

The `DETERMINISTIC` keyword in line 6 says that the function will always return the same result if it receives the same input values.

Using this function would require data preparation in advance. Technically, we could pass just the user ID to the function and do all the necessary queries internally, but this would be detrimental to performance[225].

So, here is an example of a query that uses this function:

MySQL — Example of a query to use the function that determines the user status

```
1   SELECT `u_id`,
2          `u_login`,
3          SUM(`f_size`) AS `files_volume`,
4          COUNT(`f_id`) AS `files_count`,
5          GET_USER_STATUS(SUM(`f_size`),
6                          COUNT(`f_id`),
7                          'NUMBER') AS `user_status`
8     FROM `user`
9     JOIN `file`
10      ON `u_id` = `f_owner`
11  GROUP BY (`u_id`)
```

Lines 3 and 4 are not necessary here and are presented just for clarity. Similar calculations are performed on lines 5 and 6 (and the result is immediately passed as a parameter to the `GET_USER_STATUS` function).

Once again, look at the size of the function code and the size of the query using it. And imagine that all the same calculations would have to be written directly into the query (by the way, the syntax of such a solution would be even more complicated). This example demonstrates the usefulness of stored functions very clearly.

And we are left with one last task: producing information about the file size in a human-readable form with a measurement unit adjustment (e.g., “10.35 MB”, “1.24 GB”, etc.).

The solution algorithm here is as follows:

- determine the range within which the file size falls (bytes, kilobytes, megabytes, etc.)
- round the result to hundredths;
- add measurement unit.

For maximum convenience, we can immediately implement calculation of size in units of degree 10 (KB, MB, GB, etc.) or in units of degree 2 (KiB, MiB, GiB, etc.).

[225] In a small "test experiment" the difference in performance for 500’000 users and 10’000’000 files was more than 100’000 times. I.e., the difference in performance is five orders of magnitude.

The function code looks like this:

MySQL — Function to present file size information in a human-readable form

```
1    DROP FUNCTION IF EXISTS NORMALIZE_SIZE;
2    DELIMITER $$
3    CREATE FUNCTION NORMALIZE_SIZE(size BIGINT UNSIGNED,
4                                   measurement VARCHAR(10))
5    RETURNS VARCHAR(150) DETERMINISTIC
6    BEGIN
7     DECLARE labels_2 VARCHAR(150)
8      DEFAULT '["B","KiB","MiB","GiB","TiB","PiB","EiB","ZiB","YiB"]';
9     DECLARE labels_10 VARCHAR(150)
10     DEFAULT '["B","KB","MB","GB","TB","PB","EB","ZB","YB"]';
11
12    DECLARE position_in_array_2 INT DEFAULT 0;
13    DECLARE position_in_array_10 INT DEFAULT 0;
14
15    DECLARE result_2 DOUBLE DEFAULT 0.0;
16    DECLARE result_10 DOUBLE DEFAULT 0.0;
17
18    SET position_in_array_2 = TRUNCATE(LOG(2, size) / LOG(2, 1024), 0);
19    SET position_in_array_10 = TRUNCATE(LOG(10, size) / LOG(10, 1000), 0);
20
21    SET result_2 = ROUND(size/POWER(1024, position_in_array_2), 2);
22    SET result_10 = ROUND(size/POWER(1000, position_in_array_10), 2);
23
24    IF (measurement = '2')
25     THEN
26      RETURN REPLACE(CONCAT(result_2, ' ', JSON_EXTRACT(labels_2,
27                    CONCAT('$[',position_in_array_2,']'))), '"', '');
28     ELSE
29      RETURN REPLACE(CONCAT(result_10, ' ', JSON_EXTRACT(labels_10,
30                    CONCAT('$[',position_in_array_10,']'))), '"', '');
31    END IF;
32
33   END;
34   $$
35
36   DELIMITER ;
```

When writing this code, we used some solutions beyond the scope of this book, so they will only be mentioned, and we will concentrate on what's going on here from the SQL point of view.

Lines 7–10 generate two `JSON` documents[226] as the easiest and fastest way to emulate an array in MySQL. Each of these documents contains a set of labels denoting measurement units in multiples of 2 and in multiples of 10.

Lines 18–19 define "in what range" the size value falls (bytes, kilobytes, etc.).

In lines 21–22 the final size value is determined in the corresponding units.

In lines 24–31 the final result is prepared and returned.

Here `JSON_EXTRACT` function allows us to retrieve an element at specified position of previously generated `JSON` document. The second parameter of this function is formed by the `CONCAT('$[',position_in_array_10,']')` expression, where `position_in_array_10` is the previously defined range of size values.

Using the `REPLACE` function is necessary to remove double quotes (the `"` sign), that will frame the result returned by `JSON_EXTRACT`.

[226] **JSON** — an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value). ("Wikipedia") [https://en.wikipedia.org/wiki/JSON]

Let's test the function and execute the following SQL-query:

MySQL — SQL-query to test the function

```
SELECT NORMALIZE_SIZE(1, '2'),
       NORMALIZE_SIZE (1, '10')
UNION
SELECT NORMALIZE_SIZE (100, '2'),
       NORMALIZE_SIZE (100, '10')
UNION
SELECT NORMALIZE_SIZE (1000, '2'),
       NORMALIZE_SIZE (1000, '10')
UNION
SELECT NORMALIZE_SIZE (10000, '2'),
       NORMALIZE_SIZE (10000, '10')
UNION
SELECT NORMALIZE_SIZE (1000000, '2'),
       NORMALIZE_SIZE (1000000, '10')
UNION
SELECT NORMALIZE_SIZE (250000000, '2'),
       NORMALIZE_SIZE (250000000, '10')
UNION
SELECT NORMALIZE_SIZE (47000000000, '2'),
       NORMALIZE_SIZE (47000000000, '10')
UNION
SELECT NORMALIZE_SIZE (9800000000000, '2'),
       NORMALIZE_SIZE (9800000000000, '10')
UNION
SELECT NORMALIZE_SIZE (7100000000000000, '2'),
       NORMALIZE_SIZE (7100000000000000, '10')
UNION
SELECT NORMALIZE_SIZE (5340000000000000000, '2'),
       NORMALIZE_SIZE (5340000000000000000, '10')
```

The result is as follows:

Bytes	In multiples of 2	In multiples of 10
1	1 B	1 B
100	100 B	100 B
1000	1000 B	1 KB
1000000	9.77 KiB	10 KB
1000000	976.56 KiB	1 MB
250000000	238.42 MiB	250 MB
47000000000	43.77 GiB	47 GB
9800000000000	8.91 TiB	9.8 TB
7100000000000000	6.31 PiB	7.1 PB
5340000000000000000	4.63 EiB	5.34 EB

To simplify the code, the created function does not consciously contain any checks, i.e., its behavior in some situations can be erroneous. To add such checks, the self-study task 5.4.d{363} is dedicated.



A lot of more complicated practical examples are given in Section 5 "Using stored functions and procedures" of the "Using MySQL, MS SQL Server and Oracle by examples[227]" book.

227 "Using MySQL, MS SQL Server and Oracle by examples" (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]



Task 5.4.b: create the stored routines you listed in task 5.4.a{355}.



Task 5.4.c: earlier{357} in this chapter, it was noted that after automating the execution of the NEW_DAY stored procedure, it is no longer necessary to call it from the triggers discussed in the previous section. Rework the code of the corresponding triggers to exclude the call of the specified stored procedure from them.



Task 5.4.d: earlier{360} in this chapter, when creating a function to produce information about file size in a human-readable form, we consciously (to simplify the code) did not create any checks, i.e., the behavior of this function in some situations can be erroneous. Add appropriate checks and restrictions to the function code.

5.5. Transactions

5.5.1. General Information on Transactions

So far, we have been talking about various database objects. Transactions belong to a different category — they are processes.



Transaction[228] — a set of database operations, which is an indivisible logical unit. Such a set of operations can either be executed completely and successfully (in compliance with all database consistency rules{67} and regardless of concurrently executed transactions), or not executed at all (in which case none of the operations included in this set should produce any changes in the database).

Simplified: a set of operations that either completes entirely and successfully, or also entirely cancels in case of an error in any of its operations.

The definition cited in the footnote[228] also states that a transaction is the unit of recovery and concurrency.

So, it turns out that a transaction:

- is always either executed or not executed in its entirety;
- is used when recovering from various failures and faults;
- provides a mechanism for competitive access to data.

Let's demonstrate all these properties graphically (see figure 5.5.a). For simplicity, let's take the example presented in the vast majority of literature (but let's supplement and extend it): the transfer of money between two accounts.

Suppose that two clients (e.g., two accountants of one company) almost simultaneously transfer funds from an account ("almost", because even if the time to nanoseconds coincide, one of the operations will still be considered to have started earlier).

From the moment when client 1 changes account A, the DBMS blocks client 2's actions to change the same account (since it is not yet clear how the actions of client 1 will end).

Some kind of error occurs in client 1 (let's assume that the cause is outside the DBMS, i.e., it is some problem with the storage or something similar).

In this situation, it is no longer possible to complete the money transfer performed by client 1 (because it is necessary not only to reduce the balance of the source account, but also to increase the balance of the destination account). Therefore, operations on account B will no longer be performed.

Instead, the DBMS will cancel the previously chosen data modification operations (i.e., account A balance modification) and end the transaction (reporting the error).

As soon as the transaction of client 1 is completed, client 2 can continue working with account A. In our example, client 2 did not encounter any errors and the money transfer from account A to account C was successful.

[228] **Transaction** — a unit of recovery and concurrency; loosely, a unit of work. Transactions are all or nothing, in the sense that they either execute in their entirety or have no effect (other than returning a status code or equivalent, perhaps). Transactions are often said to be a unit of integrity (or consistency) also. ("The New Relational Database Dictionary", C.J. Date)

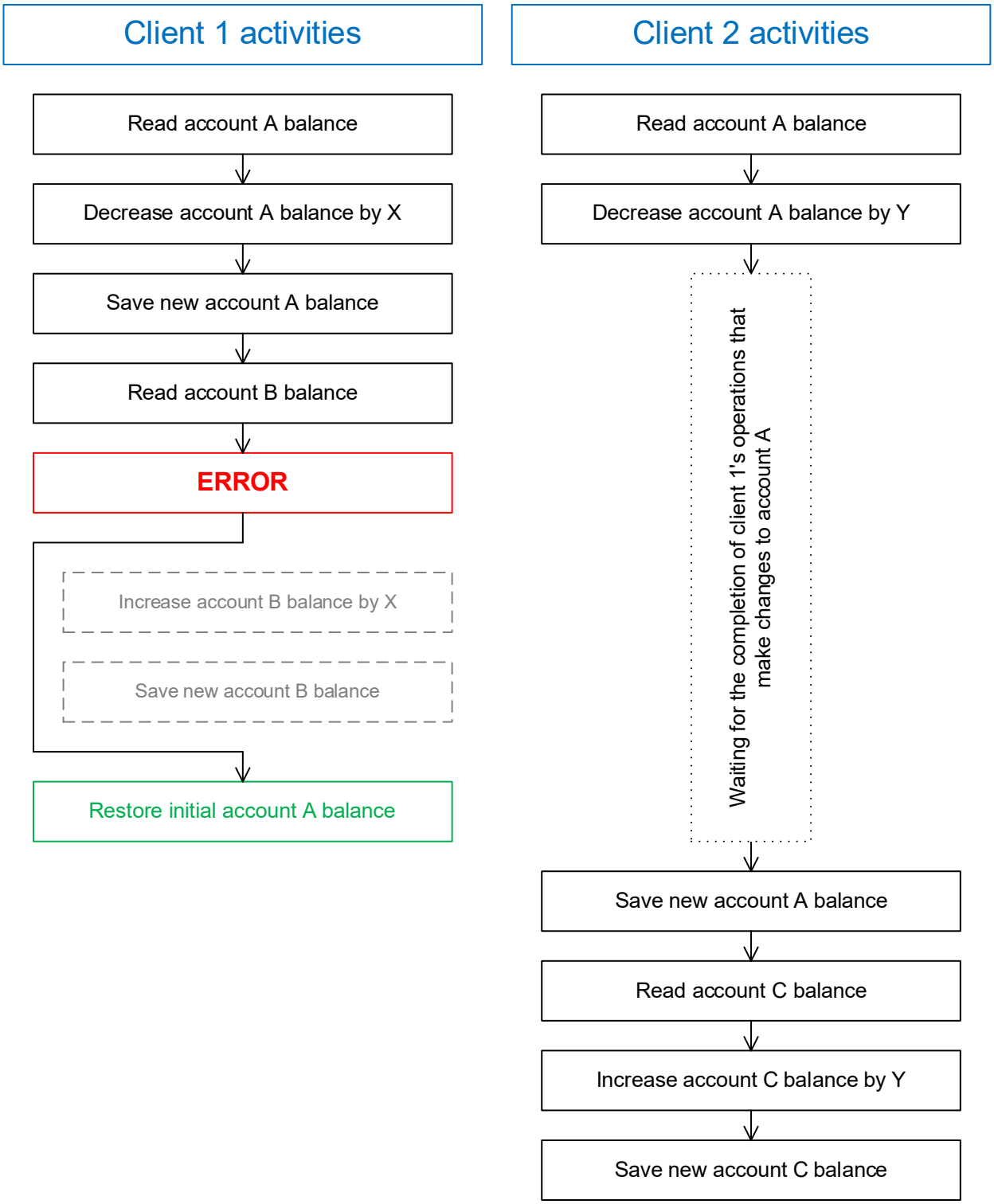


Figure 5.5.a — Transaction logic

So, one more time. A transaction:

- always is either executed or is not executed in its entirety, e.g., when client 1's transaction encountered problems, all of its changes were undone, client 2's work was completed successfully, and all of its changes became effective;
- is used when recovering from various failures and malfunctions, e.g., when client 1's operation encountered problems, all of its changes were undone, and the data was brought back to the state it was in before client 1's operations began;
- provides a mechanism for competitive access to data, e.g., the DBMS did not allow two clients to simultaneously modify the same account.

From the definition of a transaction (and so far, we have, in fact, considered its definition in detail) the properties[229] of a transaction follow. These four properties are known by the ACID abbreviation.

Atomicity means that a transaction cannot be "partially executed", i.e., that always all of its operations will either be executed (all and to the end) or not executed (none, i.e., all data changes will be cancelled). In the example in figure 5.5.a the transaction of client 2 was executed completely to the end, and the transaction of client 1 was "completely unexecuted", i.e., all changes that occurred during its execution were canceled.

Consistency means that a successfully completed transaction is guaranteed to preserve database consistency{67}, i.e., it records only acceptable data alteration results (which do not contradict any restrictions implemented at the DBMS level or added separately to the database in the form of checks{338}, triggers{341} etc.). In some cases, additional conditions are tested directly in the transaction code.

It is important to note that during the execution of a transaction, database consistency{67} can be violated, so, in the example in figure 5.5.a during the execution of transactions by both clients, there is a moment when the balance of the source account has already decreased, but the balance of the destination account has not yet increased, i.e., money "gone to nowhere". However, the DBMS "does not show" this internal state to other transactions working with the same data, i.e., from the outside transactions such consistency violations are (almost) never visible. "Almost" because there are still ways to access intermediate database states at your own risk; this will be mentioned when we consider transaction isolation levels{367}.

Isolation means that several transactions running in parallel must never affect each other's execution. If this property did not exist, we could have the situation shown in figure 5.5.b: each transaction first reads the balance of account A, then changes it, and finally checks that account A has changed by the expected amount. But since the concurrent transaction also changes this account, of course the check will fail, because the actual account balance will be different from the expected one.

We will talk more about this property when we consider the already mentioned transaction isolation levels{367}.

Durability means that the DBMS itself solves all "internal problems" and guarantees that after a transaction completion (both successful and unsuccessful) all necessary changes have been either committed or cancelled, and that the database will not fall to some "intermediate state". This property is usually associated with resistance to hardware failures, network communication glitches, etc. Even in such conditions, the DBMS always "knows" which operations have been performed and, after returning to normal operating mode, can make sure that the data is not damaged (or, in the worst case, report on the problems encountered).

[229] We emphasize in particular that here we are talking about the so-called "theoretical" properties of transactions, because in real DBMS (especially distributed ones) the situation can be so complicated that we have to look for various compromises and allow a series of exceptions from the rules defined by these properties.

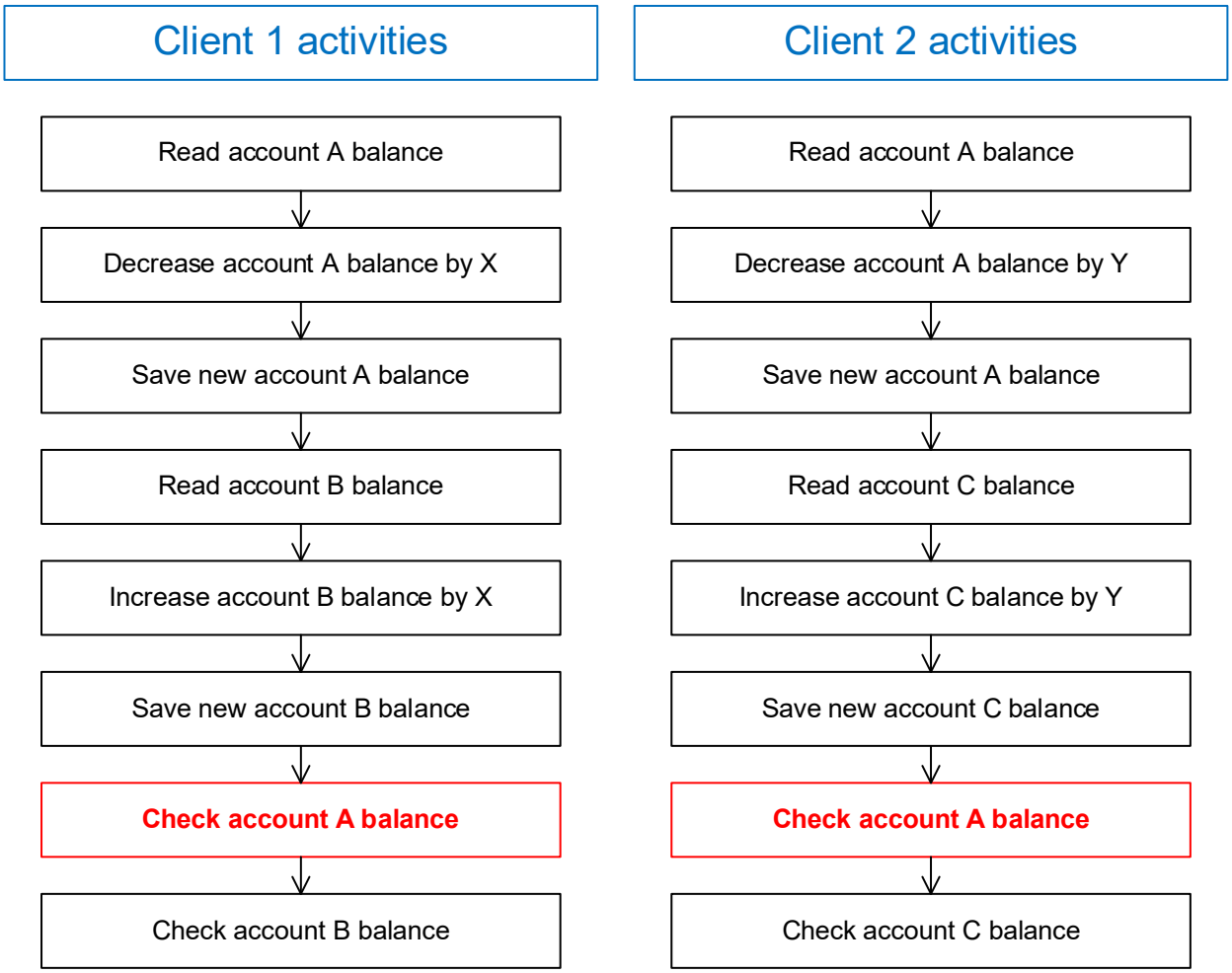


Figure 5.5.b — Violation of transactions isolation

Let's move on to consider the transaction isolation levels already mentioned.

!!! **Transaction isolation level**[230] — a value indicating how accessible the internal state of the database at the moment of transaction execution is to other, simultaneously executed transactions.

Simplified: to what extent the simultaneously executed transactions are "protected" from each other.

Figure 5.5.b shows an example of a transaction isolation violation. Indeed, if the DBMS allowed two transactions to change the same data at the same time, checking the balance of account A would fail (figure 5.5.a shows that the second transaction is put on hold, which avoids this problem).

But what if we were interested not in the exact change in the balance of account A, but simply in the fact that the balance has changed? Or if we were graphing in real time the changes in the funds available to the client (where we would allow some inaccuracies for the sake of speed)? Or if there were some more complicated situations, where we needed to provide some strictly specified behavior of the DBMS?

Obviously, such questions have been raised by many professionals working with databases, and there is a ready answer: we can control the level of isolation of transactions, i.e., the degree of their mutual influence.

Before we look at the levels themselves, it is necessary to explain what typical problems can occur when multiple transactions access the same data at the same time.

[230] **Transaction isolation level** — a measure of the extent to which transaction isolation succeeds ("Transaction Isolation Levels"). [https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-ver15]

Lost update — only those data changes that were made later are saved.

A very simplified example: several employees in the office are arguing about the settings of the air conditioner, i.e., someone turns it on, someone turns it off, someone makes it warmer, someone cooler; it does not matter who did what earlier, the air conditioner is always set the way it was set "at the very last moment".

A graphical explanation is shown in figure 5.5.c.

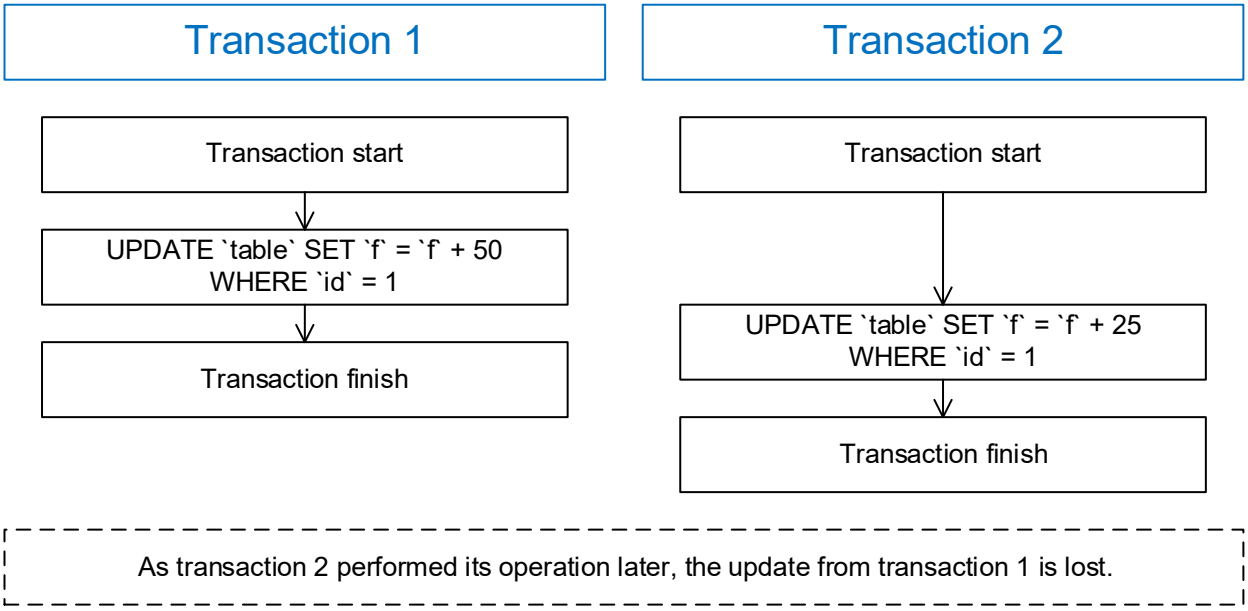


Figure 5.5.c — Lost update

Dirty read — the temporary state of data becomes available, which will later be deleted or changed due to the cancellation of the transaction that worked with them.

A very simplified example: a child overhears that his parents are going to give him a bicycle for his birthday, and happily runs to tell his friends; five minutes later the parents change their mind.

A graphical explanation is shown in figure 5.5.d.

Non-repeatable read — the same data changes over the transaction's lifetime (i.e., re-reading previously read data leads to a new result).

A very simplified example: you decided to have a cup of tea; you glanced in the cupboard to see what kind of tea was there; while you were heating the kettle, someone replaced the tea in the cupboard with another variety (or even with a coffee); you open the cupboard again and are very surprised, because just now you saw a different picture there.

A graphical explanation is shown in figure 5.5.e.

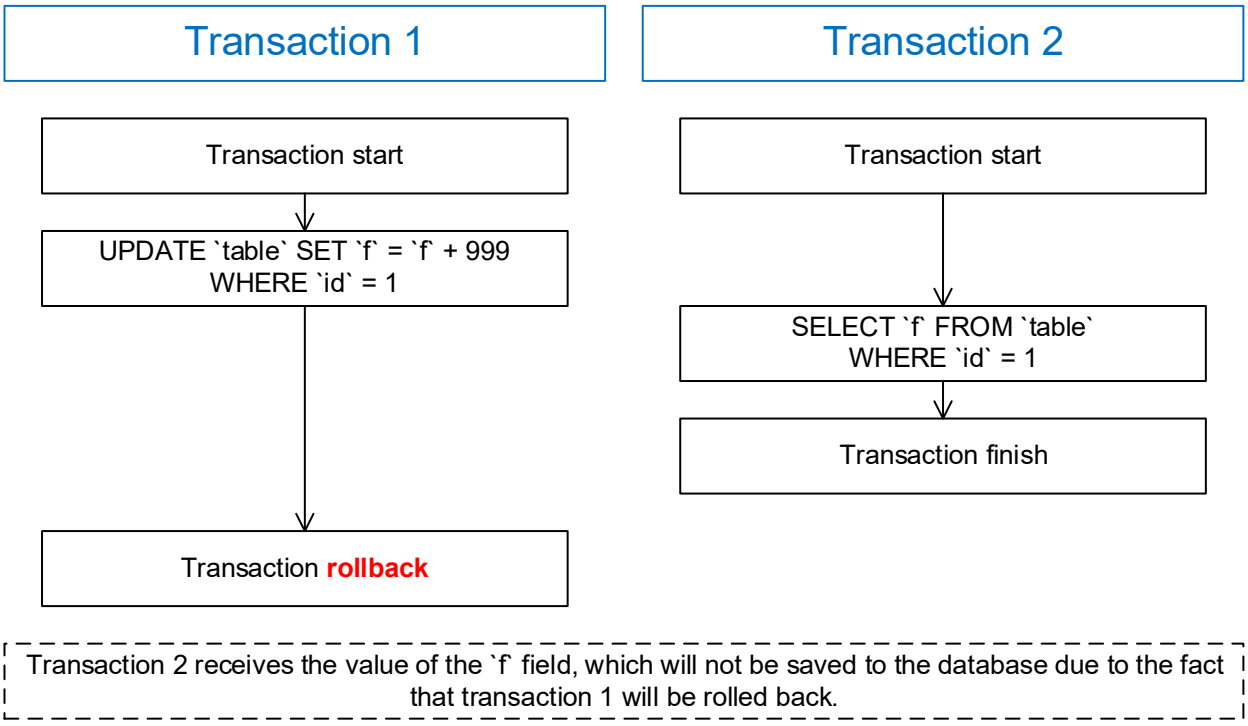


Figure 5.5.d — Dirty read

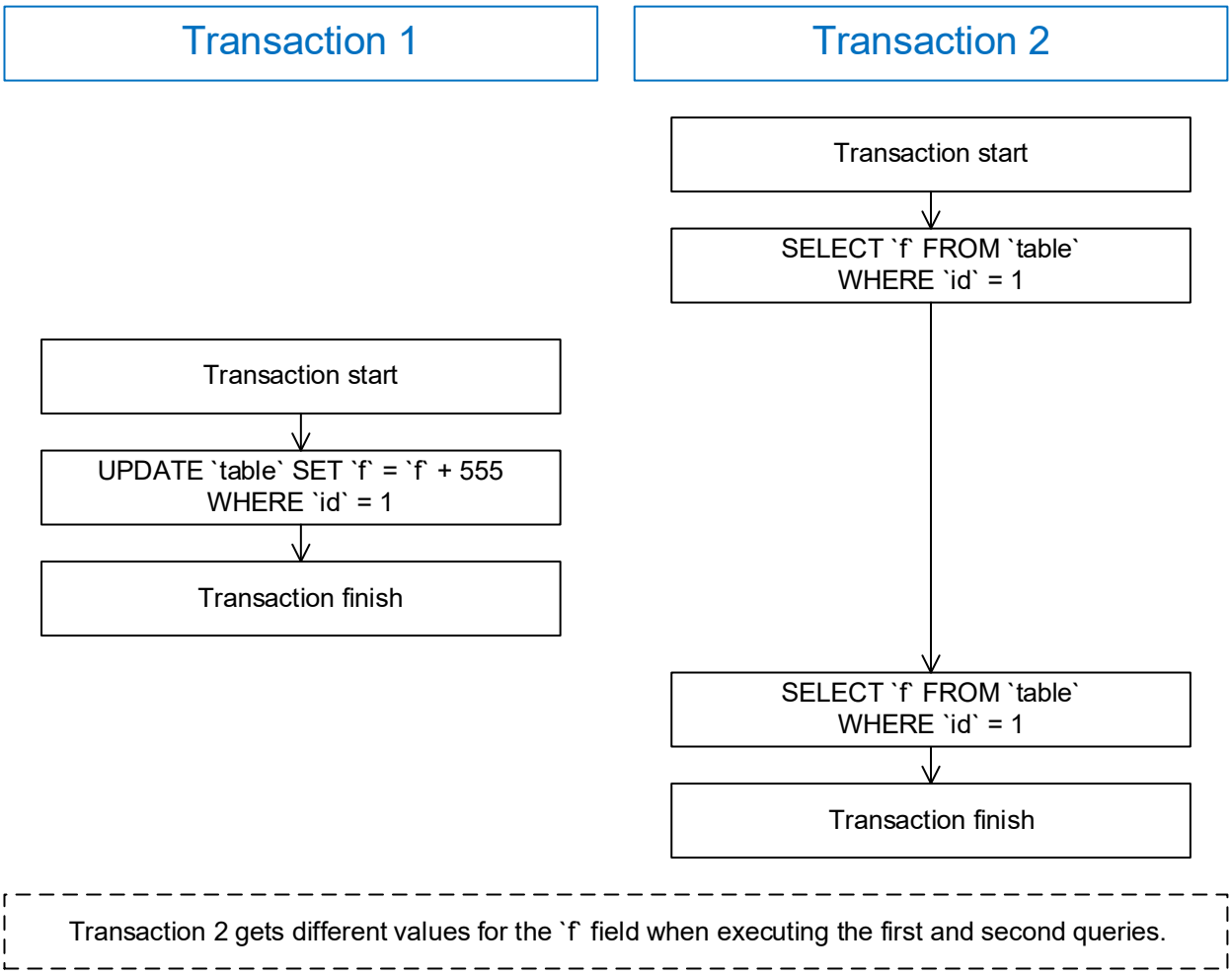


Figure 5.5.e — Non-repeatable read

Phantom reads — the number of rows to be processed changes (due to adding or deleting rows or changing values in their fields).

A very simplified example: you want to take a picture of three sparrows sitting on a branch; while you are distracted by the camera settings, two more sparrows fly in.

A graphical explanation is shown in figure 5.5.f.

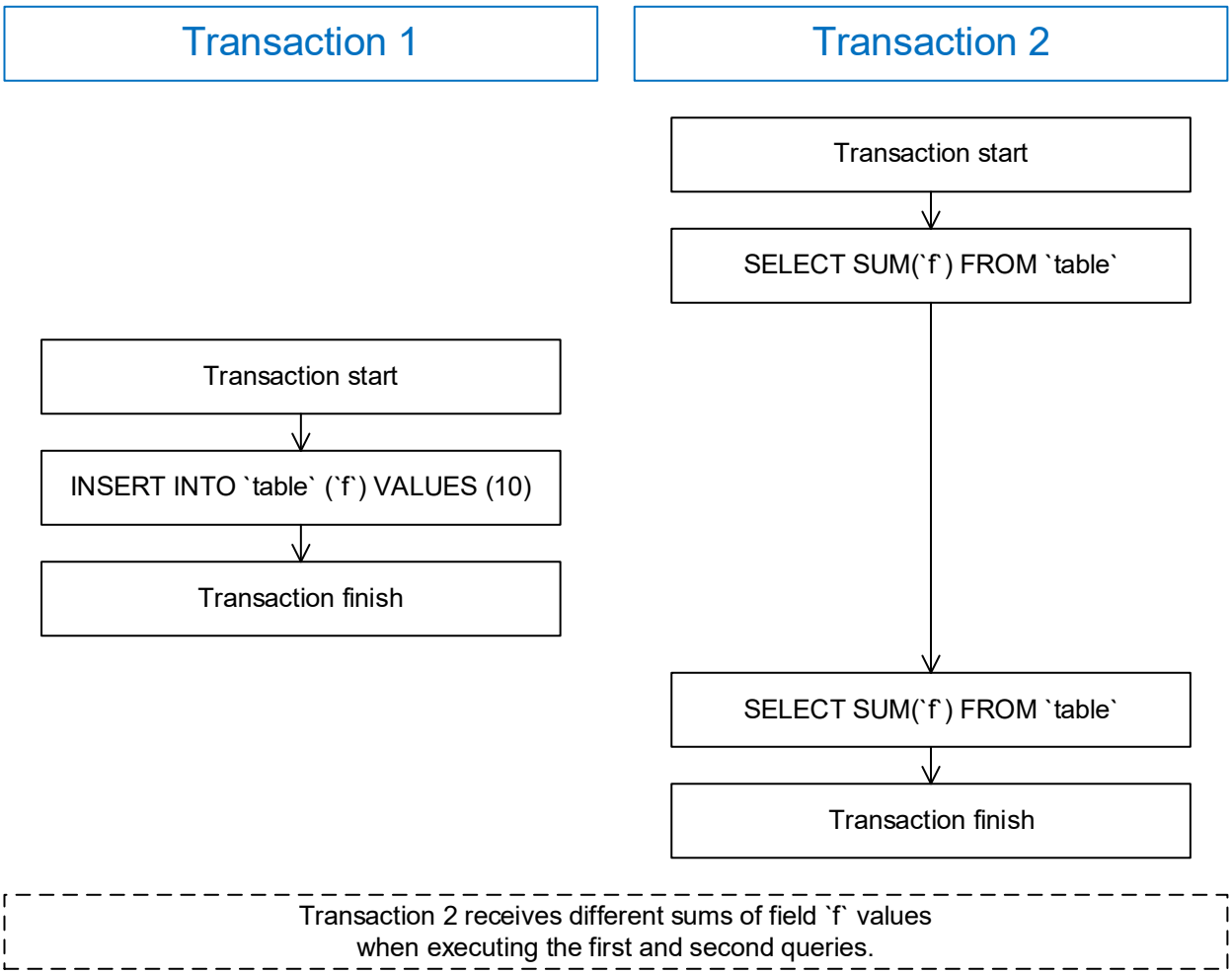


Figure 5.5.f — Phantom reads

The key difference between non-repeatable read and phantom reads lies in the nature of their occurrence: non-repeatable read deals with changes of the data itself, while phantom reads deal with changes of the amount of data (i.e., with “extra” or “missing” records).

Now let’s consider the isolation levels themselves. When a transaction is executed with some isolation level, the DBMS “protects” it from the problems described above. The isolation levels represent a hierarchy, where each next (higher) level includes all the “protection mechanisms” of the previous (lower) levels.

The hierarchy of transaction isolation levels is shown in figure 5.5.g.

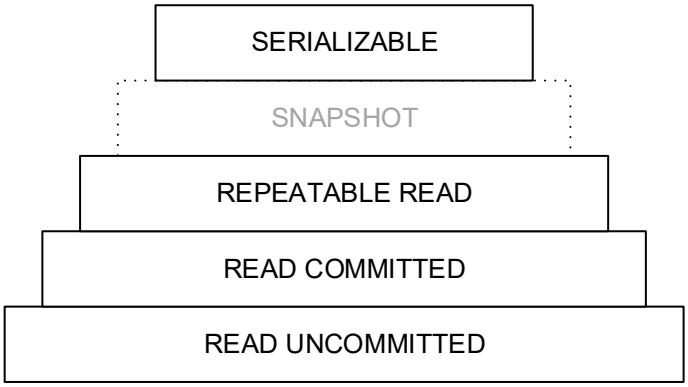


Figure 5.5.g — Hierarchy of transaction isolation levels

Read uncommitted allows reading uncommitted (i.e., before transaction commit or rollback) changes made by any transaction (both the one performing the reading and those running in parallel with it).

This level ensures that there are no lost updates, i.e., if several concurrent transactions modify the same data, the data will eventually have the value obtained by sequentially applying all the changes made. All other problems (dirty read, non-repeatable read, phantom reads) are still relevant.

Physically, to protect against lost updates, a locking of the data being changed is applied, i.e., parallel changes to the same data are in fact executed sequentially (queued up).

No read operations at this isolation level are blocked.

Read committed allows reading all changes made by the transaction itself and only validated (committed) changes made by other (concurrent) transactions.

This level ensures that there are no lost updates or dirty reads, while allowing for non-repeatable reads and phantom reads.

Physically, this level is implemented using data locking or versioning:

- with locking, the transaction modifying the data blocks the reading of that data for concurrent transactions running at the “read committed” level and higher;
- with versioning, the DBMS creates a new version (copy) of the data being modified for the transaction that modifies the data, and gives all the other (concurrent) transactions access to the old (unmodified) data.

Both options have many advantages and disadvantages, as well as implementation peculiarities, so details can be found only in the documentation of the specific version of your particular DBMS.

Repeatable read allows reading only changes made by the transaction itself, and the data read by it becomes inaccessible to concurrent transactions.

This level ensures that there are no lost updates, dirty reads, and non-repeatable reads, but allows for phantom reads.

Physically, this level is implemented by locking the read data, which prohibits concurrent transactions from changing the corresponding table records. But concurrent transactions may insert new rows, which may cause the problem of phantom reads.

Snapshot is a special (higher) case of “repeatable read” that is supported only by some DBMSes; is allows only reading of changes performed by the transaction itself, and the data read by it remains available for modification by concurrent transactions (this is the main difference from the “repeatable read” level).

Serializable allows only the execution of data changes, as if all data-modifying transactions were executed sequentially, not in parallel.

This level ensures that all problems, i.e., lost update, dirty read, non-repeatable read, and phantom reads are avoided.

Physically, this is achieved through transaction queue management and a sophisticated locking mechanism. This is the most reliable level of transaction isolation in terms of data accuracy, but it is also the slowest in terms of performance.

For simplicity and clarity, let's summarize the information about "what prevents what" in a single table.

	Lost update	Dirty read	Non-repeatable read	Phantom reads
Read uncommitted	**Prevents**	**Allows**	**Allows**	**Allows**
Read committed	**Prevents**	**Prevents**	**Allows**	**Allows**
Repeatable read	**Prevents**	**Prevents**	**Prevents**	**Allows**
Snapshot	**Prevents**	**Prevents**	**Prevents**	**Allows**
Serializable	**Prevents**	**Prevents**	**Prevents**	**Prevents**

You may still be wondering why there are so many different levels, approaches, and solutions. All this variety allows us to achieve the optimal combination of necessary data safety, accuracy, and operation performance.



Section 6.2.2 "Interaction of Concurrent Transactions" of the "Using MySQL, MS SQL Server and Oracle by examples[231]" book presents the results of experiments on cross-interaction of transactions in all possible isolation levels. These results show that the actual implementation of transaction mechanisms in a particular DBMS may differ from the theoretical assumptions.



Task 5.5.a: for the "Bank{395}" database, make a list of sequences of operations that should be performed within a single transaction. For each resulting transaction, specify its minimum acceptable level of isolation.

231 "Using MySQL, MS SQL Server and Oracle by Examples" (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

5.5.2. Transactions Management

Perhaps after such a rather complicated and voluminous theoretical material, you would expect transactions management to look complicated as well. Yes and no.

“No” in the sense that the transaction management syntax is very simple (we will look at it right now).

“Yes” in the sense that the real need for transaction management occurs in quite complex situations, and the transaction mechanisms themselves in different DBMSes have many non-obvious features, and together this results in high complexity of thinking, implementation and debugging of appropriate solutions.

But let’s start with the syntax. Any transaction has a start and one of two ways to end — successful and unsuccessful (see figure 5.5.h).

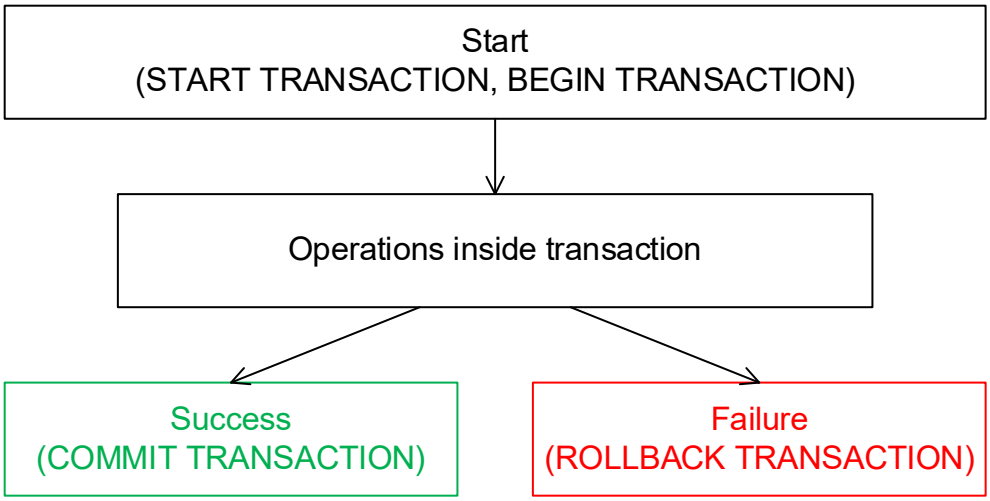


Figure 5.5.h — Sequence of operations in transactions

More complex cases (nested transactions, return to recovery points, etc.) are supported differently in different DBMSes and are objectively beyond the scope of this book.

Next, for clarity, we will use MySQL syntax (especially since our educational database of file-sharing service works exactly under the control of this DBMS):

- the start of the transaction is indicated by the **`START TRANSACTION`** command;
- successful completion of a transaction (commit) is indicated by the **`COMMIT`** command;
- unsuccessful completion of a transaction (cancellation, rollback[232]) is indicated by the **`ROLLBACK`** command.

I.e., the simplest case of using transactions would look like this (e.g., deleting a user with ID 1000):

MySQL — An example of the simplest transaction

```mysql
START TRANSACTION;
DELETE FROM `user`
      WHERE `u_id` = 1000;
COMMIT;
```

[232] For national languages speakers it is still necessary to mention that in the context of transactions, even in non-English audience, anglicisms like "commit", "rollback", etc., are almost always used as they are much shorter, clearer, more universal and simpler than similar official terms in other languages.

However, we can go even further. After all, how does “just a query” (if we don’t frame it with `START TRANSACTION` / `COMMIT` commands) work?

In fact, it works exactly the same, because the DBMS implements so called “implicit transactions”, i.e., the `START TRANSACTION` / `COMMIT` commands are executed without our involvement.

This DBMS behavior can even be controlled to some extent (in MySQL, the `autocommit` parameter is responsible for automatic commit of transactions, and the `SET autocommit = 0` command turns off automatic commit, after which all data changes have to be explicitly fixed by the `COMMIT` command).

Implicit transactions are convenient because very often the desired effect is achieved by executing just a single SQL query, and there is no need to combine several actions in a transaction; therefore, it is logical that such a “microtransaction” consisting of a single query is automatically committed (in case of successful execution) or automatically rolled back (in case of a query execution error).

Implicit transactions are executed at the “default isolation level” (every DBMS has its own one, which is set up in the DBMS configuration and may be specified by the corresponding commands).

MySQL uses the `SELECT @@TX_ISOLATION` command to determine the current isolation level (i.e., the default isolation level if you have not explicitly changed it). By default, this database works at “repeatable read{371}” level.

In MySQL, the command `SET TRANSACTION ISOLATION LEVEL level_name` is used to change the isolation level of transactions.



Before we look at the example, let’s emphasize that transactions management (as well as any other tool) is not a goal in itself. If using implicit transactions at the default isolation level, everything works, if you, the customer, and the users are happy, if there are no problems, then there is no need to complicate things.

As an example, consider solving the following problems:

- quickly get data for “file uploads per date” chart;
- in a guaranteed manner determine how many users are in each role right now.

To solve the first problem, it is important to remember that the upload datetime of an already uploaded file cannot be changed (at least in theory), while other information about files can be changed (and the fact of its change will lead to locking the corresponding rows of the table or even of the whole table[233]).

Hence, we conclude that changes to fields other than `f_upload_dt` are of no interest to us when performing this task, and they should not interfere in any way with our goal of quickly getting data for “file uploads per date” chart.

That is, we must read data from rows blocked by any update process. That is, we must read data from yet uncommitted records. That is, we need the “read uncommitted{371}” isolation level.

The code looks like this:

MySQL — Quick selection of data for “file uploads per date” chart

```
1    SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
2    START TRANSACTION;
3    SELECT YEAR(FROM_UNIXTIME(`f_upload_dt`)) AS `Y`,
4           MONTH(FROM_UNIXTIME(`f_upload_dt`)) AS `M`,
5           DAY(FROM_UNIXTIME(`f_upload_dt`)) AS `D`,
6           COUNT(`f_id`) AS `files`
7      FROM `file`
8     GROUP BY `Y`, `M`, `D`;
9    COMMIT;
```

[233] This behavior may vary with different MySQL versions, different storage engine settings, or different storage engines. This statement is given as an introductory condition for clarity.

In the second problem the keyword is “guaranteed”, i.e., here we can reinsure a little (yet the default isolation level of “repeatable read{371}” should be enough) and choose the most secure level, the “serializable{372}” one.

The code looks like this:

MySQL — Guaranteed determination of the number of users in each role

```
1    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
2    START TRANSACTION;
3    SELECT `r_name`,
4           COUNT(`ur_user`) as `users`
5      FROM `role`
6      JOIN `m2m_user_role`
7        ON `r_id` = `ur_role`;
8    COMMIT;
```

And another good example of (implicit) transactions management we’ve seen before{351} — the `SIGNAL SQLSTATE` command in triggers, which causes an exceptional situation and rolls back a running transaction.



Many more advanced practical examples are given in Section 6 “Using Transactions” of the “Using MySQL, MS SQL Server and Oracle by examples” book[234].



Task 5.5.b: based on the results of task 5.5.a{372}, write the appropriate SQL code and experimentally check your assumptions. Think of at least three “dangerous situations” for each sequence of operations from your list, in which an incorrect level of transaction isolation may cause some failures.

[234] “Using MySQL, MS SQL Server and Oracle by Examples” (Svyatoslav Kulikov) [https://svyatoslav.biz/database_book/]

Chapter 6: Database Quality Assurance

6.1. Database Quality Assurance at the Design Stage

6.1.1. General Approaches to Database Quality Assurance at the Design Stage

In fact, all of the material presented above in this book touches on issues of database design quality in one way or another. Some particularly important issues are even dealt with explicitly{10}. However, there are a few additional areas that deserve more attention[235].

Both the data itself and the database structures for data storage and processing must ensure the following principles.

Completeness. One of the common mistakes of novice database developers is an incomplete analysis of the subject area and, as a consequence, the absence of certain information in the database. E.g., when describing an employee of a certain organization, they specify only full name and date of birth, but forget about the residential address, telephone numbers (yes, as a rule, there are several), e-mail addresses (also several), passport data, information on education, etc., etc.

Indeed, it is not necessary to store all these (and many other) data in every case. But it becomes all the more important to carefully figure out which data are still needed. And all relevant decisions must be reflected in the database tables, fields, relationships, etc.

Uniqueness. This issue has previously been addressed in the context of normalization requirements, where it was presented as “data nonredundancy requirement{209}”.

From the moment the same data starts to be stored in two or more places in the database, it’s only a matter of time before problems arise, and those problems *are sure to be*.

If some insurmountable (are they *really* insurmountable?) circumstances prevent the uniqueness of some data storage, every effort should be made to ensure the synchronization (consistency) of these data by the built-in DBMS tools.

Tracking changes over time. Problems with compliance with this principle are even more common than problems with data completeness{376}. The situation is exacerbated by the fact that even the customer rarely thinks about the fact that some data will need to be processed with respect to time.

Here’s a trivial example: two large companies are about to sign a contract for the supply of a wide range of products; at the time of negotiations the price of all items suited the buying party; at the time of signing the contract price has already changed, and now no one remembers the exact prices of hundreds of positions of products on the day of negotiations.

If the database considered the change in price over time, this problem would not arise, because the price of products on the day of negotiations would be known exactly.

The same problem is indirectly related to untimely updating of caching tables and fields (if any in the database) and in general to “data up-to-date state requirement{216}”.

[235] You can read more in the “What is Data Quality and How Do You Measure It for Best Results?” article by Neil Patel [https://neilpatel.com/blog/data-quality/]

Correctness (validity). The fulfillment of this principle often forces one to search for a middle ground between flexibility and adherence to standards and constraints.

Should we, for example:

- Strictly limit the format of the phone number?
- Introduce some rules for “Name” and similar fields?
- Allow the choice of school from the list or allow user’s manual input?

In the case of strict restrictions, we simplify data processing, but we confuse users whose real data do not comply with our rules (e.g., what should a user specify as his school, if it is not on the list, and they cannot enter “manually” their version?)

If we increase flexibility, we make life easier for users, but data processing sometimes turns into a living hell (a real example: on one educational resource, users were able to “think up” more than 250 variants of the name of the same university).

Obviously, there are hybrid solutions (e.g., typing prompts and selecting a suggestion or the ability to add your own option), but they require additional implementation effort, have their limitations, and ultimately it all comes down to the specifics of the subject area and customer decisions (e.g., it makes sense to allow your own spelling of “name”, but it does not make sense to allow your own spelling of credit card number).

The main thing is to remember that such issues will arise, and to work through them at the stage of database design.

Accuracy. Most of all, this issue applies to any fractional value. With what accuracy should the user’s height or weight be entered? With what accuracy should we store the price of a product? How accurately should the time of an event be recorded?

On the one hand, it may seem that “the more accurate the better”. But remember that in this way we increase the volume of stored data and run the risk of problems with the accuracy of calculations (fractional values are usually stored and processed with approximate accuracy[236]).

On the other hand, the lack of accuracy of storage may not allow us to reflect the real properties of some person, object, process, etc. in the database.

The solution here, as in many other cases, lies in the context of the requirements of the subject area and the customer’s decisions.

Uniformity of presentation. Violation of this principle is rare and is not fatal, but it is still quite strange to see (for example) that the date of birth of an employee is stored with the accuracy of a day, and the date of birth of the employee’s child is stored with the accuracy of a second. Or that an employee’s work phone is stored with a country and operator code, while his personal phone number is stored without these codes (or that these numbers are simply stored in different formats.

Such meaningless diversity greatly violates the requirement of database usability[{10}].

[236] There is even a dedicated website for this question, with many examples and explanations. See https://0.30000000000000004.com.

At the end of this chapter, here is a small gradation of database quality[237] that will allow you to quickly assess how good your database is (or needs improvement).

Level	Description	Examples of problems
5	No major problems were found. The database is logical, consistent, and easy to maintain and modify.	-
4	Some problems arise that do not affect the applications working with the database. Such problems are fairly easy to detect and fix.	The data types are not unified, mandatory fields are not marked as **`NOT NULL`**, unique indexes are not placed, field names are not always meaningful.
3	There are serious problems with the database, obviously affecting the user experience (crashes, poor performance, etc.)	Primary keys are missing, indexes are missing (including indexes on foreign keys), data types require conversion in many queries, some of the principles discussed at the beginning of this chapter are violated.
2	There are critical problems with the database, occasionally causing applications that work with it to fail.	Most of the principles discussed at the beginning of this chapter are violated, and there are data operation anomalies{157}.
1	The database is practically inoperable.	All the principles discussed at the beginning of this chapter are violated, the fundamental database requirements{10} are violated, and there are a lot of data operation anomalies{157}.

Unfortunately, it often happens that the database developers see the result of their work as related to level 5, but in reality, this level may be much lower.

That is why the process of database design is such a long and non-trivial task, and its result requires such a thorough check.



You can read more about quality assurance “in general” in the “Software Testing. Base Course.[238]” book, and for more on data and database quality, see the “Information and Database Quality[239]” book.



Task 6.1.a: what principles discussed in this chapter are violated in the “Bank{395}” database? Make a list with the essence of the violation and recommendations for fixing the situation.



Task 6.1.b: what principles discussed in this chapter are violated in the fileshare service database{278}, {295}, {305}? Make a list with the essence of the violation and recommendations for fixing the situation.

[237] For details, see the “Grading Database Quality” article by Michael Blaha [https://www.dataversity.net/grading-database-quality-part-1-database-designs/]

[238] “Software Testing. Base Course.” (Svyatoslav Kulikov) [https://svyatoslav.biz/software_testing_book/]

[239] “Information and Database Quality” (Mario G. Piattini, Coral Calero, Marcela Genero).

6.1.2. Tools and Techniques of Database Quality Assurance at the Design Stage

As we remember, the database design process can be top-down{10} and bottom-up{10}. And in the second case, it is extremely difficult to concentrate solely on design without at least partially touching the issue of database operations.

But such a strict separation is not required. Here we just pay more attention to the quality issues at the design stage, and about its exploitation we will talk in more detail in the next section{381}.

Unfortunately, there is no “magic” tool that would allow to immediately and unambiguously show the design errors. And this is a fundamental problem, because even the most perfect tool does not have complete information about the subject area, the wishes of the customer and the needs of users.

That is, the most effective “tool” in this case is a specialist who deals with database design.

Since it is impossible to build a schema of any kind of complex database without special tools, examples of appropriate solutions were considered earlier for design at the conceptual{280}, datalogical{296} and physical{310} levels.

But these solutions can only simplify the process of collecting, analyzing, and displaying the information needed to successfully design a quality database.

The main part of the load falls on the corresponding actions (techniques, approaches), the most relevant of which we will now consider[240].

Collecting, analyzing, and discussing information. There are many sources of information in database design (e.g., the customer, users, database application developers, objectively existing subject area constraints, etc.) — each such source must be identified, and the information from it should be obtained, discussed with the other involved parties, and reflected in the database schema.

If the opinion of at least one interested party was ignored, we can safely say that the database will not fully meet the needs of those people for whom it was created.

Choosing the technical solutions. As a rule, problems here are rare, but fatal. The choice of the type of DBMS, the specific version and implementation of that DBMS, and the associated infrastructure can (and often should) affect the schema of the database being designed. Here it is important to get justified decisions and get them in advance, so that there is no need to redo some of the work already done.

Control and self-control. Even under ideal conditions, almost unattainable in real life (when all information is available, all parties involved agree on everything, nothing is forgotten, etc.) there is still a very high probability of something being mixed up, forgotten, incorrectly considered, and incorrectly reflected in the database schema.

That is why it is so important to double-check the results of your work, to get feedback from fellow experts and from all interested parties. The previously mentioned discussion of information, as well as experimentation, is also very helpful here.

Experiments. Starting from the datalogical{295} design level, it is already possible to periodically export the database schema to DBMS, fill it with test data and examine its behavior. This approach allows you not only to see performance problems and other hard-to-predict difficulties, but also to show a number of annoying gross errors (usually caused by inattention and easily traceable by the DBMS itself — from which you will get quite specific messages about specific errors).

[240] We deliberately do not cover issues of enterprise-level database design, as that affect the processes of project management, staff training, quality management, and many other issues beyond the scope of the book for beginners.

"What if?" vs "Yay, it works!" A typical mistake of novice specialists is to stop analyzing the problem after the first working solution is found. But firstly, this solution may not be the best (and further analysis would have found a better one), and secondly, this first caught solution may only cover particular cases, and in other situations (with other data, other execution conditions, other way of interaction with the database, etc.) lead to errors.

Therefore, the search for several alternatives, the selection of the best of them and the analysis of selected solutions in different situations significantly improves the quality of the developed database.

Thorough documentation. Presumably, several people will be working on a more or less complex database for a long time. How do you not forget what you came up with six months ago? How do you save your colleagues from having to constantly wonder, look for the author of this or that solution, and ask them for details? The answer is simple — documentation.

From comments to tables and their fields, to full-fledged graphical charts and accompanying detailed text descriptions — all this becomes more necessary the more extensive the database has to be designed.

Expanding your own horizons. This technique does not apply directly to databases (it is equally helpful in any activity), but nevertheless: the more you know about the subject area, the more you know about databases and related technologies (operating systems, computer networks, programming languages, etc.), the easier you will notice potential problem areas, the easier you will ask the right questions and, having received answers to them, improve the quality of the database being designed.

Of course, we want to believe that you will be lucky enough to work on carefully organized projects with a well-developed quality assurance system, and that true professionals will share their experience with you — in such an ideal environment, many of the recommendations given here will be automatically assimilated.

But even if you are alone in creating your first database, the ideas presented above are already applicable, and will help you improve the quality of the designed database.



Task 6.1.c: which of the techniques and approaches presented in this chapter caused you the most difficulty in working with the "Bank{395}" database and the filesharing service database{278}, {295}, {305}? Make a list and for each item, write down what information (or skill) you lacked[241].

[241] If you are just getting started with databases, save this list for the future, and come back to it after a while. You will be pleasantly surprised at how things have changed.

6.2. Database Quality Assurance at the Production Stage

6.2.1. General Approaches to Database Quality Assurance at the Production Stage

While in the previous section experiments{379} were mentioned only as one of the techniques, at the production stage experimental checks and observations (monitoring) become the main source of information for assessing and improving the quality.

Also, since database exploitation is by definition extremely practice-oriented, in this case it is quite difficult to separate the general ideas from the techniques and tools.

But in general, such ideas include the following.

Quality assurance is a continuous process. Of course, we are constantly thinking about quality in the process of designing the database and checking everything particularly carefully before the full launch of the entire system. But this is not the end of our work.

First, most projects are “long-playing”, and therefore the customer will have new ideas, we will have to make changes in the database and repeat many stages of testing.

Second, the situation in which the database operates may change over time. The DBMS and operating system are updated, the network infrastructure changes, the number of users working with the database changes, the amount of stored and processed data increases, and other explicit and implicit changes occur. And all this can affect the database performance (alas, usually negatively).

That’s why there is no “do and forget” option here. We will have to work with the database continuously until the day it is decommissioned.

The earlier the experiments begin, the better. This idea stems from a fundamental principle of testing: the earlier a problem is discovered, the easier, faster, and cheaper it is to fix.

“Exploitation” (in quotes, since it is, for now, internal exploitation) of the database begins with the first experiment of importing the database schema into the DBMS. And nothing prevents us from preparing datasets and special tools and running a series of tests to discover and immediately fix a wide range of problems (which can radically affect the entire design process, among other things).

In addition, the option “let it work, and once it breaks, we will fix it” in most cases is absolutely unacceptable (neither the customer nor the users will want to wait until some critical service is restored to working order).

Experiments should be brought closer to reality. Here we would like to ask a few rhetorical questions to those people who break this rule. Do you really think there will only be five users on your system? And your site will really have all 100 million news items published by the same author at the same time? And all products will have the same price? And your site will never get more than 2 or 3 visitors at a time?

The more the experiment resembles real life, the more useful information it provides. Therefore, the data (both in quantity and substance), load, infrastructure configuration, etc. — everything should be as similar to real-life conditions as possible.

There is one exception: if the operation of the system involves the use of a very large-scale and expensive infrastructure (hundreds of servers in dozens of data centers), such large-scale studies require a separate approach. But even in such a situation, nothing prevents us from using a single server for testing that is 100 % similar to those on which the database will subsequently run.

And there is also nothing to prevent us from preparing test data at some scale (for example, 1 % of the planned volume, if it is measured in petabytes) and conducting experiments with appropriate correction factors.

Observation (monitoring) complements experiments. Sometimes it is enough just to watch events unfold. Modern DBMSes (and their complementary tools) allow us to collect and analyze almost real-time data on thousands of parameters, as well as build predictions and warn of potential problems based on such collected data.

It would be foolish to ignore such features, often available for free (as part of the DBMS license).

In addition, some experiments are extremely difficult to come up with and perform fully, and monitoring can show (in advance) the development of even the most unlikely and unexpected unpleasant situations.

Automation can significantly reduce costs. Just mentioned monitoring is not possible without automation — only specialized tools can collect and analyze such a huge set of data at the required speed.

But even if we are talking about classical experiments, they include many operations, the automation of which is very beneficial:

- preparation of the initial state of the system (possibly with the deployment of virtual machines, etc.);
- test data generation;
- filling the database with test data;
- execution of queries with a given intensity;
- monitoring of database and DBMS operation during the experiment;
- collecting information about erroneous situations;
- summary report generation.

All this and much more can and should be automated. This approach will make it possible to repeat time-consuming experiments many times with a minimum of effort.

And some experiments without automation are impossible in principle (try to manually create information about a billion products, for example).

It's worth keeping the infrastructure in mind. Sometimes the problem (and its solution) may lie outside of the database or DBMS. Sometimes the problem is in the operating system, or in the network infrastructure, or in some related application, or in the hardware.

The above-mentioned monitoring allows us to track even such difficulties.

And experiments allow us to test the behavior of the database in such situations.

Sometimes you may hear objections that the infrastructure is a task of DevOps[242] engineers, and there is no need to try to solve it with databases. This is partly justified, but you as a database developer will not feel better if after (for example) rebooting one of the servers the replication process is not resumed, or other similar problems arise.

Yes, we hope that the infrastructure will work well, but we must also be prepared for those situations when the infrastructure fails us.

If you don't break it, others will. And this applies not only to security issues (although they should not be neglected in any way). It applies to any potential problem, especially when we think, "Well, that's not going to happen".

Databases operate in a very complex hardware and software environment with hundreds and thousands of components. There can be problems with any of these components. Any of these components can be misconfigured. Any part of this complex system can be attacked by hackers.

It is much more advantageous to foresee the relevant problems and verify that our database is resilient to them, than in a critical situation frantically search for a solution and deal with very unpleasant consequences.

[242] **DevOps** — a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development lifecycle and provide continuous delivery with high software quality. ("Wikipedia") [https://en.wikipedia.org/wiki/DevOps]

At the end of this chapter, let's note that DBMS manufacturers themselves pay a lot of attention to the database quality issues at the production stage, and therefore you can always find many articles and videos with specific recommendations for your specific DBMS.



Task 6.2.a: which of the ideas presented in this section have you used when working with the "Bank{395}" database and the filesharing service database{278}, {295}, {305}? Make a list and for each item, write at least three examples of technical solutions resulting from the application of the corresponding idea.

6.2.2. Tools and Techniques of Database Quality Assurance at the Production Stage

Using terminology from the field of software testing, we can say that the emphasis at the database design stage is shifted towards quality assurance, and at the production stage — towards quality control (these and many other terms are explained in detail in the book[243] dedicated to software testing.

In other words — here it is necessary to apply various testing techniques aimed at assessing the compliance of the database and its workflow to the declared quality criteria, as well as to search for existing and potential problems and errors.

And the vast majority of such techniques use experimentation and observation (monitoring) as a basic tool.

In general, database testing can be divided into three conditional subtypes:

- structural testing aimed at checking the entire database schema, individual tables{22} and their fields{20}, views{331}, stored routines{353} etc.;
- functional testing, aimed at checking the database in terms of operations performed with it;
- non-functional testing aimed at verifying performance, security, and similar quality metrics.

In the process of database testing, as a rule, we need to perform several sequential actions:

- Configure the test execution environment. Depending on various conditions, this task may involve clearing the database of existing data, filling it with test data, setting up the environment, and other procedures.
- Perform a test. Here implementation options may vary from manually executing a few queries to running a complex testing mechanism using special automation tools.
- Analyze test results. Ideally, they will meet our expectations, i.e., “everything works as it should”. But much more often we will notice various deviations, errors, problems. It is necessary to determine their causes.
- Generate a test report. This procedure refers not so much to testing databases as to “testing in principle” (and is described in detail in the relevant book[243]).



If, for some reason, testing is to be done on a real working database, it is worth remembering the golden rule: it should never be done. Without exception.

The real working database can only be observed (monitored). All testing should be done only on its copy (ideally, such copy should not be a “copy of data”, but a copy of the entire execution environment — down to the operating system, which is very easily achieved by using virtual machines).

Obviously, the goals, objectives, methods, and tools of testing should be selected in a specific situation, based on the project environment.

But for a better understanding of the basics, consider a few universal ideas that can be a starting point to test…

- Tables and their fields:
  - All tables in the database are present, have the expected names, structure, field types.
  - All tables have correctly defined primary keys (especially — composite ones).

[243] For details, see the “Software Testing. Base Course.” book (Svyatoslav Kulikov) [https://svyatoslav.biz/software_testing_book/]

    - All fields have correctly set `NULL` / `NOT NULL` properties, default values, unique indexes, “autoincrementability” properties.
    - All necessary relationships are present, indexes are built on the foreign keys, and the types of primary and foreign keys fully coincide.
    - All required fields have checks{338} are set.
- Stored routines:
    - All stored routines are present, have the expected names, have the expected input and output parameter sets (including data types).
    - Is it possible for a routine to return more or less data than expected (for example, an empty set of records instead of several records, or several records instead of one, or a `NULL` value instead of a “real” value)?
    - Is it possible to pass parameters to a stored routine that it was not designed for, and what is its response? E.g., a number that is too large, or a string that is too long, or a string of zero length (if the string is always expected to contain characters)?
    - Whether a stored routine generates a correct error message (both the message text itself and the type of the generated exception, etc.)?
- Triggers:
    - All necessary triggers are present in the database, they have the correct names, are created on the correct tables, and respond to the correct events.
    - Is it possible that the behavior of a trigger will be different than expected?
    - Does a trigger generate a correct error message (both the text of the message itself and the type of the generated exception, etc.)?
- Database and DBMS settings:
    - Are all operations on importing the database schema into the DBMS, filling it with test data, configuring both the database and the DBMS performed correctly?
    - Do applications that work with the database interact successfully with it? Do applications involving interaction with the database pass their own tests successfully?
- Performance:
    - Does the database performance meet the expected level under the planned load (considering the actual data set, number of users, nature and intensity of their operations)?
    - Does the DBMS report potential problems with its built-in mechanisms (insufficient RAM or CPU capacity, “long-running” queries, periodic mutual blocking of transactions, and other bottlenecks)?
    - Are there any performance issues with the database applications?
    - How fast are the most frequently run queries (are they the ones that have the most tangible impact on performance)?

Another effective way to understand database testing is to draw parallels between tests with applications and tests with databases. Let’s represent this in the form of a table[244].

[244] The original idea is taken from the course "Database Testing" (there you can find a lot of additional details and useful ideas). [https://www.tutorialspoint.com/database_testing/database_testing_overview.htm]

Application testing	Database testing
This involves validating applications and their components and processes, such as forms, menus, reports, navigation between pages, etc.	This involves checking for user invisible components and processes, such as tables, triggers, and data transfers between the database and the application.
A good understanding of business requirements and the subject area, application functionality, and typical user scenarios is required.	It requires *not only* a good understanding of business requirements and the subject area, application functionality and typical user scenarios, but also a good understanding of principles of database design and operation, SQL language, and the principles of server infrastructure.
Interaction with the application is done manually or using test automation tools.	Interaction with the DBMS is usually performed using automation tools and implies a much deeper study (on different data sets, in different typical and atypical situations in which the database will have to function).

The tools that will be needed at this stage of database quality assurance can also be divided into several categories, each of which will provide examples of specific software tools[245]:

- Data parsing, standardization and generation tools allow both to prepare test data sets in a single unified format and help to solve similar problems of standardization and unification in existing databases (DTM Data Generator[246], Turbo Data[247]).
- Data cleaning and merging tools allow to detect incorrect and duplicate data, as well as to perform the necessary operations to correct the situation (OpenRefine[248], WinPure[249]).
- Profiling tools allow to monitor the processes running in the DBMS and the server environment, to assess their quality by specified parameters and to report on existing or potential problems (Neor Profile SQL[250], SQL Server Profiler[251]).
- Load testing tools allow to evaluate the performance of the database (Benchmark Factory[252], Database Benchmark[253]).
- Low-level (including modular) testing tools allow to automate a wide range of database tests (SQL Test[254], tSQLt[255]).



Since further in-depth study of the topic of this chapter already goes beyond working with databases and moves into a broader area of software testing, we propose to continue diving into the details presented in the “Software Testing. Base Course.[256]” book.

[245] Keep in mind that the software industry is developing very rapidly, so some of the examples on this list can become obsolete in just a few weeks.
[246] “DTM Data Generator” [http://www.sqledit.com/dg/]
[247] “Turbo Data” [http://www.turbodata.ca]
[248] “OpenRefine” [https://openrefine.org]
[249] “WinPure” [https://winpure.com]
[250] “Neor Profile SQL” [http://www.profilesql.com]
[251] “SQL Server Profiler” [https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler]
[252] “Benchmark Factory” [https://www.quest.com/products/benchmark-factory/]
[253] “Database Benchmark” [http://stssoft.com/products/database-benchmark/]
[254] “SQL Test” [https://www.red-gate.com/products/sql-development/sql-test/]
[255] “tSQLt” [https://tsqlt.org]
[256] “Software Testing. Base Course.” (Svyatoslav Kulikov) [https://svyatoslav.biz/software_testing_book/]



Task 6.2.b: apply the techniques presented in this chapter to analyze the "Bank{395}" database and the filesharing service database{278}, {295}, {305}. Make a list of the problems found during this analysis. For each problem, write a proposal to fix it.

6.3. Additional Data Quality and Database Quality Issues

6.3.1. Typical Misconceptions About Data

Since databases (by definition) store and process data, it is important to understand:

- which data presentation formats to use;
- what values the data can take;
- how to process the data correctly;
- what restrictions on data formats, values, and processing rules should (or should not) be implemented.

In this context, it is worth noting that novice database developers are often subject to various misconceptions, many of which at first glance seem perfectly logical.

So, for example, isn't it logical that one person would only have one passport? To some extent yes, but not always (and we have already considered this case{163}).

To illustrate, we will look at common misconceptions about time[257], names[258], phone numbers[259], and addresses[260].

Yes, in many cases, the ideas presented here will be irrelevant to your database. Still, it is worth keeping them in mind (and asking appropriate clarifying questions to customers).

Information about date and time is present in one way or another in almost any database. And the very concepts of "date" and "time" are so densely present in our lives that it would seem that we know everything about them, and everything is always clear.

But the following habitual statements about the date and time are false:

- there are always strictly 24 hours in a day (no, if there was a switch from/to daylight saving time);
- a month always has 30 or 31 days (no, we forgot about February);
- there are always 365 days in a year (no, there are 366 days in a leap year);
- February always has 28 days (no, there are 29 days in a leap year in February);
- any 24-hour period begins and ends on the same day, week, month (no, unless the beginning of the period coincides with the beginning of the day);
- a week always begins and ends in the same month (no, just don't confuse the abstract concept of "month" and the actual calendar);
- week and month always start and end in the same year (no, similar to the previous point);
- the server where the database runs will always be in UTC+0 (Greenwich Mean Time) or in your local time zone (no, you cannot predict the settings on the real server where your database will run;
- the time zone of the server where the database is running will not change (no, similar to the previous point — at any time such settings can be changed);

[257] "Falsehoods programmers believe about time" [https://infiniteundo.com/post/25326999628/falsehoods-programmers-believe-about-time], "More falsehoods programmers believe about time; "wisdom of the crowd" edition" [https://infiniteundo.com/post/25509354022/more-falsehoods-programmers-believe-about-time], "Falsehoods programmers believe about Unix time" [https://alexwlchan.net/2019/falsehoods-programmers-believe-about-unix-time/]

[258] "Falsehoods Programmers Believe About Names" [https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/], "Falsehoods Programmers Believe About Names – With Examples" [https://shinesolutions.com/2018/01/08/falsehoods-programmers-believe-about-names-with-examples/]

[259] "Falsehoods Programmers Believe About Phone Numbers" [https://github.com/google/libphonenumber/blob/master/FALSEHOODS.md]

[260] "Falsehoods programmers believe about addresses" [https://www.mjt.me.uk/posts/falsehoods-programmers-believe-about-addresses/]

- the system clock of the server where the database is running is always accurate (no, similar to the two previous items — the clock on the server may be not synchronized with the exact time source at all, and may also be noticeably "behind" or "ahead" (even incrementally, up to several minutes per day));
- the clock on the server where the database is running and on the client's computer always shows the same time (no, because the server and the client may be in different time zones, and both the server's and the client's clocks may run inaccurately);
- the clock on the server where the database is running never shows the time that belongs to the distant past or the distant future (no, by analogy with the previous items);
- time has no beginning and no end (no, Unixtime, for example, has boundaries);
- one minute on the server where the database runs has exactly the same duration as one minute on any other clock (no, the clock on the server can "rush" or "lag", which will result in different lengths of time on the server and on the reference clock);
- the minimum unit of time is one millisecond (no, sometimes there is a need to store time with higher accuracy);
- the time format will always have the same degree of accuracy (no, it may well happen that we have to compare the date (e.g., 2010-01-02) with the date and time (e.g., 2010-01-02 12:34:55) and with high precision time (e.g., 12:34:55.365245).
- a timestamp of sufficient accuracy can be considered unique (no, nothing in reality prevents two events from occurring simultaneously to the nearest, for example, trillionths of a second);
- the timestamp shows the time when the event actually happened (no, nothing prevents the news of yesterday's event from being published today);
- the human-readable date and time format is always the same (no, for example, the "American" format ("month/day/year") and the "European" format ("day.month.year")
- the time difference between the two time zones will remain constant (no, sometimes some countries decide to change this difference);
- daylight saving time will be changed every year at the same time (no, and sometimes it will not happen at all if some country decides to cancel this change);
- it is easy to count the number of hours and minutes that have elapsed since a certain point in time (no, if there have been daylight saving time transitions, time zone changes, etc. in the past);
- time in Unixtime is the number of seconds since January 01, 1970 (no, this is only true for the UTC+0 time zone, in others you have to consider the time zone);
- you can wait for the clock to show exactly HH:MM:SS, sampled once per second (no, it may well happen that at the right moment in time the operating system will not give control to your program, and you "miss" the right moment);
- the week starts on Monday (no, in many countries it starts on Sunday);
- each minute contains 60 seconds (no, at least due to periodic time adjustments[261]);
- time always goes forward (no, for example, by virtue of adjustments to "rushing" clocks, the time on them may periodically "go backward");
- 24:12:34 is the wrong time (no, the time at airports and railway stations is sometimes indicated in this way);
- time zones always differ by a whole hour (no, some zones have a different offset);
- leap years occur every 4 years (no, it's a little more complicated than that[262]).

[261] "Leap second" ("Wikipedia") [https://en.wikipedia.org/wiki/Leap_second]

[262] "Leap year" ("Wikipedia") [https://en.wikipedia.org/wiki/Leap_year]

Let's go on with the names. Even if the situation is a little simpler with them, all of these statements are also false:

- every person has one canonical full name (no, we have to consider national peculiarities);
- each person has exactly N names, regardless of the meaning of N (no, for example, pseudonyms of creative people);
- names don't change (no, they do — and not only surnames, but also first names — this is legally allowed in most countries);
- names are written in a single encoding (no, we have to take into account national peculiarities);
- names sometimes contain prefixes or suffixes, but we can safely ignore them (no, in a legal context every character is important);
- names do not contain numbers (no, in some countries people can officially put numbers in names);
- first and last names are necessarily different (no, especially in anglophone countries);
- two different systems with the same person's name will use the same name for them (no, since there are no laws of physics that would make these systems synchronize);
- if we see two different first names, they are different people (no, for example, the person changed their last name (and first name));
- if a person did not change the name, their name will be written the same everywhere (no, there are known cases when in the birth certificate, in the passport and in the driver's license the name of the same person was written with misprints — different everywhere);
- a person always has a name (not, for example, a newborn).

Let's move on to telephone numbers. All of the following statements are also false:

- a phone number of a certain type (e.g., mobile) will never change type (no, a range of numbers may be transferred to another operator, and it may well use that range for landlines);
- a phone number uniquely identifies a person (no, it can be a business number answered by different people at different times);
- a person has only one phone number (no, many people have multiple sim cards);
- phone numbers cannot be reused (no, telecom operators give new subscribers the phone numbers of even deceased people);
- each country code corresponds to exactly one country (no, e.g., US and Canada have the same code "+1" and Russia and Kazakhstan have "+7")
- each country corresponds to only one code (no, there are countries with more than one code);
- there are only two ways to indicate a telephone number — in international format and in local format (no, some numbers require different prefixes, depending on where the number is dialed from);
- no prefix of a real telephone number can be a real telephone number (no, in some countries it is possible to reach another subscriber by dialing additional digits after the telephone number: e.g., the number "12345678" may belong to one person, and "123456" to another)
- a telephone number contains only digits (no, in Israel, for example, some advertisement numbers start with "*").

Now let's talk about addresses. As you can easily guess, all of the following statements are also false:

- each building has an address (no, the addressing system is not perfect, and there are plenty of buildings on the planet without an address);
- each building has no more than one address (no, houses at street intersections often have two addresses);
- each building has no more than two addresses (no, there are also three street intersections, and we may need to consider the historical timeline of when street names changed);
- each structure is located in a locality (no, there are many structures in outlying areas that are not in a locality);
- the house number is a number (no, many house numbers have letter prefixes or suffixes like "11a");
- the zero at the beginning of the house number can be ignored (no, there are cities where houses such as "11" and "011" are different houses)
- a city cannot have two streets with the same name (no, it can, e.g., Moscow has two "March 8" streets);
- there are only streets and avenues (no, there are also squares, embankments, etc.);
- it can't be that a house has a block number but no house number (no, it can, even if it's unusual).

This list of false statements is far from complete, but it gives enough indication that designing a database on the principle of "I know exactly how things work and how they don't" is likely to cause your users a lot of problems in the future.

That is why it is necessary to carefully study the subject area and all customer requirements and user needs to design a truly functional, reliable, usable database.



Task 6.3.a: supplement the lists in this chapter with those common misconceptions that you are aware of.



Task 6.3.b: are there any flaws in the "Bank{395}" database and the filesharing service database {278}, {295}, {305} caused by following the misconceptions presented in this section? Make a list of such flaws.

6.3.2. Typical Mistakes When Working with Databases and Ways to Eliminate Them

This (final) section of the book summarizes the most typical and dangerous of the previously mentioned problems and mistakes and provides a few additional recommendations.

Let's begin with the problems already discussed earlier in this book.

Problem	Solution
Database model autoconversion{26}. Automatically converting a logical database model into a physical one usually leads to usability{12}, performance{14}, and data safety{15} violations.	Only thoughtful "manual" creation of the physical model allows us to get a truly high-quality database that meets all the necessary requirements. An ostensible saving of time on automatic model creation will later result in thousands of very serious problems.
Ignoring database usability problems{15}. Usability problems{12} usually lead to more performance problems, since complex queries are harder to optimize.	As soon as we have to write complex queries to perform simple, typical, trivial operations (or simply the number of complex queries begins to increase significantly), it is worth reviewing the database schema in order to optimize it in terms of usability{12}.
Ignoring the implementation of constraints at the database level{212}. The overwhelming majority of catastrophic problems with databases occur precisely because someone once considered this or that fatal situation unlikely or impossible and did not implement a protective mechanism.	If we understand that some constraint is necessary, it should be implemented by means of a database or DBMS, without hoping that someone somewhere will do it for us.
Using a "non-English" language{25}. Despite the fact that the vast majority of database design tools and DBMSes support the possibility of naming objects in many languages, there can always be a situation in which the entire system will stop working due to encoding problems in one of the many software tools.	Always use only English for naming database objects. If you don't speak it, write in "transliteration", but use only the English alphabet anyway.
Using the first solution that works{380}. Stopping the analysis of the problem after the first working solution is found.	Since such a solution may not be the best or may only cover particular cases of the problem, it is always worth searching for several alternative solutions and choosing the best of them.
Violation of a unified style of naming the database objects{25}. On the scale of the entire database schema, the problem of non-compliance with naming standards turns into a disaster: it is very difficult to navigate in such a "jumble", it is very easy to miss a mistake.	A standard for naming database objects should be developed and strictly followed.

Violation of the unity of meaning of the data in the column{94}. When storing or selecting data, the contents of the same column have different meanings in different rows (for example, some `measurement` column stores a person's height in one row, and weight in another).	By definition, the data in a single table column ("relation attribute") must have the same meaning, i.e., be treated equally for all table entries ("relation tuples"). We should avoid writing queries or creating database structures that violate this rule.
Violation of field sequence in composite keys and indexes{105}. Incorrect sequence of fields in composite keys and indexes significantly reduces performance.	That field in the composite key or index, which will often be searched for separately from other fields, should be first.
Misinterpretation of dependencies{170}. The attempt to present (discover) dependencies{170} in relations outside the context of the subject area, i.e., as a kind of universal abstraction that is valid for all cases of its application.	The presence or absence of this or that dependency is entirely tied to the requirements of the subject area, which must be considered when analyzing dependencies and the normalization of relationship schemas.
Lack of attention to encoding{307}. If we rely on the default encoding settings, it is very likely that our database will not work correctly on some of the servers.	It is necessary to explicitly specify all encoding settings wherever technically possible.
Insufficient analysis of the subject area{376}. Incomplete analysis of the subject area leads to a lack of necessary information in the database.	It is necessary to carefully find out (from the customer, from the standards, from communicating with users) what data are needed to fully work with the database, and to create appropriate structures for storing these data.

If we talk about general recommendations, which have also been given a lot, we can form a list — not so much ideas as final advice.

Remember the purpose of using the data. Data is always stored for later efficient use. Therefore, it is important to understand what operations on the data and with what frequency will be performed. This understanding allows us to improve the usability{12} and performance{14} of the database at the design stage.

Approach normalization{207} **and other techniques wisely.** Both insufficient and excessive normalization leads to a series of very unpleasant consequences. The middle ground is hard to find, but if you have doubts, normalize first (performing the reverse process is quicker and easier).

The same applies to any other techniques — they should not become an aim in themselves. They are just tools to help with the task at hand.

Consider the DBMS capabilities. Sometimes some sophisticated solutions can be avoided because there are ready-made mechanisms at the DBMS level to achieve the goals. And even a simple understanding of typical DBMS features makes development very easy, because:

- views{331} prevent denormalization at the table level;
- indexes{103} increase performance;
- checks{338} simplify the formation of specific restrictions;
- stored routines{353} enable the implementation of things at the database level once, which may have to be implemented in several applications;

- transactions management{364} allows getting exactly the behavior of the database that we need;
- triggers{341} allow to automatically perform a whole range of useful (and sometimes necessary) operations.

Take the design seriously. We are supposed to create a database that many people will be working with for a long time — and not just as end users.

Following the naming convention{213}, comprehensive documentation{215}, the use of common design tools{215} and other obvious techniques are easy enough to use and at the same time make a tangible difference in the quality of database design.

Remember about archiving and logging. Complex databases require a special approach to the internal structure — as a rule, most of the data will be needed very rarely and can be moved to special archive tables as they "become obsolete".

Instead of deleting information, we can use so-called "soft deletion" to improve reliability, when the data is only marked as deleted, but physically remains in the database.

And the comprehensive mechanism of logging all critical actions will not only help to find out the cause of a situation, but also to restore the original state of the database if necessary.

Use knowledge from related areas. Although database design in itself is a vast and complex field, requiring a deep understanding (and appropriate specialization), knowledge of SQL, understanding of operating systems and computer networks, experience with several different DBMSes, at least some understanding of security — all this and more will allow you to take a more holistic approach to database design, consider many more different situations, anticipate many problems and protect yourself effectively from various nuisances.

That's it. Have fun designing databases!

Chapter 7: Appendixes

7.1. Database Description for Individual Assignments

For many of the self-study tasks in this book, you will need the database described in this appendix.

This is an extremely simplified educational database of a hypothetical bank (hereinafter — the “Bank” database), which contains only the bare minimum of elements.

This database contains a number of deliberate mistakes, which you are encouraged to discover and correct in the various tasks presented in this book.

The database reflects the following entities and their attributes (see figure A.1.a)

- Account (describes a bank account):
  - id (account identifier);
  - balance (account balance, in monetary units);
  - owner (reference to the owner of the account);
  - is system (an indication that the account does not belong to a person).
- Status (describes the status of an account, e.g., “active”, “blocked”, etc.):
  - id (status identifier);
  - name (status name).
- Payment operational (describes payments made in the current month):
  - id (payment identifier);
  - source account (link to source account);
  - destination account (link to the destination account);
  - date and time (date and time of payment);
  - money (amount of payment).
- Payment archive (describes payments made before the beginning of the current month):
  - id (payment identifier);
  - source account (link to source account);
  - destination account (link to the destination account);
  - date and time (date and time of payment);
  - money (amount of payment).
- Account owner (describes the owner of the account):
  - id (identifier of the owner of the account);
  - name (name of owner of the account).
- Site page (describes the page of the bank’s site):
  - id (page identifier);
  - parent page (link to parent page);
  - name (page name).
- Office (describes a bank office):
  - id (office identifier);
  - city (the city in which the office is located);
  - name (name of the office);
  - sales amount (total profit from the services rendered by the office).

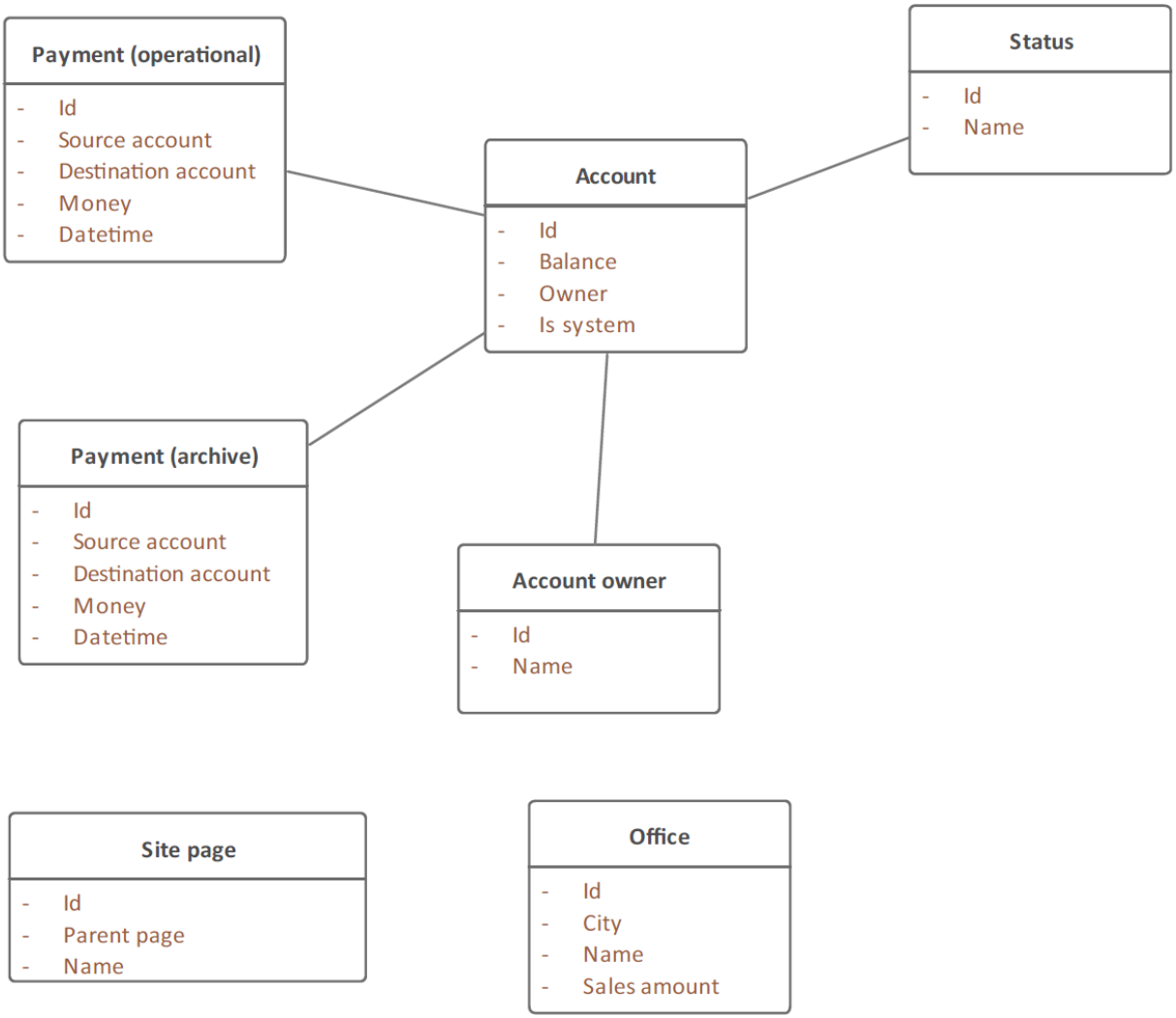


Figure A.1.a — Conceptual (infological) “Bank” database schema

Datalogical schemas in relation to MS SQL Server and Oracle are shown in figures A.1.b and A.1.c, respectively. If you are more comfortable working with the schema for MySQL, you can prepare it yourself on the basis of the handouts[4]. There you can also find the initial project of the “Bank” database in Sparx Enterprise Architect format.

As a description of the physical level of the database, you can consider the code in the following appendix.

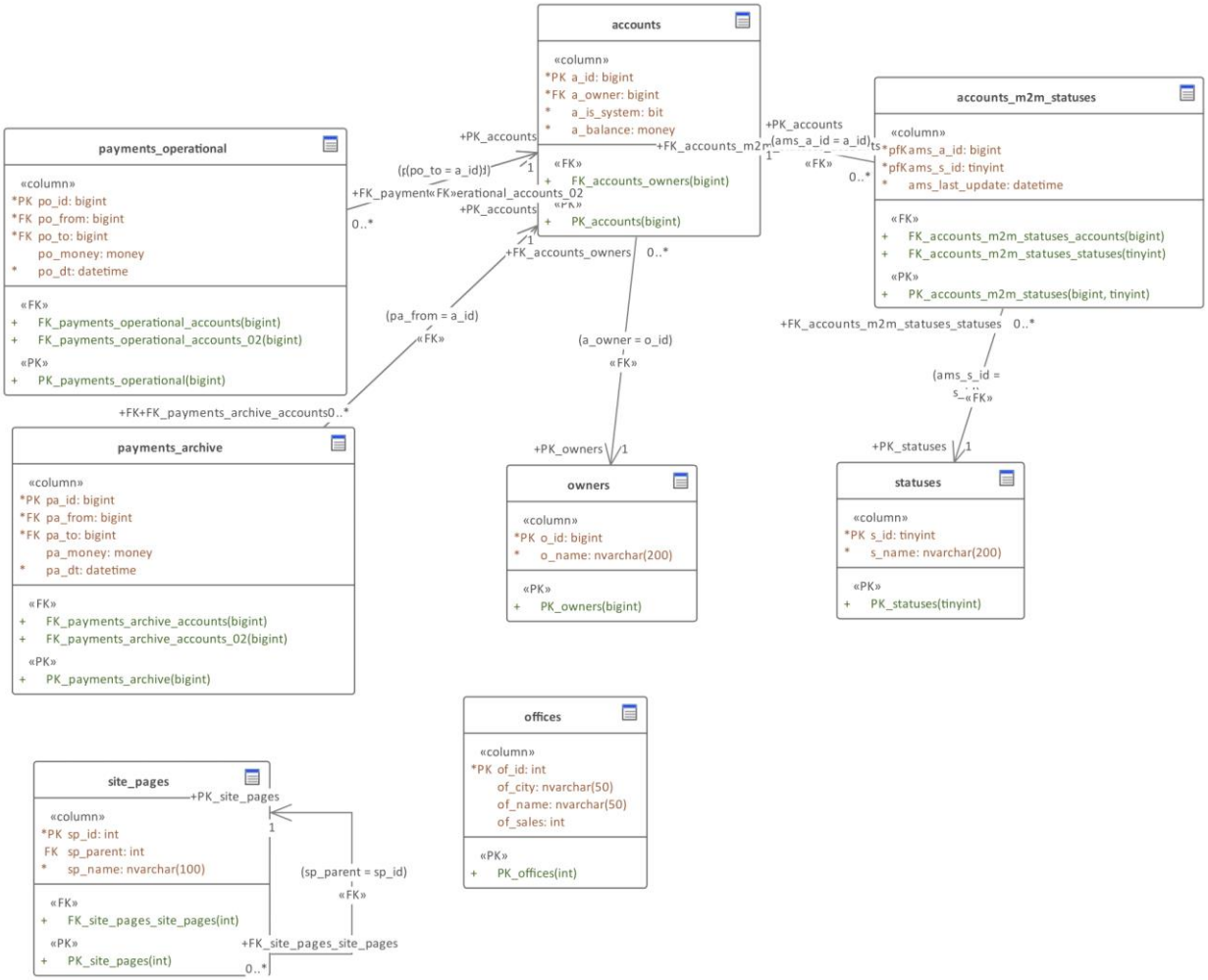


Figure A.1.b — Datalogical (logical) “Bank” database schema for MS SQL Server

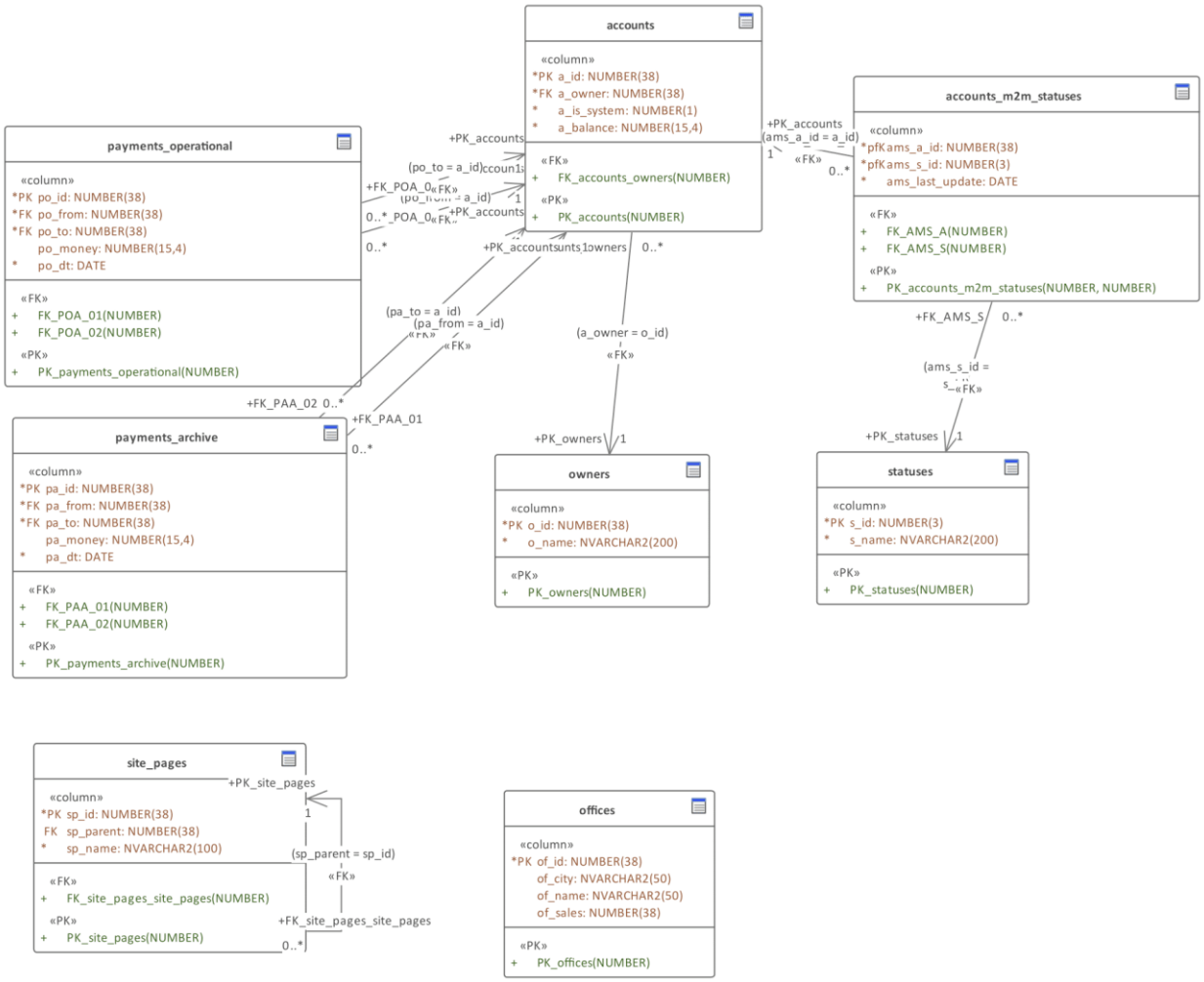


Figure A.1.c — Datalogical (logical) “Bank” database schema for Oracle

7.2. Database Code for Individual Assignments

This SQL code for creating the “Bank” database in this appendix is given “as a last resort” (in case you don’t have the handouts[4] for some reason).

MS SQL Server code:

```
IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_site_pages_site_pages]') AND
OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [site_pages] DROP CONSTRAINT [FK_site_pages_site_pages]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_operational_ac-
counts]') AND OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_operational] DROP CONSTRAINT [FK_payments_operational_accounts]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_operational_ac-
counts_02]') AND OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_operational] DROP CONSTRAINT [FK_payments_operational_accounts_02]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_archive_accounts]')
AND OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_archive] DROP CONSTRAINT [FK_payments_archive_accounts]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_payments_archive_accounts_02]')
AND OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [payments_archive] DROP CONSTRAINT [FK_payments_archive_accounts_02]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_accounts_m2m_statuses_ac-
counts]') AND OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [accounts_m2m_statuses] DROP CONSTRAINT [FK_accounts_m2m_statuses_accounts]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_accounts_m2m_statuses_sta-
tuses]') AND OBJECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [accounts_m2m_statuses] DROP CONSTRAINT [FK_accounts_m2m_statuses_statuses]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[FK_accounts_owners]') AND OB-
JECTPROPERTY(id, 'IsForeignKey') = 1)
ALTER TABLE [accounts] DROP CONSTRAINT [FK_accounts_owners]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[statuses]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [statuses]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[site_pages]') AND OBJECTPROP-
ERTY(id, 'IsUserTable') = 1)
DROP TABLE [site_pages]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[payments_operational]') AND OB-
JECTPROPERTY(id, 'IsUserTable') = 1)
DROP TABLE [payments_operational]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[payments_archive]') AND OB-
JECTPROPERTY(id, 'IsUserTable') = 1)
DROP TABLE [payments_archive]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[owners]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [owners]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[offices]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [offices]
GO
```

```
IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[accounts_m2m_statuses]') AND OB-
JECTPROPERTY(id, 'IsUserTable') = 1)
DROP TABLE [accounts_m2m_statuses]
GO

IF EXISTS (SELECT * FROM dbo.sysobjects WHERE id = object_id('[accounts]') AND OBJECTPROPERTY(id,
'IsUserTable') = 1)
DROP TABLE [accounts]
GO

CREATE TABLE [statuses]
(
	[s_id] tinyint NOT NULL IDENTITY (1, 1),
	[s_name] nvarchar(200) NOT NULL
)
GO

CREATE TABLE [site_pages]
(
	[sp_id] int NOT NULL IDENTITY (1, 1),
	[sp_parent] int NULL,
	[sp_name] nvarchar(100) NOT NULL
)
GO

CREATE TABLE [payments_operational]
(
	[po_id] bigint NOT NULL IDENTITY (1, 1),
	[po_from] bigint NOT NULL,
	[po_to] bigint NOT NULL,
	[po_money] money NULL,
	[po_dt] datetime NOT NULL
)
GO

CREATE TABLE [payments_archive]
(
	[pa_id] bigint NOT NULL,
	[pa_from] bigint NOT NULL,
	[pa_to] bigint NOT NULL,
	[pa_money] money NULL,
	[pa_dt] datetime NOT NULL
)
GO

CREATE TABLE [owners]
(
	[o_id] bigint NOT NULL IDENTITY (1, 1),
	[o_name] nvarchar(200) NOT NULL
)
GO

CREATE TABLE [offices]
(
	[of_id] int NOT NULL IDENTITY (1, 1),
	[of_city] nvarchar(50) NULL,
	[of_name] nvarchar(50) NULL,
	[of_sales] int NULL
)
GO

CREATE TABLE [accounts_m2m_statuses]
(
	[ams_a_id] bigint NOT NULL,
	[ams_s_id] tinyint NOT NULL,
	[ams_last_update] datetime NOT NULL
)
GO

CREATE TABLE [accounts]
(
	[a_id] bigint NOT NULL IDENTITY (1, 1),
	[a_owner] bigint NOT NULL,
	[a_is_system] bit NOT NULL,
	[a_balance] money NOT NULL
)
GO
```

```
ALTER TABLE [statuses]
 ADD CONSTRAINT [PK_statuses]
       PRIMARY KEY CLUSTERED ([s_id])
GO

ALTER TABLE [site_pages]
 ADD CONSTRAINT [PK_site_pages]
       PRIMARY KEY CLUSTERED ([sp_id])
GO

ALTER TABLE [payments_operational]
 ADD CONSTRAINT [PK_payments_operational]
       PRIMARY KEY CLUSTERED ([po_id])
GO

ALTER TABLE [payments_archive]
 ADD CONSTRAINT [PK_payments_archive]
       PRIMARY KEY CLUSTERED ([pa_id])
GO

ALTER TABLE [owners]
 ADD CONSTRAINT [PK_owners]
       PRIMARY KEY CLUSTERED ([o_id])
GO

ALTER TABLE [offices]
 ADD CONSTRAINT [PK_offices]
       PRIMARY KEY CLUSTERED ([of_id])
GO

ALTER TABLE [accounts_m2m_statuses]
 ADD CONSTRAINT [PK_accounts_m2m_statuses]
       PRIMARY KEY CLUSTERED ([ams_a_id],[ams_s_id])
GO

ALTER TABLE [accounts]
 ADD CONSTRAINT [PK_accounts]
       PRIMARY KEY CLUSTERED ([a_id])
GO

ALTER TABLE [site_pages] ADD CONSTRAINT [FK_site_pages_site_pages]
       FOREIGN KEY ([sp_parent]) REFERENCES [site_pages] ([sp_id]) ON DELETE No Action ON UPDATE
No Action
GO

ALTER TABLE [payments_operational] ADD CONSTRAINT [FK_payments_operational_accounts]
       FOREIGN KEY ([po_from]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No
Action
GO

ALTER TABLE [payments_operational] ADD CONSTRAINT [FK_payments_operational_accounts_02]
       FOREIGN KEY ([po_to]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Ac-
tion
GO

ALTER TABLE [payments_archive] ADD CONSTRAINT [FK_payments_archive_accounts]
       FOREIGN KEY ([pa_from]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No
Action
GO

ALTER TABLE [payments_archive] ADD CONSTRAINT [FK_payments_archive_accounts_02]
       FOREIGN KEY ([pa_to]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No Ac-
tion
GO

ALTER TABLE [accounts_m2m_statuses] ADD CONSTRAINT [FK_accounts_m2m_statuses_accounts]
       FOREIGN KEY ([ams_a_id]) REFERENCES [accounts] ([a_id]) ON DELETE No Action ON UPDATE No
Action
GO

ALTER TABLE [accounts_m2m_statuses] ADD CONSTRAINT [FK_accounts_m2m_statuses_statuses]
       FOREIGN KEY ([ams_s_id]) REFERENCES [statuses] ([s_id]) ON DELETE No Action ON UPDATE No
Action
GO

ALTER TABLE [accounts] ADD CONSTRAINT [FK_accounts_owners]
       FOREIGN KEY ([a_owner]) REFERENCES [owners] ([o_id]) ON DELETE No Action ON UPDATE No Ac-
tion
GO
```

Oracle code:

```
DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_statuses_s_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_statuses_s_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_statuses_s_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_statuses_s_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_site_pages_sp_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_site_pages_sp_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_site_pages_sp_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_site_pages_sp_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_payments_operational_po_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_payments_operational_po_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_payments_operational_po_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_payments_operational_po_id"';
END IF;
END
;
```

```
DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_owners_o_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_owners_o_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_owners_o_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_owners_o_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_offices_of_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_offices_of_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_offices_of_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_offices_of_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_TRIGGERS
  WHERE OWNER = ''
  AND TRIGGER_NAME = 'TRG_accounts_a_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP TRIGGER "TRG_accounts_a_id"';
END IF;
END
;

DECLARE
  C NUMBER;
BEGIN
SELECT COUNT(*) INTO C
FROM ALL_SEQUENCES
  WHERE SEQUENCE_OWNER = ''
  AND SEQUENCE_NAME = 'SEQ_accounts_a_id';
  IF (C > 0) THEN
    EXECUTE IMMEDIATE 'DROP SEQUENCE "SEQ_accounts_a_id"';
END IF;
END
;
```

```
DROP TABLE "statuses" CASCADE CONSTRAINTS
;

DROP TABLE "site_pages" CASCADE CONSTRAINTS
;

DROP TABLE "payments_operational" CASCADE CONSTRAINTS
;

DROP TABLE "payments_archive" CASCADE CONSTRAINTS
;

DROP TABLE "owners" CASCADE CONSTRAINTS
;

DROP TABLE "offices" CASCADE CONSTRAINTS
;

DROP TABLE "accounts_m2m_statuses" CASCADE CONSTRAINTS
;

DROP TABLE "accounts" CASCADE CONSTRAINTS
;

CREATE TABLE "statuses"
(
        "s_id" NUMBER(3) NOT NULL,
        "s_name" NVARCHAR2(200) NOT NULL
)
;

CREATE TABLE "site_pages"
(
        "sp_id" NUMBER(38) NOT NULL,
        "sp_parent" NUMBER(38) NULL,
        "sp_name" NVARCHAR2(100) NOT NULL
)
;

CREATE TABLE "payments_operational"
(
        "po_id" NUMBER(38) NOT NULL,
        "po_from" NUMBER(38) NOT NULL,
        "po_to" NUMBER(38) NOT NULL,
        "po_money" NUMBER(15,4) NULL,
        "po_dt" DATE NOT NULL
)
;

CREATE TABLE "payments_archive"
(
        "pa_id" NUMBER(38) NOT NULL,
        "pa_from" NUMBER(38) NOT NULL,
        "pa_to" NUMBER(38) NOT NULL,
        "pa_money" NUMBER(15,4) NULL,
        "pa_dt" DATE NOT NULL
)
;

CREATE TABLE "owners"
(
        "o_id" NUMBER(38) NOT NULL,
        "o_name" NVARCHAR2(200) NOT NULL
)
;

CREATE TABLE "offices"
(
        "of_id" NUMBER(38) NOT NULL,
        "of_city" NVARCHAR2(50) NULL,
        "of_name" NVARCHAR2(50) NULL,
        "of_sales" NUMBER(38) NULL
)
;
```

```
CREATE TABLE "accounts_m2m_statuses"
(
        "ams_a_id" NUMBER(38) NOT NULL,
        "ams_s_id" NUMBER(3) NOT NULL,
        "ams_last_update" DATE NOT NULL
)
;

CREATE TABLE "accounts"
(
        "a_id" NUMBER(38) NOT NULL,
        "a_owner" NUMBER(38) NOT NULL,
        "a_is_system" NUMBER(1) NOT NULL,
        "a_balance" NUMBER(15,4) NOT NULL
)
;

CREATE SEQUENCE "SEQ_statuses_s_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_statuses_s_id"
        BEFORE INSERT
        ON "statuses"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_statuses_s_id".NEXTVAL
                INTO :NEW."s_id"
                FROM DUAL;
        END;
;
/


CREATE SEQUENCE "SEQ_site_pages_sp_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_site_pages_sp_id"
        BEFORE INSERT
        ON "site_pages"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_site_pages_sp_id".NEXTVAL
                INTO :NEW."sp_id"
                FROM DUAL;
        END;
;
/


CREATE SEQUENCE "SEQ_payments_operational_po_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;
```

```
CREATE OR REPLACE TRIGGER "TRG_payments_operational_po_id"
        BEFORE INSERT
        ON "payments_operational"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_payments_operational_po_id".NEXTVAL
                INTO :NEW."po_id"
                FROM DUAL;
        END;
;
/

CREATE SEQUENCE "SEQ_owners_o_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_owners_o_id"
        BEFORE INSERT
        ON "owners"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_owners_o_id".NEXTVAL
                INTO :NEW."o_id"
                FROM DUAL;
        END;
;
/


CREATE SEQUENCE "SEQ_offices_of_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_offices_of_id"
        BEFORE INSERT
        ON "offices"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_offices_of_id".NEXTVAL
                INTO :NEW."of_id"
                FROM DUAL;
        END;
;
/

CREATE SEQUENCE "SEQ_accounts_a_id"
        INCREMENT BY 1
        START WITH 1
        NOMAXVALUE
        MINVALUE  1
        NOCYCLE
        NOCACHE
        NOORDER
;

CREATE OR REPLACE TRIGGER "TRG_accounts_a_id"
        BEFORE INSERT
        ON "accounts"
        FOR EACH ROW
        BEGIN
                SELECT "SEQ_accounts_a_id".NEXTVAL
                INTO :NEW."a_id"
                FROM DUAL;
        END;
;
/
```

```
ALTER TABLE "statuses"
 ADD CONSTRAINT "PK_statuses"
       PRIMARY KEY ("s_id") USING INDEX
;

ALTER TABLE "site_pages"
 ADD CONSTRAINT "PK_site_pages"
       PRIMARY KEY ("sp_id") USING INDEX
;

ALTER TABLE "payments_operational"
 ADD CONSTRAINT "PK_payments_operational"
       PRIMARY KEY ("po_id") USING INDEX
;

ALTER TABLE "payments_archive"
 ADD CONSTRAINT "PK_payments_archive"
       PRIMARY KEY ("pa_id") USING INDEX
;

ALTER TABLE "owners"
 ADD CONSTRAINT "PK_owners"
       PRIMARY KEY ("o_id") USING INDEX
;

ALTER TABLE "offices"
 ADD CONSTRAINT "PK_offices"
       PRIMARY KEY ("of_id") USING INDEX
;

ALTER TABLE "accounts_m2m_statuses"
 ADD CONSTRAINT "PK_accounts_m2m_statuses"
       PRIMARY KEY ("ams_a_id","ams_s_id") USING INDEX
;

ALTER TABLE "accounts"
 ADD CONSTRAINT "PK_accounts"
       PRIMARY KEY ("a_id") USING INDEX
;

ALTER TABLE "site_pages"
 ADD CONSTRAINT "FK_site_pages_site_pages"
       FOREIGN KEY ("sp_parent") REFERENCES "site_pages" ("sp_id")
;

ALTER TABLE "payments_operational"
 ADD CONSTRAINT "FK_POA_01"
       FOREIGN KEY ("po_from") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "payments_operational"
 ADD CONSTRAINT "FK_POA_02"
       FOREIGN KEY ("po_to") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "payments_archive"
 ADD CONSTRAINT "FK_PAA_01"
       FOREIGN KEY ("pa_from") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "payments_archive"
 ADD CONSTRAINT "FK_PAA_02"
       FOREIGN KEY ("pa_to") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "accounts_m2m_statuses"
 ADD CONSTRAINT "FK_AMS_A"
       FOREIGN KEY ("ams_a_id") REFERENCES "accounts" ("a_id")
;

ALTER TABLE "accounts_m2m_statuses"
 ADD CONSTRAINT "FK_AMS_S"
       FOREIGN KEY ("ams_s_id") REFERENCES "statuses" ("s_id")
;

ALTER TABLE "accounts"
 ADD CONSTRAINT "FK_accounts_owners"
       FOREIGN KEY ("a_owner") REFERENCES "owners" ("o_id")
;
```

Chapter 8: License and Distribution

This book is distributed under the “Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International[263]” 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/relational_databases_book_download/.

Source materials (schemas, scripts, etc.) are available here: https://svyatoslav.biz/relational_databases_book_download/src_rdb.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:



“Using MySQL, MS SQL Server, and Oracle by examples”

In this book: 3 DBMS, 50+ examples, 130+ tasks, 500+ queries with explanations and comments. From SELECT * to finding the shortest path in an or-graph; no theory, just diagrams and code, lots of code. It will be useful for those who: once studied SQL, but have forgotten a lot; has experience with one dialect of SQL, but wants to quickly switch to another; wants to learn to write typical SQL queries in a very short time.

Download: https://svyatoslav.biz/database_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 150 academic hours, of which about half of your time should be spent on practical tasks.

With Russian voiceover: https://svyatoslav.biz/urls/rdb_online_rus/
With English voiceover: https://svyatoslav.biz/urls/rdb_online_eng/

263 “Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International”. [https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode]

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