Databases in 127 Pages

Jyrki Nummenmaa and Aarne Ranta

Faculty of Natural Sciences, University of Tampere
CSE, Chalmers University of Technology and University of Gothenburg
Draft Version February 26, 2018

Contents

Note on the the current edition

These notes have been expanded from Aarne's 2017 version by Jyrki, who has also reorganized the chapters. Many more changes are coming during Spring 2018! This is in particular the case with exercises, which are currently separate from the book but available through the course web page.

Tampere and Gothenburg, February 26, 2018

Jyrki Nummenmaa and Aarne Ranta

Preface

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. To help you with these exercises, we have developed a tool, Query Converter, which can be used on-line to explore the theoretical concepts and link them with practical database applications.

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 prefereable even though the student would have time 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. This means that, when learning new mathematical and programming concepts, you have to learn new content 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.

The book can be read in the given order of chapters. It is, however, possible to deviate from this to some extent. The following diagram shows the relationships between chapters and following the diagram it is perfectly doable to pick another order or, depending on the readers interest, skip some parts of the book.

(Diagram TODO)

This 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, March 2016

Aarne Ranta

1 Introduction*

This chapter is an overview of the field of databases and of this course. In addition to the material printed here, the lecture will also talk about practical questions such as assignments, exercises, and the exam. This information can be found on the course web page. The goal of this chapter (and the whole first lecture) is to give you a clear picture of what you are expected to do and learn during the course.

1.1 Data vs. programs

Computers run programs that process data. Sometimes this data comes from user interaction and is thrown away after the program is run. But often the data must be stored for a longer time, so that it can be accessed again. Banks, for instance, have to store the data about bank accounts so that no penny is lost.

It is typical that 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.

A **database** is any collection of data that can be accessed and processed by computer programs. A **database system** consists of a database storage and software through which the data in the database is accessed. The 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. For instance, English texts describing the data would be both too slow and too inaccurate. But the data structure must also be **generic** enough so that it can be accessed in different ways. For instance, the data structures of some advanced programming language may be too hard to access from programs written in other languages. A further requirement is that the database should support multiple concurrent users.

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. 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 databases**, which were popular in the 1960's but were deemed too complicated by the proponents of relational databases. On the other end, **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. Speed is much more important, and then the traditional relational models can be too rich. Non-relational approaches are known as **NoSQL**, 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. However, as we will see, most 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. Another reason is that SQL implementations are highly optimized and reliable. A general purpose programmer would have a hard time matching the performance of them. Losing or destroying data would also be a serious risk.

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, designed for large amounts of data and database analytics.

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 for the course in 2016 is PostgreSQL. Earlier courses have used Oracle, so this is in a way an experiment. The main reasons to try PostgreSQL are the following advantages over Oracle:

- it follows the standard more closely
- 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. We will explain the main language constructs of SQL. We will define SQL tables. We will build a database by insertions. We will query it by selections, projections, joins, renamings, unions, intersections, and SQL groupings and aggregations. We will also take a look at low-level manipulations of strings and at the different datatypes of SQL.

Chapter 3: Entity-Relationship diagrams

A popular device in modelling is **E-R diagrams** (Entity-Relationship diagrams). This chapter explains how different kinds of data are modelled by E-R diagrams. We will also tell how E-R diagrams can be constructed from descriptive texts. Finally, we will explain how they are, almost mechanically, converted to relational schemes (and thereby eventually to SQL).

Chapter 4: Data modelling with relations

This chapter is about the mathematical concepts that underlie relational databases. Not all data is "naturally" relational, so that some encoding is necessary. Many things can go wrong in the encoding, and lead to redundancy or even to unintended data loss. This lecture gives several examples of different kinds of data. It introduces the notion of **relational schemas**, which are in SQL implemented by **table definitions**. But the level here is a bit more abstract than SQL. This chapter also explains the basics of the mathematics of relations, which are derived from set theory.

Chapter 5: Dependencies and database design

This chapter builds on the relational model and explains a technique that helps design consistent databases. 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. In this chapter, we study how to use dependencies to design databases free of theses problems.

Lecture 6: Relational algebra and query compilation

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, each denoted by Greek letters. Being so simple, relational algebra is more difficult to use for complex queries than SQL. But for the very same reason, it is easier to analyse and optimize. Relational algebra is therefore useful as an intermediate language in a DBMS. 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.

Chapter 7: 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 lecture, we will look at how SQL is embedded in Java. We will also cover some pitfalls in embedding. For instance **SQL injection** is a security hole 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 SQL code that deletes all data from a student database.

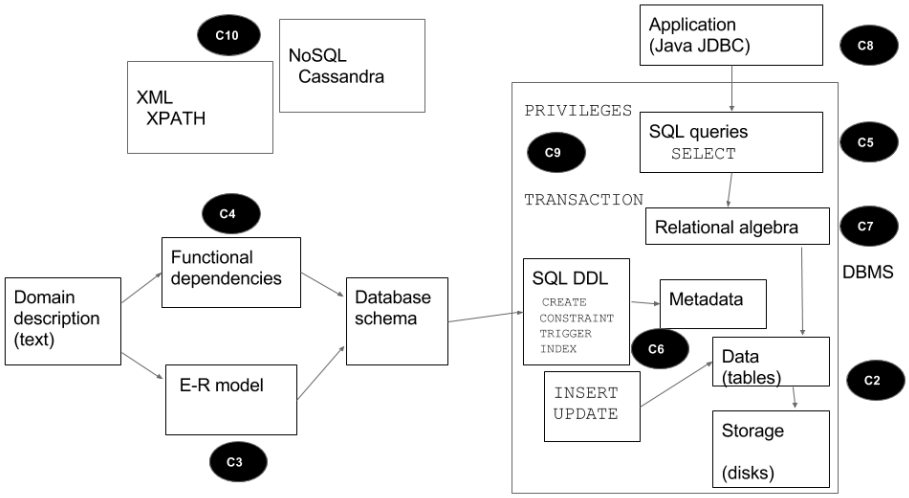
We also take a deeper look at inserts, updates, and deletions, in the presence of constraints. The integrity constraints of the database may restrict these actions or even prohibit them. An important problem is that when one piece of data is changed, some others may need to be changed as well. For instance, when value is deleted or updated, how should this affect other rows that 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. For situations like this, DBMSs support **triggers**, which are programs that do many SQL actions at once.

Chapter 8: Introduction to alternative data models

The relational data model has been dominating the database world for a long time. But there are alternative models, some of which are gaining popularity. XML is an old model, often seen as a language for documents rather than data. In this perspective, it is a generalization of HTML. But it is a very powerful generalization, which can be used for any structured data. XML data objects need not be just tuples, but they can be arbitrary trees. XML also has designated query languages, such as XPath and XQuery. This chapter introduces XML and gives a summary of XPath. On the other end of the scale, 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.



2 Tables and SQL

This chapter is about tables as a basic abstraction for present-day databases. We will study them using a common 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. This chapter covers two lectures. At first lecture, we will 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. We will query it by selections, projections, renamings, unions, intersections, groupings, and aggregations. We will also take a look at low-level manipulations of strings and at the different datatypes of SQL. The second lecture generalizes the treatment to many tables. This generalization involves just a few new SQL constructs (foreign keys, cartesian products, 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 the standard 3-character code and continents area represented with shorthand expressions for Africa, Europe, South America, North America, and Asia.

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 a title row, describing the data stored in the subsequent rows. Each of those subsequent rows, in turn, contains data of one country, as introduced in the title row: the country name, capital, area, population, the continent where the country resides (if any), and the official currency used.

The intuitive idea of a table is the basis of common database systems. The typical 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 this very form: "there exists a country named in the row, with the given capital, area and population, residing in the given continent, and using the given official currency". Those familiar with logic should see a similarity between rows and logical propositions. 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.

All the vertical columns in the table seem to have similar formats. Some columns are strings (name, capital), some numbers (area, population). Some are strings with a fixed length (currency length 3, continent length 2). Moreover, each country seems intuitively have a unique name, but different countries might have the same area or population, use the same currency, be situated on the same continent, and even, at least theoretically, even have the same capital name.

These conclusions are superficial, drawn from the outlook of the table. From that table we cannot know if some currency has 4-character code or if we would want to store areas as a non-integer decimal values. We don't even know if all values must exist: in the above table, Cuba has no continent, since it is an island outside continents. However, 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.

In this chapter, we will learn to create and manipulate database tables using the SQL language, commonly used in both commercial and open-source systems. The first thing to learn is how to create a table and to populate it with values. After that, we will learn how to write **queries**, that is, ask questions about the tables.

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 line in Windows.

In your command line shell, you can start PostgreSQL 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`. [1]

[1] If you use the school's PostgreSQL installation, you already have a database created, and you should work under that.

2.3 Creating an SQL table

Here a statement in the SQL language to create a table for data on countries. It follows partly, but not completely, our superficial observations above.

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

In that SQL statement we give a name to a table we create (`Countries`), and introduce the columns of the table, also called attributes of the table.

The first line of this `CREATE TABLE` statement just gives the keywords to create a table plus a name for the table. The name is supposed to be different from any already existing table in our database. In this particular case we have not defined any tables before, but if we the same CREATE TABLE statement twice, the second one should lead to an error.

Each attribute is given a name and a data type. While `INT` goes for integer values and `FLOAT` for floating-point representation of decimal values, there are different datatype definitions for textual data: `TEXT` is the most general, it allows for any textual data, but `CHAR(2)` and `CHAR(3)` introduce the length of the text.

Generally, SQL has several types for strings: CHAR(n), VARCHAR(n), and TEXT. If the length is variable or not known, then it is safest to choose `TEXT`. In earlier times, and in other DBMSs, these types may have performance errors. However, the 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 will check this property, if we use CHAR(n).

The `PRIMARY KEY` after `name TEXT` says that name is a primary key for the table. This means that each row in the table must have a name, and no name can appear on more than one row - in other words, that the name must be **unique** in the table. The `NOT NULL` constraints say that there must be a value for the capital and currency in all rows.

`PRIMARY KEY` implies `NOT NULL`, but the other columns (area, population, continent) need not have values. When a value is missing, we sometimes say that it is `NULL`. Such `NULL` values can mean two things: that we don't know the value, or that the value does not exist.

Names of tables must be unique in a SQL database. Therefore, giving a subsequent `CREATE TABLE` commands with the same table name leads to an error. We should also notice that SQL 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`

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, since all the data in the table gets lost!

2.4 Grammar rules for SQL

There are different variations or dialects of the SQL language, and our aim is not to cover them all nor compare them, but to introduce a subset suitable for our course 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 "what all" one can do with the statements. Without raising the abstraction with a grammar of our grammars, we rather introduce the grammar structures as they are used.

To represent the grammars, we use BNF (Backus Naur form, and you may continue reading even if you do not know what it means) 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

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

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

2.5 A further analysis of CREATE TABLE statements

In the grammar for `CREATE TABLE` statements, the first line just gives the keywords to create a table plus a name for the table. The second line introduces zero or more attributes. Their names are supposed to be unique for the table. The type, instead, is a syntactic category defined below, and so are the inline constraints, that is, the constraints given on the same line with the constrained attribute.

```
type ::=
  CHAR ( integer ) | VARCHAR ( integer ) | TEXT | INT | FLOAT | BOOLEAN
inlineconstraint ::=
    PRIMARY KEY | UNIQUE | NOT NULL | DEFAULT value
  | REFERENCES tablename ( attribute )
```

An example of an inline constraint is `PRIMARY KEY` after `name TEXT`, it says that name is a primary key for the table. Another example that appeared in the previous section is the `NOT NULL` inline constraints.

The other inline constraints given now are `UNIQUE` which means there must be a unique value for that attribute in each row, and `DEFAULT value` which allows us to define a default value for data insertion. `REFERENCES` is used when the attribute refers to some other table; Section 2.13 will explain this.

Some constraints are not for a single attribute value. The primary key may be a **composite key**, that is, composed from several attributes, which is useful when a combination of attribute values is unique even if none of the attributes alone is unique. Since such constraints refer to several attibutes, they must be separately from the introduction of the respective attributes, in the `constraint` part of the `CREATE TABLE` statement. Their grammar is:

```
constraint ::=
    PRIMARY KEY ( attribute+ )
  | UNIQUE ( attribute+ )
  | NOT NULL ( attribute )
  | FOREIGN KEY tablename ( attribute+ )
```

As an example of composite keys, let us suppose, for the moment, that country names are only unique within a continent and we have to use the `country, continent` pair as a primary key. Then we would have a constraint

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

If countries were identified, instead, by a number, we could still state the uniqueness of continent,name pairs by stating the following constraint, to which we now give a name.

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

And what might the name be used for? Well, some database management systems use the name when they report situations where the constraint would be violated. What, then, is the difference of PRIMARY KEY and UNIQUE in SQL? The difference, if any, is that some systems are more reluctant to change or remove from existing tables their key definitions than their unique definitions.

There are also further constraints, not relevant to us now.

2.6 Inserting rows to a table

A new table, when created, is empty. To insert table rows, we use the `INSERT` statement. The statements below create the contents that we saw before, just the order is different. However, we assume that the interpretation of data content is "per row" and, thus, the order of the rows carries no meaning. Equally well, you may give the statements in any order. The most convenient way is to prepare a file that contains the statements and then read in the statements in a SQL interface.

```
INSERT INTO Countries VALUES ('Sweden','Stockholm',449964,9555893,'EU','SEK') ;
INSERT INTO Countries VALUES ('Finland','Helsinki',337030,5244000,'EU','EUR') ;
INSERT INTO Countries VALUES ('Tanzania','Dodoma',945087,41892895,'AF','TZS') ;
INSERT INTO Countries VALUES ('Peru','Lima',1285220,29907003,'SA','PEN') ;
INSERT INTO Countries VALUES ('Chile','Santiago',756950,16746491,'SA','CLP') ;
INSERT INTO Countries VALUES ('China','Beijing',9596960,1330044000,'AS','CNY') ;
INSERT INTO Countries VALUES ('Slovenia','Ljubljana',20273,2007000,'EU','EUR') ;
INSERT INTO Countries VALUES ('Greece','Athens',131940,11000000,'EU','EUR') ;
INSERT INTO Countries VALUES ('Cuba','Havana',110860,11423000,NULL,'CUP') ;
```

Some database systems allow to combine several rows into the same statement:

```
INSERT INTO Countries VALUES
 ('Sweden','Stockholm',449964,9555893,'EU','SEK'),
 ('Finland','Helsinki',337030,5244000,'EU','EUR'),
 ('Tanzania','Dodoma',945087,41892895,'AF','TZS'),
 ('Peru','Lima',1285220,29907003,'SA','PEN'),
 ('Chile','Santiago',756950,16746491,'SA','CLP'),
 ('China','Beijing',9596960,1330044000,'AS','CNY'),
 ('Slovenia','Ljubljana',20273,2007000,'EU','EUR'),
 ('Greece','Athens',131940,11000000,'EU','EUR'),
 ('Cuba','Havana',110860,11423000,NULL,'CUP') ;
```

Even though the order of rows is unimportant, here the order of values within each row is highly important, and it is assumed to follow the order of the attributes as given when the table has been created.

The grammar for basic insert statement is given below.

```
INSERT INTO tablename tableplaces? value+ ;
tablespaces = ( attribute+ )
value ::=
      integer | float | 'string'
  |   value operation value
  |   NULL
operation ::= "+" | "-" | "*" | "/" | "%" | "||"
```

The operations listed stand for arithmetic addition (+), subtraction (-), multiplication (*), division (/), remainder of integer division (%), and string concatenation (||), and the tablespaces definition lets us specify the attributes for which values are given, as well as their default order. This means that we can, e.g. try the following additions.

```
INSERT INTO countries VALUES ('Cuba1','Ha' || 'vana',
                              110000 + 860,11423000,NULL,'CUP') ;
INSERT INTO countries (capital, name) VALUES ('Havana','Cuba2') ;
INSERT INTO countries VALUES ('Cuba3','Havana',110860,11423000,,'CUP') ;
```

In addition to experimenting with the arithmetics and string concatenation, we also used three different ways to introduce NULL values. In the first, we write NULL explicitly in the place of a value. In the second, we only give values for the country name and capital, and the rest of the values will be NULL. The third option is to leave a value out between commas, in which case a NULL will be stored.

In the above cases the NULL value was used indicating that Cuba is not on any continent. Below, you see an example where NULL is used for a value that exists but we consider not known.

```
INSERT INTO Countries Values
  ('India', 'New Delhi', 3287590, NULL, 'AS', 'INR');
```

Even though we had different ways to define strings, in SQL the values for all those types are string literals in single quotes (e.g. `'foo bar'`). Spaces are preserved.

What can go wrong, if we write a grammatically correct insert statement? Many things, of course. 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 (more than one row with the same primary key value), and uniqueness constraint may be violated. You are urged to try out these, to see what happens.

In PostgreSQL, there is a quick way to insert values from tab-separated files:

```
COPY tablename FROM filepath
```

Notice that a complete filepath is required. 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,currencycode,currencyname)
```

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

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

The file

http://www.cse.chalmers.se/edu/year/2018/course/TDA357/VT2018/notes/countries.tsv

can be used for this purpose. It is extracted from the Geonames database, http://www.geonames.org/

An alternative method is to generate lots of INSERT commands into a file. Such a file can also include other SQL commands - you can, for instance, save all your work in it. Then you can build your database, or parts of it, with the PostgreSQL command

```
\i file.sql
```

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 below.

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 the a part of the contents of rows without having to delete and insert whose 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 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 motivation for storing the data is to seach necessary information from it. This is done with the `SELECT FROM WHERE` statements. in SQL. We study that statement little by little. First, the statement

```
SELECT * FROM countries
```

will output the whole countries table. Now, `*` implies that all attributes are selected for the result.

```
SELECT currency, continent FROM countries ;
```

will query also the currency, continent pairs, as follows.

```
|| currency | continent |
| CUP | NULL |
| EUR | EU |
| EUR | EU |
| CNY | AS |
| CLP | SA |
| PEN | SA |
| TZS | AF |
| EUR | EU |
| SEK | EU |
```

SQL database systems typically require that all tables have a primary key. This means that there cannot be duplicate rows. However, duplicates may appear in tables produced as answers to SQL queries, as we can see above. Using the `DISTINCT` keyword we can eliminate duplicates and get a set of rows in the answer.

```
SELECT DISTINCT currency, continent FROM countries ;
```

gives the following answer:

```
|| currency | continent |
| CUP | NULL |
| EUR | EU |
| CNY | AS |
| CLP | SA |
| PEN | SA |
| TZS | AF |
| SEK | EU |
```

It is possible to restrict both the columns by name and rows by condition, e.g. to query the names and capitals of South American countries.

```
SELECT name, capital FROM countries WHERE continent == 'SA'
```

giving the answer

```
| name   | capital |
| 'Chile' | 'Santiago'|
| 'Peru'  | 'Lima'|
```

If the WHERE condition evaluates to true on a row, then that row will be included in the result set, and otherwise it will not. The `WHERE` conditions allow comparing values from different attributes, such as

```
SELECT name, capital FROM countries WHERE name == capital
```

which gives no results with our data as no country has a capital with the same name as the country itself.

Notice that only the SELECT part is compulsory; you can use it on an expression that doesn't refer to any table:

```
SELECT 2+2
```

Also, you may create new columns and new values. For instance, the following selects big countries with size just marked `big`:

```
SELECT name, 'big' AS size
FROM Countries
WHERE population > 50000000
```

or

```
SELECT capital, 'South American Capital' AS sa_capital
FROM countries
WHERE continent = 'SA' ;
```

In this section we will only consider conditions that apply to individual rows and do not compare several rows. The grammar for these simple `SELECT` queries is given below:

```
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

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

The condition `s LIKE p` compares the string `s` with the **pattern** `p`. The pattern can use **wildcards** _ (for any character) and `%` (for any substring). Thus

```
WHERE name LIKE '%en'
```

is satisfied by all countries whose name ends with "en", e.g. Sweden. So, we can write conditions such as

```
name = capital AND NOT (population > area + 100000)
capital LIKE '__vana'
1 == 3
1 < 3 OR 1==3
```

You are encouraged to try them out. In complicated expressions combining AND, OR, and NOT without parentheses, NOT has highest priority and is evaluated first, AND after that, and finally OR.

The NULL value is quite special in comparisons. It fails every comparison apart from `is NULL`. 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 = 'XY'
SELECT * FROM countries WHERE name = 'Cuba' AND continent <> 'XY'
SELECT * FROM countries WHERE name = 'Cuba' AND (continent = 'XY' OR
                                                 continent <> 'XY')
SELECT * FROM countries WHERE name = 'Cuba' AND continent is not NULL
SELECT * FROM countries WHERE name = 'Cuba' AND continent is NULL
```

In case the column names do not seem appropriate in the result of the query, it is possible to give them new names as follows.

```
SELECT name AS country, capital, area AS terrain_size from countries ;
```

2.9 Sets in SQL queries

As we noticed, the SQL tables with primary keys have sets of rows in them. However, the resulting rows of SQL queries may not always be a set. Using the `DISTINCT` keyword, duplicates were removed, thus guaranteeing a set of rows. SQL includes set operations, which allow for set union `UNION`, set difference `EXCEPT`, and set intersection `INTERSECT`. If set operations are used, then the results are automatically interpreted as sets, and duplicates are removed. Set operations can be applied to queries, even on the top level.

UNION, INTERSECT, EXCEPT correspond to mathematical set operations $\cup, \cap, -$, however the intuition is simple and just trying these expressions out should clarify the basic idea. Thus they can only be applied to tables "of the same type", i.e. tuples with the same number of elements of the compatible types. The attribute names, however, need not match: it is meaningful to write

```
SELECT capital FROM Countries
  UNION
SELECT name FROM Countries
```

The following query finds all currencies used either in North America or in Asia.

```
SELECT currency FROM countries WHERE continent = "NA"
  UNION
SELECT currency FROM countries WHERE continent = "AS"
```

To get currencies used in both North America and Asia, `UNION` needs to be replaced by `INTERSECT`, and, finally, currencies used in North America but not in Asia, `UNION` needs to be replaced by `EXCEPT`.

It is also possible to use the `ALL` keyword for set operations without using sets! This means, that the duplicates are preserved when otherwise appropriate. The following query keeps all duplicates.

```
SELECT currency FROM countries WHERE continent = "NA"
  UNION ALL
SELECT currency FROM countries WHERE continent = "AS"
```

For set operations, the values need to have the same datatype, e.g. we cannot make a set of integers and strings. The names for the columns are taken from the first set, and they do no tneed to be the same. But the number of columns must be the same (a conceivable alternative would be to pad the shorter tuples with `NULL` values, but this is not what happens).

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') ;
```

We can also use other comparison operators instead of IN. However, comparisons such as equality work between values, not between sets, and particularly not between sets and values. A comparison between a single value and a set will work in this context, though, if the set has only one value. Sometimes this can be known.

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

In this case, we could as well use the query

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

We can, however, use a comparison that targets all the values in a set, like selecting the country names for countries that have a population greater than all the population values in South America, as follows

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

Here is a useful idiom using set operations: the query

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

shows the populations of countries as 'big' or 'small', suppressing the exact numeric population.

2.10 Sorting the results

Sorting (ORDER BY) lists a set of attributes considered in lexicographical order. The direction of sorting can be specified for each attribute as DESC (descending) or ASC (ascending), where ASC is the default. Thus the following query sorts countries primarily by the currency in ascending order, secondarily by size in descending order:

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

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.

2.11 Aggregation and grouping

Aggregation functions mean functions that are used to aggregate values from several rows into single values, such as sums and averages. The usual aggregation functions are `COUNT` (of rows), `SUM` (of values), `MIN` (smallest value), `MAX` (biggest value), and `AVG` (average). This way, we can for instance get the minimum, maximum, and average population for countries in South America, and, additionally, the information on how many such countries are in our table.

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

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 ;
```

`WHERE` is used to write conditions on which rows are selected to the result. We can also restrict the result set using values obtained in aggregation, using the `HAVING` construct, e.g. to calculate the statistics only when there are rows from at least 2 countries of a continent:

```
SELECT
  MIN(population), MAX(population), AVG(population)
FROM countries
GROUP BY continent
HAVING COUNT (population) > 1;
```

Removing duplicates by the DISTINCT keyword also works inside aggregations:

```
SELECT DISTINCT currency

SELECT COUNT(DISTINCT currency)
```

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 of currencies. But what are the other attributes? The original attributes of R won't do, because each of them may appear many times. For instance, there are many EUR countries. So what is the use of this construction?

The full 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
- aggregation functions on the other attributes

In other words, the new relation has these aggregation functions as its non-key attributes. Here is an example:

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

currency | count
----------+-------
XCD      |     8
ETB      |     1
HUF      |     1
...
```

Now, most rows in this table will have count 1. We may be interested in only those currencies that are used by more than one country. The standard way of doing this is by a subquery:

```
SELECT *
FROM (
  SELECT currency, COUNT(name) AS number
  FROM Countries
  GROUP BY currency) AS C
WHERE number > 1
```

This shows clearly that `GROUP BY` really forms a table. But SQL also provides a shorthand way of expressing conditions on the groups (i.e. the rows of the relation formed by GROUP BY): HAVING:

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

If you want to order this from the biggest to the smallest count, just add the line

```
ORDER BY COUNT(name) DESC
```

```
currency | count
----------+-------
 EUR      |    35
 USD      |    17
 XOF      |     8
...
```

The other aggregation functions (SUM, AVG, MAX, MIN) work in the same way. The grouped table can have more than one of them:

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

As a final subtlety: the relation formed by GROUP BY also contains the aggregations used in the HAVING clause or the ORDER BY clause:

```
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 thus a very complex operator, because one has to look at many different places to see exactly what relation it forms. We will get more clear about this when looking at relational algebra and query compilation in Chapter 6

2.12 Using data from several tables

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

country	**capital**	**area**	**population**	**continent**	**currency**	**usd_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	NA	CUP	0.038
China	Beijing	9596960	1330044000	AS	CNY	0.150
Chile	Santiago	756950	16746491	SA	CLP	0.001

This data needs to be updated daily, as the currency rates are constantly changing. When many countries use the same currency (e.g.\ `EUR`), the same update has to be performed on several rows, which causes a problem: what about if we forget to update the value on all of those rows? We will then have an **inconsistency** in the value, 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. The chapters on database design (Chapters 3,5) will talk more about how to divide data into separate tables; the basic intuition that we follow is avoidance of redundancy that may lead to inconsistencies.

So, instead of adding a new attribute to the table on countries, we create a new table which contains just the information on the values of currencies. This table may also contain other information on currencies, such as their full names:

currency	**name**	**usd_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 table is

```
CREATE TABLE currencies (
  code TEXT PRIMARY KEY,
  name TEXT,
  usd_value FLOAT )
```

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

Let us star with a table that shows, for each country, its capital and its currency code:

```
SELECT capital, code
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 about 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. 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 in the table (recalling that the `FROM` part, where the names are introduced, is executed 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: it has $9 \times 7 = 63$ rows, because `Countries` has 9 rows and `Currencies` has 7. But the `WHERE` clause shrinks its size to 9, because the `currency` is the same `code` on only 9 of the rows. [2]

The condition is in the `WHERE` part is called a **join condition**, as it controls how rows from tables are joined together. We can state also other conditions in the `WHERE` part, e.g. `WHERE co.currency = cu.currency AND co.continent = 'EU'` would only include European countries. Leaving out a join condition will produce all pairs of rows - the whole cartesian project - in the result. The reader is urged to try out e.g.

```
SELECT name.capital, currencies.name
FROM countries, currencies
```

and see the big table that results. (Hint: you can use `COUNT(*)` on the `SELECT` line to see the number of rows created.)

2.13 Foreign keys and references

The natural way to join data from two table 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 to always be the case, we can use a `FOREIGN KEY` clause within the `CREATE TABLE` statement for `Countries`. We can either do this next to the column declaration,

```
currency REFERENCES Currencies(code)
```

or add a constraint to the end of the statement,

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

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

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 uniquely found 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 so `Countries` that would violate the foreign key requirement. 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. Consider the following query listing the names of pairs of countries that have the same currency:

[2] 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.

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

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 will be data not participating in the join. In the next section, we will have a look at particular SQL statements to join data, which also deal with the problem of rows not joining with any rows in the other table.

2.14 Join operations (JOIN)

Combining data from several tables has the complication that there may be rows that are not selected as they do not combine with any row from the other table(s). Sometimes we want to see those rows, too, in the result. In such a case, a natural choice is to create a new row which has some `NULL` values in the place of values from other table(s). Such an operation is called **outer join**, whereas only selecting tuples that combine successfully is called an **inner join**. A **natural join** just matches tuples by equality on attributes with the same name. Examples below will clarify this.

The syntax of join operations is rich, as there are 24 different join operations:

```
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
```

In addition, cartesian product itself is a kind of a join. It is also called `CROSS JOIN`, but we will use the ordinary notation with commas instead.

Luckily, the JOINs have a compositional meaning. INNER is the simplest join type, and the keyword can be omitted without change of meaning. This `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 attribute exists 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 `Nordics` that shows the five Nordic countries with their capitals, and another table, `Natos`, which shows the NATO countries with their currencies. Choosing just suitable parts of the tables is enough to illustrate the effects of different joins:

```
Nordics                    Natos
 country | capital           country  | currency
 --------+--------           ---------+--------
 Sweden  | Stockholm         Norway   | NOK
 Norway  | Oslo              Germany  | EUR
```

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

Nordics INNER JOIN Natos ON Nordics.country = Natos.country
 country  | capital   | country  | currency
 -------- +-----------+----------+----
 Norway   | Oslo      | Norway   | NOK


Nordics NATURAL JOIN Natos,
Nordics INNER JOIN Natos USING(country)
 country  | capital   | currency
 -------- +-----------+----------
 Norway   | Oslo      | NOK


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

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

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

2.15 Local definitions and views

Local definitions (WITH clauses) are a simple shorthand mechanism for queries. Thus

```
WITH
  EuroCountries AS (
    SELECT *
    FROM countries
    WHERE currency = 'EuroCountries'
  )
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 constant defined in a WITH clause, but its definition is global. Views are used for "frequently asked queries". They can also simplify queries considerably by splitting them into smaller units. They are evaluated each time from the underlying tables. A view is created with the CREATE VIEW statement

```
CREATE VIEW viewname AS sql_query
```

where `sql_query` can be any SQL query considered this far.

2.16 SQL pitfalls

Here 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 not a valid query either. 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.

Renaming syntax

Renaming is made with the AS operator, which however has slightly different uses:

- 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. 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 e.g 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
```

Hence if v is NULL, SQL does not even be assume that it has the same value in all occurrences.

Another example, given in

> https://www.simple-talk.com/sql/t-sql-programming/ten-common-sql-programming-mistakes/

as the first one among the "Ten common SQL mistakes", 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.

More precisely, conditions have a three-valued logic, because of the presence of NULL. 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	T
T	U	”	U	T
F	T	T	F	T
F	F	”	F	F
F	U	”	F	U
U	T	U	U	T
U	F	”	F	U
U	U	”	U	U

A tuple satisfies a WHERE clause only if it returns T, not one with U. Keep in mind, in particular, that NOT U = U!

Set operations are set operations

Being a set means that duplicates don't count. This is what holds in the mathematical theory of relations (Chapter˜\ref{relations}). But SQL is usually about **multisets**, so that duplicates do count. However, 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
```

2.17 SQL in the Query Converter*

Notice: *The query converter is an experimental program that you might want to try. It is in no way a compulsory part of these lectures.*

You can find the query converter (command `qconv`) in

> https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/

which contains its source code. There is also an emerging web interface in

> http://www.grammaticalframework.org/qconv/

The Query Converter has an SQL parser and interpreter, which works much the same way as the PostgreSQL shell. [3] Thus you can give SQL commands in the qconv shell, and the database is queried and updated accordingly.

The query converter will give access to various concepts of this book, not just the SQL: relational algebra, E-R diagrams, functional dependencies, SQL. This is the main reason sometimes to use qconv as an SQL interpreter instead of PostgreSQL.

Only a part of SQL is currently recognized by qconv. The interpreter may moreover be buggy. The database is only built in memory, not stored on a disk. Thus you should store your work in an SQL source file. Such files can be read with the `i` ("import") command, for instance,

```
> i countries.sql
```

which uses the file

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/countries.sql

[3] As for , the SQL interpreter is not yet available in the web interface.

3 Entity-Relationship diagrams

*A popular device in modelling is **E-R diagrams** (Entity-Relationship diagrams). This chapter explains how different kinds of data are modelled by E-R diagrams. We will also tell how E-R diagrams can almost mechanically be derived from descriptive texts. Finally, we will explain how they are, even more mechanically, converted to relational schemes (and thereby eventually to SQL).*

A relational database consists of a set of tables, which are linked to each other by referential constraints. 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 be converted to tables. Thus it helps design a database with right dependencies. When the E-R model is ready, it can be automatically converted to relational database schemas.

This chapter gives just the bare bones of E-R models. Their correct use is a skill that has to be practised. This practice is particularly suited for work in pairs: you should discuss the model with your lab partner. You should debate, challenge and disagree. Sometimes there are many models that are equally good. But often a good-looking model is not so good if you take everything into account. Four eyes see more than two.

The course book contains valuable examples and discussions. You can find some more good examples in the old course slides. And of course, we will discuss and give examples during the lecture!

Figure 2 shows an example of an E-R diagram. We will hopefully add some other examples in later versions of these notes.

3.1 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 ore 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 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.

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 ::= Ident
```

This grammar is used in the Query Converter (Section 3.5). It is also useful in other ways:

- it defines exactly what combinations of elements are possible, so that you can avoid "syntax errors" (i.e. drawing impossible E-R diagrams)
- 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 (the Query Converter is just one example of such a program).

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.

3.2 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 kinds 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 the attribute "joining year".
- 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).

3.3 Converting E-R diagrams to database schemas

The standard conversions are shown in Figure 1. The conversions are unique for ordinary entities, attributes, and many-to-many relationships.

- An entity becomes a relation with its attributes and keys just as in E-R.
- A relationship becomes a relation that has the key attributes of all related entities, as well as its own attributes.

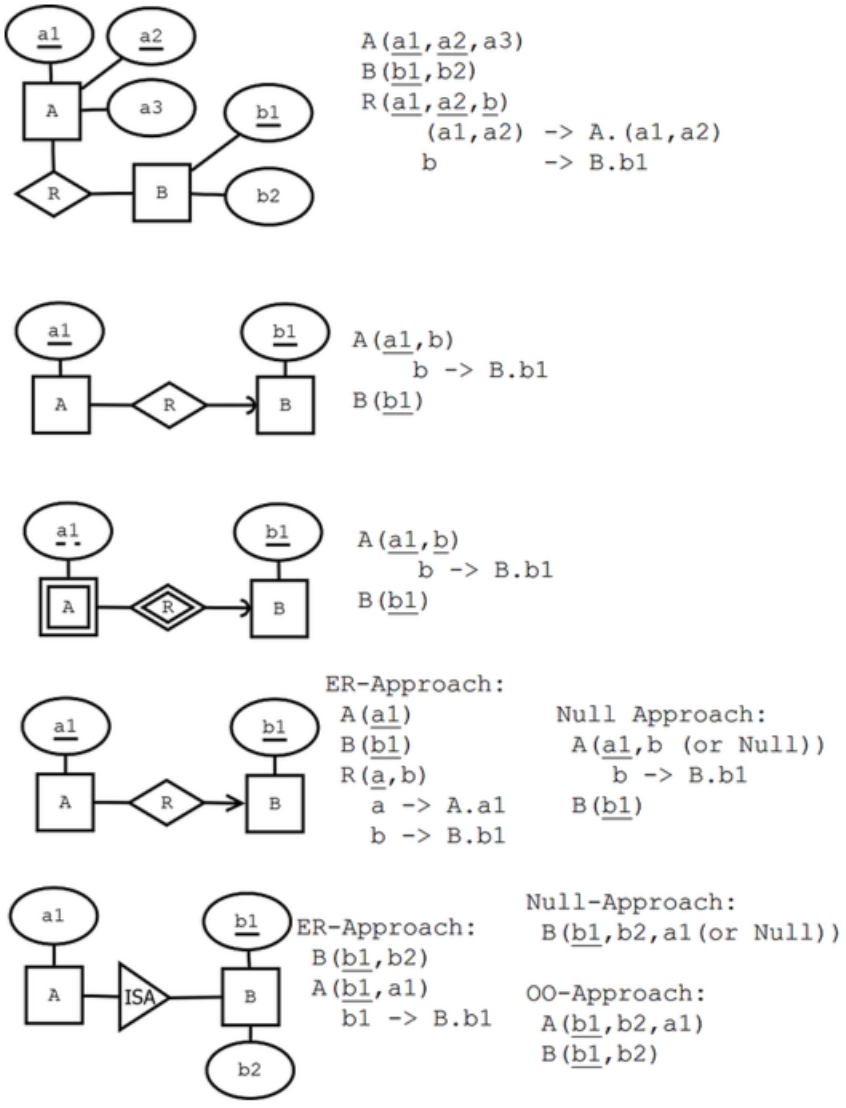


Figure 1: Translating E-R diagrams to database schemas. Picture by Jonas Almström-Duregård 2015.

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.

As the name might suggest, the E-R approach is always 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).

The course book actually uses plural nouns for entites, so that the conversion is easier. However, we have found 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. The book uses the term **entity set** for entities, which is the set of entities of the given kind.

3.4 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,

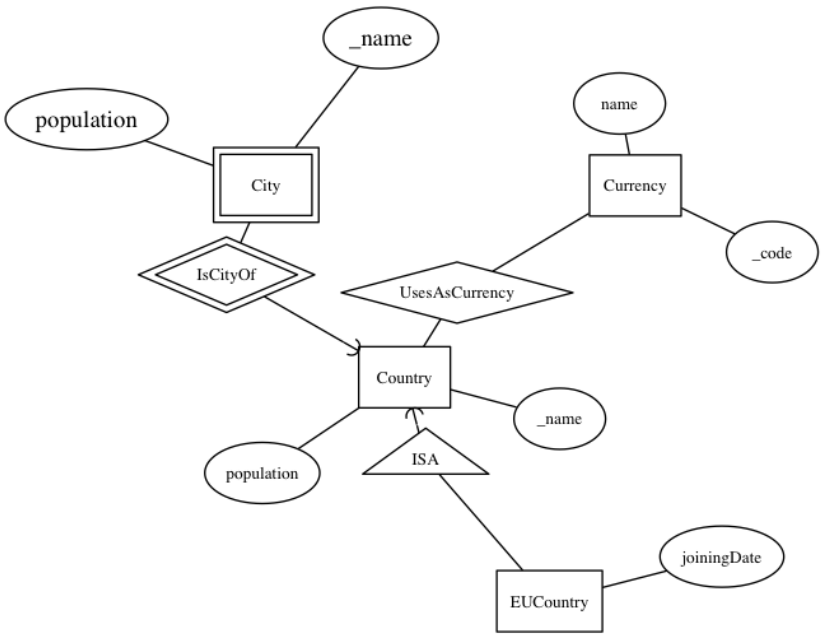


Figure 2: An E-R diagram generated from the Query Converter qconv.

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.

3.5 E-R diagrams in the Query Converter*

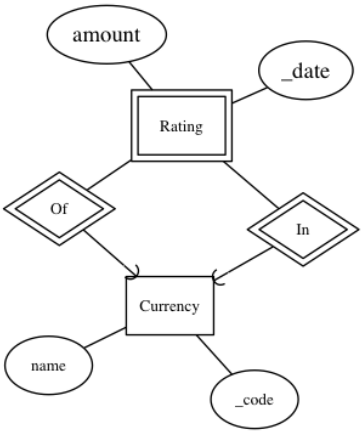
In the Query Converter (Section 2.17), you can specify an E-R model using the syntax described above. For example:

```
ENTITY Country : _name population
WEAK ENTITY City Country IsCityOf : _name population
ISA EUCountry Country : joiningDate
ENTITY Currency : _code name
RELATIONSHIP UsesAsCurrency -- Country -- Currency
```

The result is the diagram shown in Figure 2. You will also get a database schema:

```
Country(_name,population)

City(_name,population,_name)
  name => Country.name
```




```
Currencies(_code,name)
Ratings(_currencyCodeOf,_currencyCodeIn,_date,amount)
  currencyCodeOf => Currencies.code
  currencyCodeIn => Currencies.code
```

Figure 3: An E-R diagram for currency ratings, with two supporting relations, and the resulting schema.

```
EUCountry(_name,joiningDate)
  name => Country.name

Currency(_code,name)

UsesAsCurrency(_countryName,_currencyCode)
  countryName => Country.name
  currencyCode => Currency.code
```

As an experimental feature, you will also get a text:

```
A country has a name and a population.
A city of a country has a name and a population.
An eucountry is a country that has a joining date.
A currency has a code and a name.
A country can use as currency a currency.
```

Figure 3 shows another example, where the weak entity Rating has two supporting relations with Currency. This design was the result of a discussion at a lecture in 2016. It may look unusually complicated because of the weak entity with two supporting relationships. But the generated schema is entirely natural, and we leave it as a challenge to obtain it from any other E-R design.

4 Data modelling with relations

*Until now, we have used tables intuitively via the SQL language. However, there is a theoretical model behind the tables, and that theoretical model is called the **relational model**. This chapter is about the representation of tables in the relational model. The main problem is that not all data is "naturally" relational, which means some encoding is necessary. Many things can go wrong in the encoding, and lead to redundancy, inconsistencies or even to unintended data loss. This chapter gives several examples of different kinds of data. It introduces the notion of **relational schemas**, which are in SQL implemented by `CREATE TABLE` statements. But the level here is a bit more abstract than SQL. The concepts in this chapter are expressed in a mathematical notation, which is derived from set theory.*

4.1 Relations and tables

The mathematical model of relational databases is, not surprisingly, relations. Mathematically, a relation 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. The elements t_i are the **components** of the tuple. The cartesian product of which the relation is a subset is its **signature**. The sets T_i are the **types** of the components.

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. Such relations are typically, but not necessarily, infinite sets of tuples.

In the database world, a relation is usually called a **table**. Tuples are called **rows**. 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 types 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. SQL, however, makes a distinction, marked by the `DISTINCT` and `ORDER` keywords. This means that, strictly speaking, SQL tables are **lists** of tuples. If the order does not matter but the repetitions do, the tables are **multisets**.

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. In particular, functional dependency algorithms can be formulated by referring only to the attributes. But their definitions must in the end refer to tuples. By tuples, we will from now on mean labelled tuples (records) rather than set-theoretic ordered tuples. But we will be able to ignore the types of the columns.

Definition (tuple, attribute, value). A **tuple** has the form

$$\{A_1 = v_1, \ldots, A_n = v_n\}$$

where $A_1, \ldots, A_n$ are **attributes** and $v_1, \ldots, v_n$ are their **values**.

Definition (signature, relation). The **signature** of a tuple, S, is the set of all its attributes, $\{A_1, \ldots, A_n\}$. A **relation** R of signature S is a set of tuples with signature S. But we will sometimes also say "relation" when we mean the signature itself.

Definition (projection). If t is a tuple of a relation with signature S, the **projection** $t.A_i$ computes to the value v_i.

Definition (simultaneous projection). If X is a set of attributes $\{B_1, \ldots, B_m\} \subseteq S$ and t is a tuple of a relation with signature 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 signature S. Then A is **functionally dependent** on X in the relation R, written $X \to A$, if

- for all tuples t,u in R, if $t.X = u.X$ then $t.A = u.A$.

If Y is a set of attributes, we write $X \to Y$ to mean that $X \to A$ for every A in Y.

4.3 Definitions of closures, keys, and superkeys

As a starting point of modelling the constraints for a relation, the relation is charaterized by its signature S, and its functional dependencies FDs.
Assume thus a signature (i.e. set of attributes) S and a set FD of functional dependencies.
Definition. An attribute A **follows** from (or is determined by) a set of attributes Y, if there is an FD $X \to A$ such that $X \subseteq Y$.

Normally, the person who builds a database model gives a set of FDs which 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 an attribute A follows from a set of attributes Y it means that such an FD can be inferred and not necessarily explicitly given.
Definition (closure of a set of attributes under FDs). The **closure** of a set of attributes $X \subseteq S$ under a set FD of functional dependencies, denoted $X+$, is the set of those attributes that follow from X.

$$X+ = \{A \mid A \in S, Y \to A, Y \subseteq X\}$$

Definition (trivial functional dependencies). An FD $X \to A$ is **trivial**, if $A \in X$.
Algorithm (closure of attributes). If $X \subseteq S$, then the closure $X+$, can be computed in the following way:

1. Start with $X+ = X$
2. Set $New = \{A \mid A \in S, A \notin X+, A \text{ follows from } X+\}$
3. If $New = \emptyset$, return $X+$, else set $X+ = X + \cup New$ and go to 1

Definition (closure of a set of FDs). The closure of a set FD of functional dependencies, denoted by FD+, is defined as follows:

$$FD+ = \{X \to A \mid X \subseteq S, A \in X+, A \notin X\}$$

The last condition excludes trivial functional dependencies.
Definition (superkey, key). A set of attributes $X \subseteq S$ is a **superkey** of S, if $S \subseteq X+$.
A set of attributes $X \subseteq S$ is a **key** of S if

- X is a superkey of S
- no proper subset of X is a superkey of S

To give an example of the above concepts, let us start with a 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

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

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

The dependencies are much like implications in the logical sense. Thus they are **transitive**, which means that we can also infer the FD

```
country -> value
```

The set of attributes that can be inferred from a set of attributes X is the **closure** of X. Thus, since `value` can be inferred from `country`, if belongs to its closure. In fact,

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

noticing that $A \to A$ is always a valid FD, called **trivial functional dependency**.

Now, a possible key of a relation is a set of attributes whose closure is the whole signature. Thus `country` alone is a possible key. However, it is not the only set that determines all attributes. All of

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

do this. However, all of these sets but the first are just irrelevant extensions of the first. They are not keys but **superkeys**, i.e. supersets of keys. We conclude that `country` is the only possible key of the relation.

4.4 Modelling SQL key and uniqueness constraints

In SQL, we define one key and additionally we may defina a number of unique constraints. On the logical level, key and unique constraints are the same. So, both can be modelled 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`. [4]

With all 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 $\langle a, b, c, d, e\rangle$

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:	$R \bowtie_C S = \{\langle t, s\rangle \mid t \in R \text{ and } s \in S \text{ and } C\}$

[4] 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.

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. Actually, maybe it should be avoided because it relies on the names of attributes without making them explicit. But here is the definition if you want to see it:

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

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 (exercise!). Thus we can conclude: natural join is a special case of theta join, which itself is a special cases of the cartesian product. 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 to 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 is usually 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. 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 we need to join a relation with itself, we can answer 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 that transitive relation may be split into different relations. It should, however, be 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 may contain different ways to specify transitive closure. The reader is urged to check from online documentation how this is done in different language implementations such as PostgreSQL.

4.9 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.

4.10 Null values

Set theory does not have a clear meaning for NULL values, and relational database theory researchers have studied different types of represenations for missing values. On this basis, we do not have a 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? To approach that question, we want to have in independent and more formal definition. This is what functional dependencies are for. 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.

5.1 Relations vs. functions

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 same information (in particular, of the same argument-value pair). Redundancy can lead to **inconsistency**, if the information that should be the same in different places is actually not the same. This can happen for instance as a result of updates, 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 7.13). But it is better to design the database in a way that avoids the problem altogether.

Intuitively, we can require the following properties from a database design.

1. We can store our data in the database and retrieve it from there as we stored it.
2. Processing queries and updates should be simple and efficient.
3. Unnecessary data redundancy should be avoided.
4. It should be possible to ensure that the semantic constraints are fulfilled.

These properties are informal, and functional dependencies are an attempt to formalize them. To get started with the formalization, we need to define a number of new concepts and notations. Some of the basic concepts have already been defined in Section 4.2 and 4.3 above: functional dependency (FD), closure, key, superkey. We will start with a notion of **normal forms** based on these concepts, and continue with a more theoretical discussion of the properties of the normal forms.

5.2 Dependency-based design workflow

Functional dependency analysis is a mathematical tool for database design. It is quite different from E-R diagrams and should ideally be used independently of it. One way to do this (common in textbook examples), is the following procedure:
1. Collect *all* attributes into one and the same relation. At this point, it is enough to consider the relation as a set of attributes,

$$S = \{A_1, \ldots, A_n\}$$

2.Specify the functional dependencies and among the attributes. Informally,

- a **functional dependency** (FD) $A \to B$ means that, if you set the value of A, there is only one possible value of B. This generalizes to sets of attributes on both sides of the arrow.

3. From the functional dependencies, calculate the possible **keys** of the relation. Informally,

- a **key** is a combination X of attributes such that $X \to S$, i.e. all attributes of the relation are determined by the attributes in X.

4. From the FDs and keys together, calculate the **violations** of **normal forms**:

- the **third normal form** (3NF)
- the **Boyce-Codd normal form** (BCNF)

5. From the normal form violations, compute a **decomposition** of the relation to a set of smaller relations. These smaller relations each have their own FDs and keys. But it is always possible to reach a state with no violations by iterating the decomposition. The result is a set of tables, each with their own keys, which have no violations.
6. Decide what decomposition you want. All normal forms have their pros and cons. At this point, you may want to compare the dependency-based design with the E-R design.

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.

5.3 Examples of dependencies and normal forms

5.3.1 Functional dependencies, keys, and superkeys

Let us start with a 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

We assume that each country has a unique currency, and each currency has a unique value. This gives us two functional dependencies:

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

The dependencies are much like implications in the logical sense. Thus they are **transitive**, which means that we can also infer the FD

```
country -> value
```

The set of attributes that can be inferred from a set of attributes X is the **closure** of X. Thus, since `value` can be inferred from `country`, if belongs to its closure. In fact,

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

noticing that $A \to A$ is always a valid FD, called **trivial functional dependency**.

Now, a possible key of a relation is a set of attributes whose closure is the whole signature. Thus `country` alone is a possible key. However, it is not the only set that determines all attributes. All of

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

do this. However, all of these sets but the first are just irrelevant extensions of the first. They are not keys but **superkeys**, i.e. supersets of keys. We conclude that `country` is the only possible key of the relation.

5.3.2 BCNF

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.

5.3.3 3NF

Now, let us consider an example where BCNF is not quite so beneficial. Here is a table with cities, streets, and postal codes.

city	street	code
Gothenburg	Framnäsgatan	41264
Gothenburg	Rännvägen	41296
Gothenburg	Hörsalsvägen	41296
Stockholm	Barnhusgatan	11123

Here is the signature with functional dependencies:

```
city street code
city street -> code
code -> city
```

The keys are the composite keys `city street` and `code street`. But notice that the non-key FD `code -> city` refers back to a key. If we now perform BCNF decomposition, we obtain the schemas

```
Cities(_code, city)
FD: code -> city

Streets(_street, _code)
```

The problem with this decomposition is that we miss one FD, `city street -> code`. And in fact, the decomposition does not help remove redundancies. The original relation is fine as it is. It is already in the **third normal form** (3NF). 3NF is like BCNF, except that a non-key FD $X \to A$ is allowed if A is a part

of some key. Here, `city` is a part of a key, so it is fine. (An attribute that is a part of a key is called **prime**.)

Since the 3NF requirement is weaker than BCNF, it does not guarantee the removal of all FD redundancies. But in many cases, the result is actually the same: the country-currency-value table is an example.

5.3.4 Multivalued dependencies and the fourth normal form

Multivalued dependencies (MVD) are another kind of dependencies. An MVD $X \twoheadrightarrow Y$ says that X determines Y independently of all other attributes. (The precise definition is a bit complicated, and is given in Section 5.5.1.)

Here is an example: 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. Thus we can say that the products of a country are independent of its neighbours. This can be expressed as a multivalued dependency,

```
country ->> product
```

Why "multivalued"? Because the country can have several products, and not just one, as in a functional dependency. Another term suggested for multivalued dependency is **independency**: the products of a country are independent of its neighbours.

Because of the MVD, the above table has a redundancy: both cars and paper and Norway and Finland are mentioned repeatedly. The formal expression for this an **4NF violation**: an MVD where the LHS is not a superkey. 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.

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, the natural join from those tables can produce combinations not existing in the reality.

5.4 A bigger example

Let us collect everything about a domain into one big table:

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

We identify some functional dependencies and multivalued dependencies:

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

One possible BCNF decomposition gives 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 randomly choose some other dependencies to split on, and lead to less natural results. For instance, it can use capital rather than country as the key of the first table.

In the next section, we will show the precise algorithms that are involved, and which you should learn to execute by hand. The definitions might look more scary than they actually are. Most of the concepts are intuitively simple, but their precise definitions can require some details that one usually doesn't think about. The notion of MVD is usually the most difficult one.

In the sections that follow, we will go a bit deeper in analysing what makes the normal forms beneficial and what design questions they raise. In the last section of this chapter, we will look at the support given by the Query Converter (`qconv`) for dependency analysis and normal form decomposition.

5.5 Mathematical definitions for dependencies and normal forms

5.5.1 Relationas, tuples, and depencencies

Before introducing new concepts, we will repeat some of the definitions about tuples from Chapter 4. We will most of the time speak of relations just as their sets of attributes. Also the dependency algorithms refer only to the attributes. But the definitions in the end do refer to tuples. By tuples, we will now mean labelled tuples (records) rather than set-theoretic ordered tuples as in Chapter 4.

Definition (tuple, attribute, value). A **tuple** has the form

$$\{A_1 = v_1, \ldots, A_n = v_n\}$$

where $A_1, \ldots, A_n$ are **attributes** and $v_1, \ldots, v_n$ are their **values**.

Definition (signature, relation). The **signature** of a tuple, S, is the set of all its attributes, $\{A_1, \ldots, A_n\}$. A **relation** R of signature S is a set of tuples with signature S. But we will sometimes also say "relation" when we mean the signature itself.

Definition (projection). If t is a tuple of a relation with signature S, the **projection** $t.A_i$ computes to the value v_i.

Definition (simultaneous projection). If X is a set of attributes $\{B_1, \ldots, B_m\} \subseteq S$ and t is a tuple of a relation with signature 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 signature S. Then A is **functionally dependent** on X in the relation R, written $X \to A$, if

- for all tuples t,u in R, if $t.X = u.X$ then $t.A = u.A$.

If Y is a set of attributes, we write $X \to Y$ to mean that $X \to A$ for every A in Y.

Definition (multivalued dependency, MVD). Let X, Y, Z be disjoint subsets of a signature S such that $S = X \cup Y \cup Z$. Then Y has a **multivalued dependency** on X in R, written $X \twoheadrightarrow Y$, if

- for all tuples t,u in R, if $t.X = u.X$ then there is a tuple v in R such that
  - $v.X = t.X$
  - $v.Y = t.Y$
  - $v.Z = u.Z$

An alternative notation is $X \twoheadrightarrow Y \mid Z$, emphasizing that Y is **independent** of Z.

To see the power of these definitions, we can now easily prove a slightly surprising result saying that every FD is an MVD:

Theorem. If $X \to Y$ then $X \twoheadrightarrow Y$

Proof. Assume that t,u are tuples in R such that $t.X = u.X$. We select $v = u$. This is a good choice, because

1 $u.X = t.X$ by assumption

2 $u.Y = t.Y$ by the functional dependency $X \to Y$
3 $u.Z = u.Z$ by reflexivity of identity.

Note. MVDs are symmetric on their right hand side: if $X \twoheadrightarrow Y$, which means $X \twoheadrightarrow Y \mid Z$ where $Z = S - X - Y$, then also $X \twoheadrightarrow Z$. Thus in the example in Section 5.3.4, we could have written, equivalently,

```
country ->> exportedTo
```

5.5.2 Closures, keys, and superkeys

As a starting point of dependency analysis, a relation is charaterized by its signature S, its functional dependencies FD, and its multivalued dependencies MVD. We start with things where we don't need MVD.

Assume thus a signature (i.e. set of attributes) S and a set FD of functional dependencies.

Definition. An attribute A **follows** from a set of attributes Y, if there is an FD $X \to A$ such that $X \subseteq Y$.

Definition (closure of a set of attributes under FDs). The **closure** of a set of attributes $X \subseteq S$ under a set FD of functional dependencies, denoted $X+$, is the set of those attributes that follow from X.

Algorithm (closure of attributes). If $X \subseteq S$, then the closure $X+$, can be computed in the following way:

1. Start with $X+ = X$
2. Set $New = \{A \mid A \in S, A \notin X+, A \text{ follows from } X+\}$
3. If $New = \emptyset$, return $X+$, else set $X+ = X + \cup New$ and go to 1

Definition (closure of a set of FDs). The closure of a set FD of functional dependencies, denoted by FD+, is defined as follows:

$$FD+ = \{X \to A \mid X \subseteq S, A \in X+, A \notin X\}$$

The last condition excludes trivial functional dependencies.

Definition (trivial functional dependencies). An FD $X \to A$ is **trivial**, if $A \in X$.

Definition (superkey, key). A set of attributes $X \subseteq S$ is a **superkey** of S, if $S \subseteq X+$.

A set of attributes $X \subseteq S$ is a **key** of S if

- X is a superkey of S
- no proper subset of X is a superkey of S

5.5.3 Decomposition algorithms

Definition (Boyce-Codd Normal Form, BCNF violation). A functional dependency $X \to A$ **violates BCNF** if

- X is not a superkey
- the dependency is not trivial

A relation **is in Boyce-Codd Normal Form** (BCNF) if it has no BCNF violations.

Note. Any trivial dependency $A \to A$ always holds even if A is not a superkey.

Definition (prime). An attribute A is prime if it belongs to some key.

Definition (Third Normal Form, 3NF violation). A functional dependency $X \to A$ **violates 3NF** if

- X is not a superkey
- the dependency is not trivial
- A is not prime

Note. 3NF is a weaker normal form than BCNF: Any violation $X \to A$ of 3NF is also a violation of BCNF, because it says that X is not a superkey. Hence, any relation that is in BCNF is also in 3NF.

Definition (trivial multivalued dependency). A multivalued dependency $X \twoheadrightarrow A$ is trivial if $Y \subseteq X$ or $X \cup Y = S$.

Definition (Fourth Normal Form, 4NF violation). A multivalued dependency $X \twoheadrightarrow A$ **violates 4NF** if

- X is not a superkey
- the MVD is not trivial.

Note. 4NF is a stronger normal form 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

Algorithm (BCNF decomposition). Consider a relation R with signature S and a set F of functional dependencies. R can be brought to BCNF by the following steps:

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 signature $X \cup \{A\}$
   - R_2 with signature $S - \{A\}$
3. Apply the above steps to R_1 and R_2 with functional dependencies projected to the attributes contained in each of them.

One can combine several violations with the same left-hand-side X to produce fewer tables. Then the violation $X \to Y$ decomposes R to $R_1(X, Y)$ and $R_2(S - Y)$.

Note. Step 3 of the BCNF decomposition algorithm involves the **projection of functional dependencies**. This can in general be a complex procedure. However, for most cases handled during this course, it is enough just to filter out those dependencies that do not appear in the new relations.

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*

Note. This algorithm has the same structure as the BCNF decomposition. For 3NF decomposition, a very different algorithm is used, seemingly with no iteration. But an iteration can be needed to compute the *minimal basis* of the FD set.

Concept (minimal basis of a set of functional dependencies; not a rigorous definition). A **minimal basis** of a set F of functional dependencies is a set F- that implies all dependencies in F. It is obtained by first weakening the left hand sides and then dropping out dependencies that follow by transitivity. Weakening an LHS in $X \to A$ means finding a minimal subset of X such that A can still be derived from F-.

Algorithm (3NF decomposition). Consider a relation R with a set F of functional dependencies.

1. If R has no 3NF violations, return R.
2. If R has 3NF violations,
   - compute a minimal basis of F- of F
   - group F- by the left hand side, i.e. so that all depenencies $X \to A$ are grouped together
   - for each of the groups, return the schema $XA_1 \dots A_n$ with the common LHS and all the RHSs
   - if one of the schemas contains a key of R, these groups are enough; otherwise, add a schema containing just some key

Example (minimal basis). 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 basis: the second FD can be dropped because it follows from the first and the third ones by transitivity. Hence we have a minimal basis

```
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.

5.6 More definitions for functional dependencies*

Let $f = X \to Y$ be a functional dependency. We call X the **left hand side** of f and Y the **right hand side** of f. We assume that all left hand sides are non-empty.

We say that a functional dependency $X \to Y$ is **embedded** in a relation schema R if $X \cup Y \subseteq R$. For a set of dependencies F, we denote the set of left hand sides of F by LHS(F).

Usually, functional dependencies are given as semantic constraints, and only those database states where all dependencies hold in all respective relations are considered legal; the constraints can be used in the design process as well as later when the database is being used. As an input to database design, the dependencies are normally given before the relation schemas are being decided upon. With this purpose in mind, we may say that a set F of functional dependencies is *given over* a universe U.

Some dependencies may get embedded in some relation schema in the database schema while some may not. If all dependencies in a given set of dependencies F are embedded in some relation schema of a database schema $\mathbf{R}$, we say that F *is embedded in* $\mathbf{R}$.

The idea of a relation satisfying a functional dependency is fairly simple. It is not quite as clear what it should mean for a database to satisfy a functional dependency.

A *containing instance* i of a database $\mathbf{r}$ over a database scheme $\mathbf{R}$ is such a relation over the underlying universe U that each relation $r \in \mathbf{r}$, where r is a relation over $R \in \mathbf{R}$, is a subset of $\pi_R(i)$.

Let U be a universe, $\mathbf{R}$ a database schema over U, $R \in \mathbf{R}$, and let $X \subseteq U$, $Y \subseteq U$. We say that a database $\mathbf{r}$ over $\mathbf{R}$ satisfies the functional dependency $X \to Y$ if there is a containing instance i for $\mathbf{r}$ such that i satisfies $X \to Y$. If there is such a containing instance i that i satisfies all the given functional dependencies, then i is called a *weak instance* for the database $\mathbf{r}$. Further, if i is a subset of all other weak instances for $\mathbf{r}$, then i is a *representative instance* for $\mathbf{r}$.

Whether a database satisfies a set of functional dependencies can be checked with a relatively simple method called the *chase*. The chase proceeds by manipulating a *tableau*, which is like a relation except that in addition to values from domains of attributes, it may also contain *variables* as elements.

We apply the notations and terminology for relations to tableau without giving them explicitly. This should not raise any ambiguities. We construct the initial tableau as input for the chase using the following definition.

Definition (Database tableau)
Let $\mathbf{R}$ be a database schema over a universe U and let $\mathbf{r}$ be a database over $\mathbf{R}$. Let n be the total number of tuples in the relations of $\mathbf{r}$, and assume that the tuples are numbered from 1 to n. Assume also that the attributes in U are numbered from 1 to k, and we will denote the attribute with number j by A_j. Let i be the number of a tuple t in a relation r over a relation schema R. We define $t_i[A_j]$, $1 \leq j \leq k$, to be $t[A_j]$, if $A_j \in R$, and $v_{(i,j)}$, a nondistinguished

variable, otherwise. We assume that $v_{(i,j)} \neq v_{(i',j')}$, if $i \neq i'$ or $j \neq j'$. In this way, we get a tuple t_i over U for each tuple in the relations of $\mathbf{r}$. The *initial tableau for* $\mathbf{r}$ is a tableau over U consisting of the tuples t_i, $1 \leq i \leq n$.

The chase is defined as follows.

Algorithm (Chase)

- Input: An initial tableau for a database $\mathbf{r}$ and a set of functional dependencies F
- Outout: A modified tableau, denoted by chase$(\mathbf{r}, F)$

1. Apply the following rule as long as possible: If there is a functional dependency $X \rightarrow Y$ in F and tuples t and t' in the tableau such that $t[X] = t'[X]$ and $t'[A]$ is a nondistinguished variable, replace $t'[A]$ by $t[A]$ everywhere in the tableau.

Now, a database $\mathbf{r}$ satisfies F if and only if chase$(\mathbf{r}, F)$ satisfies F. If chase$(\mathbf{r}, F)$ satisfies F, then a representative instance (possibly containing null values) for $\mathbf{r}$ can be obtained from chase$(\mathbf{r}, F)$ simply by replacing all nondistinguished variables by null values.

Consider an application with professors, students and books. Each professor, student, and book has a unique name (attributes Professor, Student, and Book).

We use a relation over relation schema Instructs_on(Professor, Book) to record which professor gives instruction on which book. A relation over relation schema Studies(Student, Book) tells which student studies which book and the relation schema Supervises(Professor, Student) is for a relation that contains information on which professor supervises which student.

We now assume that each student only takes instruction from one professor (Student $\rightarrow$ Professor) and only one professor gives supervision on any book (Book $\rightarrow$ Professor). We also assume that these are the only given functional dependencies.

In our example, we will use j students with names $s_0, ..., s_j$, two professors with names p_0 and p_j, and j books with names $b_0, ..., b_j$, where j is a positive integer. In our database we have three relations. A relation r_1 over Instructs_on is empty, and the relation r_2 over Studies and r_3 over Supervises are shown below.

$r_2 =$

Student	Book
s_0	b_0
s_1	b_0
s_1	b_1
s_2	b_1
s_2	b_2
.	.
.	.
.	.
s_n	b_j

$r_3 =$

Professor	Student
p_0	s_0

The initial tableau is given below.

Professor	Student	Book
p_0	s_0	$v_{(1,3)}$
$v_{(2,1)}$	s_0	b_0
$v_{(3,1)}$	s_1	b_0
$v_{(4,1)}$	s_1	b_1
$v_{(5,1)}$	s_2	b_1
$v_{(6,1)}$	s_2	b_2
.	.	.
.	.	.
.	.	.
$v_{(j+1,1)}$	s_j	b_j

The chasing could now, for instance, proceed as follows. Since we have Student $\rightarrow$ Professor, we can replace $v_{(2,1)}$ by p_0. Using the functional dependency Book $\rightarrow$ Professor, we can replace $v_{(3,1)}$ by p_0. By repeating applications of the dependencies Student $\rightarrow$ Professor and Book $\rightarrow$ Professor, each $v_{(i,1)}$, $2 \leq i \leq j+1$ will be replaced by p_0, and, thus, the result of the chase will be as shown below.

Professor	Student	Book
p_0	s_0	$v_{(1,3)}$
p_0	s_0	b_0
p_0	s_1	b_0
p_0	s_1	b_1
p_0	s_2	b_1
p_0	s_2	b_2
.	.	.
.	.	.
.	.	.
p_0	s_j	b_j

It is easy to see that the result satisfies all the given functional dependencies, and, thus, if we take $v_{(1,3)}$ to be a null value, it is also the representative instance for our database. As a consequence, our database also satisfies the given functional dependencies.

Assume now that the relation r_2 over Supervises would also contain a tuple (p_j, s_j). Now, the tableau produced by the chase would have all the tuples as in the earlier case and a tuple $(p_j, s_j, v_{(j+2,3)})$. As this tableau does not satisfy the functional dependency Student $\rightarrow$ Professor, it can be seen that this modified database does not satisfy the given functional dependencies.

5.7 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 is and cannot be a key - e.g. Ranta would be a duplicate key value. \footnote{This is example is better known as the (City, Street, Zip) example, where there is a unique Zip for each (City,Street) pair and a unique City for each Zip. } 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 noq 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.

5.8 More definitions and algorithms for database design*

Definition (dependency preservation). We call the database D **dependency preserving** with respect to a set of functional dependencies FD, if D embeds a cover of FD, that is, 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 over U. We say that D has the *lossless join property* if every relation r_U over U decomposes losslessly onto $R\_1, ..., R\_n$. If D has the lossless join property, we also 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 D (i.e. set of relations) is lossless if and only if it contains a relation R such that closure of R contains all attributes in D.

Now, consider redundancy in the context of a single relation. We have seen above that potential problems with redundancy appeared when there were other functional dependencies than those describing a key. An example remedy was BCNF decomposition, eliminating functional dependencies where the LHS is not a superkey. It can be shown that when we consider a single update (inserting or deleting a tuple) in a single-relation database, then we *cannot have an arbitrary number of required "knock-on" updates if and only if that relation is in BCNF*. This makes BCNF a strong required property.

The BCNF decomposition algorithm always produces databases in BCNF. There is a problem, though, that this may violate dependency preservation. Let us review, again, the City-Stree-Zip example, with dependencies City,Street -> Zip, Zip->City, and a relation (City,Street,Zip). If we decompose with Zip-City, then the database is no longer dependency preserving. If we do not decompose, then the database is not in BCNF. In such a case, we can achieve a weaker property, the third normal form (3NF).

Since the 3NF requirement is weaker than BCNF, it does not guarantee the removal of all FD redundancies. But in many cases, the result is actually the same: the country-currency-value table is an example. The bad news with 3NF is that potential knock-on updates happen on the key values, and as keys may be used in foreign keys and indexes on data, this may be computationally expensive.

5.9 Acyclicity*

We found BCNF to be a strong property in tackling the knock-on effects of updates. However, in the case of multiple relations BCNF is not enough alone to guarantee a constant amount of knock-on updates. Consider the following example.

professor	**course**
p0	s0

professor	**student**

student	**course**
s0	c0
s1	c0
s1	c1
s2	c1
s2	c2
...	
sn	cn

Let us assume that the functional dependencies are student -> professor, course -> professor. and we add the tuple (pn,cn) to (professor,course) relation. We can check the functional dependency using the Chase. The start-up configuration is

student	**course**	**professor**
s0	_	p0
s0	c0	_
s1	c0	_
s1	c1	_
s2	c1	_
s2	c2	_
...		
sn	cn	_
sn	_	pn

Using the Chase we find that each empty value for professor has to be p0, but then we have a row with values (sn,p0) and (sn,pn), which violates the functional dependency student -> professor.

What a disappointment! Boyce-Codd normal form, dependency preservation and lossless join are not enough to guarantee a trouble-free database design. But what is the formal property associated with this? This is obviously related to different ways in which the same attributes are connected.

A *hypergraph* consist of a set of *nodes* N and a set of *hyperedges* E. The hyperedges are nonempty subsets of N. This means that we can identify the attributes of a universe with nodes of a hypergraph and the relation schemas with hyperedges of a hypergraph.

A *suphypergraph* of a hypergraph *(N,E)* is a pair *(N,E')*, where each hyperedge in *E'* is a subset of *E*.

There are different definitions of acyclicity of hypergraphs, but we are only interested in so called γ-acyclicity, which we call simply acyclicity.

A γ-*cycle* in a hypergraph is a sequence $R_1, A_1, ..., R_k, A_k, R_{k+1}$ of alternating edges and nodes such that

(i) $A_1, ..., A_k$ are distinct nodes,

(ii) $R_1, ..., R_k$ are distinct edges, and $R_1 = R_{k+1}$,

(iii) $k \geq 3$,

(iv) for each A_i, $1 \leq i \leq k$, we have $A_i \in R_i$, $A_i \in R_{i+1}$, and,

(v) for each A_i, $1 \leq i < k$, we have $A_i \notin R_j$, if $j \neq i, j \neq i+1$.

Intuitively, one can "walk" a round in the numbered order from the first relation schema by connecting nodes. Apart from A_k the connecting nodes are not supposed to appear in other relation schemas apart from the two they are specifically connecting.

A hypergraph (database) is γ-acyclic, if it contains no γ-cycles, otherwise it is γ-cyclic.

As an exercise, the reader is urged to check that (professor, student), (student,course),(course, professor) is γ-cyclic, whereas (professor,student,course), (course,book), (professor,course) is not.

Lossless dependency-preserving γ-acyclic BCNF databases have very strong properties. First of all, there will only be a constant amount of knock-on updates for any insert or delete operation. Query processing is also known to be simple and efficient. Further, it is known that for any pair of attributes in such an acyclic database, there is a unique minimal connection (simple semantics), which can be used e.g. in natural language query processing. When given a set of attributes X, we can join the necessary relations and project them onto X with simple relational algebra operations.

5.10 Optimal database design*

However, when and how are these nice properties achievable? The theory is complicated, but we will just some motivating examples and the results. The problematic sets of functional depdencies have two characteristic sets of problems. The City, Street, Zip problem was caused because one functional dependency determined a proper subset of another functional dependency. This is called splitting. Another problematic situation is where the right hand sides of functional dependencies have non-empty intersections, and, finally, the third problematic situation may be where the left hand sides have non-empty intersections.

To cut a relatively long story very short, we just conclude that in practice when possible, that is, when the problematic situations do not exist or can be

solved, the following algorithm will compute a database schema that is lossless, dependency preserving, acyclic, and in BNFC.

For shortness, we use the following notations in the algorithm. If $X \to Y$ but there is no $X' \subset Y$ such that $X' \to Y$, then we say that $X \to Y$ is *full* and we write $X \mapsto Y$. Further, we write $X_f^+ = \{A \mid X \mapsto A\}$, $X^{\to} = X^+ \setminus X$ and $X^{\mapsto} = X_f^+ \setminus X$.

Algorithm (A variant of Lien's decomposition)

- Input: A universe U, a set of functional dependencies F over U, and a p-ordering $X_1, X_2, ..., X_n$ of LHS(F)
- Output: A lossless database schema over U

1. Order LHS(F) so that for any $X_i, X_j \in$ LHS(F) it holds:
   (a) If $X_i \subset X_j$, then $i \leq j$.
   (b) If $X_j \to X_i$ and $X_i \not\to X_j$, then $i < j$.
2. $\mathbf{C} \leftarrow \{U\}$.
3. For each i from 1 to n do

   for each $Y \in \mathbf{C}$ such that $X_i \subset Y$ and X_i splits Y, assign
   $\mathbf{C} \leftarrow (\mathbf{C} \setminus \{Y\}) \cup \{X_i^+ \cap Y\} \cup \{Y \setminus X_i^{\to}\}$.
4. Output $\mathbf{C}$.

Example(Lien's Algorithm) Let $U = ABCDEGH$, and $F = \{ABC \to G, BC \to D, AE \to H\}$. A possible ordering of the left hand sides is AE, BC, ABC. With this ordering, Lien's decomposition produces the database schema $\{ABCE, ABCG, BCD, AEH\}$.

These properties are complicated but the good news is that they are (or will be) implemented in the query converter FD design studio. We urge the reader to try this out.

The dependency theory contains other types of dependencies. We will discuss shortly the inclusion dependencies and the multivalued dependencies.

5.11 Inclusion dependencies*

The inclusion dependencies model the referential constraints. The normal notatation for inclusion dependencies is $R(X) \subset S(Y)$ which which means nearly the same as adding each $A_i -> S.B_i$ in the definition of R, where $X = A_i \ldots A_k$ and $Y = B_i \ldots B_k$.

The difference in notation is that the inclusion dependency can be written outside of a relation definition, and it can pack up several attributes into a single statement. Further, there is no requirement that X is a key of X or Y is a key for S.

Thinking about database design starting from a set of attributes, initially there are no inclusion dependendenies, whereas the E-R design does produce

them. However, decompositions produce them. Whenever we decompose a relation schema R with an FD $X \rightarrow Y$ where $X \subset R$, $Y \subset R$, producing relation schemas $S = X \cup Y$ and $R' = R \setminus Y$, then obviously X is a superkey of S and the idea of the decomposition is to recover the decomposed data by joining on X, and, consequently, an inclusion depdendency $S.X' \subseteq R'.X$ would be in order.

If there is no further FD to decompose R', then X is a key and the inclusion dependecy can be used to create referential constraints. If there are further FDs to decompose R', then the inclusion depdencies must be adapted accordingly. The details are left as an exercise to the interested reader. [5]

5.12 Combining practice and theory in database design

Until now, we have introduced a basic method to produce database designs from ER diagrams and on the other hand we have characterized good designs and we have given design algorithms based on the use of dependencies.

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.

Obviously, we need to consider how we can combine ER design and dependencies. First of all, some FDs can be directly derived from an ER diagram. In fact, we can use ER diagrams to produce all FDs, if we allow separate diagrams or parts of them to give additional information on dependencies, and the attributes naming is considered global. Thereby, we can separately construct an entity with City, Street, and Zip, and (City,Street) as key, and another entity with the attributes Zip and City, with Zip as the key. Another alternative is just to input additional FDs in a separate text file or similar.

A practical option, not based on the specific design theory above, is to produce the relations from the ER diagram and then further decompose them using the FDs. This solution is practical in the sense that designs produced from ER diagrams are most likely to avoid nontrivial MVDs and therefore only FD decomposition or similar is needed for the parts of the design where there are additional FDs emerging from the additional ER diagram fragments.

You need to decide what decomposition you want. All normal forms have their pros and cons. At this point, you may want to compare the dependency-based design with the E-R design. This also depends on the application. If query performance is crucial and one wants to avoid joins, and updates seldom

[5] Should we extend the details or give a reference to the book by Mannila and Räihä?

happen or are otherwise known to be feasible on pre-joined data, then you may decompose less or store joins directly.

5.13 Relation analysis in the Query Converter*

The `qconv` command `f` reads a relation from a file and prints out relation info:

- the closure of functional dependencies
- superkeys
- keys
- normal form violations

The command `n` reads a relation from the same file format and prints out decompositions in 3NF, BCNF, and 4NF.

The format of these files is as in the following example:

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

The Haskell code in

> https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/Fundep.hs

is a direct rendering of the mathematical definitions. There is a lot of room for optimizations, but as long as the number of attributes is within the usual limits of textbook exercises, the naive algorithms work perfectly well.

5.14 Further reading on normal forms and functional dependencies*

The following is a practical guide going through all the normal forms from 1NF to 5NF:

> William Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory", Communications of the ACM 26(2), Feb. 1983, 120-125. http://www.bkent.net/Doc/simple5.htm

6 Relational algebra and query compilation

Relational algebra is a mathematical query language. It is much simpler than SQL, as it has only a few operations, each denoted by Greek letters or mathematical symbols. Being so simple, relational algebra is more difficult to use for complex queries than SQL. 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 4.

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 9, but excluding subqueries*

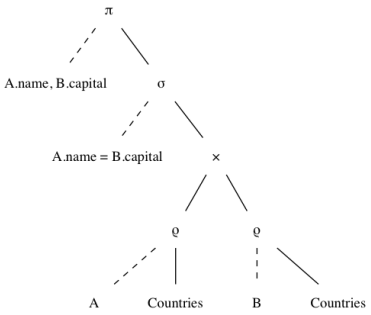
Figure 4: A grammar of relational algebra. Operator names and other explanations in **boldface**. Corresponding SQL keywords in `CAPITAL TYPEWRITER`.

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 4 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'
```

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

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'
```

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

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

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

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

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

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 2.16).

6.4.2 Grouping and aggregation

As we saw in Section 2.11, 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 is do this 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 Relational algebra in the Query Converter*

As shown in Section 2.17, qconv works as an SQL interpreter. The interpretation is performed via translation to relational algebra close to the textbook style. The notation is actually LaTeX code, and the code shown in these notes is generated with qconv (possibly with some post-editing).

The SQL interpreter of qconv shows relational algebra expressions in addition to the results. But one can also convert expressions without interpreting them, by the `a` command:

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

The source files are available in

https://github.com/GrammaticalFramework/gf-contrib/blob/master/query-converter/

The files of interest are:

- MinSQL.bnf, the grammar of SQL, from which the lexer, parser, and printer are generated,
- RelAlgebra.bnf, the grammar of relational algebra, from which the lexer, parser, and printer are generated,
- SQLCompiler.hs, the translator from SQL to relational algebra,
- Algebra.hs, the conversion of logical to algorithmic ("physical") query plans,
- Relation.hs, the code for executing the physical query plans,
- OptimizeAlgebra.hs, some optimizations of relational algebra.

6.7 Indexes

Indexes are another example of query optimization. An index is an efficient lookup table, making it fast to fetch information. A DBMS automatically creates an index for the primary key of each table. One can manually create and drop keys for other attributes 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 $k \ll n$ is the number of tuples per attribute)
  - 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:

	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

7 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.

Our main objective is to embed SQL in Java programs. The supplementary materials of the book are intended to cover the same examples for more languages, both object-oriented and not object-oriented. We will also cover some pitfalls in embedding.

First of all, 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.

Secondly, there is a mismatch between object-oriented and relational higher level data types, that is, objects and tables. Object-oriented model uses pointers or references from one object to another for navigation. Relational databases are based on values and queries for combining data. This gives flexibility and even though there are databases to store objects directly as they are, the flexible declarative query facilities have maintained the success of relational databases.

Thirdly, there are security issues. For instance **SQL injection** is a security hole 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.

7.1 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 5 is the smallest complete program we could figure out that does 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 the course material (Assignment 5) - a template from which Figure 5 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.

7.2 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.

```
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 5: 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 `*`.

```
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 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.

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.

7.3 Managing the primitive datatypes*

One needs to ensure that the host language datatypes can hold what comes from the database and that the host program only trys 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 an example of such wrapping. 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 like this means you are being systematic about your design choices.

First, the base class for all data types:

```
abstract class SqlDataType
{  /* status codes used instead of exceptions: */
   public final static int attrValueOk = 0;
   public final static int attrValueMissing = 1;
   public final static int attrValueFormatNotOk = 2;
   public final static int attrValueTooSmall = 3;
   public final static int attrValueTooLarge = 4;
   public final static int attrValueNotAllowed = 5;
   private boolean isPrime; // means "is a part of primary key"
   public SqlDataType() { isPrime = false; }
   public void setPrime (boolean isPrime) { this.isPrime = isPrime; }
   public boolean isPrime() { return isPrime; }
   abstract void setString (String input);
   public String toString() { return ""; }
   abstract boolean stringOk (String input); // for reading in
   static boolean stringIsOk (String input) { return true; }
   abstract int setAndCheck (String input, boolean notNull, String attributeName);
}
```

Then, the base class for all string types. This is really long, unfortunately.

```
package fi.uta.cs.sqldatatypes;

import fi.uta.cs.sqldatamodel.InvalidValueException;
import fi.uta.cs.sqldatamodel.NullNotAllowedException;
```

```
import fi.uta.cs.sqldatamodel.ValueTooLongException;

/**
 * Concrete implementation of string-based SQL data types.
 *
 * Value data type is java.lang.String.
 * JDBC data type is java.lang.String.
 */
public class SqlString extends SqlDataType {
private String value;
private int maxLength;

public SqlString( int maxLength ) {
super();
value = null;
this.maxLength = maxLength;
}

 //  If -1 is given, the length is not limited.
public SqlString( int maxLength, String value )
{
this( maxLength );
try {
fromString( value );
} catch( InvalidValueException e ) {
// Ignored
}
}

// ============================================================================
// Bean property methods
// ============================================================================

public String getValue() {
// String is immutable
return value;
}

public void setValue( String value ) throws NullNotAllowedException, ValueTooLongException {
if( value==null && !isNullAllowed() )
throw new NullNotAllowedException();
if( value!=null && maxLength>=0 ) {
if( value.length()>maxLength ) throw new ValueTooLongException();
}
setValueUnchecked( value );
}

public void setValueUnchecked( String value ) {
if( value == null ) {
this.value = null;
```

```
} else {
// String is immutable
this.value = value;
}
}

public int getMaxLength() {
if( maxLength<0 ) return Integer.MAX_VALUE;
return maxLength;
}

// ========================================================================
// JDBC property methods
// ========================================================================

public String jdbcGetValue() {
return getValue();
}

public void jdbcSetValue( String value ) throws NullNotAllowedException, ValueTooLongException {
setValue( value );
}

// ========================================================================
// SqlDataType methods
// ========================================================================

public String toString() {
if( value == null ) return "";
return value;
}

public void fromString( String str ) throws NullNotAllowedException, ValueTooLongException {
String newValue = null;
if( str!=null ) {
newValue = str;
}
setValue( newValue );
}

public boolean isValid() {
if( (value==null) && (!isNullAllowed()) ) return false;
if( value!=null && maxLength>=0 ) {
if( value.length()>maxLength ) return false;
}
return true;
}

public boolean equals(Object obj) {
if( !(obj instanceof SqlString) ) return false;
```

```
SqlString strObj = (SqlString)obj;

if( value==null || strObj.value==null ) {
if( value==null && strObj.value==null ) return true;
return false;
}

return value.equals( strObj.value );
}

public Object clone() throws CloneNotSupportedException {
return super.clone();
}

public String getLongestString() {
int stringLength = getMaxLength();
if( stringLength == Integer.MAX_VALUE ) {
// Use 64K for unlimited strings
stringLength = 65536;
}
StringBuffer sb = new StringBuffer(stringLength);
        for(int i = 0; i < stringLength; i++) {
         sb.append("M");
        }
        return sb.toString();
}
}
```

After this, VARCHAR only needs the relevant constructors.

```
public class SqlVarchar extends SqlString {
public SqlVarchar( int maxLength ) {
super( maxLength );
}
public SqlVarchar( int maxLength, String value ) {
super( maxLength, value );
}
}
```

7.4 Wrapping relations and views in classes*

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 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. These classes are actually created by a tool, and they are arranged accordingly: 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.

```
package fi.accounts;  // only a part of the code is given here
import java.sql.*;    // the supplementary materials have the rest
import java.util.Iterator;
import fi.uta.cs.sqldatamodel.*;
import fi.uta.cs.sqldatatypes.*;

//**
 * Generated database access class for table Accounts.
 *
 */
public class DbAccounts extends SqlAssignableObject implements Cloneable {
    private SqlVarchar numberData;
    private SqlVarchar numberKC;
    private SqlVarchar holderData;
    private SqlVarchar typeData;
    private SqlInteger balanceData;
// most methods cut out - insert given here:
    /**
     * Inserts this object to the database table Accounts.
     * @param con Open and active connection to the database.
     * @throws SQLException if the JDBC operation fails.
     * @throws ObjectNotValidException if the attributes are invalid.
     */
    public void insert(Connection con) throws SQLException, ObjectNotValidException {
        if( !numberData.isValid() ) throw new ObjectNotValidException("number");
        if( !holderData.isValid() ) throw new ObjectNotValidException("holder");
        if( !typeData.isValid() ) throw new ObjectNotValidException("type");
        if( !balanceData.isValid() ) throw new ObjectNotValidException("balance");
        String prepareString = "insert into Accounts (number, holder, type, balance) values (?, ?, ?,
        PreparedStatement ps = con.prepareStatement(prepareString);
        ps.setObject(1, numberData.jdbcGetValue());
        ps.setObject(2, holderData.jdbcGetValue());
        ps.setObject(3, typeData.jdbcGetValue());
        ps.setObject(4, balanceData.jdbcGetValue());
        int rows = ps.executeUpdate();
        ps.close();
        if( rows != 1 ) throw new SQLException("Insert did not return 1 row");
    }
```

There is a class derived from the db class, offering setters and getters, in a fashion used in the so called beans in Java.

```
public class BeanAccounts extends DbAccounts {
```

```
    /**
     * Default constructor.
     */
    public BeanAccounts() {
        super();
    }

    /**
     * Gets the value of the property number.
     * @return Value as java.lang.String.
     */
    public java.lang.String getNumber() {
        return getNumberData().getValue();
    }
```

Finally, the top level class only inherits the previous ones and the application specific code, if any, can be added here.

```
package fi.accounts;
/**
 * Skeleton class for application-specific functionality.
 */
public class Accounts extends BeanAccounts {

    /**
     * Default constructor.
     */
    public Accounts() {
        super();
    }

    // @todo Insert your additional attributes and methods here.

}
```

7.5 SQL injection

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 the such a thing if the statement asks you to insert your name.

A better help prpvided 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.

7.6 Three-tier architecture and connection pooling*

This chapter has mainly covered issues in organizing software to access the tables and views in the database. In a normal web application, there can easily be hundreds of thousands of client programs such as web pages sending service requests. However, the time between a user's actions is so long that a single server process can handle many users one-by-one. If the users connections are such that the users keep on identifying themselves and the information of the user activity history is sent with the request or stored in a place where it can be read, then the server program can switch between different users easily.

Several server programs, in turn, will connect the database, but only momentarily. Each database connection requires resources from the server contacting the database as well as from the database server itself. Since distributed databases are complicated, it would nice, if possible, to have just a single database server and use it's resources sparingly. For this reason, when the servers start, they open up the database connections and then recycle them between server processes. This is called connection pooling. Some systems change dynamically the number of connections available for pooling.

The architecture, as described briefly above, has three tiers: the client tier, the application server tier, and the database tier. The database server on the database tier typically only takes care of the database. If it takes care of some part of the application logic, it should be through triggers and functions in the database. The rest of the logic is spread between the client tier, typically run in a browser, and the server tier, on the application servers.

Nowadays JDBC connection pooling is often build into the database systems, however if not, ready made source code is easy to find. One should check how many connections are being used, though.

7.7 Authorization and grant diagrams

Typical database applications have multiple users. Typically, there is a need to control what different users can see from the data. There are different ways to do this.

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?
  | 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 6 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.

Example. What privileges are needed for the following? TODO

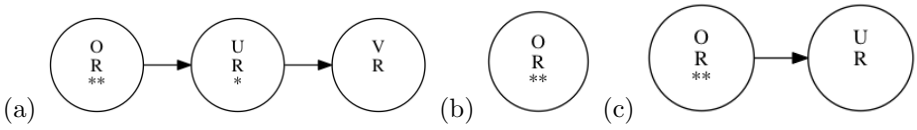


Figure 6: Grant diagrams, resulting from: (a) `O: GRANT R TO U WITH GRANT OPTION ; U: GRANT p TO V` (b) (a) followed by `O: REVOKE R FROM U` (c) (a) followed by `O: REVOKE GRANT OPTION FOR R FROM U`

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.

7.8 Table modification and triggers

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. For situations like this, DBMSs support ***triggers****, which are programs that do many SQL actions at once. 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".*

7.9 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) because 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.

7.10 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.

7.11 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)
  ) ;
```

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 "constraint" for Accounts, trying to set a maximum for the money in the bank:

```
  CONSTRAINT too_much_money_in_bank CHECK (sum(balance) < 1000000)
```

The problem is that constraints may not use aggregation functions (here, `sum(balance)`), because they are about tuples, not about whole tables. Such a constraint could be stated in an **assertion**, but these are not allowed in PostgreSQL even though they are standard SQL. We will however be able to state this and many other types of contstraints using triggers, as we will see below.

7.12 ALTER TABLE

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 contains `NULL` values, unless some other default is specified as `DEFAULT`.
- `DROP COLUMN` with the attribute name
- `ADD CONSTRAINT` with the usual constraint syntax. Rejected if the already existing table violates the constraint.
- `DROP CONSTRAINT` with a named constraint

7.13 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 is written separately

Here is the part of the syntax that we will need:

```
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:

- The statements may refer to NEW.attribute (in case of INSERT and UPDATE) and OLD.attribute (in case of UPDATE and DELETE).
- FOR EACH ROW means that the trigger is executed for each new row affected by an INSERT, UPDATE, or DELETE statement.
- FOR EACH STATEMENT means that the trigger is executed once for the whole statement.
- A trigger is an **atomic transaction**, which either succeeds or fails totally (see Chapter 9 for more details).
- The PL/PGSQL functions have also access to further information, like function now() telling the current time, and variables user telling the current database user, and TG_OP telling the type of database operation (INSERT/UPDATE/DELETE).

Let us consider some examples.

The first one is a check, that we could have also done when defining the table:

```
CREATE OR REPLACE FUNCTION check_min_savings() RETURNS TRIGGER AS $$
BEGIN
  IF NEW.type = 'savings' AND NEW.balance < 100 THEN
       RAISE EXCEPTION 'Balance too low for %', NEW.Number ;
  END IF;
  RETURN NEW;
END
$$ LANGUAGE plpgsql ;
```

The related trigger:

```
CREATE TRIGGER minSavings
  BEFORE INSERT OR UPDATE ON Accounts
  FOR EACH ROW
  EXECUTE PROCEDURE check_min_savings() ;
```

So, we can see that we can use triggers to check consistency. Since the activities of functions are not limited, we could also here check the sum of the balances of all accounts in the bank. The second example is doing exactly that: limiting the maximum total balance of the bank. This trigger also doesn't change anything, and could therefore be defined as an ASSERTION, if they were permitted.

```
CREATE OR REPLACE FUNCTION maxBalance() RETURNS TRIGGER AS $$
BEGIN
  IF ((SELECT sum(balance) FROM Accounts) > 16000)
    THEN RAISE EXCEPTION 'too much money in the bank' ;
  END IF ;
END
$$ LANGUAGE 'plpgsql' ;

CREATE TRIGGER max_balance
  AFTER INSERT OR UPDATE ON Accounts
  FOR EACH STATEMENT
  EXECUTE PROCEDURE maxBalance() ;
```

The third trigger introduces some application logic: it is a trigger that updates balances in Accounts after each inserted Transfer. In PostgreSQL, we first have to define a function that does the job by `CREATE FUNCTION`. After that, we create the trigger itself by `CREATE TRIGGER`.

```
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' ;

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!

```
CREATE OR REPLACE FUNCTION maxBalance() RETURNS TRIGGER AS $$
BEGIN
  IF ((SELECT sum(balance) FROM Accounts) > 16000)
    THEN RAISE EXCEPTION 'too much money in the bank' ;
  END IF ;
END
$$ LANGUAGE 'plpgsql' ;

CREATE TRIGGER max_balance
  AFTER INSERT OR UPDATE ON Accounts
  FOR EACH STATEMENT
  EXECUTE PROCEDURE maxBalance() ;
```

Our fourth example is for logging the transfers:

```
CREATE OR REPLACE FUNCTION make_transfer_log() RETURNS TRIGGER AS $make_transfer_log$
BEGIN
  IF (TG_OP = 'INSERT') THEN -- special variable TG_OP tells the operation
    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; -- no matter, as the update has already been done
END $make_transfer_log$
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() ;
```

There is a particular risk with using triggers and functions: The function executions may activate further triggers and in fact it is possible to create a non-terminating loop of trigger activations and function executions. A non-terminating execution is generally bad news, but additionally the functions may be rapidly filling up the database as the execution is progressing. One safety feature is to use the pg_trigger_depth() function which returns a 0 if we are triggering the first function execution and when functions trigger further functions, each time the value of pg_trigger_depth() increases by 1.

The combinations of BEFORE/AFTER, OLD/NEW, and perhaps also ROW/STATEMENT, can be a bit tricky. The can be 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.

7.14 Transactions

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. However, one of the important facilities that database systems give is the possibility of concurrent access to data by several users. This requires special techniques. Suppose the following scenario. Assume that user Mary wants to record a fact that 10 people moved from Finland to Sweden.

So, Mary issues the SQL statements

```
UPDATE countries
SET population = population - 10 WHERE name = 'Finland' ;

UPDATE countries
SET population = population + 10 WHERE name = 'Sweden' ;
```

Even though the `UPDATE` statement is just one statement, it should be clear that the computer program (like database system) maintaining the data needs to do several operations: It needs to read the population or a country and then it needs to make a calculation for the new value, and then it finally needs to write back the changed value. So, deep inside the database system, the `UPDATE` statements will be executed as a process containing a series of operations. Let's assume they are as follows.

```
Mary : READ population value x from the row where name = 'Finland'
Mary : WRITE population value x-10 to the row where name = 'Finland'
Mary : READ population value y from the row where name = 'Sweden'
Mary : WRITE population value y+10 to the row where name = 'Sweden'
```

Various things may stop the processing at any moment, such as electrity cable being cut of someone powering the computer off. Let's suppose that such a failure happens right after Mary has issued her commands and, in practice, her process stops right berfore the last operation, leaving the population value for Finland updated and the population value for Sweden not updated. If no proper attention to the situation is paid, this will lead into incorrect values in the database. If e.g. Mary just re-issues the commands after the failure, then 10 gets subtracted twice from Finland's population value while only adding it once to Sweden's population value. The set of operations that Mary wants to perform in the database is *atomic* - either none or all of them should be performed.

Suppose now, that at the same time user John wants to record a fact that 2 people moved from Sweden to China. He issues that SQL statements

```
UPDATE countries
SET population = population - 2 WHERE name = 'Sweden' ;

UPDATE countries
SET population = population + 2 WHERE name = 'China' ;
```

which leads into the following operations

```
John : READ population value z from the row where name = 'Sweden'
John : WRITE population value z-2 to the row where name = 'Sweden'
John : READ population value w from the row where name = 'China'
John : WRITE population value w+2 to the row where name = 'China'
```

A computer is fast enough to serve several users without a significant delay by giving them turns when some of their operations can be performed. Similarly, many users can be connected to a database system, and the system serves them by turns. This means that their operations are interleaved somehow in the execution. Let's assume now that the operations by Mary's and John's processes interleave as follows.

```
Mary : READ population value x from the row where name = 'Finland'
Mary : WRITE population value x-10 to the row where name = 'Finland'
Mary : READ population value y from the row where name = 'Sweden'
John : READ population value z from the row where name = 'Sweden'
John : WRITE population value z-2 to the row where name = 'Sweden'
John : READ population value w from the row where name = 'China'
John : WRITE population value w+2 to the row where name = 'Sweden'
Mary : WRITE population value y+10 to the row where name = 'Sweden'
```

For the updates to be done correctly, the final value of Sweden's population should be `p+10-2` where p is Sweden's population before the updates. When all of the operations of Mary's process are performed before all of the operations of John's process, we get the desired outome. However, observing closely the interleaved operations reveals that in that case the final value of Sweden's population ends as `p+10`, which, of course, is not the desired outcome.

The reason why things go wrong is that both Mary's and John's operations are meant to be a whole that is executed in *isolation* from other users. Before or after that whole the other users can access the data and everything should be ok.

Isolation and atomicity can be supported by using *transactions*, each of which containing a sequence of operations such that either they all get executed or none of them gets executed. In SQL, a transation can be started by giving the command

```
START TRANSACTION
```

After that the SQL operations changes in the database will be 'buffered' in such a way that they are only visible for the user herself, and not to others. The transaction can be terminated by giving command

```
COMMIT
```

making the updates permanent and visible to others, or by command

```
ROLLBACK
```

which cancels all the updates of the transaction.

Once the changes are made permanent, they are assumed to be *durable*, meaning that the database system will contain the data after failures like the computers running the database system being shut down abruptly.

Atomicity, Consistency, Isolation, and Durability make up the so-called *ACID* properties, generally expected from database systems.

A **transaction** is a sequence of statements that is executed together. A transaction succeeds or fails as a whole. For instance, a bank transfer can consist of two updates, collected to a transaction:

```
BEGIN ;
UPDATE Accounts
  SET (balance = balance - 100) WHERE holder = 'Alice' ;
UPDATE Accounts
  SET (balance = balance + 100) WHERE holder = 'Bob' ;
COMMIT ;
```

If the first update fails, for instance, because Alice has less than 100 pounds, the whole transaction fails. We can also manually interrupt a transaction by a `ROLLBACK` statement.

Individual SQL statements are automatically transactions. This includes the execution of triggers, which we used for bank transfers in Chapter 6. Otherwise, transactions can be created by grouping statements between `BEGIN` and `COMMIT`.

Transactions are expected to have so-called **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.

Hint. The main purpose of transactions is to keep the data and data access consistent. But a transaction can also be faster than individual statements. For instance, if you want to execute thousands of `INSERT` statements, it can be better to make them into one transaction.

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 in the chapter on SQL in software applications.

7.15 Maintaining isolation*

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 problems appear. The need for isolation is motivated above. There are two main approaches for maintaining isolation: *locking* and *multiversioning*.

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 lockes: a Read lock (R-lock, shared lock), and a Read-Write lock (RW lock, exclusive lock). If a process needs to read an 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*.

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.

To maintain maximum isolation, the transaction only gives releases the locks at COMMIT or ROLLBACK. 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 tables big.

Multiversioning is based on the idea that several versions of the data item are stored. Transactions are given a number or a timestamp when they start. The basic idea is that a writing transaction updates the list of versions by writing a new version along with its timestamp in the list. Reading transactions select from the list the value valid for their time. For instance, if the list, assumed here to be ordered, has <timestamp, value> pairs <100,5.0>, <104,2.0>, <109,11.0> for some data time 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.

We can remove such values from the list that there are no such transactions anymore that would need those values. For instance, if all executing transactions have at least timestamp 105, then we could remove <100,5.0> from the list.

If removal of old versions is done appropriately, then reads are always successful. However, to figure out if the writes can be executed, the reads have to record the last timestamp when a data itme is read. Let's suppose that the data item A above is read at time 112 and a transaction with timestamp 111 wants to write a value for A. Now, to simulate a correct order of transactions, then the transaction with timestamp 111 cannot write that value anymore, as the value it wants to write should have been read by the transaction with timestamp 112, and, consequently, the transaction with timestamp 111 must be rolled back. This, however, may lead to a need to roll back other transactions that have read data written with timestamp 111.

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.

7.16 Interferences and isolation levels

If we maintain the transactions as in the previous section, then the effects of the execution are the same as if the transactions would be executed in a serial order - thereby we 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 requireent 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**. These levels are defined as follows:

- SERIALIZABLE: the transaction only sees data that was committed before the transaction began.
- 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 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** of the following kinds:

- **Dirty reads**: read data resulting from a concurrent uncommitted transaction.

  ```
  T1 __________________READ a ____________
  T2 ________INSERT a _________ROLLBACK___
  ```

- **Non-repeatable reads**: read data twice and get different resulats (because of concurrent committed transaction that modifies or deletes the data).

  ```
  T1 _____READ a ________________________READ a
  T2 _____________UPDATE a=a'___COMMIT _______
  ```

- **Phantoms**: execute a query twice and get different results (because of concurrent committed transaction).

  ```
  T1 SELECT * FROM A ___________________________SELECT * FROM A
  T2 _________________INSERT INTO A a___COMMIT _______________
  ```

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.

7.17 The ultimate query language?*

SQL was once meant to be a high-level query language, easy to learn for non-progammars. In some respects, it is like COBOL: very verbose with English-like keywords, and a syntax that with good luck reads like English sentences: *select names from countries where the continent is Europe*. Real natural language queries of course have a richer syntax and may require effort from the compiler to execute. Here are some examples of syntactic forms easily interpretable in SQL:

```
what is the capital of Sweden
SELECT capital FROM countries WHERE name='Sweden'

which countries have EUR as currency
SELECT name FROM Countries WHERE currency='EUR'

which countries have a population under 1000000
SELECT name FROM Countries WHERE population<1000000

how many countries have a population under 1000000
SELECT count(*) FROM Countries WHERE population<1000000

show everything about all countries where the population is under 1000000
SELECT * FROM Countries WHERE population<1000000

show the country names and currency names for all countries and currencies
  such that the continent is Europe and currency is the currency code
SELECT Countries.name, Currencies.name FROM Countries, Currencies
  WHERE continent='Europe' AND currency=Currencies.code
```

It is possible to write grammars that analyse these queries and translate them to SQL, just like and SQL compiler translates SQL queries to relational algebra. This was in fact a popular topic in the 1970's and 1980's, after the successful LUNAR system for querying about moon stones:

http://web.stanford.edu/class/linguist289/woods.pdf

Natural language question answering is becoming popular again, in systems like Wolfram Alpha and IBM Watson. They are expected to give

- more fine-grained search possibilities than plain string-based search
- support for queries in speech, like in Google's Voice Search and Apple's Siri.

The main problems of natural language search are

- precision: without a grammar comparable to e.g. SQL grammar, it is difficult to interpret complex queries correctly
- coverage: queries can be expressed in so many different ways that it is difficult to have a complete grammar
- ambiguity: a query can have different interpretations, which is sometimes clear from context (but exactly how?), sometime not:
  - *Please tell us quickly, Intelligent Defence System: Are the missiles coming across the ocean or over the North Pole?*
  - Calculating..... *Yes.*

Interestingly, once we can solve these problems for one language, other languages follow the same patterns:

```
vad är Sveriges huvudstad
vilka länder har EUR som valuta
vilka länder har en befolkning under 1000000
hur många länder har en befolkning under 1000000
visa allt om alla länder där befolkningen är under 1000000
visa landnamnen och valutanamnen för alla länder och valutor
  där kontinenten är Europa och valutan är valutakoden
```

Natural language queries are in the intersection of database technology and artificial intelligence. The problem can be approached incrementally, by accumulating technology and knowledge. Much of the research is on collecting the knowledge automatically, by for instance using machine learning. This is the case in particular when the knowledge is in unstructured form such as text. However, when the knowledge is already in a database, the question becomes much more like query compilation.

8 Introduction to alternative data models

*The relational data model has been dominating the database world for a long time. But there are alternative models, some of which are gaining popularity. XML is an old model, often seen as a language for documents rather than data. In this perspective, it is a generalization of HTML. But it is a very powerful generalization, which can be used for any structured data. XML data objects need not be just tuples, but they can be arbitrary trees. XML also has designated query languages, such as XPath and XQuery. This chapter introduces XML and gives a summary of XPath. A more general approach is that of **document databases**, which are designed in particular textual structured or semi-structured data. One example of such systems is MongoDB, shortly also covered in this chapter. A particular concern in document databases is a query language or facility, which has search structures more suitable for such documents than SQL. The chapter concludes with Big Data and "NoSQL" approaches more generally, with Cassandra as example.*

8.1 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>
```

```
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 7: A grammar of XML and XPath.

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 7, 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)>
```

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 8 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.

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:

```
-- 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 8: A complete XML document for arithmetic expressions and the corresponding Haskell code, with $23 + 15x$ as example.

```
<!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.

8.2 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 7. Here are some examples:

- `/Countries/country` all <`country`> elements right under <`Countries`>
- `/Countries//@name` all values of `name` attribute anywhere under <`Countries`>
- `/Countries/currency/[@name = "dollar"]` 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.

8.3 XML and XPath in the query converter*

TODO: update this in the web version of qconv

The Query Converter has a functionality for converting an SQL database into an XML object, with its schema as DTD. It also implements a part of the XPath query language. The command

```
x
```

without an argument prints the current database as an XML document (with DTD and the elements). The command `xp` takes an XPath query as an argument and shows the result of executing it:

```
xp /QConvData//@name
```

extracts all values of the `name` attribute everywhere in the descendants of `/QConvData`. (The XPath interpreter is not complete and has some bugs.)

The XML encoding of SQL tuples uses attributes rather than child elements. It does not (yet) express the `IDREF` constraints. All tables are wrapped in an element called `QConvData`.

The command

```
ix <FILE>
```

reads an XML file, parses it, and validates it if it has a DTD. The `xp` command with an XPath query applies to all XML files that have been read either in this way or by conversion from SQL. Hence one can also query XML databases that are not representable as SQL.

8.4 MongoDB*

MongoDB, another public open-source database product, uses the document metaphora for data storage. This makes it flexible and useful for TODO

8.5 Pivot tables and OLAP*

TODO

8.6 NoSQL data models*

Big Data is a word used for data whose mere size is a problem. What the size is depends on many things, such as available storage and computing power. At the time of writing, Big Data is often expected to be at least terabytes (10^{12} bytes), maybe even petabytes (10^{15}).

In relational databases, each table must usually reside in one computer. 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)

NoSQL is not just one data model but several:

- key-value model

- column-oriented model
- graph model

We will take a closer look at Cassandra, which is a hybrid of the first two models.

8.7 The Cassandra DBMS and its query language CQL*

"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.[6] 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:

[6] The distribution and replication of data in Cassandra is more like in Dynamo.

```
cqlsh> CREATE KEYSPACE mykeyspace
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 1 };
```

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.

8.8 Physical arrangement of data on disk*

HBase is another open-source commmunity-driven implementation based on the 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. Similar possibility exists with Cassandra.

The data is also replicated, in a configurable way. The data is stored using the Hadoop HDFS file system.

Further features include so called **sharding** where data placement on servers is defined based on the data values. In HBase the data can be split horizontally (using row sets e.g. placing data of customers on a particular continent physically nearby) and vertically by using attribute families.

If avoiding downtime and service break is a priority, then **key-value stores** such as Amazon's key-value database or ??? There, the data is hashed on disk based on key values. In ??? the data storage can dynamically be extended by introducing new servers on-the-fly. As a result, the system will not encounter service breaks due to data reorganization. However, data retrieval is harder than in HBase - the basic choice to access the data is writing a MapReduce program.

A further complication due to distributed storage and parallel access lies within the reliability and transactional aspects. However, discussion of these requires considerable knowledge of distributed system theory, and as such, it is skipped in this book.

8.9 NoSQL and MapReduce*

The precise definition of NoSQL is "not only SQL", which does not exclude SQL but adds to it other ways to access that data. In practice, however, when talking about NoSQL databases, people often mean systems that do not support SQL at all. The main reason for such systems are computational needs that are hard to fulfill with SQL, and, in particular combined with large amounts of data. These models are often implemented to support distributed-parallel computing using a computing cluster. There are also traditional database systems that support distributed large data sets, but there are primarily two things that drive the users to the new systems. The first is money, as these systems are often open-source community-developed software products, and participating in the development even gives a possibility to support the features necessary for a company, or alternatively just gives a cost-efficient solution. The other are the application needs, as there are needs to do perform such machine learning and data mining tasks for which SQL is not a feasible option, and certain traditional database services may not be so important. In particular, the MapReduce computational model is popular.

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 functinal programming. The computations can be made in parallel in different computers. It was originally developed at Google, on top of the **Bigtable** data storage system.

TODO more details of MapReduce

Also, the traditional ACID properties may not be required by the new applications - it is not so fatal if sometimes some comment in the social media is lost, some users see some dirty data which has been deleted, and so on.

The efficiency of Big Data databases often cricitally depends on the data arrangement on disk. We will review some options for this in this chapter, along with some Big Data database products.

8.10 Further reading on NoSQL*

The course book covers XML, in chapters 11 and 12. The NoSQL approach is more recent, so we must refer to other material. I have found the following useful and readable:

- CQL Reference Manual https://docs.datastax.com/en/cql/3.1/index.html *The source used in this chapter.*
- Martin Fowler, Introduction to NoSQL https://www.youtube.com/watch?v=qI_g07C_Q5I *A very good overview talk without hype.*
- "Cassandra Essentials Tutorial". http://www.datastax.com/resources/tutorials *Recommended by Oscar Söderlund in his Spotify guest talk.*
- Kelley Reynolds, "Understanding the Cassandra Data Model from a SQL Perspective", 2010. http://rubyscale.com/blog/2010/09/13/understanding-the-cassandra-data-model-from-a-sql-perspective/
- Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C; Wallach, Deborah A; Burrows, Michael 'Mike'; Chandra, Tushar; Fikes, Andrew; Gruber, Robert E, "Bigtable: A Distributed Storage System for Structured Data", 2006. http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, "Dynamo: Amazon's Highly Available Key-value Store", 2007. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- Jeffrey Dean and Sanjay Ghemawat, MapReduce: "Simplified Data Processing on Large Clusters", 2004. http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf
- Ralf Lämmel, "Google's MapReduce Programming Model - Revisited", 2008. http://userpages.uni-koblenz.de/~laemmel/MapReduce/paper.pdf

A Appendix: SQL in a nutshell

Figure 9 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 9, and hence incomplete. But it is completely formal and is used for implementing an SQL parser.[7]

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

[7] 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 9: A grammar of the main SQL constructs.