ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Internet Engineering

Maciej Nikodem

ADVANCED DATABASES

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Internet Engineering

Maciej Nikodem

ADVANCED DATABASES

Wrocław 2011

Copyright © by Wrocław University of Technology
Wrocław 2011

Reviewer: Jacek Mazurkiewicz

ISBN 978-83-62098-85-9

Published by PRINTPAP Łódź, www.printpap.pl

Contents

Chapter 1

Introduction

The file system that comes with any operating system is a quite primitive kind of database management system in which data are kept in big unstructured named clumps called files. The great thing about the file system is its invisibility and capability to store any data that may be represented in a digital form. However, not every way of storage is efficient in particular situations, especially when datasets are large and frequent searching over these data is required. Obviously anyone (or at least a large group of people) can write their own program for storing, reading and searching in large files, however, it is very unlikely that these programs will run efficiently. Moreover, if many applications are allowed to access the same data, than new challenges arise – programs may attempt to read and write the same data file simultaneously raising questions about data correctness, integrity or leading to deadlocks of application.

Database management systems are solution that allows to overcome a large amount of difficulties that may arise in a multiuser data access. Database management systems (DBMSs) are created to free the programer from taking too much care about accessing the data within a program. ”Too much” means that DBMS does not solve all of the problems but when configured and used properly, and when datasets, that DBMS is managing, are properly constructed then difficulties are minimised and most problems are efficiently solved. This points out two important aspects that should be kept in mind when implementing any database-based applications:

- database and database management system are two different things,
- database management system requires a wisely constructed database in order to benefit from its functionality.

Database can be seen as a logical structure used to store data. It consist of columns, that store data of particular kind, type and possibly with additional restrictions. Columns are organised in tables that store related data. Data stored in several tables can be also related, so database also stores information about such relations. On the other hand DBMS is an application that is responsible for managing both the logical and physical data storage, interfaces users /applications to the database and ensures that no data in the database can be lost (or generated) incidentally. It is important to keep this difference in mind as both terms are often used interchangeably.

Since DBMS ”only” ensures database is logically correct (integral) therefore it has to be taught when database is correct and when it doesn't. DBMS determine status of the database based on constraints (e.g. primary keys, foreign keys, indices, constraints, etc.) and/or procedures (e.g. triggers, stored procedures, functions). Both mechanisms are defined for a particular database and are required to verify its condition. In other words – if database has no constrains defined on it then DBMS cannot verify whether these constraints are meet or no. Consequently, DBMS can interface the user to the database but cannot ensure the data inserted to or read from the database is logically correct.

The remaining part of this script presents a narrow part of topics related to databases and DBMS. First part focuses on relational theory which is a theoretical background required to create efficient and useful database structures. Second part focuses on mechanisms that are used to ensure correctness of the database as well as functions and procedures that are provided by DBMS in order to manipulate the database and the data. Practical examples presented in this script focus mostly on two DBMSs – MySQL[1] and Microsoft SQL Server (MsSQL)[2]. Some information and examples from Oracle[3] and DB2[4] servers are also presented.

At the end of this script you will find a list of bibliography. However, it is important to notice that this bibliography mostly consist of Internet addresses that point to documentation of different DBMS as this is the most up to date source of information about this kind of software. For general description of databases and DBMS the reader is referred to [9],[10] as well as different Web pages, e.g. http://infolab.stanford.edu/~ullman/fcdb.html.

[1]http://www.mysql.com/
[2]http://www.microsoft.com/sqlserver/en/us/default.aspx
[3]http://www.oracle.com/us/index.html
[4]http://www.ibm.com/software/data/db2/

Chapter 2

Database concepts

2.1 Representing real world

Notion of contemporary relational databases was first introduced by Edgar Frank Codd in 1970 with his paper *A Relational Model of Data for Large Shared Data Banks* [7]. This paper states four important aspects of databases and proposes a relational model and normal forms as a method to ensure them. According to Codd main purposes of using databases are:

- users must be protected from having to know how the data is organised in the database – internal storage,
- activities of users should remain unaffected when internal storage of data is changed,
- there should be a clear distinction and independence between order of data stored in internal storage and order of presentation,
- indices are optional – they may be used to facilitate and speed up access to data but data should be available regardless of whether indices exist. Moreover, activities of users should remain invariant as indices come and go.

Above mentioned properties do not exhaust all aspects of data storage since they focus on data availability and user activities. These properties can be complemented with other aspects that concern data storage (e.g. data compression in order to reduce the memory required), correctness (e.g. coding, backups of data to ensure data recovery in case of errors) or security (e.g. access control ensuring that only legitimate users can read/write data).

Irrespectively of their name databases store not only data but also information. The difference is that information conveys dependencies between data and gives particular meaning to data that otherwise can mean arbitrary many things. Information is about interpreting the data.

Example 2.1
Table 2.1 represents some data stored in a table. Although it may be a very effective method of storage, there is no particular meaning of this data.

Table 2.1: Some data stored in a table

1	5	9
2	5	7
3	5	2
2	6	12
3	6	3
4	7	1
6	7	1

However, if we have knowledge about how this data should be interpreted then we will be able to gain full information from it. For example each row of the table represents a component ID and the amount of this component (quantity) that have to be used in order to assembly a particular part (cf. tab. 2.2). □

Table 2.2: Table representing assembly process

component	part	quantity
1	5	9
2	5	7
3	5	2
2	6	12
3	6	3
4	7	1
6	7	1

Another difference is that in most cases word *datum* is used to describe elementary, atomic facts that cannot be divided further. Such data is easier to store effectively, e.g.

minimising overall memory requirements. On the other hand in most cases information can be divided into smaller pieces such that each conveys some partial information while all together give original information. Since information has granular nature thus manipulating information does not necessarily affect it whole. Instead it can affect only a piece of information. Storing information requires databases to reflect this granular nature of information in all aspects. First of all, information should be stored in such a way that allows to divide it into pieces and perform operations on either a piece or the whole information. When operating on a piece of information no other information should be affected. Second, databases have to ensure that once some information is stored it can be read without any change – no information is lost. Moreover, information that were not written into the database cannot be read out – database cannot generate information (it can represent information in different way, e.g. aggregate, but cannot produce new facts!).

In the field of databases, aforementioned difficulties are known as insert, update and delete *anomalies*. They may appear in a database depending on its structure. Proper design of database and normalisation process ensures that all of these anomalies are either eliminated or are left intentionally to ensure completeness and correctness of the information stored in the database management system (see chapter 3.3.2).

Update anomaly is mostly due to information redundancy in table/database which means that the same data can be stored several times. When redundant information is modified then the database management system has to ensure that all redundant entries are modified. Update anomaly means that the system fails to do so leaving some occurrences of information (or part of it) not modified and causing information in the database to be incorrect. Avoiding update anomaly is also important in order to ensure that information stored in the database is consistent and valid. Otherwise, update anomaly may lead to inconsistent information or may lead to information generation.

Delete anomaly may occur in normalised databases with no redundancy where data is stored in separate, related tables (e.g. separate table for cities, street names, house numbers and students with references to city, street and house number of their inhabitancy). When deleting piece of information in such a database then we may accidentally delete data that is part of other information. In a given example, delete anomaly means that removing student John Smith from New York with all details of his inhabitancy will leave all other students from New York with no information about the city they live – we will lose a piece of information about them that was not intended to be deleted.

In contrast to previous anomalies *insert anomaly* can occur in redundant and normalised databases. It happens when storing some information into the database requires some other information to be stored simultaneously. It is worth mentioning that, unlike other anomalies, insert anomaly may be desired ensuring completeness of data when storing information into the database. For example student database may require to store inhabitancy information simultaneously with personal information of each student.

Chapter 3

Theory of relational database systems

3.1 Relational model

Relation is defined on sets $S_1, S_2, \ldots, S_n$ and is represented as a set of tuples each of which has its first element from S_1, second from S_2, and so on. In mathematical sense relation is a subset of the Cartesian product of $S_1 \times \ldots \times S_n$. Set S_j is usually called a j-th domain or *attribute* of the relation. Relation is of degree n (or n-ary relation) if it is a subset of Cartesian product of n sets. Relation description consist of relation name followed by the list of its attributes in brackets:

$$\mathrm{R}\,(\mathrm{A}, \mathrm{B}, \ldots)\,. \tag{3.1}$$

Whenever it is required to represent functional dependencies between attributes of the relation, then they follow attributes of the relation:

$$\mathrm{R}\,(\mathrm{A}, \mathrm{B}, \ldots; \{\mathrm{A} \rightarrow \mathrm{CD}\}, \{\mathrm{BE} \rightarrow \mathrm{C}\})\,. \tag{3.2}$$

Set of attributes will be denoted as $\mathbb{X}, \mathbb{Y}, \mathbb{Z}$.

Relations are often represented in form of arrays/tables but it is important to remember that such a convenient representation is not an essential part of the relational approach. Representing relations in form of a table introduces notion of rows that are often used as a synonym for tuples. However tuples of relation and rows of a table differ significantly. Informally it can be said that rows of a table in a database management system, store simple data while values of elements of a tuple can be complex, e.g.

element of a tuple can be a tuple of some other relation[1]. A good example is a relation that stores information about students. Such relation may consist of student first name, last name and address which is another realtion consisting of street name, flat number, city and post code:

$$\text{Student}\,(\text{fname, lname, Addr}\,(\text{street, flat, city, pcode}))\,. \tag{3.3}$$

The same relation represented in a form of a single database table would, most likely, have the following structure:

$$\text{Student}\,(\text{fname, lname, street, flat, city, pcode})\,. \tag{3.4}$$

Additionally, relations store only data that represents some information so if there is no information, no data is stored in fact. In contrast, modern database management systems often store a special `NULL` value that represents no information[2]. Finally, there are no duplicate tuples in relation as well as in the result of relational operation, while database management system can store tables with duplicate rows and can return result sets with duplicates in response to SQL commands.

Table representation of n-ary relation R has the following properties [7]:

1. each row represents a tuple of R consisting of n elements,
2. the ordering of rows is immaterial,
3. all rows are distinct,
4. ordering of columns is insignificant – it corresponds to the ordering of domains S_j in Cartesian product but is not significant for interpretation of the relation,
5. significance of each column is conveyed by labelling it with the name of the corresponding domain.

Relation is subject to modifications – as time progress new tuples can be inserted into the relation, existing tuples can be deleted as well as some components in existing tuples can be modified. It is quite unlikely that in some time instant all values from domains S_j are stored in relation (columns of table). More likely relation consists of only some subsets of these domains. These subsests are called *active domain* at a particular instant of time.

[1]Nowadays there are exceptions from this rule as many DBMS start allow to store complex structures in a single table row, e.g. XML data

[2]According to SQL standards (e.g. [1], [2]) `NULL` is *A special value, or mark, that is used to indicate the absence of any data value.*

Remark – Nowadays, regardless of the definitions of *domain* and *active domain*, most databases use word *domain* (instead of active domain) to address set of values stored in a particular column of the table.

Normally, one attribute (or a combination of attributes) of the relation has values that uniquely identify each tuple. Such attribute (or set) is called a *primary key*. Primary key is non redundant. It means that no two tuples with the same values of primary key exist in the relation. There may be more then one set of attributes uniquely identifying each tuple of the relation. If so then primary key can be chosen arbitrarily while all possible primary keys are called *candidate keys*. When representing a relation attributes of the primary key are underlined. A social security number (SSN) is a good example of an attribute for primary key

$$\text{Student}\Big(\underline{\text{SSN}}, \text{fname}, \text{lname}, \text{Addr}\,(\text{street}, \text{flat}, \text{city}, \text{pcode})\Big). \tag{3.5}$$

Primary keys can be determined based on functional dependencies between attributes of relation. *Functional dependency* (FD) determines attributes (or sets of attributes) that unambiguously define other attributes. For example functional dependency

$$\text{AB} \rightarrow \text{C} \tag{3.6}$$

means that values of attributes A and B together clearly determine value of attribute C. In other words, if we take all tuples for which value of attributes A and B equals *a* and *b*, respectively, then value of attribute C for these tuples will be the same. There are five axioms regarding functional dependencies (so called *Armnstrong's axioms*):

1. decomposition: if $\mathbb{XY} \rightarrow \mathbb{ZP}$ then both set of attributes $\mathbb{Z}$ and $\mathbb{P}$ are uniquely determined by $\mathbb{XY}$, i.e.:

$$\mathbb{XY} \rightarrow \mathbb{ZP} \quad \Rightarrow \quad \mathbb{XY} \rightarrow \mathbb{Z} \;\; \text{and} \;\; \mathbb{XY} \rightarrow \mathbb{P}. \tag{3.7}$$

   However, observe that neither $\mathbb{X}$ nor $\mathbb{Y}$ itself uniquely determine $\mathbb{Z}$ and $\mathbb{P}$.

2. union:

$$\mathbb{X} \rightarrow \mathbb{Y}, \;\; \text{and} \;\; \mathbb{X} \rightarrow \mathbb{Z} \quad \Rightarrow \quad \mathbb{X} \rightarrow \mathbb{YZ}. \tag{3.8}$$

3. transitivity:

$$\mathbb{X} \rightarrow \mathbb{Y}, \; \mathbb{Y} \rightarrow \mathbb{Z} \quad \Rightarrow \quad \mathbb{X} \rightarrow \mathbb{Z}. \tag{3.9}$$

4. augmentation: for any set of attributes $\mathbb{X}, \mathbb{Y}, \mathbb{Z}$
$$\mathbb{X} \rightarrow \mathbb{Y} \quad \Rightarrow \quad \mathbb{X}\mathbb{Z} \rightarrow \mathbb{Y}\mathbb{Z}. \tag{3.10}$$
5. reflexivity:
$$\mathbb{Y} \subseteq \mathbb{X} \quad \Rightarrow \quad \mathbb{X} \rightarrow \mathbb{Y}. \tag{3.11}$$

There is no requirement that all attributes of the relation have to be part of some functional dependencies. While some attributes may appear in a number of FDs others may not be part of any FD at all. Following FDs can be defined in relation Student:

$$\begin{aligned} \text{SSN} &\rightarrow \text{fname, lname, Addr} \\ \text{pcode} &\rightarrow \text{city} \\ \text{street, flat, city} &\rightarrow \text{pcode} \end{aligned}$$

Functional dependencies are used to determine attributes of primary key for the relation. Primary key is the smallest possible set of attributes of a relation such that remaining attributes can be determined based on the attributes from the primary key and functional dependencies. Formally, primary key is defined as the smallest set of attributes $\mathbb{X}$ of a relation $R(\mathbb{X}\mathbb{Y})$ such that its closure $\mathbb{X}^+$ is a set of all attributes, i.e. $\mathbb{X}^+ = \mathbb{X}\mathbb{Y}$. *Closure* of the set of attributes $\mathbb{X}$ is determined based on functional dependencies. At first closure equals to the set of attributes, i.e.: $\mathbb{X}^+ = \mathbb{X}$. Closure is then **extended** with every attribute $Y \notin \mathbb{X}^+$ such that there is a functional dependency $X \rightarrow Y$ and $X \in \mathbb{X}^+$. This is repeated as long as there are no attributes outside closure or no new attribute can be added to it. Algorithm 1 presents details of the procedure.

Algorithm 1 Finding closure of set of attribiutes.

```
Require: relation $R(\mathbb{X}, F)$ where $\mathbb{X}$, F is set of attributes and functional dependencies
    respectively. Set of attributes $\mathbb{Y} \subseteq \mathbb{X}$.
Ensure: closure of set $\mathbb{Y}$.
1: set $\mathbb{Y}^+ = \mathbb{Y}$
2: for every functional dependency $\{\mathbb{A} \rightarrow \mathbb{B}\} \in F$ such that $\mathbb{A} \subseteq \mathbb{Y}^+$ do
3:     set $\mathbb{Y}^+ = \mathbb{Y}^+ \cup \mathbb{B}$,
4: end for
```

Example 3.1
Find closure A^+ in relation R(ABCD) with FDs $AB \rightarrow C$, $A \rightarrow B$ and $D \rightarrow C$. First, it follows from $A \rightarrow B$ that $A^+ = AB$. Later, from $AB \rightarrow C$ we have that $A^+ = ABC$. □

Primary key is the smallest set of attributes $\mathbb{X}$ such that its closure $\mathbb{X}^+$ contains all attributes of relation R. According to the definition, primary key in R from example 3.1 equals $\underline{\mathrm{AD}}$.

Example 3.2
Let R be a relation that stores information about cars registered in Poland

$$\mathrm{R}\left(\mathrm{VIN},\ \mathrm{numberPlate},\ \mathrm{colour},\ \mathrm{model}\right), \tag{3.12}$$

where VIN is a Vehicle ID Number. Since following FDs exist in this relation

$$\begin{aligned} \mathrm{VIN} &\rightarrow \mathrm{numberPlate},\ \mathrm{colour},\ \mathrm{model} \\ \mathrm{numberPlate} &\rightarrow \mathrm{VIN},\ \mathrm{colour},\ \mathrm{model}, \end{aligned}$$

therefore both VIN and numberPlate attributes are candidate primary keys. Either of them can be used as a primary key for R. □

As presented in the above example it may happen that there are several set of attributes that can be a primary key for the relation. Such possible primary keys are called *candidate keys*. When primary key is selected from candidate keys then all attributes that belong to primary key are called *prime attributes*, while remaining are refereed to as *non-prime attributes*. Set of attributes that includes all prime attributes is called *superkey*.

Example 3.3
Let R be a relation representing students, course and grade obtained

$$\mathrm{R}\left(\mathrm{ID},\ \mathrm{fname},\ \mathrm{lname},\ \mathrm{course},\ \mathrm{grade}\right). \tag{3.13}$$

There are two FDs in R:

$$\begin{aligned} \mathrm{ID} &\rightarrow \mathrm{fname},\ \mathrm{lname}, \\ \mathrm{ID},\ \mathrm{course} &\rightarrow \mathrm{grade}, \end{aligned}$$

therefore ID, course is a primary key for relation R

$$\mathrm{R}\left(\underline{\mathrm{ID}},\ \mathrm{fname},\ \mathrm{lname},\ \underline{\mathrm{course}},\ \mathrm{grade}\right). \tag{3.14}$$

□

A common requirement for tuples of relation R is to reference tuple (or tuples) of relation S (possibly $S = R$ means that reference concerns elements of the same relation). In other words reference denotes dependencies and connections between information stored in different relations (or the same relation) and are represented by *foreign keys*. *Foreign key* is an attribute (or set of attributes) of relation R that is not a primary key of R but its values are from the primary key of relation S. Note, that whereas foreign key in R is not a primary key of R it is possible that attributes of foreign key are proper subset of attributes constituting primary key in R. Because values of foreign key come from the primary key of relation S, therefore, active domains of foreign key in R can only have values that come from the active domains of primary key in S. A relation R can have zero or more foreign keys. Foreign keys in the same relation may have some attributes in common.

3.2 Relational algebra

Relations are sets so in general all of the usual operations that are applicable to sets are also applicable to relations. However, whereas the result of operation applied to sets is always a set this is not necessary true for relation. For example union of binary (two elements) relation and ternary (three elements) relation is not a relation anymore, since there are two or three attributes in resulting tuples. Operations of *relational algebra* act upon one or two operands that transform a single relation (or pair of relations) into a new relation. Although in relational algebra any operation outputs result that is a relation, this is not necessary true for contemporary database management system. Database management systems draw on relational algebra but for user convenience implement extended set of operands that may produce sets of data that are not relations. In particular every relational operator in relational algebra is designed to operate on and produce sets of data that are free of duplicate tuples. This is not necessary true for database management systems.

There is a number of relational operations that can be defined [8] but eight operations are basic:

- theta-select (σ) – selects a subset of tuples from relation.
- projection (π) – restricts list of attributes in tuples.
- Cartesian product ($\times$) – concatenates tuples from two relations.
- theta-join ($\bowtie^{\theta}$) – selects and concatenates tuples from two relations.
- natural join ($\bowtie$) – select and concatenates tuples from two relations removing duplicate attributes.

- union ($\cup$) – unionises two relations that store similar facts.
- intersection ($\cap$) – finds common part of two relations that store similar facts.
- difference ($\setminus$) – finds tuples that exit in one relation but do not exit in the second one.

Theta-select (σ) takes two arguments where one is a relation and the other is a condition that all returned tuples have to satisfy. If no comparator was specified then it is assumed by default that the equality comparator should be used. *Theta* in the name of the operator stands for the comparators that can be used with select operand. In contemporary database management systems all arithmetic comparators as well as logic operands and Boolean comparisons can be used.

Example 3.4
Suppose that the selection operator $\sigma_{\text{salary}>=10000}$ is applied to the following relation

R

emp_id	SSN	fname	lname	salary
12	ASW14324HM	John	Smith	10 000
13	QWE423145G	Alice	Pawn	12 000
14	TRQ12387TT	Bob	Knight	9 500
25	WRO65321TQ	John	Smith	10 000

As a result we get a relation that consist of three tuples: □

$\sigma_{\text{salary}>=10000}(R)$

emp_id	SSN	fname	lname	salary
12	ASW14324HM	John	Smith	10 000
13	QWE423145G	Alice	Pawn	12 000
25	WRO65321TQ	John	Smith	10 000

Projection (π) operator acts on a single relation and a list of its attributes that are to be projected. It generates a relation that consists all distinct tuples from the original relation but attributes of each tuple are limited to these that were listed as a command parameter. Duplicates (if any) are removed so that the resulting tuples form a relation.

Example 3.5
Suppose that the project operation $\pi_{\text{fname, lname, salary}}$ is applied to the following relation

R

emp_id	SSN	fname	lname	salary
12	ASW14324HM	John	Smith	10 000
13	QWE423145G	Alice	Pawn	12 000
14	TRQ12387TT	Bob	Knight	9 500
25	WRO65321TQ	John	Smith	10 000

Projecting only these three attributes will lead to the following relation (with no duplicate tuples). □

$\pi_{\text{fname, lname, salary}}(R)$

fname	lname	salary
John	Smith	10 000
Alice	Pawn	12 000
Bob	Knight	9 500

Cartesian product operator ($\times$) employs two relations R, S and generates a relation $R \times S$ that consist of all tuples from relation R concatenated with every tuple from S. If there are r and s tuples in relations R, S respectively, then the resulting Cartesian product contains $r \cdot s$ tuples. Therefore, Cartesian product operator is more of theoretical interest. It is used to define more complicated join operators, however, should never be used in real life database systems.

Theta-join operator ($\bowtie^{\theta}$) takes two relations R and S as an argument and outputs relation that contains tuples from relation R concatenated with tuples of relation S, but, in contrast to Cartesian product, tuples are concatenated only when specified condition is true. Condition that is used to decide whether two tuples are to be concatenated involves comparing attribute (or attributes) from tuple of relation R with attribute (or attributes) from every tuple of relation S. The attributes to be compared are explicitly indicated in join command. Similar as for theta-select operator, *theta* stands for one of possible comparators that can be used when defining the join condition.

If input arguments R, S for theta-join operator are relations then the result of operation is a relation too. In particular there are no duplicate tuples in the result and if there are any attributes in relations R and S that have the same name then names are preceded with relation name. If join condition is omitted then it is assumed that equality comparator is used and join is carried out with respect to all attributes with the same names in both relations.

Example 3.6
Lets assume we have two relations R and S

R

emp_id	name	home_city
12	John Smith	Washington
13	Alice Pawn	Princeton
14	Bob Knight	Seattle
25	John Smith	Warsaw

S

city	state
Washington	District Columbia
Princeton	New Jersey
Seattle	Washington

If we join both relations using condition R.city = S.city then we will get the relation $R \bowtie^{\theta}_{R.\text{home\_city}=S.\text{city}} S$ □

$R \bowtie^{\theta}_{R.\text{home\_city}=S.\text{city}} S$

emp_id	name	home_city	state	city
12	John Smith	Washington	District Columbia	Washington
13	Alice Pawn	Princeton	New Jersey	Princeton
14	Bob Knight	Seattle	Washington	Seattle

In theta-join attributes used in join condition are always included in tuples of the resulting relation. As a consequence resulting relation contains two attributes (or more if more then one attribute were used in join condition) that differ in names but have exactly the same value in each tuple. *Natural join* operator ($\bowtie$) is similar to theta-join however one of the redundant attributes is omitted from the result. Retained attribute is assigned a name of an attribute from the original relations and the name that comes first alphabetically is chosen. Natural join is the most useful join operator in theory of database systems, in particular it is used in normalisation of relations (see chapter 3.3.2).

Example 3.7
If we take relations R, S from previous example then natural join of these relation equals □

$R \bowtie_{R.\text{home\_city}=S.\text{city}} S$

emp_id	name	state	city
12	John Smith	District Columbia	Washington
13	Alice Pawn	New Jersey	Princeton
14	Bob Knight	Washington	Seattle

Union operator (∪) brings together in one relation all tuples from two relations that are its operands, provided that both of them contain the same facts. In particular to union relation R and S it is required that both of them have the same number and types of attributes. Moreover, any duplicate tuples are removed from the result of union operator. If above conditions are not meet then, due to different number of elements in tuples and/or different types of values in the same attribute, the result will not be a relation. When two relations can be unionised then we say that they are union-compatible.

Intersection operator (∩) is applicable to two relations R and S if they are union-compatible (i.e. conform requirements for being operands of union operator). The result of the operation consists of all tuples that appear in both original relations. Since R, S are relations thus $R \cap S$ does not contain duplicate tuples either.

Difference operator (\) is also applicable to two union-compatible relations. The result of applying relation difference to relations R and S (i.e. $R \setminus S$) is a relation that contains all the tuples from R except those that appear in S.

Example 3.8
Suppose we have two relations R and S:

R

emp_id	fname	lname
12	John	Smith
13	Alice	Pawn
14	Bob	Knight
25	John	Smith

S

cust_id	fname	lname
12	John	Smith
23	Alice	Pawn
14	Bob	Knight
36	John	Smith

The result of applying union, intersect and difference operators to these relation are presented on page 21. □

Relational operators mentioned above are basic operator that can be used to define more complex operators. For more details reader should refer to [8], while now we focus on three variants of join operator defined previously, that are widely used in modern relational database management systems. It is also important to notice that these operations are not valid in relational algebra as they introduce `NULL`[3] values which does not exist in relational model. These operators are called *outer joins* to distinguish them from join defined earlier that is also called *inner join*.

Left outer join of relations R and S (i.e. $R \ltimes S$) is a union of inner join $R \bowtie S$ and remaining tuples of R (that were rejected during inner join) expanded with S-like tuple

[3]According to SQL standards (e.g. [1], [2]) `NULL` is *A special value, or mark, that is used to indicate the absence of any data value.*

$R \cup S$

emp_id	fname	lname
12	John	Smith
13	Alice	Pawn
14	Bob	Knight
25	John	Smith
23	Alice	Pawn
36	John	Smith

$R \cap S$

cust_id	fname	lname
12	John	Smith
14	Bob	Knight

$R \setminus S$

emp_id	fname	lname
13	Alice	Pawn
25	John	Smith

$S \setminus R$

cust_id	fname	lname
23	Alice	Pawn
36	John	Smith

which values are set to NULL. In other words, left outer joins takes all rows from left relation (R) and concatenates them with corresponding tuples from right relation (S). If no corresponding tuple in S is found then tuple from R is concatenated with NULL values. Left outer join ensures that resulting relation contains all tuples from relation R. Similarly *Right outer join* operator (⋊) takes all tuples from right relation (S) and concatenates them with corresponding tuples from left relation (R) or NULL values if no corresponding tuple is found. In contrast to left outer join the result contains all tuples from relation S. *Full outer join* operator (⋈) is a union of left outer join and right outer join. The result contains all tuples from both relations that are arguments of this operand. Similarly to theta-join and natural join operator all outer joins take an additional condition that tuples of both relations have to satisfy in order to be included in the result. Resulting relation contains all attributes from both relations exactly as it was the case for theta-join operator.

Example 3.9
The relation R and S contain information about customers and suppliers respectively:

R

rec_id	name	address
12	John Smith	1600 Pennsylvania Ave, Washington
13	Alice Pawn	4 Wall Street, New York
14	Bob Knight	8949 Wilshire Boulevard, Beverly Hills

S

sup_id	name	address
12	John Smith	4 Wall Street, New York
13	Alice Pawn	4 Wall Street, New York
36	John Smith	25 Carleton Street, Cambridge

Following tables represent results of left, right and full outer joins of relations R and S:
□

$R ⟕_{\text{rec\_id=sup\_id}} S$

R.rec_id	R.name	R.address	S.sup_id	S.name	S.address
12	John Smith	1600 Pennsylvania Ave, Washington	12	John Smith	4 Wall Street, New York
13	Alice Pawn	4 Wall Street, New York	13	Alice Pawn	4 Wall Street, New York
14	Bob Knight	8949 Wilshire Boulevard, Beverly Hills	NULL	NULL	NULL

$R ⟖_{\text{rec\_id=sup\_id}} S$

R.rec_id	R.name	R.address	S.sup_id	S.name	S.address
12	John Smith	1600 Pennsylvania Ave, Washington	12	John Smith	4 Wall Street, New York
13	Alice Pawn	4 Wall Street, New York	13	Alice Pawn	4 Wall Street, New York
NULL	NULL	NULL	36	John Smith	25 Carleton Street, Cambridge

$R ⟗_{\text{rec\_id=sup\_id}} S$

R.rec_id	R.name	R.address	S.sup_id	S.name	S.address
12	John Smith	1600 Pennsylvania Ave, Washington	12	John Smith	4 Wall Street, New York
13	Alice Pawn	4 Wall Street, New York	13	Alice Pawn	4 Wall Street, New York
14	Bob Knight	8949 Wilshire Boulevard, Beverly Hills	NULL	NULL	NULL
NULL	NULL	NULL	36	John Smith	25 Carleton Street, Cambridge

3.3 Normal forms

In order to store relations in a database management system it is required to represent them in form of a two-dimensional column-homogenous tables. Since such tables can store only simple data types, therefore eliminating nonsimple domains/attributes from relations becomes essential. Moreover, the elimination procedure, called *normalisation*, allows to minimise the probability of delete, insert and update anomalies that may appear in relations with nonsimple attributes.

Normalisation is a step by step procedure that transforms relation from one normal form to another based on *functional dependencies* i.e. dependencies between set of attributes that arise from real live (cf. definition of functional dependencies on page 13). During this process relation $R(\mathbb{X})$ is decomposed into smaller relations $R_1(\mathbb{X}_1)$ and $R_2(\mathbb{X}_2)$ with attributes that are proper subset of original attributes (i.e. $\mathbb{X}_1, \mathbb{X}_2 \in \mathbb{X}$) and every original attribute belongs either to relation R_1 or R_2 (i.e. $\mathbb{X}_1 \cup \mathbb{X}_2 = \mathbb{X}$).

3.3.1 Decomposition of relations

Decomposition means that information stored in original relation will be also divided and pieces of information will be stored separately in new relations. If R is removed, then in order to reconstruct original information relations R_1 and R_2 have to be joined together. This is done through *natural join* operator ($\bowtie$) that is one of the basic operator in relational algebra. Natural join operand ($R_1 \bowtie R_2$) takes every tuple from relation R_1 and searches for corresponding tuple in R_2. Corresponding tuples are found by value comparison of attributes that are common for both relations. When found then tuple from R_1 is expanded with attributes from R_2 that are not in R_1. If there are $k > 1$ corresponding tuples in R_2 then k tuples will be added to the result of natural join operator but duplicate tuples are rejected.

Example 3.10
Table $R_1 \bowtie R_2$ represents a natural join of relation R_1 with R_2: □

R_1

A	B	C
1	4	12
2	1	12
2	3	9

R_2

A	C	D
2	12	3
1	12	4
1	12	5
2	9	5
2	9	2

$R_1 \bowtie R_2$

A	B	C	D
1	4	12	4
1	4	12	5
2	1	12	3
2	3	9	2
2	3	9	5

Decomposition of relation also requires to decompose and/or modify functional dependencies. This is obvious since functional dependencies of relation involve only attributes of the relation. Therefore, if we have $R(ABCD; \{D \rightarrow C\}, \{A \rightarrow D\})$ then functional dependency $\{D \rightarrow C\}$ is not valid in decomposed relation R_1 (ABC) since D is not an attribute of R_1. However, because $\{A \rightarrow D\}$ exist in original relation, thus $\{A \rightarrow C\}$ holds in R_1.

As a consequence, whenever relation is decomposed deduction of functional dependencies has to be done in order to find functional dependencies that are valid in decomposed relations. Algorithm 2 presents procedure for deducting functional dependencies.

Relation decomposition can be achieved in one of four ways:

- into independent relations – with no lose of data and functional dependencies,
- dependency preserving – with no lose of functional dependencies (i.e. data are lost),
- lossless – with no lose of data (i.e. functional dependencies are lost),
- with lose of data and functional dependencies.

When relation $R(\mathbb{X}, F)$ is divided into $R_1(\mathbb{X}_1, F_1)$ and $R_2(\mathbb{X}_2, F_2)$ with no loose of data and functional dependencies then:

1. all functional dependencies F^+ (i.e. that can be deducted from functional dependencies F) can be also deducted from functional dependencies in F_1 and F_2, that is
$$F^+ = (F_1 \cup F_2)^+ , \tag{3.15}$$

Algorithm 2 Deducting functional dependencies for decomposed relation

```
Require: relation R ($\mathbb{X}$, F) where $\mathbb{X}$, F is set of attributes and functional dependencies
    respectively, and its decomposition $R_1$ ($\mathbb{X}_1$) where $\mathbb{X}_1 \subset \mathbb{X}$.
Ensure: functional dependencies $F_1$ that hold in $R_1$ ($\mathbb{X}_1$)
 1: let $F^+$ be a set of dependencies deducted from R ($\mathbb{X}$, F),
 2: set $F^+ = F$,
 3: for every $\mathbb{A} \subseteq \mathbb{X}$ do
 4:     find closure $\mathbb{A}^+$,
 5:     for every $\mathbb{B} \in \mathbb{A}^+ \setminus \mathbb{A}$ do
 6:         if $\mathbb{A} \rightarrow \mathbb{B} \notin F^+$ and is a nontrivial functional dependency then
 7:             add $\mathbb{A} \rightarrow \mathbb{B}$ to $F^+$,
 8:         end if
 9:     end for
10: end for
11: leave in $F^+$ functional dependencies $\mathbb{A} \rightarrow \mathbb{B}$ such that both $\mathbb{A}$ and $\mathbb{B}$ are subset of
    $\mathbb{X}_1$.
```

2. natural join of relations R_1 and R_2 gives exactly the same data as stored in R (no data is lost and no new data is created).

When one of the above condition is not satisfied then decomposition loses data or functional dependencies respectively. If neither condition is satisfied then decomposition loses information about both functional dependencies and data.

In order to verify whether decomposition is *dependency preserving* it is required to check whether functional dependencies, that can be deducted from decomposed relations, are the same as those deducted from the original relation. Therefore, in general case we have to first decompose relation and then verify whether this decomposition preserves functional dependencies or not.

Formally we can say that decomposition of relation R($\mathbb{X}$; F) into relations $R_1(\mathbb{X}_1; F_1)$ and $R_2(\mathbb{X}_2; F_2)$ preserves functional dependencies if and only if:

1. $\mathbb{X}_1 \cup \mathbb{X}_2 = \mathbb{X}$,

2. $(F_1 \cup F_1)^+ = F^+$.

Example 3.11

Find functional dependencies in R_1 (*ACD*) and R_2 (*BCD*) that are decomposition of R (ABCD; {AB → C}, {A → B}, {D → C}).

We start with determining F^+ for relation R. According to algorithm 2 we take all possible subsets of attributes $\mathbb{X} = \{A, B, C, D\}$, calculate its closures and deduce functional dependencies:

A : $A^+ = \{A,B,C\} \Rightarrow \{A \rightarrow B\}, \{A \rightarrow C\}$,

B : $B^+ = \{B\}$,

C : $C^+ = \{C\}$,

D : $D^+ = \{C,D\} \Rightarrow \{D \rightarrow C\}$,

AB : $AB^+ = \{A,B, C\} \Rightarrow \{AB \rightarrow C\}$,

AC : $AC^+ = \{A,B, C\} \Rightarrow \{AC \rightarrow B\}$, but this is trivial dependency since $\{A \rightarrow BC\}$,

AD : $AD^+ = \{A,B, C, D\} \Rightarrow \{AD \rightarrow BC\}$, but this is trivial due to augmentation axiom and since both A and D themselves determine C,

...

Finally we get $F^+ = \{\{A \rightarrow BC\}, \{D \rightarrow C\}\}$ which yields R_1 (ACD; $\{A \rightarrow C\}, \{D \rightarrow C\}$) and R_2 (BCD; $\{D \rightarrow C\}$). Note that there is no functional dependency for attribute B which means that $\{A \rightarrow B\}$ (that holds in R) cannot be deducted from R_1, R_2. It follows that during this decomposition of R functional dependencies are lost. □

Unlike for decomposition with no lose of FD, where after the decomposition it can be verified whether it preserves dependency or not, there are rules that ensure decomposition of relation will be *lossless* (will preserve data). Precisely, relation $R(\mathbb{X}, F)$ is decomposed with no lose of data, into $R_1(\mathbb{Y}\mathbb{Z}_1, F_1)$ and $R_2(\mathbb{Y}\mathbb{Z}_2, F_2)$ if and only if:

1. $\mathbb{Y} \cup \mathbb{Z}_1 \cup \mathbb{Z}_2 = \mathbb{X}$,
2. $\mathbb{Z}_1 \cap \mathbb{Z}_2 = \emptyset$,
3. $\mathbb{Y} \rightarrow \mathbb{Z}_1 \in F$.

Example 3.12
Decomposition of R (ABCD; $\{AB \rightarrow C\}, \{A \rightarrow B\}, \{D \rightarrow C\}$) into R_1(ACD; $\{A \rightarrow C\}, \{D \rightarrow C\}$) and R_2 (BCD; $\{D \rightarrow C\}$) does not preserve data. It is so since for this decomposition $\mathbb{Y} = \{CD\}$, $\mathbb{Z}_1 = \{A\}$, $\mathbb{Z}_2 = \{B\}$ and these sets of attributes don't satisfy third requirement for lossless decomposition.

R

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2	c_1	d_1

R_1

A	C	D
a_1	c_1	d_1
a_2	c_1	d_1

R_1

B	C	D
b_1	c_1	d_1
b_2	c_1	d_1

The fact that this decomposition does not preserve data can be presented on a simple example with three relations R, R_1 and R_2. It can be easily verified that $R_1 \bowtie R_2$ has two additional tuples when compared to original relation R.

Remark – Joining decomposed relation may be a good method to show that decomposition is in fact lossy. However, it is not a method to decide whether decomposition is lossless or not. Mismatched example may falsely show decomposition is lossless.

$\square$

3.3.2 Normalisation

Normalisation is a systematic process of transforming relations in order to ensure that their structure is suitable for database management systems, table-like representation and ensuring data integrity (e.g. free of insert, update and delete anomalies). Since 1970, when Edgar F. Codd [7] proposed first normal form (1NF), database theorist proposed seven additional normal forms (2NF, 3NF, BCNF, 4NF, 5NF, DKNF, 6NF) among which first five are the most significant.

Relation is in *first normal form (1NF)* if it can be stored in a two-dimensional, column-homogenous table that stores data of simple types. To represent relation in 1NF it is required to:

- eliminate repeating data (information redundancy) in individual tuples,
- create separate relations for each set of data (each entity),
- designate a primary key for each relation.

First normal form focuses on the shape of a tuple which must contain constant number of attributes. It excludes existence of optional attributes which are used only in some tuples while they are left empty in others. Instead, if variable amount of similar information has to stored it is required to create separate relations for each set of information.

EXAMPLE 3.13
Lets assume that we want to store information about employees, access level, permissions at given level and their current and last salary. This information can be stored in relation R (fname, lname, access, perm, salary, prevsalary) but what happens when we add a requirement to store last but one salary also? Whereas adding additional attribute is the simplest answer it is not the solution since in practice it requires modification of database structure and program. On the other hand, we may separate information into two relations and add primary key as follows:

$$\mathrm{R}_1\left(\underline{\text{emp\_id}}, \text{fname, lname, } \underline{\text{access}}, \text{perm}\right),\ \ \mathrm{R}_2\left(\underline{\text{emp\_id}}, \text{salary}\right) \tag{3.16}$$

then we can smoothly accommodate a dynamic number of salaries. Additional attribute valid_until in relation R_2 can be used to distinguish between current and previous salaries. Both relations $\mathrm{R}_1, \mathrm{R}_2$ are in 1NF. □

Relations that are in 1NF can still feature anomalies. Lets consider relation $\mathrm{R}_1\left(\underline{\text{emp\_id}}, \text{fname, lname, } \underline{\text{access}}, \text{perm}\right)$ from example 3.13. In this relation insert anomaly occurs since there is no possibility to store information about an employee that has no access level and permissions. When updating information on employee last name that has several access level and we forget to update all entries for this user then update anomaly occur. Finally if an employee quits the company and related tuples are deleted then we will also lose information about access levels and permissions he had.

Second normal form (2NF) focuses on functional dependencies between primary key and non-prime attributes (i.e. attributes that are not part of primary key). In second normal form all non-prime attributes have to be full functionally dependent on primary key. Full functional dependency on primary key means that there is no non-prime attribute that depends on part of the primary key. Formally, if $\mathrm{R}(\mathbb{X}, \mathrm{F})$ and $\mathbb{Y} \subset \mathbb{X}$ is a candidate key then for every set of attributes $\mathbb{Z} \subseteq \mathbb{X} \setminus \mathbb{Y}$:

$$\mathbb{Y} \to \mathbb{Z} \text{ and } \neg\exists\, \mathbb{U} \subset \mathbb{Y} : \mathbb{U} \to \mathbb{Z} \tag{3.17}$$

Lets focus on relation R_1 from example 3.13. There are two functional dependencies in this relation: {emp_id → fname, lname} and {emp_id, access → perm}. There is only one candidate key in this relation, namely emp_id, access which means that attributes fname, lname are functionally dependent on part of the primary key. Therefore R_1 is not in 2NF.

Relation that is in 1NF can be transformed to 2NF through normalisation procedure. Normalisation is a relation decomposition procedure presented in algorithm 3. Note that this procedure is the same for normalisation between 1NF, 2NF, 3NF and BCNF, as well as it is a lossless decomposition of relation (cf. lossless decomposition requirements on page 25).

Algorithm 3 Relation normalisation

```
Require: relation $R(\mathbb{X}, F)$ in $x$NF to be normalised to $(x+1)$NF
Ensure: relations $R_i(\mathbb{X}_i, F_i)$ that are decomposition of $R(\mathbb{X}, F)$ and are in $(x+1)$NF
1: find functional dependency $\mathbb{Y} \rightarrow \mathbb{Z}$ that violates condition for $(x+1)$NF,
2: find closure $\mathbb{Y}^+$,
3: decompose R based on this functional dependency into relations:

       $R_1(\mathbb{Y}^+, F_1)$,

       $R_2((\mathbb{X} \setminus \mathbb{Y}^+) \cup \mathbb{Y}, F_2)$,

   where $F_1, F_2$ are functional dependencies deducted for decomposed relations,
4: if $\mathbb{R}_i$ is not in $(x+1)$NF then
5:     normalise $\mathbb{R}_i$
6: end if
```

Example 3.14

Relation R($\underline{\text{emp\_id}}$, fname, lname, $\underline{\text{access}}$, perm; {emp_id → fname, lname}, {emp_id, access → perm}) is not in 2NF since attributes fname, lname are not full functionally dependent on the primary key. Therefore, functional dependency {emp_id → fname, lname} violates the 2NF and relation needs to be decomposed.

1. find closure of emp_id:

$$\text{emp\_id}^+ = \{\text{emp\_id, fname, lname}\}$$

2. deduce all nontrivial functional dependencies from R:

$$\{\text{emp\_id} \rightarrow \text{fname, lname}\}$$
$$\{\text{emp\_id, access} \rightarrow \text{perm}\}$$

3. decompose R into:

$$R_1(\underline{\text{emp\_id}}, \text{fname, lname}, \{\{\text{emp\_id} \rightarrow \text{fname, lname}\}\})$$
$$R_2(\underline{\text{emp\_id}}, \underline{\text{access}}, \text{perm}, \{\{\text{emp\_id, access} \rightarrow \text{perm}\}\})$$

that are both in 2NF.

□

Remark – If primary key of relation consist of one attribute only then relation is in 2NF.

In 2NF every non-prime attribute is fully dependent on the candidate key, however, such a functional dependency can be transitive. Transitive dependency $A \rightarrow C$ means that attribute C depends functionally on some other attribute B which in turns depends on A. Transitive dependency means that there is no direct functional dependency but such a dependency can be deducted.

Example 3.15
Lets extend relation R_1 from example 3.14 with additional attribute fnameinitial. Value of new attribute functionally depends on fname attribute, so new relation equals

$R_1(\underline{\text{emp\_id}}, \text{fname}, \text{lname}, \text{fnameinitial}, \{\text{emp\_id} \rightarrow \text{fname}, \text{lname}\}, \{\text{fname} \rightarrow \text{fnameinitial}\})$.

Since {emp_id → fname} and {fname → fnameinitial} thus there is a transitive dependency {emp_id → fnameinitial}. Therefore, R_1 is in 2NF but not in 3NF. □

Transitive dependencies in 2NF may lead to insert, update and delete anomalies and should be removed through normalisation to 3NF.

In *third normal form (3NF)* all non-prime attributes are directly functionally dependent on every primary key or superkey (set of attributes that contains primary key). Formally, relation R(ABCD; F) is in 3NF if and only if:

- it is in 2NF,
- for every functional dependency $\mathbb{X} \rightarrow \mathbb{Y}$ in F one of the following conditions is satisfied:
  - $\mathbb{Y}$ is a subset of $\mathbb{X}$ – if so then according to reflexivity axiom (see Armstrong's axioms, page 13) this functional dependency is trivial, or
  - $\mathbb{X}$ is a superkey – if so and due to the fact that R is in 2NF and augmentation axiom this functional dependency is trivial, or
  - $\mathbb{Y}$ is a subset of primary key of the relation R.

Based on Amstrong's axioms if either of two first conditions hold then functional dependency $\mathbb{X} \rightarrow \mathbb{Y}$ is trivial and can be simply dropped. Third condition means that in 3NF primary key (or subset of its attributes) can be functional dependent on non-prime attribute of the relation. It is also allowed that there exist a functional dependency between different attributes of primary key.

Example 3.16
Relation

$\text{R}(\underline{\text{emp\_id}}, \text{fname}, \text{lname}, \text{fnameinitial}, \{\text{emp\_id} \rightarrow \text{fname}, \text{lname}\}, \{\text{fname} \rightarrow \text{fnameinitial}\})$.

is in 2NF and isn't in 3NF due to the fact, that fnameinitial is in transitive functional dependency on the primary key. Relation R can be decomposed with respect to $\{\{\text{fname} \rightarrow \text{fnameinitial}\}\}$ yielding relations:

$$\text{R}_1(\underline{\text{fname}}, \text{fnameinitial}, \{\{\text{fname} \rightarrow \text{fnameinitial}\}\})$$
$$\text{R}_2(\underline{\text{emp\_id}}, \text{fname}, \text{lname}, \{\{\text{emp\_id} \rightarrow \text{fname}, \text{lname}\}\})$$

that are both in 3NF. □

Since in 3NF every non-prime attribute is directly functional dependent on the primary key, therefore, update anomaly will not occur for non-prime attributes. However, since attributes of primary key can still be functionally dependent on other attributes thus insert and update anomaly may still occur.

Boyce-Cood normal form (BCNF) was proposed to eliminate update, insert and delete anomalies. This is achieved through eliminating functional dependencies between non-prime attributes and prime attributes as well as between attributes of the primary key. Formally, relation is in BCNF if and only if all functional dependencies result from primary key (i.e. primary key is on the right side of every functional dependency).

Example 3.17
Relation $\text{R}(\underline{\text{street}}, \underline{\text{city}}, \text{zip\_code}, \{\text{street}, \text{city} \rightarrow \text{zip\_code}\}, \{\text{zip\_code} \rightarrow \text{city}\})$ is in 3NF since all non-prime attributes (i.e. zip_code) are directly functional dependent on the primary key. However, since there is a functional dependency between non-prime attribute and primary key attribute (i.e. city), therefore, relation is not in BCNF. When we decompose relation R with respect to functional dependency $\{\text{zip\_code} \rightarrow \text{city}\}$ then we will get the following relations:

$$\text{R}_1(\underline{\text{zip\_code}}, \text{city}, \{\{\text{zip\_code} \rightarrow \text{city}\}\})$$
$$\text{R}_2(\underline{\text{zip\_code}}, \underline{\text{street}})$$

Decomposed relations are in BCNF and this decomposition is lossless. However, functional dependency $\{\{\text{street}, \text{city} \rightarrow \text{zip\_code}\}\}$ was lost and cannot be deducted from R_1 and R_2. □

As presented in example 3.17 normalisation to BCNF always leads to lose of functional dependencies. It is so, since functional dependencies in 3NF (where every non-prime attribute depends on the primary key – $\{\mathbb{X} \rightarrow \mathbb{Y}\}$), that do not satisfy BCNF, have non-prime attribute on left and prime attribute on the right side (i.e. $\{A \rightarrow B\}$ where $B \subseteq \mathbb{X}$ and $A \subseteq \mathbb{Y}$). When decomposing relation according to algorithm 3 and with respect to $\{A \rightarrow B\}$ then non-prime attribute A occurs in both decomposed relations while prime attribute B appear only in one of them (in the other one $\mathbb{X} \setminus B$ appears). However, since prime attributes $\mathbb{X}$ do not appear in either of decomposed relation, therefore functional dependency $\mathbb{X} \rightarrow B$ does not belong to any of these relations. Moreover, it cannot be deducted from functional dependencies in either of decomposed relations, since non of them consist of all prime attributes $\mathbb{X}$.

Fourth normal form (4NF) deal with multivalued dependencies. Despite similarities *multivalued dependencies* are not functional dependencies and therefore are represented with two-head arrow (i.e. if $\{A \twoheadrightarrow B\}$ then $A \not\twoheadrightarrow B$). Multivalued dependency $\{A \twoheadrightarrow B\}$ does not mean that A clearly determines B but that A relates to set of values of attribute B. If there are multivalued dependencies in relation then again update, insert and delete anomalies may occur. This will happen if there is, in relation $R(\mathbb{X}; F)$:

- more then one nontrivial multivalued dependency in relation, or
- one multivalued dependency such that $\mathbb{Y} \twoheadrightarrow \mathbb{Z}$ and $\mathbb{Y} \neq \mathbb{X} \setminus \{\mathbb{Z}\}$.

Intuitively, relation $R(\mathbb{X}; F)$ is in 4NF when all functional and multivalued dependencies result from relation key – in BCNF this requirement only applied to functional dependencies. Relation that is in BCNF can be thus decomposed to 4NF using algorithm 3. Decomposition does not necessarily preserve functional dependencies.

Example 3.18
Lets consider relation R(emp_id, language, skill) that stores information about languages and skills of employees. Since there are no functional dependencies in this relation, therefore, all tree attributes form primary key and so R is in BCNF. Nevertheless there are two multivalued dependency

$$\{\{\text{emp\_id} \twoheadrightarrow \text{language}\}\}$$
$$\{\{\text{emp\_id} \twoheadrightarrow \text{skill}\}\}$$

since each employee may speak several languages and posses several skills.

Lets assume that we have two employees: Alice (emp_id = 1) and Bob (emp_id = 2). Alice speaks English, French and Polish and holds certificates in accounting and project management. Bob speaks English and German and used to work as bricklayer, plumber and painter. This information can be stored in relation R as follows
Despite fact that this relation is in BCNF it still features anomalies. For example assume that Alice doesn't speak French. If we delete corresponding tuple then we also

R

emp_id	language	skill
1	English	accounting
1	French	project management
1	Polish	
2	English	bricklayer
2	German	plumber
2		painter

lose information about Alice skill – delete anomaly. Similarly, if Alice learns Japanese then we have to add new tuple with empty (or set to `NULL`) skill attribute – insert anomaly.

If we decompose relation R with respect to one of the multivalued dependency then:

$$\mathrm{R}_1(\underline{\text{emp\_id}}, \underline{\text{language}}; \{\text{emp\_id} \twoheadrightarrow \text{language}\}),$$
$$\mathrm{R}_2(\underline{\text{emp\_id}}, \underline{\text{skill}}; \{\text{emp\_id} \twoheadrightarrow \text{skill}\}),$$

and both relations are in 4NF. Note that in this example decomposed relations $\mathrm{R}_1, \mathrm{R}_2$ are exactly the same independently of whether decomposition was conducted with respect to {emp_id $\twoheadrightarrow$ language} or {emp_id $\twoheadrightarrow$ skill}. This is not the case in general. □

Table 3.1: Summary of normal forms

Normal form	Description
1NF	Relation has no repeating attributes, each relation represents single facts, each attribute has simple domain (no complex data types)
2NF	Every non-prime attribute is fully functional dependent on relation candidate key (this functional dependency may be transitive). If candidate key consist of exactly one attribute then relation is in 2NF. Decomposition to 2NF is lossless and dependency preserving.
3NF	Every non-prime attribute is directly functional dependent on candidate key (there is no transitive dependency on any candidate key). Decomposition to 3NF is lossless and dependency preserving.
BCNF	Every functional dependency has primary key on the left side. There are no non-trivial functional dependencies between non-prime and prime attributes and between two (or more) prime attributes. Decomposition to BCNF is lossless but does not preserve functional dependencies.
4NF	Every functional and non-trivial multivalued dependency result from relation primary key (have key on the left side). Decomposition to 4NF is lossless but does not necessarily preserve functional dependencies.

Chapter 4

Relational database management systems

Relational database management system draw on relational model presented in chapter 3, however, real life implementation have extended some of its aspects. Also some requirements resulting from relational model have been alleviated in order to develop systems that are more convenient. Due to differences between relational model and database management systems different names were introduced. In particular tables reflect relations from the relational model; columns – attributes; rows – tuples, and data types – domains. Also a number of additional elements (database objects) have been added to further simplify and improve properties of database-based system.

4.1 Structured Query Language

4.1.1 BNF syntax - general SQL notation

BNF (Backus Normal Form) is a notation for representing syntactics of the Structured Query Language. In practice SQL uses extended version of BNF which defines each semantic element of language using characters, character strings, other syntactic elements and symbols. According to [1] BNF used to represent SQL supports following symbols:

<> angle brackets delimit the names of syntactic elements i.e. non-terminal symbols of the SQL that are defined somewhere else,

::= is a definition operator that separates the element defined (that appears to the left of this symbol) from its definition (that appears to the right),

[ ] square brackets indicate optional elements in the formula so elements within the brackets can be either explicitly specified or omitted,

{ } braces indicate group of elements (parts) in the formula, it behaves similarly to brackets in traditional algebra,

| is an alternative operator which means that parts of the formula preceding and following the operator are alternative to each other – either preceding of following element appears in the formula. This operator is usually enclosed in braces or square brackets to specify alternative for some elements of the formula,

... lower dots indicate that the element to which they apply may be repeated in the formula any number of times. If lower dots follow the closing brace then they apply to the whole group enclosed in braces. When they follow the element then they apply to this element only,

!! double exclamation introduces English text when the formula is expressed in text rather then BNF notation.

Square brackets and braces can be nested to any depth as well as alternative operator may appear at any depth.

To read the definition of SQL element and/or generate a valid SQL command it is required to transform the element definition from the BNF notation to a correct instance of an SQL element. This can be done according to the algorithm 4.

SQL language is composed of three basic sets of commands:

- data definition language (DDL) - used to create, modify and delete objects in database,
- data manipulation language (DML) - used to manipulate the data stored in the database,
- data control language (DCL) - used to control privileges required to access the data.

Algorithm 4 Generating an instance of SQL element from BNF definition of the element.

```
Require: BNF definition of the SQL element.
Ensure: instance of the element.
 1: for every alternative part in the BNF definition select one of possible alternatives
    and replace the alternative part with it,
 2: replace each lower dots with one or more instances of element they apply to,
 3: for every part of the definition enclosed with square brackets, either delete this part
    and brackets or change square brackets to braces,
 4: for every part of the definition enclosed in braces do
 5:     generate an instance of SQL from this part of definition. Algorithm 4 is executed
        recursively with part of the definition as its input argument.
 6: end for
 7: for every non-terminal element in the definition do
 8:     find definition of the non-terminal element,
 9:     generate an instance of SQL from BNF definition of the non-terminal element.
        Algorithm 4 is executed recursively with definition of the non-terminal element
        as its input argument.
10: end for
```

4.1.2 Data Definition Language

Data definition language (DDL) consist of statements that enable to create, modify and delete database objects such as tables, views, stored procedures, triggers, indices, relation and cryptographic keys, sequences and so on. Three basic DDL commands are:

- CREATE - used to create new objects in the database,
- ALTER - used to modify objects that already exist in the database,
- DROP - used to remove objects from the database.

In all database management systems executing DDL statements is only restricted to users that hold `admin` privilege (e.g. `ddladmin`) but in some cases additional permissions may be required as well. For example a REFERENCE privilege is required to create view with SCHEMABINDING clause (cf. chapter 6.1.1).

4.1.3 Data Manipulation Language

Data manipulation language (DML) is the most often used set of SQL statements. DML is used to actually manipulate the data stored in database tables either directly or indirectly (e.g. through data inserts, deletes to/from views). There are four basic DML commands:

- SELECT - used to select data stored in a table,
- INSERT - used to insert new data into a database table,
- UPDATE - used to modify the data already stored in a table,
- DELETE - used to remove selected data from the table.

Except for this basic commands DML provides a number of various operators and functions that enable to operate the data efficiently. In particular contemporary database management systems enable standard software constructions such as loops (for, while) and conditional branches (if). Aggregate functions, standard mathematical, string and date operators are also supported. Similar to DDL execution of DML statements is subject to privileges but the difference is in the granularity of privileges. Access rights control execution of each basic DML command separately thus enabling the particular user to be granted ability to execute only some commands (cf. chapters 6.1.1 and 6.1.2).

4.1.4 Data Control Language

Data control language (DCL) is used to create users, assign them to groups (roles) and define privileges they are entitled to use. DCL will differ significantly depending on whether discretionary or mandatory access control is used. When discretionary access control is used (which is the case for most RDBS) then there are two basic commands used to control privileges:

- GRANT - used to give new privilege to a user or group,
- REVOKE - used to cancel privilege that was given earlier.

Some DBMS (e.g. MsSQL) also support DENY command that prohibits users/groups from using some privileges. If DENY command is supported then REVOKE cancels both granted and denied privileges.

Granularity and types of privileges differ between different DBMS, however, there is a set of privileges that all such systems have in common, these are privileges concerning:

- data manipulation operations such as INSERT, UPDATE, DELETE and SELECT,
- database objects such as CREATE, DROP, ALTER, REFERENCE, CONTROL, EXECUTE, etc.

More details on access control can be found in chapter 6.1.

4.2 Database basics

4.2.1 Data types

Data type in database management system is equivalent to domain in relational model so it defines a set of values that can be represented in a table's column. Traditionally value determined by data type was atomic so there was no logical subdivision of it [1] – these data types are called predefined or "build-in". However, current SQL standards (i.e. [2]) extended capabilities and introduced collection types such as array and multiset types.

There are 17 basic predefined data types that can be divided into 7 groups [2]:

- character data types - CHARACTER, CHARACTER VARYING, CHARACTER LARGE OBJECT (CLOB),
- binary data type - BINARY LARGE OBJECT (BLOB),
- integer data types - SMALLINT, INTEGER, BIGINT,
- non-integer data types - NUMERIC, DECIMAL, FLOAT, REAL, DOUBLE PRECISION,
- data and time data types - DATE, TIME, TIMESTAMP,
- boolean data type - BOOLEAN,
- interval data type - INTERVAL.

For each data type there is a set of rules and restrictions that apply when reading, writing, assigning or comparing values of this type. Obvious and the most important is the set of values that is allowed by data type. The value can be either `NULL` value or non-`NULL` value. A *`NULL`* value is a special value that is a member of every data type distinct from all non-`NULL` values. It has no literal associated with it although the keyword `NULL` is commonly used to indicate its value (e.g. to distinguish `NULL` value from empty character string). NULL value behaves similarly to NaN (not-a-number) in traditional arithmetic that is:

- two NULL values are incomparable,
- any arithmetic operator outputs NULL when at least one of its arguments is NULL,
- concatenation of character strings outputs NULL when one of strings is NULL,
- character string equal to NULL is of the length NULL.

CHARACTER string is a sequence of characters taken from a single character set of positive, non-negative length. Columns that are of character type are described by four elements:

- the name of the data type – that is CHARACTER, CHARACTER VARYING, CLOB,
- the length (for CHARACTER data type) or the maximum length (for CHARACTER VARYING and CLOB) of the character string,
- the character set – the set of characters that can be put into the character string,
- collation name – that is the name of the method that will be used to compare different character strings.

There are also there variants of standard character types – NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, NATIONAL CHARACTER LARGE OBJECT. The only difference to the standard character types is that they have an implementation-defined character set. The difference between CHARACTER and CHARACTER VARYING data types of length n is that the former one forces all character strings to be of length n. It means that storing short character strings in CHARACTER(n) data type column appends the string with white spaces to the total length of n. For columns of both data types, if the length of the assigned character string is grater than n, then string will be either truncated (if truncated characters are all white spaces) or exception will be raised. In general assigning, comparing and operating on character strings is possible only when strings are of the same character set. Strings of different character sets can be also compared if there is a collation that is applicable to both character sets. A number of functions for operating on character strings is also defined in the standard, the most commonly used are:

- || – is a concatenation operator that joins characters of its operands,

- SUBSTRING – returns part of the character string determined by either starting position and length or pattern and escape string (with keywords SIMILAR and UNESCAPE),
- OVERLAY – replaces any occurrence of a substring with new substring,
- LOWER and UPPER – converts all to characters to lower and upper case respectively,
- TRIM – removes all leading and trailing white spaces from the string,
- CHAR_LENGTH – returns number of characters in the character string,
- POSITION – returns position of first occurrence of substring in given string,
- [NOT] LIKE – returns Boolean true or false depending on whether character string matches the given pattern. Pattern can contain percent (%) and underscore (_) character to escape any sequence of characters and a single character respectively,
- [NOT] SIMILAR – is an extended variant of [NOT] LIKE that uses regular expressions to define the pattern.

BINARY LARGE OBJECT (BLOB) is a sequence of bytes that has neither character set nor collation associated with it and it is only described by a data type and maximum length of the binary string expressed in bytes. As a result, all binary strings are assignable. Similarly to character strings, binary strings can be truncated when stored in BLOB column but only when all truncated bytes are equal zero. Otherwise, exception is raised. Binary string can be also compared. However, they can be only compared for equality and two binary strings are equal if they have the same length. SQL standard defines concatenation operator as well as SUBSTRING, OVERLAY, TRIM, OCTET_LENGTH, POSITION and [NOT] LIKE functions for BLOBs that are analogous to functions defined for character strings.

Numeric data types are used to store integer and real numbers. Columns that are of numeric type are parametrised by four properties:

- name of the numeric type – that is NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION,
- precision of the numeric type,
- scale of the numeric type if it is an exact numeric type,

- indication of whether precision and scale are represented in decimal or binary.

Depending on the storage method numeric types can be divided into exact and approximate numeric types. An *exact numeric type* (NUMERIC, DECIMAL, INTEGER, SMALLINT) has two parameters: precision and scale. Precision is a positive number grater then 0 that determines number of significant digits in a numeric type. Precision is the maximal number of digits that can be used to represent an exact numeric value, that can be stored both on the left and right of the decimal point. Scale is a non-negative (can be equal 0) number that determines number of digits that can be stored to the right of the decimal point. When scale equals 0 then numeric type stores integer values. Exact numeric types store all numbers in the data type range exactly.

Value of an *approximate numeric type* (FLOAT, REAL, DOUBLE PRECISION) consist of a mantissa (M), an exponent (E) and a sign bit (S). Mantissa is a signed numeric value while exponent is signed integer that specifies the magnitude of the mantissa with respect to the particular (usually binary) radix (β). Precision for an approximate numeric values is defined as a number of significant bits that are used to represent the mantissa. Value of a approximate numeric data type is the mantissa multiplied by the radix to the exponent:

$$(-1)^S \cdot M \cdot \beta^E \tag{4.1}$$

Approximate numeric type cannot represent exactly all values in the data type range, which means that some of them are rounded or truncated towards the value that can be represented. In contemporary databases approximate numeric types use standard representation of floating point numbers defined in IEEE 754 [4].

SQL standard defines rounding and truncation as a procedures to transform numeric values cannot be represented with given numeric data type. These procedures are used whenever value cannot be represented in an approximate data type or value has larger scale then the exact numeric type. Value t obtained by truncation for a given numeric type is not further from zero then the original value v that was truncated. The absolute value of t is thus smaller or equal to v and the difference between v and t is less then the difference between two successive values of that numeric type. In contrast to truncation value r that is a result of rounding is the closest numeric value that can be represented in a given numeric type. The difference between v and r is less then the half of the difference between two successive values of the numeric type. If there are two values r_1 and r_2 satisfying the above condition then it is implementation-dependent which value is taken.

Using exact numeric types is somehow more intuitive since precision and scale refer to number of decimal digits. Therefore, if exact numeric data type can store numbers with precision P and scale S then all numeric values that have at most $P - S$

decimal digits to the left and S decimal digits to the right of the decimal point, can be stored exactly - i.e. exactly the same value will be read. On the other hand if approximate numeric type is used then even simple decimal numeric values may be rounded/truncated. This is due to the fact that all contemporary database management systems (as well as almost all computer systems) use IEEE 754 standard that represents mantissa in binary radix. Unfortunately, P binary digits may be not enough to represent numeric value that has even a few decimal digits to the right of the decimal point. Consequently it is a common misunderstanding that single/double precision IEEE 754 numeric can store decimal numbers with 7/15 decimal digits to the right of the decimal point.

Example 4.1
IEEE 754 stores normalised numeric values (other then 0, NaN, $\pm\infty$, etc.) using the unsigned mantissa $1 \leq M < 2$ (only fractional part is stored), unsigned exponent E encoded with bias B and a single bit S to represent the sign of the value. Number of bits required for storage of numeric value equals 32 bits for single precision (1 bit to store S, 8 bits for E and 23 bits for M) and 64 bits for double precision (1 bit to store S, 11 bits for E and 52 bits for M). Exponent bias equals +127 and +1023 for single and double precision respectively. The value of the numeric stored in single/double precision IEEE 754 format is calculated as follows:

$$(-1)^S(1 + M) \cdot 2^{E-B}. \tag{4.2}$$

When storing decimal value $(7.1)_{10}$ in IEEE 754 numeric format we will get

0 10000001 11000110011001100110011

and

0 10000001 1100011001100110011001100110011001100110011001100110

for single and double precision representation respectively. If we now calculate value of both numerics we get:

$$(-1)^0(1 + 0.774999976158142089843 75) \cdot 2^{129-127} = 7.0999999... \tag{4.3}$$

for single precision format, and

$$(-1)^0(1 + 0.77499999999999991118215802998748) \cdot 2^{129-127} = 7.0999999... \tag{4.4}$$

for double precision format. It can be clearly seen that approximate numeric data types cannot store exact value of such simple numeric and may introduce errors especially when used in some computations. □

Using approximate numeric data type may have serious consequences to the result of the computations performed in database management systems. The following example works for MsSQL Server 2008.

EXAMPLE 4.2
Assume we have a table T that consist of two columns: AmountR and AmountD, of **real** and **decimal(18,2)** types respectively. At first this table is empty and we execute the following SQL command:

```
INSERT INTO T (AmountR, AmountD) VALUES (0.01, 0.01).
```

Now selecting the data with

```
SELECT AmountR, AmountD FROM T
```

command returns correct values of AmountR=0.01 and AmountD=0.01. We may conclude that both values are stored correctly, however, if we execute the following command:

```
SELECT 100000000000*AmountR - 100000000000*AmountD FROM T
```

then we get -64 as a result. This indicates that values stored in both columns are interpreted differently. Similar difference will occur when we use aggregation function such as SUM or AVG. For example command

```
SELECT SUM(AmountR) AS SumOfAmountR, SUM(AmountD) AS SumOfAmountD
FROM T
```

returns SumOfAmountR=0.00999999977648258 while SumOfAmountD=0.01 even if there is only one row in this table! □

Database management systems provide a wide range of operands and functions for numeric variables that can be divided into three groups

- type converting functions – these functions can be used to convert particular numeric data type to other numeric data types (e.g. integer value to exact/approximate numeric value) as well as to other data types (e.g. character, binary, money etc.),
- mathematical operands – that are used to perform calculations on numeric values,
- aggregate functions – that are used to calculate aggregate values for selected sets of data.

There are two variants of conversions between data types: implicit and explicit. Implicit conversions are those that occur without specifying the conversion function (e.g.

1. user-defined (highest)	2. sql_variant	3. xml
4. datetime	5. smalldatetime	6. float
7. real	8. decimal	9. money
10. smallmoney	11. bigint	12. int
13. smallint	14. tinyint	15. bit
16. ntext	17. text	18. image
19. timestamp	20. uniqueidentifier	21. nvarchar
22. nchar	23. varchar	24. char
25. varbinary	26. binary (lowest)	

Table 4.1: Precedence of data types in MsSQL DBMS

CONVERT, CAST in MsSQL). This conversions are performed automatically by the DBMS that converts input data into types that allow to execute desired operations. Implicit conversion of different data types is based on the data type precedence that orders data types from highest to lowest precedence. Whenever operator combines operands of different types then these operands with data type of lower precedence are converted to data type of the operand with the highest precedence. Since DBMS determines the desired data type based on the precedence rules, therefore it may happen that data types used will be different then assumed by the database programer and finally will lead to different results than expected (see example 4.3). On the other hand, explicit conversions require specification of convert function and the data type to convert to. Basic mathematical functions that can be used on numeric data types include:

+ is an addition operator that adds two numeric values,

- subtracts two numeric values or calculate negative value if only subtrahend is specified,

* is an arithmetic multiplication operator that returns the data type of the argument with the higher precedence,

/ divides two numerics and returns the data type of the argument with the higher precedence. If both arguments are integers then / returns quotient of the division,

% is a modulo operator that returns reminder of one integer divided by another,

& is a bitwise logical AND operator that can be applied to integer values,

| is a bitwise logical OR operator for integer values,

ˆ is a bitwise exclusive OR (EXOR) operator,

˜ is a bitwise logical NOT operator.

Contemporary database management systems can also compute basic aggregate functions that operate on a set of values and return a single value as a result. All aggregate functions mentioned below have an optional argument that specifies whether duplicate values are discarded or not. Basic aggregate functions include:

- COUNT – returns number of items in a group,
- SUM – returns sum of values in a group,
- MAX/MIN – returns maximal/minimal value from a group,
- AVG – returns average value of elements in a group,
- VAR – returns statistical variance of the elements,
- STDEV – returns statistical standard deviation of the elements,

Example 4.3
Assume we want to use MsSQL DBMS (instead of a calculator) to find the percentage of the salary that Alice spends every month. Alice earns 4 500 USD and spends 3 250 USD per month. Executing the following SQL query

```
SELECT (3250/4500)*100
```

yields 0 since both 3250 and 4500 are integer values for which / operator returns quotient value of the division – that is in fact 0. The same will happen if we leave out brackets. If we reorder terms of the multiplication operand and execute:

```
SELECT 100*3250/4500
```

then the result is 72 since DBMS first computes multiplication and then calculates quotient from the division of two integers – quotient from 325 000 divided by 4 500 is 72. Finally, if we execute

```
SELECT 100*3250.0/4500
```

then the result is 72,222222 since DBMS converts all terms to fractional numeric data type before performing calculations. □

Money data type is a special variant of an exact numeric data type that has fixed precision and scale. Precision of a numeric data type varies between DBMS from 10 to 19 decimal digits while scale is constant and equal 4. Nevertheless, money data type is suitable for storing basic facts such as salaries in employee table it may cause numerous problems if more complex dependencies or larger amount of data are stored. Problems may arise when there are complex arithmetic dependencies between data that may lead to lose of precision, unwanted rounding and inconsistent data.

Example 4.4
Assume table Invoices stores name of the item, NET unit price, no o items sold, NET value, VAT rate and GROSS value with price and values stored in money data type columns. If we want to store information about 1000 laptops sold with 22% VAT rate for total GROSS value of 1 000 000 USD then Invoice table should store the following information:

Invoices					
Item	NET unit price	Quantity	NET value	VAT rate	GROSS value
Laptop ...	819.6721311475	1000	819 672.131147	22%	1 000 000.0000

However, due to restricted scale of money data type the DBMS will store

Invoices					
Item	NET unit price	Quantity	NET value	VAT rate	GROSS value
Laptop ...	819.6721	1000	819 672.1311	22%	1 000 000.0000

As a result we get inconsistent data since

$$819.6721 \cdot 1000 \cdot 1.22 = 999\,999.9620 \neq 1000\,000.00.$$

□

Due to difficulties that arise from using monetary data types it is advised to use exact numeric data types with scale matched to the actual needs.

4.2.2 Differences in implementations of data types in different DBMS

Whereas all database management system implement all predefined data types there are differences in data type names (e.g. character types in MsSQL are called: **char**, **varchar** and **CLOB**) and slight differences in processing the data. MySQL databases management system, for example, silently truncates too long character strings even if non-white spaces are removed. However, this default behaviour can be changed through proper adjustment of system parameters. On the other hand Oracle does not remove a trailing white spaces when inserting character string to a CHARACTER column of shorter length, but rises an exception instead.

There is also a difference in string concatenation operator. Whereas MySQL, PostgreSQL and DB2 use standard-defined || operator MsSQL uses + operator instead.

MsSQL has no BOOLEAN data type but uses additional BIT data type to represent Boolean values as 0, 1 or `NULL`. It also has different implementation of TIMESTAMP data type that cannot be used to store date and time values. Originally, in MsSQL 2000/2005, column of **timestamp** data type is equivalent to 8 byte long BINARY or BINARY VARYING type and is updated automatically every time a row containing this column is inserted or updated. Since there can be only one timestamp column in each table and its value is guaranteed to be unique within the whole database, therefore, it can be used as a mechanism for version-stamping of rows. In new versions of MsSQL server timestamp column is depreciated and a new data type **rowversion** should be used instead.

There are also differences in interpreting `NULL` values. According to the standard two `NULL` values are incomparable but this brings in a question whether single column that is not allowed to store duplicates is allowed to store more then one `NULL` value. In other words, the question is whether in practice *incomparable* should be interpreted as `NULL` = `NULL` or `NULL` ≠ `NULL`. The answer depends on the database management system – most database management systems (e.g. PostgreSQL, MySQL, DB2) allow a number of `NULL`s to be stored in such column (so it follows that `NULL` ≠ `NULL`), MsSQL allows at most one `NULL` value (so `NULL` = `NULL`) while it can be both in Oracle depending on situation.

4.3 Database objects

Traditionally database management systems were developed in order to simplify data storage, data access and separate users from physical representation of the information. Based on relational model and algebra DBMS implemented tables that are still solely responsible for storing data in a way that enables reconstruction of information. Therefore, tables composed of columns are the most obvious object in relational DBMS. However, when DBMS were developed it appeared that tables itself cannot ensure all the properties we would like to have. This caused the requirement for additional objects that are complement to tables. Probably the most obvious are primary and foreign keys that result directly from relational model and are required to reconstruct information from data stored in tables. Views, indices, triggers, stored procedures, functions, cursors and data domain are only an example of other objects that can be found in contemporary relational DBMS.

4.3.1 Tables, views

Table in relational database management system corresponds to relation in relational model. As such table consist of columns, that correspond to attributes of relation, and rows that correspond to tuples. The most important differences between tables and relations in relational model (cf. chapter 3.1) are:

- in general elements stored in columns of tables can only be of simple data types while attributes can store complex types, e.g. relations,
- relations does not allow duplicate tuples while duplicate rows in tables of DBMS are acceptable,
- tuples in relation are unordered whereas table implies particular ordering of rows, nevertheless DBMS does not guarantee any ordering of rows unless an ORDER BY clause is specified.

Beside tables have additional metadata associated with each column that define data type, default value and constrains on that column. The data in the table is physically stored in the database which is the main difference when compared to views.

Tables contain columns that are specified with three basic parameters:

- name,
- data type,
- constraint.

Column constrains can be any of the following:

- `NULL`,
- DEFAULT,
- UNIQUE, or
- CHECK.

`NULL` constraint allows to define whether column accepts `NULL` values or not. Typically, when column is defined it is assumed that it allows `NULL` value to be stored in it. On the other hand, if NOT NULL clause is used then column will not accept any `NULL` values.

DEFAULT constraint defines default value for a column that will be inserted into the column when no value for that column is set in INSERT statement. If column is defined with NOT NULL constraint and has no DEFAULT value defined then any

INSERT statement will fail if it does not set value for that column. Such behaviour is alleviated in MySQL where DBMS inserts data type dependent default value – e.g. empty string for char, varchar and related columns, 0 for numeric columns, or current date/time for datetime columns.

UNIQUE constraint tells the DBMS that values within a single column have to be distinct – no two rows can have the same value in that column. It should be noticed that UNIQUE constraint differs from UNIQUE index. The former one is defined for a single column while the later is defined for an ordered set of columns (possibly only one column). UNIQUE index will be discussed in section 4.3.2.

CHECK constraint allows to enforce simple business rules on data stored in particular column through defining restrictions that data inserted into column has to meet. CHECK constraint may only use simple rules such as checking whether numeric values fall into the desired range, or strings are of the desired length. By default CHECK constraint is only used to verify new data inserted or updated into the column – if check fails then INSERT/UPDATE also fails. Additionally most database management systems enable to check data already stored in a column against the constraint. CHECK constraint is implemented in most modern DBMS, however, this is not the case for MySQL which does not support CHECK clauses at this time.

Most database management systems allow for creating a *temporary tables* that exist in the database as long as user/ application that created these tables is connected to the database. In MsSQL temporary tables are created when table name starts with single or double number character ('#'). Single # means that table is temporary and available only in current connection to the database. When connection is terminated the table is automatically dropped. Double ## means that temporary table is available to all connections to the current database. Such a table is dropped when all connections that have used that table are terminated.

View is a table-like database object that consist of rows returned as a result of SELECT command. Since views display rows from table(s) (or other view(s)) thus rows contained in a view does not have to be stored in the database. Therefore in most cases only view definition is stored in the database. Consequently, from the functional point of view whenever SELECT statement references the view then DBMS substitutes the view name with its definition (i.e. SELECT statement included in view definition).

Since view is defined using SQL SELECT command, therefore, it may contain columns from several tables (may use JOIN operations) and restrict rows that are included in the view (using WHERE clauses). View definition can also use aggregate functions and clauses GROUP BY and HAVING. Since views are defined on a underling table(s) therefore defining the view there is no need to define data types, defaults or constraints for each column. These parameters will be inherited from columns of the underling table(s).

Because vies are similar to tables therefore similar operations can be applied to them. In particular data can be selected from view in exactly the same way as it is possible for tables. Also all relational algebra operators apply to views.

In some cases data can be inserted, updated and deleted from views. However, since views does not store data but contain selected data from underlining table(s), therefore, inserting, updating and deleting data from view actually modifies the underling table(s). Such modifications to the base tables are only possible if view is *updatable* which means that DBMS must be able to unambiguously determine column(s) of a single base table that are about to be modified. Requirements for a view to be updatable may vary between different database management systems, however some requirements are general:

- any modifications (UPDATE, INSERT, DELETE) to the view must reference columns from only one base table,
- columns being modified must directly reference columns/data in the underling table. In particular columns of a view that are derived as:
  - an aggregate function (e.g. AVG, COUNT, MIN, MAX, etc.),
  - a computation from other columns, or
  - a result of a set operators UNION, EXCEPT, INTERSECT,

  are not updatable,
- all columns from the base table being modified that do not have default value defined and do not allow `NULL` value are included in a view. If such columns are not included in a view then INSERT/UPDATE statements cannot set values for them. This in turn causes the SQL INSERT/UPDATE statement executed against the base table to fail since values for required column(s) are not set.
- columns being modified are not included in GROUP BY, HAVING or DISTINCT clauses.

If any of the above conditions is not satisfied then view (or some columns from the view) is not updatable.

Satisfying the above mentioned conditions does not ensure all modifications to the view are possible. For example, to successfully execute an INSERT statement against the view it has to be both updatable and has to set values for all columns of the base table that do not have default value defined and do not allow `NULL` value. Falling to do so will cause the underlining INSERT statement, executed against the base table, to fail since values for required columns are not given in the statement. Since updatable view can be only used to modify columns included in view definition, therefore, if view does not include all such columns of the base table then executing INSERT statements against the view is not possible at all.

Ability to insert and update data in an updatable view may be also restricted by view definition. Most contemporary DBMS support *WITH CHECK OPTION* clause that forces all data modification to be verified against view definition. If view was defined with this option then data contained in INSERT/UPDATE operations is verified whether it meets the criteria included in the WHERE clause of the SELECT statement used to define the view. Data can be modified only if it will remain visible through the view after the modification is committed.

Example 4.5
Database management system contains a table Student that stores information about security social number (ssn), first name (fname), last name (lname) and age (age) and a view defined as follows

```
CREATE VIEW view_T AS
SELECT ssn, fname, lname, age FROM Student WHERE age>26
WITH CHECK OPTION.
```

View view_T is updatable however executing commands

```
INSERT INTO view_T (ssn, fname, lname, age) VALUES (42, 'Ann', 'Knight', 21)
```

and

```
UPDATE view_T SET age=24 WHERE fname='John'
```

will cause DBMS to report an error. It is so since view is defined with WITH CHECK OPTION and value of age in both command does not satisfy the condition age>26 used in this definition. □

Views does not store data but rather define which columns and rows from some table(s) to contain in a view. Therefore, views are susceptible to changes in a structure of the base table(s). By default DBMS verify correctness of the view when defined and accessed through SQL statements against this view. When view is not accessed then base table can be altered and possibly some columns contained in a view can be dropped. DBMS will allow to drop such columns unless *SCHEMABINDING* option was specified when view was defined. When view is defined with SCHEMABINGIND option then DBMS prevents the base tables to be altered in a way that would affect the view (e.g. columns of the base tables, that are included in a view, cannot be dropped). To modify the base table the view must be first modified or dropped.

4.3.2 Indices

Relational model assumes that tuples in relation are unordered and similarly rows in a table have no particular ordering. In most real live applications, however, information are ordered so database management systems have to have mechanisms to speed up data ordering. Similarly, since people have an ability to assimilate restricted set of information at one time, thus it is reasonable to incorporate mechanisms that enable quick searching of data in large datasets. Quick search mechanism is also required for some relational algebra operations. For example JOIN operations of tables T and V require to find rows in V that correspond to the row selected from T. All of the above mentioned requirements can be efficiently satisfied using indices.

Indices are used to logically organise rows of the table in order to speedup access to this data – that is to speedup execution of SELECT statements. Regardless of the implementation, indices can be seen as some kind of lists that tells the DBMS where to find data it is looking for or how to traverse through the rows of the table to get some column sorted in ascending/descending order. Introducing indices into the database requires additional information to be generated and stored along with the table data. Consequently, whenever data in the table is modified the index needs also to be updated and/or rebuild. As a result indices affect execution of INSERT, UPDATE and DELETE statements, which may be time consuming if indices are large.

Indices are implemented in a way that minimises storage requirement and time overhead required to rebuild the index when data is modified. Particular implementation depends also on the type of data that is indexed, so that different index structures are used for indexing string, numeric and spatial data types.

The most basic and the most often supported structure of indices used in DBMS are:

- hash tables,
- B-trees and B^+-trees,
- generalised search trees,
- full-text,
- R-trees and R^+-tress,
- spatial.

R/R^+-trees and spatial indices are dedicated to index spatial information (e.g. location of objects) and are based on *B-trees*-like tree structure. They are becoming popular in contemporary database management systems and implemented as a standard. Generalised search trees are generalisation of B/B$^+$-trees that can be used to build a variety of search tress. Providing a concurrent and recoverable height-balanced search tree as well as ability to use with arbitrary any data type that can have hierarchy defined are the most important virtues of this index structure. Hash tables and B/B$^+$-trees are standard structures for indices used in DBMS. They may be easily used to index string, numeric and date/time data types. Indices are defined to speed up data access through building an additional data structure that is used to look for data. However different SELECT statements may require different indices as well as different indices may improve only some operations on the data. In particular *order of columns* in index definition is also relevant.

Example 4.6
Assume we have a two column hash index Ind1 on a table Student defined as follows:

```
CREATE INDEX Ind1 USING HASH ON Student(fname, lname)
```

When index Ind1 is constructed DBMS takes every row of the table Student, appends binary representation of value stored in column fname with binary representation of lname and calculates the hash value. The resulting hash value will be different than hash value for index Ind2 defined as

```
CREATE INDEX Ind2 USING HASH ON Student(lname, fname).
```

Both indices will also differ from Ind3:

```
CREATE INDEX Ind3 USING HASH ON Student(lname).
```

As a consequence all three hash indices speed up different SELECT statements. In particular any of the above indices can be used to speed up execution of

```
SELECT * FROM Student WHERE fname='John' AND lname='Smith',
```

while only Ind1 is useful for speeding up

```
SELECT * FROM Student WHERE fname='John'
```

and none improves execution of

```
SELECT * FROM Student WHERE fname LIKE 'Jo%'
```

command.

Lets focus on the first SELECT command:

```
SELECT * FROM Student WHERE fname='John' AND lname='Smith'.
```

When Ind1/Ind2 is used then DBMS simply calculates hash value of values given in

WHERE clause and use index to filter out rows of the table Student that have the same hash value. When Ind3 is defined then DBMS uses it to filter out all these rows that contain information about students with last name 'Smith' and later goes through this record set one by one to leave only these records that have fname='John'. In case of the second SELECT command

```
SELECT * FROM Student WHERE fname='John'
```

only index Ind1 is useful. Using Ind2 or Ind3 in this case is pointless since they cannot speed up execution of this query – with high probability hash value of 'John' differ from hash values of lname concatenated with fname and lname itself. Even if, by a small chance, these hash values are equal then the resulting row is not the result set we are looking for, because we are looking for all students with fname='John' and not only those that have their last name 'John'. Executing the last command (that looks for all student with first names that begin with 'Jo') DBMS will not use any of the above defined indices since 'Jo' may be followed with a vast number of different characters and computing all possible hash function is inefficient. Therefore, when this command is executed DBMS simply scans the whole table to find rows that meet the criteria. □

It may follow from the example 4.6 that no indices improve searches over the string data when *wildcharacters* (i.e. %, _) are used, however, this is only true for indices that are based on hash table. If B/B$^+$-tree and fulltext indices are used then some queries may be speed up with an index. For example if B-tree index is used then the B-tree structure is constructed by analysing values stored in a column letter by letter. The resulting B-tree arranges string values based on ASCII codes of successive letters of each string. Therefore, such an index can speed up execution of

```
SELECT * FROM Studenci WHERE fname='Jo%'
```

command by analysing only part of the B-tree to get all the rows that have first names starting with 'Jo'. B-tree index cannot, however, improve execution of

```
SELECT * FROM Studenci WHERE fname='%hn'
```

statement since % corresponds to any string of characters and there is no other way then going through all the rows to find these that satisfy WHERE clause. Apart from improving searches in tables of the database indices may be also used to ensure uniqueness of data within the set of columns. *Unique index* defined on a set of columns ensures that no two rows can have the same values stored in this columns. If unique index is defined on a single table then it is very similar to unique property for that column. However, unique property for a column does not create an index that improves searches over that column while unique index does. Because unique index creates an index therefore order of columns in its definition is important to set of SQL statements whose execution will be improved. While important to efficiency the order of columns is irrelevant for preventing duplicates in a table.

The only one difficulty about using unique indices is with *`NULL` values*. As stated in SQL standard (e.g. [2]) `NULL` values are incomparable, which means that we can neither tell `NULL=NULL` nor `NULL≠NULL`. If so then the obvious question is if unique index (and similarly unique constraint) allows to store multiple `NULL`s in unique column? Answer depends on the database management system and its implementation. For example PostgreSQL, MySQL and DB2 allow multiple `NULL`s to be stored in a column with unique index (unique constraint) defined, MsSQL does not allow multiple `NULL`s, while Oracle allows if remaining columns store different values (whole rows differ) and does not allow otherwise.

Most database management systems distinguish between so called clustered and non-clustered indices. *Clustered* index arranges the physical data storage so that rows of the table are sorted according to the index. In other words clustered index arranges data both logically and physically. Physical ordering of the rows speeds up data access and simplifies read/write operations to the memory/disk. On the other hand when data stored in a table is modified then DBMS has to rearrange data storage witch may be time consuming if large amount of data is modified. Since physical storage can be organised only in one way thus each table may have at most one clustered index defined. If more indices are required then they have to be *non-clustered*, which means that they only introduce logical order of the rows. In comparison to clustered indices non-clustered indices yield smaller overhead when data in the table is modified since no physical data has to be rearranged. On the other hand when non-clustered index is used then data related logically (e.g. all students whose name starts with 'Jo') may be distributed in memory/disk. Therefore, several memory/disk access may be required and execution of SELECT statements with non-clustered index may be slower then with clustered index.

There are no precise rules when to use clustered and non-clustered indices however some recommendations can be given. In particular clustered indices should not be used on columns that frequently and/or significantly change since then costly rearrangement of storage may be required – use non-clustered indices instead. On the other it is advised to use clustered indices when data in a table is accessed sequentially. Such situation occurs during JOIN operations, execution of range queries that use BETWEEN, >, ≥, < or ≤ operators and queries with ORDER BY or GROUP BY clauses.

4.3.3 Stored procedures and functions

Stored procedures and *functions* are another types of objects that are available in most contemporary database management systems and can be run by users and application that access the database. Stored procedures differ from functions in that they do not return values in place of their name and therefore cannot be used directly in an expression. To run a stored procedure DBMS require to run it with an EXEC/CALL SQL statement. There are no other difference so the remaining part of this section applies both to stored procedures and functions unless stated differently.

Stored procedure consists of set of SQL statements that are executed when stored procedure is run. SQL statements that are part of a stored procedure can be almost any valid commands of the SQL language with a few exceptions. In particular stored procedures can modify data stored in the database, alter database structure (in MsSQL stored procedure cannot create stored procedures, views and triggers) or change access privileges as well as define local variables and execute other procedures and functions. Stored procedure is atomic which means that either all SQL commands included in its definition are executed correctly and committed to the database or execution is terminated and all changes already made are rolled back.

Procedures can have arguments that can be either input or both input-output. Some database management systems (e.g. MySQL) also allow arguments to be of output type. Input type arguments correspond to arguments passed by value in software programing languages, while output and input-output types can be regarded as arguments passed by reference. Changes made to both output and input-output type arguments are durable after the procedure is terminated. The difference is that output type arguments cannot be used to pass values into the procedure and when procedure starts value of output type parameters is set to `NULL` (MySQL). Similar as in other programing languages arguments have their name and data type defined and may also have a default values defined.

Stored procedures may be used in order to ensure business and/or integrity rules that cannot be forced by other means, extend access control and improve efficiency of the DBMS. Since procedures can return datasets therefore an example of application is to conceal database structure and give the user access only to the selected or anonymised data that is returned by a stored procedure or function. DBMSs (e.g. MsSQL, MySQL) also have additional mechanisms to control who is allowed to execute stored procedure. There is a separate privilege (EXECUTE) in MsSQL that is required to execute the stored procedure. In MySQL when stored procedures is defined then one can decide whether it will be executed within his security context – so called DEFINER – or INVOKER. If DEFINER security context is used then every statement within this procedure is executed with respect to privileges of the creator of the stored procedure, irrespectively who runs it. With INVOKER security context statements from the procedure will be executed with respect to privileges of the user that runs the procedure.

Stored procedures are stored in the database management server in a pre-compiled form that is suitable for faster execution when compared to executing the same commands send from the application one by one. If there are large number of statements or operation to perform on the database are complex then using a stored procedure we may benefit from lower communication overhead (only name of the procedure and its arguments have to be send to the DBMS) and reusability – the same procedures does not have be implemented separately in different programs using the same database. All of the above mentioned benefits are not for free and using stored procedures involves some shortcomings. In particular stored procedures and functions are computational power and memory demanding thus affecting the overall performance of database management system. Computational power requirements are additionally intensified due to the fact that procedures use SQL language. Since SQL is a declarative language thus in many situations implementation of business rules may be more complex then when the same is implemented in procedural language. For the same reason and due to lack of good developing environments debugging stored procedures is inconvenient.

4.3.4 Triggers

Triggers can be interpreted as a special type of stored procedures that cannot be run by a database user or application manually but are run automatically when pre-defined event is detected. Traditionally such events that cause trigger to execute are related to modification of data in the database table and view. Such triggers are called a DML triggers. Additionally, some database management systems (e.g. MsSQL) also allow to define triggers that execute automatically in response to modification of database objects (so called DDL triggers) and users login to the database (so called logon triggers).

There are three *events* that may cause DML triggers to execute: data insert, update and delete that occur when single INSERT/UPDATE/DELETE command is executed. When triggering event is detected then DBMS checks whether BEFORE trigger is defined. If so, then execution of triggering event is postponed and SQL statements from the BEFORE trigger are executed. When BEFORE trigger is finished then postponed event is executed again. After the executing event finishes DBMS checks whether AFTER trigger is defined for that event. If so, then SQL statements included in AFTER trigger are executed. This is the most typical implementation of DML triggers however there are some difference between different database management systems. For example, MsSQL does not have an AFTER trigger but uses an INSTEAD OF trigger. If INSTEAD OF trigger is defined then triggering SQL command is not executed at all and is replaced with commands included in this trigger. Similar solution is used in DB2 but IBM allows to fire the trigger BEFORE, AFTER or INSTEAD OF triggering command (so INSTEAD OF is an additional option to BEFORE and AFTER).

It follows that DML triggers may be run BEFORE or AFTER/INSTEAD OF data modification, thus they are suitable for verification of data stored in the database and ensuring integrity of information. To do so statements included in the trigger have to be able to access data stored both in the table/view triggering command refers to and in command itself.

Accessing data from the table/view is relatively easy since trigger can contain SQL commands that simply select or modify this data, however, some restrictions may apply. In BEFORE INSERT/UPDATE trigger data selected from the table/view does not contain new values inserted/modified by triggering command. In AFTER UPDATE-/DELETE trigger the original data stored in the table prior to execution of triggering command is no longer available – since command was already executed and data has been modified/deleted. To access data included in a triggering SQL command triggers use two variables NEW and OLD. Structure of these variables is the same as a structure of a table row for which trigger was defined, but they contain data that is about to be inserted into the table (NEW) or removed from the table (OLD). Similarly there are some restrictions on using these variables. NEW is a read-write variable that can be only accessed in triggers that are executed in response to INSERT and UPDATE operation – NEW contains all-`NULL` values in DELETE trigger since DELETE command does not insert any data into the table. In contrast, OLD is a read-only variable that contains all-`NULL` values in INSERT trigger.

Using NEW/OLD variables is the most common solution found in most database management systems. Different solution is used in MsSQL where a table-like variables INSERTED and DELETED are used. The difference between NEW/OLD and INSERTED/DELETED is that NEW/OLD store data included in a single SQL command while INSERTED/DELETED can store multiple rows if more then one row is affected with triggering command. The difference becomes obvious if we consider the following example

EXAMPLE 4.7
Assume we have a table Employee that stores information about all employees of the company (including salary) and has an UPDATE trigger defined. How many times will be the trigger fired when following SQL command is executed

```
UPDATE Employee SET Salary=Salary+100 ?
```

In most database management systems (e.g. MySQL, Oracle) triggers are fired once for each triggering event that is affected row of the table. Therefore, the above SQL command will fire UPDATE trigger once for each row of the table with NEW/OLD variables adjusted to each execution of the trigger. In MsSQL, however, trigger will be fired only once and INSERTED/DELETED variables will contain all the data that are about to be inserted to/deleted from the table. □

Enhanced variant of triggers is implemented in DB2 management systems since triggers can be either fired for a single row of a table being modified (as in MySQL) or for the whole statement regardless of how many rows of the table are affected (as in MsSQL). If trigger is fired for each modified row of the table then NEW/OLD variables represent single row of data being modified. When trigger is fired only once for a statement then NEW/OLD are table-like variables that store all the data affected by the triggering command.

Since triggers contain SQL commands thus they may modify data in database tables. Consequently, triggers can fire other triggers or the same trigger again. It may happen that execution of *nested triggers* may cause infinite loop that would have cause the database management system to fail. To prevent such situation DBMS define maximum level to which triggers can be nested. Every trigger fired increases current nested level while termination of the trigger decreases it. If current nested level exceed maximum nested level allowed then execution of trigger is terminated.

Chapter 5

XML extensions to RDBMS

eXtensible Markup Language (XML) has been widely adopted as a method to represent data. Its main advantage is platform-independence and ease of exchanging information among loosely coupled, disparate systems, for example in business-to-business and workflow applications.

XML format is increasingly present in modern applications thus it was also necessary to enable databases to store XML documents rather then transforming them to (or generating from) the relational form. Storing native XML data in a relational database provides benefits in the areas of data management and query processing. Additionally, most DBMS already support XML documents and provide users with built-in functions and algorithms that enable to manipulate these documents easily.

The following section focuses on Microsoft SQL database server and mechanisms that are provided for managing XML documents in that server.

5.1 XML columns

MsSQL server has a dedicated *`xml` data type* that is used to store XML documents. This data type is specialised and stores the XML document in a parsed tree structure in BLOB field. This *storage* method preserves XML structure and includes information about containment hierarchy, document order as well as elements and attributes values. Specifically, the InfoSet content of the XML data is preserved with respect to insignificant white spaces, order of attributes, namespace prefixes, and XML declaration.

Prior to dedicated `xml` data type XML documents were stored in a CLOB fields or where mapped to relational tables (called shredding). Quite obviously, both methods are impractical – CLOB make it difficult to parse, search and crop parts of XML documents, while object-relational and relational-object mapping is time consuming, impractical for recursive structures and ordered data (relational representation is unordered while in XML documents order of elements is crucial). Storing XML documents in a dedicated data type allows easier access to data, enables to create indices for elements of the XML document, and to use dedicated commands and functions.

According to Microsoft the way XML data should be stored depends on the type of XML document as well as operations that will be applied to it. If XML is highly structured with known schema, the relational model is likely to work best for data storage. On the other hand, if the structure is semi-structured or unstructured, or unknown, then using `xml` data type is a good choice. It is also a good solution if storage has to be platform-independent in order to ensure portability. Additionally, it is an appropriate option if some of the following properties are satisfied:

- data is sparse or the structure of the data is unknown, or may change significantly in the future,
- data represents containment hierarchy, instead of references among entities, and may be recursive,
- order of data is inherent,
- data will be queried or updated partially based on its structure.

If none of these conditions are met then you should use the relational data model. For example, if data is in XML format but application just uses the database to store and retrieve it, then a `varchar` column is all that is require.

Storing the data in an `xml` column has additional benefits that include verification whether the XML data is well formed or valid, and support for fine-grained query and updates into the XML data. In general the most important reasons to use native `xml` data type includes:

- requirement to share, query and modify XML data in an efficient and transacted way on database level with fine-grained data access,
- guarantee that the data is well formed and also optionally validate the data against the XML schemas,
- requirement for efficient and scalable searching over the XML data that can be achieved with dedicated XML indices,
- requirement for SOAP, ADO.NET, and OLE DB access to XML data.

The `xml` column implements the ISO standard XML data type, thus it can store well-formed XML version 1.0 documents as well as fragments of XML content with an arbitrary number of top-level elements.

Typed and untyped XML

Each column of `xml` data type can be optionally associated with a collection of XML schemas which in that case is called *typed*. Association of `xml` column with XML schema provides validation constraints and data type information. Validation constrains are similar to table and cell-level constraints that can be defined in relational tables and are checked whenever data in the XML document is modified. Database server rollbacks changes if these constraints are not met. Schemas can also provide information about data types of attributes and elements in the `xml` column. This enables more precise operational semantics when compared to untyped `xml`. For example DBMS can verify if arguments of arithmetic operations are decimals or strings. Because of this, storage of typed `xml` columns can be more compact then untyped.

There are several situations that suggest whether typed or untyped `xml` column should be used. In particular untyped column should be used when:

- schema for the XML document is unknown,
- schemas are known, but for some reasons users don't want the database server to validate the data. For example, validation is done on application level and/or database has to store parts of the XML documents that may not be valid.

On the other hand typed columns should be used when schemas are known and:

- database server is required to validate your XML documents according to the schema,
- users want to query XML documents with high efficiency.

Typed XML columns can store XML documents or content and additionally have to provide the collection of XML schemas. The difference between XML documents and content is that documents can have exactly one top-level element whereas content allows the column to store multiple top-level element in a single XML instance.

Listing 5.1 presents an example of schema definition in MsSQL database management server. Listing 5.2 presents a definition of very simple table with typed `xml` column.

Listing 5.1: Example of XML Schema definition

```
CREATE XML SCHEMA COLLECTION XMLStudent
AS N'<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema elementFormDefault="qualified"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="Student">
<xsd:complexType><xsd:complexContent><xsd:restriction
    base="xsd:anyType">
<xsd:sequence>
  <xsd:element name="FName" type="xsd:string" />
  <xsd:element name="LName" type="xsd:string" />
  <xsd:element name="StudentID" type="xsd:integer" />
  <xsd:element name="Address" maxOccurs="unbounded">
    <xsd:complexType><xsd:complexContent><xsd:restriction
        base="xsd:anyType">
    <xsd:sequence>
      <xsd:element name="Type" type="xsd:string" />
      <xsd:element name="Address1" type="xsd:string" />
      <xsd:element name="Address2" type="xsd:string" />
    </xsd:sequence>
    </xsd:restriction></xsd:complexContent></xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:restriction></xsd:complexContent></xsd:complexType>
</xsd:element>
</xsd:schema>'
```

5.2 XPath, XQuery and extensions to SQL

Database management systems that support `xml` data type also support an extended set of SQL command as well as an XPath and XQuery language. XPath [13] and XQuery [14] are standardized by W3C consortium and are dedicated to manipulate XML documents and their content. Additionally, most DBMSs support extension to the XML language that allows to modify structure of the XML documents – so called XML DML language.

Listing 5.2: Example of CREATE TABLE command with typed `xml` column

```
CREATE TABLE Students(
    PK_STRUDENT int identity(1,1),
    FName varchar(50) DEFAULT '',
    LName varchar(50) DEFAULT '',
    Data xml ([XMLStudent])
)
```

5.2.1 Extensions to SQL language

Extensions to standard SQL language enable to generate an XML document from the relational data. In MsSQL server this transformation can be done in the following ways:

- casting sting types to `xml` data type,
- using SELECT command with optional FOR XML clause,
- using constant assignments,
- using bulk loads.

The first method allows to convert a string containing an XML document into an `xml` data type object, that can be stored into the `xml` column. The string to be converted has to include a valid XML structure, in particular all XML tags have to closed. The result of the function is an untyped XML document. Casting from string to `xml` is a default operation when data is inserted or modified in `xml` column. Listing 5.3 presents an example of such situation.

Optional clause FOR XML in SELECT statement tells the MsSQL server to convert the result set of SELECT command into an XML document. The general structure of the command is as follows:

SELECT *list_of_columns* FROM *table_name* ... FOR XML *mode*,

where *mode* can have one of the four values:

- AUTO,
- PATH,

Listing 5.3: Example of INSERT SQL command with XML data

```
INSERT INTO Students (FName, LName, Data) VALUES
  ('John', 'Smith',
   '<Student><FName>John</Fname><LName>Smith</LName>
    <Address><Type>Home</Type><Address1>Wroclaw</Address1><Address2/>
    </Address>
    </Student>' )
```

Listing 5.4: Result of a SQL command with FOR XML AUTO clause

```
<Students FName="John" LName="Smith" />
<Students FName="Alice" LName="Knight" />
```

- RAW,
- EXPLICIT.

In AUTO mode DBMS generates an XML document in which each row of the result set is a single XML tag <table_name> with attributes that take names from names of the returned columns and values from each row of the result set. For example SQL command:

SELECT FName, LName FROM Students FOR XML AUTO,

returns an XML content presented on listing 5.4.
Similar command with PATH mode will return each row as a single XML element <row> with children elements that correspond to each column of the result set (listing 5.5). Remaining modes enable for further structure adjustment of the resulting XML documents – details can be found in [11].

5.2.2 XPath

XPath [13] is a language that is used to navigate through elements and attributes of XML document. It is also a basis of XQuery language. XPath was designed specifically to reflect the hierarchical structure of the XML documents, to simplify navigation

Listing 5.5: Result of a SQL command with FOR XML PATH clause

```
<row>
  <FName>John </FName>
  <LName>Smith /LName>
</row>
<row>
  <FName>Alice </FName>
  <LName>Knight /LName>
</row>
```

between nodes of the document and compute values from its content. To navigate through XML document XPath uses paths and axis that point to specific node or set of nodes. Paths can be either absolute or relative. Absolute paths start from the root of the XML document and begin with a single slash / sign. On the other hand relative path begins with a double slash // sign and select all nodes in the XML document. For example, the difference between the following two XPath expressions

/Students/Address/Address1
//Address1

is that the first expression selects Address1 element that is located in Address and Student element. On the other hand, the second expression selects all Address1 elements despite their exact location in the XML document. It is worth to remember that XPath expressions return the node of the XML document together with all the subnodes.

In order to select sequence of XML elements relative to the current (also called *context*) node the XPath defines axes. *Axes* (fig. 5.1) allow to traverse the XML structure in forward (towards the end of the XML document) and reverse (towards the beginning of the document) directions:

- forward axes:
  - child – contains the direct children of the context node,
  - descendant – is a transitive closure of the child axis. It contains all the descendants of the context node, i.e. child nodes of the context node, child nodes of all childes and so on.
  - following-sibling – contains the context node's following siblings, i.e. those children of the context node's parent that occur after the context node in the

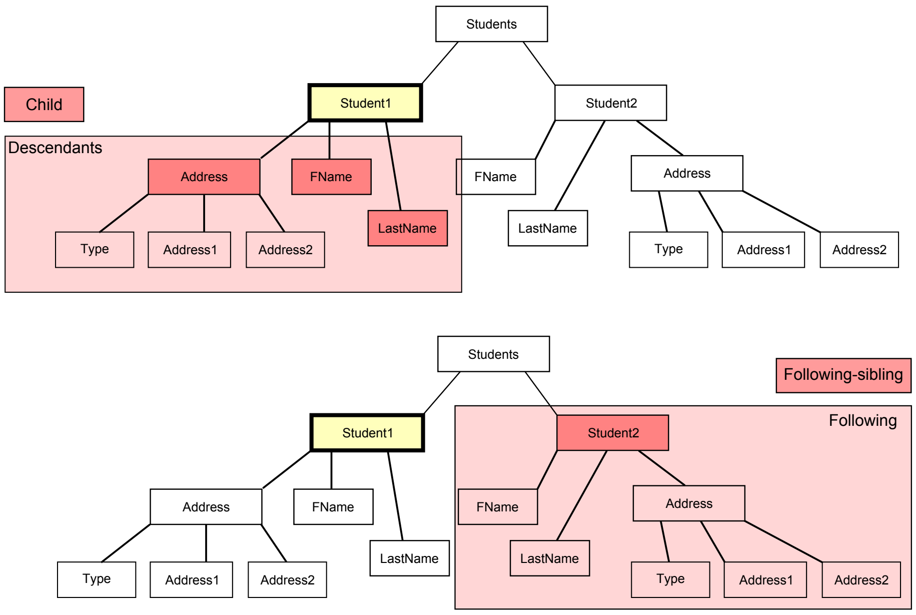


Figure 5.1: Some axes of the XML document

document order,

- following – contains all nodes that are descendants of the context's node parent, are not descendants of the context node itself, and occur after the context node in the document order,
- descendant-or-self – contains the context node and its descendants,
- self – contains just the context node itself,
- attribute – contains the attributes of the context node,

• reverse axes:

- parent – contains the parent of the context node, or an empty sequence if the context node has no parent,
- ancestor – is a transitive closure of the parent axis. It contains all the ancestors of the context node, i.e. the parent, the parent of the parent, and so on.

  - preceding-sibling – contains the context node's preceding siblings, i.e. those children of the context node's parent that occur before the context node in the document order,
  - preceding – contains all nodes that are descendants of the root of the tree in which the context node is found, are not ancestors of the context node, and occur before the context node in the document,
  - ancestor-or-self – contains the context node and its ancestors. It follows that root node is always included in the ancestor-or-self axis of any node.

XPath also defines node tests which are conditions that must be true for each node selected by a step of the XPath expression. The condition may be based on the kind of the node (element, attribute, text, document, comment, or processing instruction), the name of the node, or (in the case of element, attribute, and document nodes), the type annotation of the node. There are several tests that might be used in path expressions, here only the most commonly used are listed:

- node() – matches any node,
- text() – matches any text node (i.e. node that contains text),
- element() – matches any element node.

Path expressions can also contain predicates. *Predicates* are always enclosed in square brackets and are used to filter sequence of XML elements. Those for which the predicate value is true are retained, and those for which the predicate value is false are discarded. Predicate can be used to select particular XML element from the sequence of elements or elements with particular value of the attribute. For example:

```
/Students/Address[2]
/Students/Address[Type="Home"]
```

returns second address of each student and address that has Type attribute equal to "Home" respectively. Predicates can be verified against any element of the path expression, thus:

```
/Students[FName="John"][LName="Smith"]/Address[Type="Home"]/Address2
```

will return the second part of John Smith's home address.

It should be not surprising that predicates may consist of *comparison operands* that can be divided into following groups:

- value comparisons,

- general comparisons,
- node comparisons.

Value comparisons – `eq` (equal), `ne` (not equal), `lt` (lower then), `le` (lower equal), `gt` (greater then) and `ge` (greater equal) – are used for comparing single values. When compared operands are evaluated and converted to their least common type using type promotion which is analogous to types precedence in RDBMS (cf. tab. 4.1 in chapter 4.2.1). General comparison operators include =, ! =, <, <=, > and >=. These operators are existentially quantified comparisons that may be applied to operand sequences of any length. The result of a general comparison that does not raise an error is always true or false. When evaluating a general comparison in which either operand is a sequence of items, an implementation may return true as soon as it finds an item in the first operand and an item in the second operand that have the required magnitude relationship. Node comparisons are used to compare two nodes, by their identity or by their document order. An example of such operand are:

- is – returns true only if the left and right operands evaluate to exactly the same single node,
- << – returns true only if the node identified by the left side occurs before the node identified by the right side in document order,
- >> – is the opposite to the previous one.

Apart from the above basic capabilities XPath also implements `for` expressions (that can iterate over sequence of nodes), conditional expression (`if ... then ... else`), quantified expressions (`some|every ... in ...`), cast function, and a number of predefined functions (e.g. first(), last(), count(), position(), etc.).

5.2.3 XQuery

XQuery [14] is a query language dedicated to operate on XML data and is the same for XML what SQL language to relational databases. Similar to SQL XQuery is a declarative language. XQuery relies on XPath (cf. chapter 5.2.2) and XML schema data types. XPath expressions are used to navigate through the XML documents in order to access sequence of nodes. Moreover, any expression that is syntactically valid and executes successfully in both XPath and XQuery returns the same result in both languages. Since these languages are so closely related, their grammars and language descriptions are generated from a common source to ensure consistency. Sequence of nodes returned by path expression is an input to XQuery command that outputs a sequence of XML nodes or atomic values.

The basic building block of XQuery is the expression, that is a string of characters, and may be constructed from keywords, symbols, and operands. XQuery allows expressions to be nested with full generality. XQuery provides a feature called a FLWOR expression (pronounced ”flower”) that supports iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name FLWOR comes from following keywords:

- for,
- let,
- where,
- order by,
- return.

The for and let clauses generate an ordered sequence of tuples each containing XML node, or sequence of XML nodes. The optional where clause serves to filter the tuple stream, retaining some of them and discarding others. The optional order by clause can be used to reorder the stream. The return clause constructs the result of the FLWOR expression. The return clause is evaluated once for every tuple in the tuple stream, after filtering by the where clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the results of these evaluations, concatenated as if by the comma operator.

5.2.4 XPath and XQuery in Microsoft SQL Server

This section presents some basic operations on XML documents that can be used in MsSQL server. All of the remaining examples will be based on one XML Schema presented on listing 5.6. The schema defines the XML structure with `Student` element as root. `Student` contains obligatory integer attribute `StudentID` and an optional attribute element `Nationality`. It also consist of sequence of elements `FName`, `LName` and `Address`. Since these elements are defined as a sequence, therefore, in XML documents they will have to appear in the same order. Additionally, first two elements have to appear exactly once while `Address` element can appear 0 or more times within a single `Student` element. `Address` is of complex type and contains two string elements –

Listing 5.6: Definition of XML Schema

```
<?xml version="1.0" encoding="UTF-16"?>
<xsd:schema elementFormDefault="qualified"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="Student">
<xsd:complexType>
  <xsd:sequence>
     <xsd:element name="FName" type="xsd:string" />
     <xsd:element name="LName" type="xsd:string" />
     <xsd:element name="Address" minOccurs="0" maxOccurs="unbounded" >
           <xsd:complexType>
     <xsd:sequence>
        <xsd:element name="Address1" type="xsd:string" />
        <xsd:element name="Address2" type="xsd:string" />
     </xsd:sequence>
     <xsd:attribute name="Type" type="xsd:string" />
                <xsd:complexType>
          </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="StudentID" type="xsd:integer" />
  <xsd:attribute name="Nationality" type="xsd:string" use="optional"/>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

`Address1` and `Address2`. It also contains an optional attribute `Type`. Listing 5.7 presents an example of valid XML document with respect to the schema.

In order to insert data into an `xml` column (called `XMLData`) a standard INSERT command can be used with XML data given as a string of characters. Listing 5.8 presents an example of such an INSERT command. It is worth to notice that the XML data included in the SQL command has two root elements. Such data is a valid XML – i.e. satisfies XML schema defined, however, it is not well-formed. Presented SQL command will raise an error if the `xml` column was defined with DOCUMENT clause. However, it was defined with CONTENT clause then its execution is successful.
Data stored in an XML column can be accessed and modified using the following methods:

- selecting the whole `xml` column in a standard SQL command,
- using dedicated functions to access elements of the XML column.

Listing 5.7: Example of valid XML document

```
<Student StudentID="192000" Nationality="PL">
  <FName>Sandra </FName>
  <LName>Knight </LName>
  <Address Type="Office">
    <Address1 >1600 Pennsylvania Ave</Address1>
    <Address2>Washington , D.C.</Address2>
  </Address>
  <Address Type="Home">
    <Address1 >1 Neverland Rd</Address1>
    <Address2>Los Angeles </Address2>
  </Address>
</Student>
```

MsSQL offers the following functions that allow to manipulate the XML content:

- query(),
- value(),
- exist(),
- nodes(),
- modify().

Each function takes an XQuery command as input and optional additional parameters and outputs a result of execution of this command, possibly casted to the standard SQL data type.

Function *query()* specifies an XQuery against an instance of the `xml` data type and returns an untyped XML data. For example:

```
SELECT Data.query('/Student/Address[@Type="Home"]') AS XMLData
FROM Students
```

returns all rows from the Students table (fig. 5.2). Each row contains one column that stores an XML data composed of all `Address` elements from the XML data stored in that row, which have the attribute `Type` equal to "Home". The function returns all `Address` elements that satisfy this condition together with all of its descendants. If there

Listing 5.8: Example of INSERT command with XML data

```
INSERT INTO Students (Data) VALUES
  (N'<Student StudentID="192000">
          <FName>Sandra</FName><LName>Knight</LName>
          <Address Type="Office">
              <Address1>1600 Pennsylvania Ave</Address1>
              <Address2>Washington, D.C.</Address2>
          </Address>
          <Address Type="Home">
              <Address1>1 Neverland Rd</Address1>
              <Address2>Los Angeles</Address2>
          </Address>
     </Student>
     <Student StudentID="171100">
     <FName>Brice</FName>
     <LName>Creek</LName>
   </Student>')
```

are a few `Address` elements of that type in a single row of the table then the resulting row includes all these elements. If `xml` column doesn't contain any XML element that satisfy XQuery condition then the corresponding row in the output is empty.

Figure 5.2 shows empty rows in the result set. These rows correspond to table rows that does not have an `Address` element with attribute `Type` equal "Home". Consequently the XQuery command:

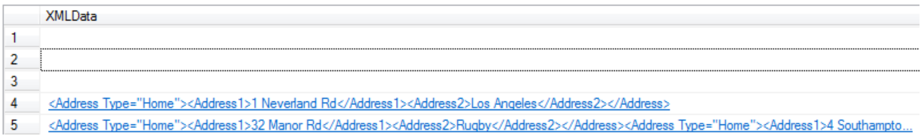
/Student/Address[@Type="Home"]

returns no XML elements. In order to eliminate such rows from the SQL result set method *exist()* and WHERE clause can be used. Method *exist()* returns 1 for the XQuery expression that returns a nonempty result and 0 otherwise. Previous SQL command can therefor be modified to:

```
SELECT Data.query('/Student/Address[@Type="Home"]') AS XMLData
FROM Students
WHERE Data.exist('/Student/Address[@Type="Home"]')=1,
```

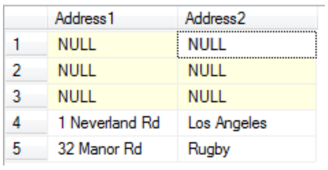
which returns only last two rows of the previous result set (cf. fig. 5.2).

Method *value()* performs an XQuery against content of the XML column and returns a value of SQL type. This method takes two input arguments – first is an



	XMLData
1	
2	
3	
4	<Address Type="Home"><Address1>1 Neverland Rd</Address1><Address2>Los Angeles</Address2></Address>
5	<Address Type="Home"><Address1>32 Manor Rd</Address1><Address2>Rugby</Address2></Address><Address Type="Home"><Address1>4 Southampto...

Figure 5.2: Result of simple SQL command with query() method. Note that the last row contains two root elements, while first rows are empty since no XML elements matched the predicate used.



	Address1	Address2
1	NULL	NULL
2	NULL	NULL
3	NULL	NULL
4	1 Neverland Rd	Los Angeles
5	32 Manor Rd	Rugby

Figure 5.3: Result of simple SQL command with value() method.

XQuery expression, second is a string that represents name of a valid SQL data type, e.g. 'varchar(30)'. In contrast to *query()* this method requires an XQuery expression that returns a singleton (one element set) or empty set. It returns a scalar value and allows to extract a value from an XML instance stored in an `xml` type column. In this way, it is possible to specify SELECT queries that combine or compare XML data with data in non-XML columns. For example:

```
SELECT Data.value('(/Student/Address[@Type="Home"]/Address1)[1]', 'varchar(50)')
AS Address1,
Data.value('(/Student/Address[@Type="Home"]/Address2)[1]', 'varchar(50)')
AS Address2
FROM Students
WHERE Data.exist('/Student/Address[@Type="Home"]')=1,
```

will return two elements of address in separate columns. Note that there is an index used in both *query* function (i.e. [1]) that ensures the XQuery will return at most one element from the XML data. Figure 5.3 presents result of execution of the above SQL command. Observe, that the last row contains only one set of address information despite the fact that there where two addresses of type "Home" stored in the `xml` column (cf. fig. 5.2). It is so, since we used index in XQuery expression. We could also use index [2] in order to get information about second home address.

An obvious question arise how can we know how many particular XML elements (e.g. addresses of type ”Home”) there are in an XML data structure stored in a column? This information can be returned using a `count()` function that is defined in an XQuery language. For example:

```
SELECT Data.value('count(/Student/Address[@Type="Home"])[1]', 'int') AS Cnt
FROM Students,
```

returns number of home addresses stored in `xml` column in each row of the table.

Method `value()` can be also used to search for elements in XML column that satisfy a particular condition when compared with SQL column. For example, assume that table students contains relational column ’FirstAddress’. In order to find the `Address2` for which `Address1`=FirstAddress the following command can be used:

```
SELECT Data.query('(/Student/Address[@Type="Home"]/Address2)[1]') AS Address2
FROM Students
WHERE Data.value('(/Student/Address[@Type="Home"]/Address1)[1]', 'varchar(50)')
=FirstAddress.
```

Nevertheless the above command executes correctly it is recommended, due to efficiency reasons, to use `exist()` method with `sql:column()` method:

```
SELECT Data.query('(/Student/Address[@Type="Home"]/Address2)[1]') AS Address2
FROM Students
WHERE Data.exist('/Student/Address[@Type="Home"]/Address1
= sql:column("FirstAddress")')
```

Method `modify()` modifies the contents of an XML column. This method takes an XML DML [12] statement to insert, update, or delete nodes from the XML data. The `modify()` method of the `xml` data type can only be used in the SET clause of an UPDATE statement. There are three keywords that can be used in XML DML language:

- insert,
- delete,
- replace value of.

Keyword `replace value of` allows to change value of a particular element in the XML document. For example:

```
UPDATE Students SET Data.modify('replace value of (/Student/FName)[1] with "Adam"')
WHERE Data.exist('/Student[FName="John"]')=1,
```

changes first name of a student John into Adam.

Keyword *`delete`* deletes an element from the XML document. The element to be deleted is identified with XQuery expression that may point either single element or set of elements to be deleted. Also all the nodes or values that are contained within the selected nodes, are deleted. If the expression returns an empty sequence, no deletion occurs and no errors are returned. For example:

```
UPDATE Students SET Data.modify('delete /Student/Address')
WHERE Data.exist('/Student[@StudentID=171100]')=1,
```

deletes all the addresses from and XML documents from all rows of the table in which XML document contains information about student with `StudentID`=171100. Note that if an XML document contains several students and at least one of them has `StudentID`=171100 then addresses of all students in that document will be deleted.

Keyword *`insert`* allows to insert one or more elements identified by an XML content or XQuery expressions as child nodes or siblings of the node identified by second XQuery expression. Similar as in *`replace value of`* the second XQuery expression has to identify a singleton. The expression that determines things to be inserted can result in a node, and also a text node, or in an ordered sequence of nodes. If the expression results in a value or a sequence of values, the values are inserted as a single text node with a space separating each value in the sequence. *`Insert`* keyword allows to specify where new nodes should be entered, this parameter can have the following values:

- into – nodes will be inserted as a direct descendants (child nodes) of the node identified by the second expression. If the node determined by second expression has any child nodes then it is required to use additional *`as first`* or *`as last`* argument to specify where the new node should be added.
- after – nodes will be inserted as siblings directly after the node identified by second expression,
- before – nodes will be inserted as siblings directly before the node identified by second expression.

For example:

```
UPDATE Students SET Data.modify( 'insert
<Address><Address1> Janiszewskiego 11/17 </Address1><Address2 /></Address>
as last into (/Student[@StudentID=171100])[1]')
WHERE Data.exist('/Student[@StudentID=171100]')=1,
```

inserts an address for student with `StudentID`=171100. Note, that `Address2` element is empty but it cannot be omitted since then the inserted XML content would not meet the requirements imposed by XML schema (cf. listing 5.6) and error message would be returned.

Chapter 6

Security of databases

Security in database management systems refers to procedures and algorithms that enable to protect stored data from being accessed by unauthorised person. To ensure authorised access to data DBMS have to implement three basic mechanisms:

- user identification and authentication,
- user authorisation, and
- data encryption.

Identification is a procedure that tells user identity to the system. Identification can be simply achieved with unambiguous login assigned to each user and entered when logging in. It enables user to introduce him/herself to the system, however, it does not ensure that the user is who he claims to be. Verification of user identity is achieved through authentication procedure. In such procedure system ask users to give some credentials that are known both to the system and the identified user.

Authorisation refers to procedure that verifies whether authenticated (or at least identified) user is allowed to take requested actions so it verifies user privileges.

EXAMPLE 6.1
Money withdraw at automatic teller machines (ATMs) is subject to both identification, authentication and authorisation. Bank card with magnetic card or microprocessor identifies owner of the bank account, personal identification number (PIN) authenticates the card holder to the bank while authorisation verifies whether money withdraw should be allowed or not (e.g. due to withdraw limits or lack of money). □

6.1 Access control

Access control is about managing and verifying privileges of users that request DBMS to perform some actions. While access control seems to be a very straightforward concept there are numerous challenges when it comes to implementation.

Theory of access control strategies differentiates between users, subject and objects. Group of users consist of people and computer software that access the system and can be identified as an individual entity. Individual user can login into the system a number of times as well as he may run a number of applications. Each instance of user may have different privileges thus becoming different subjects. Objects (or securables) are any elements in the system that may be accessed by subjects, while privileges determine operations and actions that may (or may not) be taken. Since objects in DBMS form a hierarchical structure, therefore, in many cases privileges assigned to superior object are inherited by subordinate objects. For example SELECT privilege granted to table implicitly grants SELECT privileges to all columns of that table.

Existing database management systems implement either discretionary access control, mandatory access control or both.

6.1.1 Privileges in database management systems

Contemporary database management systems share similar *access privileges* that may be granted to subjects. All privileges can be divided into three sets:

- data manipulation privileges,
- data control and procedure/function related privileges,
- server-level privileges.

Data manipulation privileges are the best known and the most often used as they directly determine which data can or cannot be accessed by subjects. Select, update, insert and delete rights determine whether it is possible to execute SELECT, UPDATE, INSERT and DELETE SQL statements respectively. In most DBMS these privileges enable to grant or deny access to whole tables and views. Some DBMS (e.g. MsSQL) allow to control data manipulation privileges in greater detail enabling to grant access only to selected columns of the table/view. Column level access control applies only to SELECT and UPDATE commands. INSERT and DELETE commands affect whole row of the table thus they require that user has corresponding privileges to the whole table (and thus to all the columns).

Data control privileges determine what kind of DDL statements can be executed by user. This group of privileges may differ in number and functionality significantly

Table 6.1: Major privileges in MsSQL 2008 DBMS and kinds of objects to which they apply

Privilege	Applies to
SELECT	Tables, views, table's/view's columns
UPDATE	Tables, views, and their columns
INSERT	Tables, views
DELETE	Tables, views
CONTROL	Stored procedures, functions, tables, views
REFERENCES	Tables, views, and their columns
EXECUTE	Stored procedures, scalar and aggregate functions
ALTER	Stored procedures, functions, tables, views
VIEW DEFINITION	Stored procedures, functions, tables, views

between different database management servers. In MsSQL 2008 [3] server there are five main permissions:

- CONTROL – refers to ownership-like capabilities. The grantee of the CONTROL permission has all defined permissions on the object as well as on all subordinated objects. For example, CONTROL on a database object implies all privileges on the database, on all schemas in this database, and on all objects within all schemas within the database.
- ALTER – determines whether subject is allowed to change properties of object, except for ownership. Similar to CONTROL privilege, if ALTER is granted to the object then user is allowed to alter, create and drop any object within that object. For example, ALTER privilege on a database includes the ability to create, alter and drop objects from the database.
- CREATE – confers the ability to create objects within a database.
- VIEW DEFINITION – enables the user to access metadata (e.g. definition) of the object.
- REFERENCES – this privilege is required to define foreign keys and views with SCHEMA BINDING clause.

Table 6.1 presents major privileges in MsSQL 2008 DBMS and kinds of objects to which they apply. Most DBMS use privileges that allow user to perform some operation – so called *positive privileges*. In such a situation user that haven't been granted a particular privilege (e.g. SELECT) is not allowed to use that privilege (e.g. display rows of the table). When granted the privilege the corresponding SQL operations can be executed and the privilege can be revoked to prevent user from future execution of these operations. Granting and revoking privileges is controlled by GRANT and REVOKE commands respectively that can be executed by entitled user (this depends on the access control model used, see successive sections).

Some DBMS (e.g. MsSQL) enable do define both positive and *negative privileges* that allow to prevent user to execute some operations. Similarly to positive privileges, negative privileges can be also revoked. In such situation a grantee that was revoked a negative privilege is no longer denied to use it. However, it does not tell whether or not grantee is allowed to use this privilege, since to do so it maybe required to have this privilege granted. If DBMS supports negative privileges then users in such a system may be granted, denied or have no privileges to an object. When either first or the second situation is the case the user is allowed or denied to use the privilege on that object. In the third case effective privilege depends on other criteria such as privileges owned by the group user is assigned to, or privileges user inherits from superior objects. Negative permissions make the access control more flexible, however, also require methods to cope with situations when a user is both granted and denied privilege at the same time.

6.1.2 Discretionary access control

In *discretionary access control (DAC)* system does not distinguish between users and subjects simply matching single user to a single subject. As a consequence when logged user runs an application it inherits privileges of that user. Therefore, users and subjects are synonyms when talking about the DAC model (which is not the case in mandatory access control). This simplifies privilege management while having serious consequences to the security level attained and thus application of database management systems.

Another simplification arise from the fact that managing access privileges is discretionary with respect to object owner. It means, that owner of an object (e.g. database, table, stored procedure, etc.) decides what are the privileges other subjects have to this object. Owner of an object can grant privileges to that object to any user, as well as can grant some users with the ability to grant privilege on their own. This is achieved through granting the privilege with GRANT OPTION clause. Above mechanisms simplify management of access control rights but also require all users of the DBMS to take care of privileges owned by other to objects they created. In large DBMS this may cause a lot of difficulties. In particular it has to be determined what are the privileges owned by users to newly created objects if no permissions where granted by the object creator. Such situation is called *incompleteness* and can be solved in three ways:

- users are denied any privilege on the new created objects,
- users are granted all privileges, or
- users inherit privileges from the object superior to the new created one. For example, if new table is created then users of the database inherit all privilege they have on the database to the table.

Management of privilege in large, multi-user DBMS may be impractical if each user is granted separate privileges. To simplify management contemporary database systems allow to create groups (also called roles) of users that share the same privileges to some objects. Similar to database objects, user groups can have a hierarchical structure thus groups can consist of both users and other groups. This simplifies privilege management but at the same time requires to decide what are the effective privileges of the particular user if he is a member of some group (or groups) with different set of privileges assigned to the user and groups. This issue becomes even more important when DBMS allows for *negative privileges* and user can be granted a privilege that is denied on the group user belongs to – so called *inconsistency*. If such situation occurs then database system has to decide whether grant or deny privileges are superior. Such conflict can be solved in four ways:

- more specific privilege is superior,
- less specific privilege is superior,
- grant privilege is superior,
- deny privilege is superior.

If first or second rule is used then effective privilege results from the database object hierarchy. For example, if more specific privilege are superior and user is granted a SELECT privilege on a table and denied SELECT privilege on the database then he is allowed to select data from the table. On the other hand, if less specific privileges are superior, then executing the select command will rise an access denied error. Remaining rules set priority to either grant or deny privileges respectively. If such solution is used then user is granted (denied) a particular privilege if this privilege is granted (denied) to that user or any group user belongs to, irrespectively of object hierarchy.

Example 6.2
Table 6.2 represent privileges of four users to the same table and its columns, where '+' and '-' denote positive and negative privilege respectively. For which user executing the SQL command:

```
SELECT * FROM Table1
```

will succeed/fail? □

Table 6.2: User privileges on table Table1 and its columns (example 6.2)

Privileges	Objects		
Users	Table1	Table1.fname	Table1.lname
Alice	+SELECT		
Bob	+SELECT	-SELECT	+SELECT
John	-SELECT		+SELECT
Marry	-SELECT	+SELECT	+SELECT

Table 6.3: Result of execution of SELECT command on table Table1 by users with different privileges (example 6.2)

Method of solving inconsistency	Alice	Bob	John	Marry
more specific superior	OK	FAIL	FAIL	OK
less specific superior	OK	OK	FAIL	FAIL
positive superior	OK	OK	FAIL	OK
negative superior	OK	FAIL	FAIL	FAIL

In MsSQL database management server, which supports both positive and negative privileges, it is assumed that deny privilege is superior to grant, except for the column-level SELECT privilege. If user is denied SELECT on the table, but granted SELECT on some columns of that table, then user is allowed to select data from that columns only.

Revoking privileges in DAC model is up to the owner of the object as well as users that were granted privileges with GRANT OPTION clause. If no privileges were granted with GRANT OPTION clause then revoking is a straightforward assuming that owner of an object remembers who was granted the privilege. Situation changes when privileges were granted with GRANT OPTION which enables grantees to hand privileges over to other users. In such case revoking privilege from the grantee should also revoke this privileges from users that were handed this privilege over.

Example 6.3
Alice is an owner of table T1. She grants all privilege to T1 with GRANT OPTION to Bob, and SELECT privilege with GRANT OPTION to Eve. Later on, Mathew is granted all privileges to T1 by Bob, and a SELECT privilege from Eve. Consequently,

Mathew was handed over all privileges on T1 (see fig.6.1 upper). Now assume that Alice does no longer trust Bob and revokes his privileges on table T1. What should be the privileges of Mathew if Bob has no privileges on T1 any more?

Observe, that if Mathew is left with all privileges on T1 then there is no way these privileges can be revoked by Bob since to grant/revoke privileges user has to hold that privilege itself (fig. 6.1 lower left). Therefore, it is reasonably to revoke Mathew's privileges simultaneously with revocation of Bob's privileges. Precisely, Mathew should lose all the privileges he was granted by Bob that were not granted to him by other users. It follows that after revoking Bob's privileges on T1 Mathew should be left only with SELECT privilege on T1 since this was granted by Eve (fig. 6.1 lower right). □

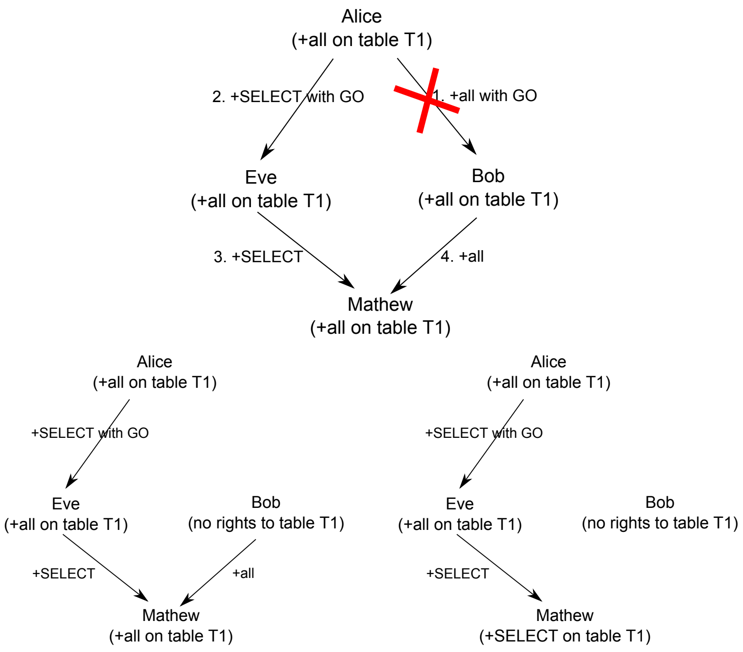


Figure 6.1: Example of privileges handed over by different users (upper) and two possible results of their revocation – worse (lower left), better (lower right) (example 6.3)

To overcome problem of revoking privileges that were previously granted with GRANT OPTION clause and thus possibly handed over to other users, some database systems (e.g. MsSQL) implement a revocation command with CASCADE clause. In such situation owner of a database object can easily revoke privileges from all users. In DBMS where cascaded revocation is not available (e.g. MySQL) users has to be very careful when granting privileges with GRANT OPTION. It is so since managing the privileges that were handed over by grantees, who have GRANT OPTION for that privilege, is troublesome and requires access to the database that sores information about all databases and users defined in the DBMS – in MySQL and MsSQL servers this database is called INFORMATION SCHEMA.

Discretionary access control in connection to separate privileges for tables and views has one potential threat to the security of access control.

Example 6.4
Bob and Alice are two users of the same database and both hold the privilege to create new objects in this database. Assume Bob has created a table Bob_secrets and revoked all privileges on that table from Alice. Consequently, Alice cannot select any data from that table, but since she can create new objects she may try to create view that will reference all the data from Bob_secrets table. As an owner of that view she should have all the privilege to it and should be able to access Bob's data. The above scenario is sound but intuitively such situation cannot happen since it would compromise security of the database system. The question is which of the above mentioned steps cannot be executed?

The scenario assumes that Alice executes the following SQL commands:

1. CREATE VIEW Alice_view AS SELECT * FROM Bob_secrets ,
2. SELECT * FROM Alice_view.

Result of the above commands will go differently in different database management systems but in none Alice will be able to execute successfully the last command and display Bob's secrets. For example, in MySQL DBMS Alice will not be allowed to execute the first command since to create a view that references table (or other view) the user has to have SELECT privilege on that table (or view). In MsSQL, on the other hand, the first command succeeds, however Alice is not granted the SELECT privilege on the view she created. Moreover, as long as she doesn't have the CONTROL privilege on the database, she is not allowed to modify privileges for the view she created (she cannot grant privileges for herself). If she holds CONTROL privilege on the database, then she has all permissions to all objects in the database. Moreover, she can access the view as well as Bob's secret table, so there is no need for Alice to create the view any more. □

Another issue in DAC model is an indirect access to data stored in the table. Indirect access to the table means that user that is not granted the SELECT privilege on that table can use other commands to find out what data is stored in the table. Presumably this will not give the ability to access all the data and will require a lot of time to find out what is stored in the table but anyway it will compromise the access control security.

Assume Bob, who is a spy, has gained all the privileges to table Secret_agents(SSN, fname, lname, address) owned by Alice, except for the SELECT privilege. At first it seems that he cannot read the data stored in the table. However, he is allowed to perform INSERT, UPDATE and DELETE operations and in particular he may try to execute the following SQL statement:

```
UPDATE Secret_agents SET fname=fname WHERE lname='Smith'.
```

Will the command be executed correctly by the DBMS? If so, then Bob will infer that Smith is in fact Alice's secret agent, despite the fact that he cannot read the contents of the table directly. To prevent such indirect access to data DBMSs prevent user from executing SQL statements with WHERE clause if no SELECT privilege has been granted to that user. In other words Bob is not allowed to execute previous statem, but still can run:

```
UPDATE Secret_agents SET fname=fname.
```

This will not reveal who is a secret agent but usual after executing such a query DBMS responds with information how many rows of a table/view were affected. If so then Bob will know how many agents work for Alice. In extreme case if Bob has eliminated someone he suspected to be Alice's agent, he may check whether the number of agents in the table falls. If not then wrong person was eliminated.

If UPDATE privilege is revoked from Bob but he knows, that SSN is a unique column (possibly primary key since SSN functionally determines all remaining columns), then he may check whether a particular SSN is stored in a table. If he knows the SSN of a person he suspects as an agent then he executes

```
INSERT INTO Alice_agents (SSN) VALUES ('123456789').
```

If this command raises an error saying that the SSN value violates the unique index (primary key) then he knows that the person is in fact an agent. Otherwise, Bob just needs to run a DELETE statement to clear the table in order not to disclose his activity.

Last but not least possibility is to use DELETE command. This is particularly helpful if Bob has no information on unique indexes and/or keys for the table. This prevents him from using INSERT statements but he may still draw on DELETE privilege in a very similar way – Bob simply tries to delete the row related to person with SSN he knows:

```
DELETE FROM Alice_agents WHERE SSN='123456789'.
```

If the suspected person is in fact an agent then row of a table will be deleted and DBMS will respond with number of rows affected greater then 0. Ok, but how to prevent Alice from detecting that someone was messing up with her secret table? One possibility is to execute a DELETE statement within a transaction that is afterwards rolled back, so no changes are written to the database (to read more on transactions see chapter 6.3.2).

The main difficulty in DAC model is that it has too many dependencies that affect the correctness of the security policy. This includes large number of different privileges, privilege inheritance and their priorities as well as the fact that privileges are (or can be) controlled by every user which makes it difficult to assure that every object has privileges as specified.

6.1.3 Mandatory access control

The most important difference between discretionary (DAC) and *mandatory access control (MAC)* arise from the fact, that security policy (i.e. access privileges) are centrally controlled by a security administrator rather then each user separately. Moreover, users do not have the ability to overwrite the policy, and for example, grant access to their files to other users. Secondly in MAC model differentiates between users and subjects, which means that different applications run by the same user may have different access rights. Third difference comes from the fact, that MAC doesn't address the privileges directly but rather determine whether subject is allowed to access objects. If this is the case then either subject is allowed to read and write data, or additional DAC is used to determine what operations can be performed. According to Trusted Computer System Evaluation Criteria (TCSEC) [15] a MAC model is "a mean of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorisation (e.g., clearance) of subjects to access information of such sensitivity".

One of the best known implementation of the MAC model was proposed by David Elliott Bell and Leonard J. LaPadula [6] (see also [5]). The BLP model, verifies user permissions based on security class related to the subject (c_S) and object (c_O). Each security class consist is a pair

$$c = \langle L, C \rangle , \tag{6.1}$$

where S denotes security level and C is a set of security categories.

Security level is a single element from the set of security levels. The set of security levels is fully ordered witch means that for any two elements from the set a and b either $a > b$ or $a < b$. Traditionally this set consist of four elements:

$$\mathrm{U} < \mathrm{C} < \mathrm{S} < \mathrm{TS}, \tag{6.2}$$

where U, C, S and TS denote unclassified, confidential, secret and top secret security levels respectively. Set of security categories is a subset of all possible security categories which constitute an unordered set of elements. Categories are simply names that determine areas of interests or application – traditionally categories can be defined as 'Army', 'Air force', 'Marines', 'Special forces', 'Infantry', 'Intelligence', etc.

In order to decide whether subject with security class c_S can access object with class c_O it is required to determine the relation between these two classes. Comparison of security classes can only have one out of three results:

- $c_S \geq c_O$,
- $c_S \leq c_O$, or
- $c_S \neq c_O$ otherwise.

Since set of all categories cannot be ordered therefore deciding whether $c_S \geq c_O$ or $c_S \leq c_O$ is decided as follows:

$$c_S \geq c_O \quad \text{iff} \quad L_S \geq L_O \text{ and } C_O \subseteq C_S, \tag{6.3}$$

$$c_S \leq c_O \quad \text{iff} \quad L_S \leq L_O \text{ and } C_S \subseteq C_O. \tag{6.4}$$

It is important to note, that whenever neither $c_S \geq c_O$ nor $c_S \leq c_O$ holds then security classes of subject and object are incomparable ($c_S \neq c_O$) and access is not granted.

Example 6.5
For the security classes:

- $c_1 = \langle TS, \{\text{Army, Air force}\}\rangle$,
- $c_2 = \langle TS, \{\text{Army, Infantry}\}\rangle$,
- $c_3 = \langle S, \{\text{Air force}\}\rangle$,
- $c_4 = \langle S, \{\text{Army, Infantry}\}\rangle$,

following relations hold:

- $c_1 \geq c_3$,
- $c_2 \geq c_4$, and
- $c_1 \neq c_2, c_1 \neq c_2, c_2 \neq c_3, c_3 \neq c_4$.

□

BLP model introduces two rules for accessing objects. *Simple security property* states that subject is allowed read access to an object if and only if

$$c_S \geq c_O. \tag{6.5}$$

It follows that subject can read objects that are on the same or lower security level and have security categories that are subset of categories assigned to the subject. This access rule is also called *no read up* rule. The second rule is so called **-property* (it should be read as "star property") and states that subject can modify an object if and only if

$$c_S \leq c_O. \tag{6.6}$$

*-property means that subjects can only write information to objects that are on the same or higher security level and for which set of categories contains set of categories assigned to the subject. Some publications also distinguish between write and append access. If such distinction takes place then write access refers to the process of writing the objects that can be also read. This is the case when subject and object are in the same security class which means that

$$c_S \geq c_O \text{ and } c_S \leq c_O. \tag{6.7}$$

On the other hand append refers to accessing the objects that have higher security class compared to the security class of a subject. Nevertheless, this difference is only a naming convention since in both cases *-property has to be met.

Simple security and *-properties ensure no data will be revealed to unauthorised subject since it is not possible to read objects with higher security class. Also subject with high security class cannot leak the secret information since it cannot be written on security level lower than security level of this subject. Unfortunately in many practical situations writing objects in lower security level is required. Lets consider the following example.

Example 6.6
A general and a lieutenant are in security class $c_G = \langle TS, \{Army, Airforce\}\rangle$ and $c_L = \langle S, \{Airforce\}\rangle$ respectively. Since $c_S \geq c_L$ thus lieutenant cannot read object (e.g. files) created by general but can create objects that will be available to general. On the other hand, general can read everything that was created by lieutenant but cannot write an object that will be available to him. As a consequence general is not able to create an object storing orders for the lieutenant which is clearly unacceptable in real life. □

To overcome problem mentioned in the above example the BLP model differentiates between maximal security class ($c_S^{\max}$) and current security class (c_S) of a subject. Maximal security class is the highest class subject can have, while current security class is the class subject is currently using. The difference between both classes is in security level – in current class security level is smaller or equal to the security level of the maximal class

$$c_S = \langle L, C \rangle, \tag{6.8}$$

where $L \leq L_S^{\max}$ and $c_S^{\max} = \langle L^{\max}, C \rangle$. Getting back to the example it follows that whenever general wants to issue an order to the lieutenant he is required to login to the system on the security level S, create the order, and then login back to security level TS in order to have access to top secret objects.

Unfortunately BLP model does not solve all the security issues. In particular it is still possible to leak the information indirectly, using subliminal channels. In BLP model Bob with high security class can mount a subliminal channel to Alice that has the lowest security class. This is possible since both Bob and Alice can have write access to objects in Bob's security class. Since in computer systems no two subjects can simultaneously have write access to an object (e.g. file) thus Bob may lock the file for writing depending on the bit of information he wants to convey to Alice (fig. 6.2). If they agree that they will attempt to access the file repeatedly, lets say every second, then depending on whether Alice successfully gains the file access or not she knows the bit of information "sent" by Bob. This mechanism can be then used to leak any information from the high security class that is accessible to Bob.

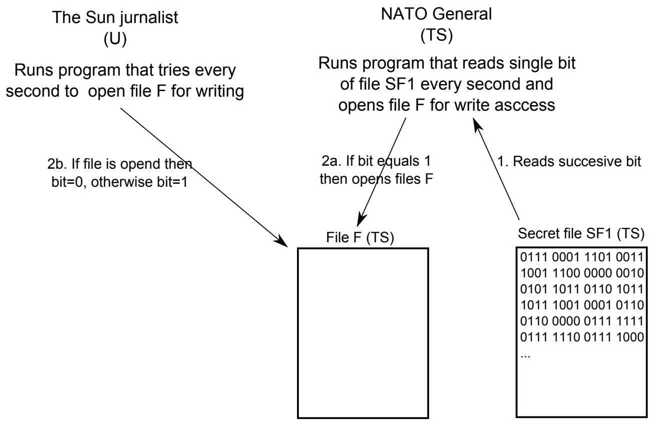

The Sun jurnalist
(U)
Runs program that tries every second to open file F for writing
NATO General
(TS)
Runs program that reads single bit of file SF1 every second and opens file F for write asccess
2b. If file is opend then bit=0, otherwise bit=1
2a. If bit equals 1 then opens files F
1. Reads succesive bit
File F (TS)
Secret file SF1 (TS)
0111 0001 1101 0011
1001 1100 0000 0010
0101 1011 0110 1011
1011 1001 0001 0110
0110 0000 0111 1111
0111 1110 0111 1000
...

Figure 6.2: Example of the subliminal channel in BLP model

Centralised privilege management as well as restrictive access control policy enable to ensure high level of security. Unfortunately mandatory access control procedures are difficult to implement, inconvenient for most users and still prone to covert channels. Since MAC is less user friendly then DAC thus most computer systems and database management systems implement only discretionary access control. Mandatory control is used in military and administrative applications so it is implemented in dedicated computer systems and databases. Also some widely available operating systems (e.g. Microsoft Windows Vista, 2008 and 7) and DBMS (e.g. Oracle, DB2) have built-in plugins that enable to use mandatory access control.

6.2 Data privacy

Ensuring access control enables DBMS to control whether or not users are allowed to access particular database objects and/or perform some operations (e.g. create or modify tables). Unfortunately, even if access control is properly configured it doesn't ensure 100% security as adversary may try to access data stored on the computer hard drive, in the memory or being transmitted over the Internet. Attacker may also gain temporary access to the database (e.g. as a consequence of breaking into the computer system) or database backups and copy all the information. To prevent secret information from being disclosed to third parties DBMS allow to encrypt data stored in the databases and tables. Data encrypted with good encryption algorithms (e.g. AES) is secure, even if cryptograms are disclosed, since it is infeasible to decrypt it if the encryption key is unknown.

There are there levels for encrypting the data stored in the DBMS

- file system encryption,
- database-level encryption,
- cell-level encryption.

6.2.1 Encryption algorithms and key hierarchy

Encryption algorithms are procedures that transform text that can be understood by everyone (so called plaintext) into unreadable form (ciphertext). The transformation is reversible so ciphertext can be decrypted to get a readable plaintext. Transformation from plaintext to ciphertext and vice verse goes according to encryption and decryption procedures. These procedures take plaintext (or ciphertext) and an encryption key as an input and output a ciphertext (plaintext). Using wrong key to decrypt a ciphertext gives an output that have nothing to do with the original plaintext that was encrypted.

Encryption algorithms can be divided into symmetric and asymmetric algorithms. *Symmetric algorithms* use a single key to encrypt plaintext and decrypt the resulting ciphertext. If such an algorithm is used to secure communication over the Internet all parties that communicate have to share common encryption key prior to transmission. Requirement to exchange the encryption key prior to transmission and in safe manner is the biggest drawback of these group of algorithms. The advantage of symmetric encryption is the use of simple operations to transform plaintext into ciphertext (usually substitutions, permutations, addition or multiplications) and relatively short encryption keys – between 128 and 256 bits. Consequently, symmetric encryption algorithms can encrypt data with high throughput which makes them suitable for processing large amount of data. Nowadays there are several symmetric encryption algorithms that were verified and proved to be secure enough to be used in order to ensure privacy. The most popular algorithms are: Advanced Encryption Standard (AES), International Data Encryption Algorithm (IDEA), RC6, Sprent, Twofish and Blowfish.

Asymmetric algorithms use two different keys to encrypt and decrypt the data. Both keys, however, are not random but are related to each other according to a mathematical formula and some additional secret. Interrelationship ensures also that knowing one key, but without knowledge on the secret, it is infeasible to calculate the other one. This allows to public one key and keep the other one secret. Public key can be then used for encrypting plaintext that can be later decrypted only with the secret key from the same pair. Asymmetric algorithms does not require to exchange the encryption key in a secure way, but this is achieved at cost of higher computational complexity and larger encryption keys that vary from 512 up to 2048 bits depending on the algorithm. Consequently, asymmetric algorithms run much slower then symmetric and are not suitable for encrypting large amounts of data. In contrast to previous group asymmetric algorithms are suitable for creating a digital signatures for messages. In fact, this is the most often used application of these algorithms. Nowadays most DBMS implement two asymmetric algorithms – RSA, that can be used both for data encryption and signing, and Digital Signature Standard (DSS) that is dedicated for signing purposes.

Security of all encryption algorithms is up to the security of *encryption keys*. In particular encryption keys have to be stored in a safe way ensuring that they cannot be accessed by unauthorised users. Within the database management systems keys are stored in an encrypted form and can be decrypted either by the DBMS or by the user. To decrypt the key DBMS/user has to know security password or another key that was used during encryption. It follows that some keys protect others which leads to the hierarchy of keys.

One the most developed *key hierarchy* was created in MsSQL database management server [3]. MsSQL uses a 6 level hierarchical encryption and key management infrastructure in which keys from lower levels are protected by keys from higher level

Table 6.4: Encryption algorithms in different database management systems

	Database management systems		
Algorithm	MsSQL 2008	MySQL 5.5	Oracle
DES	+	+	
3DES	+		+
AES	128/192/256	128	128/192/256
RC2	+		
RC4	+		
RSA	512/1024/2048		

or optional password. Precisely, keys from each layer are used to encrypt the keys from the layer below by using a combination of certificates, asymmetric keys, and symmetric keys. Additionally, asymmetric and symmetric keys can be stored outside of MsSQL Server in so called Extensible Key Management (EKM) module. The highest level of key hierarchy (fig. 6.3) stores an DBMS service master key (SMK) that is protected by the operating system and available only to the operating system. This key is generated when instance of DBMS is installed and cannot be deleted. This key is used to protect database master keys (DMKs). Database master key is used to protect single database. It is used for encrypting the whole database when transparent data encryption (TDE) is turned on (see chapter 6.2.3). When TDE is enabled then to access the database it is required to get the Service Master Key, decrypt Database Master Key and use this key to decrypt the database. This work when database is run on a single server, however, database can be backed up and moved to another server. Since TDE yields backups to be encrypted as well and the new DBMS has different service master key with high probability, therefore, there has to be some other way to access the DMK. This challenge was solved with an optional password that may be used to protect the DMK. If password is defined then DMK is available either to the original DBMS or to anyone that knows the password. When encrypted backup is restored on a new server then user is asked to enter the password, original database master key is decoded and encrypted with SMK of the server on which backup is restored. Database master key may be changed at any time, but when TDE is enabled than any modification of the DMK yields the whole database to be decypted and reencrypted with modified key. Service master key and database master key are all symmetric keys.

Database master key is also used for securing certificates[1] and asymmetric key within a database which are used in turn to protect symmetric keys. Certificates, asym-

[1]Certificate is an asymmetric public key digitally signed by trusted authority that binds the key to the identity of some person, device or service that holds a corresponding private (secret) key.

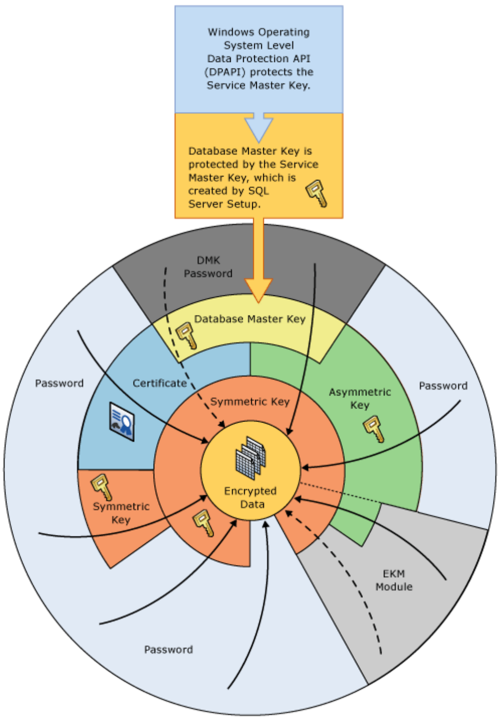

Windows Operating System Level Data Protection API (DPAPI) protects the Service Master Key.
Database Master Key is protected by the Service Master Key, which is created by SQL Server Setup.
DMK Password
Database Master Key
Password
Certificate
Symmetric Key
Asymmetric Key
Password
Encrypted Data
Symmetric Key
EKM Module
Password

Figure 6.3: Layers of encryption in MsSQL database management server (figure from http://msdn.microsoft.com/en-us/library/ms189586.aspx)

metric and symmetric keys are used to encrypt the data stored in the database tables and they may be used for cell-level encryption (see chapter 6.2.4). Additionally, certificates, asymmetric and symmetric keys may be protected by passwords. Passwords may be also used to encrypt the data.

6.2.2 File level encryption

Encrypting the *file system* is an option of almost all contemporary operating systems. It enables to store files on hard drives in an encrypted form, that can be read only by the operating system that knows the correct encryption key. Such key can be either entered manually by user when starting the computer, or be stored on a memory stick and loaded whenever needed. The most important advantage of file system encryption is its transparency to the applications running in the computer system. Encryption and decryption is performed automatically whenever application is reading from or writhing to the disk respectively. Transparency ensures applications do not have to be modified which means that file system encryption can be turned on and off whenever needed. The drawback is that data is decrypted when read from the disk so everyone with access to the operating system typically has access to all the data encrypted on that system. If either an attacker access memory, memory is paged by the operation system or system is hibernated then unencrypted data is stored to disk and can be accessed easily. In addition, operating system backups still require another solution because a system administrator can see the data unencrypted during backup routines. Also turning encryption on and off requires the hard drive to be encrypted/decrypted with may take a lot of time. Another disadvantage of disk encryption is that it typically does not support separation of duties between the system administrator and database administrator.

If we consider a DBMS running on an encrypted file system then it is ensured that stealing the hard disk from the computer will not allow the adversary to read the data. However, if adversary manages to brake into the operating system, then he may easily copy the database file into other file system (which is not encrypted) and then access the data stored inside. This method does not prevent data from being read out if someone gets access to the DBMS since data manipulated by the DBMS is already decrypted. Nevertheless this drawbacks, this kind of encryption is sometimes the only one solution since some DBMS does not support encryption or available methods of encryption are inadequate for particular application (e.g. are too time consuming).

6.2.3 Database-level encryption

Some database management systems allow for encrypting a particular databases stored in the DBMS. If available, this type of encryption is similar to file system encryption as it is transparent to the applications using the database. Unlike the previous type, *database-level* encryption encrypts only selected database (not the whole file system) and is implemented by the DBMS. Database level encryption offers the advantage of being transparent to client applications so, similar like with file system encryption, does not require the application to be modified when encryption is turned on. For the same reason database level encryption is also called a transparent data encryption (*TDE*).

Database level encryption operates at I/O level thus any data written to the database files (on the disk) is encrypted. Encryption also refers to backups and database snapshots, so they are encrypted on the disk as well. However, similar to file system encryption, data that is in use by the DBMS is not encrypted since transparent encryption does not provide protection at memory level and while transmitted to and from the server. A disadvantage of this method is that it incurs CPU and memory overhead during every database access. As reported by Microsoft and Oracle the average performance loss is about 5% but in the worse case it can be as high as 28%.

6.2.4 Cell-level encryption

Cell-level encryption enables to encrypt every single cell of the database table. Cell-level encryption can either use a symmetric or asymmetric encryption algorithms, however, for better performance symmetric algorithms should be used. Since encrypted data is a sequence of bytes therefore, encrypted data is not a string of characters even if the plaintext consisted only of alphanumeric chars. Consequently, in order to store encrypted data in the column of a database table it is required to use binary data type for that column (e.g. varbinary). Additionally most DBMS does not differentiate between columns of varbinary data type that store binary data and encrypted data. This has significant consequences since enabling the encryption on a varchar column requires:

- to change column data type to varbinary,
- modify (most usually drop) all the indices that were declared for that column,
- modify the user application that is accessing that column.

First two modifications refer to DBMS system itself and are direct consequence of encryption. Both modifications may have enormous consequences to the database performance. First of all, not all database management system enable to index the binary data type columns. Even if indexing of binary columns is possible then indexing encrypted data has nothing to do with ordering the plaintexts. It follows that searching encrypted data is slower compared to searching corresponding plaintexts first due to lack of indices, second due to requirement to decrypt all the ciphertexts.

Cell-level encryption also requires the application to be modified since suitable encryption/decryption clauses have to be included in SQL commands. For example, in MsSQL server there are following encryption/decryption clauses:

- symmetric encryption:
  - CREATE SYMMETRIC KEY – is an SQL command that create symmetric encryption key and assigns an encryption algorithm to it,
  - EncryptByKey – is a function that performs encryption with a symmetric key and encryption algorithm declared for that key,

  - DecryptByKey – is a function that allows to decrypt information encrypted with EncryptByKey,
  - EncryptByPassPhrase – is a function that encrypts a plaintext using a password as an encryption key,
  - DecryptionByPassPhrase - decrypts whatever was encrypted with EncryptByPassPhrase with the password as a key.

- asymmetric encryption:
  - CREATE ASYMMETRIC KEY – is an SQL command that create asymmetric encryption key and assigns an encryption algorithm to it,
  - EncryptByAsmKey (EncryptByCert) – are functions that encrypt messages with asymmetric key (certificate),
  - DecryptByAsmKey (DecryptByCert) – are functions that decrypt ciphertexts encrypted with EncryptByAsmKey (EncryptByCert),

- digital signing:
  - SignByAsymKey (SignByCert) – signs a massage using the asymmetric key (certificate),
  - VerifySignedByAsmKey (VerifySignedByCert) – verifies signature generated with SignByAsymKey (SignByCert) functions.

If we want to store first and last names encrypted with AES algorithm then following SQL command has to be executed:

```
INSERT INTO Employees (fname, lname) VALUES
        (EncryptByKey(Key_GUID('MyFirstKey'), 'John'),
        EncryptByKey(Key_GUID('MyFirstKey'), 'Smith')) .
```

Prior to executing the above statement it is required to create a symmetric encryption key which can be done in following way:

```
CREATE SYMMETRIC KEY MyFirstKey WITH ALGORITHM = AES_256
        ENCRYPTION BY PASSWORD 'SecurePassword' .
```

In MsSQL command that creates the encryption key also declares the encryption algorithm that will be used. Later on, when data is being encrypted, there is no need to specify the encryption algorithm to be used.

Some database management systems (e.g. Oracle) simplify cell-level data encryption by making it more transparent to the user and applications. Oracle enables to set a parameter ENCRYPT when creating a table and defining a column in that table. Column can be of any type, and is accessed via standard SQL commands with no additional functions. Database management system automatically detects whether or not particular column is encrypted and modifies SQL commands if necessary.

It is important to remember that cell-level encryption lowers performance of DBMS more significantly then transparent data encryption (database level encryption). Therefore, cell-level encryption should be used only when it is really needed. In particular, it should be considered if it is really necessary to encrypt columns that:

- are used in range queries – i.e. SQL commands with WHERE clause that contains BETWEEN operator or looks for rows that fall into the range of values,
- change very often,

You should also avoid to encrypt primary key columns or columns with indices since indexing encrypted columns does not improve searching. Encryption also degrades overall performance that can be as high as 30% lower, compared to database without a cell-level encryption.

6.3 Data integrity

Database management systems are interfaces between the users and the data. The most important advantage of this interface is separation of physical data access procedures and their logical interpretation. This enable user/programer to focus on manipulating the data rather then taking care on how this data is stored and whether it is correct and coherent. On the other hand DBMS has to maintain procedures and algorithms for ensuring that data is correct. Integrity constrains and transactions are mechanisms that are used for this purpose.

6.3.1 Integrity

Integrity constraints ensure static correctness of the data stored in the database. These constraints ensure data corresponds to (represents) real live objects and data entered into the database is accurate, valid and consistent. For example, integrity constraints can ensure that speed of movement cannot be greater then speed of light, age of human being cannot be negative or each e-mail address has to contain @ sign and at least one dot. Data integrity also address uniqueness and ensures quality of data. For example, if an employee is entered with an `id`=123, the database should not allow another employee to have the same ID value. If you have an `gender` column then rows can only take 'F' and 'M' values for that columns while other values should be rejected. Moreover, when there is another table that stores names of the departments and each employee is assigned to exactly one department, then database server has to ensure that each employee is assigned to the department that exist in the department table. Determining the correct values and integrity constraints as well as deciding how to enforce them are two important steps in planning table and database structure.

Data integrity falls into four categories:

- entity integrity,
- domain integrity,
- referential integrity,
- user-defined integrity.

Entity integrity

Entity integrity states that row of a table is unique, not `NULL`, can be uniquely identified and data stored in different rows will not mix. Entity integrity enforces the integrity of the identifier column (or columns). This can be achieved through primary key of a table, indexes, UNIQUE constraints or, IDENTITY properties.

Domain integrity

Domain integrity is the validity of entries for a given column. It can be enforced by restricting the data type of the column, the format (through CHECK constraints and rules), or the range of possible values (through FOREIGN KEYs, CHECK constraints, DEFAULT and NOT NULL definitions, and rules). Domain integrity can be as simple as a data type with list of values allowed. A good example is aforementioned `gender` column that can be declared as `char(1)` column with acceptable values of 'F' and 'M'.

Referential integrity

Referential integrity preserves the defined relationships between tables when records are entered or deleted. In most database servers, referential integrity is based on relationships between foreign keys and primary keys or between foreign keys and unique keys in two related tables. Referential integrity ensures that key values are consistent across tables. Such consistency requires that there be no references to nonexistent values and that if a key value changes, all references to it change consistently throughout the database. An example of referential integrity is relationship between employees and departments they work. Referential integrity in that case force the ID of the department, stored in the employee table, to be a valid ID that exist in department table as well.

Enforcing referential integrity is usually achieved through foreign keys that prevent user from

- inserting rows to a related table if there is no associated row in the primary table (e.g. employee cannot be assigned to the department that does not exist),
- updating the primary table that results in orphaned rows in a related table (e.g. changing the ID of the department without updating IDs in corresponding rows of the related table),
- deleting rows from primary table if there are matching rows in the related table (e.g. dropping the department that has employees assigned).

To ensure referential integrity on updates and deletes in primary table foreign keys allow to decide what to do if row from the primary table is updated or deleted. This is done when foreign key is defined through declaration an ON DELETE and ON UPDATE actions. There are four basic actions that can be taken:

- CASCADE – row in referencing table is updated/delete whenever row in primary table is updated/deleted,
- NO ACTION – means that no action will be executed and commands run against the primary table will be rolled back (no data will be updated or dropped),
- SET NULL – referencing columns in the referencing table are set to `NULL` whenever primary column from primary table is updated or the corresponding row is deleted,
- SET DEFAULT – this action is similar to SET NULL however it inserts the DEFAULT value to the referencing columns instead of `NULL`.

User-defined integrity

User-defined integrity allows to define specific business rules that do not fall into one of the other integrity categories. All of the integrity categories support user-defined integrity (all column- and table-level constraints in CREATE TABLE, stored procedures, and triggers).

6.3.2 Transactions

A database *transaction* is a unit of work executed on database management system that is treated in a coherent and reliable way. In particular multiple transactions, e.g. run by many users, are independent of each other and can be executed concurrently. Unit of work can be made up of one or more commands, for example, a number of SELECT, INSERT, UPDATE, DELETE operations or operations that change the physical structure of the database. In the very simple case the transaction consist of a single SQL command executed against the database management system. In other words, simple command

```
SELECT * FROM Employee,
```

is executed within a transaction. This is default behaviour of most DBMS, however, some of them allow to disable this functionality. If so then, each SQL command executed has to be ”manually” committed or rolled back with SQL commands COMMIT and ROLLBACK respectively. For example, disabling the auto commit is possible in MySQL and Oracle database servers by simply running an SQL command

```
SET AUTOCOMMIT=0
```

for MySQL and

```
SET AUTOCOMMIT OFF
```

for Oracle server. When auto commit is enabled then one can start the transaction by executing

```
BEGIN TRANSACTION
```

or

```
START TRANSACTION
```

commands. When started all successive SQL commands are executed within transactions and are not committed to the database until this is done explicitly by COMMIT command that commits all the commands from the transaction. Obviously, transaction can be also rolled back with ROLLBACK command which yields all the commands from the transaction to be rolled back also.

By definition transaction has to meet four requirements that are often refered to by the acronym ACID:

- atomicity,
- consistency,
- isolation,

- durability.

Atomicity

Atomicity states that transactions are executed on ”all-or-nothing” basis which means that all operations included within single transaction are either executed correctly or terminated and rolled back. If all operations are executed correctly then transaction is committed and all changes are permanently stored in the database. On the other hand, if single operation from the transaction fails then transaction is terminated and all changes already made within that transaction are cancelled and rolled back. No changes made by transactions that were rolled back are stored in the database.

Consistency

Consistency in database management systems refers to its truthfulness that it stores correct representation of real word and assurance that every modification made to the database transforms one consistent state into another. Consistency is ensured based on integrity rules that determine the conditions and dependencies between data in the database required to maintain its consistent state. Transaction consistency ensure that when transaction terminates (either due to commitment or rollback) the state of the database is correct. If, for some reason, a transaction violates integrity rules, the entire transaction is terminated, rolled back and the database returns to initial, consistent state.

Following example presents that ensuring data consistency when multiple transactions run in parallel may be difficult.

EXAMPLE 6.7
Listing 6.1 presents two transactions that access sequentially the same row of the table. Purpose of both transactions is to verify whether there is enough money on account 123 to withdraw the required amount. If the amount of money is large enough then new amount (after withdraw) is stored in the database. If the amount available on account is to small then no money is withdraw since bank does not allow debits. Now, since both transactions run simultaneously thus it may happen that there is enough money for each withdraw but to less to withdraw both amounts. Consequently, both transactions successfully manage to withdraw required amount despite fact, that the resulting balance will be negative. As a result there is a debit on account 123 which can be treated as inconsistency since bank does not allow debits and so this would not happen in real live.

□

Listing 6.1: Consistency challenges in simultaneous execution of two transactions

```
   Transaction 1                                 Transaction 2
-----------------------------------------------------------------------------------
1. BEGIN TRANSACTION
2. SELECT amount FROM Account WHERE id=123
3.                                              BEGIN TRANSACTION
4.                                              SELECT amount FROM Account
                                                   WHERE id=123
5   if (amount>=withdraw) {
      UPDATE Account
          SET amount=amount-withdraw
             WHERE id=123
    }
6.                                              if (amount>=withdraw){
                                                  UPDATE Account
                                                      SET amount=amount-withdraw
                                                      WHERE id=123
                                                }
7. COMMIT                                       COMMIT
```

Isolation

Isolation refers to the requirement that concurrent transactions cannot access data that has been modified during a transaction that has not been completed. In other words, each transaction must remain unaware of other concurrently executing transactions with one exception – transaction may be forced to wait for another transaction to complete if it requires the data already modified by to other one. Strict isolation, however, prevents transactions from running concurrently so quite often it is desirable to sacrify isolation at cost of greater concurrency. Less restrictive isolation may lead to *anomalies*, which occur when one transaction access data already read/modified by other transaction. There are three main anomalies:

- dirty read,
- unrepeatable read,
- phantoms.

Dirty read, also called *read uncommitted* occurs when transaction reads data that were modified by other transaction that has not finished yet. In such a case transaction reads data, that may disappear from the database if the transaction that have modified it is rolled back.

Listing 6.2: Dirty read in simultaneous execution of two transactions

```
   Transaction 1                                  Transaction 2 (READ UNCOMMITTED)
---------------------------------------------------------------------------------
1. BEGIN TRANSACTION
2.                                                BEGIN TRANSACTION
3. UPDATE Employee
     SET salary=salary+1000 WHERE id=123
4.                                                SELECT salary FROM Employee
                                                    WHERE id=123
5. ROLLBACK
6.                                                COMMIT
```

Example 6.8
Listing 6.2 presents two concurrent transactions that access the same data. After transaction 1 updates the data in the Employee table, but before it terminates, second transaction reads contents of the table. Since transaction 1 has already modified row in the Employee table, thus transaction 2 will get its new value in its result set. However, after a while transaction 1 is rolled back which cause all the modification made by this transaction to be deleted from the database (database returns to its original state, i.e. consistent state prior to transaction 1). It follows that transaction 2 have read data that has never existed in a table, since transaction 1 was never committed.

□

Unrepeatable read occurs when transaction reads the same data twice (or more) and every time the resulting result set contains the same rows but data in these rows changes. Such situation may happen when one transaction is reading the table several times while at the same time another transaction updates information that is part of the result set.

Example 6.9

Two transactions presented on listing 6.3 run concurrently and access the same data. At the beginning transaction 2 selects all the employees that live in Warsaw. While processing the successive commands of transaction, transaction 1 updates phone numbers for all employees living in Warsaw by adding proper area code. Consequently, when transaction 2 again selects information about all the employees living in Warsaw the result set contains the same rows but data in that rows has changed. □

Listing 6.3: Unrepeatable read in simultaneous execution of two transactions

```
   Transaction 1                              Transaction 2 (UNREPEATABLE READ)
---------------------------------------------------------------------------------
1. BEGIN TRANSACTION
2.                                           BEGIN TRANSACTION
3.                                           SELECT salary FROM Employee
                                                WHERE id=123
4. UPDATE Employee
      SET salary=salary+1000 WHERE id=123
5. COMMIT
6.                                           SELECT salary FROM Employee
                                               WHERE id=123
7.                                           COMMIT
```

The last anomaly that may occur when transactions are run concurrently are phantom reads. This term denotes situation in which transaction, that executes the same SELECT command within single transaction, gets different result sets – i.e. result sets that consist of different set of rows, despite fact, that the same command was executed. Such situation may occur when one transaction executes the same command several times while another transaction is deleting old or inserting new data that meet criteria specified in WHERE clause of the SELECT statement.

Example 6.10
In listing 6.4 transaction 2 executes the same SQL command twice but prior to second execution transaction 1 inserts new row into the table Employee. Consequently, second SELECT command returns larger set of rows. However, the additional row returned is not a valid, since transaction 1 was later on rolled back.

□

It is important to notice that uncommitted read occurs when transaction tries to read data modified by another transaction that has not been finalised yet. On the other hand unrepeatable reads and phantoms occur when transaction modifies data that were already accessed (read) by another transaction, that will access the same data again before it terminates. It follows that to prevent anomalies it is required to:

- prevent transaction from reading data from unfinished transactions,
- prevent transaction from accessing the rows that were already accessed by another transaction that is not completed,
- prevent transaction from adding/deleting rows of table that meet criteria specified in WHERE clause used in another transaction that is not completed.

Listing 6.4: Phantom reads in simultaneous execution of two transactions

```
   Transaction 1                                  Transaction 2
--------------------------------------------------------------------------------
1. BEGIN TRANSACTION
2.                                               BEGIN TRANSACTION
3.                                               SELECT salary FROM Employee
4. INSERT INTO Employee (id, salary, name)
      VALUES (148, 10000, 'John Smith')
5.                                               SELECT salary FROM Employee
6. ROLLBACK
7.                                               COMMIT
```

This is achieved with transaction *isolation levels* that implement above mentioned restrictions. The less restrictive isolation level, and thus the most concurrent one, is called READ UNCOMMITTED. When transaction is run in this isolation level then it is allowed to read all the data, even if it was modified by transaction that haven't finished yet. Next isolation level is called READ COMMITTED. Transactions running on this isolation level are not allowed to read data that was modified by other transaction until this transaction is finished (either committed or rolled back). READ COMMITTED prevents transaction from reading data that may be rolled back in a while, however it does not ensure that data will not be modified by other transactions. Third isolation level, REPEATABLE READS, ensures that transaction running with this isolation level prevents any other transaction from modifying the data it has accessed. The last, and the most restrictive isolation level is SERIALIZABLE. When transaction is run in this isolation level then no other transaction can modify, insert or delete any data that satisfies WHERE clauses from all the commands executed in this transaction. Serializable forces all transactions to run sequentially so it degrades benefits that arise from concurrency, however, ensure anomalies will not occur.

In all isolation levels, when transaction cannot execute some operation then it is locked until the requested data is available again and can be accessed. To ensure transactions do not lock each other for long it is required to make them as short as possible, so they run for a short period of time and terminate. If it is required to lock access to some rows for longer, then other techniques (e.g. table locks) should be used.

Four isolation levels mentioned above allow to ensure that anomalies will not occur. Table 6.5 presents isolation levels and anomalies they prevent. Transaction isolation level can be controlled by user of the database management server and specified before the transaction is started. If not specified then server default isolation level is used. To specify isolation transaction level it is enough to execute an SQL command

Table 6.5: Isolation levels and anomalies they prevent. Plus sign (+) means that particular isolation level prevents anomaly; Minus (-) means that it does not prevent anomaly, so it may occur.

	Anomaly		
Isolation level	uncommitted read	unrepeatable reads	phantoms
READ UNCOMMITTED	–	–	–
READ COMMITTED	+	–	–
REPEATABLE READ	+	+	–
SERIALIZABLE	+	+	+

SET TRANSACTION ISOLATION LEVEL *isolation_level*,

where *isolation_level* specifies the name of isolation level requested.

Durability

Durability is the ability to recover the committed transaction if any kind of system failure occurred (either hardware or software error). Durability guarantees that once successful finalisation of a transaction has been notified to user, there is no possibility that the results of transaction are lost (not stored in the database). Whatever happens transaction's data changes will survive any system failure as well as all integrity constraints are satisfied. Durability does not imply a permanent state of the database. A subsequent transaction may modify data changed by a prior transaction without violating the durability principle.

Bibliography

[1] SQL 1992. *Information Technology - Database Language SQL*. Digital Equipment Corporation, 1992.

[2] SQL 2003. *ISO/IEC 9075:2003 Information Technology - Database Language SQL*. Digital Equipment Corporation, 2003.

[3] Microsoft SQL Server 2008. *SQL Server Books Online*. Microsoft Corporation, 2011. http://msdn.microsoft.com/en-us/library/ms130214%28v=sql.100%29.aspx.

[4] IEEE 754-2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE Computer Society, 2008. http://standards.ieee.org/findstds/standard/754-2008.html.

[5] David Elliott Bell. Looking back at the bell-la padula model. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 337–351, Washington, DC, USA, 2005. IEEE Computer Society.

[6] David Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE CORP Bedford MA, November 1973. http://handle.dtic.mil/100.2/AD770768.

[7] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[8] Edgar F. Codd. *The Relational Model for Database Management*. Addison-Wesley, 1990.

[9] Ramez Elmasri and Navathe Shamkant. *Fundamentals of Database Systems*. Addison Wesley, 6 edition, 2010.

[10] Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. *A First Course in Database Systems*. Prentice Hall, 3 edition, 2007.

[11] XML in Microsoft SQL Server. *Implementing XML in SQL Server*. Microsoft Corporation, 2011. http://technet.microsoft.com/en-us/library/ms189887%28SQL.100%29.aspx.

[12] XML DML in Microsoft SQL Server. *XML Data Modification Language (XML DML)*. Microsoft Corporation, 2011. http://technet.microsoft.com/en-us/library/ms177454%28SQL.100%29.aspx.

[13] XPath language. *XML Path Language (XPath) 2.0*. W3C Consortium, December 2010. http://www.w3.org/TR/xpath20/.

[14] XQuery language. *XQuery 1.0: An XML Query Language*. W3C Consortium, December 2010. http://www.w3.org/TR/xquery/.

[15] TCSEC. *Trusted Computer System Evaluation Criteria*. Department of Defense, December 1983. http://csrc.nist.gov/publications/history/dod85.pdf.

Index