Information Systems Lecture Notes

Gábor Bodnár

RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria
email: Gabor.Bodnar@risc.uni-linz.ac.at
www: http://www.risc.uni-linz.ac.at/people/gbodnar

January 23, 2005

Contents

Chapter 0

Introduction

0.1 Foreword

The intended audience of the course is students of mathematics in the fifth semester of their studies. The goal is to provide a brief but up to date introduction to information systems with special attention to web-based solutions. The topic is treated as a supplementary subject to the studies in mathematics, with no desire of comprehensive, detailed and far reaching discussion of any of the areas it deals with. These notes introduce only the essential concepts providing additional examples for easier understanding. After the course, the student should have a clear concept of elementary problems and solution techniques in relational data modeling and he should have basic knowledge in relational database manipulation. With additional programming skills (PHP/Perl) he should be able to write simple web-interfaces to relational databases, using SQL. He should be able to create moderately complex XML documents, validate them using XSchema, address its parts using XPath and, provided he knows HTML, transform it into a displayable HTML document by XSLT.

These lecture notes are built up as follows. In Chapter 1 we present an introduction to the theoretical foundations of relational data modeling. In Section 1.2 we discuss the Entity-Relationship (ER) model from the basic concepts till the standard ER-techniques to be used for data modeling. In Section 1.3 we continue by applying the results of Section 1.2 for relational database design, now dealing also with manipulation of tables and redundancy elimination i.e. normalization. Section 1.4 contains the SQL specifications (a standard language to form queries for- and to create, destroy and manipulate relational databases) and via several examples demonstrates its usage.

In Chapter 2 we briefly discuss more practical issues of on-line databases to "scratch the surface" of the enormous amount of knowledge accumulated in the nowadays ubiquitous web-based information systems with relational database support. In Section 2.1 we discuss two relevant kind of on-line information systems: transaction management systems, and static archives (which are often filled with transaction data for later analysis). We describe the arising problems in both environments and the rigorous requirements they pose for information systems which intend to achieve success in these scenarios. Finally, in Section 2.2 we introduce state of the art *free* software from which one can build web based

information systems for virtually any purpose.

In Chapter 3 we provide a small survey on a few XML technologies which are related to hierarchical data description and retrieval. In Section 3.1 we introduce the generic XML framework with the example of XHTML followed by a discussion on data identification problems and solutions in Section 3.2. Section 3.3 deals with document validation. Section 3.4 provides means to refer to parts of XML document and to navigate in them, and in Section 3.5 we present an introduction to the topic of XML document transformations e.g. for data presentation using HTML. Section 3.6 addresses the problem of data retrieval in more general context.

As this document is of purely introductory nature, I tried to supply enough references for further studies. Most of the information sources are freely available on the Internet.

0.2 Information Sources

In this section we summarize the information sources these lecture notes rely on. The sources are by no means the only ones that can be used to study the topics discussed in these notes, however they provide a good introduction to them.

For the (relational) data modeling part of the notes a standard reference is [3], which gives an extensive overview and a thorough introduction to databases. It is also recommendable to look at the excellent lecture notes of [14].

For the part on on-line databases one can also use [3], and in particular for search engines [7].

For the XML part a brief, to the point introduction can be found in [2]. However the original specifications are all available on line: [8, 9, 10, 13, 11].

It cannot be emphasized enough to search the Internet for further information. By the nature of the topic, an enormous amount of information is available on-line, awaiting to be discovered.

Chapter 1

Data Modeling

Data modeling is an important part of database design. The data model focuses on what data should be stored in the database, exploring the relation between data entities. The goal of the data model is to ensure that all data object required by the database is accurately represented. It provides the overall logical structure (the "blueprint") of the database. In this chapter we take a look at relational data modeling techniques and SQL, a commonly used data manipulation language.

1.1 Introduction

Databases are now ubiquitous in any areas where information has to be stored and processed. Their importance comes from the value of information they store. The most important criteria for them can be described by two keywords reliability (covering many subareas from robustness to issues related to concurrency and security) and efficiency (covering not only data manipulation speed but also its flexibility towards new requirements or the degree to which database supports application software development).

A *Database management system* (DBMS) consists of

- a collection of structured, interrelated and persistent data,
- a set of application programs to access, update and manage the data.

Example 1.1. Let us take tasks of the administration of a university. It has to maintain files of teachers, students, courses, the relationship between courses and students for each semester, the grades arising from the previous relationships and countless other things. The administration has to be able to perform the following tasks: add/remove a student/teacher/course, assign courses to teachers/students each semester, add/modify grades of a student for the courses she took, generate the list of grades of students, generate the list of courses given by a teacher, etc.

This amounts to managing large pieces of data with additional constraints (for instance courses can build on each other) and numerous reports to be generated. Moreover it should not be too difficult to upgrade the system to meet new requirements that come from the real world (if the law changes, a new faculty/institute/branch of studies starts, etc.).

Thinking a bit about this example shows the major problems we run into when trying to solve the problem in an ad-hoc way. Here are the problems commonly known to be coped with in a DBMS:

- Data redundancy and inconsistency.
  - The same information should not appear at several places of the system.
  - Related data should be updated consistently.
- Data integrity.
  - The system should ensure that the data stored in it fulfills certain prescribed constraints.
  - The system should adopt to the change of the constraints.
  - The system should be able to recover from crashes.
- Data access.
  - The system should be able to generate answers to a large class of queries.
  - The system should support efficient data retrieval (e.g. by indexing, hashing).
- Data isolation.
  - Data(-elements) of different types and/or magnitudes should be handled by the system (like text documents, numerical data, photos, etc.).
- Concurrency.
  - The system should be able to work with multiple users at the same time.
  - The concurrent modifications should not disturb the consistency of the data.
  - Avoid deadlocks.
- Security.
  - Handle access rights for users.
  - Secure remote access to the database.

Data abstraction is the most important tool to treat the above problems conceptually. There are three levels of abstraction (sometimes also referred to as database schemes):

- *Physical level*: It deals with the storage and retrieval of the data on the disk (data and index file structures). This level is usually hidden from the users by the DBMS.
- *Conceptual level*: In this level the questions of what data are stored and what are the relationships between these data are decided. This is the level of the database administrator

- *View level*: It describes a subset of the data in the database for a particular set of users. The same data can be viewed in many different ways (for instance the teacher wants list of the students subscribed for his course while the dean of studies is interested in the number of people attending the course). This is the level of the users of the database.

Data models are collections of conceptual tools to describe data, data relations, data constraints and data semantics. There are object-based, record-based and physical data models.

Object-based models:

- Describe data on the conceptual and view levels.
- Provide flexible structuring capabilities.
- Data constraints can be specified explicitly.

Examples are: Entity-Relationship (ER) model, Object-oriented model, Functional model, etc. etc.

Record-based models:

- Describe data on the conceptual and view levels.
- Used for databases with fixed record structure; the sizes of the fields of the records are usually also fixed.
- There exist query languages for record based models (e.g. SQL).

Examples are: Relational model, Network model, Hierarchical model, etc.

A snapshot of the data contained in a database at a given point in time is called an *instance* of the database. In contrast, the database *scheme* is the overall design of the database (structure, relations, constraints in general).

An important concept is the one of data independence, which expresses that changes in the the database scheme at a given level of abstraction should not affect the schemes at higher levels. Thus we have:

- *Physical data independence*: Changes in the physical level should not make changes in the data model necessary and therefore application programs need not be rewritten.
- *Logical data independence*: Changes in the conceptual level should not make necessary the application programs to be rewritten.

Standard terminology in DBMSes:

- *Data Definition Language (DDL)*: Used to describe the database scheme (structure, relations, constraints). The compiled DDL statements are called the *data directory* (the metadata describing the database).
- *Data Manipulation Language (DML)*: Used to pose queries (select) or to modify (insert, update, delete) the database. In *nonprocedural* DMLs the user only specifies what data is needed, while in *procedural* DMLs also the way it should be retrieved.

- *Database Manager*: The program which provides an interface between the users of the database (that can be application programs) and the data stored in the database on the physical level. This program is responsible for enforcing most of the requirements for a DBMS.
- *Database Administrator*: The person controlling the DBMS. He defines database schemes, defines the storage structure and the access methods, provides authorization for data access, specifies integrity constraints.
- *Database Users*: Application programmers (interact with the system via DML calls from their programs), Naive users (interact with the system via application programs).

The overall structure of a DBMS consists of the following components:

- *File manager*: Responsible for low-level data storage and retrieval on the disk (this functionality is usually provided by the operating system).
- *Database manager*: See above.
- *Query processor*: Translates statements in a query language into low-level instructions the Database manager understands.
- *DDL compiler*: Converts DDL statements into database metadata stored in the data directory.
- *Data file*: Store the data themselves.
- *Data directory*: Stores information about the structure of the database.
- *Indices*: Accelerate data retrieval from the database.

1.2 The Entity-Relationship Model

The ER model views the real world as the set of entities and relationships between them. This modeling technique maps well to the relational model (the constructs of the ER model can easily be turned into corresponding tables) discussed in Section 1.3.

1.2.1 Entities, Attributes, Relationships

An *entity* is an object with identity, i.e. an object which is distinguishable from other objects. Entities are either concrete or abstract objects, persons, things, documents, events, programs, etc., about which we intend to collect information.

An *entity set* is the set of all entities of the same type (for instance the set of all students inscribed to the university, of the set of all cars registered in Austria).

Entities are described by *attributes* which are, from the formal point of view, functions assigning to an entity an element from a *domain* of permitted *values*.

Example 1.2. Let the entity set be the students of the university, let us describe a student for the sake of simplicity only with the attributes: first-name, last-name, student-ID, birth-date. The first-name and last-name attributes can take

values from the nonempty strings of latin characters which will be the domain of these attributes (viewing the attributes as functions, from a strict mathematical point of view what we call here domain is actually the codomain of the function, but let us not change the well established terminology of the ER model). The student-ID can be a seven-digit positive numbers and the birth-date can be a string of the format dd-mm-yyyy with the usual semantics.

An entity (a particular student of the university) can be described then as {(first-name, Susan),(last-name, Becker),(student-ID, 0256098), (birth-date, 27-06-1978)}. Of course, in any kind of information processing the entity appears only via its description by the set of attributes which are relevant from the point of view of the task to be carried out.

A *relationship* represents association between two or more entities.

A *relationship set* is the set of relationships of the same type. Recall that a relation over the sets $E_i, \ldots, E_r$ is a subset of $E_1 \times \cdots \times E_r$ (with the ER terminology this is a relationship set).

Example 1.3. Continuing Example 1.2, let another entity set be the set of courses offered in the current semester at the university. Let the attributes describing a course be the course-ID, teacher-ID, course-title. Let a course be described by {(course-ID, 322527),(teacher-ID, 024532),(course-title, Analysis I)}. If the student of the other exercise takes this course, there is a relationship between the two entities (the student and the course).

If we take the entity sets of all students and all the courses offered, the arising relationship set will describe which student takes what courses this semester.

It can be easily recognized that a relationship can also be taken as an abstract entity, and hence a relationship set can be described just as an entity set.

Example 1.4. Continuing Example 1.3, the relationship set that associates students to the courses they take in a semester can be described via the entity set whose elements are described by the attributes student-ID and course-ID.

The relationship between Susan Becker and the Analysis I course would then be described by {(student-ID, 0256098),(course-ID, 322527)}.

1.2.2 Classification of Relationships

The *degree* of a relationship is the number of entity sets it associates to each other. In Example 1.4 the degree of the relationship is two (students, courses).

Higher degree relationships can be decomposed into degree-two ones, thus these are the most important occurrences.

The (mapping) *cardinality* of a relationship expresses the number of entities that can be associated via it. The cardinality of a binary relationship between entity sets A and B can be

One-to-One (1:1) An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.

One-to-Many (1:N) An entity in A is associated with any number of entities in B. An entity in B is associated with at most one entity in A.

Many-to-Many Entities in A and B are associated with any number from each other.

Remark 1.5. Because of symmetry, a Many-to-One relationship between A and B is a One-to-Many relationship between B and A.

Example 1.6. A typical One-to-One relationship would be the one between the students and the ID cards: Every student can have at most one ID card and every ID card can belong to at most one student (if the student lost the card, it should be immediately invalidated but it might take some time to produce a new card).

A typical One-to-Many relationship would be the one between members and books of a library: A member can borrow many books but a book (which means here an instance) can be borrowed by only one person at a time.

A typical Many-to-Many relationship would be the one between students and courses: A student can take many courses and a course can be attended by many students.

The *existence dependency* of a binary relationship expresses that the existence of an entity is dependent on another, related entity. If in a relationship between entities X and Y, the existence of X depends on the existence of Y, we call Y the *dominant* and X the *subordinate* entity.

Example 1.7. Let us take the examples the relationships between teachers and courses and between students and courses. The first one has existence dependency with teachers being the dominant entities and courses the subordinate ones. That is, for each course there must be someone to teach it. There is no existence dependency between students and courses, which means that certain courses are not obligatory for any students, thus there might be nobody taking them in a given semester.

The *direction* of a binary relationship indicates the originating entity set. The entity set from which a relationship originates is the parent, while the one the relationship terminates is the child entity. In a One-to-One relationship the direction is from the independent entity to a dependent entity. If both entities are independent, the direction is arbitrary. With One-to-Many relationships, the entity occurring once is the parent. The direction of Many-to-Many relationships is arbitrary.

Example 1.8. Let us take the One-to-One part in Example 1.6. The direction should be from students to ID-cards, as the existence of a student implies the existence of an ID-card (could be invalidated if the student has lost it).

1.2.3 Keys

Since entities must be distinguishable, we must be able to choose a set of attributes for each entity set, so that there is not two entity in the set described by the same set of attribute values. Usually the set of attributes is more than enough to distinguish entities, and what we try to do is to find small subsets of attributes that still have this property.

A *superkey* is a nonempty set of attributes which allows us to uniquely identify an entity of the entity set.

A superkey which does not have a proper subset which is also a superkey is called a *candidate key*.

A *primary key* is a candidate key which is chosen by the database administrator to identify entities in an entity set.

Remark 1.9. There might be more than one candidate keys (since set inclusion is only a partial order).

Which candidate key becomes a primary key is apparently a matter of choice.

An entity set that does not possess sufficient attributes to form a primary key is called a *weak entity set*. One that does have a primary key is called a *strong entity set*. A weak entity set must be part of a One-to-Many relationship in a subordinate role while the entity set with the dominant role must be a strong one so that it allows distinction between the entities of the weak set via the relation.

A *discriminator* is a set of attributes in the weak entity set that allows the distinction of the entities via a given relation with a strong entity set on which the weak one existence depends. A *primary key for a weak entity set* is the primary key of the strong entity set with which it is related together with a discriminator with respect to this relation.

Example 1.10. Take the entity set of all the telephones in Austria, where an entity is described with the attribute phone-number (without any country or area codes). This is a weak entity set because many phones in different areas can have the same number (although the entities, that is the phones, are different). Make the system work we have to introduce the entity set of phone areas, where an entity is described by the attribute area-code.

The One-to-Many relation associates to an area each phone in it. Since the areas cover Austria completely, no phones exist without an area, thus there is an existence dependency in which the areas are the dominant and the phones are the subordinate entities.

The area-code is a superkey, a candidate key, and (after we declare it so) a primary key of the entity set of phone areas. The phone-number is a discriminator of the entity set of telephones with respect to the primary key we have chosen for the phone areas, via the relation we established between areas and phones. And the area-code with the phone number forms a primary key for the entity set of telephones in Austria.

The primary key of a relationship set is formed from the primary keys of the entity sets in the relationship set.

A *foreign key* is a set of attributes for an entity set in a One-to-Many binary relation on the "Many" side, which identifies the corresponding parent entity for each entity of the set. Foreign keys provide means to maintain data integrity in databases (called referential integrity). For instance in the example of Figure 1.1, if we add member-ID to the attributes of the Book entity set (with a special null value for books not borrowed) it becomes a foreign key that identifies the borrower of any book in the library.

1.2.4 Entity-Relationship Diagrams

ER diagrams are a graphical way to express the logical structure of a database. There is no standard for ER diagrams; therefore there are many "dialects" appearing in articles, CASE tools, etc. We use the following notation:

- Entity sets are represented by labeled rectangles, for weak entity sets the rectangles have double border. Entity set labels (names) should be singular nouns.

- Relationship sets are represented by a solid line connecting two entity sets. The name of the relationship is written beside the line. Relationship names should be verbs.
- Attributes, when included, are listed inside the entity set rectangle. Attribute names should be singular nouns. Members of the primary key can be denoted by underlining.
- If the line ends in an arrow, the relationship cardinality is Many for the entity set at that end. Otherwise the cardinality is One.
- Existence dependency is denoted by a bar crossing the line of the relationship set at the dominant entity set.

Figure 1.1 provides an example.

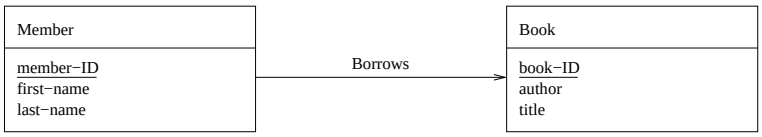


Figure 1.1: An ER diagram

A database conforming to an ER diagram can be represented by a collection of tables (more on relational databases comes in Section 1.3). For each entity set corresponds a table with as many columns as many attributes the entity set is described with. A row of the table will represent an entity, containing the corresponding attribute values. For weak entity sets we attach to the columns of the primary key of the strong entity set on which it depends. Relationship sets can also be described this way.

Tables will contain only a subset of all possible rows: the ones which represent entities in the system we describe. As the system changes in time, we may add, delete or modify rows. Table 1.1 shows an example.

area-code	area-name
0732	Linz
0662	Salzburg
0512	Innsbruck
05574	Bregenz
01	Wien
⋮	

phone-number	owner-first-name	owner-last-name	area-code
4554323	Monika	Beispiel	01
243486	Thomas	Muster	0732
875678	Hans	Schmidt	0662
875678	Hans	Schmidt	0732
34452	John	Example	05574
⋮			

Table 1.1: ER diagram to tables

1.2.5 Entity-Relationship Design

Many-to-Many relationships cannot be taken over directly to the relational model, which is important in practice; therefore such relationships should be resolved to One-to-Many relationships as early in the design phase as possible. The method is to introduce a new entity type for the relation itself and associate the original entity types to this new one via One-to-Many relationships.

Example 1.11. Take the entity sets of students and courses and the Many-to-Many relationship set between them that associates students and the courses they attend. We introduce then a new entity set with name Attendance with two attributes: student-ID, course-ID, whose entities express the fact that a certain student attends a given course. Then establish a relationship between a student and an entity in Attendance if the student-ID is the same, and we act analogously for courses and entities of Attendance. The ER diagram is shown in Figure 1.2.

Please note that the new ER diagram requires that for each Attendance entity a student and a course entity exists.

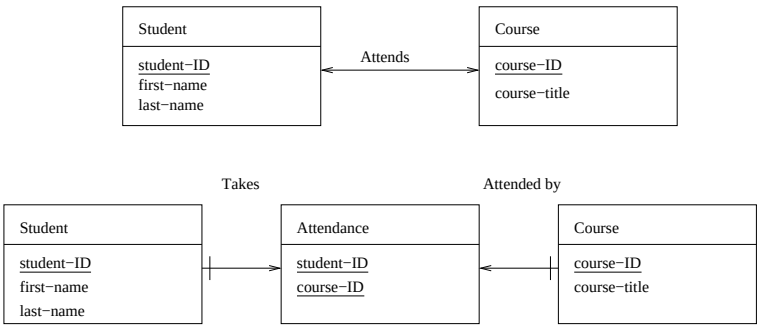


Figure 1.2: Resolving Many-to-Many relationships

Complex relationships of degree greater than two can, and should, be resolved to binary One-to-Many relationships using the same technique as for resolving Many-to-Many relationships (see the first two ER diagrams in Figure 1.3). Also description of relationships between relationships should use this technique for a conceptually clearer presentation. That is, one should create new entities for relationship sets, considering them on a higher abstraction level, and establish the relationships between them (see Figure 1.3). This method is called *aggregation*.

If two or more entity sets share common attributes we can form their *generalization* by taking only the common attributes, which can be considered as creating an abstraction type (entity set) that describes similarities of the original entity sets.

The generalized entity set, which is at a higher level of abstraction, is also called the *supertype* and the original entity sets, at the lower abstraction level, are the *subtypes*. Viewing the construction from top-down the subtypes inherit the attributes of the supertype. Subtypes can be mutually exclusive (an entity can be a member only one of the subtypes) or overlapping (otherwise).

This process can be applied repeatedly in both directions, resulting a generalization hierarchy. Generalization can be expressed by a triangular element of the ER diagram via which the supertype is connected to the subtypes.

Example 1.12. Take the entity set of the vehicles registered in Austria, described by the attributes serial-number, registration-number, type, owner-ID (and perhaps also others). By attaching new attributes that describe important information only for special types of the vehicles we can get subtypes for

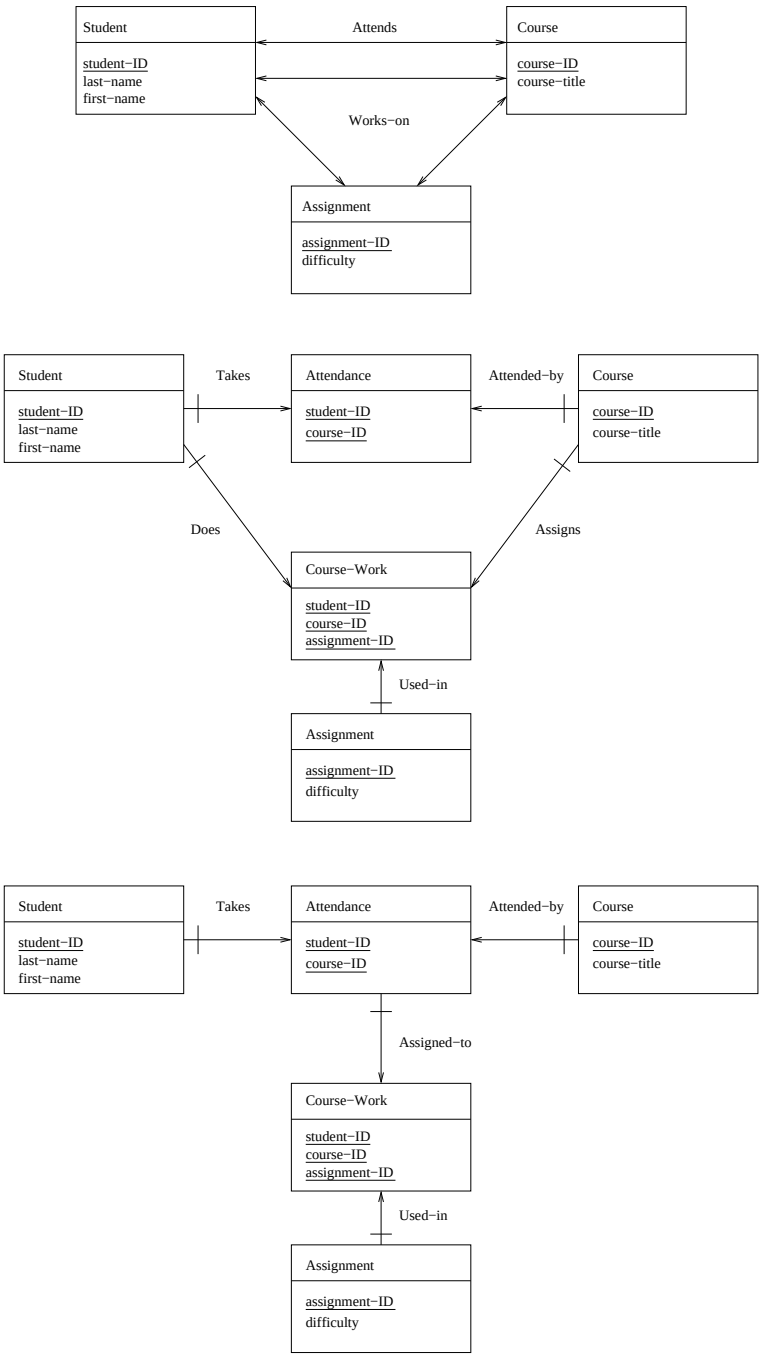


Figure 1.3: Resolving complex relationships and aggregating

personal cars (adding for instance the attributes: PS and color), trucks (adding maximal-weight and type-of-load), buses (adding nr-of-seats and comfort-level), etc.

In practice, there is the important phase of requirement analysis before the phase of data modeling. Very briefly, requirement analysis is used to achieve the following goals:

- Determine what data (viewed as elementary objects) is necessary to include in the database.
- Collect the descriptive information about these objects.
- Identify and classify relationships between these objects.
- *Identify what transactions will be executed on the database and how will they affect the data objects.*
- *Identify data integrity rules.*

The last two points concern already the functional view of the database.

Then in the data modeling phase, using the ER model, the tasks are the following:

- Identification of entity and relationship sets.
- Drafting the initial ER diagram.
- Refining the ER diagram by resolving complex relationships.
- Add key attributes to the diagram.
- Adding non-key attributes.
- Constructing generalization hierarchies.
- Adding integrity rules to the model.

The ER modeling method provides a wide variety of possibilities and there is no standardize way to achieve optimal designs. One has to balance between possibilities like to describe a class of objects by an entity set or via a relationship set, whether to allow weak entity sets or prefer strong ones, whether to apply generalization or aggregation or omit them to keep the ER model simpler, and so on.

1.2.6 Exercises

Since 1 October 2003 car dealers can offer models of competitors in the same exhibition area in the EU. A dealer is contracted with certain car manufacturers who supply him with certain models which are eventually bought by certain customers.

Taking this simplified model of car selling at a car dealer do the following.

Exercise 1.2.1. Identify the entities and the entity sets they belong in this model.

Exercise 1.2.2. Which attributes would you choose to describe entities in the entity sets you identified previously?

Of course the sets of attributes can be made quite big if one wants to model reality more and more accurately. Here it is enough to introduce a few attributes without the desire of completeness.

Exercise 1.2.3. Enumerate the relationships between the entities in this model.

Exercise 1.2.4. Sketch a first ER diagram of this model.

Exercise 1.2.5. Resolve complex relationships (relationships with degree > 2 and N:M relationships) and draw the refined ER diagram of the model.

Exercise 1.2.6. Decide which are the most important entities in the model and choose primary keys for them.

You might want to introduce new attributes at this step: remember that, for instance, student IDs are introduced artificially to describe students (as important entities) in the corresponding model.

Exercise 1.2.7. Prescribe integrity constraints for the models. (Which entity existence depends on which other? Are there foreign keys?)

1.3 The Relational Model

The relational model was formally introduced by E. F. Codd in 1970 and became the dominant modeling technique in the IT sector. The relational model represents data in the form of two-dimensional tables, and a relational database is a collection of such tables each having a unique name.

1.3.1 Relational Structure

Let D_i be *domains* (sets) for *attributes* (denoted by distinct names, say A_i) for $i = 1, \dots, n$. Then an n-ary *relation* over the D_i is a subset of their cartesian product:

$$R \subseteq D_1 \times \cdots \times D_n.$$

If R is a finite set, it can be described as a table with $m = \#R$ rows and n columns:

$$\begin{array}{ccc} d_{11} & \dots & d_{1n} \\ d_{21} & \dots & d_{2n} \\ \vdots & & \\ d_{m1} & \dots & d_{mn}. \end{array}$$

A row (or tuple) represents that the attribute values appearing in it are in relationship with each other by R. The attribute values are arranged in each row that the jth column contains the value fromD_j.

Remark 1.13. We typeset attribute and table names in typewriter style, with the first letter of table names capitalized.

Example 1.14. Relations represented by tables can be found in Table 1.1 on page 14. In the first separated row we find the attribute names; the tables themselves consist of the data below. The domain of `area-code`, `phone-number` is the natural numbers and the domain of the other attributes is the set of strings (of course, with more effort we could also find smaller domains).

From the viewpoint of the modeling process, we consider every attribute value in a table *atomic*. In other words the table is flat (the cells has no internal relational structure).

A *relational database* is a collection of tables with unique names.

Remark 1.15. From now on we will use the words "relation" and "table", just as "row" and "tuple", synonymously.

From the definitions it follows that a table has no two identical rows and the order of rows is unspecified. Moreover the order of columns can also be changed as long as each row contains the corresponding attribute values in the same column.

Atomicity is a relative notion. For instance virtually all relational database managers provide the domain of dates as a build in type, i.e. an atomic value. However, if we want, we can split it further to year, month and day components.

We make distinction between the overall design of the database, which is called the *database scheme*, and a snapshot of the database at a point in time, which is called a *database instance*. We also call a list of attributes with their corresponding domains a *relation scheme*.

The notions of *superkey*, *candidate key*, *primary key*, *foreign key* from Section 1.2 apply to the relational model. In context of relations a nonempty set K of attributes is a superkey for R if for any $r, s \in R$, $r \neq s$ implies $r|_K \neq s|_K$ (where $r|_K$ denotes the attribute values of r belonging to attributes in K).

Example 1.16. Continuing Example 1.14, let us call the first table `Phone-areas` and the second `Phone-numbers`. Then we get a relational database, which can play the role of a simple phone book. We fixed the order of columns (i.e. attributes) in a certain way, but actually any other order would do. There are no two identical rows in any of the tables due to the `area-code` attribute in `Phone-numbers`.

The set {`area-code`,`area-name`} is a superkey in `Phone-areas`, {`area-code`} is a candidate key, which is chosen to be the primary key for `Phone-areas`. Similarly, in `Phone-numbers` {`area-code`, `phone-number`} is chosen to be the primary key.

We can also notice that the `area-code` attribute appears in both of the tables. This way the two tables are related to each other in a One-to-Many relationship (`area-code` is a foreign key in `Phone-numbers`).

Finally, we can mention that sometimes it is unavoidable to "leave certain cells blank" in a table, simply because information might not be available there. This is done by introducing special *null values* in the domains of the attributes (we will denote them generally by NULL). However, this makes maintaining the integrity of the database more difficult, since in each transformation (updating and deleting rows, etc.) one has to check the special case of nullity. Moreover, since null values represent missing information comparisons involving them are always false.

It is also commonly required that attributes of primary keys are never null (entity integrity). And for foreign keys it is required that (in a row) either all attribute values in the key are null or they match a row in the table for which the foreign key is the primary key (referential integrity).

1.3.2 Relational Algebra

The relational algebra is a (procedural) query language that allows us to execute several kind of operations on a relational database. The fundamental operations are the following: select (unary), project (unary), rename (unary), cartesian product (binary), set theoretic union (binary), set theoretic difference (binary).

The set of additional operations are defined in terms of the fundamental ones are: set theoretic intersection (binary), natural join (binary), division (binary).

`Student`

sid	fname	lname
0251563	Werner	Schmidt
0245654	Andrea	Ritter
0245675	Daniela	Schmidt

`Attendance`

sid	cid
0251563	327456
0251563	327564
0245654	327456

`Course`

cid	title
327456	Analysis I
327564	Algebra I

Table 1.2: A small database of students and courses

The operations may have additional parameters beside their input arguments; they are put in the subscript of the notation. The definitions of the operations are as follows:

select The operations selects tuples of the relation that satisfy the predicate which is an additional parameter to it. It is denoted by σ. For example

$$\sigma_{\texttt{lname}=\text{"Schmidt"}}(\texttt{Student})$$

results the relation (represented in table form)

0251563	Werner	Schmidt
0245675	Daniela	Schmidt

.

The atomic predicates allowed in the parameter of σ are $=, \neq, <, \leq, <, \geq$, moreover with the logical connectives $\wedge, \vee$ compound formulas can be build.

project The operation projects the relation onto the set of specified arguments and additionally eliminates multiple rows. It is denoted by Π. For example

$$\Pi_{\texttt{fname},\texttt{lname}}(\texttt{Student})$$

results the relation

Werner	Schmidt
Andrea	Ritter
Daniela	Schmidt

while the operation

$$\Pi_{\texttt{lname}}(\texttt{Student})$$

results only

Schmidt
Ritter

.

cartesian product The operation combines each tuple from its first argument with each tuple from the second. The result is a relation whose attributes are the disjoint union of the attributes of the arguments. The number of rows of the cartesian product is the product of the numbers of rows of the arguments. It is denoted by $\times$. For example

$$\texttt{Student} \times \texttt{Course}$$

results the relation

0251563	Werner	Schmidt	327456	Analysis I
0251563	Werner	Schmidt	327564	Algebra I
0245654	Andrea	Ritter	327456	Analysis I
0245654	Andrea	Ritter	327564	Algebra I
0245675	Daniela	Schmidt	327456	Analysis I
0245675	Daniela	Schmidt	327564	Algebra I

.

Whenever ambiguity may arise we can prepend the relation name before attribute names to classify the intended attributes explicitly. For instance, if we had a `Teacher` relation, with attributes `fname` and `lname`, and we took `Student`$\times$`Teacher`, for the last names in the resulting table we would need to specify if we want to take `Student.lname` or `Teacher.lname`.

rename The operation just changes the name of its argument to the one given to it in its parameter. This is useful, for instance, when we need to take the cartesian product of a relation with itself. It is denoted by ρ. For example if we want to select the students with identical last names, we could use the following

$$\Pi_{\texttt{Student.fname},\texttt{Student.lname}}(\sigma_P(\texttt{Student} \times \rho_{\texttt{Student2}}(\texttt{Student})))$$

where P is the predicate

$$\texttt{Student.lname} = \texttt{Student2.lname} \wedge \texttt{Student.sid} \neq \texttt{Student2.sid}.$$

The result is the relation

Werner	Schmidt
Daniela	Schmidt

.

union The operation takes the set theoretic union of two compatible relations, where compatible means that the sets of attributes and the corresponding domains are the same. It is denoted by $\cup$. Let us define a new relation for teachers

`Teacher`

`fname`	`lname`
Wolfgang	Schiffer
Sabrina	Kaiser

and as an example let us take

$$\Pi_{\texttt{fname},\texttt{lname}}(\texttt{Student}) \cup \texttt{Teacher}$$

which results

Wolfgang	Schiffer
Werner	Schmidt
Sabrina	Kaiser
Daniela	Schmidt
Andrea	Ritter

.

difference The operation takes the set theoretic difference of two compatible relations. It is denoted by $-$.

intersection The operation takes the set theoretic intersection of two compatible relations. It is denoted by $\cap$ and it can be defined as $A \cap B = A - (A - B)$.

natural join The operation combines a cartesian product a selection and a projection operation into a single operation. This is justified by its frequent applications and that it can be more efficiently implemented. It is denoted by $\bowtie$. Let A and B be relations with attribute sets R and S respectively and let $\{r_1, \ldots, r_n\} = R \cap S$, then

$$A \bowtie B = \Pi_{R \cup S}(\sigma_{A.r_1 = B.r_1 \wedge \cdots \wedge A.r_n = B.r_n}(A \times B)).$$

If $R \cap S = \emptyset$ then $A \bowtie B = A \times B$; we also have $A \bowtie B = B \bowtie A$, and the operation binds from left to right.

For example $\texttt{Student} \bowtie \texttt{Attendance}$ results

sid	fname	lname	cid
0251563	Werner	Schmidt	327456
0251563	Werner	Schmidt	327564
0245654	Andrea	Ritter	327456

and thus $(\texttt{Student} \bowtie \texttt{Attendance}) \bowtie \texttt{Course}$ results

sid	fname	lname	cid	title
0251563	Werner	Schmidt	327456	Analysis I
0251563	Werner	Schmidt	327564	Algebra I
0245654	Andrea	Ritter	327456	Analysis I

which describes in a single relation which student takes what course.

θ-join This is a general version of the join operation in which the selection predicate can be any valid formula on the attributes of the operands. In this case, without loss of generality, we can assume that R and S are disjoint (we can always achieve this formally, using the rename operation). Then the θ-join of A and B over the formula

$$\theta = A.r_1 \ \theta_1 \ B.s_1 \wedge \cdots \wedge A.r_n \ \theta_n \ B.s_n$$

is

$$A \bowtie_\theta B = \sigma_\theta(A \times B),$$

where $r_1, \ldots, r_n \in R$, $s_1, \ldots, s_n \in S$ are such that they are valid input arguments for the predicates $\theta_1, \ldots, \theta_n$.

division The division operation requires that the set of attributes of the divisor is a subset of the set of attributes of the dividend. Then it leaves those tuples of the dividend in the result (projected to the difference attribute set) which have an occurrence for each tuple in the divisor over the common attributes. It is denoted by $\div$. Let A and B be relations with attribute sets R and S respectively and let $S \subseteq R$, then

$$A \div B = \Pi_{R-S}(A) - \Pi_{R-S}((\Pi_{R-S}(A) \times B) - A).$$

For example $((\texttt{Student} \bowtie \texttt{Attendance}) \bowtie \texttt{Course}) \div \texttt{Course}$ results

0251563	Werner	Schmidt

.

The relation of the dividend can be found above with all the five attributes. The common attributes of this relation and `Course` is `cid` and `title`. The projection results only one tuple which comes from many tuples in the dividend whose `cid` and `title` attributes matched every tuple in the divisor. In still simpler words there is only one student who takes all the courses which can be found in the database.

assignment It is denoted by $\leftarrow$, and works as the assignment operator in any imperative programming languages. If a name of a relation of the database stands on the left hand side of the operator, the statement results the modification of the database. On the other hand, if a new name gets assigned to a relation, it does not mean to introduce a new relation into the database. The new name will act only as a temporary variable which can be used to simplify expressions in subsequent statements. For example the division operation could be rewritten as $tmp_1 \leftarrow \Pi_{R-S}(A)$, $tmp_2 \leftarrow \Pi_{R-S}((\Pi_{R-S}(A) \times B) - A)$ and $A \div B = tmp_1 - tmp_2$.

There are also the following extended operations defined:

generalized projection This operation is analogous to projection with the additional feature that arithmetic expressions formed by the attributes and constants from the corresponding domains are allowed as parameters for the operation.

For example, let us assume that we have generated a relation `Analysis-result` (via the operations above) for students that contain the attributes `sid`, `pts1`, `pts2`, `pts3`, where the pts'es are the points the students reached on the first, second and third exam of the Analysis I course. Then the generalized projection

$$\Pi_{\texttt{sid},(\texttt{pts1}+\texttt{pts2}+\texttt{pts3})/3}(\texttt{Analysis-result})$$

will describe the average score of the students. With a more complicated expression one might even define the grade immediately.

inner join The joins we discussed so far are inner joins. The "inner" qualification is often skipped because a join is an inner-join by default. An inner join, opposed to an outer-join, includes only those rows of the operands in the result that satisfied the join condition. For instance, in the `Student` $\bowtie$ `Attendance` operation the student "Daniela Schmidt" does not appear in the result, because there is no matching row for her tuple in the `Attendance` table.

outer join This operation comes in three variants (listed below). The purpose of the operation is to include in the result not only those rows of the left/right/both operands that satisfy the join condition, but also ones that do not have matching pairs. In this sense an outer join is an extension of the join operation which guarantees that all rows of the chosen operands will appear in the result (possibly with NULL values padded for the attributes coming from the other operand).

- *left outer-join* the result will contain also those tuples from the first (left) argument of the operation that did not match any tuple from the second (right) argument, and pads the missing attributes with null values.
- *right outer-join* analogously to the previous case, the tuples of the second (right) argument are carried over and padded with null values as necessary.
- *full outer-join* this operation, analogously to the previous cases, carries over every tuples from its arguments and pads them with null values ("on the left" or "on the right") as necessary.

For instance in the example for join the tuple (0245675, Daniela, Schmidt) could not be matched with any tuples from the other relation; therefore it does not appear in the result. A left outer-join (`Student` $\bowtie$ `Attendance`) $\bowtie$ `Course` would result

sid	fname	lname	cid	title
0251563	Werner	Schmidt	327456	Analysis I
0251563	Werner	Schmidt	327564	Algebra I
0245654	Andrea	Ritter	327456	Analysis I
0245675	Daniela	Schmidt	NULL	NULL

self join A join operation is called self join if both operands are the same (they can be made formally different by a rename operation). For instance, let us consider the relation `People` with relation scheme `id, fname, lname, boss-id`, where the `id` is an internal identifier of the people of the company and `boss_id` is the identifier of the boss of the employee (for the top ranking people the `boss_id` is NULL). Then the following self join produces the table in which a row contains an employee and his boss:

$$\texttt{People} \bowtie_{\texttt{People.boss\_id=Peopel2.id}} \rho_{\texttt{People2}}(\texttt{People}).$$

To obtain accumulative information from the database we can use the notion of *aggregation operations*. In general such an operation has the following form

$$_{G_1,\ldots,G_n}\mathcal{G}_{F_1A_1,\ldots,F_mA_m}(E),$$

where E is a relational algebra expression, the G_i are attributes to group on, and F_i is an aggregate function (sum, avg, count, min, max) to be applied on attribute A_i.

The semantic of the operation is the following. First the tuples in E are grouped by the G_i, such that in each group the tuples have the same G_i-value for each i and tuples in different groups have at least one difference in the G_i-values. A tuple of the resulting relation is $(g_1, \ldots, g_n, a_1, \ldots, a_m)$, where the $g_1, \ldots, g_n$ identifies a group and $a_1, \ldots, a_m$ are the results of the corresponding aggregate functions on the attribute values of the tuples in the group.

Example 1.17. Let us consider a relation named `Credits`,

`Credits`

sid	cid	credit
0251563	327456	3
0251563	327564	2
0245654	327456	3

where `credit` gives the number of credits the student with `sid` obtained for the course with `cid`. Then the aggregation operation

$$_{\texttt{sid}}\text{Creditsum}_{\textbf{sum}\ \texttt{credit}}(\texttt{Credits})$$

results the relation

0251563	5
0245654	3

.

It describes the total number of credits each student has.

The following operations are provided for database modification:

delete Let E be a relational algebra expression resulting a compatible relation to a relation R. Then the deletion of E from R is just defined as

$$R \leftarrow R - E.$$

insert Let E be a relational algebra expression or an explicitly specified relation resulting a compatible relation to a relation R. The insertion of E into R is defined by taking union

$$R \leftarrow R \cup E.$$

For example, to insert a new student in the `Student` relation, we could do

$$\texttt{Student} \leftarrow \texttt{Student} \cup \{(0246534, \text{Markus, Klein})\}.$$

update Update can change values inside a tuple without deleting, modifying and inserting again. It is denoted by δ and a general update is of the form

$$\delta_{A \leftarrow E}(R),$$

where R is a relation, A is an attribute and E is an arithmetic expression with constants and attributes in R, resulting a value in the domain of A.

If one wants to apply the update only on a subset of tuples fulfilling the predicate P, one can use $\delta_{A \leftarrow E}(\sigma_P(R))$.

Let us take the example we had for the generalized projection, and let us extend the relation with a new attribute `total`. Initially we set all test-points to zero and whenever a test takes place and new points are inserted in the database, we update the `Analysis-result` relation by

$$\delta_{\texttt{total} \leftarrow \texttt{pts1+pts2+pts3}}(\texttt{Analysis-result}).$$

Finally, we say a few words about views. It is advantageous if the relational database can be "viewed" independently from the structure it possesses (for instance from the point of view of security or to obtain simpler database schemes for certain target user groups). This can be done by declaring certain relations, which need not be members of the logical design of the database, in terms of the ones we already have defined. Then we work with these relations as if they were in the database, resolving every operation to the relations of the database via the view definitions.

A *view* is defined via a new command

$$\texttt{create view}\ V\ \texttt{as}\ Q$$

where V is a new name (identifier) for the view to be created and Q is a query of the relational algebra. After the definition we allow V to appear anywhere in a relational algebra statement where a relation of the database can appear.

Please note the following caveats:

- Whenever a view is used to modify the database the resolution of the operation may require extensive usage of NULL values. For instance if the view is defined via projection from a relation and a new tuple is inserted into the database via the view, the attributes which got eliminated in the projection can by default only have null values.
- As views only represent relations that are derived from the relations of the database via queries, they have to be recomputed after each modification of the database. This can be problematic if, for instance, null values were introduced for attributes with respect to which relations were joined to create the view (null values result false for comparisons).
- From the definition it also follows that views can be used to define new views, hence resolution of views can require several steps.

Remark 1.18. The relational calculus is another (nonprocedural) query language, in which a query is expressed as a set of tuples for which a given predicate is true. Its restriction to a subset of expressions, which are called the safe expressions of the language, is equivalent in expressive power to the relational algebra. The relational calculus is not covered in this document.

1.3.3 Functional Dependencies

Integrity constraints are used to express conditions the database have to fulfill in order the data it holds to be consistent. However complex constraints can

drastically set back performance, because they have to be checked for each modification of the database.

We discuss briefly: domain constraints, referential integrity, assertions, triggers and functional dependencies.

Domain constraints restrict the possible values of attributes. They are usually easy to check and thus provided by many DBMSes. A simple example is to specify the attribute `credit` in the `Credits` relation to be of numeric and take values from the range from 0 to 5.

Let A and B be relations with describing attribute sets R and S respectively. Then $P \subseteq R$ is a *foreign key* for a candidate key $L \subseteq S$ of B if for every tuple t in A there is a tuple u in B such that $\Pi_P t = \Pi_L u$. Such a condition is a *referential integrity* constraint. It can also be expressed as $\Pi_P A \subseteq \Pi_L B$.

Example 1.19. For example, take the relations `Student` and `Attendance` from Section 1.3.2. In `Attendance` the attribute `sid` is a foreign key for the primary key `sid` of `Student`. Back in Example 1.11 we expressed the referential integrity in the ER diagram by a line crossing the arrow from the Student entity set to the Attendance one.

Modifications in the database can destroy referential integrity; therefore, using the previous notation we must have

- when a tuple t is inserted into A, there must already be a tuple u in B with $\Pi_P t = \Pi_L u$,
- when a tuple u is deleted from B and there is still a tuple t in A with $\Pi_P t = \Pi_L u$, then either abort the deletion or delete also t from A (depending on the application),
- when tuples in A are updated analogous actions has to be taken as in the case of insertion and when updating B as in the case of deletion (to maintain referential integrity in both cases).

An *assertion* is a predicate the database should always satisfy. The previous integrity constraints are special cases of assertion. Using the university database example, an assertion could be to require that no teacher has more than five courses in a semester.

A *trigger* is a statement (in the query language) the DBMS executes automatically whenever a set of conditions becomes true. For instance, deletion of a tuple from the `Student` relation could trigger deletion of referring tuples from the `Attendance` relation.

Functional dependencies

Functional dependencies represent integrity constraints on sets of attributes in the database. They are in strong correspondence with causal-, temporal-, etc. dependencies in the real world; therefore finding them out is an important part in the modeling process.

Let R be a set of attributes and let P and S be subsets of R. Then S *functionally depends* on P, denoted as $P \to S$, if for any legal relation A with R being its set of describing attributes, for any tuples t and u, $\Pi_P t = \Pi_P u$ implies $\Pi_S t = \Pi_S u$.

In the definition above "legal" means any relation that describes the real world situation we are modeling. Viewing it the other way around, prescribing functional dependencies for the relations of the database means constraining the set of legal relations, that is, functional dependencies are a tool to describe what relations are legal.

We say that S *irreducibly (or fully) functionally depends* on P if $P \to S$ and we do not have $Q \to S$ for any proper subset of $Q \subset P$. We call $P \to S$ *irreducible* if $Q \not\to S$ for every $Q \subset P$.

Remark 1.20. We can see that this concepts generalizes the concept of superkeys: the fact that K is a superkey can be expressed as $K \to R$.

Example 1.21. Consider the relation (`Student` $\bowtie$ `Attendance`) $\bowtie$ `Course`, shown on page 22. In this relation {`sid`} $\to$ {`fname`, `lname` }, {`cid`} $\to$ {`title`}, and the key candidate is {`sid`, `cid`} from which the set of all attributes functionally depends.

A functional dependency is called *trivial* if it is satisfied by all relations.

Armstrong's axioms for functional dependencies say that for $P, Q, S, T \subseteq R$ we have

- $Q \subseteq P$ implies $P \to Q$ (reflexivity),
- $P \to Q$ implies $P \cup S \to Q \cup S$ (augmentation),
- $P \to Q$ and $Q \to S$ implies $P \to S$ (transitivity).

These axioms are sound (consistent) and complete.

Some additional rules derived from the axioms are:

- $P \to P$ (self-determination),
- $P \to Q \cup S$ implies $P \to Q$ and $P \to S$ (decomposition),
- $P \to Q$ and $P \to S$ implies $P \to Q \cup S$ (union),
- $P \to Q$ and $S \to T$ implies $P \cup S \to Q \cup T$ (composition),
- $P \to Q$ and $Q \cup S \to T$ implies $P \cup S \to T$ (pseudotransitivity).

A functional dependence $P \to S$ is called *transitive* (via Q) if there exist functional dependencies $P \to Q$ and $Q \to S$ with $S \not\to Q$ and $Q \not\to P$.

If F is a set of functional dependencies the set of all dependencies it implies is called the *closure of* F, denoted with F^+.

The set of attributes of R determined by $P \subseteq R$ under F is called the *closure of* P, denoted as P^+

Let F and G be two sets of functional dependencies, if $F^+ \subseteq G^+$ then we say that G *covers* F. If F and G mutually cover each other, they are called *equivalent*.

A set of dependencies is called *irreducible* if

- the right hand side of every functional dependency in F is a singleton,
- no attribute can be discarded from the left hand side of any functional dependency in F without changing F^+,

- no functional dependency in F can be discarded without changing F^+.

Example 1.22. Let F be the set of three functional dependencies in Example 1.21. A few functional dependencies implied by F are

{sid}	→	{fname,lname}	F_1
{cid}	→	{title}	F_2
{sid,cid}	→	{sid,fname,lname,cid,title}	F_3
{sid}	→	{fname}	// by decomposition
{sid}	→	{lname}	// by decomposition
{sid,cid}	→	{sid,cid}	// by reflexivity
{sid,cid}	→	{fname,lname,title}	// by composition.

The closure of {sid,cid} under F is {sid,fname,lname,cid,title}, as can easily be seen from F_3. We can also easily obtain that $\{F_1\}$ is covered by F and that $\{F_1, F_2\}$ is equivalent to F (F_3 can be obtained from the last two rows from above and they are obtained from F_1 and F_2 alone). We can also construct an equivalent irreducible set of functional dependencies from the above ones:

{sid}	→	{fname}
{sid}	→	{lname}
{cid}	→	{title}.

1.3.4 Normal forms

Normalization provides means to reduce data redundancy in the database, and to make data retrieval more efficient. In the process of normalization we decompose relation schemes in a way that the resulting schemes satisfy certain functional dependency constraints and at the same time allow us to reconstruct relations on the original scheme without data loss.

Let R be a relation scheme (a set of attributes), a set of relation schemes $R_1, \ldots, R_n$ is a *decomposition* of R if $R = R_1 \cup \cdots \cup R_n$. We call $R_1, \ldots, R_n$ a *lossless join decomposition* of R if for all legal relation A on R we have

$$A = \Pi_{R_1}(A) \bowtie \cdots \bowtie \Pi_{R_n}(A).$$

Remark 1.23. Please note that in the previous situation we always have $A \subseteq \Pi_{R_1}(A) \bowtie \cdots \bowtie \Pi_{R_n}(A)$.

(Heath's theorem) Let F be a set of functional dependencies for a relation schema R (describing legal relations). A decomposition $P \cup S = R$ is a lossless join decomposition if $P \cap S \to P$ or $P \cap S \to S$ is in F^+.

Let $R_1, \ldots, R_n$ and F be as above. The *restriction* of F to R_i, denoted as F_i, is the set of all functional dependencies of F^+ that contain only attributes in R_i. If $F^+ = (F_1 \cup \cdots \cup F_n)^+$, we call $R_1, \ldots, R_n$ a *dependency preserving decomposition* of R.

The point in dependency preserving decompositions is to be able to check data integrity by checking whether the modified tuples fulfill functional dependencies in the relations of the schemes R_i, instead of testing also interrelational dependencies in the database.

Example 1.24. For the relation scheme {sid,fname,lname,cid,title} the decomposition {sid,fname,lname}, {sid,cid,title} is a lossless join decomposition, where {sid,cid}, {cid,title} is a lossless join decomposition for

the latter. This can be easily checked via functional dependencies. Moreover from the results of Example 1.22, we get that it is also a dependency preserving decomposition.

A relation scheme is in *first normal form (1NF)*, if all the attribute values in it are atomic (by definition there cannot be two identical tuples in the relation).

For defining 2NF and 3NF we assume that the relation scheme has only one candidate key which is the primary key. An attribute not member of any candidate key is referred to as *non-key attribute*.

A relation scheme is in *second normal form (2NF)* (with respect to a set of functional dependencies F), if it is 1NF and every non-key attribute irreducibly functionally depends on the primary key.

A relation scheme is in *third normal form (3NF)* (with respect to a set of functional dependencies F), if it is 2NF and every non-key attribute non-transitively depends on the primary key.

A relation scheme is in *Boyce-Codd normal form (BCNF)* (with respect to a set of functional dependencies F), if the left hand side of every nontrivial irreducible functional dependency is a candidate key.

Example 1.25. Consider first the usual example from page 22 with relation scheme {`sid, fname, lname, cid, title`}. The irreducible set of functional dependencies under consideration is the one at the end of Example 1.22. The only candidate key is the primary key {`sid, cid`}.

Before further discussion, please also note that now we are concerned with relation schemes and the properties have to be fulfilled by any legal relation on them. Therefore the table on page 22 is just a particular example.

The relation scheme is in 1NF, but not in 2NF because, for instance {`fname`} functionally depends on {`sid`} which is a proper subset of the primary key.

Let us consider then the relation scheme {`sid, pts1, pts2, grade`} of a teacher giving only one course, which describes that on the course a student achieved so and so many points on the tests and finally obtained a grade. The primary key is {`sid`}. This scheme is in 2NF, but not in 3NF because of {`pts1, pts2`} $\rightarrow$ {`grade`}.

The main idea of 3NF and BCNF, informally speaking, is that when one considers the graph of nontrivial functional dependencies the arrows should point from candidate keys to non-key attribute sets. The difference is that 3NF does not fulfill this requirement when there are more than one composite candidate keys and they overlap. If we drop the requirement that the relation scheme has only one candidate key we can see this difference via a somewhat artificial example.

Let us assume that course names also include some kind of unique identifier, so that in `Course` the attribute `title` is also a candidate key. Then in the relation scheme {`sid, cid, title, grade`} describing that a student on a course given with its ID and name got some grade. This relation might be created to list grades publicly, so that no names of students should appear, however the name of the course might be more informative than just its ID. The candidate keys are {`sid, cid`}, {`sid, title`} (let the first be the primary key) and we have the following functional dependencies:

$$
\begin{array}{lcl}
\{\texttt{sid,cid}\} & \rightarrow & \{\texttt{title}\} \\
\{\texttt{sid,cid}\} & \rightarrow & \{\texttt{grade}\} \\
\{\texttt{sid,title}\} & \rightarrow & \{\texttt{cid}\} \\
\{\texttt{sid,title}\} & \rightarrow & \{\texttt{grade}\} \\
\{\texttt{cid}\} & \rightarrow & \{\texttt{title}\} \\
\{\texttt{title}\} & \rightarrow & \{\texttt{cid}\}.
\end{array}
$$

Allowing more than one candidate keys, the relation scheme with this set of functional dependencies is 3NF (the only non-key attribute is `grade` which irreducibly and non-transitively depends on the primary key). But it is not BCNF because the left hand sides of the nontrivial irreducible functional dependencies $\{\texttt{cid}\} \rightarrow \{\texttt{title}\}$ and $\{\texttt{title}\} \rightarrow \{\texttt{cid}\}$ are not candidate keys.

We could decompose this relation scheme—as {`sid, cid, grade`}, {`cid, title`}—to get the resulting relation schemes in BCNF. (End of Example 1.25)

As we saw, BCNF is more rigorous then 3NF so it might not always be the optimal solution to try to achieve it. Therefore the goals of good relational database design is to achieve (by decomposition) that the relation schemes of the database scheme are in BCNF, are lossless join decompositions and are dependency preserving. If this is cannot be achieved, 3NF is also acceptable instead of BCNF.

Remark 1.26. There are also higher normal forms denoted by 4NF and 5NF. We do not cover them in this discussion.

1.3.5 Indexing and Hashing

In this section we give a brief overview on some of the issues that are important from the practical point of view. They are concerned with efficient information retrieval from the database by using files with special structure to accelerate search in the tables. In this section, we will talk about files and records, instead of tables and tuples, to indicate that the data structures are considered on the physical level.

Indices

The time it requires to process a query can become very costly as the files of the database grow. To speed up the retrieval process DBMSes provide data structures, called index or hash files, for several kind of fast search algorithms. Indices accelerate queries involving attributes whose domains are ordered, while hashes map any kind of attribute to an address space (e.g. numbers from 0 to 255) allowing us to split up the search for the records with the requested attribute values.

The techniques can be evaluated from several viewpoints: types of access supported efficiently (search for specific values, value ranges, etc.), access time, insert/delete/update times, space overhead.

The set of attributes on which an index (or hash) file is created is called the *search key* for the index (or hash). Relations of the database can have multiple indices and hashes assigned to them.

Given a file of the database whose records are ordered sequentially according to a set of attributes. The index whose search key is this set of attributes is called the *primary index*. In this case the indexed file is also said to be in *index*

sequential order with respect to this index. Such a search key is usually (but not necessarily) the primary key of the relation stored in the file.

An index is called *dense* if an index record appears (in the index file) for every search key value in the indexed file. Otherwise, when only some of the search key values appear in the index file, the index is called *sparse*.

For sparse indices we first locate the search key value in the index file which is not greater than the search key value we are looking for. This provides a pointer to the indexed file and have to continue the search in the indexed file from the pointed record on. This is slower as if we had a dense index, however it can also require a lot less space for the index file.

Another solution to cope with the index size problem is to use *multi-level indices*. In this approach the outer index file is a sparse index for the search key that indexes not the data file but another dense or sparse index on the same search key. There can be a hierarchy of indices in which the lowest will index the data file.

It is also advantageous to fit indices with physical storage units (i.e. disks, cylinders, blocks), such that one can store the outermost index file in memory and the index hierarchy will guide the DBMS to the right disk, cylinder and the appropriate block on it which should be read in.

Insertion and deletion requires updating the affected index files. If we delete a record from a data file, we have to check for each index file (on this data file) whether the deleted record was the last with the given search key value. If yes the index record for that value has to be deleted from the index file. For sparse indices deleting a search key value from the index file can be substituted with updating it to the next larger value in the data file, if this larger value is not already in the index file. If such an update is not possible, the search key value must be deleted from the index file.

When a new record is inserted in the data file, for dense indices we have to insert the new search key value in those index files where the it is not already included. For sparse indices the index file has to be updated only if a new block is created with the insertion of the new record; i.e. this first record of the new block will be indexed. (Remark: it depends on the indexing policy what is considered a new block: e.g. a block can consist of all the records in the table for which the `lname` attribute starts with the same letter).

Indices that define not the sequential order of the file are called *secondary indices*. Such indices have to be dense, because records of the indexed file are not sorted by the search key. If the search key is not a candidate key in the indexed file, the index file can be organized as a two level index. We search for the given search key value on the outer level, which points to the appropriate part of the inner level, that stores pointers to the records of the data file with the given search key value.

B-trees

There are several data structures used in index files each suitable for different kind of search methods. The most popular ones are B- and B^+-trees. The most important feature of B-trees is that they are always balanced, thus the depth of the tree grows only logarithmically in the growth of the indexed records.

To construct a B-tree we fix an even integer n and require that

- the root stores minimum 1 and maximum n key values,

Dense primary index

Search key: `sid`	Pointer
0245654	1
0251563	2
0252375	4
0252483	5

Dense secondary index

Search key: {`sid`,`cid`}		Pointer
0245654	327456	1
0251563	327456	3
0251563	327564	2
0252375	327456	4
0252483	327564	5

Indexed file (sorted by the attribute `sid`)

Address	`sid`	`fname`	`lname`	`cid`	`title`
1	0245654	Andrea	Ritter	327456	Analysis I
2	0251563	Werner	Schmidt	327564	Algebra I
3	0251563	Werner	Schmidt	327456	Analysis I
4	0252375	Stefan	Braun	327456	Analysis I
5	0252483	Franz	Lander	327564	Algebra I

Secondary index (outer)

Search key: `cid`	Pointer
327456	a
327564	d

Secondary index (inner)

Address	Pointer
a	1
b	3
c	4
d	2
e	5

Figure 1.4: Indexing example

- each node of the tree, with the exception of the root, stores minimum $n/2$ and maximum n key values,
- leaf nodes are on the same level.

Between search key values in a node pointers to subtrees are placed. The subtrees have the property that each value stored in them lie between the two values neighboring the pointer that links the subtree to the node. (Of course, to each search key value belongs a bucket of pointers to the indexed file, addressing the records that have the given search key value. We do not explicitly display these pointers in the examples.) An example can be found in Figure 1.5 with $n = 4$.

Insertion into a B-tree occurs at leaf nodes. If there is enough space on the node—as in the case of inserting 3 in the example of Figure 1.5—we just add the value to its appropriate place. If the new element causes a node overflow (the number of key values become greater than n on the node)—as in the case of inserting 17 in the example of Figure 1.5—we split the node into two, taking out the middle element and inserting it into the parent. We also update the neighboring pointers so that they refer to the subtrees we just created by the splitting. If the parent also overflows, we continue the spitting and propagating till we get a node with at most n elements. In the worst case we have to create a new root.

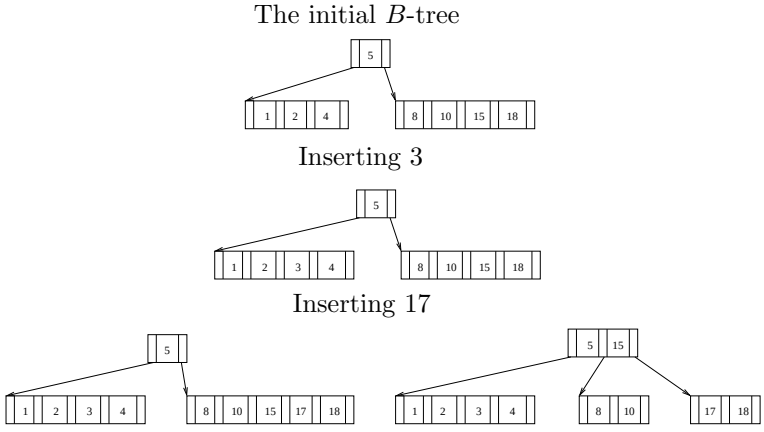
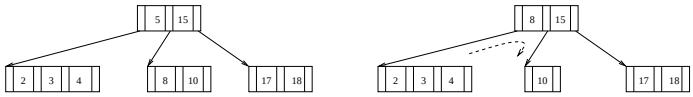


Figure 1.5: Insertion into a B-tree ($n = 4$)

Deletion from a B-tree requires first to locate the key, say K, to be deleted. We delete K from the tree, and if it was not in a leaf we shift up the first (leftmost) element of the child node of the pointer on the right of the deleted value. If the child is also not a leaf we use the same strategy to fill the place of the search key value which was shifted up to the parent. Continuing like this we achieve that eventually there will be one less search key values in a leaf node of the tree (see the deletion of 5 in the example of Figure 1.6).

If this causes an node underflow (the number of key values become less than $n/2$ on the node) we need to redistribute elements. If we can pull over elements from a sibling through the parent node we do it (as in the deletion of 5 in the example of Figure 1.6). Otherwise, we must have that the underflow-node with

the sibling has less than n elements altogether. Then we unify the two nodes and the element in between from the parent node to form a single new node, and discard the taken element of the parent (see the deletion of 8 in the example of Figure 1.6). If this action causes node underflow in the parent, we have to try to pull over element from siblings or if it does not work, unify the node with a sibling. Of course this can cause node underflow in the parent of the parent; therefore the process can propagate up to the root of the tree, and if it happens to become empty after a unification step, we just drop it and the new root will be the unified node.



Deleting 5 and pulling over elements from the left sibling via the parent

Deleting 8 and unifying 4 with the first two siblings

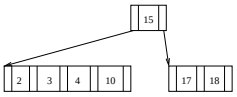


Figure 1.6: Deletion from a B-tree ($n = 4$)

A B^+-tree is a B-tree in which key values (with pointers to the record or to a bucket of pointers to records in the indexed file with this search key value) are stored only at the leaf nodes. Non-leaf nodes store separator search key values between the pointers, such that a configuration P_1, K, P_2 has the property that in the subtree pointed at by P_1 there are only search key values $\leq K$ and in the subtree pointed at by P_2 all search key values are $> K$.

Hashes

Indices are useful in queries for ranges of search key values, when after the start of the range is found the index can be traversed to retrieve consecutive records. Hashing, on the other hand, is more useful in accessing particular records.

In hashing we prescribe an address space to which search key values have to be mapped. A *hash function* h maps from the domain of search key values to the address space we fixed. A hash function is by no means injective; at each address we have to provide storage for a list of pointers (identifying the records whose search key values hash to this address). Such a storage is called a *bucket*.

To search records with a given search key K, we look through the records identified by the pointers in the bucket with address $h(K)$. Insertion and deletion is easy: compute the hash value for the modified record and insert/delete [a new/the old] pointer in/from the bucket for the obtained hash value.

The problem is how to define the function h for a generic domain so that it can be quickly computed and it distributes the search key values as uniformly as

possible between the addresses. If a hash function satisfies the latter requirement we can choose the size of the address space and of the buckets according to the expected number of records. However, bucket overflows are usually unavoidable in practice.

Dynamic hashing copes with the problems of bucket overflow by allowing dynamic splitting and merging of the buckets in which the pointers are kept according to the fluctuation of the size of the hashed file.

Extendible hashing is a specific type of dynamic hashing which results fairly low performance overhead. The idea behind is similar to the one of B-trees. Let the function h map to b-bit unsigned binary integers (e.g. $b = 32$). We use only the i initial bits of the hash values to determine the bucket address, where i can initially be 1 and it increases/decreases with the size of the database.

The i-bit hash values are listed in the *bucket address table* with pointers to buckets, i.e. for each hash value the pointer shows us the bucket in which we can find the pointers to the records whose search key values hash to the given hash value. The bucket with index j is addressed by the first i_j bits of the first i bits of the hash values (clearly $i_j \leq i$). This means that 2^{i-i_j} rows of the bucket address table points to the jth bucket (the bucket address table has 2^i rows altogether). The i_j numbers are stored in the buckets.

Locating records with given search keys should be clear.

Inserting a pointer to a record with a new key value K into the hashing scheme requires first to compute $h(K)$ and determine from the first i bits the bucket the pointer goes to; let it be of index j. If the bucket is not yet full, we insert the pointer into it.

If it is full, and $i_j < i$, there are at least two entries in the bucket address table pointing to j. We create a new bucket, say k, and set $i_k = i_j + 1$ and increase i_j by one. We distribute the initial segments of hash values that were formerly mapped to j equally between j and k (e.g. it maps to j if the last bit is 0 and maps to k otherwise). Then we rehash all records that belonged to bucket j and redistribute them in the same way between j and k.

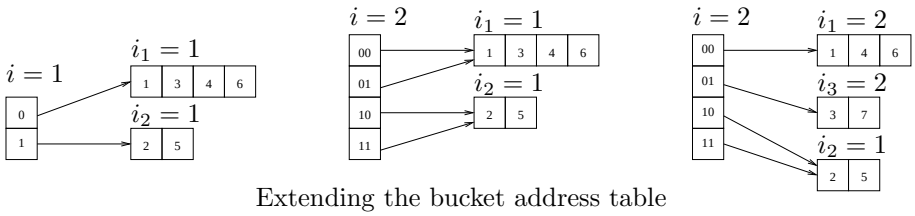
There is a small chance that all of them will go into one of the buckets (i.e. the last bit of the first i_k-bit segment of the hash values is the same for all the records). In this case we have to increase number of bits, split and rehash again.

If the bucket is full and $i_j = i$ we have to increase i and i_j by one. This means that the size of the bucket address table doubles. We set the pointers of each row of the new table to the bucket to which the row identified by the first $i - 1$ bits pointed in the old table (i is considered with its new increased value here). Then we have that two entries point to each bucket, in particular to the jth one, and we can act as we did in the previous case (see Figure 1.7).

Deletion is done similarly by applying unifying strategies to merge buckets with only a few pointers in them. The strategy can be taken from B-trees.

For a final word on indexing and hashing we can remark that the decision on which one to use for which search keys has to regard:

- If the periodic re-organization of index and hash file structures pays off.
- What is the relative frequency of insertion and deletion?
- Do we want better average access times or worst case access times?
- The type of queries we expect on the database.



Extending the bucket address table

Inserting record 7 with $h(K_7) = 01$...

Splitting up bucket 1, redistributing and inserting 7

$h(K_1) = 00$, $h(K_2) = 10$, $h(K_3) = 01$, $h(K_4) = 00$, $h(K_5) = 11$, $h(K_6) = 00$

Figure 1.7: Insertion into an extendible hash scheme (max bucket size is 4)

The last two items can decide whether we should use an index or a hash. Indices usually does not provide as good average access times as hashes for a particular search key, on the other hand they are better in worst case timings. Also if the queries are asking for whole range of values, indices are usually a better choice.

1.3.6 Exercises

Let us move on from selling cars (Section 1.2.6) to maintaining the administration of already sold cars. Let us assume that the simplified database of the police contains the following tables: `Person, Car, Category, Penalty, Speeding`. Let the relation schemes be the following.

For `Person`: `id` (9 digit number), `fname,lname,address` (character data storing the first and last names and the adresses respectively).

For `Car`: `rnr` (character data; the registration number), `id` (the ID of the owner), `model` (character data identifying the model of the car).

For `Category`: `code` (character data identifying the penalty category), `fee` (a number giving the amount in EUR to be payed in the corresponding penalty dategory).

For `Penalty`: `slimit` (a number describing a speed limit), `speed` (a number with `slimit` < `speed`, describing the minimum speed value of the penalty zone), `code` (character data identifying the penalty category).

For `Speeding`: `slimit` (a number giving the speed limit on the road where the radar is installed), `rnr` (the registration number of a car), `speed` the speed value of the car. For example:

Person

id	fname	lname	address
129432374	Michaela	Schumacher	4040 Linz, Freistaedter Str. 45
123695034	Gerhold	Berger	1010 Wien, Altstrasse 12
132487549	Hans	Brecht	6020 Innbruck, Technikerstr. 36

Car

rnr	id	model
L 3452 AS	129432374	VW Passat
W 2336 KL	123695034	Mercedes C220
FR 543 HS	143547895	Nissan Micra
I 5436 S	132487549	BMW 530D

Category

code	fee
S1	30
S2	50
S3	100
S4	500

Penalty

slimit	speed	code
50	55	S1
50	60	S2
50	80	S4
80	85	S1
80	100	S3
80	120	S4

Speeding

slimit	rnr	speed
50	L 3452 AS	80
50	I 5436 S	72
80	W 2336 KL	109
80	L 3452 AS	93
50	W 2336 KL	54
80	W 2336 KL	115

Let us extend the list of predicates which can be used to describe conditions with the pattern matching operator 'LIKE(pattern, arg)', where 'pattern' is a string value in which '%' will match any sequence of characters and '?' will match any single character. The predicate is true if the argument string 'arg' matches the pattern.

Determine relational algebra expressions that carry out the following computations.

Exercise 1.3.1. Compute another pernalty table where instead of the code of the penalty category the fee appears.

Exercise 1.3.2. Compute the table of first and last names of people who own a Mercedes.

Exercise 1.3.3. Compute the table of registration numbers and models of cars in the area of Linz.

Exercise 1.3.4. Compute the table of first and last names, addresses, speed limits and speeding values from the data captured by the radar.

Exercise 1.3.5. Compute the table of names of people who were captured by a radar violating every speed limit value in the `Penalty` table (i.e. both 50 and 80).

Let us assume that the penalty system is accumulative, so that a driver has to pay the sum of the penalty fees of the categories his speeding falls into. For example, driving with 85 on a road with speed limit 50 falls into the S1, S2, S4 categories (because all the three rows of the `Penalty` table for limit 50 applies); thus the driver has to pay $30 + 50 + 500 = 580$ EUR.

Exercise 1.3.6. Let v denote a given speed. Compute a penalty table for v which contains the speed limits and the accumulated sum of fees one has to pay if one drives with speed v under the given speed limit.

Exercise 1.3.7. Compute the table of first and last names, addresses, registration numbers, speed limits, speeding values and accumulated fees from the data captured by the radar.

Exercise 1.3.8. Delete all entries from the `Speeding` table which do not require paying a fee.

Exercise 1.3.9. Choose primary keys for all the tables. Check that the set of all attributes irreducibly functionally depends on the chosen keys.

Do you think that in the `Person` and `Car` tables the attributes `address` and `model` were chosen appropriately? Wouldn't it have been simpler to select all the cars in the area of Linz if the `address` attribute were split up into relevant parts?

Exercise 1.3.10. Redesign the relation schemes of the `Person` and `Car` tables so that the problems mentioned above are easier to handle.

Exercise 1.3.11. After solving the previous exercise, normalize the database scheme to 3NF.

Exercise 1.3.12. After solving the previous exercise, set up the data in the example tables using now the new database scheme.

Exercise 1.3.13. Construct a B-tree with maximum node size 4, which contains the search key values: 1,3,5,7,9,10,16,17,20,21. Then insert the following values one after the other 11, 12, 13, 14. Then delete 10 and finally 12.

Exercise 1.3.14. From the insertion and deletion procedures for B-trees, derive insertion and deletion procedures for B^+-trees.

1.4 SQL

In this section we give a brief introduction to SQL (Structured Query Language) originally developed by IBM Research. It is a standard, and its current version is SQL 2003. The full specification of SQL is more than 2000 pages and covers features which we will not even mention here. Our goal is to introduce only some basic constructions of the language, from which many will correspond to operations in the relational algebra we discussed in Section 1.3.2.

SQL specifies a data definition language (DDL), a data manipulation language (DML), and embedded SQL (to make relational databases accessible in other programming languages, like C, Pascal, PL/I). It also supports basic integrity enforcing, user authorization and transaction management (more on these in Chapter 2).

In SQL terminology a relation is a *table*, and attribute is a *column* and a tuple is a *row*. We will adopt this terminology in this section. We denote keywords of the language capitalized, and we use the EBNF notation specify the structure of statements.

1.4.1 Data Definition

A part of SQL is a data definition language (DDL), that allows us to define/modify/delete tables (i.e. relation schemes) with domains for attributes, define or drop indices, specify integrity constraints, authorization information and storage structure information.

The standard domains (i.e. types) for attributes are the following:

char(n) a character string of fixed length n,

varchar(n) a variable length character string of maximal length n,

int an integer (length is hardware dependent),

smallint a small integer (length is hardware dependent),

numeric(i, d) a numerical value with i digits in the integer part (and a sign) and d digits in the decimal part,

real a single precision floating point number,

double a double precision floating point number,

float(n) a floating point number with user specified length,

date a date storing the years in four digits and the months and the days in two-two,

time a time in hh:mm:ss format.

Additional types, e.g. BLOB (Binary Large OBject) can also be supported by DBMSes.

Type coercion between compatible types are supported (e.g. real to double), just as the usual arithmetic and comparison operations on numerical types, and string operations and comparisons on strings (e.g. pattern matching). Certain simple domains can also be created by the user

```
CREATE DOMAIN name_type AS CHAR(20)
```

The following DDL commands are always understood in context of a database (collection of tables). In practice, this is either the default database that is created when the DBMS is intalled, or, after creating new databases (means for this are provided by the DBMS, e.g. the command 'CREATE DATABASE' in PostgreSQL), it can be specified by the user. (This also means that in practice a DBMS usually handles a whole cluster of databases, and when we want to work with one, we have to tell the interface to which database it should connect.)

Creating a table with a given relation schema can be done in the following way:

```
CREATE TABLE table_name (
      col_name col_type  [NOT NULL]
  [,  col_name col_type  [NOT NULL]]*
  [,  integrity_constraint])
```

Example 1.27. The definition of the `Student` table, using the name_type defined above, can be done as follows.

```
CREATE TABLE Student (
      sid         char(7)          NOT NULL
  ,   fname       name_type
  ,   lname       char(20)         NOT NULL
  ,   PRIMARY KEY (sid))
```

Integrity constraints could be expressed with the CHECK clause. For instance, checking the existence dependency via the foreign keys can be done as follows.

```
CREATE TABLE Attendance (
      sid       char(7)          NOT NULL
    , cid       char(6)          NOT NULL
    , PRIMARY KEY (sid,cid)
    , CHECK  (sid IN (SELECT Student.sid FROM Student)) AND
             (cid IN (SELECT Course.cid FROM Course)))
```

In PostgreSQL there is a language construction to handle this very frequent problem (i.e. to maintain referential integrity). Using it, the same example can be put as:

```
CREATE TABLE Attendance (
      sid       char(7)          REFERENCES Student (sid)
    , cid       char(6)          REFERENCES Course (cid)
    , PRIMARY KEY (sid,cid)
```

The referenced columns must form a candidate key in the referenced table, and by default it is the primary key (which is taken if no colum name is given in the parentesized section).

A table (relation scheme) can be modified with the ALTER TABLE clause.

```
ALTER TABLE table_name [ADD col_name col_type] | [DROP col_name]
```

A table can be deleted with the DROP TABLE clause.

```
DROP TABLE table_name
```

An index can be created with the CREATE INDEX statement.

```
CREATE INDEX index_name ON table_name (col_name [,col_name]*)
```

And it can be deleted by the DROP INDEX statement.

```
DROP INDEX index_name
```

Example 1.28. This defines an index with name "stid" on the `Student` table by the attribute `sid`.

```
CREATE INDEX stid ON Student (sid)
```

1.4.2 Simple Queries

Queries can be stated by the SELECT clause in SQL.

In its simplest form retrieves all the rows in the projection of the cartesian product of the named tables to the specified columns.

```
SELECT [DISTINCT | ALL] [table.]column [, [table.]column ]*
      [INTO new_table_name]
      FROM table [, table ]*
```

The optional parameter DISTINCT results elimination of multiple rows (getting 1NF), while ALL forces to leave all rows in the result. All the columns of a table can be denoted by an asterisk (see one of the subqueries in Example 1.30). The INTO clause can specify the target table in which the results should be saved.

To specify conditions the retrieved rows have to fulfill, we can use the WHERE clause (it appears always in context of a SELECT, INSERT, DELETE, UPDATE clause).

WHERE [NOT] [table.]column operator [value | [table.]column]
[AND | OR [NOT] [table.]column operator [value | [table.]column]]*

The "operator" can be any operator of the language which is applicable to the types of the columns appearing on its sides. "Value" can be any literal value belonging to a type of the language that makes sense in context of the "operator".

To get the rows of the result table sorted in ascending or descending order, we can use the ORDER BY clause (it appears always in context of SELECT).

ORDER BY [table.]column [ASC | DESC] [, [table.]column [ASC | DESC]]*

We can rename relations and attributes in SELECT and FROM clauses by the AS clause.

old_name AS new_name

Example 1.29. To select all the last names from the `Student` table with elimination of multiple rows we can use

```
SELECT DISTINCT lname
    FROM Student
```

To select all first and last names of students called Schmidt we can use

```
SELECT Student.fname, Student.lname
    FROM Student
    WHERE Student.lname = 'Schmidt'
```

Here the "Student." prefixes are actually redundant as there is only one table in the context.

In this example we select students IDs and first and last names who attend the course with ID: 327456, and order by last name and first name. We also abbreviate `Student` by s and `Attendance` by a.

```
SELECT s.sid, s.fname, s.lname
    FROM Student AS s, Attendance AS a
    WHERE s.sid = a.sid AND a.cid = 327456
    ORDER BY s.lname ASC, s.fname ASC
```

The last example selects courses which are first parts of a course sequence (in the pattern matching operator LIKE, '_' matches any single character and the '%' matches any sequence of characters, the escape character is '\').

```
SELECT cid, title
    FROM Course
    WHERE title LIKE '% I%'
    ORDER BY title ASC
```

SQL queries can be nested, forming subqueries inside a query (see Example 1.30).

There are also the set theoretic operations UNION, INTERSECT, EXCEPT (for difference). The operations eliminate duplicate rows by default, so we have to force keeping all rows with the ALL parameter, if desired so. There are also the IN and NOT IN operations to test set membership. The EXISTS operator tests if the result of a subquery is nonempty.

Example 1.30. First let us note in general that usually there are numerous ways to get the same result and there can be great differences in the performance of the queries that we can use. The ones of this example need not be optimal from this point of view because their purpose is to demonstrate the constructs they use and not the practical application.

This example selects the intersection of the students who take the Analysis I course with the ones who take the Algebra I course.

```
(SELECT Student.fname, Student.lname
      FROM Student, Attendance
      WHERE Student.sid = Attendance.sid AND Attendance.cid = 327456)
INTERSECT
(SELECT Student.fname, Student.lname
      FROM Student, Attendance
      WHERE Student.sid = Attendance.sid AND Attendance.cid = 327564)
```

The next example selects the names of the students attending at least one course.

```
SELECT fname, lname
      FROM Student
      WHERE sid IN (   SELECT Student.sid
                       FROM Student, Attendance
                       WHERE Student.sid = Attendance.sid)
```

The same with the EXISTS operator.

```
SELECT fname, lname
      FROM Student
      WHERE EXISTS (SELECT *
                       FROM Attendance
                       WHERE Student.sid = Attendance.sid)
```

And again the same (maybe in the simplest way).

```
SELECT DISTINCT Student.fname, Student.lname
      FROM Student, Attendance
      WHERE Student.sid = Attendance.sid
```

SQL supports the following aggregate functions: AVG, MIN, MAX, SUM, COUNT, EVERY, ANY the groups can be formed using the GROUP BY clause. The HAVING clause can be appended to a query after GROUP BY to restrict the rows in the result using conditions on the aggregated values. If a WHERE clause is also present in the query, its conditions are applied first, then the aggregate functions are applied and finally the resulting rows are filtered by the HAVING clause.

Null values cannot be used in standard operations as they will give false value in any comparison. However there is a special operation: IS NULL, which allows us to test whether the value of an attribute is null. Aggregate functions, except count, ignore rows with null values for the attributes of its arguments.

Example 1.31. Let us take a table `Result` with scheme `sid, cid, grade` describing that a student with `sid` on the course with `cid` got `grade`. To compute the average of the grades of a student we can use the following query.

```
SELECT sid, AVG(grade)
    FROM Result
    GROUP BY sid
```

Selecting those students who have better averages than 2 could be done as follows.

```
SELECT sid, AVG(grade)
    FROM Result
    GROUP BY sid
    HAVING AVG(grade) <= 2
```

Counting for each course the number of people passed on the exam can be done by the following query.

```
SELECT cid, COUNT(DISTINCT sid)
    FROM Result
    WHERE grade < 5
    GROUP BY cid
```

The DISTINCT keyword can be used to filter out the case if a student first passed with 4 and then improved to 2 (and both rows are in the table).

1.4.3 Database Modification

The statements of database modifications have the same structure as of queries in SQL. Deletion is carried out by the DELETE FROM clause.

```
DELETE FROM table_name
    WHERE condition
```

The WHERE clause can have the structure as defined in Section 1.4.2.

Inserting into a table can be done by the INSERT INTO clause. We either have to specify the row to be inserted or to form a query which results the rows to be inserted.

```
INSERT INTO table_name [(col_name [, col_name]*)]
    [VALUES (value [,value]*) | select_cause]
```

If the column specifications are omitted, values should appear for each column of the table (in the same order as the columns were defined). If column names are given the number of values must match the number of given columns. With specifying fewer columns than the table has we can introduce null values. The select subclause (see Section 1.4.2) has to result rows that meet the same requirements we had for explicit value specification.

Updating a subset of rows in a table can be carried out by the UPDATE clause.

```
UPDATE table_name
    SET col_name = expression [, col_name = expression]*
    WHERE condition
```

"Expression" can be any valid expression in the language, including literal constants, operators and column names, as long as it results an value coercible to the type of the column. The WHERE clause can have the structure as defined in Section 1.4.2.

Example 1.32. Inserting a new row into the `Student` table could be done in the following way.

```
INSERT INTO Student
    VALUES ('0245768', 'Alexander', 'Wurz')
```

After the table definition

```
CREATE TABLE Namelist (
    fname    char(20)
,   lname    char(20))
```

we can fill the new table with the names of students using the following insert clause.

```
INSERT INTO Namelist
    SELECT fname, lname FROM Student
```

Adding a student whose first name is not yet known (to be added later), could be done as follows.

```
INSERT INTO Namelist (lname)
    VALUES ('Hackl')
```

Then later ...

```
UPDATE Namelist
    SET fname = 'Hans'
    WHERE fname IS NULL AND lname = 'Hackl'
```

Deleting all students whose last names do not start with R or S (e.g. to leave the students the teacher got in his exercise group after distributing them by the alphabetical order) can be done with following statement.

```
DELETE FROM Namelist
    WHERE NOT (lname LIKE 'R%' OR lname LIKE 'S%')
```

1.4.4 Views and Joins

Finally, we briefly discuss how to construct views and joins in SQL. The CREATE VIEW statement has the following structure.

```
CREATE VIEW view_name AS query
```

The query is any legal query defined in Section 1.4.2. After its definition the view name can, in principle, appear everywhere where the name of a regular relation can. However in certain situations this can lead to problems (see Example 1.33); therefore most DBMSes allow modifications via views only if the view is defined using a single table.

The DROP command also has a version for views.

```
DROP VIEW view_name
```

We have the following join types: *inner join*, *left outer join*, *right outer join*, *full outer join*. The join operation is specified by a further condition (given by the NATURAL, ON, USING clauses) so that the full join specification is as follows.

table_name {{ LEFT | RIGHT } [OUTER] | NATURAL | [FULL] OUTER |
[INNER] } JOIN table_name { ON condition | USING(col [,col]*) }

The condition after the ON keyword can specify the joining conditions (e.g. when the column names do not match for the tables) cf. θ-join in Section 1.3.2. The USING clause can be applied when the join is taken over more than one column. For outer joins the join condition is mandatory, while for inner join (which is the default join; therefore the keyword can be omitted) it is not (if no condition is given the cartesian product is returned).

So the most general form of joins can be achieved with the ON clause. The USING clause is a safer version of the NATURAL join, since the operation is performed only on the specified columns (for instance, if one adds an update_timestamp column to each of the tables all the natural join constructions will break).

Example 1.33. Let us create the table (`Student` $\bowtie$ `Attendance`) $\bowtie$ `Course` from page 22.

```
CREATE TABLE StudentCourse (
      sid       char(7)
    , fname     char(20)
    , lname     char(20)
    , cid       char(6)
    , title     char(30))
```

And fill it up with the data

```
INSERT INTO StudentCourse
      SELECT *
      FROM (Student INNER JOIN Attendance ON
                Student.sid = Attendance.sid)
                INNER JOIN Course USING(cid)
```

Then let us define a view on the `StudentCourse` table that eliminates IDs.

```
CREATE VIEW StCr AS
      SELECT fname, lname, title
      FROM StudentCourse
```

If the user of this view can insert and update records via this view, he can introduce null values in the `StudentCourse` table (namely for the `sid` and `cid` columns).

Constructing the same view directly via the join construction is as follows.

```
CREATE VIEW Sc AS
      SELECT Student.fname, Student.lname, Course.title
      FROM (Student INNER JOIN Attendance ON
                Student.sid = Attendance.sid)
      INNER JOIN Course USING(cid)
```

Now the user who works with `Sc` cannot insert a new row. The reason is that the view is defined in terms of the `Student`, `Attendance`, `Course` tables, so the insertion into `Sc` should manifest as insertions into these tables. However, the

`sid` and `cid` data are missing (the user of `Sc` does not even know about these attributes), and if we wanted to substitute null values for them, the new row wouldn't even appear in the `Sc` table (as join cannot be formed on null values). This is the reason why database modification via views is allowed only if the view is defined in terms of a single table.

1.4.5 Embedded SQL

For several reasons it is advantageous to allow access to relational databases from general purpose programming languages (Perl, C, Pascal, Ada, Fortran, etc.). The SQL standard defines the way to embed it into a *host language*; the standard is called *embedded SQL*.

Embedded SQL provides the full functionality of the (interactive) SQL language, and due to its design principles, it also has a few additional constructs. The SQL statements in the host language are enclosed in an EXEC SQL, END-EXEC pair (the latter can be substtuted by a semicolon).

Inside an SQL statement variable names of the host language can be referred by attaching a colon as a prefix. To save attribute values of a single-row result into host variables, one can use the INTO clause of SELECT. The program in the host language that uses embedded SQL must also contain an SQLSTATE variable, in which the status code is returned after every SQL statement.

Example 1.34. To read in the first and last names of a student from the `Student` table with a given ID, stored in the host variable `id`, into the variable names `fn`, `ln` of the host language, we can use the following statement.

```
EXEC SQL SELECT fname, lname
        INTO :fn, :ln
        FROM Student
        WHERE sid = :id ;
```

The most important difference between the standard SQL and the embedded SQL is the way of data retrieval on the set level (when the result has many rows). In embedded SQL the program in the host language can access the rows of the result one-by-one, and the mechanism to support this is called *cursor*.

For each query (expected to return more than one rows as its result) one should define a cursor as

```
EXEC SQL DECLARE cursor_name CURSOR FOR query ;
```

and then retrieve the rows of the result in some kind of loop until SQLSTATE reports the end of data event and then close the cursor.

```
EXEC SQL OPEN cursor_name ;
do {   // not part of the definition
        EXEC SQL FETCH cursor_name INTO host_var [, host_var]* ;
}while(SQLSTATE != EndOfDataCode);   // not part of the definition
EXEC SQL CLOSE cursor_name ;
```

What happens is that when the cursor is opened the DBMS creates a temporary table for the result of the query, from which the rows are fetched. When a cursor is closed the temporary table is deleted.

Dynamic SQL is a part of embedded SQL which allows us to execute SQL statements which are generated in runtime (the previous SQL statements had to be hard-coded into the program of the host language). The execution is split up in two phases: in the first the system compiles the SQL statement and in the second it executes it.

```
s = "INSERT INTO Students VALUES (0245768, 'Alexander', 'Wurz')";
EXEC SQL PREPARE sqlstatement FROM :s ;
EXEC SQL EXECUTE sqlstatement ;
```

1.4.6 Exercises

Get access to an RDBMS (PostgerSQL, MySQL, MSAccess, etc.) and try the commands out in practice.

Exercise 1.4.1. Create the example tables of Section 1.3.6 using SQL.

Exercise 1.4.2. Translate all the relational algebra expressions which you obtained as solutions for exercises in Section 1.3.6 into SQL queries and try them on the example tables.

Exercise 1.4.3. Create a view for listing speeding values with the data computed in Exercise 1.3.7.

Chapter 2

Information Systems On-Line

In this chapter we take a short outlook on problems and applications that occur usually in information systems working on a network, serving (or collecting information from) a large class of users. Typical examples are: serving web pages with dynamic content, providing 24/7 services (commerce, booking, shopping, etc.), Internet search engines, and so on. Usually these systems are supported by powerful RDBMSes, thus this chapter is also related with Chapter 1.

In Section 2.2 the reader is pointed to a few free software systems that can be used to build on-line information systems. These software, though cost nothing, in many respect represent the state of the art in their classes, and thus are widely used.

2.1 On-Line Databases

In real world applications it is necessary to access databases also remotely via some network (e.g. the Internet) and to allow several users to work at the same time with the database. In such a situation the areas of concurrency and transaction management, and also security issues become prominent. We discuss the first two briefly in Section 2.1.2.

2.1.1 Security Control

Security control in DBMSes has the basic forms of *discretionary* and *mandatory* control. Basically all the DBMSes support the first kind and some additionally support the second kind. Defining a discretionary control scheme consists of defining *authorities*: named security clauses that grant certain privileges to certain users on certain tables (or views). The corresponding SQL command is:

```
GRANT { privilege [, privilege]* | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, tablename]*
    TO user [, user]* [ WITH GRANT OPTION ]
```

where

```
privilege = { SELECT | INSERT | UPDATE | DELETE |
    REFERENCES | TRIGGER }
user = { username | GROUP groupname | PUBLIC }
```

Granting REFERENCES allows to specified user to define foreign key in the given table or to the given table (to actually define a foreign key the user should have this right both on the referencing and referenced tables). Granting TRIGGER allows the creation of a trigger on the specified table. The SELECT, INSERT, UPDATE, DELETE specifications refer to the corresponding operations. The PUBLIC keyword means all the users (also later when new users are defined, they will automatically belong to PUBLIC). The WITH GRANT OPTION allows the specified user to grant the given privilege further to other users.

The opposite command is REVOKE:

```
REVOKE [ GRANT OPTION FOR ]
    { privilege [, privilege]* | ALL [ PRIVILEGES ] }
    ON [ TABLE ] tablename [, tablename]*
    FROM user [, user]* [ CASCADE | RESTRICT ]
```

The CASCADE option will revoke the privilege not only from the specified (class of) users but also from users who got the privilege from a user in the revocation class. If the RESTRICT option is given and there exist such a chain of one user granting the privilege to another, the revocation will fail. If none of RESTRICT or CASCADE is given, the default is applied (it is implementation dependent, e.g. in PostgreSQL the default is RESTRICT).

A user has the "sum" of privileges granted to him directly or via groups. This means that revoking a privilege from PUBLIC, does not necessarily mean that no one has this privilege anymore, because it might have been granted to other groups or users directly (and those grants are not revoked by revoking the privilege for PUBLIC).

If a creator of a view loses the SELECT privilege on any of the underlying tables, the view is dropped. If the creator of a view loses a (non SELECT) privilege on any of the underlying tables, he loses this privilege also on the view. So do all users to whom this privilege was granted. (To DROP an object is a privilege of the creator of the object and it is outside the scope of discretionary control.)

In mandatory control, data objects are assigned classification levels and users have clearance levels. The levels form an ordering. Then a user of clearance level i can retrieve a data object of classification level j if $i \geq j$; moreover, the user can update the object if $j = i$. The second restriction is there to imply that, for instance data cannot be copied into an object of lower classification level, and that users cannot create data which they cannot retrieve afterwards. A user with clearance level i might be allowed to switch to level k ($< i$) to insert/update data in the kth classification level.

Mandatory control is not implemented by most commercial DBMSes, but there exists modifications that do support it (usually used in military or government circles).

Another interesting security criterion is called "Statistical Security". Databases created for statistical purposes should allow agglomerative queries to be

performed, but they should forbid to obtain (or even to deduce) information about individual rows. (Think, for instance, of a DB with collected medical data from which one can determine how many people suffered from a given illness last year in the country, but one should not be able to find out the names of these people.) Such DBs try to protect individual data by methods like allowing only queries where at least n rows are involved in the aggregation (for some $n > 1$). However, this does not give full protection (try to construct a counterexample!).

Once the security scheme is implemented on the database, the security subsystem of the DBMS can identify each user by a login name, authenticated by a password. However, if the clients need not be in a closed network, it can also be mandatory to communicate via a *secure channel*.

One of the most common protocols is *SSH* (Secure SHell), which can establish secure connections between hosts, encrypt data that are exchanged also in the authentication phase, and provide secure tunneling capabilities for a wide variety of applications. Another such protocol is *SSL* (Secure Sockets Layer), which provides secure data exchange similarly to SSH, and used mostly in WWW applications. Both protocols allow to use public-key cryptography, usually based on the RSA cryptographic scheme. These cryptographic tools are considered strong nowadays (relying on the difficulty of factoring integers), and the corresponding protocols are widely used.

Remark 2.1. In short the RSA (Rivest-Shamir-Adleman) algorithm can be summarized as follows:

Key generation: generate two random primes p, q of approximately the same bit-length such that $n = pq$ is of the required length (e.g. 1024 bits). Set $h = (p-1)(q-1)$, choose an integer $0 < e < h$ such that $\gcd(e, h) = 1$ and compute $0 < d < h$ such that $ed \equiv 1 \mod h$. The public key is the pair (n, e) and the private key is (n, d); the values p, q, h must also be kept secret.

Encryption: Let the message be m such that $m < n$ (in practice a long message can be broken up into segments with length smaller than $\log_2(n)$ and the bit sequence is just understood as the number m). The (segment of the) encrypted message corresponding to (the segment represented by) m is: $c = m^e \mod n$.

Decryption: Let (a segment of) the encrypted message be c, the corresponding (segment of the) decrypted message is: $m = c^d \mod n$.

2.1.2 Transaction Management

Transaction management is of utmost importance to maintain consistency of the database, for instance being able to recover the database to a consistent state even after a system crash. This feature could have been discussed also in Chapter 1 without mentioning concurrency, focusing only on recovery. First we discuss database recovery, which is based on storing extra information about the tables of the database. In this case, however, this redundant extra information shows up only at the physical level, so that at the conceptual level one has no access to the extra data. What one sees on the conceptual level is a construct of the DML that allows us to move the database from a consistent state to another consistent state in a single logical step (even if the "move" is a long sequence of modifications).

Because of dependencies between the tables of the database (established by foreign keys) which reflect relationships between entities described by the corresponding relation schemes, a modification of the database that start from a consistent state and reaches a consistent state requires in general to modify many tables. If in this process, a system failure occurs when some of the tables are modified while others are not (e.g. the disk fills up) after solving the problem (e.g. freeing up space on the disk, or installing a larger disk and copy the content of the old one to the new) the database is in an inconsistent state. The DBMS should prevent such inconsistencies to show up.

This logical step (or unit) of modifications is called a *transaction*. In DBMSes it is usually start with the keyword

BEGIN TRANSACTION

and ends with one of the keywords

COMMIT
ROLLBACK.

Inside the transaction unit a sequence of DDL and/or DML statements can be placed. Transactions cannot be nested.

The *transaction manager* of the DBMS "saves" the current state of the database at the start of a transaction. Then the transaction block is being executed and when an error is encountered a ROLLBACK is requested to restore the saved state of the database, and the the transaction terminates (unsuccessfully). On the other hand, if no error occurs and the transaction block terminates with a COMMIT, it signals to the transaction manager that the modifications are done and the new state of the database can be regarded consistent.

A ROLLBACK can be triggered explicitly by user writing the transaction block, or implicitly by the DBMS (see the "disk full"-problem above, or think of the violation of some integrity constraint in an attempted modification of a table by a statement in the transaction block).

The recovery is solved by using *log* (or *journal*) files, into which the DBMS archives states of the database for each transaction. The actual implementations can be highly complex, especially when concurrency is supported. The "all or nothing" property of the transactions, or in other terms the fact that either the whole transaction commits or everything is rolled back to the initial state, is also called the *atomicity* property.

A typical application program is then consists of a sequence of transactions.

Example 2.2. `Student, Attendance, Course` as in Section 1.3.2. Using dynamic SQL embedded in C with a DBMS that supports transactions (using the syntax above) we could write a little C-function to cancel a course as follows.

```
/* cid has to be the pointer to the course ID converted to string,
   returns 0 on success 1 on error,
   SQL_OK_Code is expected to be a defined symbol. */
int cancelCourse(char *cid){
   char query1[50]; char query2[50];
   strcpy(query1, "DELETE FROM Attendance WHERE cid = ");
   strcpy(query2, "DELETE FROM Course WHERE cid = ");
   strcat(query1, cid); strcat(query2, cid);
```

```
    EXEC SQL BEGIN TRANSACTION;
    EXEC SQL PREPARE q1 FROM :query1;
    EXEC SQL EXECUTE q1;
    if(SQLSTATE != SQL_OK_Code){
       EXEC SQL ROLLBACK; return 1; }
    EXEC SQL PREPARE q2 FROM :query2;
    EXEC SQL EXECUTE q2;
    if(SQLSTATE != SQL_OK_Code){
       EXEC SQL ROLLBACK; return 1; }
    EXEC SQL COMMIT;
    if(SQLSTATE != SQL_OK_Code){ return 1; }else{ return 0; }
}
```

To cancel a course one also has to delete the records of students in `Attendance` who already registered and then remove the course from the `Course` table. If during the execution of any of the queries an error occurs, the tables are rolled back to the initial state (no matter how many rows have been deleted till the point of the error) and the C-function reports an error. If the COMMIT is executed successfully, the modifications have been made permanent in the database. Otherwise an implicit ROLLBACK has been made by the DBMS, or if the system just crashed when the COMMIT was executed, the implicit ROLLBACK will be made when the DBMS is restarted and it detects that there is a COMMIT statement broken due to the crash.

Remark 2.3. In distributed database systems, where the tables of the database can be located on different machines, there is an additional subtlety of committing modifications. In this context the DBMSes on the machines are usually referred to as resource managers, and the master transaction manager as a coordinator. The commit process is called a *two-phase commit*. In the first phase the coordinator asks the resource managers to prepare for commit: this makes them decide locally if the modifications are committable and write the commit logs out to disks; when it is done they reply "OK" or "Error". The coordinator then collects answers from the local resource managers and writes its decision on the transaction into its log (if not all local managers replied "OK" the decision must be rollback, otherwise it might depend on further conditions). After this it sends the global decision to the local resource managers which will act according to it (i.e. roll back if told so, even if the local modification would be committable).

Another important issue in database management is concurrency. It appears prominently in *on-line transaction processing (OLTP)*. OLTP is mostly concerned with processing small amount of data per transaction, often in time-constrained environment (e.g. real time systems). The transactions intend to access the database simultaneously and the DBMS has to resolve any interference between the transactions.

Remark 2.4. For the further discussion it is advantageous to make a distinction between a consistent and a *correct* state of a database. Correct means consistent (i.e. fulfilling every integrity constraints of the model) and reflecting reality. Correctness is a concept on the meta level (with respect to the relational model of the database) and therefore it cannot be enforced in the system. For instance

if the name of a student contains a typo in the `Student` table, the database containing it is consistent but not correct. On the other hand, what will be at the center of our interest in the rest of this section is to guarantee that a correct database gets transferred into a correct one by an execution of individually correct transactions concurrently.

Transactions which are valid on their own, when executed concurrently can leave the database in an incorrect or even in an inconsistent state. There are three basic kind of interference:

Lost update The transactions read and update the same row of a table concurrently as in the diagram below.

Time	Transaction 1	Transaction 2
$t1$	retrieve r	
$t2$		retrieve r
$t3$	update r	
$t4$		update r

This obviously leads to incorrect result if the update of r depends on the value retrieved.

Uncommitted dependency A transaction reads an uncommitted change of a row by another transaction, which afterwards is rolled back, so the first transaction works on false assumptions (left diagram). Another version of this is when the first transaction updates the row which was previously updated by the other transaction, which is then rolled back. The update of the first transaction gets lost (right diagram).

Time	TA 1	TA 2
$t1$		update r
$t2$	retrieve r	
$t3$		rollback

Time	TA 1	TA 2
$t1$		update r
$t2$	update r	
$t3$		rollback

Inconsistent analysis The first transaction computes an aggregate function on the rows of a table which are updated by the second transaction.

Time	Transaction 1	Transaction 2
$t1$	retrieve r_1	
$t2$	aggregate r_1	
$t3$		update r_3
$t4$	retrieve r_2	
$t5$	aggregate r_2	
$t6$		update r_1
$t7$	retrieve r_3	
$t8$	aggregate r_3	

The result of the aggregate function is wrong no matter how we view it, because it computed with the old value of r_1 and the new value of r_3; therefore reflects a state of the database which need not be correct.

Such situations can be handled with *locking* mechanisms, that deny access to a row, or a set of rows, in a table for other transactions while modifications are being done. We distinguish the following lock types:

Exclusive lock (write lock, X-lock) Requests for any other lock on the object must wait till the active exclusive lock is released.

Shared lock (read lock, S-lock) Requests for other shared locks on the objects are immediately granted, exclusive locks must wait until the object is released.

The locking is usually done implicitly by the transaction manager depending on the operations in the transaction.

The locking protocol can then be defined as follows. A transaction that intends to retrieve a row must first acquire an S-lock on it. If a transaction intends to modify a row first it must acquire an X-lock on it, or, if it already has an S-lock on it, it must "upgrade" it to an X-lock. If a locking request from a transaction cannot be immediately granted, the request goes into a waiting queue and from this the requests are granted on a first-come first-served basis. If the object has an S-lock on it, other S-lock requests (for this object) can be immediately granted. However, this can force the X-lock requests in the waiting queue to wait for a long time (potentially forever). This phenomenon is called *starvation*. The system should be prepared to avoid starvation, e.g. by not granting out of queue locks anymore if the first in the waiting queue is waiting for more than a given amount of time.

In this scenario a new problem can occur, if two or more transactions wait on each other to release a lock. This situation is called a *deadlock*. We can see this in the diagram of the "lost update" problem now with locking applied.

Time	Transaction 1	Transaction 2
$t1$	acquire S-lock on r	
$t2$	retrieve r	
$t3$		acquire S-lock on r
$t4$		retrieve r
$t5$	X-lock request on r	
$t6$	wait	X-lock request on r
$t7$	wait ...	wait ...

To predict and avoid deadlocks can be very difficult, and the detection of a deadlock is also not trivial (finding circles in graphs that represent which transaction waits for which others). So in practice, the transaction managers just assume that if a transaction has not done any work for more than a given time limit, it is deadlocked. In such a case the system chooses a "victim" transaction from the set of presumably deadlocked ones, which will suffer an implicit rollback, and therefore releases all the locks it held. (This is also a reason why a valid transaction can be rolled back in an OLTP environment.)

Another important concept in showing correctness of an execution of a given set of transactions is *serializability*. An (interleaved) execution (also called as a *schedule*) of a set of transactions (which are individually correct by assumption) is called *serializable* if and only if it produces the same result as some serial execution of the same set of transactions; and this does not depend on the (a priori correct) initial state we start from.

As the transactions are assumed to be individually correct, if a schedule is serializable then it transfers the database from a correct state into a correct state. Serializability can be difficult to test for an arbitrary schedule; therefore the following theorem is important.

If all transactions obey the two-phase locking protocol, then all possible schedules are serializable.

The *two-phase locking protocol* requires that a transaction must acquire a lock (of the appropriate kind) on a tuple before it operates with it and that after releasing a lock the transaction must never acquire a lock again. (In other terms, the transactions must acquire locks and they are expected to separate the lock acquisition and the lock release phases.)

There is still a problem with recoverability stemming from the uncommitted dependency problem. Take the diagram on the left from the discussion of the problem, now with locking applied.

Time	Transaction 1	Transaction 2
$t1$		acquire X-lock on r
$t2$		update r
$t3$		release X-lock on r
$t4$	acquire S-lock on r	
$t5$	retrieve r	
$t6$	release S-lock on r	
$t7$	commit	
$t8$		rollback

Since the second transaction is rolled back the first one should also be rolled back because it used a value of r which was updated by the second and then restored. However, it is not possible to roll the first transaction back since it has already committed.

Therefore to achieve recoverability, we require that a transaction which uses data modified by another transaction must not commit before the other terminates (and if that rolls back, the first must be rolled back too). Such a schedule is called *cascade-free*.

Remark 2.5. To increase the transaction throughput of the DBMS, in practice systems do not require full serializability and cascade-freeness. Of course, this can lead to interference in critical situations. The systems control the tightness of the requirements via isolation levels.

Another approach to avoid starvation (used frequently together with isolation levels) is the multi-version concurrency control. In this approach each transaction sees the snapshot (or version) of the database which is created at its starting time. Reads do not block writes and vice versa, only writes on the same row can block each other. The implementation of a multi version system is much more complicated than of a traditional system, however higher performance can be achieved with fewer compromises.

Finally the requirement for an online transaction processing system can be summarized in the ACID test—Atomicity, Consistency (or Correctness), Isolation and Durability.

Atomicity Results of transactions are either all committed or rolled back ("all or nothing" principle).

Consistency (or Correctness in the more rigorous sense) The database is transformed by a transaction from a valid state into another valid state. (Consistency, however need not be preserved at intermediate points of the transactions.)

Isolation (or serializable and non-cascaded in the more rigorous sense) The result of a transaction is invisible for other transactions until the transaction commits.

Durability Once a transaction has committed, its results are permanently recorded in the database, so that they survive any subsequent system crashes.

Example 2.6. Although the concepts should be clear on their own, here are a few wordy examples to illustrate what these requirements mean when the on-line information system is, say, a discussion forum.

Atomicity: When a new user registers he might have to type in lots of data through possibly several pages. Whenever a page is submitted in the progress of the registration process, some of the tables of the system might get updated right away. If the user changes his mind at the last page and cancels the registration the whole transaction is rolled back as if the registration process have never been started.

Consistency: Let the consistency criteria require that all postings on the forum should have a user ID which is a foreign key for the table of users. If a careless administrator tries to delete the account of a user without first deleting all his postings the forum, the DBMS should trigger an implicit rollback on the transaction, since it would create orphaned postings (postings without a valid user ID) violating integrity constraints.

Isolation: When a new user registers (as in the point of Atomicity), as long as the registration process is not completed, that is the corresponding transaction is not committed, the updates which are made in the tables of the database are not visible for other transactions. For instance, if the administrator of the site starts a query to produce some statistics on the registered users of the forum, the new user will not appear in the statistics because his registration was not yet committed when the administrator's query started; although the user's data might have already been inserted into some of the tables of the database.

Durability: Let us suppose that the site of the forum also sells things on-line and let a user order something. He sends his request, which the e-commerce module of the system processes successfully, but a millisecond after transaction commits a fatal malfunction in the UPS cuts off the power for the computers of the service provider. The HW staff manages to bypass the smoking UPS in about 10 minutes and after another 5 minutes of booting and initialization the system is up and running again. The e-commerce subsystem continues to process the new order (inserted by the committed transaction) and the user gets his gadget in a few days.

Example 2.7. Typical examples of OLTP systems are: e-commerce (on-line shopping-, booking-, banking systems), e-trading, on-line service providing (discussion forums, email, homepage, etc.), distributed source archiving systems (for software development, publishing, etc.), e-governments and e-administrations and countless other applications.

2.1.3 Static Archives

The operational information systems (OLTP systems) of enterprises, accumulate a large amount of data via archiving. These data should be used for further analysis in decision support.

A *data warehouse* is a collection of integrated, nonvolatile enterprise data, which is often restructured, pre-analyzed, and reflect evolutionary processes of enterprise. Practically it is a huge database for analytical purposes.

Integration collects data for the warehouse for several data sources, typically from the operational databases of the enterprise, and stores them in a restructured form. Nonvolatile means that the data is usually read-only once it is loaded into the warehouse. The data is usually collected throughout a long period of time, so it reflects not momentary states but the changes or evolution of the states in the given time period. Since the amount of data can be enormous (terabytes) it is not uncommon that certain pre-analysis is done on the data as it gets loaded into the warehouse (e.g. some partial sums on certain columns are computed, and the total sum on that column is updated).

The queries occurring in the context of data warehouses can be much more complex and sophisticated than the ones on the operational databases. Consequently it is important that the DBMS is of high performance, having reasonably fast responses on ad-hoc queries or what-if analysis. The warehouse databases are usually less normalized than the operational databases. Furthermore, when data is loaded into the warehouse, additional error correction might also be applied (e.g. filling up missing information with defaults). A disadvantage of data warehouses is their cost.

The recognition that many queries are executed only on restricted subsets of a data warehouse initiated the idea of data marts. A *data mart* a special-purpose, integrated, usually volatile and time variant database, supporting only certain decisions. The data mart can be extracted from the data warehouse of the enterprise, or it can as well collect data on its own. Sometimes, if the database schemes are carefully prepared and "synchronized" the data warehouse can be consolidated from the data marts of the enterprise.

The data warehouses and data marts provide the basis for *on-line analytical processing (OLAP)*. The additional concept that plays essential role in OLAP is *multi-dimensionality*. Analytical processing requires data aggregation using many different groupings, along different subsets of attributes and this is modeled better in the multi-dimensional view.

A multi-dimensional database is viewed on the conceptual level as a multi-dimensional array. The space of array positions is spanned by the independent attributes, and in the array elements the dependent ones are stored.

Example 2.8. A typical OLAP example is a simplified sales table of a company. It is a three-dimensional array in which the fist dimension is spanned by the customers of the company, the second is by the products and the third dimension is time (say in day-resolution). The quantities go into the cells of the array: e.g. if customer #302453 purchased from product #10029823 on 12. Aug 1997, 300 pieces, then into the cell identified by the corresponding customer ID, product ID and the date goes 300.

In reality a multi-dimensional databases have to be more flexible than that. Since the data is being analyzed with their help, the analysis might reveal that

the initial distribution of the independent and dependent attributes does not reflect reality that well as another distribution. In this case it might occur that an independent and a dependent attribute has to swap roles (this is called pivoting). Also the independent attributes might form hierarchies (e.g. hour, day, week, month, year) with respect to which the dependent values can be aggregated. Moving from a lower level of aggregation to a higher one is called "drill up" and the opposite is called "drill down".

The advantage of multi-dimensional databases is that they can give fast responses to queries. Their disadvantage is the phenomenon of data explosion, against which developers try to apply sparse representation techniques. Also multi-dimensional databases are not as matured as relational ones, thus in OLAP the underlying DBMS is often a relational one.

When the multi dimensional view of the data is based on traditional relational databases the subject is referred to as ROLAP (R stands for relational). The multi dimensional model is represented in the simplest case by a star scheme (over the tables of the database) in the relational model. At the corners of the star are the dimension tables, each representing an independent dimension (e.g. date, product, etc.) and in the middle is the fact table which connects all the dimension tables with a multiple join. A row in the fact table contains foreign keys to each dimension tables, providing the coordinates of the corresponding cell in the multidimensional model and the values of the dependent attributes at the given coordinates (see Figure 2.1). When the hierarchy of the dimensions are represented explicitly via normalization the scheme is named snowflake.

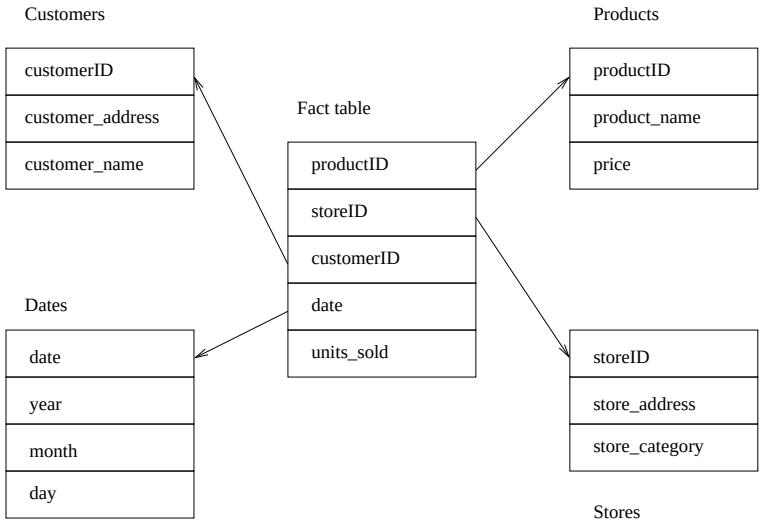


Figure 2.1: Star scheme representation of multidimensional models

The SQL specifications (at least from 1999 on) allow simplified aggregation queries that apply several groupings for the given aggregation function. The extension of the GROUP BY clause allows to prescribe the following grouping schemes. The user can specify sets of attributes on which to group explicitly.

GROUP BY GROUPING SETS ((col_name [,col_name]*)
[, (col_name [,col_name]*)]*)

The ROLLUP scheme takes the initial sublists of a list of attributes to group on including the empty list (which results an overall sum).

GROUP BY ROLLUP (col_name [,col_name]*)

The CUBE scheme takes all the possible subsets of the given set of attributes.

GROUP BY CUBE (col_name [,col_name]*)

Remark 2.9. Assume there there are columns with names `a` and `b`. Please note that

GROUP BY ROLLUP (a, b)

is equivalent to

GROUP BY GROUPING SETS ((a, b), (a), ())

and

GROUP BY CUBE (a, b)

is equivalent to

GROUP BY GROUPING SETS ((a, b), (a), (b), ())

In the resulting tables of the queries using these special groupings the attributes on which no grouping is made are set to NULL. In these cases, the meaning of the null value is not the same as originally.

Example 2.10. Let us take the specification `Result` table from Example 1.31 and let it contain the following data.

`Result`

sid	cid	grade
0251563	327456	1
0251563	327564	2
0245654	327456	1

Then the query

```
SELECT sid, cid, AVG(grade) AS avg_grade
      FROM Results
      GROUP BY GROUPING SETS ((sid), (cid))
```

results

sid	cid	avg_grade
0251563	NULL	1.5
0245654	NULL	1.0
NULL	327456	1.0
NULL	327564	1.5

Data mining is an important application relying on the integrated and consistent data that data warehouses can provide. The task of it is exploratory data analysis. Data mining tools usually can also cooperate with OLAP tools and utilize their strength in query processing and in the multi-dimensional view of the data.

Data mining tools apply statistical techniques to the large amount of data available in the data warehouse and look for patterns (trends, regularities) in the data. That is, the purpose of data mining is to discover new information in the data. Commonly used methods come from the area of machine learning:

Association rules They are used for identification of attribute values that occur more frequently together than it would be expected if the values (or the real objects they represent) were independent.

For example, if we analyze the archives of a supermarket, we might find that after looking at 200000 bills, in 70% of the 100000 cases when somebody bought bred she also bought milk. We say then that the association rule (bread $\Longrightarrow$ milk) has 70% confidence with 50% support on a 200000 population.

Genetic algorithms They simulate the process of evolution in nature. The model works with a population of individual entities (data items), operations of mutation and crossover and a fitness function on individuals. A cycle of the algorithm usually consists of evaluating the fitness function on each individual and select the best ones. Then compute a new population using the mutation and crossover operations on the selected subset and replace the old one with it.

A common application of genetic algorithms is multi-dimensional optimization (the fitness function is being optimized) in cases where there are no fast special purpose algorithms available.

Decision trees A classification method to isolate homogeneous classes of data for some target attributes and describe the classes with as few as possible test attribute values. Initially the whole population of data items are assigned to the root of the tree and all test attributes are considered available. Then at an intermediate step, the algorithm takes a leaf node of the tree and if all the data items have the same target attribute values the leaf is declared to be a class and is not divided further. Otherwise, an available test attribute is chosen which maximizes some measurement of the information gain when the data items of the node are split up into groups distinguished by the value of the chosen attribute. The subsets in the resulting partitioning of the items become new leafs of the tree, attached to the node which was split up.

Clustering Algorithms in this category associate the data items which are similar to each other (with respect to some metric) by forming groups of them. There are several kind of clustering algorithm and they can also be parametrized to fine tune their behavior (e.g. setting thresholds that determine when to split a cluster into smaller ones and when to do the reverse process, etc.).

2.1.4 Search Engines

A special large scale database application is the well known category of search engines, also known as *information retrieval systems* (IR systems for short). "Search enginology" is a science on its own, including not only database management but several steps from data acquisition to ranking and presentation of the results. The internal details and algorithms of specific search engines are industrial secrets, so there is very little information available publicly. In this section we take a very brief outlook on how search engines work.

The first important component of a search engine is the database. It is an *inverted index*, where the search keys are the terms we can search for and the entries are pointers to documents which contain them. Moreover there is also additional information stored that reflect the relevance of the document in context of the given search key.

Data Acquisition

A primary task in building up and maintaining such a database is data acquisition. This means not only to find valid URLs on the web but also relevant information about them which can be used in the indexing process. There are two basic kind of data acquisition, the first is the human powered one, in which the authors of the web pages submit their URLs or the editors of the company collect the information and submit it to the document processing unit. The second is the automated way, in which programs, also called "web crawlers", "spiders" or "robots", roam the Internet going from link to link and collect information on the web pages they find.

These programs often referred to as *software agents*, to emphasize that they have an otherwise unusual degree of autonomy. They usually start with an archived list of URLs (considered as the best ones in the database) and follow the links found on these pages. They provide data for the parser and indexer subsystem that extends/updates the database (also called the index) whenever appropriate. They might filter the data they find on a particular page, selecting only the keywords or other meta tags, or sending only a limited amount of data from the beginning of the page, etc. Recent programs even dig into the (doc, ppt, pdf, ps, etc.) documents they find available on the Internet to explore the domain of information that exists in these special formats.

Most of these agents are "well behaved" in the sense that they respect the wish of the webmaster if he wants to keep them away from certain parts of his site. The policy for the software agents can be set up by creating a file `robots.txt` in the root of the served directory structure. The file has a fixed structure which is parsed by the robot. For instance, with the following file the administrator can exclude all robots from the `cgi-bin` and the `tmp` directories.

```
User-agent: *
Disallow: /cgi-bin/
Disallow: /tmp/
```

Document Processing

Once the data has been retrieved the parsing and indexing phase comes (as we saw, some preliminary parsing of filtering also be done in the crawler). This is

done by the *document processing* unit of the system.

The first phase consists of document preprocessing. This brings the various kinds of documents which were retrieved in several different formats into some standardized format that can be handled by the algorithm of the subsequent phases. It also breaks down the document into units which will be retrievable from the database (e.g. the URLs, sentences, paragraphs, sections, whole documents, etc.) and isolates these pieces.

The second phase identifies indexable elements of the documents; the process is also called *tokenization*. A token or search term can be more than a single word; the concept comprises also phrases that may contain spaces or even punctuation. The set of rules which guide the tokenizer changes from search engine to search engine, and is not disclosed to the public.

In the third phase the so called *stop words* are eliminated from the set of terms to be indexed. This way one can save on average 40% of texts to be indexed and this leads to saves in system resources. Typical stop words are: a, the, and, or, but, in, over, for, is, are, etc.

The fourth phase consists of term *stemming*. Its job is to remove suffixes from words in a possibly recursive way, ending up in a stemmed form. For instance, if the user entered the word "computing" he might also be interested in results containing the words compute, computes, computed, computer, etc. So the stemmed form of all these words would be "compute". which will then be indexed. Weak stemming strips off only inflectional suffixes (-s, -es, -ed, etc.) while strong stemming removes both inflectional and derivational (-able, -ability, -aciousness, etc.) suffixes. With this step again a substantial amount of system resources can be saved and the scope of searches can easily be extended to associated words. On the other hand this also decreases search precision which might be undesirable in certain situations.

The fifth phase extracts the indexed terms into an inverted file which lists the index entries and contains pointers to their occurrences as well as their frequency in the document. Advanced document processors also have phrase and named entity recognizers (the latter will identify for instance that "Schwarzenegger" is a name in the text) and even categorizers (e.g. to detect that the named entity is a person/country/company/etc.—the best search engines can most probably recognize the famous Austrian as a person).

In the sixth phase weights are assigned to the terms occurring in the document. A commonly used weighing scheme is the *TF-IDF* one, which compares the Term Frequency in the document with the Inverse Document Frequency for the term in the database. The formula to compute it is $TF \cdot \log(N/IDF)$ where N is the number of documents in the database. It is usually applied together with length normalization to compensate the higher term frequency of longer documents.

Finally the database is updated with the data of the inverted index constructed for the new document. That is, the search terms that were found in the new document will now also (inverse) index it in the main database, with the computed weights and pointers to the occurrences.

Query Processing

The other important unit of a search engine is the *query processor* that parses and processes the queries of the users. It shares many steps with the document

processor.

The first step is the tokenization of the input query. Usually in this case it just means to break the query string down into alphanumeric substrings separated by whitespaces and/or punctuation signs.

Then the tokenized string is parsed to detect control keywords like Boolean operators (AND, OR, etc.), proximity operators (NEAR), and reserved punctuation (quotation marks). After this the system might immediately form the query for the database and execute it.

If more preprocessing is required then usually stop word removal and stemming comes next. This, just as in the case of the document processor, can simplify the query a lot, especially search engines that apply natural language processing (NLP).

After this the query is formed for the inverse index. The way it is done can depend on the type of matching the system uses—the process of searching the database for documents that satisfy the requirements of the query. For instance if Boolean matching is used, a tree of a logical expression is created (leafs are search terms and other nodes are logical connectives). This is again a point where the query can be executed on the database.

Advanced search engines may go further and before the execution, the query is expanded. The expansion includes search terms that did not appear in the original query but are synonyms, generalizations or specializations of terms that appear. For instance searching for "car" might also include "auto" and "vehicle".

Moreover, if there are more than one query terms, they can be weighed by the query processor (or in unusual cases the weight information can come from the users). After this step the expanded and weighed query is executed on the database. The matching in most of the cases boils down to a binary search.

Ranking of Results

The next important subsystem is the one that ranks the (usually very large) set of results obtained for a query. These ranking algorithms assign scores to each document that reflect the relevancy for the given query. The scoring algorithms are also treated as industrial secret, but they usually regard the presence/absence, weights, frequency, proximity and location of query terms in the document, the size, age, attached meta data of the document, the links that point outward from the document and the links that point to the document from other documents, and many other things (like the payed entries in Google).

An interesting ranking scheme based on the inward/outward links of documents is the *HITS* algorithm (Hyperlink Induced Topic Search). It is based on the simple observation that if a document A has a link to document B then the author of A considers B as a valuable information source. Moreover, if A points to a lot of "good" documents then the opinion of the author of A is more valuable, which also increases the value of B.

A document which points to many others is a good *hub*, and a document to which many others point is a good *authority*. The goodness of documents is established in an iterative process.

For a given query the HITS algorithm needs an initial set of documents called the root set. Then this set gets extended with documents that point to documents in the root set (using the database) and with documents to which

documents from the root set point. Then each page gets assigned a hub number and a authority number, both initialized to 1. In an iteration of the algorithm these numbers are renewed by the following rules: the new authority number of a page P is the sum of the old hub numbers of the pages point to P, and the new hub number of P is the sum of the old authority numbers of the pages P points to. After the hub and authority numbers are updated they are normalized to avoid their limitless increase. The iteration is repeated until the system gets to a equilibrium-close state.

This algorithm ranks pages by popularity, and it also has its drawbacks: mutually reinforcing relationships between hosts (if they have many links to each other) and topic drifting (as the extension of the root set might include documents which are not relevant for the original query). These issues are solved or alleviated by modifications of the original algorithm.

Based on the ranking the results are listed for the user. One more information source for the sophisticated search engines is the user himself. They might allow the user to provide feedback or to modify the query and search again. Sometimes the user gives feedback just by clicking on the link of the chosen page, which points to the site of the search engine and contains some additional data to redirect the browser to the requested document. With this the user ends up where he wanted and the system at the site of the search engine also gets to know the choice of the user.

Finally we can mention that there are plenty of documents (easily locatable with any search engine) that explain how to use search engines effectively and how to improve the relevance of the search results by well posed queries. As this topic is out of the scope of the lecture we do not discuss it here.

2.1.5 Exercises

Transactions are supported by many state of the art DBMSes; therefore solving these exercises are not at all difficult. More complicated tasks would be to design the transaction blocks in time critical systems for optimal throughput. The exercises are recommended to be solved in practice, using your favorite DBMS and programming language (e.g. PostgreSQL/Perl).

The context is the exercise database introduced in Section 1.3.6.

Exercise 2.1.1. Design a simple procedure (function/subroutine) for deletions of entries from the `Car` table which enforces the following constraints.

- The entry of a car cannot be deleted if there is more than 50 EUR accumulated unpayed fee in the database. Remarks:
  - The total fee can come from many different speedings.
  - Payed penalties are assumed to be removed from the `Speeding` table.
- If the total amount of unpayed penalties is at most 50 EUR, the entry of the car can be removed after all the speeding entries of the car is removed from the `Speeding` table.

The procedure has to use transactions to modify the database so that the deletion operation transforms the database from correct state to correct state.

Exercise 2.1.2. Design a procedure for deletion of entries from the `Person` table with an input boolean parameter `cd`. If `cd` is false the person cannot be deleted as long as he has a car registered, if `cd` is true, the person requested to unregister his car(s), so also the entries in the `Car` table that belong to the person in question should be deleted. (Use the procedure of Exercise 2.1.1 to to the deletion and to decide whether the person can unregister his car(s)). The procedure has to use transactions for the database modification.

Exercise 2.1.3. Design a procedure for updating registration numbers. Input parameters are the old and the new numbers. The procedure has to do the modification in a transaction to ensure consistency of the database.

Exercise 2.1.4. Find as many free DBMSes as you can that fulfill the ACID test. Find their official web-pages, the nearest mirror for download, and documentation for it in German (or other languages except English).

2.2 In Practice

This section originally intended to provide a brief outlook to freely available software that can be used for building on-line information systems. The main components are: the DBMS, for which the choice of the author is PostgreSQL, the webserver, which would be Apache, and some powerful programming or scripting language to serve dynamic web pages and support user interaction, these would be Perl and/or PHP.

Because of the lack of time this section could not be written. However, there is more than enough information available on the Internet for the interested reader.

2.2.1 Web Servers

Information about the Apache webserver can be found in [1].

2.2.2 Database Management Systems

Information about PostgreSQL can be found in [6].

2.2.3 Dynamic Content Creation

Information about Perl can be found in [4] and about PHP in [5].

Chapter 3

XML

Extensible Markup Language (XML) is a globally accepted, vendor independent standard for representing text-based data. The organization behind XML and many other web related technologies (like the Hypertext Markup Language: HTML, Cascaded Style Sheets: CSS, etc.) is the World Wide Web Consortium (W3C). The W3C can be described best as a committee where different representatives of the development community collaboratively decide standards for the Web. XML was derived from the Standard Generalized Markup Language (SGML), which exists for decades, via substantial simplifications.

The most important goals XML was designed to achieve are:

Simplicity XML documents should be strictly and simply structured.

Compatibility XML is platform independent. It should be easy to write or update applications that make use of XML.

Legibility XML documents should be human readable.

Applications of XML include (but not limited to)

Data storage Providing a standard way for storing text data in a hierarchically structured way.

Data exchange Serving as a common language to which text based data can be transformed from various formats.

Templating Creating template documents that describe various fields and attributes, for instance in case of business forms.

HTML translation Data stored in XML format can easily be transformed into HTML for Web publication.

SOAP Simple Object Access Protocol to exchange messages in a platform independent way.

In this section we give a brief introduction to the basics of XML, we discuss namespaces, then document templates, after this we take a look at accessing data in XML documents and finally we discuss transformations of XML documents.

3.1 XML basics

In this section we describe the fundamental structure of XML documents. Those who are already familiar to HTML will notice many similarities. Before we start, here is a simple example of an XML document. Can you pick up the format requirements?

```
<?xml version="1.0" encoding="UTF-8"?>
<folder>
  <email date='20 Aug 2003'>
    <from>robert@company.com</from>
    <to>oliver@company.com</to>
    <subject>Meeting</subject>
    Could we meet this week to discuss the interface problem
    in the NTL project? --Rob
  </email>
  <email date='21 Aug 2003'>
    <from>oliver@company.com</from>
    <to>rob@company.com</to>
    <cc>amy@company.com</cc>
    <subject>Re: Meeting</subject>
    On 20 Aug 2003 rob@company.com wrote
    &gt; Could we meet this week to discuss the interface problem
    &gt; in the NTL project? --Rob
    OK. What about today at 16:00?
  </email>
</folder>
```

3.1.1 Syntax

As we can see from the example above, when we convert data (e.g. emails) into XML, its structure is indicated by additional text (adhering certain rules), or in other words the original text is *marked up*. The first kind of text, that carries the data being represented, is called *character data* or *PCDATA* (parsed character data) and the second kind, delimited by < and > is the *markup*. The markup consists of *tags*, for instance the pair <cc> and </cc> form a tag, where the former is the starting tag and the latter is the ending tag.

The outermost tag is called the *root element* (in the example it is `folder`, the actual tags are formed by adding the <, > signs and prepending / for the end-tag). There must be exactly one root element in an XML document, into which all the other tags are nested. The nesting must be a *proper* one, in the sense that for each non-root element there is a unique element which contains it and which contains no other element that also contains it.

An XML document may also contain a prolog, which is just the text before the root element. The prolog is not part of the structured data, it usually contains compiler directives or self identification of the document.

Elements

Elements are the primary structuring units of XML documents. They are marked with start- and end-tags, where the name of the end tag must match

the name of the start tag with a '/' prepended. The element, as a logical unit, contains the start-tag, its content and the end-tag. Between the start- and end-tags, elements can contain any combination of character data and other elements.

The content of elements can be

element if the element contains only elements (e.g. `folder` in the previous example),

character if it contains only character data (e.g. `to` in the previous example),

mixed if it contains both (e.g. `email` in the previous example),

empty if it contains nothing (e.g. `<x></x>`).

Empty-content elements can be abbreviated by putting a slash before the > sign of their start-tag and omitting the end-tag (e.g. `<x/>`).

The relationships between the elements can be classified quite naturally.

Child element It is an element of another one in the first nesting level (e.g. both `email` elements are children of `folder`.

Parent element It is the reverse of the child relationship (e.g. `folder` is the parent of both `email` elements).

Sibling element These are elements with the same parent (e.g. the elements `from, to, cc, subject` inside an `email` element are siblings).

Descendant element It is an element in the transitive closure of the child relationship (e.g. any of the `to` elements is a descendant of `folder`).

Ancestor element It is an element in the transitive closure of the parent relationship (e.g. `folder` is an ancestor of any other elements, except itself, which is not surprising as it is the root element).

Please note that the parent is uniquely defined in consequence of the proper nesting we required from XML documents.

Names for elements can be chosen by the user according to the following rules.

- Names are taken case sensitive.
- Names cannot contain spaces.
- Names starting with "xml" (in any case combination) are reserved for standardization.
- Names can only start with letters or with the '_', ':' characters.
- They can contain alphanumeric characters and the '_', '-', ':', '.' characters.

The colon character, although allowed in names, should only be used for namespaces (see Section 3.2).

Attributes

As the example at the beginning of the section shows, elements can also contain *attributes*, which are name-value pairs, listed in the start-tags of elements. Here are the rules for attribute insertion into elements.

- The naming rules of elements apply also for attributes.
- Elements can contain zero or more attributes.
- The names of the attributes must be unique within a start-tag.
- Attributes cannot appear in end-tags.
- Attribute values must be enclosed in single or double quotes.

Attributes can be resolved into elements and character data can be put into attributes (though might not make much sense in many cases). For instance in the example we could have written for the first email the following.

```
<email>
  <date>20 Aug 2003</date>
  <from>robert@company.com</from>
  <to>oliver@company.com</to>
  <subject>Meeting</subject>
  Could we meet this week to discuss the interface problem
  in the NTL project? --Rob
</email>
```

Or, for instance, the second email could have been formed as follows.

```
<email date='21 Aug 2003' from='oliver@company.com'
  to='rob@company.com' cc='amy@company.com'>
  <subject>Re: Meeting</subject>
  On 20 Aug 2003 rob@company.com wrote
  &gt; Could we meet today to discuss the interface problem
  &gt; in the NTL project? --Rob
  OK. What about today at 16:00?
</email>
```

There are no strict rules of when to use attributes and when elements so it is up to the style of the user to decide in this question. As a generic rule one can put those data into attributes which are not important for most applications (or humans).

Additional XML syntax

Since characters like < and & have special meaning in XML, they cannot appear directly in character data. Whenever such a special character has to be included anyway, one has to substitute it with the corresponding *entity reference* that starts with a & and ends in a semicolon. The next table summarizes the entity references which can be used to substitute special characters.

Character	Entity reference
<	<
>	>
"	"
'	'
&	&

In attribute values the same kind of quotation mark as the one used as the delimiter must be replaced with the corresponding entity reference. For example the following are equally valid attribute settings.

```
<player name='Thomas "Giant" Stevens' />
<player name="Thomas &quot;Giant&quot; Stevens" />
```

Comments can be included in the XML document anywhere outside other markup with the following syntax.

```
<!-- Comment text comes here. -->
```

Here `<!--` and `-->` are the delimiters of the comment and obviously, the text of the comment should not contain the character sequence of the ending delimiter.

The first line of the example (`<?xml version="1.0" encoding="UTF-8"?>`) is the XML declaration, which is not required, however it is recommended to include it. Currently (summer 2003) only the 1.0 version of XML is in effect so that attribute is fixed. The encoding specifies the usual ASCII one (with "UTF-16" it is the Unicode).

The prolog may also include processing instructions that certain applications might interpret. For instance the line

```
<?xml-stylesheet href="my.css" type="text/css"?>
```

will tell for a web-browser to apply the `my.css` stylesheet for formatting the XML document when it displays it.

A *CDATA* section can be used, outside other markup, to include text literally (special characters like <, & are not recognized) into XML documents. This comes good when one wants to insert larger text blocks and does not want to replace all special characters with their corresponding entity references. The starting string is `<![CDATA[` and the enclosing is `]]>`. For example one could use it to include HTML code into XML documents.

```
<example>
<![CDATA[
  <html>
    <head><title>My first web-page</title></head>
    <body><p>Hello world!</p></body>
  </html>
]]>
</example>
```

Obviously, the `]]>` character sequence should not appear in the included text.

To summarize the most important requirements for well formed XML documents we recall the following.

- XML elements are marked with start- and end-tags whose names must match.
- There can be only one root element.
- The elements must be properly nested into each other.
- Elements can have attributes to store additional character data.
- We have to obey certain naming rules for elements and attributes.
- Special characters must be replaced by entity references, except inside a CDATA block.

3.1.2 XHTML

As there are many excellent introductions to XHTML and HTML available on the Internet, it is recommended that the reader gets the information from there. The official (though somewhat dry) specifications can be found at `http://www.w3.org/TR/xhtml1/` and `http://www.w3.org/TR/html4/`.

3.2 Namespaces

Namespaces are useful in solving the following complications. First of all, with their help naming conflicts can be resolved. For instance if an XML document contains data about publishers and authors and both has a describing attribute: name, which appears as an element with tag `name` in the document. The instances

```
<name>Springer</name>
<name>Donald E. Knuth</name>
```

would describe entities of different categories, while this fact would be very difficult for a computer program to recognize.

The ad-hoc solution in this example would be to use different tags for authors and publishers. However, in general this might be not so easy, for instance if the document has to be created by merging the XML document of publishers and one of authors, ad-hoc name changes of tags might be undesirable.

Namespaces are a standard way to solve this problem. In this example, the documents of the publishers and of the authors should have unique namespaces declared in them. These can be thought of as prefixes to be prepended for names in the document. Then merging the two files is no problem anymore, because the different prefixes will identify to which category the element belongs.

The other problem in which namespaces can help is to divide the structured document further into application-specific groups. If the data the applications need from the document are separable, it is advantageous to indicate which data is used by which application via putting the corresponding tag-names into different namespaces.

3.2.1 Uniform Resource Identifiers

Before we go into the details of XML namespaces, let us shortly discuss URIs (Uniform Resource Identifiers) and URLs (Uniform Resource Locators) as they will often be used in namespace definitions.

URIs are character sequences with restricted syntax to reference things with identity (e.g. `mailto:gbodnar@risc.uni-linz.ac.at`). URLs are a special subclass of URIs, the ones that identify a resource via a representation of their primary access mechanism (e.g. `http://www.risc.uni-linz.ac.at/education/courses/`).

We distinguish absolute and relative URIs. Absolute URIs refer to the resource independent of the context in which the identifier is used. Relative URIs describe the difference between the current context and the absolute identifier of the resource in a hierarchical namespace.

Example 3.1. Take for instance the URL `http://www.risc.uni-linz.ac.at` which is an absolute identifier for the main homepage of RISC. From any webpage over the Internet, if a link uses this URL as a pointer, a click on it will take the browser to the homepage of RISC. The absolute URL for the page with the list of RISC related courses in the current semester is `http://www.risc.uni-linz.ac.at/education/courses/`.

Within the RISC home page the link "Education" in the middle column uses the relative URL `/education/courses/` (actually the implementation uses only `/education/`, which is redirected to the above given URL). This describes how to get the target URL from the URL of the current page, namely by appending it to current URL. If this relative URL appears on another homepage, it will point to the corresponding location within that site.

Please note that if you move the cursor over the link the browser usually displays the appended absolute URL in the status bar (viewing the page source reveals the used URLs).

To determine an absolute URI from a relative one requires a context, which is called the *base URI*. There are the following methods to do this

- The base URI may be embedded into the document so that relative URIs in the document have the context explicitly set.
- If not in the previous case, the base URI can be determined by the retrieval context of the entity (resource) of one encapsulation level higher (e.g. the resource is a table which appears in a web page).
- If not in the previous cases, i.e. the resource is not encapsulated into another entity, the last URI, used for retrieving the document which contains the current URI, is used as a base URI (e.g. the case of the `/education/courses/` example).
- If none of the above conditions apply, the base URI is defined by the context of the application (in this case different applications can interpret the same URI differently).

URIs can have up to four components: scheme, authority, path, query. Each component has a rigorous syntax which it has to obey, but we will not present

these rules in full detail here (the interested reader is advised to consult the RFC 2396 document).

The *scheme* component defines the semantics for the rest of the string of the URI. The scheme component starts with a lower case letter and is separated from the rest of the URI by a colon. Relative URIs are distinguished from absolute ones in that they do not have a scheme component. An example is the `http` character sequence in the URLs that indicates the scheme for "hypertext transfer protocol" used on the WWW.

The *authority* component defines the top hierarchical element in the namespace. It is typically defined by an Internet based server in the form

```
[userinfo@]host[:port]}
```

a few examples follow.

```
gbodnar@risc.uni-linz.ac.at
secure.access.com:443
193.170.37.129
```

The *path* component contains data specific to the given authority and/or scheme, identifying the resource within their context. The hierarchical structure is indicated by '/' signs, which separate path segments from each other. In Example 3.1 the relative URL `/education/courses/` is actually a path component that appears also in the corresponding absolute URL.

The *query* component is a string to be interpreted by the resource. It is separated from the path component with a question mark. A natural usage of query components is for instance encoding data filled in by the user at a web form, and passing it as the set of input parameters to the program which is supposed to process them. The example is an input query for the LEO online dictionary.

```
http://dict.leo.org/?search=namespace&lang=en
```

In this example the so called"GET" method is applied for parameter passing. In this standard the parameters are listed in the query string as "attribute=value" pairs separated by ampersands.

3.2.2 Namespaces in XML

The most important aspect of a namespaces is uniqueness (in order to avoid collisions). So namespaces must be identified with a URI, preferable with one to which the owner of the document has rights.

Example 3.2. For instance, if a company owns the `http://big.company.com/` domain name, it is preferable that namespaces start with this URL for XML documents created at this company. So `http://big.company.com/product-info` could be a URI used in a namespace declaration, where the corresponding namespace would be used in naming elements in XML documents which that describe products of the company.

Please note that namespaces need not correspond to existing URLs, they are simply strings of characters which are supposedly unique and additionally might carry some self explanatory information on the purpose of the namespace.

Of course, uniqueness can be guaranteed only if the company owns the domain name and within the company the URIs are maintained centrally so that no two developer will invent and use the same URI.

Namespaces can be declared within any element of an XML document, using the reserved attribute name `xmlns`. For instance

```
<list xmlns:p="http://big.company.com/product-info">
  <p:product>
    <p:name>Disney Mouse Pad</p:name>
    <p:id>231069948</p:id>
    <p:price>3.49 EUR</p:price>
  </p:product>
</list>
```

declares that inside the `list` element the prefix `p` (the string between `xmlns:` and the equation sign) will be used as an alias for the namespace `http://big.company.com/product-info`. Prefixes are to simplify the notation in an XML document, as it would be cumbersome and unreadable to apply the long URI each time.

Whenever a prefix is given in a namespace declaration, all descendants of the element which belong to the namespace must be named with the prefix prepended and separated by a colon from the tag name (e.g. `p:id`). This is also called *qualifying* the name (e.g. `id` is qualified to be in the namespace with alias `p`) and the name with a prefix is called a *qualified name*. Both start- and end-tag names have to be qualified (consistently). Attribute names can also be qualified.

If no prefix is given, so that the attribute is just `xmlns`, in a namespace declaration, it will declare the *default namespace* for the element. In all descendants of the element unqualified tag names will automatically fall into the default namespace. Therefore the list of products example above can also be written as follows.

```
<list xmlns="http://big.company.com/product-info">
  <product>
    <name>Disney Mouse Pad</name>
    <id>231069948</id>
    <price>3.49 EUR</price>
  </product>
</list>
```

However there is a difference between the two ways: In the first case the `list` element is not in the namespace while in the second it is.

Default namespace declarations are available to the element in which the declaration takes place and to its descendants. With a prefixed namespace declaration the element in which the declaration takes place belongs to the namespace only if its name is qualified with the prefix we define. So an equivalent qualified version of the previous default namespace declaration example is the following.

```
<p:list xmlns:p="http://big.company.com/product-info">
  <p:product>
    <p:name>Disney Mouse Pad</p:name>
```

```
    <p:id>231069948</p:id>
    <p:price>3.49 EUR</p:price>
  </p:product>
</p:list>
```

The section of the XML document to which the namespace declaration applies is called its *scope*. If one uses a qualified name outside the scope of the corresponding namespace, the qualification will be simply ineffective. For instance, in the following XML document the second book element is not in the namespace `http://www.mybookstore.com/catalog`, the string `b:book` in that case stands just for itself.

```
<catalog>
  <b:book xmlns:b="http://www.mybookstore.com/catalog"
      isbn="0-321-19784-4">
    <b:author>C. J. Date</b:author>
    <b:title>An Introduction to Database Systems</b:title>
  </b:book>
  <b:book isbn="0-8129-9191-5">
    <b:author>G. Brill</b:author>
    <b:title>Codenotes for XML</b:title>
  </b:book>
</catalog>
```

Remark 3.3. Attribute names are not automatically in the namespace of the including elements. For instance in the example above the `isbn` attribute is not a member of the declared namespace.

Prefixes used as aliases for namespaces can be reassigned in an XML document, however it is not advised to do so in order to keep the document as readable as possible.

3.3 XML Schema

XML schemas are templates for XML documents. With XML schemas one is allowed to specify what structures the modeled XML documents may have and impose restrictions on their contents. XML schemas are XML documents themselves that must follow strict syntactical rules, defined in the Schema Primer Recommendation, available at `http://www.w3.org/TR/xmlschema-0/`. This specification can be considered as a schema for schemas.

This concept replaces the one of Document Type Definition (DTD), which was used originally for the same purpose. There are several advantages of schemas over DTDs.

- While DTD syntax was inconsistent with XML, schemas are valid XML documents themselves, so there is no additional parser necessary in XML applications.
- XML schemas provide a wide range of data types (integers, dates, strings, etc.) for element content and attribute values.
- Schemas allow the users to define custom data types, and support inheritance between types.

- Schemas also provide powerful features to express element groupings, and other properties of elements (uniqueness, substitutability, etc.).

In brief, schemas provide a flexible grammar to describe XML document templates in an XML compliant way. There are also several software products available to validate schema definitions and XML documents against well defined schemas.

3.3.1 Schema declaration

Schema definitions apply namespaces to reference constructions in the schema grammar and to declare new element names for some target namespace. The root element of an XML schema must always be `schema`, which should be qualified to be in the namespace `http://www.w3.org/2001/XMLSchema`. This namespace contains the vocabulary (set of predefined names) for schema definitions.

The `schema` element can also include the attribute `targetNamespace`, whose value defines the namespace for the new names (of elements, attributes, types) declared in the schema. This is not a namespace declaration for the schema itself, but it will force the referencing document to use this namespace for the names defined in this schema.

If no `targetNamespace` is defined, the definitions and declarations of the schema will validate elements in the instance document that do not fall into any namespace (i.e. they are unqualified and they are outside the scope of any default namespace declarations). XML documents need not use namespaces, so if we want to write a schema that validates such an XML document we must omit the target namespace definition.

In the `schema` element, several additional namespaces (e.g. a default namespace) can be defined. They can be used to qualify names placing them into the desired namespace. This is particularly useful when user defined types appear in element or attribute declarations (we will discuss this later).

The target namespace applies by default only to global objects. Global elements, attributes and types are declared in elements of the schema which are children of the `schema` element. The declarations in deeper nesting levels are called local declarations (or, when we talk about the corresponding elements/attributes in the instance document, simply local elements/attributes).

Whether local elements/attributes must or must not be qualified to be in the target namespace is controlled by two attributes of the `schema` element: `elementFormDefault` and `attributeFormDefault`. Their values can be `"unqualified"` (default) or `"qualified"`. In the local element/attribute declarations the values of the `elementFormDefault` and `attributeFormDefault` attributes (whose scope is the whole schema) can be overridden using the `form` attribute.

Example 3.4. To declare an XML schema for the namespace `http://big.company.com/product-info` with element and attribute qualifications enforced one can write the following.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://big.company.com/product-info"
            elementFormDefault="qualified"
            attributeFormDefault="qualified">
```

```
  <!--Schema definition comes here.-->
</xsd:schema>
```

Please note that the `http://www.w3.org/2001/XMLSchema` namespace is assigned to the `xsd` prefix, and thus the `schema` element has to be appropriately qualified to be in the predefined namespace of schema definition grammar. The last two attributes imply that all local elements and attributes declared this schema will have to be qualified to be in the given target namespace in an XML document referencing the schema, unless they have the following attribute in their declaration: `form="unqualified"`.

An XML document that references a schema is called an *instance* of the schema. An instance can only use the vocabulary (set of declared names) defined by the schema. A schema reference in an instance document is typically placed into the root element.

To properly reference a schema one has to do three things: first to declare the target namespace of the schema in the instance (except if there is no target namespace defined in the schema), so that it is available for further elements. Secondly one has to define the schema instance namespace, which is the predefined namespace `http://www.w3.org/2001/XMLSchema-instance`, to be able to do the actual referencing. And the last action is referencing the schema whose instance this document is; for this the `schemaLocation` attribute (which belongs to the schema instance namespace) has to be assigned the URI of the schema.

Example 3.5. Let us assume that the schema of Example 3.4 has the URI `http://big.company.com/schemas/product-info.xsd`. Then an XML document that references the schema can start as follows.

```
<list xmlns="http://big.company.com/product-info"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://big.company.com/schemas/product-info.xsd">
  <!--Some content.-->
</list>
```

The default namespace declaration uses the target namespace of the referenced schema. The `xsi` prefix will qualify "reserved" names, for instance, to bind the schema to the document by the attribute `xsi:schemaLocation`.

3.3.2 Types, Elements, Attributes

The main part of XML schemas define what elements and attributes can an instance have and what types of data are allowed to be contained by them. First we discuss elements, then attributes and finally types.

The most trivial elements are the simple content ones, which can contain only character data, possibly in some prescribed format, and they cannot contain child elements. This is an example of a simple element declaration where the tag-name that identifies the element must be `simple` and the content is some text.

```
<xsd:element name="simple" type="xsd:string"/>
```

Please note that it is also assumed that the schema definition is as in Example 3.4, so `xsd` prefix is bound to the schema definition namespace.

The `element` tag is defined in the `xsd` namespace and denotes that an element definition follows. The `name` attribute describes the name of the element being declared and the `type` attribute its data type which is also identified with a qualified name (i.e. string is defined in the meta-schema).

Complex element definitions allow us to specify the internal structure of the declared elements. We will use the following XML schema constructions:

sequence This requires that the instance contains the elements that appear in the schema in the same order and by default exactly once.

all This allows that elements in the definition group appear in any order and by default exactly once. The `all` element can occur only in global declarations and only as a single element group declarator. The children of a `all` element should be individual elements (no element groups are allowed).

choice This allows only one of the elements in the definition group to appear in an instance document inside the complex element.

The following two attributes can appear in local element declarations and control the number of occurrence of the declared element in the instance document:

minOccurs Specifies the minimal number of occurrences of the declared element in its parent element.

maxOccurs Specifies the maximal number of occurrences of the declared element in its parent element.

Optional element can be prescribed via the attribute `minOccurs="0"`, and elements that can occur arbitrarily many times can be prescribed via the attribute `maxOccurs="unbounded"`.

Example 3.6. The following definition (schema fragment) prescribes a fixed structure for the `product` element.

```
<xsd:element name="product">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="id" type="xsd:unsignedInt"/>
      <xsd:element name="price" type="xsd:float"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

An example instance of this element definition can be taken from Section 3.2 with the slight modification of dropping the “EUR” characters from the `price` element.

```
<product>
  <name>Disney Mouse Pad</name>
  <id>231069948</id>
  <price>3.49</price>
</product>
```

In the next example each child element must appear exactly once, but the order is arbitrary.

```
<xsd:element name="product_extension">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="made_in" type="xsd:string"/>
      <xsd:element name="oem" type="xsd:boolean"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>
```

An example instance of this element can be the following.

```
<product_extension>
  <oem>true</oem>
  <made_in>Japan</made_in>
</product_extension>
```

In the following example the occurrence constraints on child elements are prescribed explicitly.

```
<xsd:element name="product_sold_in">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="country" type="xsd:string"
                   minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

An example instance of this element can be the following.

```
<product_sold_in>
  <country>Germany</country>
  <country>United Kingdom</country>
</product_sold_in>
```

With the `group` element we can define element groups on the global level and we can refer to them in other declarations.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="payment">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="customer" type="xsd:string"/>
      <xsd:element name="address" type="xsd:string"/>
      <xsd:group ref="payment_method"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:group name="payment_method">
```

```
  <xsd:choice>
    <xsd:element name="credit_card" type="xsd:string"/>
    <xsd:element name="bank_transer" type="xsd:string"/>
  </xsd:choice>
</xsd:group>
</xsd:schema>
```

A valid instances is

```
<payment>
  <customer>K. Boehm</customer>
  <address>Goethestr 5, Linz</address>
  <credit_card>VISA1208549856745634/0508</credit_card>
</payment>
```

and another is

```
<payment>
  <customer>K. Boehm</customer>
  <address>Goethestr 5, Linz</address>
  <bank_transfer>IBAN57486739485934590000/34660</bank_transfer>
</payment>
```

So far the elements were either simple or complex with only child element content. One can also declare an element with mixed content using the `mixed=true` attribute setting in the declaration. For instance, the following declaration allows that the element `note` contains arbitrarily many `emp` elements and character data in between.

```
<xsd:element name="note">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="emp" type="xsd:string"
                   minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The following element is then an instance of this scheme.

```
<note>The extra lecture will be <emp>tomorrow at 16:00</emp>
in the lecture hall <emp>HS 13</emp>.</note>
```

The `nillable` attribute can be used to declare that an instance of the element can have a special attribute `nil` (from the schema instance namespace) which marks explicitly that the instance has no meaningful value (similarly to the NULL values in RDBMSes). The following example declares that the `date` element is nillable.

```
<xsd:element name="date" type="xsd:date" nillable="true"/>
```

Then the following instance is valid.

```
<date xsi:nil="true"></date>
```

Attributes are declared similarly to child elements with the difference that attributes always have simple content. In an attribute definition the `name` and `type` attributes will prescribe the corresponding qualities of the attribute being declared. The `use` attribute controls requirements for the declared attribute and the `value` attribute can set default or fixed values for it. The `use` attribute can have the following values:

required The declared attribute must be explicitly defined every time its element occurs in the instance of the schema.

default The declared attribute is optional and if it is not given the `value` attribute specifies its value.

optional The declared attribute is optional, no default is given.

fixed The declared attribute is optional, but if it appears, its value has to be the one given in the `value` attribute.

prohibited The declared attribute must not appear in any occurrence of the corresponding element and it has no predefined value.

Attributes can be defined only inside complex element declarations, so if we want to define a simple element with attributes, we have to define it as a complex element with simple content using the `simpleContent` element. The attribute declarations always come after the child element declarations.

Example 3.7. In the first declaration of Example 3.6 we could define the currency for the price tag as an attribute as follows.

```
<xsd:element name="product">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="id" type="xsd:unsignedInt"/>
      <xsd:element name="price">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:float">
              <xsd:attribute name="currency" type="xsd:string"
                             use="required"/>
            </xsd:extension>
          </xsd:simpleContent>
        </xsd:complexType>
     </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

An example instance of this element can be taken from Example 3.6 and now we can (actually we have to) add the currency as an attribute of `price`.

```
  <product>
    <name>Disney Mouse Pad</name>
```

```
    <id>231069948</id>
    <price currency="EUR">3.49</price>
  </product>
```

Or using Example 3.6, we include a time stamp attribute for the `product_extension` element.

```
<xsd:element name="product_extension">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="made_in" type="xsd:string"/>
      <xsd:element name="oem" type="xsd:boolean"/>
    </xsd:all>
    <xds:attribute name="time_stamp" type="xsd:dateTime"
                   use="optional"/>
  </xsd:complexType>
</xsd:element>
```

Reusing the corresponding instance from Example 3.6 we can have the following elements.

```
  <product_extension>
    <oem>true</oem>
  </product_extension>
  <product_extension time_stamp="2003-05-30T13:20:00.000+01:00">
    <oem>false</oem>
    <made_in>Japan</made_in>
  </product_extension>
```

In local declarations we can refer to global ones via the `ref` attribute. This is just to say that the definition of the element should be taken as the definition of the referenced one. This makes schema declarations more readable, modularizing the definitions.

For instance, we could have taken out the definition of the `price` element in the first declaration of Example 3.7 to the global level and then in the declaration of `product` we could have written the following.

```
      ...
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="id" type="xsd:unsignedInt"/>
      <xsd:element ref="price"/>
      ...
```

Alternatively, we can define a new type on the global level, say `price_type`, and use it as we use the standard types in the element declarations. In this case, if a target namespace is set, `price_type` will become a member of the target namespace, thus we have to find means to reach it when we refer to it in the element declaration. This can be done by defining an alias with the same namespace as the target namespace. The whole schema looks like:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://big.company.com/product-info"
  xmlns:p="http://big.company.com/product-info">
```

```
 <xsd:element name="product">
   <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="name" type="xsd:string"/>
           <xsd:element name="id" type="xsd:unsignedInt"/>
           <xsd:element name="price" type="p:price_type"/>
         </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="price_type">
   <xsd:simpleContent>
         <xsd:extension base="xsd:float">
           <xsd:attribute name="currency" type="xsd:string"
                                          use="required"/>
         </xsd:extension>
   </xsd:simpleContent>
 </xsd:complexType>
</xsd:schema>
```

Here is an instance of this schema.

```
<p:product xmlns:p="http://big.company.com/product-info">
  <name>Disney Mouse Pad</name>
  <id>231069948</id>
  <price currency="EUR">3.49</price>
</p:product>
```

Think it over why certain elements are qualified and others are not.

XML Schema supports modularity by enabling a schema to include other schemas using the `include` element.

```
<xsd:include schemaLocation="http://big.company.com/schemas/s1.xsd"/>
```

This will include the schema definitions in the file locatable by the given URI into the current schema. It is required that the target namespace of the included and including schemas be the same.

It is also possible to use named types from other schemas having target namespaces other than the target namespace of the current schema. In this case we have to assign an alias to the target namespace of the imported schema and use the `import` element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://big.company.com/product-info"
  xmlns:html="http://www.w3.org/1999/xhtml"
  xmlns:e="http://big.company.com/employee-info">

  <xsd:import namespace="http://www.w3.org/1999/xhtml"/>
  <xsd:import namespace="http://big.company.com/employee-info"
         schemaLocation="http://big.company.com/schemas/e.xsd"/>

  <!-- Rest of the schema. -->
</xsd:schema>
```

The declarations of the imported schemas are accessible via the aliases of corresponding namespaces.

XML schema provides more than forty built-in data types, from which we have already seen a few above (the complete list is available under the URL at the beginning of this section). We also created new types in an anonymous way so that they could be used only at the place of definition. Moreover we have seen how XML schema allows to introduce new named types that can be used as the built-in ones after their definition.

Example 3.8. Here is another example of using named type. From the first declaration of Example 3.6 we can extract the complex type definition as follows.

```
<xsd:complexType name="product_type">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="id" type="xsd:unsignedInt"/>
    <xsd:element name="price" type="xsd:float"/>
  </xsd:sequence>
</xsd:complexType>
```

And then the declaration of the product element could simply be

```
<xsd:element name="product" type="product_type"/>
```

The main mechanisms to define new data types are restriction and extension. Built-in data types provide one or more "facets" through which one can create new types by restriction. An example can be the type of strings that match a given regular expression. The following example restricts the strings of the new type to be of the format `nnn-nn-nnnn` where in place of an `n` any digit can stand (e.g. 323-90-0982).

```
<xsd:simpleType name="serial_number">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-\d{2}-\d{4}"/>
  </xsd:restriction>
</xsd:simpleType>
```

The complete list of facets for each built-in type can be found at `http://www.w3.org/TR/xmlschema-0/`.

For complex data types one can also use extension to add new elements and attributes to an already existing type. This example extends the `product_type` declaration, from Example 3.8, with the elements of `product_extension`.

```
<xsd:complexType name="extended_product_type">
  <xsd:complexContent>
    <xsd:extension base="product_type">
      <xsd:sequence>
        <xsd:element name="made_in" type="xsd:string"/>
        <xsd:element name="oem" type="xsd:boolean"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

As in the relational model we might want to ensure that certain elements or attributes have unique values within some scope in an instance of the XML schema. This can be achieved by using the `unique` tag in the element definition which must come after all child and attribute definitions. The general structure of a `unique` tag is as follows.

```
<unique name="dummy">
  <selector xpath="element_set_selection"/>
  <field xpath="uniqe_field_selection">
</unique>
```

The name attribute is only formal, the `xpath` attribute of the `selector` element specifies the set of elements within which the element or attribute specified by the value of the `xpath` attribute of the `field` element must be unique. The identification uses the XPath specifications, which we will discuss in Section 3.4. The field specification is relative to the path given in the selector. Attribute names must be prefixed with an @ character.

Example 3.9. The following schema fragment prescribes that in an instance of the schema within the `list` element the `product` elements will have to have a child `id` with unique value.

```
<xsd:element name="list">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="product">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="id" type="xsd:unsignedInt"/>
            <xsd:element name="price" type="xsd:float"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:unique name="unique_id_declaration">
    <xsd:selector xpath="product"/>
    <xsd:field xpath="id"/>
  </xsd:unique>
</xsd:element>
```

In the example we also assumed that the `xsd` prefix is assigned as in Example 3.4.

If we want to ensure referential integrity between contents of elements, that is we want that if certain values of attributes or elements match values of other attributes or elements in the instances of the schema, we can use the `key` and `keyref` tags. The mechanism is analogous to the one of the `unique` tag.

The key declarations must always occur at the end of an element declaration (after child element and attribute declarations). Elements or attributes which are declared to be keys must always be present with a unique value (which cannot be `nil`). Referring element or attribute values must always have a corresponding key value.

The construction parallels the notion of foreign keys in RDBMSes.

Example 3.10. In the following schema fragment we prescribe that the `attends` document can contain `student` and `course` elements with the structure given below. Then we declare that the `cid` element of the `course` elements must be a key in the attends elements and this key is referred by the `cid` elements of the `student` elements. This means that the `cid` elements in the `course` elements must be unique in an `attends` element and whenever a `cid` value appears in a `student` element, there must be a corresponding `course` element with the same value.

```
<xsd:element name="attends">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="student">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="sid" type="xsd:integer"/>
            <xsd:element name="cid" type="xsd:integer"
                         maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="course" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="cid" type="xsd:integer"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:key name="course_key">
    <xsd:selector xpath="course"/>
    <xsd:field xpath="cid"/>
  </xsd:key>
  <xsd:keyref name="course_ref" refer="course_key">
    <xsd:selector xpath="student"/>
    <xsd:field xpath="cid"/>
  </xsd:keyref>
</xsd:element>
```

3.3.3 Exercises

Exercise 3.3.1. Consider the following schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="member">
  <xsd:complexType>
```

```
   <xsd:all>
    <xsd:element name="bid">
     <xsd:complexType>
      <xsd:simpleContent>
       <xsd:extension base="xsd:integer">
        <xsd:attribute name="timestamp" type="xsd:dateTime"
                       use="required"/>
       </xsd:extension>
      </xsd:simpleContent>
     </xsd:complexType>
   </xsd:element>
   <xsd:element name="name" type="xsd:string" minOccurs="1"
                maxOccurs="1"/>
   </xsd:all>
   <xsd:attribute name="id" type="xsd:integer" use="required"/>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Find the points at which the following XML document violates the prescriptions of the previous schema.

```
<member id="A2342" item-id="3489873">
 <name>Thomas Keller</name>
 <bid timestamp="2003-12-30T18:20:35" currency="EUR">34</bid>
 <bid timestamp="2003-12-30T18:28:32" currency="EUR">36.5</bid>
</member>
```

Exercise 3.3.2. Modify the XML schema of Exercise 3.3.1 in a way that the example XML document becomes valid with respect to the modified schema.

Exercise 3.3.3. Consider the following XML document, with the intended restrictions that the `id` attribute of the `entry` element is mandatory, the `author`, `title`, `year` elements have to be present, where there can be more than one author of a publication. No namespaces are used.

```
<bib>
  <entry id="Cox:00">
    <author>
      <fname>David</fname>
      <lname>Cox</lname>
    </author>
    <title>Update on Toric Geometry</title>
    <year>2000</year>
  </entry>
  <entry id="Becker_Weispfenning:93">
    <author>
      <fname>Thomas</fname>
      <lname>Becker</lname>
    </author>
    <author>
      <fname>Volker</fname>
```

```
      <lname>Weispfenning</lname>
    </author>
    <title>Groebner bases</title>
    <publisher>Springer</publisher>
    <year>1993</year>
  </entry>
</bib>
```

Write an XML Schema for such kind of documents, that validates, in particular, this example.

Exercise 3.3.4. Consider the following XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="mdb">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="movie" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="subtitle" type="xsd:string" minOccurs="0"/>
       <xsd:element name="director" type="xsd:string"/>
       <xsd:element name="cast">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="role" maxOccurs="unbounded">
           <xsd:complexType>
            <xsd:sequence>
             <xsd:element name="name" type="xsd:string"/>
             <xsd:element name="actor" type="xsd:string" minOccurs="0"/>
             <xsd:element name="voice" type="xsd:string" minOccurs="0"/>
            </xsd:sequence>
           </xsd:complexType>
          </xsd:element>
         </xsd:sequence>
        </xsd:complexType>
       </xsd:element>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Invent an XML document which is valid with respect to this schema.

3.4 XPath

XPath is a part of the XSL (eXtensible Stylesheet Language) family. Its primary purpose is to provide common syntax and functionality to address parts of XML

documents. Beside this, it can also be used for basic manipulation of strings, numbers and booleans. XPath operates on the logical structure of an XML document and uses a syntax to address their parts that resembles to the path constructions in URLs.

XPath models an XML document as a tree of nodes, which can be imagined as an extension of the tree of elements of the XML document. In this extended tree not only the XML elements, but also their attributes, the character data they may contain, the namespaces and the processing instructions appear as nodes.

Beside the data model, another important concept of the language is the XPath expression. With expressions it is possible to compute objects (like strings, numbers, sets of nodes) from the data of XML documents. The advanced expressions of XPath rely on a library of functions, which is general enough to allow simple text and numerical data manipulation.

Expressions are always evaluated in some context, which is described by the context node, context size and context position (think of a list of nodes on which an expression is to be evaluated; the size of the list is the context size and when the expression is evaluated on the ith element on this list, the context position is i), variable bindings and namespace declarations.

XPath is prominently used in XSLT to access and apply simple transformations on data of XML documents, e.g. to convert them to HTML files, which can be then displayed in web browsers. XPath expressions can appear as attribute values in XML documents, in this case the special characters must be substituted by entity references.

3.4.1 Data Model

The nodes in the tree of the XPath data model can be of the following types: root, element, text, attribute, namespace, processing instruction, comment. Each node represents a part of the text of the underlying XML document. The nodes can be ordered by the document order, in which a node precedes another if the first character that belongs to that node comes before the first character of the other node in the XML document. The nodes in the data model do not share children, moreover, every node (except the root node) has exactly one parent node which is an element node or the root node.

Remark 3.11. Please note that the root node is the first one in the document order, and that an element and its attributes always precede the descendants of the element. Namespace nodes precede the attribute nodes by definition.

Every node has an associated string value which can be computed in the following ways. The string value of the root node is the concatenation of all the string values of its descendants in document order.

Every XML element has an element node in the XPath model. The string value of an element node is the concatenation of the of all text node descendants in document order.

An attribute node in the XPath data model has the element node corresponding to the XML element in which the attribute is defined as its parent. However the attribute node is not considered as a child of its parent element. The string value of an attribute is simply its value as a string (after some nor-

malization specified by the XML standard). Attribute that declare namespaces do not appear as attribute nodes, but as namespace nodes.

Every element gets assigned a set of namespace nodes, one for each namespace declared in the element itself or in ancestors, in whose scope the element is. The element node is the parent of the namespace nodes, but just like attributes, the namespace nodes are not considered as children of the element. The string value of namespace nodes is the URI bound to the prefix identifying the namespace.

Remark 3.12. Why is an attribute or a namespace node not a child of its parent? The purpose of this distinction is to preserve, by default, the structure of the modeled XML document (where only elements can be nested). The default relationship is the child-parent and several other derived relationships (e.g. descendant-ancestor). The relationships that belong together in this sense will be collected in an "axis" of path directions.

How can then the attributes or namespaces that belong to an element be retrieved? By considering the corresponding relationship, e.g. attribute-parent, which will have its own axis.

Text nodes represent character data in the XML document in a way that text nodes do not have immediate text node siblings (i.e. a text node contains as much character data as it can) and they contain at least one character. The string value of a text node is just the character data contained by it. Characters inside comments, processing instructions and attribute values are not represented by text nodes. They can be retrieved as string values of the corresponding node types.

The string value of a comment node is the comment it contains (without the '`<!--`', '`-->`' symbols). Please see Example 3.14 for a node tree of an XML document.

3.4.2 Location Paths

Location paths are special expressions for selecting a set of nodes, possibly but not necessarily relative to the context node. A location path consists of location steps composed together from left to right, separated by '/' (slashes). The first location step selects a set of nodes relative to the context node. Then each resulting node is used as a context node for the next location step which results again a set of nodes, and so on. If the first character of a location path is '/', the initial context node will be the root node. In this case the location path is called absolute, otherwise it is relative.

Example 3.13. A first example could be the way one navigates in a hierarchically organized directory structure on a file system (the analogy is so natural that even the syntax is taken over in the abbreviated notation of path expressions). The following location path specification selects all the `summary` elements which are children of any element in the `reports` child of the parent node of the context node. The context node can be imagined as the current working directory if we use the file system analogy.

```
../reports/*/summary
```

On a UNIX file system this would correspond to selecting all the `summary` files in any subdirectory of the `reports` directory of the parent directory.

Please note that there is still an important difference between file systems and XML documents: in a file system a directory cannot contain more than one subdirectories or files with the same name, however, an XML element can have many children with the same name. So while the `../reports` specification selects at most one subdirectory on a file system, the same selects every `reports` children of the parent of the context node when considered as a location path for an XML document.

Location steps have the following parts:

axis It specifies the (in-tree) relationship between the context node and the nodes selected by the location step.

node test Specifies the node type for the nodes selected by the location step (it is separated by `::` from the axis).

predicates It specifies further expressions with boolean value, to refine the selected node set. This part can be omitted.

The following axes are available: child, descendant, parent, ancestor, following-sibling, preceding-sibling, following, preceding, attribute, namespace, self, descendant-or-self, ancestor-or-self. Most of the names are self explaining, e.g. the 'descendant' axis contains the descendants of the context node (recall that attribute and namespace nodes are excluded), the 'following-sibling' axis contains all the following siblings of the context node (with higher context positions). The 'attribute' axis contains the attributes of the context node.

The 'following' (resp. 'preceding') axis contains all nodes in the same XML document that come after (resp. before) the context node in the document order (obtained by left to right depth first search) excluding descendants (resp. ancestors) and attribute and namespace nodes. The 'following-sibling', 'preceding-sibling', 'attribute' and 'namespace' axes are empty for non-element nodes. The 'self' axis contains the context node itself.

There are three principal node types for axes. If the axis can contain elements, the principal node type is 'element'. For the 'attribute' and 'namespace' axes the types are 'attribute' and 'namespace' respectively, and for other axes it is 'element' again. Only those nodes pass a node test which have the same principal node type as the axis preceding the node test. One can specify exact matching of node names for a given one by adding it after the axis specifications separated by '::'. For example `child::x` will select all the children nodes of the context node with name `x`. The node test denoted as `*` selects all nodes on the given axis that match the principal node type of the axis. For instance `child::*` selects all element children of the context node (because the principal node type of the 'child' axis is 'element'). The `text()` node test selects text data nodes on the given axis.

Example 3.14. Let us consider the example XML document with emails at the beginning of Section 3.1. Figure 3.1 on page 93 illustrates a part of the node tree of the document (the parsing direction is clockwise/top-down instead of left to right). Namespace nodes are omitted to simplify the figure.

The absolute path

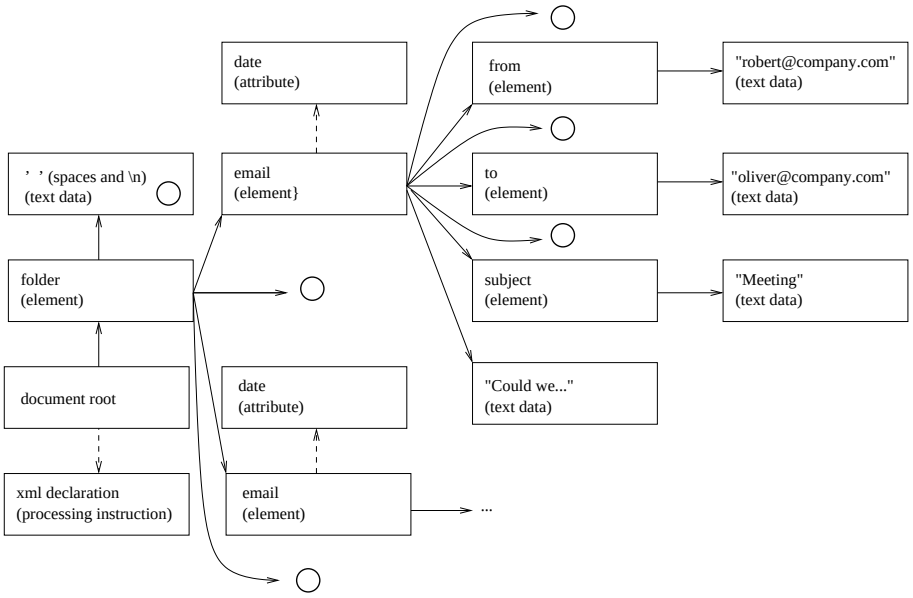


Figure 3.1: A simplified XPath data model

```
/child::folder/child::email/attribute::*
```

selects all the `date` attributes of all the `email` elements. The first '/' selects the root node as the context node then the first path element selects the child axis and the node test selects the `folder` child of the root. Then again the child axis is selected with all the `email` children. The third path element selects the attribute axis and the node test selects all attributes of the context node elements.

The 'ancestor', 'ancestor-or-self', 'preceding' and 'preceding-sibling' axes are called reverse axes (as they contain nodes which are before the context node in the document order); the others are forward axes. The proximity position of a member of a node set with respect to an axis is its position in the node set considered as an ordered list in the document oder for forward axes and in the reverse document order for reverse axes. The counting starts at one.

Predicates filter a node set with respect to an axis in a way that the predicate expression is evaluated for each node in the node set as context node, the size of the node set as context size and the proximity position of the node with respect to the axis as context position. The new node set, filtered by the predicate, contains the nodes resulting true in this evaluation procedure. Predicate expressions can be attached to the path element by enclosing them in '[', ']'.

For instance, in Example 3.14, the location path

```
/child::folder/child::*[position()=1]/attribute::*
```

selects only the attribute of the first `email` element in the XML document.

There is an abbreviated syntax available for location paths; some of the most important elements are listed in Table 3.1

Abbreviation	Meaning
	by default the `child::` prefix can be omitted
`*`	stands for 'for all'
`@*`	selects all attributes of the context node
`@name`	selects the `name` attribute of the context node
`.`	selects the context node
`..`	selects the parent of the context node
`/`	selects the root node
`//`	selects as `descendant-or-self`

Table 3.1: XPath abbreviations

Example 3.15. The following location path selects the `from` elements of the `email` elements for which the `date` attribute is `21 Aug 2003`. We use here the abbreviated notation.

```
/folder/email[@date = '21 Aug 2003']/from
```

3.4.3 Expressions, Core Functions

The simplest expressions are numerical, string literals, variable references and function calls. Numbers mean double precision IEEE floating point numbers, and string literals define themselves with the possibility of using special encodings (like Unicode). Variables can be assigned using other XML technologies, for instance XSLT, and they differ from variables in usual programming languages in that we cannot change freely their value after the assignment. The value the variable `x` holds can be referenced by `$x`.

The basic arithmetic operations are available for numbers. But please note that usually the operators should be separated from the names by whitespaces. For example `x-y` evaluates to the node set that contains the children of the context node with this name. On the other hand, `x - y` will take the first child of the context node with name `x` and the first one with name `y` and will attempt to convert their string value to numbers and evaluates to the difference of these numbers.

More complicated expressions are location paths (discussed in Section 3.4.2) and boolean expressions. The latter are expressions that evaluate to true or false. The basic boolean expressions contain comparisons $=, !=, >, <, <=, >=$ with the usual semantics for non-node set operands (where strings can be converted to numerical types and numerical types to booleans as needed). If at least one of the operands is a node set then a comparison expression is true if there exists at least one node in the node set whose string value (perhaps after the necessary conversion) satisfies the predicate. More complicated formulas can be formed from atomic logical expressions using the '`and`' and '`or`' connectives.

Function calls consist of a function name, which identifies a function in the function library determined by the expression evaluation context, and a parameter list whose elements can be converted to the type expected by the function at the given position of the list.

The core function library provides a minimal set of function specifications that has to be supported by any XPath implementation. Here we summarize

only a small subset of this library (first comes the type of the value the function returns, then the name of the function):

Node set functions

- `number last()` returns the context size from the expression evaluation context.
- `number position()` returns the context position from the expression evaluation context.
- `node-set id(object)` if the argument is a node set then it returns with union of this `id` function applied to the string value of each element of the set. If the argument is of other type, it is converted to a string, tokenized by whitespace characters, and the function returns with the node set containing the elements of the same document as the context node that have unique IDs given in the list of tokens.

String functions

- `sting string(object?)` converts an object to a string. The default argument is the node set in the context of the expression evaluation. A node set is converted into a string by returning the string value of the first element of the set in document order. The objects of type number and boolean are converted using the usual semantics.
- `number string-length(string?)` returns the number of characters in the string, which is by default the string value of the context node.

Boolean functions

- `boolean boolean(object)` converts an object to a boolean value. Numbers different from zero or NaN (Not a Number), nonempty node sets, strings with positive length are converted to true.
- `boolean not(boolean)` negation.

Number functions

- `number number(object?)` converts an object to a number. Strings are converted to numbers by the usual semantics. The boolean false is converted to 0 and true is converted to 1. A node set is converted to string and then it is converted to a number. The default argument is the node set in context.
- `number sum(node-set)` returns the sum of the values obtained by converting each element in the node set into a number.
- `number round(number)` rounding the argument by the usual semantics.

The full specification of the functions in the library can be found in [10].

3.5 XSL Transformations

The XSL Transformation language, or XSLT for short, is another member of the XSL family, whose primary purpose is to make possible the transformation of XML documents into other XML documents or even into non-XML text data (e.g. HTML documents). There are three components of the transformation process. One is an XSLT stylesheet, which is a valid XML document that adheres the grammar of the XSLT specifications. The other is the source XML document and the third is the resulting document, which is generated in the transformation process by applying the stylesheet on the source document. The structure of the output document can be completely different from the structure of the input one.

The XSLT document contains a collection of templates that prescribe patterns and transformation rules for the elements of an XML document that match the pattern. In the pattern specification the XPath language gets a prominent role. The XSLT language also provides additional functionality, for instance named templates (whose usage is analogous to the one of functions in programming languages), if-then and for-each structures, sorting of elements in node sets, etc.

In practice XSLT is very often used to transform XML documents into HTML ones that can be displayed by web-browsers. Some browsers are actually XSLT-compatible. In such cases an XML document can be processed in the following workflow.

- The server sends the requested XML document to the browser.
- The browser parses the document and requests from the server the XSLT stylesheet the document references.
- The server provides the XSLT document.
- The browser applies the stylesheet on the XML document obtaining an HTML document. This can still reference a CSS (Cascaded Style Sheet) that contains further formatting information.
- If necessary, the browser requests the CSS from the server and applies it on the HTML document.
- Finally the browser displays the result.

Another alternative is that the XSLT stylesheet is applied on the XML document on the side of the server. This is necessary to support non-XSLT compatible browsers.

Example 3.16. Perhaps the best is to start with an example which shows how a simple XML document can be transformed into an HTML one via a straightforward XSLT. Let the XML file contain the following.

```
<?xml version="1.0"?>
<info>
  <name>Josef Keller</name>
  <email>jkeller@jku.at</email>
</info>
```

The XSLT stylesheet contains the following data.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html><body>
      <p>Name: <xsl:value-of select="./info/name/text()"/><br/>
      Email: <xsl:value-of select="./info/email/text()"/></p>
    </body></html>
  </xsl:template>
</xsl:stylesheet>
```

The result of applying the stylesheet on the XML document (modulo whitespaces) is the following.

```
<html><body>
  <p>Name: Josef Keller<br/>
  Email: jkeller@jku.at</p>
</body></html>
```

In Example 3.16, we can observe the following important requirements for XSLT stylesheets. There is a namespace declaration that binds the prefix `xsl` to the namespace `http://www.w3.org/1999/XSL/Transform`. The elements qualified with this prefix will be taken as template instructions for the XSLT processor and the others will appear in the resulting document literally. The stylesheet still has to be a well-formed XML document thus the non-qualified elements still have to adhere the XML rules. Finally, that the `xsl:value-of` elements use XPath expressions to locate and retrieve the text data from the corresponding elements of the source document.

The data model of XML documents used in XSLT is the same as the one in XPath. The only modification to it is that nodes that contain only whitespace characters are eliminated from the tree (in the figure of Example 3.14 they are marked with a circle).

An XSLT stylesheet is represented by an `xsl:stylesheet` element in the containing XML document. The element children of `xsl:stylesheet` are called the top level elements. Here are some of the most important element types an `xsl:stylesheet` element can contain:

`xsl:include` To replace the include instruction with the top level elements of the referenced stylesheet. Example:

```
<xsl:include href="my_other_stylesheet.xsl"/>
```

`xsl:output` To specify in what format the output should be generated. Example:

```
<xsl:output method="html"/>
```

`xsl:variable` To bind a value to a variable name. Example:

```
<xsl:variable name="x">/my/long/location/path</xsl:variable>
```

Later the long path name can be referenced by `$x`.

`xsl:template` To define a template of the stylesheet.

3.5.1 Templates

As remarked above, an XSLT stylesheet is a collection of templates. An XSLT processor starts applying the stylesheet for the source XML document by looking for a template whose pattern matches the root element of the document. If there is such a template, like the one in Example 3.16, its transformation is executed and a fragment of the output document might get created. This template can also trigger the application of further templates via the

```
<xsl:apply-templates select="location-path"/>
```

instruction for descendant nodes.

Generated text in the output document can be computed with the following instruction.

```
<xsl:value-of select="XPath-expression"/>
```

If template application is requested for a node and there is no matching template found, the following default template is applied.

```
<xsl:template match="*|/">
  <xsl:apply-templates select="*"/>
</xsl:template>
```

This template matches any child element nodes or the root element and just propagates template application to the children of the context node.

For attributes and text data nodes the following default template is defined.

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

That is, the result is just the string value for such nodes.

For processing instructions and comments the default template prescribes to do nothing.

```
<xsl:template match="processing-instruction()|comment()"/>
```

A general template definition looks as follows.

```
<xsl:template match="pattern" name="name" priority="priority">
  <!--Transformation rules come here.-->
</xsl:template>
```

The pattern can be a location path (defined in Section 3.4.2) or alternatives of such, separated by '|' characters (standing for the 'OR' logical connective). Recall that `text()` selects any text node, `processing-instruction()` selects processing instruction nodes, and `id("AB12")` selects the element with unique ID "AB12".

The `name` attribute is optional and allows us to define a named template which can be referenced at another point of the stylesheet. The `priority` attribute is also optional and assigns a value to the template which is used in conflict resolution, when more than one template can be applied in a certain context.

Example 3.17. Let us consider again the XML document containing emails at the beginning of Section 3.1. We want to transform this document into HTML format to display it in a nicer form. Here is a stylesheet for that.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="folder">
    <html><body>
      <h1>Emails</h1>
      <xsl:apply-templates select="email"/>
    </body></html>
  </xsl:template>
  <xsl:template match="email">
    <p>
    From: <xsl:value-of select="from/text()"/><br/>
    To: <xsl:value-of select="to/text()"/><br/>
    Date: <xsl:value-of select="@date"/><br/>
    Subject: <xsl:value-of select="subject/text()"/><br/><br/>
    <pre><xsl:apply-templates select="text()"/></pre>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

When the stylesheet is applied to the XML document, the root element is matched by the default template which just requests template application for each elements. For the processing instruction the default template applies which results no output. Then the `folder` child comes and the first template matches for it. Please note that the `folder` pattern selects all element children of the context node, which is the root in this case.

With the first template the frame of the HTML document is generated, and template application is requested for each `email` children (of element type) of the `folder` node. For the template search the context node will be now the `folder` node (and not the root).

The `email` elements of the `folder` element will be processed sequentially, applying the second template of the stylesheet. This just forms some heading including the text data from the children element named `from, to, subject` and the value of the attribute `date`. Finally it also appends the text data of the `email` node itself, which is just the body of the message. The result (modulo whitespaces) is the following.

```
<html><body>
<h1>Emails</h1>
  <p>
  From: robert@company.com<br/>
  To: oliver@company.com<br/>
  Date: 20 Aug 2003<br/>
  Subject: Meeting<br/><br/><pre>
  Could we meet this week to discuss the interface problem
  in the NTL project? --Rob
  </pre></p>
```

```
  <p>
  From: oliver@company.com<br/>
  To: rob@company.com<br/>
  Date: 21 Aug 2003<br/>
  Subject: Re: Meeting<br/><br/><pre>
  On 20 Aug 2003 rob@company.com wrote
  &gt; Could we meet today to discuss the interface problem
  &gt; in the NTL project? --Rob
  OK. What about today at 16:00?
  </pre></p>
</body></html>
```

Templates can be applied recursively in a natural way. For instance the following template will apply itself as long as there are `item` children of the context node. Please note that the depth of the recursion is always finite.

```
<xsl:template match="item">
  <div style="margin-left:20px">
    <xsl:value-of select="text()"/>
    <xsl:apply-templates select="item"/>
  </div>
</xsl:template>
```

3.5.2 Additional features

One might wish to include attribute values in the output document which are generated from data of the source document. This would not be possible by the standard constructions discussed so far, without violating XML rules in the stylesheet. To solve this problem, two constructions are available in XSLT. The first is the `xsl:attribute` instruction which inserts the named attribute with the retrieved value into the element created by the first element creator ancestor of the `xsl:attribute` element in the stylesheet. The second construction is more compact, using the '{', '}' delimiters to include the XPath expression right in the place it goes to.

Example 3.18. Assume that one wants to define the text color in the HTML output according to the value of the `color` attribute of the `message` elements in the source XML document. The corresponding template can be the following.

```
<xsl:template match="message">
  <font>
    <xsl:attribute name="color">
      <xsl:value-of select="@color"/>
    </xsl:attribute>
    <xsl:value-of select="text()"/>
  </font>
</xsl:template>
```

Using the more compact notation the same can be achieved by the following template.

```
<xsl:template match="message">
```

```
  <font color="{@color}">
    <xsl:value-of select="text()"/>
  </font>
</xsl:template>
```

Elements can be generated not only by including them literally in the stylesheet but also via the `xsl:element` stylesheet element. It has a mandatory attribute `name` that specifies the name of the element to be created, and in which we can use the dynamic attribute value expansion discussed above. That is, we can also create elements in the output document dynamically.

Let us consider the following XML document.

```
<?xml version="1.0" encoding="utf-8"?>
<system>
 <stamp>12-03-02 23:13</stamp>
 <msgs>
  <msg type="info">System started</msg>
  <msg type="info">Logging in user 'maryk'</msg>
  <msg type="warn">User 'bobm' not found</msg>
 </msgs>
</system>
```

Let the goal be to create output XML documents from source documents of the type above, so that the structure is not changed, except that we replace the `msg` elements with ones having names specified by the `type` attributes in the source document. An XSLT stylesheet doing this is the following.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="system">
    <system>
      <xsl:apply-templates select="stamp"/>
      <xsl:apply-templates select="msgs"/>
    </system>
  </xsl:template>
  <xsl:template match="stamp">
    <stamp><xsl:value-of select="text()"/></stamp>
  </xsl:template>
  <xsl:template match="msgs">
    <msgs>
      <xsl:apply-templates select="msg"/>
    </msgs>
  </xsl:template>
  <xsl:template match="msg">
    <xsl:element name="{@type}">
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

The transformation of the above example document with this stylesheet looks as follows.

```
<?xml version="1.0" encoding="utf-8"?>
<system>
  <stamp>12-03-02 23:13</stamp>
  <msgs>
    <info>System started</info>
    <info>Logging in user 'maryk'</info>
    <warn>User 'bobm' not found</warn>
  </msgs>
</system>
```

When in the source XML document a node has many repeating children that require the same way of processing, an iterative structure can be used via the `xsl:for-each` instruction. It takes the nodes in the selected node set in document order and applies the given transformation to each of them.

Example 3.19. Taking again the same source XML document as in Example 3.17, we can process the emails as follows.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="folder">
    <html><body>
      <h1>Emails</h1>
      <xsl:for-each select="email">
        <p>
        From: <xsl:value-of select="from/text()"/><br/>
        To: <xsl:value-of select="to/text()"/><br/>
        Date: <xsl:value-of select="@date"/><br/>
        Subject: <xsl:value-of select="subject/text()"/><br/><br/>
        <xsl:apply-templates select="text()"/>
        </p>
      </xsl:for-each>
    </body></html>
  </xsl:template>
</xsl:stylesheet>
```

Conditional processing is also possible with the `xsl:if` and `xsl:choose` instructions. The `xsl:if` construction does not have an 'else' branch.

Example 3.20. Let us take the example XML document at the end of Section 3.2 without the namespace specifications. If one wants to compile a table whose rows contain the text content of the `book` elements, such that every second row has grey background, one can use the following template.

```
<xsl:template match="book">
  <tr>
    <xsl:if test="position() mod 2 = 0">
       <xsl:attribute name="bgcolor">grey</xsl:attribute>
    </xsl:if>
    <td><xsl:value-of select="author/text()"/></td>
    <td><xsl:value-of select="title/text()"/></td>
```

```
    <td><xsl:value-of select="@isbn"/></td>
  </tr>
</xsl:template>
```

Here is the triple color version too, just to demonstrate the `xsl:choose` instruction.

```
<xsl:template match="book">
  <tr>
    <xsl:choose>
      <xsl:when test="position() mod 3 = 0">
        <xsl:attribute name="bgcolor">grey</xsl:attribute>
      </xsl:when>
      <xsl:when test="position() mod 3 = 1">
        <xsl:attribute name="bgcolor">blue</xsl:attribute>
      </xsl:when>
      <xsl:otherwise>
        <xsl:attribute name="bgcolor">white</xsl:attribute>
      </xsl:otherwise>
    </xsl:choose>
    <td><xsl:value-of select="author/text()"/></td>
    <td><xsl:value-of select="title/text()"/></td>
    <td><xsl:value-of select="@isbn"/></td>
  </tr>
</xsl:template>
```

We mention also the possibility to sort the node set matched by a pattern of a template or selected by a `for-each` instruction. The `sort` instruction has several attributes; the node set can be defined via the `select` attribute and the ordering will be done on the string values of the nodes in the node set (except if the `data-type` attribute prescribes something else). Ascending or descending sorting can be specified via the `order` attribute. If an `xsl:template` or `xsl:sort` element has more than one `sort` children then in the order of appearance the first defines the primary sort key, the second defines the secondary sort key and so on. The template will process the elements of the node set in the sorted order.

Example 3.21. Assume that we have an XML document with the following structure.

```
<products>
  <product>
    <id>ASN32255</id>
    <price currency="EUR">12.5</price>
  </product>
  <product>
    <id>ASN54345</id>
    <price currency="EUR">3.67</price>
  </product>
  <product>
    <id>ASN31345</id>
    <price currency="EUR">24.22</price>
```

```
  </product>
</products>
```

We would like to produce an HTML table from these products with the prices being in ascending order. Here is an XSLT stylesheet to do that.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="products">
    <html><body>
      <h1>Product list</h1>
      <table>
      <tr><th>ID</th><th>Price</th></tr>
      <xsl:for-each select="product">
        <xsl:sort select="price" order="ascending" data-type="number"/>
        <xsl:sort select="id" order="ascending"/>
        <tr><td><xsl:value-of select="id"/></td>
        <td align="right">
         <xsl:value-of select="price/text()"/>
         <xsl:value-of select="price/@currency"/>
        </td></tr>
      </xsl:for-each>
      </table>
    </body></html>
  </xsl:template>
</xsl:stylesheet>
```

Products will be sorted by their price and equally priced products are sorted by their ID. Please note also that the first `xsl:value-of` instruction selects only the `id` child of a `product` element while the second selects the text data node of the `price` child. The second is clearly what we want and the first is also correct because the default template will just fetch the text data of the `id` element.

Example 3.22. Finally this simple example shows how can one use the XSLT facilities to establish references between elements of the source XML document. Let us assume that the source XML file contains letter drafts which should be formatted and displayed. Let the source document store the dates by numbering months. We want to convert the dates to the following format `month-name dd, yyyy`. Here is the sketch of the source XML file.

```
<root>
  <letter>
    <date y="2003" m="08" d="28"/>
    <from>Sender</from>
    <to>Recipient</to>
    <address>Address of recipient</address>
    Letter body.
  </letter>
  <months>
    <month id="01">January</month>
    <!-- And so on -->
```

```
    <month id="12">December</month>
  </months>
</root>
```

The last element declares which month number gets which character name (as at this point of the XML introduction provided by these notes we cannot establish references between elements of different XML documents, we have to add the `months` element as a child of the source document). The XSLT stylesheet could be the following.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="root">
    <html><body>
      <xsl:apply-templates select="letter"/>
    </body></html>
  </xsl:template>
  <xsl:template match="letter">
    <p>
    To: <xsl:value-of select="to"/><br/>
    <xsl:value-of select="address"/><br/><br/>
    Dear <xsl:value-of select="to"/>,<br/><br/>
    <xsl:apply-templates select="text()"/><br/><br/>
    <xsl:value-of select="concat(
      /root/months/month[@id=current()/date/@m],
      ' ', date/@d, ',', date/@y)"/><br/>
    Best regards,<br/>
    <xsl:value-of select="from"/>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

In the date retrieval we can use the `concat` function to produce the concatenation of the resulting string values. The `current` function retrieves the current context outside the square brackets in the location path, so that `current()/date/@m` will refer to the `m` attribute of the `date` element of the `email` element currently processed (and not to an attribute of some descendant of the `months` element).

3.5.3 Exercises

Exercise 3.5.1. Consider the following XML document.

```
<logs>
  <allocation>
    <item id="beamer1">Sharp Beamer (1024x768)</item>
    <reserved-for>Josef Niel</reserved-for>
    <from>2003-12-08T13:00:00</from>
    <till>2003-12-09T18:00:00</till>
  </allocation>
  <allocation>
```

```
    <item id="laptop2">Toshiba Satellite Pro 6100</item>
    <reserved-for>Josef Niel</reserved-for>
    <from>2003-12-08T13:00:00</from>
    <till>2003-12-09T18:00:00</till>
  </allocation>
  <allocation>
    <item id="laptop1">Toshiba Satellite Pro 6100</item>
    <reserved-for>Barbara Hill</reserved-for>
    <from>2003-12-03T10:00:00</from>
    <till>2003-12-07T08:00:00</till>
  </allocation>
</logs>
```

Write an XSLT stylesheet that produces a small XHTML document which contains a table whose rows collect the allocation data of 'Josef Niel'.

Exercise 3.5.2. Take the XML document of Exercise 3.3.3 and write an XSLT stylesheet arranges the authors of each bibliographic entry in alphabetical order and then also arranges the entries in ascending order by the publication years. The resulting document should have the same structure as the source document (it also has to be valid with respect to the schema that validated the source document).

Exercise 3.5.3. Consider the example of bidding from Exercise 3.3.1, let now the XML document look as follows.

```
<bidding>
 <items>
  <item id="3489873">CD Player</item>
  <item id="3423987">Stereo Amplifier</item>
 </items>
 <member id="A2342" item-id="3489873">
  <name>Thomas Keller</name>
  <bid timestamp="2003-12-30T18:20:35" currency="EUR">34</bid>
  <bid timestamp="2003-12-30T18:28:32" currency="EUR">36</bid>
 </member>
 <member id="A2542" item-id="3489873">
  <name>Hugo Browning</name>
  <bid timestamp="2003-12-30T18:23:35" currency="EUR">35</bid>
  <bid timestamp="2003-12-30T18:32:32" currency="EUR">40</bid>
 </member>
 <member id="A2342" item-id="3423987">
  <name>Thomas Keller</name>
  <bid timestamp="2003-12-31T18:20:35" currency="EUR">23</bid>
 </member>
</bidding>
```

Write an XSLT stylesheet that produces an XHTML document containing a table for each item in the `items` element. The rows of a table are the bids received for the corresponding item in descending order by the value of the bid. The information displayed in the row should also contain the currency, the name of the person and the timestamp.

3.6 XML Query

The purpose of XML Query (XQuery for short) is to provide a language for extracting data from XML documents. The queries operate on single documents or fixed sets of documents and can select whole documents or subtrees of documents that match conditions on content and structure.

The query language is functional (but it also includes universal and existential quantifiers), supports simple and complex data types defined in XML Schema, defines aggregate functions, handles null values and fully compliant with XML namespaces.

Just as in XSLT, the expressions play the central role in XQuery. The value of an expression is always a sequence, which is a list of zero or more atomic values (as in XML Schema) or nodes (of the given node types, as in XPath). The `xs:` prefix is assumed to be bound to the URI `http://www.w3.org/2001/XMLSchema`, the `fn:` prefix to the URI `http://www.w3.org/2003/11/xpath-functions` and the `xdt:` prefix to the URI `http://www.w3.org/2003/11/xpath-datatypes`.

3.6.1 Data Model and Types

The data model of XQuery is an extended version of the data model of XPath (also referred to as the XPath 2.0 data model). The root node type is called document-node type (identified by the unique URI of the document, so that collections of document nodes can also be treated). The data model can contain various extra information for each node, e.g. the parent, the children, the in-scope namespaces, an indication whether the node is nilled, the type, etc. These information can be obtained via *accessors* (interface specifications for functions which should be included in any implementations). Here we briefly discuss only the type annotation of nodes in the data model.

Type information can be assigned to element or attribute nodes either using the post schema validation infoset (PSVI) which, as its name shows, is obtained by validating the document against schema(s), or, if the PSVI cannot provide type information for a node, just using the `xdt:untypedAny` type for element nodes and the `xdt:untypedAtomic` type for attribute nodes (see below). For anonymous types (having no names assigned in the schema) the parser assigns internal identifiers.

The typed value of a node can be extracted by applying the `fn:data` function on the node, which corresponds to the `dm:typed-value` accessor, (the string value—recall the notion from the discussion of XPath—can be obtained by applying the `fn:string` function). The typed value of a node is computed in the following way:

- For text, document, and namespace nodes, the typed value of the node is the same as its string value, as an instance of the type `xdt:untypedAtomic`.
- The typed value of a comment or processing instruction node is the same as its string value, as an instance of the type `xs:string`.
- The typed value of an attribute with type annotation `xdt:untypedAtomic` is its string value as an instance of this type. With other type annotation

it is derived from the string value in a way consistent with the schema validation.

For element nodes the typed value can be computed in the following way:

- If the element has a type of `xdt:untypedAtomic` or a complex type with mixed content, the typed value is the string value of the node as an instance of `xdt:untypedAtomic`.
- If the element has a simple type or a complex type with simple content, the typed value is a sequence of zero or more atomic values derived from the string value of the node and its type in a way that is consistent with the schema validation.
- If the node has a complex type with empty content, the typed value is the empty sequence.
- If the node has a complex type with element only complex content, its typed value is undefined.

XQuery is a strongly typed language. It uses the XML Schema data types, which have to be qualified with the predefined `xs` prefix (e.g. `xs:integer`), and the following additional data types (with predefined `xdt` prefix):

`xdt:anyAtomicType` an abstract data type of all atomic values,

`xdt:untypedAny` a concrete data type for values of element nodes not having any specific dynamic type assigned to them,

`xdt:untypedAtomic` a concrete data type for atomic values not having more specific type assigned to them,

`xdt:dayTimeDuration` and `xdt:yearMonthDuration` both are concrete subtypes of `xs:duration` respectively.

Expressions are evaluated in two phases, called *static analysis* and *dynamic evaluation*.

In the static analysis phase, which depends only on the expression and statically available information (one which is available before the actual evaluation of the expression, e.g. in-scope namespaces, defined variables, available functions), a concrete or an abstract data type may be assigned to the expression "in advance". This phase is also useful to quickly detect static errors in the expression.

In the dynamic evaluation phase, that comes after the static analysis, the value of the expression is computed with respect to the dynamic context (analogous to the concept of 'context' in XPath, containing additional information on the available documents and node collections, and the current date and time). The type determined in this phase is called the the dynamic type of the expression.

Since expressions always have sequence values, to simplify the notation we do not make distinction between an atomic value or a node and the singleton sequence containing it. However when a sequence of values is longer than one, we have to talk about *sequence types*. The main difference between types and sequence types is the possibility of prescribing occurrence constraints on the

appearance of the given type in the sequence. The character '`?`' denotes zero or one, '`*`' denotes zero or more and '`+`' denotes one or more occurrence. The node kinds are denoted by their names with a '`()`' suffix, `node()` standing for a node of any kind. For the detailed specification we refer to [11]. A few examples are:

- `xs:date` refers to the built-in Schema type `date`,
- `attribute()?` refers to an optional attribute,
- `element()` refers to any element,
- `element(po:shipto, po:address)` refers to an element that has the qualified name `po:shipto` and has the type annotation `po:address` (or a subtype of that type),
- `node()*` refers to a sequence of zero or more nodes of any type.

Type matching is also a part of the expression evaluation process, i.e. a sequence of values with given sequence type has to be checked against an expected sequence type (e.g. input argument list of a function). The type matching results true if the given sequence type is known and matches the known expected sequence type, or if it can be derived by extensions or restrictions from the expected type. In other cases the result is false, except if the expected type is unknown or it is not possible to determine if the given type can be derived from the expected one, in which case a type error is risen.

3.6.2 Expressions

XQuery expressions can be split in the following main categories: primary-, path-, sequence-, arithmetic-, comparison- and logical-, constructors, FLWOR-, conditional- and quantified expressions. Here we discuss them only briefly, for the complete information we refer to [11].

Primary expressions include literals, variable references, context item expressions, constructors, and function calls. A primary expression may also be created by enclosing any expression in parentheses. A few examples of literals:

- `"3.14"` denotes the string of the characters '3', '.', '1', '4',
- `3.14` denotes the decimal value 3.14,
- `314` denotes the integer value 314,
- `314e-2` denotes the double value 3.14,
- `"Pratt &amp; Whitney"` denotes the string 'Pratt & Whitney'.

Variable references can be constructed by prefixing a (possibly qualified) variable name with a '$' sign (e.g. `$x`). Variables can be declared in the query prolog (a series of declarations, schema and module imports that modify the context for the evaluation of the body of the query) or can be bound by XQuery expressions (like FLWOR and quantified expressions).

Function calls consist of the (possibly qualified) name of the function followed by a parenthesized list of arguments. The core function library of XPath is available, but one can also define new functions in the query prolog. The input parameters undergo a type matching, in which certain values can be split into a sequence of atomic values (atomization) and values with type `xdt:untypedAtomic` are attempted to be type casted to the required types. If the type matching was successful, the formal parameters of the function are bound to the corresponding input values, however if the input value is a subtype of the prescribed type of the formal parameter, it keeps its more specific type.

Path expressions are basically known to us from XPath (see location paths in Section 3.4).

Sequence expressions can be formed via the comma operator applied on (possibly sequence) expressions. The result is always a "flat" sequence, so that the elements of the nested sequences are shifted up to the outermost level. For instance:

- `(1,(2,3),(),(3,4))` results `(1,2,3,3,4)`,
- `(book)` results the sequence of all `book` children of the context node.

Sequences can also be combined by the operators: `union`, `intersect`, `except` performing the corresponding set-theoretic operations (eliminating duplicates from the resulting sequence).

Arithmetic expressions can be built from atomic values that can be type casted to an accepted input type of the applied operator, which can be: addition '`+`', subtraction '`-`', multiplication '`*`', division '`div`' (integer division '`idiv`'), and modulus '`mod`'.

Comparison expressions can be of the type: value comparisons (`eq, ne, lt, le, gt, ge`), general comparisons (`=, !=, <, <=, >, >=`) and node comparisons (`is, <<, >>`). Value comparisons require atomic types, e.g. `book/author eq "Kennedy"` is true if and only if the `book` element node has exactly one `author` child and its typed value is `"Kennedy"` as an instance of `xs:string` or `xdt:untypedAtomic`.

General comparisons can also be applied when the atomization of the operands produce longer sequences. The result of the comparison is true if and only if there is a pair of atomic values from the operands which are in the required relationship (applying the corresponding value comparisons for the atomic values, e.g. `eq` for `=`).

In node comparisons the operands must be single nodes or the empty sequence. The `is` operator test if the operands are the same, in the sense of having the same identity (recall that in XML documents an element can have many children with the same name and the same content, each having its own identity). The `<<` and `>>` operators compare nodes with respect to the document order. For instance: `/book[@isbn="0-387-94269-6"] is /book[@id="L23432"]` is true if and only if the two path expressions evaluate to the same single node in the document.

Logical expressions can be formed from subexpressions that evaluate to boolean type using the `and` and `or` operators.

Constructors are provided for every kind of node types to create XML structures in a query. There are direct and computed constructors, where the direct constructors resemble to the corresponding concept in XSLT, where it is possible to embed dynamically computed values by enclosing the defining expressions in {}. For instance, the following direct element constructor creates a new node with its own identity in the result of the evaluation.

```
<p id="{$b/@isbn}">Book:<br/>
{string($b/title)}, {string($b/author)} [{string($b/@isbn)}]</p>
```

If the variable `b` is bound to the node

```
<book isbn="0-812909191-5">
  <title>Codenotes for XML</title>
  <author>G. Brill</author>
</book>
```

the result is the following.

```
<p id="0-812909191-5">Book:<br/>
Codenotes for XML, G. Brill [0-812909191-5]</p>
```

The curly braces can be included in the result either by doubling them: "{{" and "}}" will represent a '{' and a '}' respectively, or by using the corresponding character references: &#x7b and &#x7d.

The direct element constructor automatically validates the created element against the available in-scope schemas (depending on the set validation mode this might result in an error or assigning the type `xdt:untypedAny` to the element and its children if the validation fails).

Computed constructors start with a keyword, identifying the node type to be created (element, document, text, processing-instruction, comment, namespace), followed by the name (or an expression, in braces, resulting a name, which is possible only for non-namespace nodes) for element, attribute, namespace or processing instruction node. Finally the content expression defines the content of the node. Computed constructors can be nested, just as the direct ones. The previous example with computed constructors would look as follows.

```
element p {
  attribute id {$b/@isbn},
  "Book:", element br {},
  string($b/title), ",", string($b/author), ", [",
  string($b/@isbn), "]"
}
```

An important use of computed constructors is to be able to assign the name of the created element dynamically.

FLWOR expressions support iteration and binding variables to intermediate results. The acronym was created from the parts of such expressions: `for`, `let`, `where`, `order by`, `return`. The `for` and `let` clauses in a FLWOR expression generate a sequence of tuples of bound variables, called the tuple stream. The `where` clause serves to filter the tuple stream, retaining some tuples and discarding others. The `order by` clause imposes an ordering on the tuple stream. The `return` clause constructs the result of the FLWOR expression.

The following example (from [11]) of a FLWOR expression includes all of the possible clauses. The `for` clause iterates over all the departments in an input document, binding the variable $d to each department number in turn. For each binding of $d, the `let` clause binds variable $e to all the employees in the given department, selected from another input document. The result of the `for` and `let` clauses is a tuple stream in which each tuple contains a pair of bindings for $d and $e ($d is bound to a department number and $e is bound to a set of employees in that department). The `where` clause filters the tuple stream by keeping only those binding-pairs that represent departments having at least ten employees. The `order by` clause orders the surviving tuples in descending order by the average salary of the employees in the department. The `return` clause constructs a new `big-dept` element for each surviving tuple, containing the department number, headcount, and average salary.

```
for $d in fn:doc("depts.xml")//deptno
let $e := fn:doc("emps.xml")//emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
  <big-dept>
    {$d,
    <headcount>{fn:count($e)}</headcount>,
    <avgsal>{fn:avg($e/salary)}</avgsal>}
  </big-dept>
```

The `for` clause can contain more than one variables, and then the constructed tuple stream will contain variable bindings for each combination of values for the variables from the Cartesian product of the sequences over which the variables range. The `let` clause can also contain more than one variable, however it binds each variable to the result without iteration. This example illustrates the difference:

```
for $i in (<x/>,<y/>,<z/>)
return <a>$i</a>
```

results

```
<a><x/></a><a><y/></a><a><z/></a>
```

while

```
let $i := (<x/>,<y/>,<z/>)
return <a>$i</a>
```

results

```
<a><x/><y/><z/></a>
```

Each variable bound in a `for` clause can have associated positional variable that iterates on the integers from 1 on as the variable iterates on the sequence. The positional variable comes after the ordinary one separated by the 'at' keyword. For instance

```
for $author at $i in ("G. Brill", "C. J. Date"),
  $book at $j in ("Codenotes for XML", "Database Systems")
```

generates the following tuple stream.

```
($i = 1, $author = "G. Brill", $j = 1, $book = "Codenotes for XML")
($i = 1, $author = "G. Brill", $j = 2, $book = "Database Systems")
($i = 2, $author = "C. J. Date", $j = 1, $book = "Codenotes for XML")
($i = 2, $author = "C. J. Date", $j = 2, $book = "Database Systems")
```

The `where` clause just contains an expression which is evaluated for each tuple in the tuple stream, and which should provide a boolean value. The tuples result in false get filtered out. The `return` clause will then be evaluated for each remaining tuple in the filtered tuple stream taken in the order prescribed by the `order by` clause (if it is omitted, the order will be the one of the tuple stream).

Conditional expressions are analogous to the well known 'if-then-else' constructions in imperative programming languages. An example (from [12]).

```
<bib>
  { for $b in doc("http://bstore1.example.com/bib.xml")//book
    where fn:count($b/author) > 0
    return
        <book>
            { $b/title }
            { for $a in $b/author[position()<=2] return $a }
            { if (fn:count($b/author) > 2) then <et-al/> else () }
        </book> }
</bib>
```

Quantified expressions support the universal (`every`) and the existential (`some`) quantifiers. The quantifier keyword is followed by possibly several `in`-clauses that bind sequences to variables, from which a tuple stream is formed as for the FLWOR expressions. The final part of the expression is a test-clause, separated by the `satisfies` keyword from the rest of the expression, which is evaluated for each tuple in the stream. The value of the existentially quantified (`some`) expression is true if at least one of the evaluations of the test expression on the tuples from the stream is true (the empty stream yields false value). The `every` expression is true if every evaluation of the test expression evaluates to true (the empty stream yields true).

Examples (from [11]): this expression is true if every part element has a discounted attribute (regardless of the values of these attributes).

```
every $part in //part satisfies $part/@discounted
```

This expression is true if at least one employee element satisfies the given comparison expression.

```
some $emp in //employee satisfies ($emp/bonus > 0.25 * $emp/salary)
```

3.6.3 Query Prolog

The prolog is a semicolon separated series of declarations (of variables, namespaces, functions, etc.) and imports (of schemas, modules) that create the environment for query processing.

A *namespace declaration* binds a namespace prefix to a namespace URI, adding the (prefix, URI) pair to the set of in-scope namespaces for the query. Example:

```
declare namespace x = "http://www.my-company.com/";
```

Binding of prefixes to namespace URIs can locally be overridden by namespace declarations in the prolog, whose prefix binding can be overridden inside the query in an element constructor. Default namespaces can be defined for elements and attributes. Example:

```
declare default element namespace "http://example.org/names";
declare default function namespace "http://example.org/math-functions";
```

A *schema import* adds a named schema to the in-scope schema definitions. The following example imports the schema for an XHTML document, specifying both its target namespace and its location, and binding the prefix `xhtml` to this namespace. Example:

```
import schema namespace xhtml="http://www.w3.org/1999/xhtml"
   at "http://example.org/xhtml/xhtml.xsd";
```

A *variable declaration* adds a variable binding to the in-scope variables. The value of the variable can be provided by an initializing expression or by the external environment. Examples (from [11]): The following declaration specifies both the type and the value of a variable. This declaration causes the type `xs:integer` to be associated with variable `$x` in the static context, and the value 7 to be associated with variable `$x` in the dynamic context.

```
declare variable $x as xs:integer {7};
```

The following declaration specifies a type but not a value. The keyword `external` indicates that the value of the variable will be provided by the external environment. At evaluation time, if the variable `$x` in the dynamic context does not have a value of type `xs:integer`, a type error is raised.

```
declare variable $x as xs:integer external;
```

A *function declaration* adds a user defined or external function to the available in-scope functions. User defined functions must include an expression that defines the result in terms of the parameters. In order to allow main modules to declare functions for local use within the module without defining a new namespace, XQuery predefines the namespace prefix `local` to the namespace `http://www.w3.org/2003/11/xquery-local-functions`, and reserves this namespace for use in defining local functions.

Example (from [11]): This local function accepts a sequence of valid employee elements (as defined in the in-scope element declarations), summarizes them by department, and returns a sequence of valid `dept` elements.

```
declare function local:summary($emps as element(employee)*)
   as element(dept)*
{
   for $d in fn:distinct-values($emps/deptno)
   let $e := $emps[deptno = $d]
```

```
   return
      <dept>
         <deptno>{$d}</deptno>
         <headcount> {fn:count($e)} </headcount>
         <payroll> {fn:sum($e/salary)} </payroll>
      </dept>
};
```

This application of the previous function computes a summary of employees in Denver.

```
local:summary(fn:doc("acme_corp.xml")//employee[location = "Denver"])
```

3.6.4 Exercises

Exercise 3.6.1. Solve the exercises of Section 3.5.3 using now XQuery instead of XSLT.

Bibliography

[1] Apache. http://www.apache.org/.

[2] Brill, G. *Codenotes for XML*. Random House, 2001.

[3] Date, C. J. *An Introduction to Database Systems*. Addison Wesley, 1995.

[4] Perl. http://www.perl.com/.

[5] PHP. http://www.php.net/.

[6] PostgreSQL. http://www.postgresql.org/.

[7] Search Engine Watch. http://searchenginewatch.com/.

[8] XML. http://www.w3.org/XML/.

[9] XMLSchema. http://www.w3.org/XML/Schema.

[10] XPath. http://www.w3.org/TR/xpath.

[11] XQuery. http://www.w3.org/TR/xquery/.

[12] XQuery Use Cases. http://www.w3.org/TR/xquery-use-cases/.

[13] XSLT. http://www.w3.org/Style/XSL/.

[14] Zaiane, O. Database systems and structures, 1998. lecture notes (also availble on line: http://www.cs.sfu.ca/CC/354/zaiane/material/notes/contents.html).