Databases in 137 Pages

Jyrki Nummenmaa and Aarne Ranta

Faculty of Information Technology and Communication Sciences,
Tampere University

Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg

Draft Version for the Chalmers/GU course in Spring 2021

Please don't distribute outside the course

Contents

1 Introduction **7**
- 1.1 Data vs. systems . . . 8
- 1.2 A short history of databases . . . 8
- 1.3 SQL . . . 9
- 1.4 DBMS . . . 10
- 1.5 The book contents . . . 10
- 1.6 The big picture . . . 13

2 Tables and SQL **15**
- 2.1 SQL database table . . . 15
- 2.2 SQL in a database management system . . . 16
- 2.3 Creating an SQL table . . . 17
- 2.4 Grammar rules for SQL . . . 18
- 2.5 A further analysis of CREATE TABLE statements . . . 19
- 2.6 Inserting rows to a table . . . 21
- 2.7 Deleting and updating . . . 23
- 2.8 Querying: selecting columns and rows from a table . . . 23
- 2.9 Conditions in WHERE clauses . . . 25
- 2.10 Expressions in SELECT fields . . . 27
- 2.11 Set operations on queries . . . 28
- 2.12 Set operations in conditions . . . 30
- 2.13 Sorting results . . . 31
- 2.14 Aggregation and grouping . . . 31
- 2.15 Using data from several tables . . . 34
- 2.16 Foreign keys and references . . . 37
- 2.17 Database diagrams and mutual references . . . 38
- 2.18 Join operations . . . 39
- 2.19 Local definitions and views . . . 40
- 2.20 Dimensional analysis from relational data* . . . 42
- 2.21 SQL as a basis for queries in natural language* . . . 42
- 2.22 SQL pitfalls . . . 43

3 Conceptual modeling using Entity-Relationship diagrams **47**
- 3.1 Introduction . . . 47
- 3.2 Entities and relationships: a complete example . . . 48
  - 3.2.1 Relationship participation types . . . 49
  - 3.2.2 Composite and multivalued attributes . . . 50
- 3.3 E-R syntax . . . 51
  - 3.3.1 Composite attributes . . . 52
- 3.4 From description to E-R . . . 53
- 3.5 Converting E-R diagrams to database schemas . . . 53
- 3.6 A word on keys . . . 55

4 Relational data modelling **57**
4.1 Relations and tables . . . 57
4.2 Functional dependencies . . . 59
4.3 Keys and superkeys . . . 61
4.4 Modelling SQL key and uniqueness constraints . . . 62
4.5 Referential constraints . . . 62
4.6 Operations on relations . . . 63
4.7 Multiple tables and joins . . . 64
4.8 Transitive closure . . . 66
4.9 General constraints on data . . . 67
4.10 Multiple values . . . 68
4.11 Null values . . . 68

5 Dependencies and database design **69**
5.1 Design intuition . . . 69
5.2 Normal forms, synthesis and decomposition . . . 73
5.3 Database design workflow . . . 77
5.4 Acyclicity and functional dependencies on database level* . . . 78
5.5 Inclusion dependencies* . . . 78
5.6 Multivalued dependencies and the fourth normal form . . . 79
5.7 An example with both multivalued and functional dependencies . 81

6 SQL query processing with relational algebra **83**
6.1 The compiler pipeline . . . 83
6.2 Relational algebra . . . 83
6.3 Variants of algebraic notation . . . 85
6.4 From SQL to relational algebra . . . 85
6.4.1 Basic queries . . . 85
6.4.2 Grouping and aggregation . . . 86
6.4.3 Sorting and duplicate removal . . . 88
6.5 Query optimization . . . 88
6.5.1 Algebraic laws . . . 88
6.5.2 Example: pushing conditions in cartesian products . . . 89
6.6 Indexes . . . 89
6.7 Physical computing environment . . . 91

7 Database system reliability **92**
7.1 Transactions and the ACID properties . . . 92
7.2 Atomicity and Durability of database transactions . . . 95
7.3 Isolation of transactions . . . 95
7.3.1 Locking . . . 97
7.3.2 Timestamping . . . 98
7.3.3 Interferences and isolation levels . . . 100
7.4 Consistency . . . 102
7.4.1 Active element hierarchy . . . 102
7.4.2 Referential constraints and policies . . . 103

7.4.3 CHECK constraints . . . 104
7.4.4 Triggers: a first example . . . 105
7.4.5 The syntax of triggers . . . 106
7.4.6 A more complex trigger example . . . 107
7.5 Authorization and grant diagrams . . . 108

8 SQL in software applications 111
8.1 Embedding SQL to Java using a minimal JDBC program . . . 111
8.2 Building queries and updates from input data . . . 114
8.3 Building Java software with database access . . . 115
8.4 Connecting to a SQL database from Haskell using HDBC . . . 117
8.5 Connecting to a SQL database from Python using Python Database API . . . 118
8.6 SQL injection and prepared statements . . . 119
8.7 Three-tier architecture and connection pooling* . . . 120

9 Document databases 121
9.1 JSON . . . 121
9.2 Querying JSON . . . 122
9.3 MongoDB* . . . 123
9.4 XML and its data model . . . 124
9.5 The XPath query language . . . 127
9.6 YAML* . . . 128

10 Big Data 129
10.1 Partitioning the data . . . 129
10.2 MapReduce . . . 131
10.3 The Cassandra DBMS and its query language CQL . . . 132
10.4 Further considerations . . . 134

A Appendix: SQL in a nutshell 136

Note on the the current edition

These notes were expanded from Aarne's 2017 version in 2020–21 by Jyrki, who has also reorganized the chapters. After that, some material that is not relevant for the course at Chalmers and GU was left out. This is in particular the case with exercises, which are available through the course web page.

We are grateful to Ana Bove for comments on the earlier version.

Tampere and Gothenburg, January 21, 2021

Jyrki Nummenmaa and Aarne Ranta

Preface

This book is based on the idea of specific learning outcomes, which are at the core of utilizing database systems when building applications. As such, the book is both useful to a practitioner outside of the academic world, while at the same time giving the basis for an academic first course on databases. The specific learning outcomes targeted in the book are the following skills:

- Manipulation of a SQL database, including data definition to create the tables, as well as to update and query them.
- Conceptual design of the data and database content for an application.
- Systematic formal definition of the properties of the database data.
- Optimization of the database design using dependencies.
- Understanding the principles on what happens when a query is processed in a database system, thereby helping to write better queries and to optimize certain aspects of how the data is stored.
- Implementing a database application.
- Knowledge on new data models used on database systems, such as document databases and so called Big Data.

This book originates from the courses given by the authors. More specifically, the Databases course (TDA357/DIT620) at the IT Faculty of University of Gothenburg and Chalmers, and the courses Introduction to Databases and Database programming given at the University of Tampere.

The text is intended to support the reader's intuition while at the same time giving sufficiently precise information so that the reader can apply the methods given in the book for different applications. The book proceeds from experimentation and intuition to more formal and precise treatment of the topics, thereby hopefully making learning easier. In particular, Chapters 2, 3 and 4 are suitable as study material for people who in practice have no background in computing studies.

A particular virtue of the book is its size. The book is not meant to be a handbook or manual, as such information is better offered in the Internet, but a concise, intuitive, and precise introduction to databases.

The real value from the book only comes with practice. To build a proper understanding, you should build and use your own database on a computer. You should also work on some theoretical problems by pencil and paper.

By far the best way to study the book is to participate in a course with a teacher - this gives regularity to the study, as it is easy to try to consume too much too quickly when self-studying. For self-study purposes each chapter includes an estimate on how fast the student who has no initial knowledge should progress. The human mind needs time to arrange the new materials and therefore a reasonable steady pace is preferable even though the student would have time available to study the book all day long.

We will use a running example that deals with geographical data: countries and their capitals, neighbours, currencies, and so on. This is a bit different from many other slides, books, and articles. In them, you can find examples such as course descriptions, employer records, and movie databases. Such examples may feel more difficult since they are not completely common knowledge. Using familiar common-sense examples means that when learning new mathematical and programming concepts, you do not need to learn new world knowledge at the same time.

We find it easier to study new technical material if the contents are familiar. For instance, it is easier to test a query that is supposed to assign "Paris" to "the capital of France" than a query that is supposed to assign "60,000" to "the salary of John Johnson" - even though of course most people are familiar with work and salaries, but a particular sum may seem strange, etc. There is simply one thing less to keep in mind. It also eliminates the need to show example tables all the time, because we can simply refer to "the table containing all European countries and their capitals", which most readers will have clear enough in their minds. No geographical knowledge is required, and it is not important at all if the reader can position the countries and cities on the map.

Of course, we will have the occasion to show other kinds of databases as well. The country database does not have all the characteristics that a database might have, for instance, very rapid changes in the data. The exercises and suggested programming assignments will include such material.

In addition to the authors' own database exploration and study, the book has drawn inspiration from various sources, even if it seems to be quite an original compilation of content. A lot of inspiration obviously comes from other database textbooks, such as Garcia-Molina, Ullman, and Widom, *Database systems: The Complete Book*), and by earlier course material at Chalmers by Niklas Broberg and Graham Kemp. We are grateful to (list needs to be expanded) Grégoire Détrez on general advice and comments on the contents, and to Simon Smith, Adam Ingmansson, and Viktor Blomqvist for comments during the course. More comments, corrections, and suggestions are therefore most welcome - your name will be added here if you don't object!

Gothenburg, February 2019

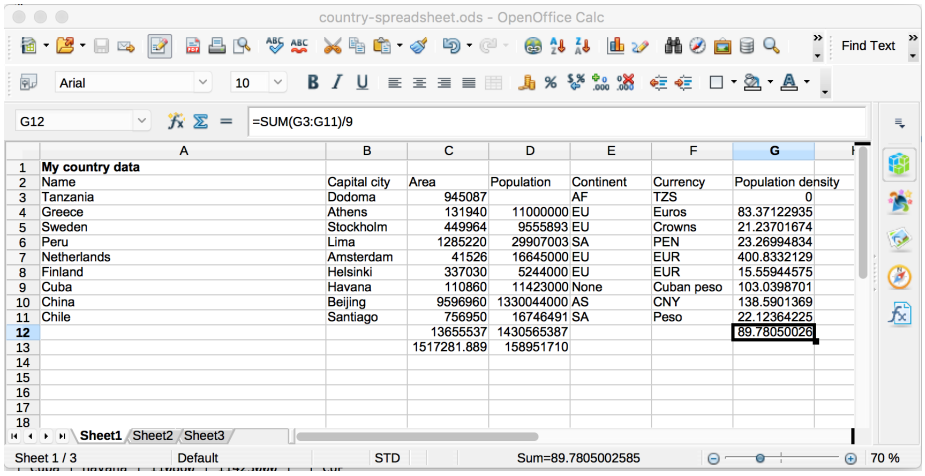
Jyrki Nummenmaa and Aarne Ranta

1 Introduction

Spreadsheets are a common form to organize the data as tables, for calculations, comparisons, etc. Figure 1 shows a spreadsheet table that has information about various countries. A quick look already reveals that most of the rows contain information of a single country: the country name, capital, area, population, continent where the country resides, currency of the country, and population density, obviosly computed from population and area. The first row contains a title row for the table, while the second row contains names for columns, explaining their contents. Then follow the rows containing information for each country, and finally a row with some some summary information.

Are spreadsheets databases? If not, how are they different? Why aren't the spreadsheets enough? Who needs databases on top of them? These are common questions by spreadsheet users when they hear about databases the first time. We give a short explanation here, but the more in-depth information on databases in this book should answer these questions properly.

In short, the existence of databases is justified by the needs for correctness and reliability, combined with a high volume of data and users. Correctness comes in different forms. Consider, for instance, the data types in each column. In column F, the currency information comes in different and inconsistent formats, some as codes and others as full currency name.



	A	B	C	D	E	F	G
1	**My country data**						
2	Name	Capital city	Area	Population	Continent	Currency	Population density
3	Tanzania	Dodoma	945087		AF	TZS	0
4	Greece	Athens	131940	11000000	EU	Euros	83.37122935
5	Sweden	Stockholm	449964	9555893	EU	Crowns	21.23701674
6	Peru	Lima	1285220	29907003	SA	PEN	23.26994834
7	Netherlands	Amsterdam	41526	16645000	EU	EUR	400.8332129
8	Finland	Helsinki	337030	5244000	EU	EUR	15.55944575
9	Cuba	Havana	110860	11423000	None	Cuban peso	103.0398701
10	China	Beijing	9596960	1330044000	AS	CNY	138.5901369
11	Chile	Santiago	756950	16746491	SA	Peso	22.12364225
12			13655537	1430565387			89.78050026
13			1517281.889	158951710			
14							
15							
16							
17							
18							

Figure 1: An example spreadsheet

The spreadsheet has no population information for Tanzania, and it is in fact natural that some values are not known. In spreadsheets the standard summing of values treats the empty cell as zero. This has knock-on effects: the population density of Tanzania also becomes zero, and the average of population densities is also affected, which may be hard to detect at first glance. Missing values is a complicated issue, but databases have at least some systematic and consistent

way to treat the missing values.

In spreadsheets people do computations that refer to other cells. When you copy-paste parts of your sheet, you sometimes get the desired outcome and sometimes not. The spreadsheet refers to cells using row numbers and column ids, like population of China being at D10, instead of refering to it somehow as the value of *population column of the row where China is the country name.*

The more data there is, the more complicated it is to work with row numbers and column ids, and there can be millions or billions of rows and thousands of columns data. When the data is accessed by computer programs, their maintenance gets complicated with row number and column id references.

Databases have to support large amounts of concurrent users, without messing up the users view to the data nor the content of the data. While having the spreadsheet in a cloud service does allow you to share the spreadsheet for concurrent use, the concurrency mechanisms are unlikely to handle succesfully millions of changes per second in the spreadsheet.

The databases are expected to store the data in a reliable and correct way. Banks, for instance, have to store the data about bank accounts so that no penny is lost and, in addition, they need records of the banking events for e.g. book-keeping purposes. Companies need to store and protect the data about their products, customers, selling and purchasing events, etc.

1.1 Data vs. systems

Just like a spreadsheet is accessed with some program, like OpenOffice or Excel or Numbers, the data in the database is also accessed through software that is called a **database management system**.

We say that a database is **any collection of organized data that can be accessed and processed by via a database management system**. A database management system must support both **updates** (i.e. changes in the data) and **queries** (i.e. questions about the data). The data must be **structured** so that these operations can be performed efficiently and accurately. Typically, database management systems support multiple concurrent users and controlled data access.

It is typical that this data lives much longer than the programs that process it: decades rather than just years. Programs, even programming languages, may be changed every five years or so, while the data has permanent value for the organizations. On the other hand, while data is maintained for decades, it may also be changed very rapidly. For instance, a bank can have millions of transactions daily, coming from ATM's, internet purchases, etc. This means that account balances must be continuously updated. At the same time, the history of transactions may be kept for years, for e.g. legal reasons.

1.2 A short history of databases

When databases came to wide use, for instance in banks in the 1960's, they were not yet standardized. They could be vendor specific, domain specific, or even

machine specific. It was difficult to exchange data and maintain it when for instance computers were replaced. Finding information happened through complicated programs, and combining different pieces of information from the database was cumbersome. As a response to this situation, **relational databases** were invented in around 1970. They turned out to be both structured and generic enough for most purposes. They have a mathematical theory that is both precise and simple. Thus they are easy enough to understand by users and easy enough to implement in different applications. As a result, relational databases are often the most stable and reliable parts of information systems. They can also be the most precious ones, since they contain the results from decades of work by thousands of people.

Despite their success, relational databases have recently been challenged by other approaches. Some of the challengers want to support more complex data than relations. For instance, XML (Extended Markup Language) supports **hierarchical data**, and hierarchical databases were popular in the 1960's but were deemed too complicated by the proponents of relational databases. Recently, **Big Data** applications have called for simpler models. In many applications, such as social media, accuracy and reliability are not so important as for instance in bank applications. On the other hand, performance is much more important, and then the traditional relational models can be too rich. Non-relational approaches are known as **NoSQL**, also interpreted as **Not Only SQL**, by reference to the SQL language introduced in the next section.

1.3 SQL

Relational databases are also known as **SQL databases**. SQL is a computer language designed in the early 1970's, originally called Structured Query Language. The full name is seldom used: one says rather "sequel" or "es queue el". SQL is a **special purpose language**. Its purpose is to process of relational databases. This includes several operations:

- **queries**, asking questions, e.g. "what are the neighbouring countries of France"
- **updates**, changing entries, e.g. "change the currency of Estonia from Crown to Euro"
- **inserts**, adding entries, e.g. South Sudan with all the data attached to it
- **removals**, taking away entries, e.g. German Democratic Republic when it ceased to exist
- **definitions**, creating space for new kinds of data, e.g. for the main domain names in URL's

These notes will cover all these operations and also some others. SQL is designed to make it easy to perform them - easier than a **general purpose programming language**, such as Java or C. The idea is that SQL should be easier to learn as well, so that it is accessible for instance to bank employees without computer science training. And finally, SQL is meant to be safer in certain aspects, such as termination of the programs. However, as we will see, most end users of databases today don't even need SQL. They use some **end**

user programs, for intance an ATM interface with menus, which are simpler and less powerful than full SQL. These end user programs are written by programmers as combinations of SQL and general purpose languages.

Now, since a general purpose language could perform all operations that SQL can, isn't SQL superfluous? No, since SQL is a useful intermediate layer between user interaction and the data. One reason is the high level of abstraction in SQL.

1.4 DBMS

The implementations of SQL are called SQL **database management systems** (DBMS). Here are some popular systems, in an alphabetical order:

- IBM DB2, proprietary
- Microsoft SQL Server, proprietary
- MySQL, open source, supported by Oracle
- MariaDB, open source, successor of MySQL,
- Oracle, proprietary
- PostgreSQL, open source
- SQLite, open source
- Teradata, proprietary, designed for large amounts of data and database analytics.

The DBMS implementations are highly optimized and reliable. A general purpose programmer would have a hard time matching their performance and the wide selection of services available, and, most importantly, the reliability that the database systems offer. Losing or destroying data would be a serious risk.

Each DBMS has a slightly different dialect of SQL. There is also an official standard, but no existing system implements all of it, or only it. In these notes, we will most of the time try to keep to the parts of SQL that belong to the standard and are implemented by at least most of the systems.

However, since we also have to do some practical work, we have to choose a DBMS to work in. The choice assumed in this book is PostgreSQL. Its main advantages are:

- It follows the standard more closely than many other systems.
- It is free and open source, hence easier to get hold of.

1.5 The book contents

Chapter 1: Introduction

This is the chapter you are reading now. The goal of this chapter is to make it clear what you are expected to learn and to do to study this book.

Chapter 2: Tables and SQL

We start by getting our hands dirty with SQL, which is based on the simple concept of a table. This will give us the intuition on how data is stored and

retrieved from SQL databases. You will learn the main language constructs of SQL. They enable you to

- Define SQL tables
- Insert and modify the data in the tables
- Query the tables in various ways: selecting data based on the values, select desired columns of the tables, combine data from different tables, use set-theoretical operations on tables, and group and aggregate the data
- Utilize some widely used low-level manipulations of strings and other SQL datatypes.

Chapter 3: Conceptual modelling using Entity-Relationship diagrams

A popular device in modelling is **E-R diagrams** (Entity-Relationship diagrams). You will learn how different kinds of data are modelled by E-R diagrams. We will show how E-R diagrams can be constructed from descriptive texts by following grammatical clues. Finally, you will learn how E-R diagrams are converted to relational schemas (and thereby eventually to SQL).

Chapter 4: Relational data modelling

This chapter is about the mathematical concepts that underlie relational databases. Not all data is "naturally" relational, so that some encoding is necessary to make it relational. Many things can go wrong in the encoding, and lead to redundancy or even to unintended data loss.

To do things systematically right, we need a good formalism for being specific about what we are doing. For this, we will use the basics of the mathematics for relations, based on set theory, and for this purpose we review the basics necessary for us. This chapter is here for you to learn how to formalize the properties of your data precisely, and to map the exact data definition into SQL.

Chapter 5: Dependencies and database design

This chapter builds on the relational model and explains a technique that helps to design databases that avoid certain problematic situations. Mathematically, a relation can relate an object with many other objects. For instance, a country can have many neighbours. A function, on the other hand, relates each object with just one object. For instance, a country has just one number giving its area in square kilometres (at a given time). In this perspective, relations are more general than functions.

However, it is important to acknowledge that some relations *are* functions. Otherwise, there is a risk of **redundancy**, repetition of the information. Redundancy can lead to **inconsistency**, if the information that should be the same in different places is actually not the same. Inconsistency and redundancy are examples of problems with database design. Redundancy can also lead to problems in answering queries and inconsistent data is even worse.

With this chapter, you will learn how to use dependencies to design databases avoiding these problems.

Chapter 6: Query processing with relational algebra

The relational model is not only used when designing databases: it is also the "machine language" used when executing queries. Relational algebra is a mathematical query language. It is much simpler than SQL, as it has only a few operations, often denoted by Greek letters to make the presentation more succinct. Being so simple, it would actually serve as a good basis for making queries in practice.

SQL is the standard language to query databases, but relational algebra is useful as an intermediate language in DBMS implementations. SQL queries can be first translated to relational algebra, which is further optimized before it is executed.

A lot of the query efficiency depends on how the data is organized in the database. A crucial stage in query processing is to find the data with desired values. We can speed up the search by adding indices, search structures to find the data more efficiently. Index definition can be done using SQL. Other factors of query processing efficiency are related to physical resources for computation. They are discussed briefly.

The objective of this chapter is that you can learn the basics about this translation and based on this, learn to improve the efficiency of query processing.

Chapter 7: Database system reliability

We also take a deeper look at inserts, updates, and deletions, in the presence of constraints. The integrity constraints of the database may restrict these actions or even prohibit them. An important problem is that when one piece of data is changed, some others may need to be changed as well. For instance, when value is deleted or updated, how should this affect other rows that use that value to link some data together? Some of these things can be guaranteed by constraints in basic SQL. But some things need more expressive power. For example, when making a bank transfer, money should not only be taken from one account, but the same amount must be added to the other account. For situations like this, DBMSs support **triggers**, which are programs that can do many SQL actions at once.

Chapter 8: SQL in software applications

End user programs are often built by combining SQL and a general purpose programming language. This is called **embedding**, and the general purpose language is called a **host language**. In this chapter, we will look at how SQL is embedded to a procedural language (Java) and a functional language (Haskell), as well as the increasingly popular "multi-paradigm" programming language Python.

We will also cover some pitfalls in embedding. The basic problem in the embedding is that the datatypes in the database and in the programming language do not match. Another is that careless programming may create security holes and open up the system to things like **SQL injection**, where an end user can include harmful SQL code in the queries and data manipulations. In one famous example, the name of a student includes a piece of SQL code that deletes all data from a student database.

Chapter 9: Document databases

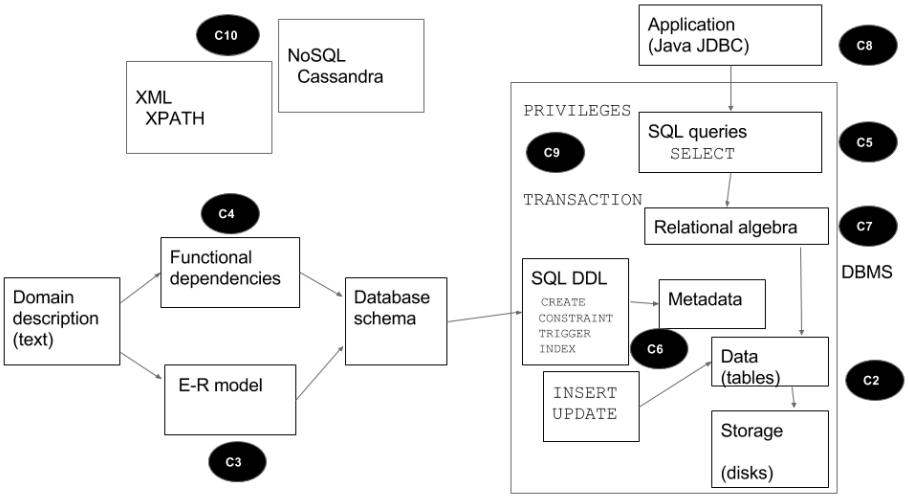
The relational data model has been dominating the database world for a long time, and it is still the appropriate model for many needs. There are, however, data that are not that naturally encoded as tables. In particular documents with hierarchical heterogeneous structure have been a tough case for relational databases. These documents are modelled typically as key-value pairs, using JSON (JavaScript Object Notation) or similar language. As the values may hierarchically consist of lists of key-value pairs, the notion is general and flexible. Another apporach is XML (eXtensible Markup Language). The hierarchical document structure is modelled through markup tags. Web page language HTML is an example of XML use. This chapter introduces JSON and XML, and query methods for them.

Chapter 10: Big Data

There are models simpler than SQL, known as "NoSQL" models. These models are popular in so-called big data applications, since they support the distribution of data on many computers. NoSQL is implemented in systems like Cassandra, originally developed by Facebook and now also used for instance by Spotify.

1.6 The big picture

TODO: update the chapter numbers (black ovals) in the picture.


C10
NoSQL
Cassandra
XML
XPATH
C4
Functional
dependencies
Domain
description
(text)
Database
schema
E-R model
C3
Application
(Java JDBC)
C8
PRIVILEGES
C9
SQL queries
SELECT
C5
TRANSACTION
Relational algebra
C7
DBMS
SQL DDL
CREATE
CONSTRAINT
TRIGGER
INDEX
Metadata
C6
Data
(tables)
C2
INSERT
UPDATE
Storage
(disks)

2 Tables and SQL

This chapter is about tables as a basic data structure in databases. We will study them using the database language SQL, starting from the basics and advancing little by little. The concepts are introduced using examples, and no prior knowledge of databases is required. We will first explain the main language constructs of SQL by using just one table. We will learn to define database tables using SQL. We will insert data to the tables using SQL, as well as delete and update existing data. We will write queries including selections, projections, renamings, unions, intersections, groupings, and aggregations. We will also take a look at the different datatypes of SQL and operations that manipulate their values. After introducing the main operations of SQL, we generalize the treatment to many tables. This generalization involves just a few new SQL constructs: foreign keys, cartesian products, and joins. But it is an important step conceptually, since it raises the question of how a database should be divided to separate tables. This question will be the topic of the subsequent chapters on database design.

2.1 SQL database table

The table below shows data about various countries, where currency is represented with their standard 3-character code and continents with shorthand expressions for Africa (AF), Europe (EU), South America (SA), North America (NA), and Asia (AS).

name	**capital**	**area**	**population**	**continent**	**currency**
Tanzania	Dodoma	945087	41892895	AF	TZS
Greece	Athens	131940	11000000	EU	EUR
Sweden	Stockholm	449964	9555893	EU	SEK
Peru	Lima	1285220	29907003	SA	PEN
Netherlands	Amsterdam	41526	16645000	EU	EUR
Finland	Helsinki	337030	5244000	EU	EUR
Cuba	Havana	110860	11423000		CUP
China	Beijing	9596960	1330044000	AS	CNY
Chile	Santiago	756950	16746491	SA	CLP

The first row with text in **bold** is the **title row**, describing the role of data stored in each **column** of the subsequent rows. Each of those subsequent rows contains data about one country.

The table is the basic structure of databases in the sense of SQL. One interpretation is that each row describes a complex object: here, a country with all its relevant attributes. Another interpretation is that each row of the table states a fact. For instance, the first row after the title row states the fact that

> *there exists a country named Tanzania, whose capital is Dodoma, whose area is 945087 and population 41892895, which lies on the African continent, and uses a currency whose code is TZS.*

Each row states a fact of the same format. Those familiar with logic should see a similarity between rows and logical propositions. In this interpretation, the table is a **set** of facts, which means that the order of the rows is seen unimportant, and re-ordering the rows does not change the information content. However, we will see that SQL also provides a method to order the output produced from a table in a desired way, for instance alphabetically or by the size of the country.

The values stored in each column have the same **type**, such as string (name, capital), or number (area, population). Some are strings with a fixed length, e.g. in our example table currency has length 3, and continent length 2. Defining the types for each column is an important part in guaranteeing that the table makes sense: for instance, that the population is always a number and as such it can be compared with other populations. This way, the choice of type must be such that it takes into account the values that are going to be inserted in the table in the future. If for instance some currencies had 4-character codes, that should be taken into account when choosing the type for currency. When we use a table to store values in a database, we will define such properties explicitly, and then the database system will ensure that the values stored fulfill those properties.

It is not necessary that all values exist on all rows: in the above table Cuba has no continent, since it is considered here an island outside continents (technically, it is usually considered to be in NA). Some values or their combinations can be unique for each row and they can thereby used to identify each row. In the above table name seems to be such a column on its own.

2.2 SQL in a database management system

We will in the following assume that you have a working installation of PostgreSQL and access to a command line shell. It can be a Unix shell, called Terminal in Mac, or Command Prompt in Windows. Usually the working installation keeps on running automatically, even when the computer is restarted, but you need to connect to it.

In your command line shell, you can start a program that connects to the PostgreSQL database with the command

```
psql Countries
```

if `Countries` is the name of the database that you are using. If you are administrating your own PostgreSQL installation, you may use the Unix shell command

```
createdb Countries
```

to create such a database; this you will only have to do once. After this, you can start PostgreSQL with the command `psql Countries`.

2.3 Creating an SQL table

Below is a statement in the SQL language to create a table for data on countries.

```
CREATE TABLE Countries (
  name TEXT PRIMARY KEY,
  capital TEXT NOT NULL,
  area FLOAT,
  population INT,
  continent CHAR(2),
  currency CHAR(3) NOT NULL
  ) ;
```

In this SQL statement we give a name to a table we create (`Countries`), and introduce the columns of the table. The names of the columns are called the **attributes** of the table.

The first line of this `CREATE TABLE` statement gives the keywords to create a table plus a name for the table. The name must be different from any already existing table in our database. Initially, we have not defined any tables, but if we give the same CREATE TABLE statement twice, the second one leads to an error as a table with that name already exists.

Each attribute is given a name and a type. The most commonly used types include

- `INT` for integer values
- `FLOAT` for floating-point decimal values
- `TEXT` for character strings of any length
- `CHAR(n)` for strings of length n (1,2,...)
- `VARCHAR(n)` for strings whose length is at most n

Thus SQL has several types for strings: `CHAR(n)`, `VARCHAR(n)`, and `TEXT`. This is partly historical heritage: in old systems, fixing the length of strings improved the performance, for instance minimizing the storage needs. However, the current PostgreSQL manual says as follows: "*There are no performance differences between these types... In most situations text or character varying (= varchar) should be used.*" Following this advice, we will in the following mostly use `TEXT` as the string type. However, if it is necessary that all strings have exactly a specific length, like for instance currency codes, then the database system can guarantee this property by using `CHAR(n)`.

The `PRIMARY KEY` after `name TEXT` is a **constraint** saying that name is the **primary key** for the table. This means that every row must have a unique name. In other words: if we know the name of a country, we know all the other attributes. The `NOT NULL` is another constraint, meaning that an attribute must have a value on all rows: here, that every country must have a currency and a capital.

`PRIMARY KEY` implies `NOT NULL`, but the columns without these constraints (area, population, continent) need not have values. When a value is missing, it is said to be `NULL`. Such `NULL` values can mean two things: that we don't know the value, or that the value does not exist.

A peculiar feature of the SQL language is that it is **case-insensitive**: `Countries`, `countries`, and `COUNTRIES` are all interpreted as the same name. But a good practice that is often followed is to use

- capital initials for tables: `Countries`
- small initials for attributes: `name`
- all capitals for SQL keywords: `CREATE`

However, string literals (in single quotes, used for e.g. attribute values) are case-sensitive. Hence `'sweden'` is treated as distinct from `'Sweden'`.

If you want to get rid of the table you created, you can remove it with the following command, and then create it with different properties.

```
DROP TABLE Countries
```

This must obviously be used with caution: if there is data in the table, it will all get lost!

A table once created can be changed later with an `ALTER TABLE` statement. The most important ways to alter a table are:

- `ADD COLUMN` with the usual syntax (attribute, type, inline constraints). The new column may by default contain `NULL` values, unless specified otherwise.
- `DROP COLUMN` with the attribute name
- `ADD CONSTRAINT` with the usual constraint syntax. Rejected if the already existing data in the table violates the constraint.
- `DROP CONSTRAINT` with a named constraint

Notice that `DROP COLUMN` must also be used with caution since the data already in the column gets lost.

2.4 Grammar rules for SQL

There are different variations and dialects of the SQL language. Our aim is not to cover them all or compare them, but to introduce a subset suitable for most tasks and giving a reasonable overview of the language.

While different database language feature come up, we will introduce their grammatical description, which makes it possible to understand "all the things" one can do with these constructs. Throughout the book, we will introduce the grammar structures as they are used. Appendix 1 collects the grammar of the whole SQL into one place.

To represent the grammars, we use the BNF notation (Backus Naur Form), with the following conventions:

- CAPITAL words are SQL keywords, to take literally
- small character words are names of syntactic categories, such as table names etc, defined each in their own rules
- | separates alternatives
- `+` means one or more, separated by commas
- `*` means zero or more, separated by commas
- `?` means zero or one

- in the beginning of a line, `+ * ?` operate on the whole line; elsewhere, they operate on the word just before
- `##` start comments, which explain unexpected notation or behaviour
- text in double quotes means literal code, e.g. `"*"` means the operator `*`
- other symbols, e.g. parentheses, also mean literal code (quotes are used only in some cases, to separate code from grammar notation)
- parentheses can be added to disambiguate the scopes of operators

Also recall that SQL syntax is **case-insensitive**:

- **keywords** are usually written with capitals, but can be written by any combinations of capital and small letters
- the same concerns **identifiers**, i.e. names of tables, attributes, constraints
- however, **string literals** in single quotes are case sensitive

As the first example, the grammar of `CREATE TABLE` statements is specified as follows:

```
statement ::=
    CREATE TABLE tablename (
  * attribute type inlineconstraint*
  * [CONSTRAINT name]? constraint
    ) ;
```

In other words, these statements consist of

- the keywords `CREATE TABLE` followed by a left parenthesis "("
- a list, possibly empty, of attribute definitions, each of which has
  - an attribute
  - a type
  - a list, possibly empty, of inline constraints
- a list, possibly empty, of constraints, each of which has
  - an optional name preceded by the keyword `CONSTRAINT`
  - a constraint

This grammar rule assumes that the syntactic categories `tablename`, `attribute`, `type`, `inlineconstraint`, `name`, and `constraint` are defined in some other grammar rules. Those rules will follow in the next section.

2.5 A further analysis of CREATE TABLE statements

We start with the grammar rules for types and inline constraints.

```
type ::=
    CHAR ( integer )
  | VARCHAR ( integer )
  | TEXT | INT | FLOAT | BOOLEAN

inlineconstraint ::=
    PRIMARY KEY | UNIQUE | NOT NULL
  | DEFAULT value
  | REFERENCES tablename ( attribute )
```

```
    | CHECK ( condition )
```

An example of an inline constraint is `PRIMARY KEY` after `name TEXT` in `Countries` table above. It says that `name` is a the primary key for the table. Another example that appeared in `Countries` is `NOT NULL`, which says that the value may not be NULL. The constraint `UNIQUE` means each row must have a different value for that attribute.

As for the constraints that we haven't seen yet, `DEFAULT value` which allows us to define a default value for the attribute. `REFERENCES` is used when the attribute refers to some other table; Section 2.16 will explain this. `CHECK` states some condition that the value must satisfy: for instance `CHECK ( area <> 0)` prohibits 0 values of the area.

An inline constraint can only refer to one attribute, the one on the same line. Some constraints must refer to many attributes at the same time. A common example is when the primary key is a **composite key**, that is, composed from several attributes. Composite keys are used when no attribute alone is unique, but a combination of them is. Suppose, for the moment, that many continents have a country named "United States". Then the country name alone cannot be used as a primary key, but the pair `(continent, name)` may be possible. The constraint will then read

```
PRIMARY KEY (continent, name)
```

Since such constraints refer to several attibutes, they must be separately from the introduction of the attributes themselves, in the `constraint` part of the `CREATE TABLE` statement. Their grammar is similar to inline constraints, except that they have to mention the list of attributes affected:

```
constraint ::=
    PRIMARY KEY ( attribute+ )
  | UNIQUE ( attribute+ )
  | FOREIGN KEY tablename ( attribute+ )
  | CHECK condition
```

A common practice in databases is to use unique id numbers as primary keys. In this way, composite keys can be avoided. But the database designers might still want to express it as a constraint that continent-name pairs must be unique:

```
CONSTRAINT continent-name-pair-is-unique UNIQUE (continent, name)
```

Besides being informative, giving a good name to the constraint is beneficial in that the system uses it in the error message whenever the constraint is violated.

In general, `PRIMARY KEY` has the same effect as `UNIQUE` and `NOT NULL` together. What is the difference, then? One difference is that a table can have only one `PRIMARY KEY` but many `UNIQUE NOT NULL` combinations. Another difference is that some systems are more reluctant to change or remove from existing tables their key constraints than their unique definitions.

Exercise 2.1 Make your own SQL CREATE TABLE statement for the Country data. Use data types and constraints that are different from the book.
Exercise 2.2 Consider the Country table above and the following SQL CREATE TABLE statement. List the things that make it unreasonable for storing data on countries.

```
CREATE TABLE Countries (
  name CHAR(6),
  capital TEXT NOT NULL,
  area INT,
  population FLOAT ,
  continent CHAR(2) PRIMARY KEY,
  currency INT
  ) ;
```

Exercise 2.3 Suppose you want to sell items, which have at least a name, a description and a value. Write a CREATE TABLE statement for a table containing data of those items.

2.6 Inserting rows to a table

A new table, when created, is empty. To put data intor the table, one can use `INSERT` statements, such as

```
INSERT INTO Countries VALUES
  ('Peru','Lima',1285220,29907003,'SA','PEN') ;
```

This can be done from the PostgreSQL prompt. But a more convenient way for a lot of data is to prepare a file that contains the statements and then read in the statements into the program:

```
\i countries-data.sql
```

In PostgreSQL, an even quicker way to insert values is from tab-separated files (i.e. each value in a row is separated by a tabulator character):

```
COPY tablename FROM filepath
```

Notice that a complete filepath is required, starting with the root (`/` in Linux and MacOS, \ in Windows). The data in the file must of course match your database schema. To give an example, if you have a table

```
Countries (name,capital,area,population,continent,currency)
```

you can read data from a file that looks like this:

```
Andorra    Andorra la Vella       468     84000    EU   EUR
United Arab Emirates   Abu Dhabi  82880   4975593  AS   AED
Afghanistan   Kabul               647500  29121286 AS   AFN
```

The file

http://www.cse.chalmers.se/~aarne/db-course/countries.tsv

can be used for this purpose. It has in turn been extracted from a file received from the Geonames database,

http://www.geonames.org/

The order of values in `INSERT` statements must be the same as the order of the attributes given when creating the table. This makes it a bit brittle, and also unpractical if the data is read from a file that assumes a different order. More control is obtained by specifying the attributes separately before the `VALUES` keyword:

```
INSERT INTO Countries
  (continent,name,capital,currency,population,area) VALUES
    ('SA','Peru','Lima','PEN',29907003,1285220) ;
```

The grammar of insert statement is the following:

```
statement ::= INSERT INTO tablename tableplaces? value+ ;

tablespaces ::= ( attribute+ )

value ::=
   integer | float | 'string'
 | value operation value
 | NULL
 | ## nothing between the commas, meaning NULL

operation ::= "+" | "-" | "*" | "/" | "%" | "||"
```

(We will come back to the arithemetic operations listed here in Section ??.)

The value of an attribute is set to `NULL` in the following cases:

- it is given with the identifier `NULL`
- it is given as the empty string between commas: `value1,,value3`
- it is left out from the explicit attribute list before the `VALUES` keyword
- it is left out when the list of values given is shorter than the table's attribute list

Thus listing of methods obviously goes from the most controlled to the least controlled one.

Many things can still go wrong with a grammatically correct insert statement. The data types of the values may be incorrect. There may be NULL values where they are not allowed. The primary key constraint may be violated, if the inserted primary key value is already in use in the table. So may the other constraints. You are urged to try out these, to see what happens. A good practice, much of the time, is to include enough constraints in the `CREATE TABLE` statement to prevent accidental insertion of incorrect data.

Exercise 2.4 Write INSERT INTO statements to add data to the table designed in Exercise 2.3.

2.7 Deleting and updating

To get rid of all of your rows you have inserted, you may either remove the table completely with the `DROP TABLE` command or use the following form of `DELETE FROM` command:

```
DELETE FROM Countries
```

This will delete all rows from the table but keep the empty table. To select just a part of the rows for deletion, a `WHERE` clause can be used:

```
DELETE FROM Countries
WHERE continent = 'EU'
```

will delete only the European countries. The condition in the `WHERE` part can be any SQL condition, which will be explained in more detail later.

By using a sequence of `DELETE` and `INSERT` statements we can modify the contents of the table row by row. It is, however, also practical to be able to change a part of the contents of rows without having to delete and insert those rows. For this purpose, the `UPDATE` statement is to be used:

```
UPDATE Countries
SET currency = 'EUR'
WHERE name = 'Sweden'
```

is the command to issue the day when Sweden joins the Euro zone.

The `UPDATE` command can also refer to the old values. For instance, when a new person is born in Finland, we can celebrate this by updating the population as follows:

```
UPDATE Countries
SET population = population + 1
WHERE country = 'Finland'
```

2.8 Querying: selecting columns and rows from a table

The purpose of storing data is to make possible to seach necessary information from it. This is done in SQL with the `SELECT FROM WHERE` statements. We will go through the use of those statements little by little.

The simplest example is the statement

```
SELECT * FROM countries
```

which returns the whole countries table. The table can be restricted by adding a `WHERE` clause:

```
SELECT * FROM countries WHERE continent = 'EU'
```

results in a table that shows only the European countries,

name	capital	area	population	continent	currency
Greece	Athens	131940	11000000	EU	EUR
Sweden	Stockholm	449964	9555893	EU	SEK
Netherlands	Amsterdam	41526	16645000	EU	EUR
Finland	Helsinki	337030	5244000	EU	EUR

The asterisk `*` means that all attributes are selected for the result. But another set of attributes can also be listed, as in

```
SELECT name, population FROM countries WHERE continent = 'EU'
```

which returns a table with just two columns:

name	population
Greece	11000000
Sweden	9555893
Netherlands	16645000
Finland	5244000

Thus the result of a `SELECT-FROM-WHERE` query is the intersection of `WHERE` which cuts the table horizontally, and `SELECT`, which cuts it vertically. A moments reflection shows that the `WHERE` clause must generally be executed before `SELECT`, since it may refer to attributes that `SELECT` removes. The previous example is a case in point.

A table that has a primary key cannot have duplicate rows. However, duplicates may appear in tables produced as answers to SQL queries, because of `SELECT`. For example

```
SELECT currency  FROM countries
```

returns

currency
CUP
EUR
EUR
CNY
CLP
PEN
TZS
EUR
SEK

with duplicated currencies. The `DISTINCT` keyword can be used to eliminate those duplicates and return each currency only once:

```
SELECT DISTINCT currency FROM countries ;
```

gives the following answer:

currency
CUP
EUR
CNY
CLP
PEN
TZS
SEK

2.9 Conditions in WHERE clauses

The grammar for queries covered so far is the following:

```
statement ::=
  SELECT DISTINCT? attribute+ FROM table+ WHERE condition

condition ::=
   expression comparison expression
 | expression NOT? BETWEEN expression AND expression
 | condition boolean condition
 | expression NOT? LIKE 'pattern*'
 | expression NOT? IN values
 | NOT? EXISTS ( query )
 | expression IS NOT? NULL
 | NOT ( condition )

comparison ::=
  = | < | > | <> | <= | >=

expression ::=
   attribute
 | value
 | expression operation expression

boolean ::= AND | OR

pattern ::=
   %
 | _
 | character  ## match a specific string/char
 | [ character* ]
 | [^ character* ]
```

Most productions in this grammar have to do with the conditions that can be used in the `WHERE` clause. As we saw earlier, such `WHERE` clauses are also used in `UPDATE` and `DELETE` statements, and conditions are also used in `CHECK` constraints. Hence a closer look at conditions is in place here.

The simplest `WHERE` conditions are comparisons of values, which can come from attributes and constants. Both numeric values and strings can be compared, and the symbols in this case are similar to ordinary programming languages, with the slightly deviant symbol for inequality, <>:

```
SELECT name FROM countries WHERE population > 5000000

SELECT name FROM countries WHERE name = capital

SELECT name FROM countries WHERE continent <> 'EU'
```

Values that are compared can in general be **expressions** built from attributes and constants. Thus the following query selects countries with population density at least 100:

```
SELECT name FROM countries WHERE population/area >= 100
```

Conditions can moreover be combined with boolean operators `AND` and `OR`, as well as `NOT`:

```
SELECT name FROM countries WHERE currency = 'EUR' AND
                                 NOT (continent = 'EU')
```

Boolean and arithmetic operators have **precedences**, which make it possible to omit parentheses, but it is always possible to use them to ensure proper grouping.

The comparison operator `LIKE` compares a string with a **pattern**. Patterns are built from letters in combination with **wildcards**:

- `_` matching any single character
- `%` matching an arbitrary string

Thus

- `name LIKE '%en'` is true of names ending with "en", such as "Sweden"
- `code LIKE '___'` is true of codes that consist of three characters

Some more patterns, such as character ranges, are given in the Appendix.

The NULL value needs special attention in comparisons: every ordinary comparison with `NULL` fails. So does in particular

```
e = e
```

if the expression `e` has value `NULL`. However, there is a special comparison operator "IS NULL', which can be used:

```
e IS NULL
```

which is true if `e` has the value `NULL`.

To give an example, recall that in our example table the continent of Cuba is `NULL`. Hence all of the following queries return an empty table: So, **only** the answer to the last one of the following queries contains a row, the others will evaluate to an empty set of rows.

```
SELECT * FROM countries WHERE name = 'Cuba' AND continent = 'EU'

SELECT * FROM countries WHERE name = 'Cuba' AND continent <> 'EU'

SELECT * FROM countries WHERE name = 'Cuba' AND
                        (continent = 'EU' OR continent <> 'EU')
```

But the following query returns the row with Cuba:

```
SELECT * FROM countries WHERE name = 'Cuba' AND continent is NULL
```

We will return to the logic of NULL values in Section 2.22.

2.10 Expressions in SELECT fields

In the previous examples, the `SELECT` field has contained a list of attributes or the shorthand `*` denoting all attributes. But the full syntax is more general in two ways:

- any **expression** can be used, not just attributes
- columns can be **renamed** with the `AS` keyword

The simplest example of expressions is constant expressions not involving any attributes. Thus the "hello world" program in SQL is

```
SELECT 'Hello world'
```

Similarly, to use SQL as a calculator, you can write

```
SELECT 2+2
```

These are the minimal queries in SQL: no `FROM` fields with tables are needed. However, the result of a query is always displayed as a table:

?column?
4

The column title is often the expression used in the `SELECT` field, but can also be a dummy title as `?column?` above. Renaming makes it possible to use some other title:

```
SELECT 2+2 AS value
```

value
4

Here is a more complex example, showing countries with their population densities. The operator `FLOOR` drops the digits after the decimal point. The attribute `name` is renamed to `country`, which is more informative than just `name`:

```
SELECT name AS country, FLOOR(population/area) AS density
FROM countries
```

country	density
Sweden	21
Finland	15
Tanzania	44
Peru	23
Chile	22
China	138
Slovenia	98
Greece	83
Cuba	103

Exercise 2.5 Write the following queries and try them to query data from the Countries table.

- Which countries have a population over 100 000 000 and what is their population?
- Which countries use euros as a currency?
- Which European countries do not use euros as a currency?
- Change the currency to euros for all European countries.
- Delete all European countries from the database.
- Insert the original data of European countries to the database.

2.11 Set operations on queries

SQL queries can be combined with set operations, familiar from discrete mathematics:

- *a* `UNION` *b*, returning all rows from tables *a* and *b*
- *a* `INTERSECT` *b*, returning the rows that apper in both *a* and *b*
- *a* `EXCEPT` *b*, returning the rows that apper in *a* but not in *b*

For example,

```
SELECT name FROM Countries WHERE continent = 'EU'
  UNION
SELECT name FROM Countries WHERE continent = 'AF'
```

returns the names of European and African countries. The same names would be obtained by the query that uses `OR` in the `WHERE` field:

```
SELECT name FROM Countries
WHERE continent = 'EU' OR continent = 'AF'
```

But the following query would not be easy to express without `UNION`:

```
SELECT name AS place FROM Countries
  UNION
SELECT capital AS place FROM Countries
```

It returns all names of places, be it countries or capitals.

Set operations can only be applied to tables "of the same type", that is, tuples with the same number of elements of same types. Thus a union of names and populations would not be valid. However, as the above example shows, the attribute names need not match, union of name and capital is ok. The resulting table uses the attribute name of the first query, `name`, as the title of the column. This is probably not always the desired outcome, and renaming gives a nicer result.

Renaming expressions is also a way to hide precise information. Consider the task of classifying countries to big and small ones, without showing their actual sizes. Assuming that a big country is one with the population at least 50 million, we can write

```
SELECT name, 'big'   AS size
FROM countries WHERE population >= 50000000
  UNION
SELECT name, 'small' AS size
FROM countries WHERE population <  50000000
```

to show the sizes of all countries as just "big" or "small", hiding the exact numeric populations.

There is a subtlety in the set operations: they are defined always to return **sets**, in the mathematical sense that every row counts just once. This is opposed to **multisets**, where duplicates of rows count separately. While SQL tables with primary keys have no duplicate rows, a `SELECT` query that omits the primary key attribute can result in a table with duplicates, as shown by the query

```
SELECT currency FROM countries
```

The `DISTINCT` keyword can be used to remove duplicates,

```
SELECT DISTINCT currency FROM countries
```

However, since `UNION` is a set operation, the same effect can be obtained with

```
SELECT currency FROM countries
  UNION
SELECT currency FROM countries
```

Hence the intuition that the union of two sets is at most as big as each of the set fails here: the SQL union of a table can be smaller than the table itself!

To prevent duplicate removal, SQL provides the `ALL` keyword. The following query keeps all duplicates of currencies, actually even doubling them:

```
SELECT currency FROM countries
  UNION ALL
SELECT currency FROM countries
```

Exercise 2.6 Write the following queries using set operations and try them to query data from the Countries table.

- Which countries use euros as a currency and do not have a population density at least 50?
- Which European countries do not use euros as a currency, have a population density at most 100, and have a capital whose name starts with 'A'?

2.12 Set operations in conditions

A `WHERE` condition can test membership in a set of values, which can be given explicitly:

```
SELECT name
FROM countries
WHERE currency IN ('EUR','USD')
```

This membership test is a shorthand for two equality comparisons combined with `OR`. But one can also compare membership in the results of a query:

```
SELECT name
FROM countries
WHERE continent IN
  (SELECT continent FROM countries WHERE currency = 'EUR')
```

The WHERE clause contains a **subquery**, that is, another SQL query that defines a set of values. This subquery returns the continents where Euro is used as currency. The main query returns the countries on such continents.

Ordinary omparisons such as equality work between values, but not between queries, and particularly not between queries and values. However, a comparison between a single value and a set is meaningful, if the set has only one value:

```
SELECT name
FROM countries
WHERE continent =
  (SELECT continent FROM countries WHERE name = 'Finland')
```

In this case, we could as well use the `IN` operator instead of `=`. But other comparison operators can operate on subqueries with multiple results, if prefixed with the keyword `ALL`. Thus the following query returns all countries whose population is greater than in all South-American countries:

```
SELECT name
FROM countries
WHERE population >
  ALL (SELECT population FROM countries WHERE continent = 'SA')
```

2.13 Sorting results

The `ORDER BY` keyword forces the listing of rows in a specified order. This operation is generally called **sorting** in computing. Thus the following query sorts countries primarily by the continent in **ascending** order (from A to Z), and secondarily (wiithin continents) by size in **descending** order (from smallest to largest):

```
SELECT name, population, continent
FROM Countries
ORDER BY continent, population DESC
```

name	population	continent
Tanzania	41892895	AF
China	1330044000	AS
Greece	11000000	EU
Sweden	9555893	EU
Finland	5244000	EU
Slovenia	2007000	EU
Peru	29907003	SA
Chile	16746491	SA
Cuba	11423000	

The default order is ascending, whereas descending order can be forced by the keyword `DESC`. Cuba, with NULL continent, appears last in the listing.

`ORDER BY` is usually presented as a last field of a `SELECT` group. But it can also be appended to a query formed by a set-theoretic operation:

```
(SELECT name, 'big' AS size
 FROM countries WHERE population >= 50000000
   UNION
 SELECT name, 'small' AS size
 FROM countries WHERE population < 50000000
)
  ORDER BY size, name
```

shows first all big countries in alphabetical order, then all small ones. Without parentheses around the union query, `ORDER BY` would be applied only to the latter query.

In mathematical terms, `ORDER BY` means that tables are treated as **lists**, where not only duplicates but also their order counts. Set operations such as `UNION` and `DISTINCT` preserve the order, while removing duplicates.

2.14 Aggregation and grouping

Aggregation functions collect values from several rows into single values, such as sums and averages. The usual aggregation functions are

- `COUNT`, the number of rows
- `SUM`, the sum of values
- `MIN`, the smallest value
- `MAX`, the biggest value
- `AVG`, the average value

Thus the following query gives the minimum, maximum, total, and average population for countries in South America, as well as the number of countries:

```
SELECT
  MIN(population), MAX(population), SUM(population),
  AVG(population), COUNT(name)
FROM countries
WHERE continent = 'SA'
```

min	max	sum	avg	count
16746491	29907003	46653494	23326747.000000000000	2

Wrapping the average with `FLOOR` would remove the decimals.

What about the corresponding statistics for all continents? To do this, we need to add yet another field to `SELECT` queries: `GROUP BY`, which forms subgroups of the data, where the countries on the same continent are grouped together:

```
SELECT
    continent, MIN(population), MAX(population), SUM(population),
    FLOOR(AVG(population)) AS avg, COUNT(name)
FROM countries
GROUP BY continent
```

continent	min	max	sum	avg	count
	11423000	11423000	11423000	11423000	1
SA	16746491	29907003	46653494	23326747	2
AS	1330044000	1330044000	1330044000	1330044000	1
EU	2007000	11000000	27806893	6951723	4
AF	41892895	41892895	41892895	41892895	1

We can also group the rows by continent and then calculate the values for **all** continents, using the `GROUP BY` construction, as follows

```
SELECT
  MIN(population), MAX(population), AVG(population), COUNT(population)
FROM countries
GROUP BY continent ;
```

After grouping, we may want to restrict the result by aggregation functions. For instance, we may want to include only those continents that have more than one country in them. The natural thing would be to use a `WHERE` clause. However, SQL syntax does not permit `WHERE` after `GROUP BY`, but uses another keyword, `HAVING` for this purpose. Here is the query that we want:

```
SELECT
  continent, MIN(population), MAX(population), SUM(population),
  FLOOR(AVG(population)) AS avg, COUNT(name)
FROM countries
GROUP BY continent
HAVING COUNT(name) > 1
```

continent	min	max	sum	avg	count
SA	16746491	29907003	46653494	23326747	2
EU	2007000	11000000	27806893	6951723	4

As a final note on aggregation syntax at this point, we can mention the use of the `DISTINCT` keyword in aggregation functions:

```
SELECT COUNT(currency) FROM Countries
```

counts all duplicates, whereas

```
SELECT COUNT(DISTINCT currency) FROM Countries
```

removes duplicate currencies before counting them.

Technically, applying `GROUP BY` a to a table R forms a new table, where a is the key. For instance, `GROUP BY currency` forms a table where currencies are keys. But what are the other attributes? The original attributes of R won't do, because their values are lost in grouping. For instance, there are many `EUR` countries, and there is no one country name that could represent the whole group. So what is the exact meaning of the `GROUP BY` construction, seen as a table?

The truth about `GROUP BY` can be seen only by looking at the `SELECT` line above it. On this line, only the following attributes of R may appear:

- the grouping attribute a itself, which becomes the primary key
- aggregation functions on the any other attributes or R
- attributes that have the same value inside each group.

  The third class of attributes is not generally allowed in existing dialects of SQL. Here is an example (with the result truncated from our standard example):

  ```
  SELECT currency, COUNT(name)
  FROM Countries
  GROUP BY currency
  ```

currency	count
CNY	1
PEN	1
CUP	1
EUR	3

As we have seen, the grouped table can have more than one aggregation attributes: all of those appearing in the `SELECT` field, such as

```
SELECT currency, COUNT(name), AVG(population)
```

However, it is not only the `SELECT` field that counts. As a final subtlety of grouping: the table formed by `GROUP BY` must also contain the aggregations used in the `HAVING` and `ORDER BY` clauses, even if they are not shown in the final table:

```
SELECT currency, AVG(population)
FROM Countries
GROUP BY currency
HAVING COUNT(name) > 1

SELECT currency, AVG(population)
FROM Countries
GROUP BY currency
ORDER BY COUNT(name) DESC
```

From the semantic point of view, `GROUP BY` is a very complex operator, because one has to look at many different places to see exactly what table it forms. We will get more clear about this when looking at relational algebra and query compilation in Chapter 6.

2.15 Using data from several tables

Let's now add to our table the currency values (in US dollars).

country	**capital**	**area**	**pop.**	**cont.**	**curr.**	**value**
Tanzania	Dodoma	945087	41892895	AF	TZS	1.25
Greece	Athens	131940	11000000	EU	EUR	1.176
Sweden	Stockholm	449964	9555893	EU	SEK	0.123
Peru	Lima	1285220	29907003	SA	PEN	0.309
Netherlands	Amsterdam	41526	16645000	EU	EUR	1.176
Finland	Helsinki	337030	5244000	EU	EUR	1.176
Cuba	Havana	110860	11423000		CUP	0.038
China	Beijing	9596960	1330044000	AS	CNY	0.150
Chile	Santiago	756950	16746491	SA	CLP	0.001

The table is now so wide that we have had to abbreviate the attribute names to fit it on the page. The real problem, though, is that the column value creates a problem of **data consistency**. The value data needs to be updated daily, as the currency rates are constantly changing. Since many countries use the same currency (e.g. `EUR`), the same update has to be performed on several rows. What if we forget to update the value on all of those rows? We will then have

an **inconsistency** in the values, caused by the **redundancy** in repeating the same information many times.

To avoid this inconsistency, we will, instead of one table, store the data in two separate tables and learn how to combine data from different tables in SQL queries. The basic intuition is to avoid redundancy that may lead to inconsistencies. The current case quite obvious, but in general, this can be a tricky problem, and Chapters 3 and 5 dealing with database design will give more tools for deciding how to divide data into tables.

So, instead of adding a new attribute to the table on countries, we will now create a new table which contains just the information on the values of currencies. We can also add some other information about currencies, which can later be read from this table instead of being repeated for each country. Our new table contains the currency name, to support queries like "which countries have a currency named peso":

currency	**name**	**value**
TZS	Schilling	1.25
EUR	Euro	1.176
SEK	Crown	0.123
PEN	Sol	0.309
CUP	Peso	0.038
CNY	Yuan	0.150
CLP	Peso	0.001

The SQL statement to create the new table is

```
CREATE TABLE Currencies (
  code TEXT PRIMARY KEY,
  name TEXT,
  value FLOAT
  )
```

Notice that the primary key is the 3-letter currency code rather than the name, because many currencies can have the same name.

Now that we have split the information about countries to two separate tables, we need a way to combine this information. The general term for this is **joining** the tables. We will later introduce a set of specific `JOIN` operations in SQL. But a more elementary method is to use the `FROM` part of `SELECT FROM WHERE` statements, and give a list of tables that are used, instead of just one table.

Let us start with a table that shows, for each country, its capital and the value of its currency:

```
SELECT capital, value
FROM Countries, Currencies
WHERE currency = code
```

This query compares the `currency` attribute of `Countries` with the `code` attribute of `Currencies` to select the matching rows.

But what if we want to show the names of the countries and the currencies? Following the model of the previous query, we would have

```
SELECT name, name
FROM Countries, Currencies
WHERE currency = code
```

This query is not understandable to a human, neither is it to a database system. PostgreSQL indeed gives the error message

```
ERROR:  column reference "name" is ambiguous
LINE 1: SELECT name, name
```

This is because now both `Countries` and `Currencies` contain a column named `name`, and it is not clear which one is referred to in the query. The solution is to use **qualified names** where the attribute is prefixed by the table name:

```
SELECT Countries.name, Currencies.name
FROM Countries, Currencies
WHERE currency = code
```

We can also introduce shorthand names to the tables with the `AS` construct, and use these names elsewhere for the table (recalling that the `FROM` part, where the names are introduced, is evaluated first in SQL):

```
SELECT co.name, cu.name
FROM Countries AS co, Currencies AS cu
WHERE co.currency = cu.code
```

The first step in evaluating a query is to form the table in accordance with the `FROM` part. When it has two tables like here, their **cartesian product** is formed first: a table where each row of `Countries` is paired with each row of `Currencies`. This is of course a large table: even with our miniature database of the world's countries, it has $9 \times 7 = 63$ rows, since `Countries` has 9 rows and `Currencies` has 7. But the `WHERE` clause shrinks its size to 9, because the `currency` is the same as `code` on only 9 of the rows. [1]

The condition in the `WHERE` part is called a **join condition**, as it controls how rows from tables are joined together. We can of course also state other conditions in the `WHERE` part, as we did before: for instance,

```
WHERE co.currency = cu.currency AND co.continent = 'EU'
```

[1] In practice, SQL systems are smart enough not to build the large cartesian products if it is possible to optimize the query and shrink the table in advance. Chapter 6 will show some ways in which this is done.

would only include European countries. Leaving out a join condition will produce all pairs of rows - the whole cartesian product - in the result. The reader is urged to try out

```
SELECT Countries.capital, Currencies.name
FROM Countries, Currencies
```

and see the big table that results, with lots of unrelated capitals and currency names. You can use the `COUNT` aggregation function to see just the number of rows are created in a table. Here is an artificial example:

```
SELECT COUNT(*) FROM Countries AS A, Countries AS B, Countries AS C;
 count
-------
 729
```

This example also shows that you can form the cartesian product of a table with itself, but that you then have to rename the table to avoid ambiguous attributes.

2.16 Foreign keys and references

The natural way to join data from two tables is to compare the keys of the tables. For instance, `currency` values in `Countries` are intended to match `code` values in `Currencies`. If we want to require this always to hold, we define `currency` in `Countries` as a **foreign key** that **references** `code` in `Currencies`. We can do this either as an inline constraint next to the column declaration,

```
CREATE TABLE Countries (
  -- ...
  currency CHAR(3) REFERENCES Currencies(code)
  -- ...
  )
```

or by adding a constraint to the end of the `CREATE TABLE` statement,

```
FOREIGN KEY (currency) REFERENCES Currencies(code)
```

If the foreign key is composite, only the latter method works, just as with primary keys.

Hint: constraints can even be added after the table has been created by the `ALTER TABLE` statement, for instance,

```
ALTER TABLE Countries
  ADD FOREIGN KEY (currency) REFERENCES Currencies(code)
```

The `FOREIGN KEY` clause in the `CREATE TABLE` statement for `Countries` adds a requirement that every value in the column for `currencies` must be found exactly once in the `code` column of the `Currencies` table. It is the job of a

database management system to check this requirement, prohibiting all deletions from `Currencies` or inserts to `Countries` that would result in a currency in `Countries` that doesn't exist in `Currencies`. In this way, the database management system maintains the **integrity** of the database.

Any conditions in the `WHERE` part can be used for joining data from tables. However, there are some particularly interesting cases, like joining a table with itself. The following query lists all pairs of countries that have the same currency:

```
SELECT co1.name, co2.name
FROM Countries AS co1, Countries AS co2
WHERE co1.currency = co2.currency AND co1.name < co2.name
```

(Notice the use of < rather than <> to state that the countries are different. Using <> would repeat every pair in the two possible orders.)

It is of course possible to join more than two tables, basically an unlimited number. Adding the `currency` table to the query above we can add the currency value to the table:

```
SELECT co1.name, co2.name usd_value
FROM Countries AS co1, Countries AS co2, Currencies
WHERE
  co1.currency = co2.currency
  AND co1.name < co2.name
  AND co1.currency = Currencies.name
```

Thanks to our foreign key requirement, we know that each currency in the `Countries` table is found in the `Currencies` table. What we do not know is if there is a currency that is not used in any country. In that case, there would be data not participating in the join. This is also the case with NULL values. In the next section, we will have a look at particular SQL statements to join data, which also deals with the problem of rows not joining with any rows in the other table.

2.17 Database diagrams and mutual references

Tables, attributes, keys, and foreign keys can be visualized in a **database diagram**. Figure 2 contains a diagram with three tables: country, city, and currency:

- each table is a box, with the table name as its title
- the box contains the attributes, with primary key attributes in bold
- foreign keys are represented with arrows, which can be branching for multiple attributes

The diagram in Figure 2 states, as expected, that each value of `currency` in `Countries` table must exist as a value of `code` in `Currencies`. Similarly, each value of `country` in `Cities` must exist as `name` in `Countries`.

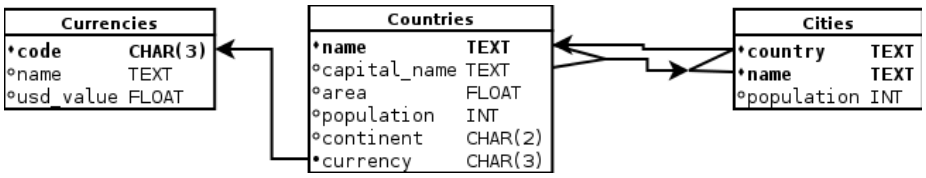


Figure 2: Example database diagram

Extra complexity is added by a mutual reference: each `(name,capital)` pair in `Countries` has to appear as a `(country,name)` pair in `Cities`. This make sense, as every capital must be a city of the same country. But it makes updates and inserts tricky: to introduce a new country, say France, we need first to insert a NULL capital, because Paris can only be inserted when France exists

```
INSERT INTO Countries VALUES ('France',NULL,...)
INSERT INTO Cities    VALUES ('Paris', 'France',...)
UPDATE Countries SET capital = 'Paris' WHERE name = 'France'
```

There are variations in the graphical representation of different features that are used in the diagrams. However, the idea is to represent the key features and in particular the foreign key structure.

2.18 Join operations

Join operations combine date from different tables. The simplest join operation - the one that we have already used - is the cartesian product. It is also called `CROSS JOIN`, but we will use the ordinary notation with commas instead. The full set of join operations contains no less than 24 combinations:

```
table ::=           -- 24 = 8+8+8
   tablename
 | table jointype JOIN table ON condition       -- 8
 | table jointype JOIN table USING (attribute+) -- 8
 | table NATURAL jointype JOIN table            -- 8

jointype ::=                   -- 8 = 6+2
   LEFT|RIGHT|FULL OUTER?   -- 6 = 3*2
 | INNER?                   -- 2
```

Luckily, the `JOIN`s have a compositional meaning. `INNER` is the simplest join type, and the keyword can be omitted without change of meaning. The inner `JOIN` with an `ON` condition gives the purest form of join, similar to cartesian product with a `WHERE` clause:

```
FROM table1 JOIN table2 ON condition
```

is equivalent to

```
FROM table1, table2
WHERE condition
```

The condition is typically looking for attributes with equal values in the two tables. With good luck (or design) such attributes have the same name, and one can write

```
L JOIN R USING (a,b)
```

as a shorthand for

```
L JOIN R ON L.a = R.a AND L.b = R.b
```

well... almost, since when `JOIN` is used with `ON`, it repeats the values of a an b from both tables, like the cartesian product does. `JOIN` with `USING` eliminates the duplicates of the attributes in `USING`.

An important special case is `NATURAL JOIN`, where no conditions are needed. It is equivalent to

```
L JOIN R USING (a,b,c,...)
```

which lists all common attributes of L and R.

Cross join, inner joins, and natural join only include tuples where the join attributes exist in both tables. Outer joins can fill up from either side. Thus **left outer join** includes all tuples from L, **right outer join** from R, and **full outer join** from both L and R.

Here are some examples of inner and outer joins. Assume a table `Capitals` that shows countries with their capitals, and another table, `Currencies`, which shows countries with their currencies. The tables are incomplete, each displaying a different set of countries. Here are the tables:

```
Capitals                      Currencies
 country | capital              country  | currency
 --------+--------             ---------+--------
 Sweden  | Stockholm            Norway   | NOK
 Norway  | Oslo                 Germany  | EUR
```

The effects of different joins are shown in Figure 3. (Notice: to try them in a DBMS, you have to wrap them in `SELECT * FROM ...`)

2.19 Local definitions and views

Local definitions (WITH clauses) are a simple shorthand mechanism for queries. There grammar is

```
query ::= WITH localname AS ( query ) query
```

```
Capitals CROSS JOIN Currencies
country  | capital   | country  | currency
-------- +-----------+----------+----
Sweden   | Stockholm | Norway   | NOK
Sweden   | Stockholm | Germany  | EUR
Norway   | Oslo      | Norway   | NOK
Norway   | Oslo      | Germany  | EUR

Capitals INNER JOIN Currencies ON Capitals.country = Currencies.country
country  | capital   | country  | currency
-------- +-----------+----------+----
Norway   | Oslo      | Norway   | NOK

Capitals INNER JOIN Currencies USING(country)
Capitals NATURAL JOIN Currencies
country  | capital   | currency
-------- +-----------+----------
Norway   | Oslo      | NOK

Capitals FULL OUTER JOIN Currencies USING(country)
country  | capital   | currency
-------- +-----------+---------
Sweden   | Stockholm |
Norway   | Oslo      | NOK
Germany  |           | EUR

Capitals LEFT OUTER JOIN Currencies USING(country)
country  | capital   | currency
-------- +-----------+---------
Sweden   | Stockholm |
Norway   | Oslo      | NOK

Capitals RIGHT OUTER JOIN Currencies USING(country)
country  | capital   | currency
-------- +-----------+--------
Norway   | Oslo      | NOK
Germany  |           | EUR
```

Figure 3: Effects of different joins

For example,

```
WITH
  EuroCountries AS (
    SELECT *
    FROM countries
    WHERE currency = 'Euro'
  )
SELECT *
FROM EuroCountries A, EuroCountries B
WHERE ...
```

is a way to avoid the duplication of the query selecting the countries using the Euro as their currency.

A **view** is like a query in a WITH clause, but its definition is global. A view is created with the `CREATE VIEW` statement

```
statement ::= CREATE VIEW viewname AS ( query )
```

Views are used for "frequently asked queries". They can happily contain redundant information. For example, the following view can be used to display currency rates for different countries:

```
CREATE VIEW CurrencyRates AS (
  SELECT Co.name as country, Co.currency, value
  FROM Countries as Co, Currencies
  WHERE Co.currency = code
) ;
```

Just creating a view does not yet display its data. But it can be queried just like tables created with `CREATE TABLE`, by writing it in the `FROM` field. The simplest example is of course

```
SELECT * FROM CurrencyRates
```

Views are evaluated from the underlying tables each time they are used in queries. Hence, if you have created a view once, it will display different information whenever the underlying tables have changed. Views don't occupy their own storage in the database, and using them can simplify queries considerably.

2.20 Dimensional analysis from relational data*

(not covered in the course)

2.21 SQL as a basis for queries in natural language*

(not covered in the course)

2.22 SQL pitfalls

Even if SQL is simple and easy to learn, it does not always behave the way one would expect. In this section, we list some things that do not feel quite logical in SQL query design, or whose semantics may feel surprising.

Tables vs. queries

Semantically, a query is always an expression for a table (i.e. relation). In SQL, however, there are subtle syntax differences between queries and table expressions (such as table names):

- **A bare table expression is not a valid query.** A bare `FROM` part is hence not a valid query. The shortest way to list all tuples of a table is
  `SELECT * FROM table`
- Set operations can only combine queries, not table expression.
- Join operations can only combine table expresions, not queries.
- A cartesian product in a FROM clause can mix queries and table expressions, but...
- When a query is used in a FROM clause, it must be given an AS name.
- A WITH clause can only define constants for queries, not for table expressions.

In the underlying semantics of SQL, relational algebra, table and query expressions are one and the same thing, which makes the notation more compact and predictable (see Chapter 6).

Renaming syntax

Renaming is made with the AS operator, which however works in two opposite ways:

- In WITH clauses, the name is before the definition: `name AS (query)`.
- In SELECT parts, the name is after the definition: `expression AS name`.
- In FROM parts, the name is after the definition but AS may be omitted: `table AS? name`.

Cartesian products

The bare cartesian product from a FROM clause can be a *huge* table, since the sizes are multiplied. With the same logic, if the product contains an empty table, its size is always 0. Then it does not matter that the empty table might be "irrelevant":

```
SELECT A.a FROM A, Empty
```

results in an empty table even if `A` is not empty. This is actually easy to understand, if you keep in mind that the `FROM` part is executed before the `SELECT` part.

NULL values and three-valued logic

Because of NULL values, SQL follows a three-valued logic: `TRUE`, `FALSE`, `UNKNOWN`. The truth tables as such are natural. But the way they are used in `WHERE` clauses is good to keep in mind. Recalling that a comparison with `NULL` results in `UNKNOWN`, and that `WHERE` clauses only select `TRUE` instances, the query

```
SELECT ...
FROM ...
WHERE v = v
```

gives no results for tuples where `v` is `NULL`. The same concerns

```
SELECT ...
FROM ...
WHERE v < 10 OR v >= 10
```

In other words, if `v` is `NULL`, SQL does not even assume that it has the same value in all occurrences.

Another example[2], involves `NOT IN`: since

```
u NOT IN (1,2,v)
```

means

```
NOT (u = 1 OR u = 2 OR u = v)
```

this evaluates to `UNKNOWN` if `v` is `NULL`. In that case, `NOT IN` is useless as a test.

To see more precisely what happens, let us look at the three-valued logic of SQL conditions. Comparisons with NULL always result in UNKNOWN. Logical operators have the following meanings (T = `TRUE`, F = `FALSE`, U = `UNKNOWN`)

p	**q**	**NOT p**	**p AND q**	**p OR q**
T	T	F	T	T
T	F	F	F	T
T	U	F	U	T
F	T	T	F	T
F	F	T	F	F
F	U	T	F	U
U	T	U	U	T
U	F	U	F	U
U	U	U	U	U

A tuple satisfies a `WHERE` clause only if it returns T, not if it returns U. Keep in mind, in particular, that the negation of UNKNOWN is UNKNOWN!

[2] https://www.simple-talk.com/sql/t-sql-programming/ten-common-sql-programming-mistakes/, first example among the "Ten common SQL mistakes"

Sets and multisets in SQL

Being a set means that duplicates don't count. This is what holds in the mathematical theory of relations (Chapter 4). But SQL is a funny combination of sets and **multisets**, which do allow duplicates.

First of all, typical SQL tables have a primary key. When a primary key exists, the rows in an SQL table are unique and thereby the SQL tables are sets of rows. But since SQL queries do not remove duplicates, the results of queries can be multisets. However, some SQL query operations remove duplicates, thereby producing sets.

In partiular, the set operations UNION, INTERSECT, EXCEPT do remove duplicates. Hence

```
SELECT * FROM table
  UNION
SELECT * FROM table
```

has the same effect as

```
SELECT DISTINCT * FROM table
```

and may result in a table with less rows than the original table.

Summarization pitfalls

There are several ways to perform meaningless summarization. Consider the following table, which records currency exchange events for a bank. Each event has information of money exchange transaction: how much money was exchanged, what was the rate used, and how much money did the client receive.

event_id	**date**	**currency1**	**currency2**	**give**	**rate**	**receive**
101	2018-11-1	EUR	GBP	1500	1.12	1612
101	2018-11-1	USD	EUR	1500	1.20	1800
101	2018-11-2	EUR	GBP	1000	1.12	1120

We may be interested in how much money was exchanged on a day. However, summing up as follows is not a good idea.

```
SELECT SUM (give)
FROM exchange
GROUP BY date
```

Why? Because the values of changed are given in different *units* (currencies). Thereby, blindly summing them up does not make sense. We can get meaningful sums by either adding currency1 to GROUP BY part, or select only rows with a given currency, such as euros.

Another obviously meaningless summarization would be summing up the dates - that, however, the SQL database is unlikely to accept. This might theoretically have some meaning if dates started from "the beginning of the world",

that is, they would have a natural zero point. Since they do not, summing up is not acceptable.

How about summing up the rates? It is hard to imagine that it would make any sense either. They also have different units. But wait - some of them have the same unit, for instance the ones which are used to convert euros to Swedish crowns. But summing them up does not seem like a great idea either - why? They are by nature multipliers that make little sense without accompanying values so summing them alone gives results that make no sense.

In many cases where sum is not meaningful average is. For instance, the average exchange rate makes sense where the sum of exchange rates does not.

There are also summarizations that theoretically could make sense, but in practice the application is hard to find. For instance, you may sum up the heights of a group of people, which could make sense e.g. in buying a yoga mat roll for them, but in general it would be hard to imagine reasonable uses for such a summation.

Most SQL databases would happily perform most of the above summarizations, since they have no metadata that could describe units and scales.

3 Conceptual modeling using Entity-Relationship diagrams

Entity-Relationship (E-R) diagrams are a popular way to communicate database design to stakeholders, who don't necessarily have technical knowledge of databases. E-R diagrams specify the connections between dataobjects and attributes in a graphical fashion, showing how they are related to each other and what depends on what. In this way, they can also help the technically knowledgeable database designers to get a clear view of the database structure. In fact, it is possible to generate a database schema automatically from an E-R diagram, specifying all tables, attributes, primary keys, and foreign keys. In this chapter, we start with an intuitive explanation of E-R diagrams and proceed to showing how database schemas are derived from them.

3.1 Introduction

The projects where databases are used typically have many stakeholders. Consider, for instance, selling concert tickets online. You have to take into account the customers who buy the tickets, but they are not normally participating in designing the system. Concert organizers should obviously be participating, even though they might not have technical knowledge about how such systems are built. The same concerns business experts, whose skills are needed to understand how concerts are arranged, when tickets should be on sale, how to manage cancellations, and so on.

The technical knowledge is provided by computing professionals. These professionals need to understand the different roles of the users of the system, as well as how the system connects to other information systems, such as payment services. Professionals are also needed to design the user interface, and, of course, to design and implement the database that holds the information about concerts, customers, and reservations.

The related data content needs to be communicated between stakeholders. Since some of them may be stricly non-technical people, the SQL language is not the ideal way to communicate the content. Typically, a description on a higher, conceptual level is appropriate. In this chapter, we will introduce a technique called **Entity-Relationship (E-R) modeling** to describe the data content. This method is simple to understand, which has been the key to its success. The model has a graphical representation, and the E-R models are often called **E-R diagrams**.

A relational database, as we have seen, consists of a set of tables, which are linked to each other by referential constraints (foreign keys). This is a simple model to implement and flexible to use. But designing a database directly as tables can be hard, because only some things are "naturally" tables; some other things are more like relationships between tables, and might seem to require a more complicated model. E-R modelling is a richer structure than just tables, but it can in the end be automatically converted to a table design.

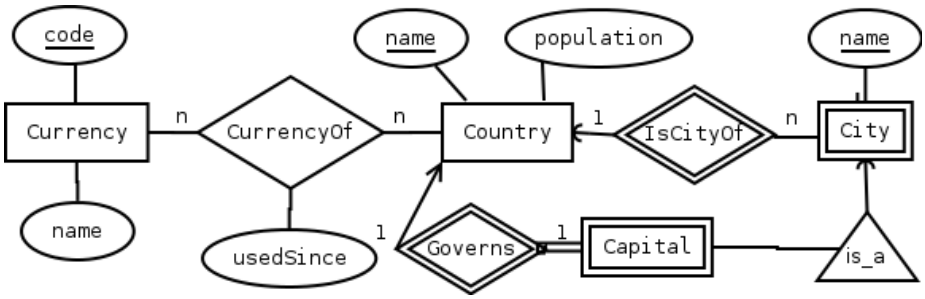


Figure 4: An example E-R diagram

Since it is easier than building tables directly, it helps design databases with right dependencies.

In this chapter, we will learn to model the data content as E-R diagrams. We will also tell how E-R diagrams can almost mechanically be derived from descriptive texts. Finally, you will learn how they are, even more mechanically, converted to relational schemas, and thereby eventually to SQL `CREATE TABLE` statements.

In this chapter, we can just give the bare bones of E-R models. Building them in practice is a skill that has to be practised. This practice is particularly suited for work in pairs or small groups, discussing the pros and cons of different modeling choices. Sometimes there are many models that are equally good, but a model that first looks good may finally not be so good if you take everything into account. Taking everything into account is easier if many people are contributing.

3.2 Entities and relationships: a complete example

Figure 4 shows an E-R diagram with countries, cities, and currencies. Intuitively, countries, cities and currencies are **entities** in the real world. In E-R diagrams, entities are represented as rectangles.

Entities have **attributes**, which are drawn as ellipses connected to entities. For example, the entity Currency has the attributes code and name. The code attribute is underlined, which means that it is the **key** of the currency entity.

For those familiar with SQL tables, the above description readily suggests that entities correspond to tables, and attributes are the names of their columns. But let us postpone the precise correspondence to a later section, and continue to other components of E-R diagrams, which are a bit less straightforward to relate to SQL.

Entities are connected by **relationships**, drawn as diamonds. For example, "being a currency of" is a relationship between currencies and countries. Relationships can have their own attributes, here exemplified by the "usedSince", the date since which a currency is used in a country.

The relationship between a currency and a country represents a basic rela-

tionship type. In the figure we do not require that a currency is always used in a country, and not even that a country always needs to have a currency (there are indeed countries that only use foreign currencies). As we will see later, the diagram permits a country to have several currencies, taken into use at different dates.

Some of the entities and relationships are drawn with double contours. An example is City, which is related to Country with the double-contour relationship isCityOf. Such entities and relationships are called **weak**. The idea is that a weak entity can only be identified by a relationship to a supporting entity. For example, the name *London* does not in itself identify a city, because there is a London in U.K. and another London in Canada. The full reference to a city must thus be *London, UK* or *London, Canada* (which in reality is called *London, Ontario*, but our simplified diagram does not show states within countries). The reader might notice that this pair of attributes correspond to a composite key, where the country name is a foreign key.

Another weak entity in our example is Capital: a capital is always related to some country, and cannot exist without it. At the same time, it makes sense to require that a capital is a city in the country. For this purpose, we need yet another shape in the diagram: an **ISA** relationship between Capital and City ("a capital *is a* city"). These relationships are marked with triangles.

Notice, finally, what is *not* possible in E-R diagrams:

- there are no connecting lines directly between entities, or between relationships
- there are no relationships between relationships
- weak relationships and ISA relationships have no attributes

3.2.1 Relationship participation types

The lines in Figure 4 between entities and relationships have further markings: arrowheads and numbers 1 and n. They express the **multiplicity** of relationships: whether it relates just one or more entities of a kind, or maybe none. For instance, a country can have several currencies, and one and same currency can be used in several countries. This is indicated by the number n and at the same time, redundantly, by headless arrows (just straight lines) from CurrencyOf to Currency and Country.

The relationship IsCityOf between City and Country is different: each city belongs to exactly one country whereas a country may have several cities. The number 1 expresses "one country", and the same thing is expressed by the arrowhead pointing to Country. Further, the relationship between country and capital is shown as one-to-one in the diagram, stating that each country has only one capital and one capital is the capital of only one country. (This is not quite true in the real world, because for instance South Africa has three capitals.)

We use, redundantly, two ways to show the multiplicity in Figure 4. One is by writing 1 or n (or m) above the connecting lines. Notice the placement of these numbers: since there is 1 country for each city, there is a 1 on the line

to country, and since there are n cities, where n is assumed to be an arbitrary non-negative integer, in a country, there is an n on the line to the city. The marking is similar in other lines, apart from the is-a relationship which does not need this specification.

The other property is totality, which means that an entity always has to participate in the relationship. We can safely say that a capital must be a capital of a country, so there is total participation. Totality is marked with a double line on that participant's side.

We have also drawn, redundantly, another way to mark similar information. If there is no arrow at the end of the line, then there is multiple participation. If there is a sharp arrow at the end of the line, then there is a total single participation (see Capital and Country). If there is a round arrowhead, then there is a partial single participation (see City and Country).

3.2.2 Composite and multivalued attributes

SQL is a standardized language. The standard is not completely followed in all database products, but at least to a large extent. The graphical E-R modeling language is not standardized, and many dialects exist. The different ways to mark multiplicity is an example of this. We introduce here one particular extension that has important implications.

In Figure 4 the attributes are non-composite, in the style of the database model. However, on a conceptual model it makes sense to model things like address being composed of things like streetname and number, etc. In Figure 5 we see an example utilizing composite attributes. There, a person has an address that is composed of several further attributes, one of them being street address, which is further composed of street name etc.

Being able to use composite attributes to make up an address helps in understanding how addresses are formed, even though one may choose to store them as address lines or something like that in the database. However, the implication for the keys is different: A composite key is not made up by underlining several attributes, but from the attributes that make up the composed underlined attribute. This changes the reasonable semantics of the diagram: when composite attributes *cannot exist*, each underlined attribute is a part of the primary key, and when composite attributes *can exist*, each underlined attribute forms a separate key, coming from the composing attributes.

In SQL databases, on one row each attribute has just a single value. In E-R diagrams it is possible to say that there are several values for an attribute, like a person may have several telephone numbers. This is represented as a double-lined ellipsis, like the phone number (phone_no) in Figure 5.

The representation in Figure 4 follows the original representation of E-R diagrams. It has an obvious drawback: the attributes take a lot of space, and when the diagrams get too big, the space taken by them is just too much. A simple way to optimize is to draw the diagram without attributes. In a design tool, the attributes may be accessed e.g. by clicking on the entity or relationship.

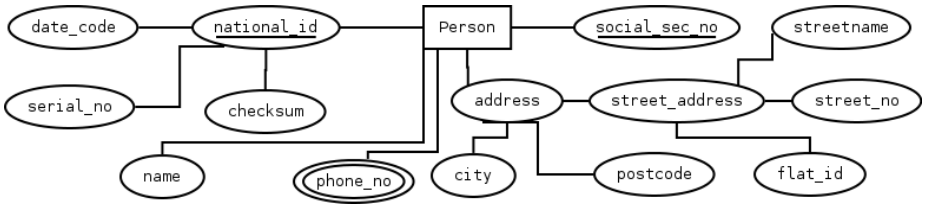


Figure 5: Composite and multivalued attributes example

3.3 E-R syntax

Standard E-R models have six kind of elements, each drawn with different shapes:

entity	rectangle	a set of independent objects
relationship	diamond	between 2 or more entities
attribute	oval	belongs to entity or relationship
ISA relationship	triangle	between 2 entities, no attributes
weak entity	double-border rectangle	depends on other entities
supporting relationship	double-border diamond	between weak entity and its supporting entity

Between elements, there are connecting lines:

- a relationship is connected to the entities that it relates
- an attribute is connected to the entity or relationship to which it belongs
- an ISA relationship is connected to the entities that it relates
- a supporting relationship is connected to a weak entity and another (possibly also weak) entity

Notice thus that there are no connecting lines directly between entities, or between relationships, or from an attribute to more than one element. The ISA relationship has no attributes. It is just a way to indicate that one entity is a **subentity** of another one. By subentity we mean an entity that represents a subtype of the other entity, such as capital being a subtype of city, and thereby the entity capital being is a subentity of the entity city.

The connecting lines from a relationship to entities can have arrowheads:

- sharp arrowhead: the relationship is to/from at most one object
- round arrowhead: the relationship is to/from exactly one object
- no arrowhead: the relationship is to/from many objects

The attributes can be underlined or, equivalently, prefixed by _. This means, precisely as in relation schemas, that the attribute is a part of a **key**. The keys of E-R elements end up as keys and referential constraints of schemas.

Here is a simple grammar for defining E-R diagrams. It is in the "Extended BNF" format, where `+` means 1 or more repetitions, `*` means 0 or more, and `?` means 0 or 1.

```
Diagram ::= Element+
```

```
Element ::=
   "ENTITY"        Name                 Attributes
 | "WEAK" "ENTITY" Name Support+        Attributes
 | "ISA"           Name SuperEntity     Attributes
 | "RELATIONSHIP"  Name RelatedEntity+  Attributes

Attributes ::=
   ":" Attribute*            # attributes start after colon
 |                           # no attributes at all, no colon needed

RelatedEntity ::= Arrow Entity ("(" Role ")")?   # optional role in parentheses
Support ::= Entity WeakRelationship

Arrow     ::= "--" | "->" | "-)"
Attribute ::= Ident | "_" Ident
Entity, SuperEntity, Relationship, WeakRelationship, Role, Name ::= Ident
```

Representing the E-R diagrams using a grammar has its benefits:

- it defines exactly what combinations of elements are possible, so that you can avoid "syntax errors" (i.e. drawing impossible E-R diagrams), and
- it can be used as input to programs that do many things: not only draw the diagrams but also convert the model to other formats, such as database schemas and even natural language descriptions.

Notice that there is no grammar rule for an Element that is a supporting relationship. This is because supporting relationships can only be introduced in the Support part of weak entities.

3.3.1 Composite attributes

We discussed the composite attribute extension to the E-R diagrams above. They require modifications to the grammar. We can do this by extending the `Attribute` syntax with composite attributes in parentheses:

```
Attribute ::=
    "(" PrimitiveAttribute Ident* ")"
  | PrimitiveAttribute

PrimitiveAttribute ::=
        Ident Ident*
  | "_" Ident Ident*
```

For clarity, the primitive attributes within a composed attribute are always non key attributes, however the composite attribute itself may be a key attribute.

3.4 From description to E-R

The starting point of an E-R diagram is often a text describing the domain. You may have to add your own understanding to the text. The expressions used in the text may give clues to what kind of elements to use. Here are some typical examples:

entity	CN (common noun)	"country"
attribute of entity	the CN of X	"the population of X"
attribute of relationship	adverbial	"in 1995"
relationship	TV (transitive verb)	"X exports Y"
relationship (more generally)	sentence with holes	"X lies between Y and Z"
subentity (ISA)	modified CN	"EU country"
weak entity	CN of CN	"city of X"

It is not always the case that just these grammatical forms are used. You should rather try if they are usable as alternative ways to describe the domain. For example, when deciding if something is an attribute of an entity, you should try if it really is *the* something of the entity, i.e. if it is unique. In this way, you can decide that *the population* is an attribute of a country, but *export product* is not.

You can also reason in terms of the informal semantics of the elements:

- An entity is an independent class of objects, which can have properties (attributes) as well as relationships to other entities.
- An attribute is a simple (atomic) property, such as name, size, colour, date. It belongs to only one entity.
- A relationship states a fact between two or more entities. These can also be entities of the same kind (e.g. "country X is a neighbour of country Y").
- A subentity is a special case of a more general entity. It typically has attributes that the general entity does not have. For instance, an EU country has an attribute "EU joining year", which countries generally do not have.
- A weak entity is typically a part of some other entity. Its identity (i.e. key) needs this other entity to be complete. For instance, a city needs a country, since "Paris, France" is different from "Paris, Texas". The other entity is called **supporting entity**, and the relationships are **supporting relationships**. If the weak entity has its own key attributes, they are called **discriminators** (e.g. the name of the city).

3.5 Converting E-R diagrams to database schemas

The standard conversions are shown in Figure 6. The conversions are unique for ordinary entities, attributes, and many-to-many relationships.

- An entity becomes a table with its attributes and keys just as in E-R.

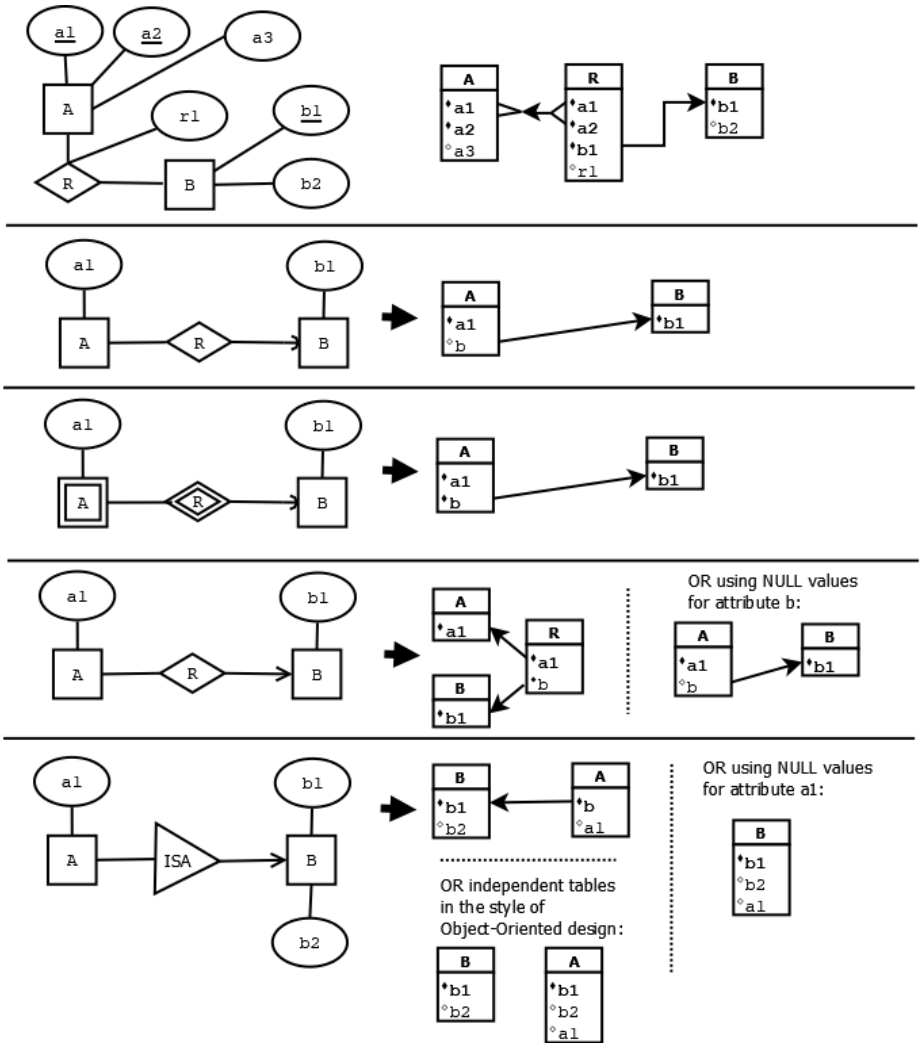


Figure 6: Translating E-R diagrams to SQL database schemas.

- A relationship becomes a table that has the key attributes of all related entities, as well as its own attributes.

Other kinds of elements have different possibilities:

- In exactly-one relationships, one can leave out the relationship and use the key of the related entity as attribute directly.
- In weak entities, one likewise leaves out the relationship, as it is always exactly-one to the strong entity.
- An at-most-one relationship can be treated in two ways:
  - the NULL approach: the same way as exactly-one, allowing NULL values
  - the (pure) E-R approach: the same way as to-many, preserving the relationship. No NULLs needed. However, the key of the related entity is not needed.
- An ISA relationship has three alternatives.
  - the NULL approach: just one table, with all attributes of all subentities. NULLs are needed.
  - the OO (Object-Oriented) approach: separate tables for each subentity and also for the superentity. No references between tables.
  - the E-R approach: separate tables for super-and subentity, subentity refers to the superentity.
- If multivalued attributes are used, then one solution is to agree on a maximum, like one person having max 4 telephone numbers, or creating a new table with entity key and multivalued attribute value pairs.
- With composite attributes, only the bottom level attributes are to be stored in the database (like street name of a street address) and the rest can be left out.

As the name might suggest, the E-R approach is generally recommended. It is the most flexible one, even though it requires more tables to be created.

One more thing: the naming of the tables of attributes.

- Entity names could be turned from singular to plural nouns.
- Attribute names must be made unique. (E.g. in a relationship from and to the same entity).

We find it more intuitive to use singular nouns for entities, plural nouns for tables. The reason is that an entity is more like a kind (type), whereas a table is more like a list. Also a term **entity set** is used, meaning the set of entities of the given kind.

3.6 A word on keys

When designing an E-R model, we are making choices that effect what kind of keys are being used in the tables. There is nothing that requires that all

relations must have singleton keys. It may be that the only natural key of a relation includes all attributes in a composite key.

Since many keys are in practice used as foreign keys in other relations, it is highly desirable that their values do not change. The key values used as foreign keys are also stored many times and included in search data structures. For these reasons, it is often more simple and straightforward in practice to create **artificial keys** that are usually just integers.

For instance, Sweden has introduced a system of "person numbers" to uniquely identify every resident of the country. Artificial keys may also be automatically generated by the system internally and never shown to the user. Then they are known as **surrogate keys**. The surrogate keys are guaranteed not to change, whereas natural values, no matter how stable they seem, might do that. Another related issue is that keys are used to compose indices for the data, used in joins, and one may not like to grow these structures unnecessarily.

4 Relational data modelling

Modelling data with E-R diagrams gives a conceptual description of the data, which can be used to produce a tentative database design. However, SQL and E-R models are not in a one-to-one correspondence, and we cannot use E-R diagrams to define all the details of our data. SQL provides more details, but it is unnecessarily complicated for reasoning about our data design and data. This reasoning is important for both studying the properties of our database design and in checking the integrity of data. Powerful mechanisms for reasoning are provided by mathematics and logic, and to utilize those mechanisms we need to describe our data using mathematics and logic. The ***relational model*** *of data, based on set thery, is the tool of choice for this purpose. With this chapter, you learn how to use the relational model to define your data as well as basic operations on the data. You will also learn the correspondence of the relational data model concepts to the expressions of SQL.*

4.1 Relations and tables

The mathematical model of relational databases is, not surprisingly, relations. Mathematically, a relation r is a subset of a **cartesian product** of sets:

$$r \subseteq T_1 \times \ldots \times T_n$$

The elements of a relation are **tuples**, which we write in angle brackets:

$$\langle t_1, \ldots, t_n \rangle \in T_1 \times \ldots \times T_n \text{ if } t_1 \in T_1, \ldots, t_n \in T_n$$

In these definitions, each T_i is a set, called a **domain**. The elements t_i are the **values** (or components) of the tuple. The domains determine the types of the values of the tuple. The cartesian product of which the relation is a subset is its **signature**.

The most familiar example of a cartesian product in mathematics is the two-dimensional space of real numbers, $R \times R$. Its elements have the form (x, y), and the components are usually called the x-coordinate and the y-coordinate. A relation with the signature $R \times R$ is any subset of this two-dimensional space, such as the graph of a function, or a geometric figure such as a circle or a triangle. Mathematical relations are typically, but not necessarily, infinite sets of tuples.

In the database world, a relation is a finite set of tuples, and it is usually called a **table**. Tuples are called **rows**. We will use the words relation and table as synonyms as well as tuple and row, when we feel appropriate. Here is an example of a table and its mathematical representation:

country	**capital**	**currency**
Sweden	Stockholm	SEK
Finland	Helsinki	EUR
Estonia	Tallinn	EUR

$$\{\langle \text{Sweden}, \text{Stockholm}, \text{SEK}\rangle, \langle \text{Finland}, \text{Helsinki}, \text{EUR}\rangle, \langle \text{Estonia}, \text{Tallinn}, \text{EUR}\rangle\}$$

When seeing the relation as a table, it is also natural to talk about its **columns**. Mathematically, a column is the set of components from a given place i:

$$\{t_i \mid \langle \ldots, t_i, \ldots\rangle \in R\}$$

It is a special case of a **projection** from the relation. (The general case, as we will see later, is the projection of many components at the same time. The idea is the same as projecting a 3-dimensional object with xyz coordinates to a plane with just xy coordinates.)

What is the signature of the above table as a relation? What are the types of its components? For the time being, it is enough to think that every type is **String**. Then the signature is

$$\text{String} \times \text{String} \times \text{String}$$

However, database design can also impose more accurate types, such as 3-character strings for the currency. This is an important way to guarantee the quality of the data.

Now, what are "country", "capital", and "currency" in the table, mathematically? In databases, they are called **attributes**. In programming language terminology, they would be called **labels**, and the tuples would be **records**. Hence yet another representation of the table is a list of records,

```
[
 {country = Sweden,  capital = Stockholm, currency = SEK},
 {country = Finland, capital = Helsinki,  currency = EUR},
 {country = Estonia, capital = Tallinn,   currency = EUR}
]
```

Mathematically, the labels can be understood as **indexes**, that is, indicators of the positions in tuples. (**Coordinates**, as in the xyz example, is also a possible name.) Given a cartesian product (i.e. a relation signature)

$$T_1 \times \ldots \times T_n$$

we can fix a set of n labels (which are strings),

$$L = \{a_1, \ldots, a_n\} \subset \text{String}$$

and an **indexing function**

$$i : L \to \{1, \ldots, n\}$$

which should moreover be a bijection (i.e. a one-to-one correspondance). Then we can refer to each component of a tuple by using the label instead of the index:

$$t.a = t_{i(a)}$$

One advantage of labels is that we don't need to keep the tuples ordered. For instance, inserting a new row in a table in SQL by just listing the values without labels is possible, but risky, since we may have forgotten the order; the notation making the labels explicit is more reliable.

A **relation schema** consists of the name of the relation, the attributes, and the domains of the attributes:

```
Countries(country : String, capital : String, currency : String)
```

The relation (table) itself is called an **instance** of the schema. The types are often omitted, so that we write

```
Countries(country, capital, currency)
```

But mathematically (and in SQL), the types are there.

One thing that follows from the definition of relations as *sets* is that the order and repetitions are ignored. Hence for instance

country	**capital**	**currency**
Finland	Helsinki	EUR
Finland	Helsinki	EUR
Estonia	Tallinn	EUR
Sweden	Stockholm	SEK

is the same relation as the one above. As SQL implementations require the existence of a primary key, there cannot be any duplicates in the database tables. In queries SQL, however, makes a distinction, marked by the `DISTINCT` and `ORDER` keywords. This means that, strictly speaking, SQL tables resulting from queries are **lists** of tuples. If the order does not matter but the repetitions do, the tables are **multisets** or **bags**.

In set theory, you should think of a relation as a *collection of facts*. The first fact is that Finland is a country whose capital is Helsinki and whose currency is EUR. Repeating this fact does not add anything to the collection of facts. The order of facts does not mean anything either, since the facts don't refer to each other.

4.2 Functional dependencies

We will most of the time speak of relations just as their sets of attributes. We will define functional dependencies as semantic constraints between sets of attributes. But their definitions must in the end refer to tuples, whose values they constrain. By tuples, we will from now on mean labelled tuples (records) rather than set-theoretic ordered tuples. We will ignore the types of the columns here, as they do not play a role with functional dependencies.

Definition (projection of a tuple). If t is a tuple of a relation with schema S, and $\backslash subi\{A\}\{i\}$ is an element of S, the **projection** (attribute value) $t.A_i$ computes to the value v_i.

If X is a set of attributes $\{B_1, \ldots, B_m\} \subseteq S$ and t is a tuple of a relation with schema S, we can form a simultaneous projection,

$$t.X = \{B_1 = t.B_1, \ldots, B_m = t.B_m\}$$

Definition (functional dependency, FD). Assume X is a set of attributes and A an attribute, all belonging to a relation schema R, and r is a relation with schema R, We say that X **determines functionally** A in the relation r, if for all tuples t,u in r it holds that if $t.X = u.X$ then $t.A = u.A$.

We are more interested in functional dependencies as semantic constraints, and we say that X **determines functionally** A, written as $X \to A$, if X functionally determines A in all possible relations with schema R. If Y is a set of attributes, we write $X \to Y$ to mean that $X \to A$ for every A in Y. A FD $X \to A$ is **trivial** if $A \in X$.

As an example, consider the following table of countries, currencies, and values of currencies (in USD, on a certain day).

country	**currency**	**value**
Sweden	SEK	0.12
Finland	EUR	1.10
Estonia	EUR	1.10

In that table, currency functionally determines value, country determines all other attributes, and value determines currency. However, not all of these seem like valid semantic constraints.

Let us assume now that each country has always exactly one currency, and each currency has always exactly one value. This gives us two functional dependencies:

```
country -> currency
currency -> value
```

However, it clearly follows from the above, that for each country there is always exactly one value (the value associated with the one currency of the country), which logically follows from the functional dependencies above.

Normally, the set of FDs identified for an application is not exhaustive in the sense that there are FDs that can be inferred from that set while not explicitly belonging to the set. So, when a set of attributes X functionally determines an attribute A, it may not necessarily be an explicitly given FD, but an FD that can be inferred from the given FDs.

Given a set of FDs, the following rules can be used to correctly compute all implied FDs.

1. If $Y \subset X$, then $X \to Y$. (Trivial)
2. If $X \to Y$, then $XA \to YA$. (Adding an attribute to both sides)
3. If $X \to Y$ and $Y \to Z$, then $X \to Z$. (Transitivity)

Generally, we will not need to compute all implied FDs as there could be a huge amount of them. We are more interested in calculating the *closure* of a set of attributes X. The closure of X, denoted by X^+, is the set of all attributes that X functionally determines, including X. Given X, we can compute X^+ by starting with $X^+ = X$ and applying repatedly the following rule as long as new attributes can be added to X^+: Iff there is a FD $X' \to Y$ such that $X' \subset X^+$ computed this far, and Y contains attributes that are not yet in X^+, then add all those attributes of Y to X^+.

For example, the closure of the singleton attribute country is {country, currency, value} and it can be computed by applying the rule once to add currency and then once to add value.

4.3 Keys and superkeys

Definition (superkey, key). Let R be a relation schema. A set of attributes $X \subseteq R$ is a **superkey** of R, if $R \subseteq X+$.
A set of attributes $X \subseteq R$ is a **key** of R if

- X is a superkey of R, and
- no proper subset of X is a superkey of R.

Supppose that we want to know if X is a key or superkey of a relation schema R. It may be that the given set of FDs contains dependencies whose attributes are not in R. Then we can just compute the closure of X using all the given FDs and take the intersection of R with the result of the closure computation. For instance, consider a table with just country and value (of its currency), and suppose that the given set of FDs contains country $\to$ currency and currency $\to$ value. By calculating the closure of country and then intersecting it with {country, value}, we can find out that the closure of country contains country and value.
Let us return to the example of the table with attributes country, curreny, value, with FDs country $\to$currency, currency $\to$ value. Closure computation gives us:

```
country+ = {country, currency, value}
```

This makes country a superkey, and, as such, a candidate when finding keys. However, `country` is not the only superkey. All of

```
country
country currency
country value
country currency value
```

are superkeys. However, all of these sets but the first are just irrelevant extensions of the first. They are superkeys but not keys. We conclude that `country` is the only possible key.

4.4 Modelling SQL key and uniqueness constraints

In SQL, we define one key and additionally we may define a number of unique constraints. On the logical level, key and unique constraints are the same. So, both can be modelled in the same way using functional dependencies.

However, to create a mapping between a SQL database definition and the relational model, we need a way to identify the primary key. We do this by marking the primary key attributes either by underlining or (in code ASCII text) with an underscore prefix.

```
JustCountries(_name,capital,currency)
    currency -> Currencies.code

Currencies(_code,valueInUSD)
```

In this example, `name` and `code` work naturally as keys.

In `JustCountries`, `capital` could also work as a key, assuming that no two countries have the same capital. To match the SQL uniqueness constraint, we add to our model a similar statement:

```
JustCountries(_name,capital,currency)
    currency -> Currencies.code
    unique capital
```

In the actual database, the key and uniqueness constraints prevent us from inserting a new country with the same name or capital.

In `JustCountries`, `currency` would not work as a key, because many countries can have the same currency.

In `Currencies`, `valueInUSD` *could* work as a key, if it is unlikely that two currencies have exactly the same value. This would not be very natural of course. But the strongest reason of not using `valueInUSD` as a key is that we know that some day two currencies might well get the same value.

The keys above contain only one attribute, and as such, they are called **singleton** keys. A key can also be **composite**. This means that many attributes together form the key. For example, in

```
PostalCodes(_city,_street,code)
```

the city and the street together determine the postal code, but the city alone is not enough. Nor is the street, because many cities may have the same streetname. For very long streets, we may have to look at the house number as well. The postal code determines the city but not the street. The code and the street together would be another possible composite key, but perhaps not as natural.

4.5 Referential constraints

The schemas of the two relations above are

```
JustCountries(country,capital,currency)
Currencies(code,valueInUSD)
```

For the **integrity** of the data, we want to require that all currencies in `JustCountries` exist in `Currencies`. We add to the schema a **referential constraint**,

```
JustCountries(country,capital,currency)
  JustCountries.currency => Currencies.code
```

In the actual database, the referential constraint prevents us from inserting a currency in `JustCountries` that does not exist in `Currencies`. [3]

With all the information given above, the representations of SQL schemas with relations is now straightforward.

Example:

```
Countries (_name,capital,population,currency)
  capital => Cities.name
  currency => Currencies.code
```

represents

```
CREATE TABLE Countries (
  name TEXT,
  capital TEXT,
  population INT,
  currency TEXT,
  PRIMARY KEY (name),
  FOREIGN KEY (capital) REFERENCES Cities (name),
  FOREIGN KEY (currency) REFERENCES Currencies (code)
  )
```

4.6 Operations on relations

Set theory provides some standard operations that are also used in databases:

Union:	$R \cup S = \{t \mid t \in R \text{ or } t \in S\}$
Intersection:	$R \cap S = \{t \mid t \in R \text{ and } t \in S\}$
Difference:	$R - S = \{t \mid t \in R \text{ and } t \notin S\}$
Cartesian product:	$R \times S = \{\langle t, s\rangle \mid t \in R \text{ and } s \in S\}$

However, the database versions are a bit different from set theory:

- Union, intersection, and difference are only valid for relations that have the same schema.
- Cartesian products are **flattened**: $\langle\langle a, b, c\rangle, \langle d, e\rangle\rangle$ becomes $\langle a, b, c, d, e\rangle$

[3] It is common to use the arrow (`->`) as symbol for referential constraints. However, we chose to use the double arrow `=>` in order not to confuse with functional dependencies.

These standard operations are a part of **relational algebra**. They are also a part of SQL (with different notations). But in addition, some other operations are important - in fact, even more frequently used:

Projection: $\pi_{a,b,c}R = \{\langle t.a, t.b, t.c\rangle \mid t \in R\}$
Selection: $\sigma_C R = \{t \mid t \in R \text{ and } C\}$
Theta join: $R \bowtie_C S = \{\langle t, s\rangle \mid t \in R \text{ and } s \in S \text{ and } C\}$

In selection and theta join, C is a **condition** that may refer to the tuples and their components. In SQL, they correspond to `WHERE` clauses. The use of attributes makes them handy. For instance.

$$\sigma_{\text{currency=EUR}}\text{Countries}$$

selects those rows where the currency is EUR, i.e. the rows for Finland and Estonia.

A moment's reflection shows that theta join can be defined as the combination of selection and cartesian product:

$$R \bowtie_C S = \sigma_C(R \times S)$$

The $\bowtie$ symbol without a condition denotes **natural join**, which joins tuples that have the same values of the common attributes. It used to be the basic form of join, but it is less common nowadays. The history behind is that it is based on design philosophy where the attributes referencing the same thing are given the same name. This makes joining by identical names a feasible idea. Otherwise, of course, the idea is not that feasible. The definition is as follows:

$$R \bowtie S = \{t + \langle u.c_1, \ldots, u.c_k\rangle | t \in R, u \in S, (\forall a \in A \cap B)(t.a = u.a)\}$$

where A is the attribute set of R, B is the attribute set of S, and $B - A = \{c_1, \ldots, c_k\}$. The + notation means putting together two tuples into one flattened tuple.

An alternative definition expresses natural join in terms of theta join, which is left as an exercise. Thus we can conclude: natural join is a special case of theta join. Cartesian product is a special case of a join. Projection is needed on top of theta join to remove duplicated columns in the way that natural join does.

4.7 Multiple tables and joins

The joining operator supports dividing data into multiple tables. Consider the following table:
Countries:

name	**capital**	**currency**	**valueInUSD**
Sweden	Stockholm	SEK	0.12
Finland	Helsinki	EUR	1.09
Estonia	Tallinn	EUR	1.09

This table has a **redundancy**, as the USD value of EUR is repeated twice. As we will see later, redundancy should usually be avoided. For instance, someone might update the USD value of the currency of Finland but forget Estonia, which would lead to inconsistency. You can also think of the database as a story that states some facts about the countries. Normally you would only state once the fact that EUR is 1.09 USD.

Redundancy can be avoided by splitting the table into two:
JustCountries:

name	**capital**	**currency**
Sweden	Stockholm	SEK
Finland	Helsinki	EUR
Estonia	Tallinn	EUR

Currencies:

code	**valueInUSD**
SEK	0.12
EUR	1.09

Searching for the USD value of the currency of Sweden now involves a join of the two tables:

$$\pi_{\text{valueInUSD}}(\text{JustCountries} \bowtie_{\text{name=Sweden AND currency=code}} \text{Currencies})$$

In SQL, as we have seen in Chapter 2, this is expressed

```
SELECT valueInUSD
FROM JustCountries, Currencies
WHERE name = 'Sweden' AND currency = code
```

Several things can be noted about this translation:

- The SQL operator SELECT corresponds to projection in relation algebra, not selection!
- In SQL, WHERE corresponds to selection in relational algebra.
- The FROM statement, listing any number of tables, actually forms their cartesian product.

Now, the SELECT-FROM-WHERE format is actually the most common idiom of SQL queries. As the FROM forms the cartesian product of potentially many tables, there is a risk that huge tables get constructed; keep in mind that the size of a cartesian producs is the product of the sizes of its operand sets. The query compiler of the DBMS, however, can usually prevent this from happening by query optimization. In this optimization, it performs a reduction of the SQL code to something much simpler, typically equivalent to relational algebra code, and it then reorganizes the code something that gives the same answer but is more efficient to evaluate in a straightforward way. We will return to relational algebra and its optimizations in Chapter 6.

4.8 Transitive closure

The initial relational model was based on the idea of relational algebra as a measuring stick for the kinds of queries the user can pose. A language equally expressive as relational algebra was called *relationally complete*. However, using relational algebra it is not possible to query for transitive closure, discussed in this section. See following table, which we can call, say `requires`.

course	**requires_course**
programming	computing principles
programming languages	programming
compilers	programming languages

This relation is *transitive* in the sense that if course B is required for course A, and course C is required for course B, then course C is also required for course A.

Now, how can we compute all courses required for `compilers`? We can directly see, just by selection, that `programming languages` is required. Joining `requires` with itself, and taking a projection and a union, we get a tuple saying that also `programming` is required. A further join with projection and union to the previous results gives us tuples for all the required courses.

If we know exactly how many times it is enough to join a relation with itself, we can write a relational algebra expression that answers this question. If not, then it is not doable with relational algebra.

For transitive closure computation, let us consider a transitive relation `R` that has only two attributes, the first being called `A` and the second `B`. Thus, it only contains one transitive relation and no additional attributes. The following *relation composition* for two-attribute relations may be used in search of new transitive relationships:

$$R \cdot R = \{\langle t.A, s.B\rangle | t \in R, s \in R, t.B = s.A\}$$

We can compute the transitive closure, denoted by `R+`, by repeatedly replacing R by $R \cup (R \cdot R)$ as long as the replacement adds new tuples.

If our relational algebra contains attribute renaming, then we can implement relation composition using join, projection, and attribute renaming.

To implement a practical transitive closure, the following things need to be considered:

- There may be more than one transitive relationship in a relation as well as more attributes, and the attributes to be used need to be specified.
- Some transitive relationship may not be just from A to B but also at the same time from B to A. As an example consider a table where each tuple contains two countries sharing a border and the order is unimportant.
- There may be some other values we want in the result apart from just the transitive relationship attributes, and their computation needs to be specified.

- In some cases the transitive relation may be split into different relations. It is possible to join them first into a single table to avoid this situation.

SQL did not initially contain a transitive closure operation. This lead to the situation that different SQL implementations started to contain different ways to specify transitive closure. For example, PostgreSQL provides a special kind of local `WITH` definitions, `WITH RECURSIVE`, to answer transitive queries.[4]

4.9 General constraints on data

There are various very natural constraints on data that we cannot describe using the methods discussed this far. These include simple value-set constraints such as *Population must be non-negative*, and *The population of a city cannot be bigger than the population of the country to which the city belongs to.*

These types of constraints can simply be expressed as queries that retrieve the violating data. For instance, requiring the population to be non-negative means that the answer set to the following query must be empty. If it is not, the answer set also contains data violating the constraint. In the case of non-negative population, the data (if any) violating the constraint can be retreived with:

$$\sigma_{\text{population}< 0}(\text{Countries})$$

The case of limiting the city population would be

$$\sigma_{\text{city.population}> \text{ country.population}}(\text{Countries} \bowtie_{\text{country.Name}=\text{city.country}} \text{Cities})$$

The SQL standard includes a possibility to define *assertions* that are SQL queries stating conditions that the data needs to obey. The SQL assertions follow the idea that one can generally query whatever SQL allows as WHERE part of queries. The above examples can be represented using SQL CHECK constraings as follows.

```
CREATE ASSERTION nonnegative_population AS
CHECK (
  NOT EXISTS
  SELECT FROM Countries WHERE population < 0)

CREATE ASSERTION city_population_less_than_country AS
CHECK (
  NOT EXISTS
  SELECT FROM Countries, Cities WHERE Countries.name = Cities.country
         AND Countries.population < Cities.population)
```

The assertions may also be written with the focus on a single database row (referring to its values). Further, there are natural constraints involving aggregation functions to express things like *a country's population must be at least the sum of its cities' populations.*

[4] http://www.postgresqltutorial.com/postgresql-recursive-query/

The generality of assertions comes with a cost. They may involve frequent heavy computation and therefore they are generally not in use in actual database systems.

Even though the assertions are not used in practice, it makes perfect sense to describe these general constraints on data. They can be taken care with other solutions, to be discussed in later chapters. If they are not described, they are easily forgotten.

4.10 Multiple values

The guiding principle of relational databases is that all types of the components are **atomic**. This means that they may not themselves be tuples. This is what is guaranteed by the flattening of tuples of tuples in the relational version of the cartesian product. Another thing that is prohibited is list of values. Think about, for instance, of a table listing the neighbours of each country. You might be tempted to write something like

country	**neighbours**
Sweden	Finland, Norway
Finland	Sweden, Norway, Russia

But this is not possible, since the attributes cannot have list types. One has to write a new line for each neighbourhood relationship:

country	**neighbour**
Sweden	Finland
Sweden	Norway
Finland	Sweden
Finland	Norway
Finland	Russia

The elimination of complex values (such as tuples and lists) is known as the **first normal form**, 1NF. It is nowadays built in in relational database systems, where it is impossible to define attributes with complex values. The database researchers have studied alternative models, the so called Non-First-Normal-Form (NFNF) models, where relations are allowed to contain relations as attributes.

4.11 Null values

Set theory does not have a clear meaning for NULL values, and relational database theory researchers have studied different variations for representing for missing values. On this basis, we do not discuss formal modeling concept for them. However, this omission does not have any severe implications.

5 Dependencies and database design

The use of E-R diagrams implies a design that can automatically be produced from the E-R design. But is it a good design? Are we even able to identify a good design before we use the database? There are usually many open questions on the use of the database: the data volume and distributions, the queries, the update activity, the number of concurrent users, etc. Even though there could be initial estimates for this, the lifetime of database is often long and new ways to use the database emerge.

We would preferably want to find general criteria for good design and methods to avoid potentially problematic designs in a general setting. We need intuitive design goals and formal properties of good database design, and a methodology to create such good designs.

Our approach is mainly based on the use of **functional dependencies**. They give us a way to formalize some of the properties of good design. They also give methods and algorithms to carry out the design. Functional dependencies are independent of E-R diagrams, but they also add expressive power: they allow us to define relationships between attributes that E-R diagrams cannot express. In this chapter, we will define the different kinds of dependencies, study their use, and see how they combine with E-R diagrams.

Otherwise, there is a risk of **redundancy**, repetition of the same information (in particular, of the same argument-value pair). Redundancy can lead to **inconsistency**, if the updates fail to update all copies of the data, which is known as an **update anomaly**. To avoid such inconsistency in a badly designed database, it may be necessary to execute an extensive amount of database updates as a knock-on effect of a single update. This could be automated by using triggers (Section 7.4.4). But it is better to design the database in a way that avoids the problem altogether.

5.1 Design intuition

To improve our intuition, let us first illustrate some examples of potentially problematic designs. Consider, again, the table below on professor who teach students and on which course. Is this design problematic? This depends on the semantic constraints, ie. the functional dependencies. Suppose, first, that

professor	**course**	**student**
Ranta	Compiler construction	Anders
Ranta	Compiler construction	Lena
Kemp	Data Science	Anders
Nummenmaa	Database systems	Lena

Consider, first, splitting this table into two as follows. The first table tells who is teaching which student and the second tells who is studying which course.

professor	student
Ranta	Anders
Ranta	Lena
Kemp	Anders
Nummenmaa	Lena

course	student
Compiler construction	Anders
Compiler construction	Lena
Data Science	Anders
Database systems	Lena

If we join these table on students, we get the following table.

professor	course	student
Ranta	Compiler construction	Anders
Ranta	Compiler construction	Lena
Ranta	Data Science	Anders
Ranta	Database systems	Lena
Kemp	Data Science	Anders
Kemp	Compiler construction	Anders
Nummenmaa	Compile construction	Lena
Nummenmaa	Database systems	Lena

This is obviously not what we want. The table includes rows that were not in our original table. We say that the way to join the data is "lossy" - however lossy does not mean losing data, which indeed did not happen, but losing information on which pieces of data belonged together.

Our second attempt is to store the data with two tables, where the first table tells who is teaching which course and the second tells who is studying which course.

professor	course
Ranta	Compiler construction
Kemp	Data Science
Nummenmaa	Database systems

course	student
Compiler construction	Anders
Compiler construction	Lena
Data Science	Anders
Database systems	Lena

If we join the tables on book, we get the original table back. In this case our join is "lossless" - no information was lost. In this case, though, the losslessness is based on the fact that no course is being taught by two professors. If e.g.

Nummenmaa starts to teach Data Science to Kati, then the situation with losslessness changes. If, however, we know that each professor always only teaches one course, then we will be safe with joining on course and we always get a lossless join. In this case our database design would have the lossless join property.

Our initial design with just one table had - trivially - no problems with losslessness. Why would we not just stick with it? If we know that each professor only teaches one course, then using just one table usually introduces redundant information that is repeated in different rows. Let's assume, again, that each course is only taught by one professor. If all data is stored in just one table and the professors Ranta and Nummenmaa swap courses so that Nummenmaa will teach compile construction and Ranta will teach Database systesm, then we would need to update a whole lot of rows in the tables (well, in our initial example just 3, but potentially). If the data is split into two tables with the lossless design, then only the two rows in (professor,course) table need to updated, no matter how many students study these courses. So, the solution where the table is split seems beneficial.

It seems, right enough, that the key structure plays a central role in solving these problems. But as a modeling construct, it is not enough. Suppose, now, that each student studies each course with one particular professor, and let us suppose the following table, with (student, course) as the key.

professor	**course**	**student**
Ranta	Compiler construction	Anders
Ranta	Compiler construction	Lena
Kemp	Data Science	Anders
Nummenmaa	Database systems	Lena
Nummenmaa	Distributed systems	Anders

This would mean that it is impossible to add a row where Anders studies Compile construction under the instruction from Nummenmaa.

Let us make a further assumption that each course is only taught by at most one professor. There is no way we can model this information with keys, since course cannot be a key: then for example Ranta would be a duplicate key value. Also, in our table we have redundant information on who is teaching which course, since the course is always repeated with the professor.

We may try to get rid of the redundancy by splitting the table into two, as follows.

professor	**course**
Ranta	Compiler construction
Kemp	Data Science
Nummenmaa	Database systems

professor	**student**
Ranta	Anders
Ranta	Lena
Kemp	Anders
Nummenmaa	Lena

We got rid of the redundant data, but other problems appeared. Joining the tables on professor we get e.g. a row (Kemp, Anders, Data Science)

Let us now consider a design where we have three tables

professor	**course**
Ranta	Compiler construction
Kemp	Data Science
Nummenmaa	Database systems

professor	**student**
Ranta	Anders
Ranta	Lena
Kemp	Anders
Nummenmaa	Lena

course	**student**
Compiler construction	Anders
Compiler construction	Lena
Data Science	Anders
Database systems	Lena

This effectively eliminates redundant data. Let us suppose now that the only key-like constraint is that each student studies each course with exactly one professor. Then, when updates take place, we can only check this constraint by joining the tables. Besides, as the reader is urged to check, the joins are not lossless. Lossy joins follow from the fact that there are no foreign keys.

If no functional dependencies exist, then the three-table desing is lossless and there is no constraint checking problem. However, then intuitive querying becomes more problematic as there are multiple relationships between attributes. E.g. there is a relationship between course and student directly stored, but another one exists and it can be realized by joining the two other tables. Further, in some high-level query systems, as e.g. using natural language, the user may want to just ask for professors and students. With two existing interpretations, there is some ambiguity.

In addition to removing redundancy, we wish to preserve the possibility to maintain dependency-wise the correctness of the database, and if we split a table into several tables, we want to be able to recover the initial table as it was. These properties are defined formally below.

Definition (Dependency preservation) We call the database schema D **dependency preserving** with respect to a set of functional dependencies FD, if

we can infer any given functional dependency in FD from the dependencies in the relations of D.

Definition (Lossless join property). Let $D = \{R_1, R_2, \ldots, R_n\}$ be a database schema using the set of attributes U. We say that D has the *lossless join property* if every relation r_U over U decomposes losslessly onto $R_1, R_2, \ldots, R_n$. If D has the lossless join property, we say that D is *lossless*. Otherwise, we say that D is *lossy*.

These two concepts are related: it is known that a dependency preserving database schema D (i.e. set of relations) is lossless if and only if it contains a relation schema R such that closure of R contains all attributes in D.

We want to find design methods that produce database schemas which imply these properties. In what follows, it is important that the reader has a good recollection of functional dependencies, their closures, and keys.

5.2 Normal forms, synthesis and decomposition

When creating a database schema from the Entity-Relationship design, we form a set of tables with keys and non-key attributes. When we form a table R with key attributes X and other attributes Y, we have in fact identified a functional dependency $X \to Y$, and then made it into a table containing the attributes $X \cup Y$ and key X.

The Entity-Relationship diagrams are simple because we have limited possibilities to express functional dependencies. We can only express the keys of the entities and from them we compose the keys for the tables. If we allow subattributes of attributes, then we can also express alternative keys for entities, but no more than that.

Let us explore classic options for database design. The first is the *synthesis* algorithm, based on the idea of synthesizing relation schemas from dependencies - basically what we are doing with E-R diagrams.

In the general case, we want to remove certain type of *redundancy* from the set of FDs. We do this by repeating these two steps until neither of them changes the set of FDs.

1. If a functional dependency $X \to Y$ is such that there is an attribute A in X so that $X \setminus \{A\} \to Y$, then replace X by $X \setminus \{A\}$.
2. If a functional dependency $X \to Y$ is such that when computing X^+ (the closure of X) without using the dependency $X \to Y$, then remove $X \to Y$.

What we get is called a *minimal cover*: we have removed unnecessary or redundant dependencies and redundant left hand side attributes of functional dependencies. It is easy to see that the computation is efficient as closure is simple to compute iterating the dependency set, and increases until the computation stops. Similarly if we try to remove each attribute at a time from the left hand sides, then there on one round we just consider each left hand side attribute once and there are less rounds than the sum of numbers of attributes.

After that we compute the database schema as follows.

Algorithm (3NF Synthesis). Input is a minimal cover F of functional dependencies over a set of attributes U.

1. Group F by the left hand side, i.e. so that dependencies $X \to A$ and $X \to B$ end up in the same group.
2. For each of the groups, return the schema containing the left hand side of the group X and all the right hand side attributes, ie if $A_1 \dots A_n$ where $X \to A_i$.
3. (Optionally:) If one of the schemas contains a key of U, these groups are enough; otherwise, add a schema containing just some key of U.

Synthesis produces database schemas in which all relation schemas are in *third normal form* (3NF), defined as follows.

A relation schema R is in 3NF if all dependencies $X \to A$ in R, where A is a single attribute, are such that either X is a key of R or A belongs to some key of R.

Let us consider the dependencies we get from E-R diagrams. If an attribute appears in more than one relation schema, it is a part of a key in all schemata. This means that forming relation schemata directly from the E-R diagram produces a relation schema in 3NF (with respect to the dependencies defined in the E-R diagram). Further, the dependency set that we get from the E-R diagram is a minimal cover. We leave the rest of the reasoning of this for the reader, but just notice that the following are equal:

- Producing the relation schema directly from the E-R diagram using the methodology of this book.
- Producing the dependency set directly from the E-R diagram and synthesizing it into a database schema, without Step 3.
- Computing a minimal cover to the set of dependencies directly obtained from the E-R diagram and then synthesizing the database schema from there, without Step 3.

In fact, when we are dealing with a database schema transformed directly from an E-R diagram, and no other functional dependencies exist, then our database schema is in Boyce-Codd Normal Form (BCNF), defined as follows.

A relation schema R is in BCNF if all dependencies $X \to A$ in R that belong to a minimal cover are such that X is a key of R. That is, in BCNF we allow only key dependencies. As a simple consequence of our definition, all relation schemas with at most two attributes are in BCNF.

BCNF is a beneficial property, as relations whose schemas are in BCNF avoid redundancy. However, not all functional dependencies can be described in the E-R diagram. As an example, let us consider addresses, which are commonly stored in information systems. A part of an address is commonly formed from a city name, street name, and postcode. Here is a table with cities, streets, and postcodes.

city	**street**	**postcode**
Gothenburg	Framnäsgatan	41264
Gothenburg	Rännvägen	41296
Gothenburg	Hörsalsvägen	41296
Gothenburg	Barnhusgatan	41102
Stockholm	Barnhusgatan	11123

Here is the relation schema with functional dependencies:

```
city street postcode
city street -> postcode
postcode -> city
```

If we use E-R diagrams and model an entity with an address, which contains attributes city, street, and postcode, then we have no way to model these dependencies in an E-R diagram. The keys or the relation are city,street and postcode,street. It is easy to check that the relation schema is in 3NF. It seems to introduce redundancy, though, as postcode,city pairs are stored multiple times. This redundance is because of the FD `postcode -> city`, and that FD is not a key FD. It follows that the table is not in BCNF.

Since the City is functionally determined by postcode, we could think of splitting the table by the FD `postcode -> city`by a *decomposition* operation, that would make a new table with the attributes of the FD and remove the right hand side of the FD from the initial table, thereby giving us two tables as follows.

street	**postcode**
Framnäsgatan	41264
Rännvägen	41296
Hörsalsvägen	41296
Barnhusgatan	41102
Barnhusgatan	11123

city	**postcode**
Gothenburg	41264
Göteborg	41102
Stockholm	11123

We can get back our initial table by a natural join on Postcode. What is a problem, though, is that when we insert new data, how are we going to check that the FD city street -> postcode still holds? To know which City Street Postcode triples already exist, we have to join the two tables we created - it seems that the decomposition step has removed the ability to efficiently check the FD City Street -> Postcode. The situation is difficult: Either we need to introduce redundancy or produce a design where we cannot check the dependencies (dependency preservation is lost).

Let us try decomposing not based on a FD, say, projecting new tables on City,Street and Street,Postcode

city	**street**
Gothenburg	Framnäsgatan
Gothenburg	Rännvägen
Gothenburg	Hörsalsvägen
Göteborg	Barnhusgatan
Stockholm	Barnhusgatan

street	postcode
Framnäsgatan	41264
Rännvägen	41296
Hörsalsvägen	41296
Barnhusgatan	41102
Barnhusgatan	11123

This is an even worse idea. When we join back on Street, we get all the original rows and additionally

city	street	postcode
Göteborg	Barnhusgatan	11123
Stockholm	Barnhusgatan	41102

which is just plain wrong. Our design has led to loss of information.

The idea of the decomposition step we used based on FD `postcode -> city` is the basis of decomposing database schemas into BCNF.

Algorithm (BCNF decomposition). Consider a relation R with schema S and a set F of functional dependencies, which is a minimal cover. Apply the following steps as long as they lead to a change.

1. If R has no BCNF violations, return R
2. If R has a violating functional dependency $X \to A$, decompose R to two relations
   - R_1 with schema $X \cup X^+$
   - R_2 with schema $S - X^+ - X$

In the City Street Postcode, the only FD which can be used to decompose is `postcode -> city` and produces, like seen above, the schemas Street Postcode and Postcode City, which are in BCNF, but the ability to check for dependency `City Street -> Postcode` has disappeared. In this case it would seem like a good choice not to decompose, particularly as postcodes do not change that often.

What to do with the other functional dependency, `currency -> value`? We could call it a **non-key FD**, which is not standard terminology, but a handy term. Looking at the table, we see that it creates a **redundancy**: the value is repeated every time a currency occurs. Non-key FD's are called **BCNF violations**. They can be removed by **BCNF decomposition**: we build a separate table for each such FD. Here is the result:

country	currency
Sweden	SEK
Finland	EUR
Estonia	EUR

currency	value
SEK	0.12
EUR	1.10

These tables have no BCNF violations, and no redundancies either. Each of them has their own functional dependencies and keys:

```
Countries (_country, currency)
FD: country -> currency
reference: currency -> Currencies.currency

Currencies (_currency, value)
FD: currency -> value
```

They also enjoy **lossless join**: we can reconstruct the original table by a natural join Countries $\bowtie$ Currencies.
Example Consider the schema

```
country currency value
country -> currency
country -> value
currency -> value
```

It has one 3NF violation: `currency -> value`. Moreover, the FD set is not a minimal cover: the second FD can be dropped because it follows from the first and the third ones by transitivity. Hence we have a minimal cover

```
country -> currency
currency -> value
```

Applying 3NF decomposition to this gives us two schemas:

```
country currency
currency value
```

i.e. exactly the same ones as we would obtain by BCNF decomposition. These relations are hence not only 3NF but also BCNF.

5.3 Database design workflow

Functional dependency analysis is a mathematical tool for database design. The classic way to apply it goes as follows:

1. Collect *all* attributes into one and the same relation, resulting in a so-called *universal relation*.
2. Specify the functional dependencies among the attributes.
3. From the functional dependencies, calculate the possible keys of the relation.
4. From the FDs and keys together, calculate the violations of normal forms
5. From the normal form violations, compute a decomposiitions of the relation to a set of smaller relations.
6. Decide which decompositions you want.

Iterating this, it is possible to reach a state with no violations of normal forms. At this point it is also possible to compare the dependency-based design with the E-R design.

It seems not so smart, though, to neglect the E-R diagram in the design. To utilize the E-R diagram information, the following workflow is recommended:

1. As a basis, get the initial relations and functional dependencies from the E-R diagram.
2. Model additional FDs.
3. From the functional dependencies, calculate the possible keys of the relations.
4. Use BCNF decomposition where seen appropriate.

Dependency-based design is, in a way, more mechanical than E-R design. In E-R design, you have to decide many things: the ontological status of each concept (whether it is an entity, attribute, relationship, etc). You also have to decide the keys of the entities. In dependency analysis, you only have to decide the basic dependencies. Lots of other dependencies are derived from these by mechanical rules. Also the possible keys - **candidate keys** - are mechanically derived. The decomposition to normal forms is mechanical as well. You just have to decide what normal form (if any) you want to achieve. In addition, you have to decide which of the candidate keys to declare as the **primary key** of each table.

Be reminded that it is a standard practice to replace the calculated keys by artificial surrogate keys and just use the calculated keys for uniqueness constraints.

You need to decide what decomposition you want. Normalization has pros and cons. This also depends on the application. If query performance is crucial and one wants to avoid joins, and updates seldom happen or are otherwise known to be feasible on pre-joined data, then you may decompose less or store joins directly.

The following section on acyclicity will sort of crash the theory we learned until now. We will see that there are further complications and that the decomposition and BCNF do not necessarily solve our design problems. These complications are related to dependency patterns that may or may not arise. Anyway, it is important to recognize the limitations of design methods.

5.4 Acyclicity and functional dependencies on database level*

(not covered in the course)

5.5 Inclusion dependencies*

(not covered in the course)

5.6 Multivalued dependencies and the fourth normal form

Consider the following information about persons. Bob speaks English. Jill speaks French. Jill speaks Swedish. Bob is a cook by profession. Jill is a lawyer by profession. Jill is also a programmer by profession. Let us consider putting this information on a single table with Name, Language, and Profession as attributes. We can easily get a row with Bob, English and cook as attribute values. But how about Jill? If we put Jill and French in a row, what would be the related profession? To cover all the combinations, we could have rows with Jill, French and lawyer, as well as Jill, French and programmer. The whole table would look like the following:

person	**language**	**profession**
Bob	English	cook
Jill	French	lawyer
Jill	French	programmer
Jill	Swedish	lawyer
Jill	Swedish	programmer

Thus we can say that the languages of persons are inindependent of their professions. This can be expressed as multivalued dependencies,

```
person ->> language
person ->> profession
```

Why "multivalued"? Because there are multiple values of language and profession for each person, and these value sets are *independent* of each other.

Multivalued dependencies (MVD) are based on this idea. An MVD $X \twoheadrightarrow Y$ in a relation schema R says that X determines Y independently of all other attributes of R. The standard formal definition is a bit more complicated, but the following observation is sufficient.

Given a relation r with schema R and '$X \subset R, Y \subset R$, then $X \twoheadrightarrow Y$ if $r = \pi_{XY} r \bowtie \pi_{X \cup \{R-Y}$.

The MVD $X \twoheadrightarrow Y$ is trivial if $Y \subseteq X$ or $R \subseteq X \cup Y$. In this book, we are considering non-trivial MVDs.

Let $Z = R - (X \cup Y)$ and consider a non-trivial MVD $X \twoheadrightarrow Y$. Note that Y and Z are in a symmetric position, so we also have $X \twoheadrightarrow Z$. An alternative notation is $X \twoheadrightarrow Y \mid Z$, emphasizing that Y is **independent** of Z. Interestingly, if $X \rightarrow Y$, then also $X \twoheadrightarrow Y$.

In short, if r is equal to joining the projections of r on XY and R-Y, then $X \twoheadrightarrow Y$.

Fourth normal form (4NF). If there is a non-trivial MVD $X \twoheadrightarrow Y$ in a relation schema and X is not a superkey, that is a *fourth normal form* (4NF) violation, and the relation is not in 4NF. If no 4NF violations exist in a relation, then that relation is in 4NF.

If we add a row with values Jill, Finnish, pilot, then what? Then the relation would not be symmetric on language and profession for a person. This could

be an error, but it could also mean that language and profession are not independent, and the relation gives the languages in which a person can practice a profession. Also, the 4NF violation would go away.

Note that MVDs and 4NF are different from FDs and BCNF in an interesting way: A BCNF violation can only be created by adding a tuple to the database whereas a 4NF violation can be created also by deleting a tuple from a database.

As another example of MVDs, let us consider the following. Sweden produces cars. Sweden produces paper. Norway is a neighbour of Sweden. Finland is a neighbour of Sweden. Thus, we get a table of countries, their export products, and their neighbours:

country	**product**	**neighbour**
Sweden	cars	Norway
Sweden	paper	Finland
Sweden	cars	Finland
Sweden	paper	Norway

In this table, Sweden exports both cars and paper, and it has both Finland and Norway as neighbours. Intuitively, export products and neighbours are two independent facts about Sweden: they have nothing to do with each other, and we have the MVDs country $\twoheadrightarrow$ product, country $\twoheadrightarrow$ neighbour, or, equivalently, country $\twoheadrightarrow$ product | neighbour.

The **4NF decomposition** splits the table in accordance to the violating MVD:

country	**product**
Sweden	cars
Sweden	paper

country	**neighbour**
Sweden	Norway
Sweden	Finland

These tables are free from violations. Hence they are in 4NF. Their natural join losslessly returns the original table.

It is relatively easy to see that 4NF is a stronger requirement than BCNF. If $X \to A$ violates BCNF, then it also violates 4NF, because

- it is an MVD by the theorem above
- it is not trivial, because
  - if $\{A\} \subseteq X$, then $X \to A$ is a trivial FD and cannot violate BCNF
  - if $X \cup \{A\} = S$, then X is a superkey and $X \to A$ cannot violate BCNF

The 4NF decomposition algorithm can be formulated as follows.

Algorithm (4NF decomposition). Consider a relation R with signature S and a set M of multivalued dependencies. R can be brought to 4NF by the following steps:

1. If R has no 4NF violations, return R

2. If R has a violating multivalued dependency $X \twoheadrightarrow Y$, decompose R to two relations
   - R_1 with signature $X \cup \{Y\}$
   - R_2 with signature $S - Y$
3. Apply the above steps to *R1* and *R2*

In the previous example, we could actually prove the MVD by looking at the tuples (see definition of MVD below). Finding a provable MVD in an instance of a database can be difficult, because so many combinations must be present. An MVD might of course be assumed to hold as a part of the domain description. This can lead to a better structure and smaller tables. However, if the assumption fails, the natural join from those tables can produce combinations not existing in the reality.

A generalisation of 4NF is the *project-join normal form* (PJNF) also called *fifth normal form*, which takes into account the possibility that a relation could be a result of several joins instead of just one.

In practice, it is very rare that anyone would actually use a relation violating 4NF.

5.7 An example with both multivalued and functional dependencies

First of all, it is worth noting that the interactions between MVDs and FDs can be very complicated. This example only gives an idea of what can be done with a set of MVDs and a set of FDs.

The assumed starting point for the design is a set of attributes, and a set of dependencies. Without any dependencies, we would just end up with a single table containing all the attributes.

Let us assume the following set of attributes

```
country capital popCountry popCapital
currency value product neighbour
```

and the following functional dependencies and multivalued dependencies:

```
currency ->  value
country  ->  capital popCountry currency
capital  ->  country popCapital
country  ->> product
country  ->> neighbour
```

Decomposing with the FDs in the order they are given (the third one does not decompose anymore), we get the following tables.

```
_country capital popCountry popCapital currency
_currency value
_country _product _neighbour
```

This looks like a good structure, except for the last table. Applying 4NF decomposition to this gives the final result

```
_country capital popCountry popCapital currency
_currency value
_country _product
_country _neighbour
```

A word of warning: mechanical decomposition can split other dependencies, and lead to less natural results. This also affects the keys, for instance we could end up with capital rather than country as the key of the first table.

6 SQL query processing with relational algebra

Relational algebra is a mathematical query language. It is much simpler than SQL, as it has only a few operations, each usually denoted by Greek letters or mathematical symbols. But for the same reason, it is easier to analyse and optimize. Relational algebra is therefore useful as an intermediate language in a database management system. SQL queries can be first translated to relational algebra, which is optimized before it is executed. This chapter will tell the basics about this translation and some query optimizations.

6.1 The compiler pipeline

When you write an SQL query in PostgreSQL or some other DBMS, the following things happen:

1. **Lexing**: the query string is analysed into a sequence of words.
2. **Parsing**: the sequence of words is analysed into a **syntax tree**.
3. **Type checking**: the syntax tree is checked for semantic well-formedness, for instance that you are not trying to multiply strings but only numbers, and that the names of tables and attributes actually exist in the database.
4. **Logical query plan generation**: the SQL syntax tree is converted to a **logical query plan**, which is a relational algebra expression (actually, its syntax tree).
5. **Optimization**: the relational algebra expression is converted to another relational algebra expression, which is more efficient to execute.
6. **Physical query plan generation**: the optimized relational algebra expression is converted to a **physical query plan**, which is a sequence of algorithm calls.
7. **Query execution**: the physical query plan is executed to produce the result of the query.

We will in this chapter focus on the logical query plan generation. We will also say a few words about optimization, which is perhaps the clearest practical reason for the use of relational algebra.

6.2 Relational algebra

Relational algebra is in principle at least as powerful as SQL as a query language, because basic SQL queries can be translated to it. Yet the language is much smaller. In practice the comparision is a bit more complicated, because there are different variants of both SQL and relational algebra.

The grammar of the relational algebra used in this book is shown in Figure 7.

Since this is the "official" relational algebra (from the textbook), a couple of SQL constructs cannot however be treated: sorting in `DESC` order and aggregation of `DISTINCT` values. Both of them would be easy to add. More imporatantly, this language extends the algebra of Chapter 4 in several ways.

relation ::=

relname	**name of relation (can be used alone)**
$\mid \sigma_{\text{condition}}$ relation	**selection (sigma)** `WHERE`
$\mid \pi_{\text{projection+}}$ relation	**projection (pi)** `SELECT`
$\mid \rho_{\text{relname (attribute+)?}}$ relation	**renaming (rho)** `AS`
$\mid \gamma_{\text{attribute*,aggregationexp+}}$ relation	**grouping (gamma)** `GROUP BY, HAVING`
$\mid \tau_{\text{expression+}}$ relation	**sorting (tau)** `ORDER BY`
$\mid \delta$ relation	**removing duplicates (delta)** `DISTINCT`
| relation $\times$ relation	**cartesian product** `FROM, CROSS JOIN`
| relation $\cup$ relation	**union** `UNION`
| relation $\cap$ relation	**intersection** `INTERSECT`
| relation $-$ relation	**difference** `EXCEPT`
| relation $\bowtie$ relation	`NATURAL JOIN`
| relation $\bowtie_{\text{condition}}$ relation	**theta join** `JOIN ON`
| relation $\bowtie_{\text{attribute+}}$ relation	`INNER JOIN`
| relation $\bowtie^{o}_{\text{attribute+}}$ relation	`FULL OUTER JOIN`
| relation $\bowtie^{oL}_{\text{attribute+}}$ relation	`LEFT OUTER JOIN`
| relation $\bowtie^{oR}_{\text{attribute+}}$ relation	`RIGHT OUTER JOIN`

projection ::=

expression	**expression, can be just an attribute**
| expression $\rightarrow$ attribute	**rename projected expression** `AS`

aggregationexp ::=

aggregation(*|attribute)	**without renaming**
| aggregation(*|attribute) $\rightarrow$ attribute	**with renaming** `AS`

expression, condition, aggregation, attribute *as in SQL, Figure 14, but excluding subqueries*

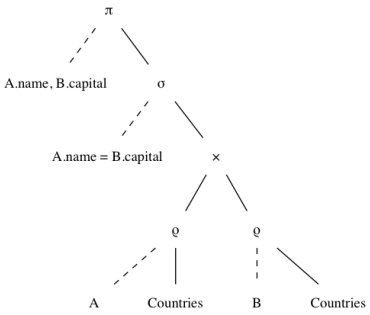
Figure 7: A grammar of relational algebra. Operator names and other explanations in **boldface**. Corresponding SQL keywords in `CAPITAL TYPEWRITER`.

The most important extension is that it operates on **multisets** (turned to sets by $\delta, \cup, \cap$) and recognizes **order** (controlled by τ).

6.3 Variants of algebraic notation

The standard notation used in textbooks and these notes is Greek letter + subscript attributes/conditions + relation. This could be made more readable by using a tree diagram. For example,

$$\pi_{A.name,B.capital}\sigma_{A.name=B.capital}(\rho_A\text{Countries} \times \rho_B\text{Countries})$$



6.4 From SQL to relational algebra

The translation from SQL to relational algebra is usually straightforward. It is mostly **compositional** in the sense that each SQL construct has a determinate algebra construct that it translates to. For expressions, conditions, aggregations, and attributes, it is trivial, since they need not be changed at all. Figure 7 shows the correspondences as a kind of a dictionary, without exactly specifying how the syntax is translated.

The most problematic cases to deal with are

- grouping expressions
- subqueries in certain positions, e.g. conditions

We will skip subqueries and say more about the grouping expressions. But let us start with the straightforward cases.

6.4.1 Basic queries

The most common SQL query form

```
SELECT projections
FROM table,...,table
WHERE condition
```

corresponds to the relational algebra expression

$$\pi_{\text{projections}}\sigma_{\text{condition}}(\text{table} \times \ldots \times \text{table})$$

But notice, first of all, that names of relations can themselves be used as algebraic queries. Thus we translate

```
SELECT * FROM Countries
```

$\Longrightarrow$ Countries

```
SELECT * FROM Countries WHERE name='UK'
```

$\Longrightarrow$ $\sigma_{name='UK'}$Countries

In general, `SELECT *` does not add anything to the algebra translation. However, there is a subtle exception with grouping queries, to be discussed later.

If the `SELECT` field contains attributes or expressions, these are copied into the same expressions under the π operator. When the field is given another name by `AS`, the arrow symbol is used in algebra:

```
SELECT capital, area/1000 FROM Countries WHERE name='UK'
```

$\Longrightarrow$ $\pi_{capital,area/1000}\sigma_{name='UK'}$Countries

```
SELECT name AS country, population/area AS density FROM Countries
```

$\Longrightarrow$ $\pi_{name\to country,population/area\to density}$Countries

The renaming of attributes could also be done with the ρ operator:

$\rho_{C(country,density)}\pi_{name,population/area}$Countries

is a more complicated and, in particular, less compositional solution, because the `SELECT` field is used in two different algebra operations. Moreover, it must invent C as a dummy name for the renamed table. However, ρ is the way to go when names are given to tables in the `FROM` field. This happens in particular when a cartesian product is made with two copies of the same table:

```
SELECT A.name, B.capital
FROM Countries AS A, Countries AS B
WHERE A.name = B.capital
```

$\Longrightarrow$

$\pi_{A.name,B.capital}\sigma_{A.name=B.capital}(\rho_A\text{Countries} \times \rho_B\text{Countries})$

No renaming of attributes takes place in this case.

Set-theoretical operations and joins work in the same way as in SQL. Notice once again that the SQL distinction between "queries" and "tables" is not present in algebra, but everything is relations. This means that all operations work with all kinds of relation arguments, unlike in SQL (see Section 2.22).

6.4.2 Grouping and aggregation

As we saw in Section 2.14, GROUP BY a (or any sequence of attributes) to a table R forms a new table, where a is the key. The other attributes can be found in two places: the SELECT line above and the HAVING and ORDER BY lines below. All of these attributes must be aggregation function applications.

In relational algebra, the γ operator is an explicit name for this relation, collecting all information in one place. Thus

```
SELECT currency, COUNT(name)
FROM Countries
GROUP BY currency
```

$$\Longrightarrow \qquad \gamma_{currency,COUNT(name)}\text{Countries}$$

```
SELECT currency, AVG(population)
FROM Countries
GROUP BY currency
HAVING COUNT(name) > 1
```

$$\Longrightarrow$$

$$\pi_{currency,AVG(population)}\sigma_{COUNT(name)>1}$$
$$\gamma_{currency,AVG(population),COUNT(name)}\text{Countries}$$

Thus the HAVING clause itself becomes an ordinary σ. Notice that, since SELECT does not show the COUNT(name) attribute, a projection must be applied on top.

Here is an example with ORDER BY, which is translated to τ:

```
SELECT currency, AVG(population)
FROM Countries
GROUP BY currency
ORDER BY COUNT(name)
```

$$\Longrightarrow$$

$$\pi_{currency,AVG(population)}\tau_{COUNT(name)}$$
$$\gamma_{currency,AVG(population),COUNT(name)}\text{Countries}$$

The "official" version of relational algebra (as in the book) performs the renaming of attributes under SELECT γ itself:

```
SELECT currency, COUNT(name) AS users
FROM Countries
GROUP BY currency
```

$$\Longrightarrow \qquad \gamma_{currency,COUNT(name)\rightarrow users}\text{Countries}$$

However, a more compositional (and to our mind more intuitive) way to do this is in a separate π corresponding to the SELECT:

$$\pi_{currency,COUNT(name)\rightarrow users}\gamma_{currency,COUNT(name)}\text{Countries}$$

The official notation actually always involves a renaming, even if it is to the aggregation expression to itself:

$$\gamma_{currency,COUNT(name)\rightarrow COUNT(name)}\text{Countries}$$

This is of course semantically justified because COUNT(name) on the right of the arrow is not an expression but an attribute (a string without syntactic structure). However, the official notation is not consistent in this, since it does not require corresponding renaming in the π operator.

In addition to GROUP BY, γ must be used whenever an aggregation appears in the SELECT part. This can be understood as grouping by 0 attributes, which means that there is only one group. Thus we translate

```
SELECT COUNT(name) FROM Countries
```

$$\Longrightarrow \qquad \gamma_{COUNT(name)}\text{Countries}$$

We conclude the grouping section with a surprising example:

```
SELECT *
FROM Countries
GROUP BY name
HAVING count(name) > 0
```

Surprisingly, the result is the whole Countries table (because the HAVING condition is always true), without a column for `count(name)`. This may be a bug in PostgreSQL. Otherwise it is a counterexample to the rule that `SELECT *` does not change the relation in any way.

6.4.3 Sorting and duplicate removal

We have already seen a sorting with groupings. Here is a simpler example:

```
SELECT name, capital FROM Countries ORDER BY name
```

$$\Longrightarrow \qquad \tau_{name}\pi_{name,capital}\text{Countries}$$

And here is an example of duplicate removal:

```
SELECT DISTINCT currency FROM Countries
```

$$\Longrightarrow \qquad \delta(\pi_{currency}\text{Countries})$$

(The parentheses are optional.)

6.5 Query optimization

6.5.1 Algebraic laws

Set-theoretic operations obey a large number of laws: associativity, commutativity, idempotence, etc. Many, but not all, of these laws also work for multisets. The laws generate a potentially infinite number of equivalent expressions for a query. Query optimization tries to find the best of those.

6.5.2 Example: pushing conditions in cartesian products

Cartesian products generate huge tables, but only fractions of them usually show up in the final result. How can we avoid building these tables in intermediate stages? One of the most powerful techniques is pushing conditions into products, in accordance with the following equivalence:

$$\sigma_C(R \times S) \;=\; \sigma_{Crs}(\sigma_{Cr}R \times \sigma_{Cs}S)$$

where C is a conjunction (AND) of conditions and

- Cr is that part of C where all attributes can be found in R
- Cs is that part of C where all attributes can be found in S
- Crs is the rest of the attributes in C

Here is an example:

```
SELECT * FROM countries, currencies
WHERE code = 'EUR' AND continent = 'EU' AND code = currency
```

Direct translation:

$$\sigma_{code="EUR"ANDcontinent="EU"ANDcode=currency}(countries \times currencies)$$

Optimized translation:

$$\sigma_{code=currency}(\sigma_{continent="EU"}countries \times \sigma_{code="EUR"}currencies)$$

6.6 Indexes

The SQL queries are evaluated based on values in the database. Therefore, it is beneficial to enable fast value-based search of the data. An index is an efficient lookup structure, making it fast to fetch information by the values of the index. The DBMS utilizes the available indices automatically, if it recognizes that it is evaluating an expression where data is searched by values for which an index exists. The most prominent examples for using an index include making a selection based on the index value or such natural join of two tables that the join attribute appears as a key in one of the tables.

A DBMS automatically creates an index for the primary key of each table. It is used to eliminate duplicate key values. One can manually create and drop keys for other attributes as needed by using the following SQL syntax:

```
statement ::=
   CREATE INDEX indexname ON tablename (attribute+)?
 | DROP INDEX indexname
```

Creating an index is a mixed blessing, since it

- speeds up queries
- makes modifications slower

An index obviously also takes extra space. To decide whether to create an index on some attributes, one should estimate the **cost** of typical operations. The cost is traditionally calculated by the following **cost model**:

- The **disk** is divided to **blocks**.
- Each tuple is in a block, which may contain many tuples.
- A **block access** is the smallest unit of time.
- Read 1 tuple = 1 block access.
- Modify 1 tuple = 2 block accesses (read + write).
- Every table is stored in some number n blocks. Hence,
  - Reading all tuples of a table = n block accesses.
  - In particular, lookup all tuples matching an attibute without index = n.
  - Similarly, modification without index = $2n$.
  - Insert new value without index = 2 (read one block and write it back).
- An index is stored in 1 block (idealizing assumption). Hence, for indexed attributes,
  - Reading the whole index = 1 block access.
  - Lookup 1 tuple (i.e. to find where it is stored) = 1 block access.
  - Fetch all tuples = 1 + k block accesses (where k is the number of tuples per attribute and $k \ll n$, i.e. k is significantly smaller than n).
  - In particular, fetching a tuple if the index is a key = 2 (1 + k with k=1).
  - Modify (or insert) 1 tuple with index = 4 block accesses (read and write both the tuple and the index).

With this model, the decision goes as follows:

1. Generate some candidates for indexes: $I_1, \ldots, I_k$
2. Identify the a set of typical SQL statements (for instance, the queries and updates performed via the end user interface): $S_1, \ldots, S_n$
3. For each candidate index configuration, compute the costs of the typical statements: $C(I_i, S_j)$
4. Estimate the probabilities of each typical statement, e.g. as its relative frequency: $P(S_j)$
5. Select the best index configuration, i.e. the one with the lowest expected cost: $\text{argmin}_i \sum_{j=1}^{n} P(S_j)C(I_i)$

Example. Consider a phone book, with the schema

```
PhoneNumbers(name,number)
```

Neither the name nor the number can be assumed to be a key, since one person can have many numbers, and many persons can share a number. Assume that

- the table is stored in 100 blocks: n=100
- each name has on the average 2 numbers (k=2), whereas each number has 1 person (k=1 ; actually, 1.01, but we round this down to 1)
- the following statement types occur with the following frequencies:

```
SELECT number FROM PhoneNumbers WHERE name=X  -- 0.8
SELECT name FROM PhoneNumbers WHERE number=Y  -- 0.05
INSERT INTO PhoneNumbers VALUES (X,Y)         -- 0.15
```

Here are the costs with each indexing configuration, with "index both" as the winner:

Query	**No index**	**Index name**	**Index number**	**Index both**
SELECT number	100	3	100	3
SELECT name	100	100	2	2
INSERT	2	4	4	6
total cost	85.3	8.0	80.25	**3.0**
why	80+5+0.3	2.4+5+0.6	80+0.05+0.2	2.4+0.1+0.9

6.7 Physical computing environment

Sufficient query processing efficiency is created by several inter-related factors. The logical database design, the query formulation, and the use of indices are all design factors that contribute to the efficiency.

The other contributing part is the physical computation environment. Sufficiently powerful processors and large main memory are parts of the contributing physical environment. It is possible to distribute the data to several database servers in a way that is rather transparent (non-visible) to the database user.

However, the simple solution, if feasible, is to have a single database server that is sufficiently powerful to process the required queries. To utilize well physical database server capacity, it may be necessary to adjust various parameters that enable efficient processing of data.

Query processing requires a potentially large amount of computer memory where the computations take place. The database data is typically stored on disk for durability, but disk is slow compared to main memory, where the data is read to perform the computations. Therefore, the DBMS maintains data in computer main memory buffers. If the database is small, all of the data may in practice be in main memory. If the buffers are too small, then the query evaluation process needs to write intermediate result chunks to disk and to re-read them from there, which slows down query processing.

7 Database system reliability

Due to the critical nature of data, intuitively the users assume the database systems to be highly reliable in several ways. With so much work used in specifying the correctness requirements for the data, we assume the database system to help in looking after the correctness of the data. For instance, the foreign key requirements should be maintained when the data is updated. Some updates are meant to be performed together, the classic example being transfering money from one account to another, where both withdrawal and deposit should either happen together, or not happen at all. The presence of concurrent users has to be correctly taken into account.

Further, once updates are done, we expect them to be permanent, that is, not unexpectedly lost later. We also expect the database system to support limiting user rights to access only the data they are entitled to.

In this chapter, you will learn what these problems mean, more precisely, and what is the role of the database system in solving them, and, most importantly, what is your role in getting things done right. You will also learn about user authorization to control who can access which data.

7.1 Transactions and the ACID properties

Normally, when we execute the SQL operations with a query interface, each correct operation is executed in the database as soon as they are given to the query interface.

For instance, a bank transfer can consist of two updates:

```
UPDATE Accounts
  SET (balance = balance - 100) WHERE holder = 'Alice' ;
UPDATE Accounts
  SET (balance = balance + 100) WHERE holder = 'Bob' ;
```

If the first update is not feasible, for instance, because Alice has less than 100 pounds, then the second update should not be performed.

A **transaction** is an *atomic* sequence of statements that is executed together: a transaction succeeds or fails as a whole, so that either *all* or *none* of its updates are performed in the database.

We mark the beginning of a transaction with `BEGIN` command, and the the `COMMIT` command indicates that we want to terminate the transaction successfully, storing all the transaction updates to the database.

The previous bank transfer collected to a transaction is as follows:

```
BEGIN ;
UPDATE Accounts
  SET (balance = balance - 100) WHERE holder = 'Alice' ;
UPDATE Accounts
  SET (balance = balance + 100) WHERE holder = 'Bob' ;
COMMIT ;
```

The transaction may fail for technical problems or failures, in which case it is the duty of the database management system to take care that none of the updates is performed. We can also cancel and roll back a transaction by a `ROLLBACK` statement.

In Postgres, individual SQL statements are by default transactions. This includes the execution of triggers, which we will use for bank transfers in Section ?.?. They can e.g. be used to watch after that a balance never goes negative. For the moment, assume that this is the case.

The `BEGIN` command starts a transaction and after that the SQL statements belong to that transaction until it either fails and is rolled back by the system, or either `COMMIT` or `ROLLBACK` statement is used to terminate the transaction.

One of the important properties of database systems is that they give concurrent access to to data to several users. In addition to **atomicity**, introduced above, the transactions also serve for **isolation**, keeping concurrent users' access separate.

Suppose that in addition to the bank transfer transaction above, which we call `T1` for now, we have another bank transfer transaction `T2` as follows.

```
BEGIN ;
UPDATE Accounts
  SET (balance = balance - 50) WHERE holder = 'Bob' ;
UPDATE Accounts
  SET (balance = balance + 50) WHERE holder = 'David' ;
COMMIT ;
```

Suppose, further, that before the transactions Bob's account is at zero. If `T1` is executed first and `T2` after `T1`, then Bob's funds are sufficient for `T2`. Consider the following timeline, where we denote Alice's account balance as `bal(A)`, Bob's account balance as `bal(B)`, and David's account balance as `bal(D)`.

T1	**T2**
BEGIN	_
Update bal(A) = bal(A)-100	_
Update bal(B) = bal(B)+100	_
_	BEGIN
_	Update bal(B) = bal(B)-50
ROLLBACK	_
_	Update bal(D) = bal(D)+50
_	COMMIT

Transaction T2 read the updated balance for Bob's account, changing it from 0 to 100, which enabled T2 to transfer money to David's account. However, as T1 was still running and in fact was then rolled back, this was an intermediate state in the database that was not made permanent. Transaction T2 read intermediate data, which is also called **dirty data**, leading to unwanted behaviour.

We want the database system to maintain a level **isolation** between transactions. The strictest case we require each transaction only see data that is permanently committed to the database. This is the default. There can be applications, where e.g. transactions take a long time or we only need approximate calculations, in which case the requirement for isolation can be flexed.

Once the changes of a transaction are made permanent, they are assumed to be **durable**, meaning that the database system will contain the data even if there are failures like the computers that run the database system being shut down abruptly.

The requirement above that a balance should always stay non-negative is about **consistency** of the data. The requirements of maintaining required foreign keys, non-null values, and correct data types are also about consistency.

Atomicity, Consistency, Isolation and Durability make up the **ACID** properties

- A, **Atomicity**: the transaction is an atomic unit that succeeds or fails as a whole.
- C, **Consistency**: the transaction keeps the database consistent (i.e. preserves its constraints).
- I, **Isolation**: parallel transactions operate isolated of each other.
- D, **Durability**: committed transactions have persistent effect even if the system has a failure.

The ACID properties are in the heart of database reliability. Having introduced those properties, we will take a closer look at how they are maintained.

Hint 1 The main purpose of transactions is to support the maintenance of ACID properties. But a transaction can also be faster than individual statements, as all of them require some overhead, when the system processes the start and finish a transaction. For instance, if you want to execute thousands of `INSERT` statements, it can be better to make them into one transaction.

Hint 2 Transactions keep users isolated from the updated data, using methods introduced later in this chapter. Generally the time of isolation should be short, and preferably not including interaction with users who may take coffee breaks etc., when the system is stopping other users from making progress. If possible, collect the necessary data first and then execute the transaction in a straightforward way.

The full syntax of starting transactions is as follows:

```
statement ::=
    START TRANSACTION mode* | BEGIN | COMMIT | ROLLBACK

mode ::=
    ISOLATION LEVEL level
  | READ WRITE | READ ONLY

level ::=
    SERIALIZABLE | REPEATABLE READ
  | READ COMMITTED | READ UNCOMMITTED
```

The READ WRITE and READ ONLY modes indicate what **interferences** are possible. This, as well as the effect of level, are discussed in more detail later in this chapter.

We will continue to study them, in the order they are listed. In short, atomicity is provided by transactions, so correct use of them has a key position. Maintaining consistency has numerous options that can be specified, so it is a relatively large topic in this book. Isolation maintenance has some options that are good to know. Durability mechanisms are executed by the database system, but it is good to know what it takes to roll back the database state in time, and, of course, taking backups is necessary.

7.2 Atomicity and Durability of database transactions

Atomicity is achieved simply by making a log of database updates and marking the updates as not committed when they are performed. Later, if the transaction is committed, the changed data is marked as normal, and the log of the ongoing transaction is removed. If the transaction is rolled back, then the data is removed along with the log.

Database updates are supposed to be durable. Once succesfully committed, they should be permanent, surviving things like the database system crashing. This is achieved by writing the update events to log files on disk before marking the transaction as committed.

Basically, what happens in transactions in terms of updates is written on the log:

- The update events, including the transaction id and the details of the update, with old and new data
- Commit events of transactions
- Abort events of transactions
- Rollback events (rolling back updates): these are reverting the effects of the update events
- Completion events, when a transaction has been completely dealt with

Durability requires backing up the database files and writing the logs on disk. This includes so-called flushing of the log files while writing, ensuring that the contents are indeed written on the disk instead of just preparing the write in the memory.

If the database crashes, then working from a database backup by re-executing the operations in the log, it is possible to recover the database until a required point in time, or until all the operations in the log belonging to completed trasnsactions are re-executed in the database.

7.3 Isolation of transactions

The technicalities of how transactions are managed is not completely in the focus of this book. There are things, though, that a database user / programmer should understand, at least to trouble shoot when problems appear. There are

two main approaches for maintaining isolation: **locking** and **timestamping**, and the latter naturally combines with **multiversioning**.

Behind the techniques there is the idea that if the result of concurrent transaction execution is the same as if the transactions would be executed in a serial order, one at a time, then we consider the execution correct.

A **schedule** is an ordering of the operations of transactions. A **serial schedule** is a schedule where transactions are executed one after the other. The following schedule is a serial schedule. The other possible serial schedule for T1 and T2 would be to execute all of T2 first, then followed by T1.

T1	**T2**
BEGIN	_
Update bal(A) = bal(A)-100	_
Update bal(B) = bal(B)+100	_
COMMIT	_
_	BEGIN
_	Update bal(B) = bal(B)-50
_	Update bal(D) = bal(D)+50
_	COMMIT

If a schedule is guaranteed to be equal (by its outcome) to a serial schedule, then it is a **serializable** schedule. It is easy to see that any schedule made out of a set of transactions is serializable, if the transactions do not update any values that more than one transaction accesses. In this case they see the same commonly accessed values, no matter in which order they access them.

The following schedule is not serial but it is serializable.

T1	**T2**
BEGIN	_
Select bal(A)	_
_	BEGIN
_	Select bal(D)
Update bal(B) = bal(B)+100	_
COMMIT	_
_	Update bal(A) = bal(A)+50
_	COMMIT

Both transactions T1 and T2 access the data item bal(A). However the outcome is the same as if T1 executed first, then followed by T2. Finally, the following schedule is not serializable, as the write of T1 affects the values seen by T2 and vice versa.

T1	**T2**
BEGIN	_
_	BEGIN
_	Update bal(A) = bal(A)+50
Select bal(A)	_
Update bal(B) = bal(B)+100	_
_	Select bal(B)
COMMIT	_
_	COMMIT

7.3.1 Locking

In locking, when a process needs access to a data item, it will ask for a lock on the item. There are two types of locks: a **Read lock** (R-lock, shared lock, S-lock), and a **Read-Write lock** (RW lock, exclusive lock, X-lock). Having an R-lock to a data item gives the right to read that item, while having an RW-lock to a data item gives the right to both read and write the data item.

The general idea is based on the following thinking. Reads do not generate any dirty data, so we can allow several transactions to hold an R-lock simultaneously. As writes generate temporary dirty data not to be read by others, only one transaction can hold an RW-lock to a data item at any time, and at that time no other transaction can even hold an R-lock. Locks are usually automatically requested by the database management system - in that case the user or programmer does not need to issue additional commands for them.

If a process needs to read a data item and has no previous lock on it, then it asks for an R-lock on the item. If some other process holds an RW-lock to the item, then no lock is given and the process either rolls back or waits. If there are only R-locks or there are no locks on the item, then the R-lock is given to the process and the information of the lock is inserted into a **lock table** holding information on all locks that hold in the database.

Similarly, if a process wants to write a data item, then it asks for an RW-lock on the item. IF some other process holds any kind of a lock, then the lock cannot be given and the requesting process either rolls back or waits. Otherwise the lock is given and inserted to the lock table.

When to release locks? Let us review the balance transfer example of Subsection ?.? and add explicit lock operations.

T1	**T2**
BEGIN	_
Take RW-lock on bal(A)	_
Update bal(A) = bal(A)-100	_
Take RW-lock on bal(B)	_
Update bal(B) = bal(B)+100	_
_	BEGIN
Release RW-lock on bal(A)	_
Release RW-lock on bal(B)	_
_	Take RW-lock on bal(B)
_	Update bal(B) = bal(B)-50
ROLLBACK	_
_	Take RW-lock on bal(D)
_	Update bal(D) = bal(D)+50
_	COMMIT

This way, the locks did not help us to prevent from reading dirty data. To maintain maximum isolation, the transaction only releases locks at the end of transaction. This creates a serializable schedule, where the order of RW-locks on same items as well as R-locks on the the items where someone at some point has an RW-lock, determine the serial order.

The waiting transactions may eventually get the locks when other transactions terminate. Otherwise they may time out and roll back. They may also stay on waiting, but there may be problems like **deadlock**, where there is a cycle of processes each waiting for the locks that the next process in the cycle holds. This can be detected by the system and some "random victim" can be sacrifised for the common good. Another problem is that a process that rolls back and restarts might not get the locks even after repeated restart - this is called **starvation**. As a further problem with locking, transactions accessing large amounts of data will create a lot of entries in the lock tables, thus making the lock tables big.

7.3.2 Timestamping

The basic idea is that when a transaction starts, it is given a timestamp (which can be just a serial number). When the transactions proceed, their operations are analysed. If the schedule thus formed can be made equal to the serial schedule of executing the transactions in timestamp order, then transactions can proceed. Else, some transaction(s) must be rolled back.

Timestamping with multiversioning is based on the idea that several versions of the data item are stored, along with the information on when they have been read and when they have been written. This gives more flexibility, as an earlier transaction (by timestamp) can still access old data correct for it, even if some later transaction has written a new value for that data item. We only consider timestamping with multiversioning here.

To read a data item value, a transaction simply goes through the list of values for that data item, and chooses the right value by the timestamp.

For instance, if the list, assumed here to be ordered, has <write_timestamp, value> pairs <0,5.0>, <104,2.0>, <109,11.0> for some data item A, then a transaction with timestamp 107 wanting to read A would pick the value 2.0 as it is the last value written before time 107. The value written at time 0 is too old and the value written at 109 is too new.

If we ensure that we do not clean up old values that some running transaction may still need, then reads cannot fail. We can remove values from the list when there are no such transactions anymore that would need those values. For instance, if all executing transactions have at least timestamp 104, then we could remove <0,5.0> from the list. We can also choose to keep them, to have access to historical data.

Writing, however, can fail if a transaction tries to write data that someone should have seen (by timestamp). To determine this, we need to store the biggest timestamp for transactions that have seen the value. This way, we need a triple <write_timestamp, latest_read_timestamp, value> for each data item version. Suppose we add for data item A the latest read timestamps as follows: <0,70,5.0>, <104,106,2.0>, <109,109,11.0>. Suppose that a transaction with timestamp 107 wants to write a value 3.0 to A. We know from the timestamps that no-one has read the value between 106 and 109, so we may add a new version, resulting to: <0,70,5.0>, <104,106,2.0>, <107,107,3.0>, <109,109,11.0>

Suppose, now, that after this a transaction holding timestamp 105 wants to write the value 6.0 to A. This is not directly doable, as transaction 106 has read the value 2.0 and for serializable execution, it should have read the value written by transaction 105. Thus, we either need to rollback transaction 105 or transaction 106, as they are in conflict. (If transaction 105 wanted to write exactly the same value 2.0 that was valid since 104, then we could get away from this by just adding a new entry as we know that that is the value read by transaction 106.)

Rolling back transactions has some complications when using timestamps. If we roll back a transaction that has written some data that another transaction has read, then also that transaction has to be rolled back. This way, the rollbacks may **cascade**.

Choosing the transaction to be rolled back when a conflicting write is about to take place, has also implications. The writing transaction clearly has to be the older transaction. If it is rolled back and then restarted, there is a risk of starvation. If younger transactions (the ones that should have seen the data item the older transaction is about to write) is rolled back, then we need to do addiditional book-keeping: We need to track *all such transactions* which means that we need to store all reading transactions' timestamps, not just the last one.

This way, cascading rollback may hit all the younger transactions. Even though reading transactions never block, this way they have to wait for earlier transactions to commit or roll back before they can decide if they can commit or if cascading rollback hits them.

A clear bonus of timestamping with multiversioning is that it has very little

overhead if there are no conflicts, as we do not need to write and check lock tables when accessing the data.

7.3.3 Interferences and isolation levels

If we maintain the transactions in complete isolations as in the previous subsection, then the effects of the execution are the same as if the transactions would be executed in a serial order, i.e. first one as a whole and then the other as a whole. Thereby such arrangements lead to serializable excutions (executions that could be re-arranged as serial).

However, big lock tables or waiting for other transactions to commit may be prohibiting for performance. For some applications we can have more flexible requirements for the concurrency, e.g. it may not be absolutely necessary that they read the last value, or it is not catastrophic if they read a value that finally was not stored permanently by the transaction. A bank might require serializable executions to maintain the balances while in a social media platform we may allow users to see comments that were not stored permanently or the uses may miss the most recent comments at some points.

Thus, sometimes Isolation requirement on the serialisability level in the ACID properties is too rigid. If we allow for multiversioning transactions to commit not waiting for others, then they might have read some value that did not become permanent (dirty data). Similarly, if locking transactions release their locks before commit/rollback and finally roll back. In such cases also repeated reads of the data may bring different results.

In practice, a database system may allow weakening the Isolation condition by using **isolation levels**. The following are recognized isolation levels:

- SERIALIZABLE: guarantees schedules that are equal to serial schedules.
- REPEATABLE READ: like READ COMMITTED, but previously read data may not be changed.
- READ COMMITTED: the transaction can see data that is committed by other transactions during the time it is running, in addition to data committed before it started.
- READ UNCOMMITTED: the transaction sees everything from other transactions, even uncommitted.

The standard example about parallel transactions is flight booking. Suppose you have the schema

```
Seats(date,flight,seat,status)
```

where the `status` is either "vacant" or occupied by a passanger. Now, an internet booking program may inspect this table to suggest flights to customers. After the customer has chosen a seat, the system will update the table:

```
SELECT seat FROM Seats WHERE status='vacant' AND date=...
UPDATE Seats SET status='occupied' WHERE seat=...
```

It makes sense to make this as one transaction.

However, if many customers want the same flight on the same day, how should the system behave? The safest policy is only to allow one booking transaction at a time and let all others wait. This would guarantee the ACID properties. The opposite is to let SELECTs and UPDATEs be freely intertwined. This could lead to **interference problems** in the form or **Dirty reads**: read dirty data resulting from a concurrent uncommitted transaction. A practical consequence could be e.g. overbooking a flight.

T1	**T2**
_	WRITE a
READ a	_
_	ROLLBACK

- **Non-repeatable reads**: read data twice and get different results (because of concurrent committed transaction that modifies or deletes the data).

T1	**T2**
READ a	_
_	UPDATE a = a'
_	COMMIT
READ a	_

- **Phantoms**: execute a query twice and get different results (because of concurrent committed transaction).

T1	**T2**
SELECT * FROM A	_
_	INSERT INTO A a'
_	COMMIT
SELECT * FROM A	_

The following table shows which interference are allowed by which isolation levels, from the strictest to the loosest:

	dirty reads	**non-repeatable reads**	**phantoms**
SERIALIZABLE	-	-	-
REPEATABLE READ	-	-	+
READ COMMITTED	-	+	+
READ UNCOMMITTED	+	+	+

Note. PostgreSQL has only three distinct levels: SERIALIZABLE, REPEATABLE READ, and READ COMMITTED. READ UNCOMMITTED means READ COMMITTED. Hence no dirty reads are allowed.

7.4 Consistency

Here we take a deeper look at inserts, updates, and deletions, in the presence of constraints. The integrity constraints of the database may restrict these actions or even prohibit them. An important problem is that when one piece of data is changed, some others may need to be changed as well. For instance, when a row is deleted or updated, how should this affect other rows that reference it as foreign key? Some of these things can be guaranteed by constraints in basic SQL. But some things need more expressive power. For example, when making a bank transfer, money should not only be taken from one account, but the same amount must be added to the other account. Database management systems support **triggers**, which are programs that do many SQL actions at once. We can utilize triggers to maintain the consistency of databases. Notably, they can be used for other things as well (even for destroying consistency, when used inappropriately). The concepts discussed in this chapter are also called **active elements**, because they affect the way in which the database reacts to actions. This is in contrast to the data itself (the rows in the tables), which is "passive".

7.4.1 Active element hierarchy

Active elements can be defined on different levels, from the most local to the most global:

- **Types** in CREATE TABLE definitions control the atomic values of each attribute without reference to anything else.
- **Inline constraints** in CREATE TABLE definitions control the atomic values of each attribute with arbitrary conditions, but without reference so other attributes (except in REFERENCES constraints).
- **Constraints** in CREATE TABLE definitions control tuples or other sets of attribute, with conditions referring to things inside the table (except for FOREIGN KEY).
- **Assertions**, which are top-level SQL statements, state conditions about the whole database.
- **Triggers**, which are top-level SQL statements, can perform actions on the whole database, using the DBMS.
- **Host program code**, in the embedded SQL case, can perform actions on the whole database, using both the DBMS and the host program.

It is often a good practice to state conditions on as local a level as possible, because they are then available in all wider contexts. However, there are three major exceptions:

- Types such as CHAR(n) may seem to control the length of strings, but they are not as accurate as CHECK constraints. For instance, CHAR(3) only checks the maximum length but not the exact length.
- Inline constraints cannot be changed afterwards by ALTER TABLE. Hence it can be better to use tuple-level named constraints.
- Assertions are disabled in many DBMSs (e.g. PostgreSQL, Oracle) be-

cause they can be inefficient. Therefore one should use triggers to mimick assertions. This is what we do in this course.

Constraints that are marked DEFERRABLE in a CREATE TABLE statement are checked only at the end of each transaction (Section 9.2). This is useful, for instance, if one INSERT in a transaction involves a foreign key that is given only in a later statement.

7.4.2 Referential constraints and policies

A referential constraint (FOREIGN KEY ... REFERENCES) means that, when a value is used in the referring table, it must exist in the referenced table. But what happens if the value is deleted or changed in the referenced table afterwards? This is what **policies** are for. The possible policies are CASCADE and SET NULL, to override the default behaviour which is to reject the change.

Assume we have a table of bank accounts:

```
CREATE TABLE Accounts (
  number TEXT PRIMARY KEY,
  holder TEXT,
  type TEXT,
  balance INT
  )
```

The type attribute is supposed to hold the information of the bank account type: a savings account, a current account, etc. Let us then add a table with transfers, with foreign keys referencing the first table:

```
CREATE TABLE Transfers (
  sender TEXT REFERENCES Accounts(number),
  recipient TEXT REFERENCES Accounts(number),
  amount INT
  )
```

What should we do with Transfers if a foreign key disappears i.e. if an account number is removed from Accounts? There are three alternatives:

- (default) reject the deletion from Accounts because of the reference in Transfers,
- CASCADE, i.e. also delete the transfers that reference the deleted account,
- SET NULL, i.e. keep the transfers but set the sender or recipient number to NULL

One of the last two actions can be defined to override the default, by using the following syntax:

```
CREATE TABLE Transfers (
  sender TEXT REFERENCES Accounts(number)
                ON UPDATE CASCADE ON DELETE SET NULL,
```

```
  recipient TEXT REFERENCES Accounts(number),
  amount INT
  )
```

ON UPDATE CASCADE means that if account number is changed in Accounts, it is also changed in Transfers. ON DELETE SET NULL means that if account number is deleted from Accounts, it is changed to NULL in Transfers.

Finally, we have a table where we log the bank transfers. For each transfer we log the time, the user, the type of modification, and the data (new changed data for insert and update, old removed data for delete).

Notice. This is a simplified example. These policies are probably not the best way to handle accounts and transfers. It probably makes more sense to introduce dates and times, so that rows referring to accounts existing at a certain time need never be changed later.

7.4.3 CHECK constraints

CHECK constraints, which can be inlined or separate, are like **invariants** in programming. Here is a table definition with two attribute-level constraints, one inlined, the other named and separate:

```
-- specify a format for account numbers:
-- four characters,-, and at least two characters
-- more checks could be added to limit the characters to digits
CREATE TABLE Accounts (
  number TEXT PRIMARY KEY CHECK (number LIKE '____-%__'),
  holder TEXT,
  balance INT,
  CONSTRAINT positive_balance CHECK (balance >= 0)
  )
```

Here we have a table-level constraint referring to two attributes:

```
-- check that money may not be transferred from an account to itself
CREATE TABLE Transfers (
  sender TEXT REFERENCES Accounts(number)
  recipient TEXT REFERENCES Accounts(number),
  amount INT,
  CONSTRAINT not_to_self CHECK (recipient <> sender)
  ) ;
```

Recall that a table once created can be changed later with an `ALTER TABLE` statement, including adding and dropping constraints.
Hence a good reason to name constraints it to make `DROP CONSTRAINT` possible.

Here is a "constraint" that is not possible in Transfers, trying to say that the balance cannot be exceeded in a transfer:

```
  CONSTRAINT too_big_transfer CHECK (amount <= balance)
```

The reason is that the Transfers table cannot refer to the Accounts table (other than in FOREIGN KEY constraints). However, in this case, this is also unnecessary to state: because of the `positive_balance` constraint in Accounts, a transfer exceeding the sender's balance would be automatically blocked.

Here is another attempted condition for Accounts, which tries to set a minimum for the money in the bank:

```
CONSTRAINT enough_money_in_bank CHECK (sum(balance) >= 1000000)
```

However, this is not valid SQL. The problem is that constraints in `CREATE TABLE` statements may not use aggregation functions (here, `sum(balance)`), because a constraint can only address individual rows, not entire tables. Global constraints in SQL are called **assertion**, and can be defined as follows:

```
CREATE ASSERTION minimum_balance AS
  CHECK (sum(balance) >= 1000000)
```

However, assertions are not supported by modern DBMS systems such as PostgreSQL, because they are considered too expensive to maintain, since they must be checked after every change in the data. What remains is to use triggers, which can be defined only to apply after narrowly specified changes, not every time anything is changed.

7.4.4 Triggers: a first example

Let us write a trigger that checks the minimum total balance of the bank. We start by defining a **function**, which checks the condition previously expressed by an assertion. The check is performed in an `IF` clause, which checks the negation: whether the total balance is under one million. If this holds, the function raises an `EXCEPTION`. The full syntax details of functions in PostgreSQL will be explained in next section.

```
CREATE FUNCTION minBalance() RETURNS TRIGGER AS $$
BEGIN
  IF ((SELECT sum(balance) FROM Accounts) < 1000000)
    THEN RAISE EXCEPTION 'too little money in the bank' ;
  END IF ;
END
$$ LANGUAGE 'plpgsql' ;
```

After definining this function, we can call it in a trigger. The full details of trigger syntax are, again, given in next section.

```
CREATE TRIGGER check_minimum_balance
  AFTER INSERT OR UPDATE ON Accounts
  FOR EACH STATEMENT
  EXECUTE PROCEDURE minBalance() ;
```

7.4.5 The syntax of triggers

Triggers in PostgreSQL are written in the language PL/PGSQL which is *almost* standard SQL, with an exception: the trigger body can only be a function call, and the function must be written separately.

Here is the part of the syntax that we will need in our discussion:

```
functiondefinition ::=
  CREATE FUNCTION functionname() RETURNS TRIGGER AS $$
  BEGIN
*   statement
  END
  $$ LANGUAGE 'plpgsql'
  ;

triggerdefinition ::=
  CREATE TRIGGER triggernane
    whentriggered
    FOR EACH ROW|STATEMENT
  ? WHEN ( condition )
    EXECUTE PROCEDURE functionname
    ;

whentriggered ::=
    BEFORE|AFTER events ON tablename
  | INSTEAD OF   events ON viewname

events ::= event | event OR events
event  ::= INSERT | UPDATE | DELETE

statement ::=
    IF ( condition ) THEN statement+ elsif* END IF ;
  | RAISE EXCEPTION 'message' ;
  | sqlstatement ;
  | RETURN NEW|OLD|NULL ;

elsif ::= ELSIF ( condition ) THEN statement+
```

Comments:

- a trigger is activated `BEFORE` or `AFTER` an event, which is a change in the database - either `INSERT`, `UPDATE`, or `DELETE`
- the statements may refer to attributes in `NEW` rows (in case of `INSERT` and `UPDATE`) and in `OLD` rows (in case of `UPDATE` and `DELETE`).
- `FOR EACH ROW` means that the trigger is executed for each new row affected by the change
- `FOR EACH STATEMENT` means that the trigger is executed just once for the whole statement, even when it affects several rows

- a trigger is an **atomic transaction**, which either succeeds or fails totally (see below for more details).
- The PL/PGSQL functions have also access to further information, such as
  - function `now()` telling the current time
  - variable `TG_OP` telling the type of event (`INSERT`, `UPDATE`, or `DELETE`)

7.4.6 A more complex trigger example

The following trigger introduces some application logic: it is a trigger that updates balances in `Accounts` after each inserted `Transfer`. The function performs two `UPDATE` actions: it takes money from the sender account, and adds money to the recipient account:

```
CREATE FUNCTION make_transfer() RETURNS TRIGGER AS $$
BEGIN
  UPDATE Accounts
    SET balance = balance - NEW.amount
    WHERE number = NEW.sender ;
  UPDATE Accounts
    SET balance = balance + NEW.amount
    WHERE number = NEW.recipient ;
END
$$ LANGUAGE 'plpgsql' ;
```

The trigger calls this function after a new `Transfer` is inserted.

```
CREATE TRIGGER mkTransfer
  AFTER INSERT ON Transfers
  FOR EACH ROW
  EXECUTE PROCEDURE make_transfer() ;
```

The function `make_transfer()` could also contain checks of conditions, between the `BEGIN` line and the first `UPDATE`:

```
IF (NEW.sender = NEW.recipient)
  THEN RAISE EXCEPTION 'cannot transfer to oneself' ;
END IF ;

IF ((SELECT balance FROM Accounts WHERE number = NEW.sender) < NEW.amount)
  THEN RAISE EXCEPTION 'cannot create negative balance' ;
END IF ;
```

However, both of these things can be already guaranteed by constraints in the affected tables. Then they need not be checked in the trigger. This is clearly better that putting them into triggers, because we could easily forget them!

Our last example adds a logging mechanism to transfers:

```
CREATE OR REPLACE FUNCTION make_transfer_log() RETURNS TRIGGER AS $$
BEGIN
  IF (TG_OP = 'INSERT') THEN
    INSERT INTO TransferLog SELECT now(), user, 'I', NEW.*;
  ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO TransferLog SELECT now(), user, 'U', NEW.*;
  ELSIF (TG_OP = 'DELETE') THEN
    INSERT INTO TransferLog SELECT now(), user, 'D', OLD.*;
  END IF;
  RETURN NULL;
END
$$ LANGUAGE plpgsql ;

CREATE TRIGGER mkLog
  AFTER INSERT OR UPDATE OR DELETE ON Transfers
  FOR EACH ROW
  WHEN (pg_trigger_depth() < 1)
  EXECUTE PROCEDURE make_transfer_log() ;
```

The trigger states an additional condition in a `WHEN` clause, in terms of the function `pg_trigger_depth()`. Its purpose is to cut the chain of iterated function calls. Since a function execution may activate further triggers, it is possible to create a non-terminating loop of trigger activations and function executions. Non-terminating execution is generally bad news, but it can moreover rapidly fill up the database with more and more rows. One safety feature is the `pg_trigger_depth()` function, which returns 0 if we are triggering the first function execution. Each time when a function trigger further functions, the value of `pg_trigger_depth()` increases by 1.

The combinations of `BEFORE`/`AFTER`, `OLD`/`NEW`, and perhaps also `ROW`/`STATEMENT`, can be tricky. It is useful to test different combinations in PostgreSQL, monitor effects, and try to understand the error messages. When testing functions and triggers, it can be handy to write `CREATE OR REPLACE` instead of just `CREATE`, so that a new version can be defined without a `DROP` of the old version.

Triggers can also be defined on views. Then the trigger is executed `INSTEAD OF` updates, inserts, and deletions: the `BEFORE` and `AFTER` directives are used only for triggers on actual tables.

7.5 Authorization and grant diagrams

Typical database applications have multiple users. There is often a need to control what different users can do with the data and see from it.

When a user creates an SQL object (table, view, trigger, function), she becomes the **owner** of the object. She can **grant privileges** to other users, and also **revoke** them. Here is the SQL syntax for this:

```
statement ::=
    GRANT  privilege+ ON object TO user+ grantoption?
```

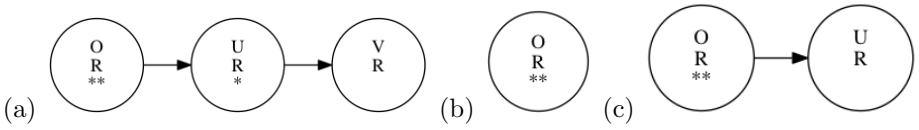


Figure 8: Grant diagrams, resulting from: (a) `O: GRANT R TO U WITH GRANT OPTION ; U: GRANT R TO V` (b) (a) followed by `O: REVOKE R FROM U` (c) (a) followed by `O: REVOKE GRANT OPTION FOR R FROM U`

```
  | REVOKE privilege+ ON object FROM user+ CASCADE?
  | REVOKE GRANT OPTION FOR privilege ON object FROM user+ CASCADE?
  | GRANT rolename TO username adminoption?

privilege ::=
    SELECT | INSERT | DELETE | UPDATE | REFERENCES | ...
  | ALL PRIVILEGES

object ::=
    tablename (attribute+)+ | viewname (attribute+)+ | trigger | ...

user ::= username | rolename | PUBLIC

grantoption ::= WITH GRANT OPTION

adminoption ::= WITH ADMIN OPTION
```

Chains of granted privileges give rise to **grant diagrams**, which ultimately lead to the owner. Each node consists of a username, privilege, and a tag for ownership (**) or grant option (*). Granting a privilege creates a new node, with an arrow from the granting privilege.

A user who has granted privileges can also revoke them. The CASCADE option makes this affect all the nodes that are reachable only via the revoked privilege. The default is RESTRICT, which means that a REVOKE that would affect other nodes is rejected.

Figure 8 shows an example of a grant diagram and its evolution.

Users can be collected to **roles**, which can be granted and revoked privileges together. The SQL privileges can be compared with the privileges in the Unix file system, where

- privileges are "read", "write", and "execute";
- objects are files and directories;
- roles are groups.

The main difference is that the privileges and objects in SQL are more fine-grained.

Usually it is only the database administrators, developers, and special users who access the data directly via SQL. Sometimes users may read the data to, say, Excel spreadsheets, using SQL. Then granting the right SQL rights are of vital importance,

Many users, though, access the data using a program provided for them. This program may be a part of the web application, so in fact the user may use a web page, which then sends a request to a server which provides the data. In such cases the program functionalities limit the data that the user can access. The program logs in to the database as a database user and gets the relevant access. It makes sense to pay attention to the rights given to the programs for various reasons, such as programing errors, various attacks, etc.

8 SQL in software applications

SQL was designed to be a high-level query language, for direct query and manipulation. However, direct access to SQL (e.g. via the PostgreSQL shell) can be both too demanding and too powerful. Most database access by end users hence takes place via more high-level interfaces such as web forms. Most web services such as bank transfers and booking train tickets access a database.

End user programs are often built by combining SQL and a general purpose programming language. This is called **embedding**, and the general purpose language is called a **host language**.

The host language in which SQL is embedded provides GUIs and other means that makes data access easier and safer. Databases are also accessed by programs that analyse them, for instance to collect statistics. Then the host language provides computation methods that are more powerful than those available in SQL.

With this chapter, you will learn the basic principles in embedding SQL into a procedural language, Java, and a functional language, Haskell, and a multi-paradigm language Python. However, the basics of these languages are necessary prerequisites for this chapter. We will also cover some pitfalls in embedding.

A practical challenge in the embedding is making the data types match, starting from the primitive data types, and including the abstract data types such as tables and views, including their schemas. There is a mismatch between the primitive data types of SQL and practically all languages to which SQL is embedded. One has to be careful with string lengths, integer sizes, etc. As a further problem, programming languages typically do not implement NULL values similarly as SQL, and the possibility of NULL values needs to be taken into account in application programming.

There are also security issues. For instance, a security hole can enable **SQL injection** where an end user can include SQL code in the data that she is asked to give. In one famous example, the name of a student includes a piece of code that deletes all data from a student database. To round off, we will look at the highest level of database access from the human point of view: natural language queries.

The code snippets and class diagrams in this chapter are used to create an idea of the proposed ways to implement database connectivity. The book website contains more and more complete examples.

8.1 Embedding SQL to Java using a minimal JDBC program

Java is a verbose language, and accessing a database is just one of the cases that requires a lot of wrapper code. Figure 9 is a minimal complete program doing something meaningful. The user writes a country name and the program returns the capital. After this, a new prompt for a query is displayed. For example:

```
> Sweden
Stockholm
>
```

The progam is very rough in the sense that it does not even recover from errors or terminate gracefully. Thus the only way to terminate it is by "control-C". A more decent program is shown in course website - a template from which Figure 9 is a stripped-down version.

The SQL-specific lines are marked `*`.

- The first one loads the `java.sql` JDBC functionalities.
- The second one, in the `main` method, loads a PostgreSQL driver class.
- The next three ones define the database url, username, and password.
- Then the connection is opened by these parameters. The rest of the `main` method is setting the user inaction loop as a "console".
- The method `getCapital` that sends a query and displays the results can `throw` an exception (a better method would `catch` it). This exception happens for instance when the SQL query has a syntax error.
- The actual work takes place in the body of `getCapital`:
  - The first thing is to create a `Statement` object, which has a method for executing a query.
  - Executing the query returns a `ResultSet`, which is an iterator for all the results.
  - We iterate through the rows with a while loop on `rs.next()`, which returns False when all rows have been scanned.
  - For each row, we print column 2, which holds the capital.
  - The `ResultSet` object `rs` enables us to `getString` for the column.
  - At the end, we close the result set `rs` and the statement `st` nicely to get ready for the next query.

The rest of the code is ordinary Java. For the database-specific parts, excellent Javadoc documentation can be found for googling for the APIs with class and method names. The only tricky thing is perhaps the concepts `Connection`, `Statement`, and `ResultSet`:

- A **connection** is opened just in the beginning, with URL, username, and password. This is much like starting the `psql` program in a Unix shell.
- A **statement** is opened once for any SQL statement to be executed, be it a query or an update, and insert, or a delete.
- A **result set** is obtained when a query is executed.

  Other statements don't return result sets, but just modify the database. It is important to know that the result set is overwritten by each query, so you cannot collect many of them without "saving" them e.g. with for loops.

```
import java.sql.*; // JDBC functionalities                                          *
import java.io.*;  // Reading user input

public class Capital
{

  public static void main(String[] args) throws Exception
  {
    Class.forName("org.postgresql.Driver") ;                    // load the driver class  *

    String url      = "jdbc:postgresql://ate.ita.chalmers.se/" ; // database url           *

    String username = "tda357_XXX" ;                            // your username          *
    String password = "XXXXXX" ;                                // your password          *

    Connection conn = DriverManager.getConnection(url, username, password);//connect to db *

    Console console = System.console();                 // create console for interaction
    while(true) {                                       // loop forever
      String country = console.readLine("> ") ;         // print > as prompt and read query
      getCapital(conn, country) ;                       // execute the query
    }
  }

  static void getCapital(Connection conn, String country) throws SQLException //          *
  {
    Statement st = conn.createStatement();              // start new statement            *
    ResultSet rs =                                      // get the query results          *
      st.executeQuery("SELECT capital FROM Countries WHERE name = '" + country + "'") ;    *
    while (rs.next())                                   // loop through all results       *
      System.out.println(rs.getString(2)) ;             // print column 2 with newline    *

    rs.close();                                         // get ready for new query        *
    st.close();                                         // get ready for new statement    *
  }
}
```

Figure 9: A minimal JDBC program, answering questions "what is the capital of this country". It prints a prompt `>` , reads a country name, prints its capital, and waits for the next query. It does not yet quit nicely or catch exceptions properly. The SQL-specific lines are marked with `*`.

8.2 Building queries and updates from input data

When building a query, it is obviously important to get the spaces and quotes in right positions! A safer way to build a query is to use a **prepared statement**. It has question marks for the arguments to be inserted, so we don't need to care about spaces and quotes. But we do need to select the type of each argument, with `setString(Arg,Val)`, `setInt(Arg,Val)`, etc.

```
static void getCapital(Connection conn, String country) throws SQLException
{
  PreparedStatement st =
    conn.prepareStatement("SELECT capital FROM Countries WHERE name = ?") ;
  st.setString(1,country) ;
  ResultSet rs = st.executeQuery() ;
  if (rs.next())
    System.out.println(rs.getString(1)) ;
  rs.close() ;
  st.close() ;
}
```

Modifications - inserts, updates, and deletes - are made with statements in a similar way as queries. In JDBC, they are all called **updates**. A `Statement` is needed for them as well. Here is an example of registering a mountain with its name, continent, and height.

```
// user input example: Kebnekaise Europe 2111

static void addMountain(Connection conn, String name, String continent, String height)
  throws SQLException
{
  PreparedStatement st =
    conn.prepareStatement("INSERT INTO Mountains VALUES (?,?,?)") ;
  st.setString(1,name) ;
  st.setString(2,continent) ;
  st.setInt(3,Integer.parseInt(height)) ;
  st.executeUpdate() ;
  st.close() ;
}
```

Now, how to divide the work between SQL and the host language, such as Java? As a guiding principle,

Put as much of your program in the SQL query as possible.

In a more complex program (as we will see shortly), one can send several queries and collect their results from result sets, then combine the answer with some Java programming. But this is not using SQL's capacity to the full:

- You miss the optimizations that SQL provides, and have to reinvent them manually in your code.
- You increase the network traffic and pull unnecessary data from the database server.
- Your code is likely to be slower than that of the database system.

Just think about the "pushing conditions" example from Section 6.5.2.

```
SELECT * FROM countries, currencies
WHERE code = 'EUR' AND continent = 'EU' AND code = currency
```

If you just query the first line with SQL and do the WHERE part in Java, you may have to transfer thousands of times of more rows that you moreover have to inspect than when doing everything in SQL.

8.3 Building Java software with database access

The above contains the basic principles for accessing the database from Java programs. However, to build larger systems, one needs to consider other aspects of software development. In a typical database-driven software system there is a large amount of tables, and many operations such as copying values between objects and database are repeated in the code, for different tables and their attributes. There is also a lot of checks and error management related to database data items. A particular challenge is the mismatch between the Java datatypes and the SQL database datatypes, including both primitive datatypes, such as different types of strings, and the abstract datatypes, that is, database tables and views, and objects created to manage the related data.

One needs to ensure that the host language datatypes can hold what comes from the database and that the host program only tries to store to the database such information that the database can hold. This way, it is beneficial to have Java datatypes that include information about the characteristics of the database field. For instance, to avoid trying to insert too long strings into the database, one needs to check the length. Even though there can be a mismatch basically for all data types, particular care is needed with various date and time variables, as they vary between SQL implementations and do not necessarily match the programming language types.

The NULL values need special treatment. JDBC has functions for testing if a value is NULL and to set it NULL. However, if a value is NULL, then no matter how the database software layer treats it, the application logic needs to take it into account and how to do that is a decision by the application logic developer.

JDBC has functions to query the characteristics of data, but it will be cumbersome to use it always for all data values. Instead, one may generate such Java code from the data description that includes the size parameters, and generates an error when data of wrong size is used. Below is a class diagram of such wrapping, just listing the most important methods. The class diagram is meant to be instructive, and it demonstrates various complications that one needs to handle. Please note that things could be implemented in a different way and a number of design choices is already built into this code. However, constructing your software in this kind of a fashion means you are being systematic about your design choices.

Relational databases use values and queries for combining data. Object-oriented model uses pointers or references from one object to another for navigation. Lists in functional programming have some similarity to tables that are produced from SQL, in particular when using list comprehensions in languages like Haskell and Python.

The main observation to be made is that object oriented navigation proceeds through references and making repeated calls to the database to get data row-by-row means misusing the SQL database. SQL databases are designed to manage queries which retrieve a set of rows at once. The set can then be consumed in a one-by-one fashion in the host language software. Complicated queries executed from the host

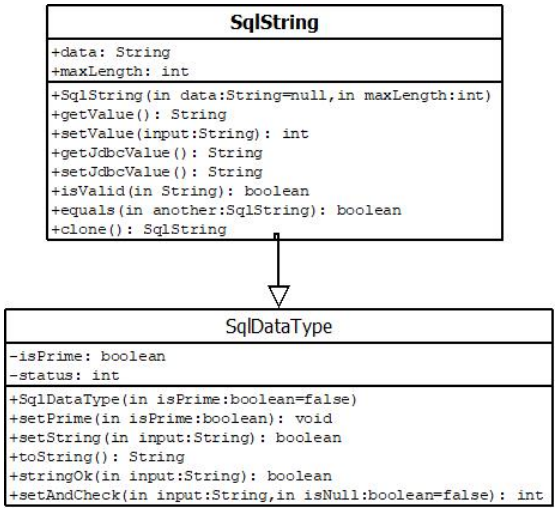


Figure 10: SQL oriented datatype example for Java

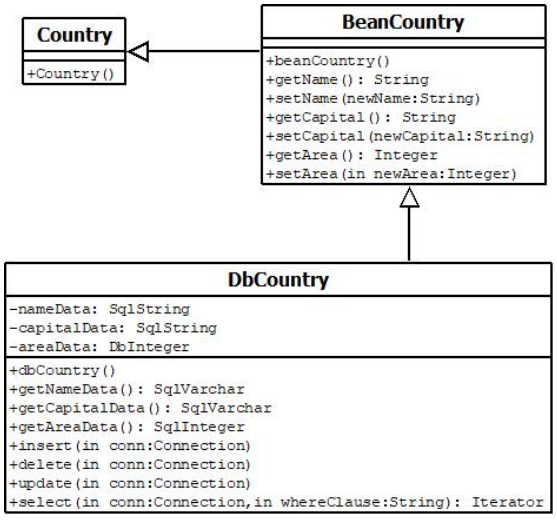


Figure 11: SQL class example

program can be implemented using views, which takes the complications of the query closer to the database system.

We will present one way, consistent to the primitive datatype management above, to wrap relations to classes. Wrapping of views can be done in a similar way apart from the fact that one cannot by default update from views. The db class holds the database functionality while the bean class offers standard setters and getters. The top level class is left for the application specific code. If the database changes, just the base and the db class can be replaced, leaving the application specific code hopefully working as before.

The example, given in Figure 11, wraps a single simplified Countries table in Java classes.

Note that these classes should be generated automatically from the database description using a database design tool, such as the QueryConverter.

8.4 Connecting to a SQL database from Haskell using HDBC

There are different options to connect to a SQL database from Haskell. Here you will learn the basics of using HDBC, which is to a certain extent similar to JDBC. HDBC does not come as a standard part of Haskell Platform, so you need to install package HDBC and to connect to a Postgres database, also the package HDBC-postgresql. The HDBC connection is used in the IO monad.

The short example code, below, exemplifies the user of HDBC. It uses a prepared statement and provides values for updating a country, in this case it changes the capital of Cuba to be Habana (Spanish for Havana) and it makes the continent NULL. It then lists the countries in the Countries table, in a fairly ugly way, but at the same time exhibiting something more about the types.

```
import Database.HDBC -- you need to install this, e.g.: cabal install HDBC
import Database.HDBC.PostgreSQL (connectPostgreSQL) -- cabal install HDBC-postgresql

main = do
  conn <- connectPostgreSQL -- this part needs to be changed for your database
            "host=localhost dbname=testdb user=testuser password=XXXXX"
  withTransaction conn updateCountry -- use transactional mode, explicit commit/rollback
  getCountries <- prepare conn "SELECT * FROM countries;"
  execute getCountries []
  result <- fetchAllRows getCountries
  putStrLn $ show result
  disconnect conn
  return ()

updateCountry conn = do
  updateCountry <-
   prepare conn "UPDATE countries SET continent = ?, capital = ? WHERE name = ?"
  execute updateCountry [SqlNull, toSql "Habana", toSql "Cuba"]
  commit conn -- so we commit here
```

The result is a list of countries, each country further repesented in a list. Removing all but Finland and Cuba, the list looks like:

```
[ [SqlByteString "Finland",SqlByteString "Helsinki",SqlDouble 337030.0,
SqlInteger 5244000,SqlByteString "EU",SqlByteString "EUR"]
[SqlByteString "Cuba",SqlByteString "Habana",SqlDouble 110860.0,
SqlInteger 11423000,SqlNull,SqlByteString "CUP"] ]
```

As the example suggests, the values are wrapped in special Sql datatypes for HDBC, where `SqlNull` represents the NULL value. Robustness can be improved by catching `SqlError` type exceptions, which could occur due to various reasons, like connection might fail. Also, if we try to update a country area to some rubbish like `toSql "oink-oink"` we will get an `SqlError`.

Even with the Sql types there is a need for caution. The needs to check for string lengths to match the database content is one issue, another is to check that the data format is correct to start with. Similarly as with Java, there is a need for a systematic way to construct your software to deal with the type issues. A potential mismatch between Haskell types and database types is that Haskell `Nothing` is not the same as SQL `NULL`, as NULL is a no-information missing value, leaving it open if a value exists or not, whereas `Nothing` is used to mark a does-not-exist missing value ie. a value does not exist at all.

According to the documentation, everything in HDBC happens within transactions. However, in practice with Postgres the situation is the same as using the Postgres interactive shell: if a transaction is not started explicitly, each database operation is a transaction of its own. However, the `withTransaction` function, used in the example, calls the function and runs it within a transaction. Changing the `commit` at the last line of the example into a `rollback` would lead to the update being rolled back.

With Java, above, we suggest implementing specific classes matching the database tables and views. Similarly, in Haskell it is advisable to define types matching the database table and view schemas, using e.g. record types.

8.5 Connecting to a SQL database from Python using Python Database API

For Python, there is a Python Database Application Programming Interface (DB-API), which defines a way to implement the functions that access the database. Different database systems have an implementation of the API, that is used as a library in the programs accessing a database.

For instance, for the Postgresql database system there is a `psycopg` software package, which implements the functions specified in the DB-API. Such a package can be called a database library or a database driver. Other software packages exist for other database systems, and there is also a package that makes it possible to access a database using Java's JDBC library.

The following code implements the same functionality as the HDBC example in the previous subsection.

```
import psycopg2  # here, version 2 of psycopg

try:
    conn = psycopg2.connect("dbname='testdb' user='testuser' password='XXXXX'")
    cur = conn.cursor()
    my_update = """UPDATE countries SET continent = %s, capital = %s WHERE name = %s"""
    my_data = (None, "Habana", "Cuba")  # query is defined separately from data, None is Null
```

```
    cur.execute(my_update, my_data)     # and they are bound together here
    conn.commit()
    cur.execute("SELECT * from countries")
    rows = cur.fetchall()
    print ("\nCountries: \n")
    for row in rows:
      print ("   ", row[1], " ", row[2], " ", row[3], " ", row[4], " ", row[5])

except (Exception, psycopg2.Error) as error:
       print("Error with database access", error)
```

The code largely follows the same lines as the Haskell code, with a couple of exceptions, commented in the code.

Similarly as with Java and Haskell, it is advisable to define types matching the database table and view schemas, using e.g. record types.

8.6 SQL injection and prepared statements

An SQL injection is a hostile attack where the input data contains SQL statements. Such injections are possible if input data is pasted with SQL parts in a simple-minded way. Here are two examples, modified from

https://www.owasp.org/index.php/SQL_Injection

which is an excellent source on security attacks in general, and how to prevent them.

The first injection is in a system where you can ask information about yourself by entering your name. If you enter the name `John`, it builds and executes the query

```
SELECT * FROM Persons
WHERE name = 'John'
```

If you enter the name `John' OR 0=0--` it builds the query

```
SELECT * FROM Persons
WHERE name = 'John' OR 0=0--'
```

which shows all information about all users!

One can also change information by SQL injection. If you enter the name `John';DROP TABLE Persons--` it builds the statements

```
SELECT * FROM Persons
WHERE name = 'John';

DROP TABLE Persons--'
```

which deletes all person information.

Now, if you use JDBC, the latter injection is not so easy, because you cannot execute modifications with `executeQuery()`. But you can do such a thing if the statement asks you to insert your name.

A better help provided by JDBC is to use `preparedStatement` with `?` variables instead of pasting in strings with Java's `+`. The implementation of `preparedStatement` performs a proper **quoting** of the values. Thus the first example becomes rather like

```
SELECT * FROM Persons
WHERE name = 'John'' OR 0=0;--'
```

where the first single quote is escaped and the whole string is hence included in the name.

In the Haskell code, the HDBC is based on the use of prepared statements.

In Python, the DB-API does not have an explicit function call to create a prepared statement separately. The `execute` function prepares the statement and then uses the parameter list to populate the `%s` parameters with values. This would not allow for recycling the SQL query. However, the DB-API specifies another function, `‘executemany‘`, that is given a list of parameter value sets, to be used repetively in the SQL query.

Since the DB-API does not have an explicit call to create a prepared statement, the `psycopg2` library also does not have a call to create a prepared statement.

8.7 Three-tier architecture and connection pooling*

(not covered in the course)

9 Document databases

The relational data model has been dominating the database world for a long time, and it is still the appropriate model for many needs. There are, however, data that are not that naturally encoded as tables. In particular documents with hierarchical heterogeneous structure have been a tough case for relational databases. A particular concern in document databases is a query language or facility, which has search structures more suitable for such documents than SQL.

These documents are modelled typically as key-value pairs, using JSON (JavaScript Object Notation) or similar language. As the values may hierarchically consist of lists of key-value pairs, the notion is general and flexible. We will introduce the usage of JSON using Postgres and MongoDB systems.

Another apporach is XML (eXtensible Markup Language). The hierarchical document structure is modelled through markup tags. Web page language HTML is an example of XML use. XML has designated query languages, such as XPath and XQuery. This chapter introduces XML and gives a summary of XPath.

9.1 JSON

JSON (JavaScript Object Notation)[5] is similar to XML as it supports arbitrarily nested, tree-like data structures. Data in JSON is grouped in **objects**, which are **key-value pairs** enclosed in curly braces. Thus the following object encodes an English-Swedish dictionary entry:

```
{"pos": noun", "english": "computer", "swedish": "dator"}
```

JSON objects are similar to objects in Java and JavaScript, and to **records** in some other programming languages. A peculiarity of JSON is that the keys are quoted, not only the string values.

Both objects and values can be collected to **arrays** in square brackets. Thus a dictionary with two entries is an array of two objects:

```
[
 {"pos": "noun", "english": "computer", "swedish": "dator"},
 {"pos": "verb", "english": "compute",  "swedish": "beräkna"}
]
```

and an entry with three Swedish equivalents for an English word has an array of three string values:

```
{"pos": "noun", "english": "boy",
 "swedish": ["pojke","kille","grabb"]}
```

The grammar of JSON is simple:

```
element ::= object | array | string | number
object  ::= "{" member* "}"
member  ::= string ":" element
array   ::= "[" element* "]"
```

[5] https://www.json.org/

Some things to notice:

- both keys and values are strings in double quotes
- number values can also be given without quotes
- strings `"null"`, `"true"`, `"false"` have special semantics
- JSON data can be freely divided into lines
- the order of key-value pairs in an object doesn't matter
- the order of items in an array does matter

  JSON is a very flexible format, and it does not require similar regularity as tables in SQL. You can e.g. create an array of elements all having different structures.

```
[
 {"pos": "noun", "english": "computer", "swedish": "dator"},
 {"name": "John", "position": "king"}
]
```

  As an element may be an array of elements, it is possible to create arbitrary nested structures.

```
[
 {"name": "John",
  "relationship": "friend",
  "skills": [ { "speaks": "English", "repairs": "bicycles"} ]
  "kids": [ { {"name": "Jill",
               "pets": [ {"fish": "goldfish",
          "dog": {"name": "Rekku", "race": "cairn terrier"} ] }
, {"name":"Bob"} ]
  {"name": "Ben",
   "relationship": "neighbour"}
]
```

While flexibility might be a bonus in that it is possible to describe e.g. values for attributes not seen before, it does introduce challenges for computer programs managing the data that does not have a predefined schema.

JSON (JavaScript Object Notation)[6]

9.2 Querying JSON

As the name tells, JSON was designed to be an object format for JavaScript. Thus it can be queried by JavaScript functions in various ways. Similar support is available in many other languages as well, so that a specific query language is not needed. For instance, the Aeson library in Haskell enables JSON queries via their corresponding Haskell datatypes by using list comprehensions and other Haskell constructs. Thus a query that finds all English words with more than one Swedish translation can be written

```
[english e | e <- entries, length (swedish e) > 1]
```

[6] https://www.json.org/

However, JSON can also be queried with XPath, introduced in Section \ref{xpathSection}, and even PostgreSQL has facilities for inserting and querying JSON data.
In PostgreSQL, an extension of standard SQL introduces the type `json` as a possible data type of a column[7]. For example, a dictionary entry could be defined as follows:

```
CREATE TABLE Entries {
 id INT PRIMARY KEY,
 entry JSON NOT NULL
 }
```

The `entry` part, alas, does not have a specified schema, so the database user must rely on the data being of an expected format. But given that the user has sufficient knowledge about the JSON in the `entry` attribute, one can make the following kinds of queries:

- retrieve JSON objects with ordinary attribute selection
  ```
  SELECT entry
  FROM Entries
  ;
  {"pos": "noun", "english": "computer", "swedish": "dator"}
  ‘ {"pos": "verb", "english": "compute", "swedish": "beräkna"}
  ```
- retrieve values from JSON objects with the -> operator
  ```
  SELECT entry -> 'english'
  FROM Entries
  ;
  "computer"
  "compute"
  ```
- retrieve values from JSON objects in unquoted text with the ->> operator
  ```
  SELECT entry ->> 'english'
  FROM Entries
  ;
  computer
  compute
  ```
- filter with the same arrow operators in `WHERE` clauses
  ```
  SELECT entry -> 'english'
  FROM Entries
  WHERE entry ->> 'pos' = 'noun'
  ;
  computer
  ```

Populating a PostgreSQL database with JSON values can be done with the usual `INSERT` statements:

```
INSERT INTO Entries VALUES (
  4,
  {"pos": "noun", "english": "computation", "swedish": "beräkning"}
  )
```

9.3 MongoDB*

(not covered in the course)

[7] http://www.postgresqltutorial.com/postgresql-json/

9.4 XML and its data model

XML (**eXtensible Markup Language**) is a notation for documents and data. For documents, it can be seen as a generalization of HTML (Hypertext Markup Language): HTML is just one of the languages that can be defined in XML. If more structure is wanted for special kinds of documents, HTML can be "extended" with the help of XML. For instance, if we want to store an English-Swedish dictionary, we can build the following kind of XML objects:

```
<word>
<pos>Noun</pos>
<english>computer</english>
<swedish>dator</swedish>
</word>
```

(where pos = part of speech = "ordklass"). When printing the dictionary, this object could be transformed into an HTML object,

```
<p>
<i>computer</i> (Noun)
dator
</p>
```

But the HTML format is less suitable when the dictionary is used as *data*, where one can look up words. Therefore the original XML structure is better suited for storing the dictionary data.

The form of an XML data object is

```
<tag> ... </tag>
```

where <tag> is the **start tag** and </tag> is the **end tag**. A limiting case is tags with no content in between, which has a shorthand notation,

```
<tag/> = <tag></tag>
```

A grammar of XML is given in Figure 12, using the same notation for grammar rules as used for SQL before.

All XML data must be properly nested between start and end tags. The syntax is the same for all kinds of XML, including XHTML (which is XML-compliant HTML): Plain HTML, in contrast to XHTML, also allows start tags without end tags.

From the data perspective, the XML object corresponds to a row in a relational database with the schema

```
Words(pos,english,swedish)
```

A schema for XML data can be defined in a DTD (**Document Type Declaration**). The DTD expression for the "word" schema assumed by the above object is

```
<!ELEMENT word (pos, english, swedish)>
<!ELEMENT pos (#PCDATA)>
<!ELEMENT english (#PCDATA)>
<!ELEMENT swedish (#PCDATA)>
```

```
document ::= header? dtd? element

header ::= "<?xml version=1.0 encoding=utf-8 standalone=no?>"
  ## standalone=no if with DTD

dtd ::= <! DOCTYPE ident [ definition* ]>

definition ::=
    <! ELEMENT ident rhs >
  | <! ATTLIST ident attribute* >

rhs ::=
   EMPTY | #PCDATA | ident
  | rhs"*" | rhs"+" | rhs"?"
  | rhs , rhs
  | rhs "|" rhs

attribute ::= ident type #REQUIRED|#IMPLIED

type ::= CDATA | ID | IDREF

element ::= starttag element* endtag | emptytag
```

```
starttag  ::= < ident attr* >
endtag    ::= </ ident >
emptytag  ::= < ident attr* />

attr ::= ident = string ## string in double quotes

## XPath

path ::=
    axis item cond? path?
  | path "|" path

axis ::= / | //

item ::= "@"? (ident*) | ident :: ident

cond ::= [ exp op exp ] | [ integer ]

exp  ::=  "@"? ident | integer | string

op   ::= = | != | < | > | <= | >=
```

Figure 12: A grammar of XML and XPath.

The entries in a DTD define **elements**, which are structures of data. The first line defines an element called `word` as a tuple of elements `pos`, `english`, and `swedish`. These other elements are all defined as `#PCDATA`, which means **parsed character data**. It can be used for translating `TEXT` in SQL. But it is moreover *parsed*, which means that all XML tags in the data (such as HTML formatting) are interpreted as tags. (There is also a type for unparsed text, `CDATA`, but it cannot be used in ELEMENT declarations, bur only for values of attributes.)

XML supports more data structures than the relational model. In the relational model, the only structure is the tuple, and all its elements must be atomic. In full XML, the elements of tuples can be structured elements themselves. They are called **daughter elements**. In fact, XML supports **algebraic datatypes** similar to Haskell's `data` definitions:

- Elements can be defined as

tuples of elements:	`E, F`
lists of elements:	`E*`
nonempty lists of elements:	`E+`
alternative elements:	`E | F`
optional elements:	`E?`
strings:	`#PCDATA`

- Elements can be **recursive**, that is, a part of an element can be an instance of the element itself. This enables elements of unlimited size.
- Thus the elements of XML are **trees**, not just tuples. (A tuple is a limiting case, with just one branching node and leaves under it.)

The **validation** of an XML document checks its correctness with respect to a DTD. It corresponds to type checking in Haskell. Validation tools are available on the web, for instance, http://validator.w3.org/

Figure 13 gives an example of a recursive type. It encodes a data type for arithmetic expression, and an element representing the expression `23 + 15 * x`. It shows a complete XML document, which consists of a header, a DTD (starting with the keyword `DOCTYPE`), and an element. It also shows a corresponding algebraic datatype definition in Haskell and a Haskell expression corresponding to the XML element.

```
-- XML

<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE expression [
  <!ELEMENT expression (variable | constant | addition | multiplication)>
  <!ELEMENT variable (#PCDATA)>
  <!ELEMENT constant (#PCDATA)>
  <!ELEMENT addition (expression,expression)>
  <!ELEMENT multiplication (expression,expression)>
  ]>

<expression>
  <addition>
    <expression>
      <constant>23</constant>
    </expression>
    <expression>
      <multiplication>
        <expression>
          <constant>15</constant>
        </expression>
        <expression>
          <variable>x</variable>
        </expression>
      </multiplication>
    </expression>
  </addition>
</expression>

-- Haskell

  data Expression =
      Variable String
    | Constant String
    | Addition Expression Expression
    | Multiplication Expression Expression

  Addition
    (Constant "23")
    (Multiplication
       (Constant "15")
        (Variable "x")))
```

Figure 13: A complete XML document for arithmetic expressions and the corresponding Haskell code, with $23 + 15x$ as example.

To encode SQL tuples, we only need the tuple type and the `PCDATA` type. However, this DTD encoding does not capture all parts of SQL's table definitions:

- basic types in XML are not so refined: basically only `TEXT` is available
- constraints cannot be expressed in the DTD

Some of these problems can be solved by using **attributes** rather than elements. Here is an alternative representation of dictionary entries:

```
<!ELEMENT word EMPTY>
<!ATTLIST word
    pos CDATA #REQUIRED
    english CDATA #REQUIRED
    swedish CDATA #REQUIRED
    >
<word pos="Noun" english="Computer" swedish="Dator">
```

The `#REQUIRED` keyword is similar to a `NOT NULL` constraint in SQL. Optional attributes have the keyword `#IMPLIED`.

Let us look at another example, which shows how to model referential constraints:

```
<!ELEMENT Country EMPTY>
<!ATTLIST Country
    name CDATA #REQUIRED
    currency IDREF #REQUIRED
    >
<!ELEMENT Currency EMPTY>
<!ATTLIST Currency
    code ID #REQUIRED
    name CDATA #REQUIRED
    >
```

The `code` attribute of `Currency` is declared as `ID`, which means that it is an identifier (which moreover has to be unique). The `currency` attribute of `Country` is declared as `IDREF`, which means it must be an identifier declared as `ID` in some other element. However, since `IDREF` does not specify *what* element, it only comes half way in expressing a referential constraint. Some of these problems are solved in alternative format to DTD, called **XML Schema**.

Attributes were originally meant for metadata (such as font size) rather than data. In fact, the recommendation from W3C is to use elements rather than attributes for data (see http://www.w3schools.com/xml/xml_dtd_el_vs_attr.asp). However, since attributes enable some constraints that elements don't, their use for data can sometimes be justified.

9.5 The XPath query language

The XPath language gives a concise notation to extract XML elements. Its syntax is quite similar to Unix directory paths. A grammar for a part of XPath is included in the XML grammar in Figure 12. Here are some examples:

- `/Countries/country` gives all <`country`> elements right under <`Countries`>.
- `/Countries//@name` gives all values of `name` attribute anywhere under <`Countries`>.
- `/Countries/currency/[@name = "dollar"]` gives all <`currency`> elements where `name` is `dollar`.

There is an on-line XPath test program in http://xmlgrid.net/xpath.html

There is a more expressive query language called **XQuery**, which extends XPath. Another possibility is to use **XSLT** (eXtensible Stylesheet Language for Transformations), whose standard use is to convert between XML formats (e.g. from dictionary data to HTML). Writing queries in a host language (in a similar way as in JDBC) is of course also a possibility.

9.6 YAML*

(not covered in the course)

10 Big Data

Big Data is a word used for data whose mere size is a problem. Big Data typically refers to amounts of data that cannot be treated with traditional methods. Data may also be big in the sense that it is generated rapidly, or there is 'big' variation in data types, attributes, etc. At the time of writing, when we just talk about data volume, Big Data is often expected to be at least terabytes (10^{12} bytes), maybe even petabytes (10^{15}). Big Data usually refers to situations where the computations cannot be feasibly performed using a single computer, even a powerful one.

The database world, though, has grown up with need for more data. There are database solutions that manage huge amounts of data in distributed SQL databases, so just the amount of data itself does not serve as a justification for new solutions. However, applications with relaxed requirements of the ACID properties, such as social media, music services, etc. have enabled other types of solutions. For example,it is not so fatal if some comment in the social media is lost, summing up huge amounts of approximate values takes into account some dirty data, and so on.

In Big Data, data is usually **distributed**, maybe to thousands of computers (or millions in the case of companies like Google). The computations must also be **parallelizable** as much as possible. This has implications for big data systems, which makes them different from relational databases:

- simpler queries (e.g. no joins, search on indexed attributes only)
- looser structures (e.g. no tables with rigid schemas)
- less integrity guarantees (e.g. no checking of constraints, no transactions)
- more redundancy ("denormalization" to keep together data that belongs together)

  In database design, we split the data into tables, following a logical design. In traditional database applications, the database system stores the database data on a single server, but with Big Data it could be that a single table is too big for a single server. So now, the tables are further split physically into different servers.

  We will have a look at some of the concepts through Postgres and Cassandra DBMS and then discuss some conclusions and observations on Big Data in databases.

10.1 Partitioning the data

To start with, we need to have a way to split, or partition our tables for distributing them. While there are various rather complicated details in this, we go through a basic arrangement to get the idea.

Suppose we have a table defined as follows:

```
CREATE TABLE rainfall (
    countryname     VARCHAR (50) NOT NULL,
    cityname        VARCHAR (50) NOT NULL,
    rainfall_date   DATE NOT NULL,
    measurement     INT NOT NULL
) PARTITION BY RANGE (rainfall_date);
```

The CREATE TABLE statement introduces, not just the table contents but also the fact that we will partition that table by dates. The partitions are created separately, e.g.:

```
CREATE TABLE rainfall_y2020q1 PARTITION OF rainfall
    FOR VALUES FROM ('2020-01-01') TO ('2020-04-01');
CREATE TABLE rainfall_y2020q2 PARTITION OF rainfall
    FOR VALUES FROM ('2020-04-01') TO ('2020-07-01');
```

which create partitions for the first two quarters of year 2020.

If, then, you execute the following statement:

```
INSERT INTO rainfall VALUES ('Finland','Tampere','2020-05-01', 5);
```

you will notice that the inserted row can be queried from both `rainfall`and `rainfall_y2020q2`but querying `rainfall_y2020_q1` does not find this row.

This means that the value is actually stored in the partition determined by the date. It can be inserted and retrieved through the top level table, though. You may also insert rows directly into the partitions, like

```
INSERT INTO rainfall_y2020q2 VALUES ('Finland','Tampere','2020-05-02', 0);
```

Inserting that row into `rainfall_y2020q2` would give an error due to the partition constraint violation.

So, you may insert data through the top level table or the partition. Note that the FROM values are inclusive while the TO values are not, so

```
INSERT INTO rainfall_y2020q2 VALUES ('Finland','Tampere','2020-07-01', 0);
```

gives an error, while

```
insert into rainfall_y2020q2 values ('Finland','Tampere','2020-04-01', 0);
```

does not.

These partitions could be further partitioned by range of another attribute.

Partitioning has efficiency benefits. The data is split into smaller sets which can be more efficient to process. Also, data management may become easier, since the partitions can be added and deleted individually. There are different values by which partitioning is meaningful, such as ownership of data, time, geographical location, organizational structure, etc.

While a partition may also be called a *shard*, it is more typical to associate *sharding* to distributing the partitions onto different servers. At the time of writing this, distributing the partitions is not a part of standard Postgres distribution but it can be done by using *Foreign data wrappers*, a mechanism to access data in another database, Postgres or something else.

One option to do sharding with Postgres is by using Citus, a system for sharding and replicating the data across servers. There is an open source distribution of Citus, available to all.

Another option is to use the *foreign data wrapper* functionality. Initially, this functionality has been available to access data in other servers and systems. But now, we want to use the foreign data wrappers for PostgreSQL data.

Sharding is also available in MongoDB.

The data can also be replicated in addition to partitioning it. In MongoDB, at the time of writing this, the replicas only serve as backup if a server fails. However, there are systems allowing different copies or replicas are used to access the data at the same time on different servers, thus increasing the responsiveness of the database system. This implies complications with synchronization of updates, a topic beyond the scope of this book.

10.2 MapReduce

A standard way to perform computations on Big Data is to use rounds of *MapReduce* computations. In MapReduce, a round of computations consists of first selecting the relevant data and mapping it by the values into suitable subsets, then followed by the reduce step in which those subsets are used to compute some results. These results may be intermediate and further MapReduce rounds may follow. The **MapReduce** query engine is similar to `map` and `fold` in functional programming. The computations can be made in parallel in different computers. While Map reads a common input and maps the data items onto different sets, Reduce is supposed to be executed independently on each data set. MapReduce was originally developed at Google, to work on top of the **Bigtable** data storage system.

The idea of MapReduce also carries over to SQL and we can demonstrate the idea using the rainfall tables above and SQL.

Let's suppose that we have a table

```
CREATE TABLE rainfall_readings (
    countryname     VARCHAR (50) NOT NULL,
    cityname        VARCHAR (50) NOT NULL,
    rainfall_date   DATE NOT NULL,
    measurement     INT NOT NULL
) PARTITION BY RANGE (rainfall_date);
```

and we want to calculate the average rainfall by city for each quarter of year 2020. The idea is very simple:

- Copy the data from rainfall_readings into rainfall. Thus the data will be split into different tables for different quarters. Suppose we have the partitions for each quarter we are interested in. This is the Map step of MapReduce.
- Execute the summarizing query on rainfall, which means it will be executed on each sub-table. This is the Reduce step of MapReduce.

  MapReduce can improve efficiency. Ultimately, if partitions are distributed onto different servers, then we can speed up Reduce proportionally to the number of servers. Even if we have all the partitions on the same server, we are dealing with smaller data sets which brings some efficiency benefit.

  MapReduce, though, is not just "SQL for big amounts of data". On the contrary, there are queries that SQL allows, but are not feasible with huge distributed data sets, like joining when the join explodes the result set, while at the same time MapReduce is not limited to the SQL language, and it is possible to perform general computations on the data. Also MongoDB, discussed in the previous chapter, supports MapReduce.

The basic MapReduce does not run on SQL databases. MapReduce may utilize simple files and various standard programming languages can be used to program the Map and Reduce functions. The main point is, though, that there is an underlying system that controls the execution, leaving the programmer free of complications of distributed computing.

10.3 The Cassandra DBMS and its query language CQL

We will now take a closer look at Cassandra, which is a hybrid of key-value and column-oriented approaches. "Cassandra is essentially a hybrid between a key-value and a column-oriented (or tabular) database. Its data model is a partitioned row store with tunable consistency... Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key... Other columns may be indexed separately from the primary key."
https://en.wikipedia.org/wiki/Apache_Cassandra

Here is a comparison between Cassandra and relational databases:

	Cassandra	**SQL**
data object	key-value pair=(rowkey, columns)	row
single value	column=(attribute,value,timestamp)	column value
collection	column family	table
database	keyspace	schema, E-R diagram
storage unit	key-value pair	table
query language	CQL	SQL
query engine	MapReduce	relational algebra

Bigtagle is a proprietary system, which was the model of the open-source Cassandra, together with Amazon's **Dynamo** data storage system.[8] The MapReduce implementation used by Cassandra is **Hadoop**.

It is easy to try out Cassandra, if you are familiar with SQL. Let us follow the instructions from the tutorial in

https://wiki.apache.org/cassandra/GettingStarted

Step 1. Download Cassandra from http://cassandra.apache.org/download/
Step 2. Install Cassandra by unpacking the downloaded archive.
Step 3. Start Cassandra server by going to the created directory and giving the command

```
bin/cassandra -f
```

Step 4. Start CQL shell by giving the command

```
bin/cqlsh
```

The following CQL session shows some of the main commands. First time you have to create a keyspace:

```
cqlsh> CREATE KEYSPACE mykeyspace
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
```

[8]The distribution and replication of data in Cassandra is more like in Dynamo.

Take it to use for your subsequent commands:

```
cqlsh> USE mykeyspace ;
```

Create a column family - in later versions kindly called a "table"!

```
> CREATE TABLE Countries (
      name TEXT PRIMARY KEY,
      capital TEXT,
      population INT,
      area INT,
      currency TEXT
   ) ;
```

Insert values for different subsets of the columns:

```
> INSERT INTO Countries
    (name,capital,population,area,currency)
    VALUES ('Sweden','Stockholm',9000000,444000) ;

> INSERT INTO Countries
    (name,capital)
    VALUES ('Norway','Oslo') ;
```

Make your first queries:

```
> SELECT * FROM countries ;

 name   | area   | capital   | currency | population
--------+--------+-----------+----------+------------
 Sweden | 444000 | Stockholm |      SEK |    9000000
 Norway |   null |      Oslo |     null |       null

> SELECT capital, currency FROM Countries WHERE name = 'Sweden' ;

 capital   | currency
-----------+----------
 Stockholm |      SEK
```

Now you may have the illusion of being in SQL! However,

```
> SELECT name FROM Countries WHERE capital = 'Oslo' ;

InvalidRequest: code=2200 [Invalid query] message=
  "No secondary indexes on the restricted columns support the provided operators: "
```

So you can only retrieve indexed values. PRIMARY KEY creates the primary index,
but you can also create a **secondary index**:

```
> CREATE INDEX on Countries(capital) ;

> SELECT name FROM Countries WHERE capital = 'Oslo' ;

 name
--------
 Norway
```

Most SQL constraints have no counterparts, but PRIMARY KEY does:

```
> INSERT INTO countries
    (capital,population,area,currency)
    VALUES ('Helsinki',5000000, 337000,'EUR') ;

InvalidRequest: code=2200 [Invalid query] message=
  "Missing mandatory PRIMARY KEY part name"
```

A complete grammar of CQL can be found in

https://cassandra.apache.org/doc/cql/CQL.html

It is much simpler than the SQL grammar. But at least my version of CQL shell does not support the full set of queries.

10.4 Further considerations

The forerunner in Big Data computations has been Google, whose solutions have been partly reproduced on the open-source Hadoop platform.

The classic way to improve query processing in SQL databases is adding indices that allow faster lookup of the required data. There are physical factors, like the computer equipment, and parameters related to e.g. available memory that affect the computing. When dealing with distributed data, there are further complications. Distributed indexing is hard to maintain. Hashing offers a more straightforward and efficient way to place and thereafter seek for the data.

Hashing over a set of servers and replicating the data offers fault-tolerance and thereby increases availability of database service. This is the principle of **key-value stores** such as Amazon's Big Data database Dynamo or open source Big Data database Riak. There, the data is replicated and hashed on disk based on key values. To allow scalability, it is possible to position data into *virtual servers* that are allocated to physical servers. Adding more physical servers then just means re-allocation of where virtual servers reside. Data retrieval, on the other hand, means traversing through data on different servers. The basic choice to access the data is writing a MapReduce program.

HBase is an open-source commmunity-driven implementation based on Google's BigTable database. It provides a low-level access to the data: all data is stored as bytes and the application has to provide its own transformation functions to interpret the data. Storing data is based on a primitive put function while reading the data is based on a primitive get function.

Disk is still a slow component of computer systems. As a solution to this, Hbase organizes the data on disk physically sequentially to support fast retrieval based on the primary key. This means that if an application needs large amounts of data, the system only needs to locate the data start position once, after which it can read the

data sequentially in order, which is relatively fast. As an example, suppose that we position timestamp first in the primary key, and know the time interval for which we need the data. This allows fast retrieval and storage. Building secondary indices does not give similar advantage, as the data is only organized on the primary key.

The obvious general drawback is that there may be a need to reorganize the data, and this can take a reasonably - or indeed unreasonably - long. This can even result into a service break. The data replication happens through the Hadoop HDFS file system, a replicating Big Data file system.

This could happen in a number of ways:

- Data can be distributed by some attribute values, like city data could be distributed by country or by continent.
- Data can be stored by columns (attributes) rather than by rows. This may be practical e.g. for some analysis tasks that need all values for some columns but none for some others.
- Columns that are likely to be used together can be stored together.
- Rows that are likely to be used together can be stored together.
- The data cab be split in a seemingly random way into different servers, e.g. using a hash function that given a key value gives an address containing the server used.
- Some data can be replicated ie stored on several servers at the same time. This improves accessibility but complicates maintenance of data.

A Appendix: SQL in a nutshell

Figure 14 shows a grammar of SQL. It is not full SQL, but it does contain all those constructs that are used in this course for database definitions and queries. The syntax for triggers, indexes, authorizations, and transactions will be covered in later chapters.

In addition to the standard constructs, different DBMSs contain their own ones, thus creating "dialects" of SQL. They may even omit some standard constructs. In this respect, PostgreSQL is closer to the standard than many other systems.

The grammar notation is aimed for human readers. It is hence not completely formal. A full formal grammar can be found e.g. in PostgreSQL references:

> http://www.postgresql.org/docs/9.5/interactive/index.html

Another place to look is the Query Converter source file

> https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/MinSQL.bnf

It is roughly equivalent to Figure 14, and hence incomplete. But it is completely formal and is used for implementing an SQL parser.[9]

The grammar is BNF (Backus Naur form) with the following conventions:

- CAPITAL words are SQL keywords, to take literally
- small character words are names of syntactic categories, defined each in their own rules
- | separates alternatives
- `+` means one or more, separated by commas
- `*` means zero or more, separated by commas
- `?` means zero or one
- in the beginning of a line, `+ * ?` operate on the whole line; elsewhere, they operate on the word just before
- `##` start comments, which explain unexpected notation or behaviour
- text in double quotes means literal code, e.g. `"*"` means the operator `*`
- other symbols, e.g. parentheses, also mean literal code (quotes are used only in some cases, to separate code from grammar notation)
- parentheses can be added to disambiguate the scopes of operators

Another important aspect of SQL syntax is **case insensitivity**:

- **keywords** are usually written with capitals, but can be written by any combinations of capital and small letters
- the same concerns **identifiers**, i.e. names of tables, attributes, constraints
- however, **string literals** in single quotes are case sensitive

[9] The parser is generated by using the BNF Converter tool, http://bnfc.digitalgrammars.com/ which is also used for the relational algebra and XML parsers in qconv.

```
statement ::=
      CREATE TABLE tablename (
    * attribute type inlineconstraint*
    * [CONSTRAINT name]? constraint deferrable?
    ) ;
  |
      DROP TABLE tablename ;
  |
      INSERT INTO tablename tableplaces? values ;
  |
      DELETE FROM tablename
    ? WHERE condition  ;
  |
      UPDATE tablename
      SET setting+
    ? WHERE condition ;
  |
      query ;
  |
      CREATE VIEW viewname
      AS ( query ) ;
  |
      ALTER TABLE tablename
    + alteration ;
  |
      COPY tablename FROM filepath ;
        ## postgresql-specific, tab-separated

query ::=
      SELECT DISTINCT? columns
    ? FROM table+
    ? WHERE condition
    ? GROUP BY attribute+
    ? HAVING condition
    ? ORDER BY attributeorder+
  |
      query setoperation query
  |
      query ORDER BY attributeorder+
        ## no previous ORDER in query
  |
      WITH localdef+ query

table ::=
      tablename
  |   table AS? tablename  ## only one iteration allowed
  |   ( query ) AS? tablename
  |   table jointype JOIN table ON condition
  |   table jointype JOIN table USING (attribute+)
  |   table NATURAL jointype JOIN table

condition ::=
      expression comparison compared
  |   expression NOT? BETWEEN expression AND expression
  |   condition boolean condition
  |   expression NOT? LIKE 'pattern*'
  |   expression NOT? IN values
  |   NOT? EXISTS ( query )
  |   expression IS NOT? NULL
  |   NOT ( condition )

expression ::=
      attribute
  |   tablename.attribute
  |   value
  |   expression operation expression
  |   aggregation ( DISTINCT? *|attribute)
  |   ( query )

value ::=
      integer | float | 'string'
  |   value operation value
  |   NULL

boolean ::=
      AND | OR

type ::=
     CHAR ( integer ) | VARCHAR ( integer ) | TEXT
   | INT | FLOAT

inlineconstraint ::=     ## not separated by commas!
     PRIMARY KEY
   | REFERENCES tablename ( attribute ) policy*
   | UNIQUE | NOT NULL
   | CHECK ( condition )
   | DEFAULT value

constraint ::=
     PRIMARY KEY ( attribute+ )
   | FOREIGN KEY ( attribute+ )
       REFERENCES tablename ( attribute+ ) policy*
   | UNIQUE ( attribute+ ) | NOT NULL ( attribute )
   | CHECK ( condition )

policy ::=
     ON DELETE|UPDATE CASCADE|SET NULL
deferrable ::=
     NOT? DEFERRABLE (INITIALLY DEFERRED|IMMEDIATE)?
tableplaces ::=
     ( attribute+ )

values ::=
     VALUES ( value+ )  ## keyword VALUES only in INSERT
   | ( query )

setting ::=
     attribute = value

alteration ::=
     ADD COLUMN attribute type inlineconstraint*
   | DROP COLUMN attribute

localdef ::=
     WITH tablename AS ( query )

columns ::=
     "*"
   | column+

column ::=
     expression
   | expression AS name

attributeorder ::=
     attribute (DESC|ASC)?

setoperation ::=
     UNION | INTERSECT | EXCEPT

jointype ::=
     LEFT|RIGHT|FULL OUTER?
   | INNER?

comparison ::=
     = | < | > | <> | <= | >=

compared ::=
     expression
   | ALL|ANY values

operation ::=
     "+" | "-" | "*" | "/" | "%"
   | "||"

pattern ::=
    % | _ | character  ## match any string/char
  | [ character* ]
  | [^ character* ]

aggregation ::=
    MAX | MIN | AVG | COUNT | SUM
```

Figure 14: A grammar of the main SQL constructs.