Structured Query Language/Standard Track Print

Introduction

Contents

It's a Translation and a Guide

This Wikibook introduces the programming language SQL as defined by ISO/IEC. The standard — similar to most standard publications — is quite technical and neither easy to read nor understand. There is therefore a demand for a text document explaining the key features of the language. That is what this wikibook strives to do: present a readable, understandable introduction for everyone interested in the topic.

Manuals and white papers by database vendors are mainly focused on technical aspects of their product. As they want to set themselves apart from each other, they tend to emphasize those aspects which go beyond the SQL standard and the products from other vendors. This is contrary to the wikibooks approach: we want to emphasize the common aspects.

The main audience of this wikibook is, therefore, people who want to learn the language, either as a beginner or for someone with existing knowledge and some degree of experience looking for a recapitulation.

What this Wikibook is not

First of all, this wikibook is not a reference manual for the syntax of standard SQL or any of its implementations. Reference manuals usually consist of definitions and explanations for those definitions. By contrast, this wikibook tries to present concepts and basic commands through textual descriptions and examples. Of course some syntax will be demonstrated. On some pages there are additonal hints about slightly differences between the standard and special implementations.

This wikibook is also not a complete tutorial. First, its focus is the standard and not any concrete implementation. When learning a computer language it is necessary to work with it and experience it personally. Hence, a concrete implementation is needed. And most of them differ more or less from the standard. Second, this wikibook is far away from reflecting the complete standard, e.g. the central part of the standard consists of about 18 MB text in more than 1,400 pages. But this wikibook can be used as a companion for learning about SQL.

How to proceed

For everyone new to SQL, it will be necessary to study the chapters and pages from beginning to end. For persons who have some experience with SQL or who are interested in a specific aspect, it is possible to navigate directly to any page.

Knowledge about any other computer language is not necessary, but it will be helpful.

This wikibook consists of descriptions, definitions, and examples. It should be read with care. Furthermore, it is absolutely necessary to personally do some experiments with data and data structures. Hence, **access to a concrete database system** where read-only and read-write tests can be done is necessary. For those tests, our example database or individually defined tables and data can be used.

Conventions

The elements of the language SQL are case-insensitive, e.g.: it makes no difference whether you write *SELECT ...*, *Select ...*, *select ...* or any combination of upper and lower case characters like *SeLecT ...*. For readability reasons, this wikibook uses the convention that all language keywords are written in upper case letters and all names of user objects e.g. table and column names, are written in lower case letters.

We will write short SQL commands within one row.

```
SELECT street FROM address WHERE city = 'Duckburg';
```

For longer commands spawning multiple lines we use a *tabular format*.

```
SELECT street
FROM   address
WHERE  city IN ('Duckburg', 'Gotham City', 'Hobbs Lane');
```

Advice: Storing and retrieving **text data** is case sensitive! If you store a cityname 'Duckburg' you cannot retrieve it as 'duckburg'.

Historical Context

One of the original scopes of computer applications was storing large amounts of data on mass storage devices and retrieving them at a later point in time. Over time user requirements increased to include not only sequential access but also random access to data records, concurrent access by parallel (writing) processes, recovery after hardware and software failures, high performance, scalability, etc. In the 1970s and 1980s, the science and computer industries developed techniques to fulfill those requests.

What makes up a Database Management System?

Basic bricks for efficient data storage - and for this reason for all Database Management Systems (DBMS) - are implementations of fast read and write access algorithms to data located in central memory and mass storage devices like routines for B-trees, Index Sequential Access Method (ISAM), other indexing techniques as well as buffering of dirty and non-dirty blocks. These algorithms are not unique to DBMS. They also apply to file systems, some programming languages, operating systems, application server and much more.

In addition to the appropriation of these routines, a DBMS guarantees compliance with the **ACID** paradigm. This compliance means, that in a multi-user environment all changes to data within one transaction are:

Atomic: all changes take place or none.
Consistent: changes transform the database from one valid state to another valid state.
Isolated: transactions of different users working at the same time will not affect each other.
Durable: the database retains committed changes even if the system crashes afterwards.

Classification of DBMS Design

A distinction between the following generations of DBMS design and implementation can be made:

- Hierarchical DBMS: Data structures are designed in a hierarchical parent/child model where every child has exactly **one** parent (with the exception of the root structure, which has no parent). The result is that the data is modeled and stored as a tree. Child rows are physically stored directly after the owning parent row. So there is no need to store the parent's ID or something like it within the child row (XML realizes a similar approach). If an application processes data in **exactly this hierarchical way**, it is very fast and efficient. But if it's necessary to process data in a sequence, which deviates from this order, access is less efficient. Furthermore, hierarchical DBMSs do not provide the modeling of n:m relations. Another fault is that there is no possibility to

navigate directly to data stored in lower levels. You must first navigate over the given hierarchy before reaching that data.

The best-known hierarchical DBMS is IMS from IBM.

- Network DBMS: The network model designs data structures as a complex network with links from one or more parent nodes to one or more child nodes. Even cycles are possible. There is no need for a single root node. In general the terms *parent node* and *child node* lose their hierarchical meaning and may be referred as *link source* and *link destination*. Since those links are realized as physical links within the database, applications which **follow the links** show good performance.

- Relational DBMS: The relational model designs data structures as relations (tables) with attributes (columns) and the relationship between those relations. Definitions in this model are expressed in a **pure declarative way** not predetermining any implementation issues like links from one relation to another or a certain sequence of rows in the database. Relationships are based purely upon content. At runtime all linking and joining is done by evaluating the actual data values, e.g.: `... WHERE employee.department_id = department.id ...`. The consequence is that - with the exception of explicit foreign keys - there is no meaning of a parent/child or owner/member denotation. Relationships in this model do not have any direction.

  The relational model and SQL are based on the mathematical theory of relational algebra.
  During the 1980s and 1990s proprietary and open source DBMS's based on the relational design paradigm established themselves as market leaders.

- Object oriented DBMS: Nowadays most applications are written in an object oriented programming language (OOP). If, in such cases, the underlaying DBMS belongs to the class of relational DBMS, the so called object-relational impedance mismatch arises. That is to say, in contrast to the application language pure relational DBMS (prDBMS) does not support central concepts of OOP:

  Type system: OOPs do not only know primitive data types. As a central concept of their language they offer the facility to define classes with complex internal structures. The classes are built on primitive types, system classes, references to other or the same class. prDBMS knows only predefined types. Secondary prDBMS insists in first normal form, which means that attributes must be scalar. In OOPs they may be sets, lists or arrays of the desired type.
  Inheritance: Classes of OOPs may inherit attributes and methods from their superclass. This concept is not known to prDBMS.
  Polymorphism: The runtime system can decide via late binding which one of a group of methods with the same name and parameter types will be called. This concept is not known by prDBMS.
  Encapsulation: Data and access methods to data are stored within the same class. It is not possible to access the data directly - the only way is using the access methods of the class. This concept is not known to prDBMS.

  Object oriented DBMS are designed to overcome the gap between prDBMS and OOP. At their peak, they reached a weak market position in the mid and late 1990s. Afterwards some of their concepts were incorporated into the SQL standard as well as rDBMS implementations.

- NoSQL: The term NoSQL stands for the emerging group of DBMS which differs from others in central concepts:
  - They do not necessarily support all aspects of the ACID paradigm.
  - The data must not necessarily be structured according to any schema.
  - Their goal is the support for fault-tolerant, distributed data with very huge volume, see also: CAP theorem.
  - Implementations differ widely in storing techniques: you can see key-value stores, document oriented databases, graph oriented databases and more.

  - They do not offer an SQL interface. In 2011 an initiative started to define an alternative language: *Unstructured Query Language* as part of SQLite.

- NewSQL: This class of DBMS seeks to provide the same scalable performance as NoSQL systems while still maintaining the ACID paradigm, the relational model and the SQL interface. They try to reach scalability by eschewing heavyweight recovery or concurrency control.

The Theory

A relational DBMS is an implementation of data stores according to the design rules of the relational model. This approach allows operations on the data according to the relational algebra like projections, selections, joins, set operations (union, difference, intersection, ...) and more. Together with Boolean algebra (and, or, not, exists, ...) and other mathematical concepts, relational algebra builds up a complete mathematical system with basic operations, complex operations and transformation rules between the operations. Neither a DBA nor an application programmer needs to know the relational algebra. But it is helpfull to know that your rDBMS is based on this mathematical foundation - and that it has the freedom to transform queries into several forms.

The Data Model

The relational model designs data structures as relations (tables) with attributes (columns) and the relationship between those relations. The information about one entity of the real world is stored within one row of a table. However, the term *one entity of the real world* must be used with care. It may be that our intellect identifies a machine like a single airplane in this vein. Depending on the information requirements it may be sufficient to put all of the information into one row of a table *airplane*. But in many cases it is necessary to break up the entity into its pieces and model the pieces as discrete entities including the relationship to the whole thing. If, for example, information about every single seat within the airplane is needed, a second table *seat* and some way of joining seats to airplanes will be required.

This way of breaking up information about real entities into a complex data model depends highly on the information requirements of the business concept. Additionally there are some formal requirements, which are independent of any application: the resulting data model should conform to a so-called normal form. Normally these data models consist of a great number of tables and relationships between them. Such models will not predetermine their use by applications; they are strictly descriptive and will not restrict access to the data in any way.

Some more Basics

Operations within databases must have the ability to act not only on single rows, but also on sets of rows. Relational algebra offers this possibility. Therefore languages based on relational algebra, e.g.: SQL, offer a powerful syntax to manipulate a great bunch of data within one single command.

As operations within relational algebra may be replaced by different but logically equivalent operations, a language based on relational algebra should not predetermine how its syntax is mapped to operations (the execution plan). The language should describe **what** should be done and not **how** to do it. Note: This choice of operations does not concern the use or neglect of indices.

As described before the relational model tends to break up objects into sub-objects. In this and in other cases it is often necessary to collect associated information from a bunch of tables into one information unit. How is this possible without links between participating tables and rows? The answer is: All joining is done based on the **values** which are actually stored in the attributes. The rDBMS must make its own decisions about how to reach all concerned rows: whether to read all potentially affected rows and ignore those which are irrelevant (full table scan) or, to use some kind of index and read only those which match the criteria. This value-based approach allows even the use of operators other than the equal-operator, e.g.:

```
SELECT * FROM gift JOIN box ON gift.extent < box.extent;
```

This command will join all "gift" records to all "box" records with a larger "extent" (whatever "extent" means).

History

As outlined above, rDBMS acts on the data with operations of relational algebra like projections, selections, joins, set operations (union, except and intersect) and more. The operations of relational algebra are denoted in a mathematical language which is highly formal and hard to understand for end users and - possibly also - for many software engineers. Therefore rDBMS offers a layer above relational algebra, which is easy to understand but nevertheless can be mapped to the underlying relational operations. Since the 1970s we have seen some languages doing this job, one of them was SQL - another example was QUEL. In the early 1980s (after a rename from its original name *SEQUEL* due to trademark problems) SQL achieved market dominance. And in 1986 SQL was standardized for the first time. The current version is SQL 2011.

Characteristics

The tokens and syntax of SQL are oriented on **English common speech** to keep the access barrier as small as possible. An SQL command like `UPDATE employee SET salary = 2000 WHERE id = 511;` is not far away from the sentence "Change employee's salary to 2000 for the employee with id 511."

The next simplification is that all key words of SQL can be expressed in any combination of upper and lower case characters. It makes no difference whether `UPDATE`, `update`, `Update`, `UpDate` or any other combination of upper and lower case characters is written. The keywords are **case insensitive**.

Next SQL is a **descriptive** language, not a procedural one. It does not pre-decide all aspects of the relational operations (which operation, their order, ...) which are generated from the given SQL statement. The rDBMS has the freedom to generate more than one execution plan from a statement. It compares the generated execution plans with each other and runs the one it thinks is best in the given situation. Additionally the end user is freed from all the gory details of data access, e.g.: Which one of a set of WHERE criteria should be evaluated first if they are combined with AND?

Despite those simplifications SQL is very powerful. Especially since it allows the manipulation of a **set of data** records with one single

statement. `UPDATE employee SET salary = salary * 1.1 WHERE salary < 2000;` will affect all employee records with an actual salary smaller than 2000. Potentially, there may be thousands of those records, only a few or even zero. It may also be noted that the operation is not a fix manipulation. The wording `SET salary = salary * 1.1` leads to an increase of the salaries by 10%, which may be 120 for one employee and 500 for another one.

The designer of SQL tried to define the language elements **orthogonally** to each other. Among other things this refers to the fact that any language element may be used in all positions of a statement where the result of that element may be used directly. E.g.: If you have a function power() which takes two numbers and returns another number, you can use this function in all positions where numbers are allowed. The following statements are syntactically correct (if you have defined the function power()) - and lead to the same resulting rows.

```
SELECT salary FROM employee WHERE salary < 2048;
SELECT salary FROM employee WHERE salary < power(2, 11);
SELECT power(salary, 1) FROM employee WHERE salary < 2048;
```

Another example of orthogonality is the use of subqueries within UPDATE, INSERT, DELETE or inside another SELECT statement.

However, SQL is not free of **redundancy**. Often there are several possible formulations to express the same situation.

```
SELECT salary FROM employee WHERE salary < 2048;
SELECT salary FROM employee WHERE NOT salary >= 2048;
SELECT salary FROM employee WHERE salary between 0 AND 2048; -- 'BETWEEN' includes edges
```

This is a very simple example. In complex statements there may be the choice between joins, subqueries and the *exists* predicate.

Fundamentals

Core SQL consists of statements. Statements consist of key words, operators, values, names of system- and user-objects or functions. Statements are concluded by a semicolon. In the statement `SELECT salary FROM employee WHERE id < 100;` the tokens SELECT, FROM and WHERE are key words. salary, employee and id are object names, the "<" sign is an operator and "100" is a value.

The SQL standard arranges statements into 9 groups:

> "The main classes of SQL-statements are:
>
> SQL-schema statements; these may have a persistent effect on the set of schemas.
> SQL-data statements; some of these, the SQL-data change statements, may have a persistent effect on SQL data.
> SQL-transaction statements; except for the <commit statement>, these, and the following classes, have no effects that persist when an SQL-session is terminated.
> SQL-control statements.
> SQL-connection statements.
> SQL-session statements.
> SQL-diagnostics statements.
> SQL-dynamic statements.
> SQL embedded exception declaration."

This detailed grouping is unusual in common speech. Usually it is distinguish between three groups:

> *Data Definition Language* (**DDL**): Managing the structure of database objects (CREATE/ALTER/DROP tables, views, columns, ...)
> *Data Manipulation Language* (**DML**): Managing and retrieval of data with the statements INSERT, UPDATE, MERGE, DELETE, SELECT, COMMIT, ROLLBACK and SAVEPOINT.
> *Data Control Language* (**DCL**): Managing access rights (GRANT, REVOKE).

Hint: In some publications the SELECT statement is said to build its own group *Data Query Language*. This group has no other statements than SELECT.

Turing completeness

Core SQL as described above is not Turing complete. It misses conditional branches, variables, subroutines. But the standard as well as most implementations offers an extension to fulfill the demand for Turing completeness. In 'Part 4: Persistent Stored Modules (SQL/PSM)' of the standard there are definitions for IF-, CASE-, LOOP-, assignment- and other statements. The existing implementations of this part have different names, different syntax and also a different scope of operation: PL/SQL in Oracle, SQL/PL in DB2, Transact-SQL or T-SQL in SQL Server and Sybase, PL/pgSQL in Postgres and simply 'stored procedures' in MySQL.

Benefit of Standardization

Like most other standards the main purpose of SQL is **portability**. Usually software designers and application developers structure and solve problems in layers. Every abstraction level is realized in its own component or sub-component: presentation to end user, business logic, data access, data storage, net and operation system demands are typical representatives of such components. They are organized as a stack and every layer offers an interface to the upper layers to use its functionality. If one of those components is realized by two different providers and both offer the same interface (as an API, Web-Service, language specification, ...) it is possible to exchange them without changing the layers which are based on them. In essence the software industry needs **stable interfaces** at the top of important layers to avoid dependence on a single provider. SQL acts as such an interface to relational database systems.

If an application uses only those SQL commands which are defined within standard SQL, it should be possible to exchange the underlaying rDBMS with a different one without changing the source code of the application. In practice this is a hard job, because concrete implementations offer numerous additional features and software engineers love to use them.

A second aspect is the **conservation of know how**. If a student learns SQL, he is in a position to develop applications which are based on an arbitrary database system. The situation is comparable with any other popular programming language. If one learns Java or C-Sharp, he can develop applications of any kind running on a lot of different hardware systems and even different hardware architectures.

Limits

Database systems consist of many components. The access to the data is an important but not the only component. Additionally there are many more tasks: throughput optimization, physical design, backup, distributed databases, replication, 7x24 availability, ... Standard SQL is focused mainly on data access and ignores typical DBA tasks. Even the `CREATE INDEX` statement as a widely used optimization strategy is not part of the standard. Nevertheless the standard fills thousands of pages. But most of the DBA's daily work is highly specialized to every concrete implementation and must be done in a different way when he switches to a different rDBMS. Mainly application developers benefit from SQL.

The Standardization Process

The standardization process is organized in two levels. The first level acts in a national context. Interested companies, universities and persons of one country work within their national standardization organisation like ANSI, Deutsches Institut für Normung (DIN) or British Standards Institution (BSI), where every member has one vote. The second level is the international stage. The national organizations are members of ISO respectively IEC. In case of SQL there is a common committee of ISO and IEC named Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 32, Data management and interchange, where every national body has one vote. This committee approve the standard under the name *ISO/IEC 9075-n:yyyy*, where *n* is the part number and *yyyy* is the year of publication. The nine parts of the standard are described in short here.

If the committee releases a new version, this may concern only some of the nine parts. So it is possible that the *yyyy* denomination differs from part to part. *Core SQL* is defined mainly by the second part: *ISO/IEC 9075-2:yyyy Part 2: Foundation (SQL/Foundation)* - but it contains also some features of other parts.

Note: The API JDBC is part of Java SE and Java EE but not part of the SQL standard.

The standard is complemented by a second, closely related standard: *ISO/IEC 13249-n:yyyy SQL Multimedia and Application Packages*, which is developed by the same organizations and committee. This publication defines interfaces and package based on SQL. They focus on special kind of applications: text, pictures, data mining and spatial data applications.

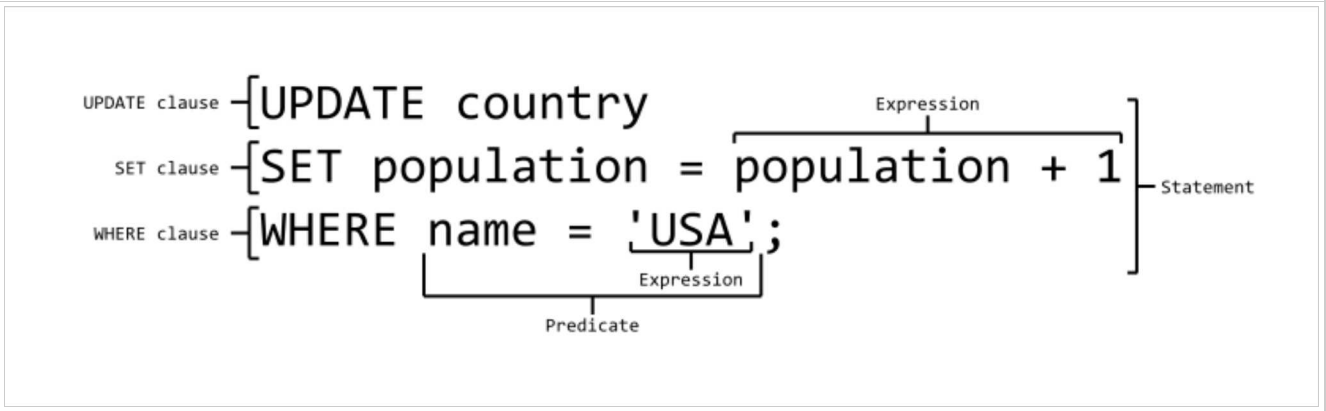
Verification of Conformance to the Standard

Until 1996 the National Institute of Standards and Technology (NIST) certified the compliance of the SQL implementation of rDBMS with the SQL standard. As NIST abandon this work, nowadays vendors self-certify the compliance of their product. They must declare the degree of conformance in a special appendix of their documentation. This documentation may be voluminous as the standard defines not only a set of base features - called *Core SQL:yyyy* - but also a lot of additional features an implementation may conform to or not.

Implementations

To fulfill their clients' demands all major vendors of rDBMS offers - among other data access ways - the language SQL within their product. The implementations cover *Core SQL*, a bunch of additional standardized features and a huge number of additional, not standardized features. The access to standardized features may use the regular syntax or an implementation specific syntax. In essence SQL is the clamp holding everything together, but normally there are a lot of detours around the official language.

SQL consists of statements which start with a key word like SELECT, DELETE or CREATE and terminate with a semicolon. Their elements are case-insensitive with the exception of fixed character string values like 'Mr. Brown'.



An example of a complete statement with several of its language elements.

- *Clauses*: Statements are subdivided into clauses. The most popular one is the WHERE clause.
- *Predicates*: Predicates specify conditions which can be evaluated to a boolean value. E.g.: a boolean comparison, BETWEEN, LIKE, IS NULL, IN, SOME/ANY, ALL, EXISTS.
- *Expressions*: Expressions are numeric or string values by itself, or the result of arithmetic or concatenation operators, or the result of functions.
- *Object names*: Names of database objects like tables, views, columns, functions.
- *Values*: Numeric or string values.
- *Arithmetic operators*: The plus sign, minus sign, asterisk and solidus (+, –, * and /) specify addition, subtraction, multiplication and division.
- *Concatenation operator*: The '||' sign specifies the concatenation of character strings.
- *Comparison operators*: The equals operator, not equals operator, less than operator, greater than operator, less than or equals operator, greater than or equals operator (=, <>, <, >, <=, >=) compares values and expressions.
- *Boolean operators*: AND, OR, NOT combines boolean values.

Create Table

Data Types

More Details

```
--
-- Frequently used data types and simple constraints
CREATE TABLE t_standard (
  -- column name   data type     default        nullable/constraint
  id               DECIMAL                      PRIMARY KEY,  -- some prefer the name: 'sid'
  col_1            VARCHAR(50)   DEFAULT 'n/a'  NOT NULL,     -- string with variable length. Oracle: 'VARCHAR2'
  col_2            CHAR(10),                                  -- string with fixed length
  col_3            DECIMAL(10,2) DEFAULT 0.0,                 -- 8 digits before and 2 after the decimal. Signed.
  col_4            NUMERIC(10,2) DEFAULT 0.0,                 -- same as col_3
  col_5            INTEGER,
  col_6            BIGINT                                     -- Oracle: use 'NUMBER(n)', n up to 38
);

-- Data types with temporal aspects
CREATE TABLE t_temporal (
  -- column name   data type     default  nullable/constraint
  id               DECIMAL                PRIMARY KEY,
  col_1            DATE,                                  -- Oracle: contains day and time, seconds without decimal
  col_2            TIME,                                  -- Oracle: use 'DATE' and pick time-part
  col_3            TIMESTAMP,                             -- Including decimal for seconds
  col_4            TIMESTAMP WITH TIME ZONE,              -- MySql: no time zone
  col_5            INTERVAL YEAR TO MONTH,
  col_6            INTERVAL DAY TO SECOND
```

```
);

CREATE TABLE t_misc (
  -- column name   data type     default  nullable/constraint
  id               DECIMAL                PRIMARY KEY,
  col_1            CLOB,                                  -- very long string (MySql: LONGTEXT)
  col_2            BLOB,                                  -- binary, eg: Word document or mp3-stream
  col_3            FLOAT(6),                              -- example: two-thirds (2/3).
  col_4            REAL,
  col_5            DOUBLE PRECISION,
  col_6            BOOLEAN,                               -- Oracle: Not supported
  col_7            XML                                    -- Oracle: 'XMLType'
);
```

Constraints

More Details

```
--
-- Denominate all constraints with an expressive name, eg.: abbreviations for
-- table name (unique across all tables in your schema), column name, constraint type, running number.
--
CREATE TABLE myExampleTable (
  id               DECIMAL,
  col_1            DECIMAL(1),   -- only 1 (signed) digit
  col_2            VARCHAR(50),
  col_3            VARCHAR(90),
  CONSTRAINT example_pk           PRIMARY KEY (id),
  CONSTRAINT example_uniq         UNIQUE (col_2),
  CONSTRAINT example_fk           FOREIGN KEY (col_1) REFERENCES person(id),
  CONSTRAINT example_col_1_nn     CHECK (col_1 IS NOT NULL),
  CONSTRAINT example_col_1_check  CHECK (col_1 >=0 AND col_1 < 6),
  CONSTRAINT example_col_2_nn     CHECK (col_2 IS NOT NULL),
  CONSTRAINT example_check_1      CHECK (LENGTH(col_2) > 3),
  CONSTRAINT example_check_2      CHECK (LENGTH(col_2) < LENGTH(col_3))
);
```

Foreign Key

More Details

```
--
-- Reference to a different (or the same) table. This creates 1:m or n:m relationships.
CREATE TABLE t_hierarchie (
  id               DECIMAL,
  part_name        VARCHAR(50),
  super_part_id    DECIMAL,      -- ID of the part which contains this part
  CONSTRAINT hier_pk             PRIMARY KEY (id),
  -- In this special case the foreign key refers to the same table
  CONSTRAINT hier_fk             FOREIGN KEY (super_part_id) REFERENCES t_hierarchie(id)
);

-- ------------------------------------------------
--             n:m relationships
-- ------------------------------------------------
CREATE TABLE t1 (
  id               DECIMAL,
  name             VARCHAR(50),
  -- ...
  CONSTRAINT t1_pk               PRIMARY KEY (id)
);
CREATE TABLE t2 (
  id               DECIMAL,
  name             VARCHAR(50),
  -- ...
  CONSTRAINT t2_pk               PRIMARY KEY (id)
);
CREATE TABLE t1_t2 (
  id               DECIMAL,
  t1_id            DECIMAL,
  t2_id            DECIMAL,
  CONSTRAINT t1_t2_pk            PRIMARY KEY (id),       -- also this table should have its own Primary Key
  CONSTRAINT t1_t2_unique        UNIQUE (t1_id, t2_id),  -- every link should occur only once
  CONSTRAINT t1_t2_fk_1          FOREIGN KEY (t1_id) REFERENCES t1(id),
  CONSTRAINT t1_t2_fk_2          FOREIGN KEY (t2_id) REFERENCES t2(id)
);

-- ---------------------------------------------------------------------------------
--       ON DELETE / ON UPDATE / DEFFERABLE
-- ---------------------------------------------------------------------------------
  -- DELETE and UPDATE behaviour for child tables (see first example)
  -- Oracle: Only DELETE [CASCADE | SET NULL] is possible. Default is NO ACTION, but this cannot be
  --         specified explicit - just omit the phrase.
  CONSTRAINT hier_fk             FOREIGN KEY (super_part_id) REFERENCES t_hierarchie(id)
                                 ON DELETE CASCADE  -- or: NO ACTION (the default), RESTRICT, SET NULL, SET DEFAULT
                                 ON UPDATE CASCADE  -- or: NO ACTION (the default), RESTRICT, SET NULL, SET DEFAULT

  -- Initial stage: immediate vs. deferred, [not] deferrable
  -- MySQL: DEFERABLE is not supported
  CONSTRAINT t1_t2_fk_1          FOREIGN KEY (t1_id) REFERENCES t1(id)
                                 INITIALLY IMMEDIATE DEFERRABLE
```

```
-- Change constraint characteristics at a later stage
SET CONSTRAINT hier_fk DEFERRED; -- or: IMMEDIATE
```

Alter Table

More Details

Concerning columns.

```
-- Add a column (plus some column constraints). Oracle: The key word 'COLUMN' is not allowed.
ALTER TABLE t1 ADD COLUMN col_1 VARCHAR(100) CHECK (LENGTH(col_1) > 5);

-- Change a columns characteristic. (Some implementations use different key words like 'MODIFY'.)
ALTER TABLE t1 ALTER COLUMN col_1 SET DATA TYPE NUMERIC;
ALTER TABLE t1 ALTER COLUMN col_1 SET SET DEFAULT -1;
ALTER TABLE t1 ALTER COLUMN col_1 SET  NOT NULL;
ALTER TABLE t1 ALTER COLUMN col_1 DROP NOT NULL;

-- Drop a column. Oracle: The key word 'COLUMN' is mandatory.
ALTER TABLE t1 DROP COLUMN col_2;
```

Concerning complete table.

```
--
ALTER TABLE t1 ADD   CONSTRAINT t1_col_1_uniq UNIQUE (col_1);
ALTER TABLE t1 ADD   CONSTRAINT t1_col_2_fk FOREIGN KEY (col_2) REFERENCES person (id);

-- Change definitons. Some implementations use different key words like 'MODIFY'.
ALTER TABLE t1 ALTER CONSTRAINT t1_col_1_unique UNIQUE (col_1);

-- Drop a constraint. You need to know its name. Not supported by MySQL, there is only a 'DROP FOREIGN KEY'.
ALTER TABLE t1 DROP  CONSTRAINT t1_col_1_unique;
-- As an extention to the SQL standard some implementations offer an ENABLE / DISABLE command for constraints.
```

Drop Table

More Details

```
--
-- All data and complete table structure inclusive indices are thrown away.
-- No column name. No WHERE clause. No trigger is fired. Considers Foreign Keys. Very fast.
DROP TABLE t1;
```

Select

Basic Syntax

More Details

```
--
-- Overall structure: SELECT / FROM / WHERE / GROUP BY / HAVING / ORDER BY

--     constants, column values, operators,     functions
SELECT 'ID: ',    id,            col_1 + col_2, sqrt(col_2)
FROM   t1
-- precedence within WHERE: functions, comparisions, NOT, AND, OR
WHERE  col_1 > 100
AND    NOT MOD(col_2, 10) = 0
OR     col_3 < col_1
ORDER BY col_4 DESC, col_5  -- sort ascending (the default) or descending
;

-- number of rows, number of not-null-values
SELECT COUNT(*),   COUNT(col_1) FROM t1;

-- predefined functions
SELECT COUNT(col_1), MAX(col_1), MIN(col_1), AVG(col_1), SUM(col_1) FROM t1;

-- UNIQUE values only
SELECT DISTINCT col_1 FROM t1;

-- In the next example col_1 many have duplicates. Only the combination of col_1 plus col_2 is unique.
SELECT DISTINCT col_1, col_2 FROM t1;
```

Case

More Details

```
--
-- CASE expression with conditions on exactly ONE column
SELECT id,
       CASE contact_type   -- ONE column name
         WHEN 'fixed line' THEN 'Phone'
         WHEN 'mobile'     THEN 'Phone'
         ELSE                  'Not a telephone number'
       END,
       contact_value
FROM   contact;

-- CASE expression with conditions on ANY column
SELECT id,
       CASE   -- NO column name
         WHEN contact_type IN ('fixed line', 'mobile')  THEN 'Phone'
         WHEN id = 4                                    THEN 'ICQ'
         ELSE                                                'Something else'
       END,
       contact_value
FROM   contact;
```

Grouping

More Details

```
--
SELECT product_group, count(*) AS cnt
FROM   sales
WHERE  region = 'west'   -- additional restrictions are possible but not necessary
GROUP BY product_group   -- 'product_group' is the criterion which creates groups
HAVING COUNT(*) > 1000   -- restriction to groups with more than 1000 sales per group
ORDER BY cnt;

-- Attention: in the next example col_2 is not part of the GROUP BY criterion. Therefore it cannot be displayed.
SELECT col_1, col_2
FROM   t1
GROUP BY col_1;

-- We must accumulate all col_2-values of each group to ONE value, eg:
SELECT col_1, sum(col_2), min(col_2)
FROM   t1
GROUP BY col_1;
```

Join

More Details

```
--
-- Inner join: Only persons together with their contacts.
--             Ignores all persons without contacts and all contacts without persons
SELECT *
FROM   person p
JOIN   contact c ON p.id = c.person_id;

-- Left outer join: ALL persons. Ignores contacts without persons
SELECT *
FROM      person p
LEFT JOIN contact c ON p.id = c.person_id;

-- Right outer join: ALL contacts. Ignores persons without contacts
SELECT *
FROM       person p
RIGHT JOIN contact c ON p.id = c.person_id;

-- Full outer join: ALL persons. ALL contacts.
SELECT *
FROM      person p
FULL JOIN contact c ON p.id = c.person_id;

-- Carthesian product (missing ON keyword): be carefull!
SELECT COUNT(*)
FROM   person p
JOIN   contact c;
```

Subquery

More Details

```
--
-- Subquery within SELECT clause
SELECT id,
       lastname,
       weight,
       (SELECT avg(weight) FROM person)    -- the subquery
FROM   person;

-- Subquery within WHERE clause
```

```
SELECT id,
       lastname,
       weight
FROM   person
WHERE  weight < (SELECT avg(weight) FROM person)    -- the subquery
;

-- CORRELATED subquery within SELECT clause
SELECT id,
       (SELECT status_name FROM status st WHERE st.id = sa.state)
FROM   sales sa;

-- CORRELATED subquery retrieving the highest version within each booking_number
SELECT *
FROM   booking b
WHERE  version =
  (SELECT MAX(version) FROM booking sq WHERE sq.booking_number = b.booking_number)
;
```

Set operations

More Details

```
--
-- UNION
SELECT firstname  -- first SELECT command
FROM   person
  UNION           -- push both intermediate results together to one result
SELECT lastname   -- second SELECT command
FROM   person;

-- Default behaviour is: 'UNION DISTINCT'. 'UNION ALL' must be explicitly specified, if duplicate values shall be removed.

-- INTERSECT: resulting values must be in BOTH intermediate results
SELECT firstname FROM person
  INTERSECT
SELECT lastname  FROM person;

-- EXCEPT: resulting values must be in the first but not in the second intermediate result
SELECT firstname FROM person
  EXCEPT   -- Oracle uses 'MINUS'. MySQL does not support EXCEPT.
SELECT lastname  FROM person;
```

Rollup/Cube

More Details

```
-- Additional sum per group and sub-group
SELECT SUM(col_x), ...
FROM   ...
GROUP BY ROLLUP (producer, model); -- the MySQL syntax is: GROUP BY producer, model WITH ROLLUP

-- Additional sum per EVERY combination of the grouping columns
SELECT SUM(col_x), ...
FROM   ...
GROUP BY CUBE (producer, model); -- not supported by MySQL
```

Window functions

More Details

```
-- The frames boundaries
SELECT id,
       emp_name,
       dep_name,
       FIRST_VALUE(id) OVER (PARTITION BY dep_name ORDER BY id) AS frame_first_row,
       LAST_VALUE(id)  OVER (PARTITION BY dep_name ORDER BY id) AS frame_last_row,
       COUNT(*)        OVER (PARTITION BY dep_name ORDER BY id) AS frame_count,
       LAG(id)         OVER (PARTITION BY dep_name ORDER BY id) AS prev_row,
       LEAD(id)        OVER (PARTITION BY dep_name ORDER BY id) AS next_row
FROM   employee;

-- The moving average
SELECT id, dep_name, salary,
       AVG(salary)     OVER (PARTITION BY dep_name ORDER BY salary
                             ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS sum_over_1or2or3_rows
FROM   employee;
```

Recursions

More Details

```
-- The 'with clause' consists of three parts:
```

```
-- First: arbitrary name of an intermediate table and its columns
WITH intermediate_table (id, firstname, lastname) AS
(
  -- Second: starting row (or rows)
  SELECT id, firstname, lastname
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  -- Third: Definition of the rule for querying the next level. In most cases this is done with a join operation.
  SELECT f.id, f.firstname, f.lastname
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
)

-- After the 'with clause': depth first / breadth first
-- SEARCH BREADTH FIRST BY firstname SET sequence_number (default behaviour)
-- SEARCH DEPTH FIRST BY firstname SET sequence_number

-- The final SELECT
SELECT * FROM intermediate_table;

-- Hints: Oracle supports the syntax of the SQL standard since version 11.2. .
--        MySQL does not support recursions at all and recommend procedural workarounds.
```

Insert

More Details

```
--
-- fix list of values/rows
INSERT INTO t1 (id, col_1, col_2) VALUES (6, 46, 'abc');
INSERT INTO t1 (id, col_1, col_2) VALUES (7, 47, 'abc7'),
                                         (8, 48, 'abc8'),
                                         (9, 49, 'abc9');
COMMIT;

-- subselect: leads to 0, 1 or more new rows
INSERT INTO t1 (id, col_1, col_2)
  SELECT id, col_x, col_y
  FROM   t2
  WHERE  col_y > 100;
COMMIT;

-- dynamic values
INSERT INTO t1 (id, col_1, col_2) VALUES (16, CURRENT_DATE, 'abc');
COMMIT;

INSERT INTO t1 (id, col_1, col_2)
  SELECT id,
         CASE
           WHEN col_x < 40 THEN col_x + 10
           ELSE                 col_x +  5
         END,
         col_y
  FROM   t2
  WHERE  col_y > 100;
COMMIT;
```

Update

More Details

```
--
-- basic syntax
UPDATE t1
SET    col_1 = 'Jimmy Walker',
       col_2 = 4711
WHERE  id = 5;

-- raise value of col_2 by factor 2; no WHERE ==> all rows!
UPDATE t1 SET col_2 = col_2 * 2;

-- non-correlated subquery leads to one single evaluation of the subquery
UPDATE t1 SET col_2 = (SELECT max(id) FROM t1);

-- correlated subquery leads to one evaluation of subquery for EVERY affected row of outer query
UPDATE t1 SET col_2 = (SELECT col_2 FROM t2 where t1.id = t2.id);

-- Subquery in WHERE clause
UPDATE article
SET    col_1 = 'topseller'
WHERE  id IN
  (SELECT article_id
   FROM   sales
   GROUP BY article_id
   HAVING COUNT(*) > 1000
  );
```

Merge

More Details

```
--
-- INSERT / UPDATE depending on any criterion, in this case: the two columns 'id'
MERGE INTO  hobby_shadow                    t   -- the target table
      USING (SELECT id, hobbyname, remark
             FROM   hobby
             WHERE  id < 8)                 s   -- the source
      ON    (t.id = s.id)                       -- the 'match criterion'
  WHEN MATCHED THEN
    UPDATE SET remark = concat(s.remark, ' Merge / Update')
  WHEN NOT MATCHED THEN
    INSERT (id, hobbyname, remark) VALUES (s.id, s.hobbyname, concat(s.remark, ' Merge / Insert'))
;

-- Independent from the number of affected rows there is only ONE round trip between client and DBMS
```

Delete

More Details

```
--
-- Basic syntax
DELETE FROM t1 WHERE id = 5;  -- no column name behind 'DELETE' key word because the complete row will be deleted

-- no hit is OK
DELETE FROM t1 WHERE id != id;

-- subquery
DELETE FROM person_hobby
WHERE  person_id IN
  (SELECT id
   FROM   person
   WHERE  lastname = 'Goldstein'
  );
```

Truncate

More Details

```
--
-- TRUNCATE deletes ALL rows (WHERE clause is not possible). The table structure remains.
-- No trigger actions will be fired. Foreign Keys are considered. Much faster than DELETE.
TRUNCATE TABLE t1;
```

Standard Track

Foundation

More than a Spreadsheet

Let's start with a simple example. Suppose we want to collect information about people - their name, place of birth and some more items. In the beginning we might consider to collect this data in a simple spreadsheet. But what if we grow to a successful company and have to handle millions of those data items? Could a spreadsheet deal with this huge amount of information? Could several employees or programs simultaneously insert new data, delete or change it? Of course not. And this is one of the noteworthy advantages of a DBMS over a spreadsheet program: we can imagine the structure of a table as a simple spreadsheet - but the access to it is internally organized in a way that **huge amounts** of data can be accessed by a **lot of users** at the **same time**.

In summary it can be said that one can imagine a table as a spreadsheet optimized for bulk data and concurrent access.

Conceive the Structure

To keep control and to ensure a good performance, tables are subject to a few strict rules. Every table column has a fixed name and the values of each column must be of the same data type. Furthermore, it is highly recommended - though not compulsory - that each row can be identified by a unique value. The column, in which this identifying value resides, is called the Primary Key. In this Wikibook we always name it *id*. But everybody is free to choose a different name. Furthermore we may use the concatenation of more than one column as the Primary Key.

At the beginning we have to decide the following questions:

1. What information units of persons (in this first example) do we want to save? Of course there is a lot of information about persons (e.g.: eye color, zodiacal sign, ...), but every application needs only some of them. We have to decide which ones are of interest in our concrete context.
2. What names do we assign to the selected information units? Each of the identified information units goes to a column of the table, which needs to have a name.
3. Of what data type are the information units? All data values within one column must be of the same data type. We cannot put an arbitrary string into a column of data type DATE.

In our example we decide to save first name, last name, date and place of birth, social security number, and the person's weight. Obviously date of birth is of data type DATE, the weight is a number and all others are some kind of strings. For strings there is a distinction between those that have a fixed length and those in which the length usually varies greatly from row to row. The former is named CHAR(<n>), where <n> is the **fixed** length, and the others VARCHAR(<n>), where <n> is the **maximum** length.

Fasten Decisions

The decisions previously taken must be expressed in a machine-understandable language. This language is SQL, which acts as the interface between end users - or between special programms - and the DBMS.

```
-- comment lines starts with two consecutive minus signs followed by a space '-- '
CREATE TABLE person (
  -- define columns (name / type / default value / nullable)
  id              DECIMAL      NOT NULL,
  firstname       VARCHAR(50)  NOT NULL,
  lastname        VARCHAR(50)  NOT NULL,
  date_of_birth   DATE,
  place_of_birth  VARCHAR(50),
  ssn             CHAR(11),
  weight          DECIMAL DEFAULT 0 NOT NULL,
  -- select one of the defined columns as the Primary Key and
  -- guess a meaningfull name for the Primary Key constraint: 'person_pk' may be a good choice
  CONSTRAINT person_pk PRIMARY KEY (id)
);
```

We choose *person* as the name of the table, which consists of seven columns. One of them plays the role of the Primary Key: *id*. We can store exclusively digits in the column *id* and *weight*, strings in a length up to 50 characters in *firstname*, *lastname* and *place_of_birth*, dates in *date_of_birth* and a string of exactly eleven characters in *ssn*. The phrase NOT NULL is part of the definition of *id*, *firstname*, *lastname* and *weight*. This means that in every row there must be a value for those four columns. Storing no value in any of those columns is not possible - but the 8-character-string 'no value' or the digit '0' are allowed because they are values. Or to say it the other way round: it is possible to omit the values of *date_of_birth*, *place_of_birth* and *ssn*.

The definition of a Primary Key is called a 'constraint' (later on we will get to know more kinds of constraints). Every constraint should have a name - it's *person_pk* in this example.

The Result

After execution of the above 'CREATE TABLE' command the DBMS has created an object that one can imagine similar to the following Wiki-table:

id	firstname	lastname	date_of_birth	place_of_birth	ssn	weight

This Wiki-table shows 4 lines. The first line represents the names of the columns - no values! The following 3 lines are for demonstration purposes only. But in the database table exists currently no single row! She is completely empty, no rows at all, no values at all! The only thing that exists in the database is the **structure** of the table.

Back to Start

Maybe we want to delete the table one day. To do so we can use the *DROP* command. It removes the table totally: all data and the complete structure are thrown away.

```
DROP TABLE person;
```

Don't confuse the DROP command with the DELETE command, which we present on the next page. The DELETE comand removes only rows - possibly all of them. However, the table itself, which holds the definition of the structure, keeps retained.

As shown in the previous page we now have an empty table named *person*. What can we do with such a table? Just use it like a bag! Store things in it, look into it to check the existence of things, modify things in it or throw things out of it. These are the four natural operations, which concerns data in tables:

- INSERT: put some data into the table
- SELECT: retrieve data from the table
- UPDATE: modify data, which exists in the table
- DELETE: remove data from the table.

Each of these four operations are expressed by their own SQL command. They start with a keyword and runs up to a semicolon at the end. This rule applies to all SQL commands: They are introduced by a keyword and terminated by a semicolon. In the middle there may be more keywords as well as object names and values.

Store new Data with INSERT Command

When storing new data in rows of a table we must name all affected objects and values: the table name (there may be a lot of tables within the database), the columnnames and the values. All this is embedded within some keywords so that the SQL compiler can recognise the tokens and their meaning. In general the syntax for a simple INSERT is

```
INSERT INTO <tablename> (<list_of_columnnames>)
VALUES                  (<list_of_values>);
```

Here is an example

```
-- put one row
INSERT INTO person (id, firstname, lastname,    date_of_birth,     place_of_birth, ssn,           weight)
VALUES             (1,  'Larry',   'Goldstein', date'1970-11-20', 'Dallas',       '078-05-1120', 95);
-- confirm the INSERT command
COMMIT;
```

When the DBMS recognises the keywords INSERT INTO and VALUES it knows what to do: it creates a new row in the table and puts the given values into the named columns. In the above example the command is followed by a second one: COMMIT confirms the INSERT operation as well as the other writing operations UPDATE and DELETE. (We will learn much more about COMMIT and its counterpart ROLLBACK in a later chapter.)

> A short comment about the format of the value for *date_of_birth*: There is no unique format for dates honored all over the world. Peoples use different formats depending on their cultural habit. For our purpose we decide to represent dates in the hierarchical format defined in ISO 8601. It may be possible that your local database installation use a different format so that you are forced to either modify our examples or to modify the default date format of your database installation.

Now we will put some more rows into our table. To do so we use a variation of the above syntax. It is possible to omit the list of columnnames if the list of values correlates exactly with the number, order and data type of the columns used in the original CREATE TABLE statement.

> Hint: The practice of omitting the list of columnnames is not recommended for real applications! Table structures change over time, e.g. someone may add new columns to the table. In this case unexpected side effects may occur in applications.

```
-- put four rows
INSERT INTO person VALUES (2,  'Tom',    'Burton',    date'1980-01-22', 'Birmingham',    '078-05-1121', 75);
INSERT INTO person VALUES (3,  'Lisa',   'Hamilton',  date'1975-12-30', 'Mumbai',        '078-05-1122', 56);
INSERT INTO person VALUES (4,  'Debora', 'Patterson', date'2011-06-01', 'Shanghai',      '078-05-1123', 11);
INSERT INTO person VALUES (5,  'James',  'de Winter', date'1975-12-23', 'San Francisco', '078-05-1124', 75);
COMMIT;
```

Retrieve Data with SELECT Command

Now our table should contain five rows. Can we be sure about that? How can we check whether everything worked well and the rows and values exist really? To do so, we need a command which shows us the actual content of the table. It is the SELECT command with the following general syntax

```
SELECT   <list_of_columnnames>
FROM     <tablename>
```

```
WHERE    <search_condition>
ORDER BY <order_by_clause>;
```

As with the INSERT command you may omit some parts. The simplest example is

```
SELECT   *
FROM     person;
```

The asterik character '*' indicates 'all columns'. In the result, the DBMS should deliver all five rows each with the seven values we used previously with the INSERT command.

In the following examples we add the actually missing clauses of the general syntax - one after the other.

Add a list of some or all columnnames

```
SELECT   firstname, lastname
FROM     person;
```

The DBMS should deliver the two columns *firstname* and *lastname* of all five rows.

Add a search condition

```
SELECT   id, firstname, lastname
FROM     person
WHERE    id > 2;
```

The DBMS should deliver the three columns *id*, *firstname* and *lastname* of three rows.

Add a sort instruction

```
SELECT   id, firstname, lastname, date_of_birth
FROM     person
WHERE    id > 2
ORDER BY date_of_birth;
```

The DBMS should deliver the four columns *id*, *firstname*, *lastname* and *date_of_birth* of three rows in the ascending order of *date_of_birth*.

Modify Data with UPDATE Command

If we want to change the values of some columns in some rows we can do so by using the UPDATE command. The general syntax for a simple UPDATE is:

```
UPDATE <tablename>
SET    <columnname> = <value>,
       <columnname> = <value>,
                ...
WHERE  <search_condition>;
```

Values are assigned to the named columns. Unmentioned columns keep unchanged. The search_condition acts in the same way as in the SELECT command. It restricts the coverage of the command to rows, which satisfy the criteria. If the WHERE keyword and the search_condition are omitted, **all** rows of the table are affected. It is possible to specify search_conditions, which hit no rows. In this case no rows are updated - and no error or exception occurs.

Change one column of one row

```
UPDATE person
SET    firstname = 'James Walker'
WHERE  id = 5;
COMMIT;
```

The first name of Mr. de Winter changes to James Walker whereas all his other values keep unchanged. Also all other rows keep unchanged. Please verify this with a SELECT command.

Change one column of multiple rows

```
UPDATE person
SET    firstname = 'Unknown'
WHERE  date_of_birth < date'2000-01-01';
COMMIT;
```

The <search_condition> isn't restricted to the Primary Key column. We can specify any other column. And the comparison operator isn't restricted to the equal sign. We can use other operators - they solely have to match the data type of the column.

In this example we change the *firstname* of four rows with a single command. If there is a table with millions of rows we can change all of them using one single command.

Change two columns of one row

```
-- Please note the additional comma
UPDATE person
SET    firstname = 'Jimmy Walker',
       lastname  = 'de la Crux'
WHERE  id = 5;
COMMIT;
```

The two values are changed with one single command.

Remove data with DELETE Command

The DELETE command removes complete rows from the table. As the rows are removed as a whole there is no need to specify any columnname. The semantics of the <search_condition> is the same as with SELECT and UPDATE.

```
DELETE
FROM   <tablename>
WHERE  <search_condition>;
```

Delete one row

```
DELETE
FROM   person
WHERE  id = 5;
COMMIT;
```

The row of James de Winter is removed from the table.

Delete many rows

```
DELETE
FROM   person;
COMMIT;
```

All remained rows are deleted as we have omitted the <search_condition>. The table is empty, but it still exists.

No rows affected

```
DELETE
FROM   person
WHERE  id = 99;
COMMIT;
```

This command will remove no row as there is no row with *id* equals to 99. But the syntax and the execution within the DBMS are still perfect. No exception is thrown. The command terminates without any error message or error code.

Summary

The INSERT and DELETE commands affect rows in their entirety. INSERT puts a complete new row into a table (unmentioned columns remain empty) and DELETE removes complete rows. In contrast, SELECT and UPDATE affect only those columns that are mentioned in the command; unmentioned columns are unaffected.

The INSERT command (in the simple version of this page) has no <search_condition> and therefore handles exactly one row. The three other commands may affect zero, one or more rows depending on the evaluation of their <search_condition>.

First of all a database is a collection of data. These data are organized in tables as shown in the example *person*. In addition, there are many other kinds of objects in the DBMS: views, functions, procedures, indices, rights and many others. Initially we focus on tables and present four of them. They serve as the foundation for our Wikibook. Other kind of objects will be presented later.

We try to keep everything as simple as possible. Nevertheless this minimalistic set of four tables demonstrates a 1:n as well as a n:m relationship.

person

The *person* table holds information about fictitious persons; see: Create a simple Table.

```
-- comment lines starts with two consecutive minus signs '--'
CREATE TABLE person (
  -- define columns (name / type / default value / nullable)
  id             DECIMAL      NOT NULL,
  firstname      VARCHAR(50)  NOT NULL,
  lastname       VARCHAR(50)  NOT NULL,
  date_of_birth  DATE,
  place_of_birth VARCHAR(50),
  ssn            CHAR(11),
  weight         DECIMAL DEFAULT 0 NOT NULL,
  -- select one of the defined columns as the Primary Key and
  -- guess a meaningfull name for the Primary Key constraint: 'person_pk' may be a good choice
  CONSTRAINT person_pk PRIMARY KEY (id)
);
```

contact

The *contact* table holds information about the contact data of some persons. One could consider to store this contact information in additional columns of the *person* table: one column for email, one for icq, and so on. We decided against it for some serious reasons.

- Missing values: A lot of people do not have most of those contact values respectively we doen't know the values. Hereinafter the table will look like a sparse matrix.
- Multiplicities: Other people have more than one email adress or multiple phone numbers. Shall we define a lot of columns email_1, email_2, ... ? What is the upper limit? Standard SQL does not offer something like an 'array of values' for columns (some implementations do).
- Future Extentions: Some day there will be one or more contact types which are unknown today. Then we have to modify the table.

We can deal with all this situations in an uncomplicated way, when the contact data goes to its own table. The only special thing is bringing persons together with their contact data. This task will be managed by the column *person_id* of table *contact*. It holds the same value as the Primary Key of the allocated person.

The general statement is, that we do have **one** information unit (person) to which **potentially multiple** information units of same type (contact) belongs to. We call this togetherness a **relationship** - in this case a **1:m relationship**. Whenever we encounter such a situation, we store the values, which may occure more than once, in a separate table together with the id of the first table.

```
CREATE TABLE contact (
  -- define columns (name / type / default value / nullable)
  id             DECIMAL      NOT NULL,
  person_id      DECIMAL      NOT NULL,
  -- use a default value, if contact_type is omitted
  contact_type   VARCHAR(25)  DEFAULT 'email' NOT NULL,
  contact_value  VARCHAR(50)  NOT NULL,
  -- select one of the defined columns as the Primary Key
  CONSTRAINT contact_pk PRIMARY KEY (id),
  -- define Foreign Key relation between column person_id and column id of table person
  CONSTRAINT contact_fk FOREIGN KEY (person_id) REFERENCES person(id),
  -- more contraint(s)
  CONSTRAINT contact_check CHECK (contact_type IN ('fixed line', 'mobile', 'email', 'icq', 'skype'))
);
```

hobby

People usually pursue one or more hobbies. Concerning multiplicity we have the same problems as before with *contact*. So we need a separate table for hobbies.

```
CREATE TABLE hobby (
```

```
  -- define columns (name / type / default value / nullable)
  id              DECIMAL      NOT NULL,
  hobbyname       VARCHAR(100) NOT NULL,
  remark          VARCHAR(1000),
  -- select one of the defined columns as the Primary Key
  CONSTRAINT hobby_pk PRIMARY KEY (id),
  -- forbid duplicate recording of a hobby
  CONSTRAINT hobby_unique UNIQUE (hobbyname)
);
```

You may have noticed, that there is no column for the corresponding person. Why this? With hobbies we have an additional problem: It's not just that one person pursues multiple hobbies. At the same time multiple persons pursue the same hobby.

We call this kind of togetherness a **n:m relationship**. It can be designed by creating a third table between the two original tables. The third table holds the id's of the first and second table. So one can decide which person persues which hobby. In our example this 'table-in-the-middle' is *person_hobby* and will be defined next.

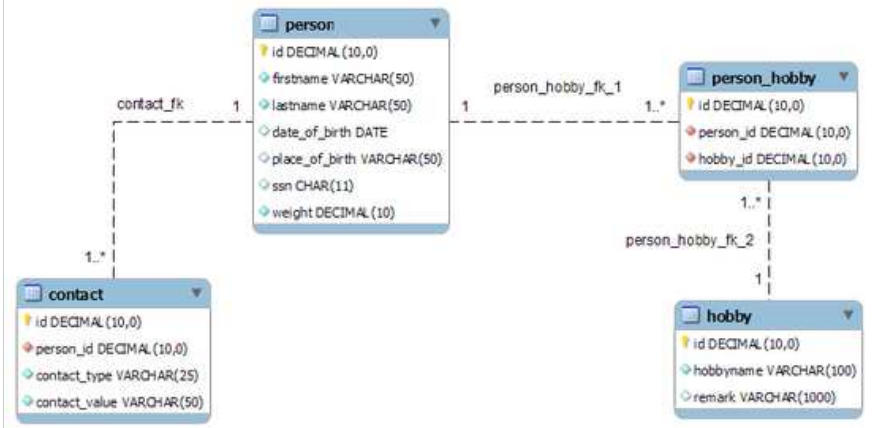
person_hobby

```
CREATE TABLE person_hobby (
  -- define columns (name / type / default value / nullable)
  id              DECIMAL      NOT NULL,
  person_id       DECIMAL      NOT NULL,
  hobby_id        DECIMAL      NOT NULL,
  -- Also this table has its own Primary Key!
  CONSTRAINT person_hobby_pk PRIMARY KEY (id),
  -- define Foreign Key relation between column person_id and column id of table person
  CONSTRAINT person_hobby_fk_1 FOREIGN KEY (person_id) REFERENCES person(id),
  -- define Foreign Key relation between column hobby_id and column id of table hobby
  CONSTRAINT person_hobby_fk_2 FOREIGN KEY (hobby_id) REFERENCES hobby(id)
);
```

Every row of the table holds one id from *person* and one from *hobby*. This is the technic how the information of persons and hobbies are joined together.

Visualisation of the Structure

After execution of the above commands your database should contain four tables (without any data). The tables and their relationship to each other may be visualised in a so called Entity Relationship Diagram. On the left side there is the 1:n relationship between *person* and *contact* and on the right side the n:m relationship between *person* and *hobby* with its 'table-in-the-middle' *person_hobby*.



Visual Representation of our Example Database

rDBMS offers different ways to put data into their storage: from csv files, Excel files, product specific binary files, via several API's or special gateways to other databases respectively database systems and some more technics. So there is a wide range of - non standardised - possibilities to bring data into our system. Because we are speaking about SQL we use the standardised INSERT command to do the job. It is available on all systems.

We use only a small amount of data because we want to keep things simple. Sometimes one needs a great number of rows to do performance tests. For this purpose we show a special INSERT command at the end of this page, which inflates your table in an exponential fashion.

person

```
--
-- After we have done a lot of tests we may want to reset the data to its original version.
-- To do so use the DELETE command. But be aware of Foreign Keys: you may be forced to delete
-- persons at the very end - with DELETE it's just the opposite sequence of tables in comparition to INSERTs.
-- Be careful and don't confuse DELETE with DROP !!
--
-- DELETE FROM person_hobby;
-- DELETE FROM hobby;
-- DELETE FROM contact;
-- DELETE FROM person;
```

```
-- COMMIT;

INSERT INTO person VALUES (1,  'Larry',  'Goldstein', DATE'1970-11-20', 'Dallas',        '078-05-1120', 95);
INSERT INTO person VALUES (2,  'Tom',    'Burton',    DATE'1977-01-22', 'Birmingham',    '078-05-1121', 75);
INSERT INTO person VALUES (3,  'Lisa',   'Hamilton',  DATE'1975-12-23', 'Richland',      '078-05-1122', 56);
INSERT INTO person VALUES (4,  'Kim',    'Goldstein', DATE'2011-06-01', 'Shanghai',      '078-05-1123', 11);
INSERT INTO person VALUES (5,  'James',  'de Winter', DATE'1975-12-23', 'San Francisco', '078-05-1124', 75);
INSERT INTO person VALUES (6,  'Elias',  'Baker',     DATE'1939-10-03', 'San Francisco', '078-05-1125', 55);
INSERT INTO person VALUES (7,  'Yorgos', 'Stefanos',  DATE'1975-12-23', 'Athens',        '078-05-1126', 64);
INSERT INTO person VALUES (8,  'John',   'de Winter', DATE'1977-01-22', 'San Francisco', '078-05-1127', 77);
INSERT INTO person VALUES (9,  'Richie', 'Rich',      DATE'1975-12-23', 'Richland',      '078-05-1128', 90);
INSERT INTO person VALUES (10, 'Victor', 'de Winter', DATE'1979-02-28', 'San Francisco', '078-05-1129', 78);
COMMIT;
```

contact

```
-- DELETE FROM contact;
-- COMMIT;

INSERT INTO contact VALUES (1,  1,  'fixed line', '555-0100');
INSERT INTO contact VALUES (2,  1,  'email',      'larry.goldstein@acme.xx');
INSERT INTO contact VALUES (3,  1,  'email',      'lg@my_company.xx');
INSERT INTO contact VALUES (4,  1,  'icq',        '12111');
INSERT INTO contact VALUES (5,  4,  'fixed line', '5550101');
INSERT INTO contact VALUES (6,  4,  'mobile',     '10123444444');
INSERT INTO contact VALUES (7,  5,  'email',      'james.dewinter@acme.xx');
INSERT INTO contact VALUES (8,  7,  'fixed line', '+30000000000000');
INSERT INTO contact VALUES (9,  7,  'mobile',     '+30695100000000');
COMMIT;
```

hobby

```
-- DELETE FROM hobby;
-- COMMIT;

INSERT INTO hobby VALUES (1,  'Painting',
                          'Applying paint, pigment, color or other medium to a surface.');
INSERT INTO hobby VALUES (2,  'Fishing',
                          'Catching fishes.');
INSERT INTO hobby VALUES (3,  'Underwater Diving',
                          'Going underwater with or without breathing apparatus (scuba diving / breath-holding).');
INSERT INTO hobby VALUES (4,  'Chess',
                          'Two players have 16 figures each. They move them on an eight-by-eight grid according to special rules
INSERT INTO hobby VALUES (5,  'Literature', 'Reading books.');
INSERT INTO hobby VALUES (6,  'Yoga',
                          'A physical, mental, and spiritual practices which originated in ancient India.');
INSERT INTO hobby VALUES (7,  'Stamp collecting',
                          'Collecting of post stamps and related objects.');
INSERT INTO hobby VALUES (8,  'Astronomy',
                          'Observing astronomical objects such as moons, planets, stars, nebulae, and galaxies.');
INSERT INTO hobby VALUES (9,  'Microscopy',
                          'Observing very small objects using a microscope.');
COMMIT;
```

person_hobby

```
-- DELETE FROM person_hobby;
-- COMMIT;

INSERT INTO person_hobby VALUES (1, 1, 1);
INSERT INTO person_hobby VALUES (2, 1, 4);
INSERT INTO person_hobby VALUES (3, 1, 5);
INSERT INTO person_hobby VALUES (4, 5, 2);
INSERT INTO person_hobby VALUES (5, 5, 3);
INSERT INTO person_hobby VALUES (6, 7, 8);
INSERT INTO person_hobby VALUES (7, 4, 4);
INSERT INTO person_hobby VALUES (8, 9, 8);
INSERT INTO person_hobby VALUES (9, 9, 9);
COMMIT;
```

Grow up

For realistic performance tests we need a huge amount of data. The few number of rows in our example database does not meet this criteria. How can we generate test data and store it in a table? There are different possibilities: FOR loops in a procedure, (pseudo-) recursive calls, importing external data in a system specific fashion and some more.

Because we are dealing with SQL we introduce an INSERT command which is portable across all rDBMS. Although it has a simple syntax it is very powerful. With every execution it will double the number of rows. Suppose there is 1 row in a table. After the first execution there will be a second row in the table. At first glance this sounds boring. But after 10 executions there are more than thousand rows, after 20 executions there are more than a million, and we suspect that only few installations are able to execute it more than 30 times.

```
INSERT INTO person (id,                              firstname, lastname, weight)
SELECT              id + (select max(id) from person), firstname, lastname, weight
FROM        person;
COMMIT;
```

The command is an INSERT in combination with a (Sub-)SELECT. The SELECT retrievs all rows of the table because there is no WHERE clause. This is the reason for the doubling. The mandatory columns *firstname* and *lastname* keeps unchanged. We ignore optional columns. Only the primary key *id* is computed. The new value is the sum of the old value plus the highest available *id* when starting the command.

Some more remarks:

- max(id) is determined only once per execution! This illustrates an important aspect of rDBMS: At a conceptual level the database has a certain state before execution of a command and a new state after its execution. Commands are **atomic** operations moving the database from one state to another - they run completely or not a bit! Both, the SELECT and the inner SELECT with the max(id), act on the initial state. They never see the result or an intermediate result of the INSERT. Otherwise the INSERT would never end.
- If we wish to observe the process of growing, we can add a column to the table to store max(id) with each iteration.
- The computation of the new *id* may be omitted if the DBMS supports AUTO INCREMENT columns.
- For performance tests it may be helpful to store some random data in one or more columns.

Basic Commands

The SELECT command retrieves data from one or more tables or views. It generally consists of the following language elements:

```
SELECT   <things_to_be_displayed>  -- the so called 'Projection' - mostly a list of columnnames
FROM     <tablename>               -- table or view names and their aliases
WHERE    <where_clause>            -- the so called 'Restriction' or 'search condition'
GROUP BY <group_by_clause>
HAVING   <having_clause>
ORDER BY <order_by_clause>;
```

With the exception of the first two elements all others are optional. The sequence of language elements is mandatory. At certain places within the command there may start new SELECT commands - in a recursive manner.

Projection

In the projection part of the SELECT command you specify a list of columns, operations working on columns, functions, fixed values or new SELECT commands.

```
-- C/Java style comments are possible within SQL commands
SELECT id,                          /* the name of a column   */
       concat(firstname, lastname), /* the concat() function  */
       weight + 5,                  /* the add operation      */
       'kg'                         /* a value                */
FROM   person;
```

The DBMS will retrieve ten rows, each of them consists of four columns.

We can mix the sequence of columns in any order or retrieve them several times.

```
SELECT id, lastname, lastname, 'weighs', weight, 'kg'
FROM   person;
```

The asterix '*' is an abbreviation for the list of all columns.

```
SELECT * FROM person;
```

For numeric columns we can apply the usual numeric operators +, -, * and /. There are also a lot of predefined functions depending on the data type: power, sqrt, modulo, string functions, date functions.

UNIQUE

It is possible to compact results in the sense of UNIQUE values. In this case all resulting **rows**, which would be identical without the UNIQUE keyword, will be compressed to one **row**. In other words: duplicates are eliminated - just like in set theory.

```
-- retrieves 10 rows
SELECT lastname
FROM   person;
-- retrieves only 7 rows. Duplicate values are thrown away.
SELECT DISTINCT lastname
FROM   person;
-- Hint: The term 'DISTINCT' refers to the complete resulting row, which you can imagine as the
--       aggregation of ALL columns of the projection. The keyword DISTINCT must follow directly behind the SELECT keyword.
--       The following query leads to 10 rows although three persons have the same lastname.
SELECT DISTINCT lastname, firstname
FROM   person;
-- 7 rows again
SELECT DISTINCT lastname, lastname
FROM   person;
```

Aliases for Columnnames

Sometimes we want to give resulting columns more expressive names. We can do so by choosing an alias within the projection. This alias is the new name within the resultset. GUIs use to show it as the column label.

```
-- The keyword 'AS' is optional
SELECT lastname as family_name, weight weight_in_kg
FROM   person;
```

Functions

There are predefined functions for use in projections (and at some other positions). The most frequently used are:

- count(<columnname>|'*'): Counts the number of resulting rows.
- max(<columnname>): The highest value in <column> of the resultset. Also applicable on strings.
- min(<columnname>): The lowest value in <column> of the resultset. Also applicable on strings.
- sum(<columnname>): The sum of all values in a numeric column.
- avg(<columnname>): The average of a numeric column.
- concat(<columnname_1>, <columnname_2>): The concatenation of two columns. Alternatively the function may be expressed by the '||' operator: <columnname_1> || <columnname_2>

Standard SQL and every DBMS offers much more functions.

We must differ between those functions which return one value per row like concat() and those which return only one row per complete resultset like max(). The former one may be mixed in any combination with column names as shown in the very first example of this page. With the later ones there exists a problem: If we mix them with a normal column name, the DBMS recognise a contradiction in the query. On the one hand it should retrieve exactly one value (in one row) and on the other hand it should retrieve a lot of values (in a lot of rows). The reaction of DBMS differ from vendor to vendor. Some throw an error message at runtime, others deliver suspicious results.

```
-- works fine
SELECT lastname, concat(weight, ' kg')
FROM   person;
-- check the reaction of your DBMS
SELECT lastname, avg(weight)
FROM   person;
```

```
-- a legal mixture of functions resulting in one row with 4 columns
SELECT min(weight), max(weight), avg(weight) as average_1, sum(weight) / count(*) as average_2
FROM   person;
```

SELECT within SELECT

If we **really** want to see the result of a resultset-oriented-function in combination with columns of more than one row, we can start a very new SELECT on a location where - in simple cases - a columname occurs. This second SELECT is an absolutely independent command.

Be careful: It will be executed for **every** resulting row of the first SELECT!

```
-- retrieves 10 rows; notice the additional parenthesis to delimit the two SELECTs from each other.
SELECT lastname, (SELECT avg(weight) from person)
FROM   person;
-- Compute the percentage of each persons weight in relation to the average weight of all persons
SELECT lastname, weight, weight * 100 / (SELECT avg(weight) from person) as percentage_of_average
FROM   person;
```

Table names

Behind the Keyword FROM we have to announce the name of the table on which the command shall work. Thereby the table name is well known and may be used as an identifier. In the first simple examples the use of an additional identifier seems to be needless. Later on it will turn into a necessary feature to formulate complex commands.

```
SELECT person.firstname, person.lastname
FROM   person;
-- Define an alias for the table name (analog to column names). To retain overview we usually
-- abbreviate tables by the first character of their name.
SELECT p.firstname, p.lastname
FROM   person AS p;  -- Hint: not all systems accept keyword 'AS' with table aliases. Omit it in this cases!
-- The keyword 'AS' is optional again.
SELECT p.firstname, p.lastname
FROM   person p;
```

Restriction

In the WHERE clause we specify some 'search conditions' which are among the named table(s) or view(s). The evaluation of this criteria is - mostly - one of the first things during the execution of a SELECT command. Before any row can be sorted or displayed, she must meet the conditions in the clause.

If we omit the clause all rows of the table are retrieved. Else the number of rows will be reduced according to the specified criteria. If we specify 'weight < 70', for example, only those rows are retrieved where the weight column stores a value less than 70. It is such that restrictions act on **rows** of **tables** by evaluation **column values** (sometime they act on other things like the existence of rows, but for the moment we focus on basic principles). As a result, we can imagine that the evaluation of the 'where clause' produces a list of rows. This list of rows will be processed in further steps like sorting, grouping or displaying certain columns (projection).

Comparisons

We compare variables, constant values and results of function calls with each other in the same way as we would do in other programming languages. The only difference is, that we use column names instead of variables. The comparison operators must match the given data types they have to operate on. The result of the comparison is a boolean value. If it is 'true' the according row will be processed furthermore. Some examples:

- 'weight = 70' compares the column 'weight' with the constant value '70' whether the column is equal to the constant value.
- '70 = weight': same as before.
- 'firstname = lastname' compares two columns - each of the **same** row - for equality. Names like 'Frederic Frederic' evaluate to true.
- 'firstname < lastname' is a legal comparison of two columns according to the lexical order of strings.
- 'LENGTH(firstname) < 5' compares the result of a function call to the constant value '5'. The function LENGTH() operates on strings and returns a number.

Boolean logic

Often we want to specify more than a single search criteria, e.g.: Are there people born in San Francisco with lastname Baker? To do this, we specify every necessary comparison independent from the next one and join them together with the boolean operators AND respectively OR.

```
SELECT *
FROM   person
WHERE  place_of_birth = 'San Francisco'
AND    lastname = 'Baker';
```

The result of a comparison is a boolean. It may be toggled between 'true' and 'false' by the unary operator NOT.

```
SELECT  *
FROM    person
WHERE   place_of_birth = 'San Francisco'
AND NOT lastname = 'Baker'; -- all except 'Baker'
-- for clarification: The NOT in the foregoing example is an 'unary operation' on the result of the
--                    comparison. It's not an addition to the AND.
SELECT  *
FROM    person
WHERE   place_of_birth = 'San Francisco'
AND (NOT (lastname = 'Baker'));   --  same as before, but explicit notated with parenthesis
```

The **precedence** of comparisons and boolean logic is as follows:

1. all comparisons
2. NOT operator
3. AND operator
4. OR operator

```
-- AND (born in SF and lastname Baker; 1 hit as an intermediate result) will be processed before
-- OR  (person Yorgos; 1 hit)
-- 1 + 1 ==> 2 rows
SELECT *
FROM   person
WHERE  place_of_birth = 'San Francisco' -- 4 hits SF
AND    lastname = 'Baker'               -- 1 hit Baker
OR     firstname = 'Yorgos'             -- 1 hit Yorgos
;

-- AND (person Yorgos Baker; no hit as an intermediate result) will be processed before
-- OR  (born in SF; 4 hits)
-- 0 + 4 ==> 4 rows
SELECT *
FROM   person
WHERE  place_of_birth = 'San Francisco' -- 4 hits SF
OR     firstname = 'Yorgos'             -- 1 hit Yorgos
AND    lastname = 'Baker'               -- 1 hit Baker
;

-- We can modify the sequence of evaluations by specifying parentheses.
-- Same as first example, adding parentheses, one row.
SELECT *
FROM   person
WHERE  place_of_birth = 'San Francisco' -- 4 hits SF
AND   (lastname = 'Baker'               -- 1 hit Baker
OR     firstname = 'Yorgos')            -- 1 hit Yorgos
;
```

Two abbreviations

Sometimes we shorten the syntax by using the BETWEEN keyword. It defines a lower and upper limit and is used mainly for numeric and date values, but also applicable to strings.

```
SELECT *
FROM   person
WHERE  weight >= 70
AND    weight <= 90;
-- An equivalent shorter and more expressive wording
SELECT *
FROM   person
WHERE  weight BETWEEN 70 AND 90; -- BETWEEN includes the two cutting edges
```

For the comparison of a column or function with a number of values we can use the short IN expression.

```
SELECT *
FROM   person
WHERE  lastname = 'de Winter'
OR     lastname = 'Baker';
-- An equivalent shorter and more expressive wording
SELECT *
FROM   person
WHERE  lastname IN ('de Winter', 'Baker');
```

Grouping

We will offer the GROUP BY clause in combination with the HAVING clause in a later chapter.

Sorting

The DBMS is free to deliver the resulting rows in an arbitrary order. Rows may be returned in the order of the Primary Key, in the chronological order they are stored into the database, in the order of an B-tree organised internal key, or even in a random order. Concerning the sequence of delivered rows the DBMS may do what it wants to do. Don't expect anything.

If we expect a certain order of rows, we must express our wishes explicitly. We can do this in the ORDER BY clause. There we specify a list of columnnames in combination with an option for ascending respectively descending sorting.

```
-- all persons in ascending (which is the default) order of their weight
SELECT *
FROM   person
ORDER BY weight;
-- all persons in descending order of their weight
SELECT *
FROM   person
ORDER BY weight desc;
```

In the above result there are two rows with identical values in the column *weight*. As this situation leads to random results, we have the possibility to specify more columns. These following columns are processed only for those rows with identical values in all preceding columns.

```
-- All persons in descending order of their weight. In ambiguous cases order the
-- additional column place_of_birth ascending: Birmingham before San Francisco.
SELECT *
FROM   person
ORDER BY weight desc, place_of_birth;
```

In the ORDER BY clause we can specify any column of the processed table. We are not limited to the ones which are returned by the projection.

```
-- same ordering as above
SELECT firstname, lastname
FROM   person
ORDER BY weight desc, place_of_birth;
```

Combine the Language Elements

Only the first two elements of the SELECT command are mandatory: the part up to the first table (or view) name. All others are optional. If we specify also the optional ones, their predetermined sequence must be kept in mind. But they are combinable according to our needs.

```
-- We have seen on this page: SELECT / FROM / WHERE / ORDER BY
SELECT p.lastname,
       p.weight,
       p.weight * 100 / (SELECT avg(p2.weight) FROM person p2) AS percentage_of_average
FROM   person p
WHERE  p.weight BETWEEN 70 AND 90
ORDER BY p.weight desc, p.place_of_birth;
```

Further Information

There are more information about additional opportunities of the SELECT command.

- Join Operation
- Grouping
- Set Operations
- Like Predicate
- Predefined Functions

Exercises

Show hobbyname and remark from the hobby table.

Click to see solution

```
SELECT hobbyname, remark
```

```
FROM   hobby;
```

Show hobbyname and remark from the hobby table. Order the result by hobbyname.

Click to see solution

```
SELECT hobbyname, remark
FROM   hobby
ORDER BY hobbyname;
```

Show hobbyname and remark from the hobby table. Choose 'Hobby' as first columnname and 'Short_Description_of_Hobby' as second columnname.

Click to see solution

```
SELECT hobbyname as Hobby, remark as Short_Description_of_Hobby
FROM   hobby;
-- columnname without underscore: Use quotes
SELECT hobbyname as Hobby, remark as "Short Description of Hobby"
FROM   hobby;
```

Show firstname and lastname of persons born in San Francisco.

Click to see solution

```
SELECT firstname, lastname
FROM   person
WHERE  place_of_birth = 'San Francisco';
```

Show all information items of persons with lastname 'de Winter'.

Click to see solution

```
SELECT *
FROM   person
WHERE  lastname = 'de Winter';
```

How many rows are stored in the contact table?

Click to see solution

```
SELECT count(*)
FROM   contact;
9
```

How many E-Mails are stored in the contact table?

Click to see solution

```
SELECT count(*)
FROM   contact
WHERE  contact_type = 'email';
3
```

What is the mean weight of persons born in San Francisco?

Click to see solution

```
SELECT avg(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco';
71.25
```

Find persons born after 1979-12-31, which weigh more than / less than 50 kg.

Click to see solution

```
SELECT *
FROM   person
WHERE  date_of_birth > DATE '1979-12-31'
AND    weight > 50;
--
SELECT *
FROM   person
```

```
WHERE  date_of_birth > DATE '1979-12-31'
AND    weight < 50;
```

Find persons born in Birmingham, Mumbai, Shanghai or Athens in the order of their firstname.

Click to see solution

```
SELECT *
FROM   person
WHERE  place_of_birth = 'Birmingham'
OR     place_of_birth = 'Mumbai'
OR     place_of_birth = 'Shanghai'
OR     place_of_birth = 'Athens'
ORDER BY firstname;
-- equivalent:
SELECT *
FROM   person
WHERE  place_of_birth IN ('Birmingham', 'Mumbai', 'Shanghai', 'Athens')
ORDER BY firstname;
```

Find persons born in Birmingham, Mumbai, Shanghai or Athens within the 21. century.

Click to see solution

```
SELECT *
FROM   person
WHERE  (   place_of_birth = 'Birmingham'
        OR place_of_birth = 'Mumbai'
        OR place_of_birth = 'Shanghai'
        OR place_of_birth = 'Athens'
       )
AND    date_of_birth >= DATE '2000-01-01';
-- equivalent:
SELECT *
FROM   person
WHERE  place_of_birth IN ('Birmingham', 'Mumbai', 'Shanghai', 'Athens')
AND    date_of_birth >= DATE '2000-01-01';
```

Find persons born between Dallas and Richland ('between' not in the sense of a geographic area but of the lexical order of citynames)

Click to see solution

```
-- strings have a lexical order. So we can use some operators known
-- from numeric data types.
SELECT *
FROM   person
WHERE  place_of_birth >= 'Dallas'
AND    place_of_birth <= 'Richland'
ORDER BY place_of_birth;
-- equivalent:
SELECT *
FROM   person
WHERE  place_of_birth BETWEEN 'Dallas' AND 'Richland'
ORDER BY place_of_birth;
```

Which kind of contacts are stored in the contact table? (Only one row per value.)

Click to see solution

```
SELECT DISTINCT contact_type
FROM   contact;
fixed line
email
icq
mobile
```

How many different kind of contacts are stored in the contact table? (Hint: Count the rows of above query.)

Click to see solution

```
SELECT count(DISTINCT contact_type)
FROM   contact;
4
```

Show contact_type, contact_value and a string of the form 'total number of contacts: <x>', where <x> is the quantity of all existing contacts.

Click to see solution

```
SELECT contact_type, contact_value,
       (SELECT concat('total number of contacts: ', count(*)) FROM contact)
```

```
FROM   contact;
-- Some systems need explicit type casting from numeric to string
SELECT contact_type, contact_value,
       (SELECT concat('total number of contacts: ', cast(count(*) as char)) FROM contact)
FROM   contact;
-- The '||' operator is some kind of 'syntactical sugar'. It's an abbreviation for the concat() function.
-- The operator is part of the SQL standard, but not implemented by all vendors.
SELECT contact_type, contact_value,
       (SELECT 'total number of contacts: ' || count(*) FROM contact)
FROM   contact;
```

DBMS offers a special service. We can **undo** a single or even multiple consecutive write and delete operations. To do so we use the command ROLLBACK. When modifying data, the DBMS writes in a first step all new, changed or deleted data to a temporary space. During this stage the modified data is not part of the 'regular' database. If we are sure the modifications shall apply, we use the COMMIT command. If we want to revert our changes, we use the ROLLBACK command. All changes up to the finally COMMIT or ROLLBACK are considered to be part of a so called **transaction**.

The syntax of COMMIT and ROLLBACK is very simple.

```
COMMIT WORK;     -- commits all previous INSERT, UPDATE and DELETE commands, which
                 -- occurred since last COMMIT or ROLLBACK
ROLLBACK WORK;   -- reverts all previous INSERT, UPDATE and DELETE commands, which
                 -- occurred since last COMMIT or ROLLBACK
```

The keyword 'WORK' is optional.

AUTOCOMMIT

The feature AUTOCOMMIT automatically performs a COMMIT after every write operation (INSERT, UPDATE or DELETE). This feature is not part of the SQL standard, but is implemented and activated by default in some implementations. If we want to use the ROLLBACK command, we must deactivate the AUTOCOMMIT. (After an - automatic or explicit - COMMIT command a ROLLBACK command is syntactically okay, but it does nothing as everything is already committed.) Often we can deactivate the AUTOCOMMIT with a separate command like 'SET autocommit = 0;' or 'SET autocommit off;' or by clicking an icon on a GUI.

To test the following statements it is necessary to work without AUTOCOMMIT.

COMMIT

Let us insert a new person into the database and test the COMMIT.

```
-- Store a new person with id 99.
INSERT INTO person (id, firstname, lastname, date_of_birth,    place_of_birth, ssn,           weight)
VALUES             (99, 'Harriet', 'Flint',  DATE'1970-10-19', 'Dallas',       '078-05-1120', 65);

-- Is the new person really in the database? The process which executes the write operation will see its results,
-- even if they are actually not committed. (One hit expected.)
SELECT *
FROM   person
WHERE  id = 99;

-- Try COMMIT command
COMMIT;

-- Is she still in the database? (One hit expected.)
SELECT *
FROM   person
WHERE  id = 99;
```

Now we remove the person from the database.

```
-- Remove the new person
DELETE
FROM   person
WHERE  id = 99;

-- Is the person really gone? Again, the process which performs the write operation will see the changes, even
-- if they are actually not committed. (No hit expected.)
SELECT *
FROM   person
WHERE  id = 99;

-- Try COMMIT command
COMMIT;

-- Is the person still in the database? (No hit expected.)
SELECT *
FROM   person
```

```
WHERE  id = 99;
```

So far, so boring.

ROLLBACK

The exciting command is the ROLLBACK. It restores changes of previous INSERT, UPDATE or DELETE commands.

We delete and restore Mrs. Hamilton from our example database.

```
DELETE
FROM   person
WHERE  id = 3; -- Lisa Hamilton

-- no hit expected
SELECT *
FROM   person
WHERE  id = 3;

-- ROLLBACK restores the deletion
ROLLBACK;

-- ONE hit expected !!! Else: check AUTOCOMMIT
SELECT *
FROM   person
WHERE  id = 3;
```

The ROLLBACK is not restricted to one single row. It may affect several rows, several commands, different kind of commands and even several tables.

```
-- same as above
DELETE
FROM   person
WHERE  id = 3;
-- destroy all e-mail addresses
UPDATE contact
SET    contact_value = 'unknown'
WHERE  contact_type = 'email';

-- verify modifications
SELECT * FROM person;
SELECT * FROM contact;

-- A single ROLLBACK command restores the deletion in one table and the modifications in another table
ROLLBACK;

-- verify ROLLBACK
SELECT * FROM person;
SELECT * FROM contact;
```

Exercises

Supose the *hobby* table contains 9 rows and the *person* table 10 rows. We execute the following operations:
add 3 hobbies
add 4 persons
commit
add 5 hobbies
add 6 persons
rollback

How many rows are in the hobby table?

Click to see solution

```
12
```

How many rows are in the person table?

Click to see solution

```
14
```

Structured Query Language/INSERT

Structured Query Language/UPDATE Structured Query Language/DELETE

Daily Operations

Data should be stored in such a way that no redundant information exists in the database. For example, if our database includes groups of people who, in each case, all pursue the same hobby, then we would rather avoid repeatedly storing the same static details about a given hobby; namely in every record about one of the hobby's enthusiasts. Likewise, we would rather avoid repeatedly storing the same detailed information about an individual hobbyist; namely in every record about one of that person's hobbies. Instead we create independent *person* and *hobby* tables and point from one to the other. This technique for grouping data in separate, redundancy-free tables is called database normalization. Such separation also tends to simplify the logic and enhance the flexibility of assembling precisely the items needed for a given purpose. This assembly is accomplished by means of the 'JOIN' operation.

The Idea

In our example database, there are two tables: *person* and *contact*. The *contact* table contains the column *person_id*, which correlates with the Primary-Key column *id* of the *person* table. By evaluating the column values we can join contacts and persons together.

person table P

ID	LASTNAME	FIRSTNAME	...
1	Goldstein	Larry	...
2	Burton	Tom	...
3	Hamilton	Lisa	...
4	Goldstein	Kim	...
...	...	...	...
...	...	...	...

contact table C

ID	PERSON_ID	CONTACT_TYPE	CONTACT_VALUE
1	1	fixed line	555-0100
2	1	email	larry.goldstein@acme.xx
3	1	email	lg@my_company.xx
4	1	icq	12111
5	4	fixed line	5550101
6	4	mobile	10123444444
...	...	...	...
...	...	...	...

Joined (virtual) table, created out of *person* and *contact*

P.ID	P.LASTNAME	P.FIRSTNAME	...	C.ID	C.PERSON_ID	C.CONTACT_TYPE	C.CONTACT_VALUE
1	Goldstein	Larry	...	1	1	fixed line	555-0100
1	Goldstein	Larry	...	2	1	email	larry.goldstein@acme.xx
1	Goldstein	Larry	...	3	1	email	lg@my_company.xx
1	Goldstein	Larry	...	4	1	icq	12111
2	Burton	Tom	...	?	?	?	?
3	Hamilton	Lisa	...	?	?	?	?
4	Goldstein	Kim	...	5	4	fixed line	5550101
4	Goldstein	Kim	...	6	4	mobile	10123444444
...	...	...	...	...	...	...	...

So, Larry Goldstein that exists only once in the stored *person* table, is now listed four times in the joined, virtual table – each time, in combination with one of his four contact items. The same applies for Kim Goldstein and his two contact items.

But what is going on with Tom Burton and Lisa Hamilton, whose contact information is not available? We may have some trouble attempting to join their *person* data with their non-existent *contact* information. For the moment, we have flagged the situation with question marks. A detailed explanation of how to transform the problem into a solution appears later on this page.

The Basic Syntax

Obviously it's necessary to specify two things with the JOIN operation

- the names of the relevant tables
- the names of the relevant columns

The basic syntax extends the SELECT command with these two elements

```
SELECT <things_to_be_displayed>       -- as usual
FROM   <tablename_1> <table_1_alias>  -- a table alias
JOIN   <tablename_2> <table_2_alias> ON <join condition> -- the join criterion
...    -- optionally all the other elements of SELECT command
;
```

Let's make a first attempt.

```
SELECT *
FROM   person p
JOIN   contact c ON p.id = c.person_id;
```

One of the table names is referenced after the FROM keyword (as previously), and the other one after the new keyword, JOIN, which (no surprise here) instructs the DBMS to perform a join operation. Next, the ON keyword introduces the column names together with a comparison operator (or a general condition, as you will see later). The column names are prefixed with the respective aliases of the table names, *p* and *c*. This is necessary, due to the fact that columns with identical names (like *id*) may exist in multiple tables.

When the DBMS executes the command, it delivers 'something' that contains all the columns from both tables, including the two *id* columns from their respective (*person* and *contact*) tables. The result contains 9 rows, one per **existing** combination of person and contact; viz., due to the 'ON' expression, person records without any corresponding contact records will not appear in the result.

The delivered 'something' looks like a new table; in fact, it has the same structure, behaviour and data as a table. If it is created from a view or as the result of a subselection, we can even perform new SELECTs on it. But there is one important difference between this and a table: Its assembled data is **not stored** in the DBMS as such; rather, the data is **computed** at run time from the values of real tables, and only held in temporary memory while the DBMS is running your program.

This key feature – assembling complex information from simple tables – is made possible by means of the two simple keywords, JOIN and ON. As you will see also, the syntax can be extended to build very complex queries, such that you can add many additional refinements to the specification of your join criteria.

It can sometimes be confusing when results don't match your intentions. If this happens, try to simplify your query, as shown here. Confusion often results from the fact that the JOIN syntax itself may become quite complex. Moreover, joining can be combined with all of the other syntactic elements of the SELECT command, which also may lead to lack of clarity.

The combination of the join syntax with other language elements is shown in the following examples.

```
--
-- show only important columns
SELECT p.firstname, p.lastname, c.contact_type as "Kind of Contact", c.contact_value as "Call Number"
FROM   person p
JOIN   contact c ON p.id = c.person_id;

-- show only desired rows
SELECT p.firstname, p.lastname, c.contact_type as "Kind of Contact", c.contact_value as "Call Number"
FROM   person p
JOIN   contact c ON p.id = c.person_id
WHERE  c.contact_type IN ('fixed line', 'mobile');

-- apply any sort order
SELECT p.firstname, p.lastname, c.contact_type as "Kind of Contact", c.contact_value as "Call Number"
FROM   person p
JOIN   contact c ON p.id = c.person_id
WHERE  c.contact_type IN ('fixed line', 'mobile')
ORDER BY p.lastname, p.firstname, c.contact_type DESC;

-- use functions: min() / max() / count()
SELECT count(*)
FROM   person p
JOIN   contact c ON p.id = c.person_id
WHERE  c.contact_type IN ('fixed line', 'mobile');

-- JOIN a table with itself. Example: Search different persons with same lastname
SELECT p1.id, p1.firstname, p1.lastname, p2.id, p2.firstname, p2.lastname
FROM   person p1
JOIN   person p2 ON p1.lastname = p2.lastname -- for second incarnation of person we must use a different alias
WHERE  p1.id != p2.id
-- sorting of p2.lastname is not necessary as it is identical to the already sorted p1.lastname
ORDER BY p1.lastname, p1.firstname, p2.firstname;

-- JOIN more than two tables. Example: contact information of different persons with same lastname
SELECT p1.id, p1.firstname, p1.lastname, p2.id, p2.firstname, p2.lastname, c.contact_type, c.contact_value
FROM   person p1
JOIN   person p2 ON p1.lastname = p2.lastname
JOIN   contact c ON p2.id = c.person_id       -- contact info from person2. p1.id would lead to person1
WHERE  p1.id != p2.id
ORDER BY p1.lastname, p1.firstname, p2.lastname;
```

Four Join Types

Earlier on this page, we saw an example of a join result wherein some rows contained person names, but no contact information – instead showing a question mark in that latter column. If the basic syntax of the JOIN operation had been used, those (question-mark) rows would have been filtered out. That (basic syntax with exclusive result) is known as an INNER join. There are also three different kinds of OUTER joins. The results of an OUTER join will contain not only all the full-data rows that an INNER join's results would, but also partial-data rows, i.e., those where no data was found in one or both of the two stored tables; thus, they're called LEFT OUTER, RIGHT OUTER and FULL OUTER joins.

So we can widen the basic JOIN syntax to the four options:

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN

Keywords surrounded by [ ] are optional. The parser infers OUTER from LEFT, RIGHT or FULL, and a plain (i.e, basic-syntax) JOIN defaults to INNER.

Inner Join

The inner join is probably the most commonly used of the four types. As we have seen, it results in exactly those rows that exactly match the criterion following the ON. Below is an example showing how to create a list of persons and their contacts.

```
-- A list of persons and their contacts
SELECT p.firstname, p.lastname, c.contact_type, c.contact_value
FROM   person p
JOIN   contact c ON p.id = c.person_id  -- identical meaning: INNER JOIN ...
ORDER BY p.lastname, p.firstname, c.contact_type DESC, c.contact_value;
```

What is most significant is that records for persons without any contact information are **not** part of the result.

Left (outer) Join

Sometimes we need a little more; for example, we might want a list of all person records, to include any contact-information records that may also be available for that person. Note how this differs from the example above: this time, the results will contain *all* person records, even those for persons who have **no** contact-information record(s).

```
-- A list of ALL persons plus their contacts
SELECT    p.firstname, p.lastname, c.contact_type, c.contact_value
FROM      person p
LEFT JOIN contact c ON p.id = c.person_id  -- identical meaning: LEFT OUTER JOIN ...
ORDER BY  p.lastname, p.firstname, c.contact_type DESC, c.contact_value;
```

In those cases where the contact information is unavailable, the DBMS will supplant it with the 'null value' or with the 'null special marker' (not to be confused with the *string* (-type) 'null value' or 'null' nor with binary 0. Nonetheless, implemention details aren't important here. The null special marker will be discussed in a later chapter).

In summary, the left (outer) join is an inner join, plus one row for each left-side match without a counterpart on the right side.

Consider the word 'left'. It refers to the left side of the formula, "FROM <table_1> LEFT JOIN <table_2>", or more specifically, the table denoted on the left side (here: *table_1*); indicating that every row of that table will be represented at least once in the result, whether a corresponding record is found in the right-side table (here: *table_2*) or not.

Another example:

```
SELECT    p.firstname, p.lastname, c.contact_type, c.contact_value
FROM      contact c
LEFT JOIN person p  ON p.id = c.person_id  -- identical meaning: LEFT OUTER JOIN ...
ORDER BY  p.lastname, p.firstname, c.contact_type DESC, c.contact_value;
```

What's the difference? We've changed the order of the table names. Note that we're still using a LEFT join, but because *contact* is now the "left" referent (the object in the FROM clause), *contact* data will now be considered as being of primary importance; therefore, all the contact rows will appear in the result - along with any corresponding information that may exist in the person table. As it happens, in the database we're using, every contact record corresponds to a person record so, in this case, it works out that the results are equivalent to what they'd have been if we'd used an inner join. Yet they're different from those of the previous left-join example.

Right (outer) Join

The right join obeys the same rules as the left join, but in reverse. Now, every record from the table referenced in the join clause will appear in the result, including those that have no corresponding record in the other table. Again, the DBMS supplies each empty right-column cell with the null special marker. The only difference is that the evaluation sequence of tables is carried out in reverse or, in other words, with the roles of the two tables swapped.

```
-- A list of ALL contact records with any corresponding person data, even if s
SELECT     p.firstname, p.lastname, c.contact_type, c.contact_value
FROM       person p
RIGHT JOIN contact c ON p.id = c.person_id  -- same as RIGHT OUTER JOIN ...
ORDER BY   p.lastname, p.firstname, c.contact_type DESC, c.contact_value;
```

Full (outer) Join

A full join retrieves every row of both the left table and the right table, regardless of whether a corresponding record exists in the respective opposite table.

```
SELECT    p.firstname, p.lastname, c.contact_type, c.contact_value
FROM      person p
FULL JOIN contact c ON p.id = c.person_id  -- identical meaning: FULL OUTER JOIN ...
ORDER BY  p.lastname, p.firstname, c.contact_type DESC, c.contact_value;
```

Given *table_1* and *table_2* below,

table_1

ID	X
1	11
2	12
3	13

table_2

ID	TABLE_1_ID	Y
1	1	21
2	5	22

the full join:

```
SELECT    *
FROM      table_1 t1
FULL JOIN table_2 t2 ON t1.id = t2.table_1_id;
```

will yield:

T1.ID	T1.X	T2.ID	T2.TABLE_1_ID	T2.Y
1	11	1	1	21
2	12	null	null	null
3	13	null	null	null
null	null	2	5	22

These results contain the (single) matching row, plus a row each for all the other records of both of the original tables. As each of these other rows represent data found in only one of the tables, they are each missing some data, so the cells representative of that missing data contain the null special marker.

Note: The full join is not supported by all DBMS. Nevertheless, because it isn't an atomic operation, it is always possible to create the desired result by a combination of multiple SELECTs with SET operations.

Cartesian Product

With inner joins, it is possible to omit the ON. SQL interprets this as a (syntactically correct) request to combine every record from the left table with every record from the right table. It will return a large number of rows; namely, the product of the respective record counts of both tables.

This special kind of an inner join is called a Cartesian product. The Cartesian product is an elementary operation of relational algebra, which is the foundation for all rDBMS implementations.

```
-- all persons combined with all contacts (some implementations replace the
-- keyword 'JOIN' with a comma)
SELECT p.firstname, p.lastname, c.contact_type, c.contact_value
FROM   person p
JOIN   contact c   -- missing ON keyword: p X c will be created
ORDER BY p.lastname, p.firstname, c.contact_type DESC, c.contact_value;

-- count the resulting rows
SELECT count(*)
FROM   person p
JOIN   contact c;
```

Be careful then; if you unintentionally omit the ON term, the result will be much larger than expected. If, for example, the first table contains 10,000 records, and the second one 20,000 records, the output will contain 200 million rows.

The n:m Situation

How can we create a list of persons and their hobbies? Remember: one person may run many hobbies and several persons may run the same hobby. So there is no direct connection from persons to hobbies. Between the two tables we have created a third one *person_hobby*. It holds the id of persons as well as the id of hobbies.

We have to 'walk' from *person* to *person_hobby* and from there to *hobby*.

```
-- persons combined with their hobbies
SELECT p.id p_id, p.firstname, p.lastname, h.hobbyname, h.id h_id
FROM   person       p
JOIN   person_hobby ph ON p.id = ph.person_id
JOIN   hobby        h  ON ph.hobby_id = h.id
ORDER BY p.lastname, p.firstname, h.hobbyname;
```

Please notice that no column of the table *person_hobby* goes to the result. This table acts only during intermediate execution steps. Even its column *id* is not of interest.

Some people do not perform a hobby. As we performed an INNER JOIN they are not part of the above list. If we want to see in the list also persons without hobbies, we must do what we have done before: use LEFT OUTER JOINs instead of INNER JOINs.

```
-- ALL persons plus their hobbies (if present)
SELECT p.id p_id, p.firstname, p.lastname, h.hobbyname, h.id h_id
FROM   person           p
LEFT JOIN   person_hobby ph ON p.id = ph.person_id
LEFT JOIN   hobby        h  ON ph.hobby_id = h.id
ORDER BY p.lastname, p.firstname, h.hobbyname;
```

Hint: If necessary we can combine every kind of join with every other kind of join in every desired sequence, eg: LEFT OUTER with FULL OUTER with INNER ... .

More Details

Criteria for join operations are not restricted to the usual formulation:

```
SELECT ...
FROM   table_1 t1
JOIN   table_2 t2 ON t1.id = t2.fk
...
```

First, we can use **any column**, not only primary key and foreign key columns. In one of the above examples we used the lastname for a join. Lastname is of type character and has no meaning of any key. To avoid poor performance some DBMS restrict the use of columns to those having an index.

Second, the comparator is not restricted to the **equal sign**. We can use any sensfull operator, for example the 'greater than' for numeric values.

```
-- Which person has the greater body weight - restricted to 'de Winter' for clarity
SELECT p1.id, p1.firstname as "is heavier", p1.weight, p2.id, p2.firstname as "than", p2.weight
FROM   person p1
JOIN   person p2 ON p1.weight > p2.weight
WHERE  p1.lastname = 'de Winter'
AND    p2.lastname = 'de Winter'
ORDER BY p1.weight desc, p2.weight desc;
```

Third, we can use an **arbitrary function**.

```
-- short lastnames vs. long lastnames
SELECT p1.firstname, p1.lastname as "shorter lastname", p2.firstname, p2.lastname
FROM   person p1
JOIN   person p2 ON LENGTH(p1.lastname) < LENGTH(p2.lastname)
-- likewise ORDER BY can use functions
ORDER BY length(p1.lastname), length(p2.lastname);
```

Exercises

Show first- and lastname plus icq number for persons having an icq number

Click to see solution

```
SELECT p.id, p.firstname, p.lastname, c.contact_value
FROM   person p
JOIN   contact c ON p.id = c.person_id
WHERE  c.contact_type = 'icq';
```

Show first- and lastname plus ICQ number plus fixed line number for persons having an ICQ number AND a fixed line. You need to join the *contact* table twice.

Click to see solution

```
SELECT p.id, p.firstname, p.lastname,
       c1.contact_value as icq,
       c2.contact_value as "fixed line"   -- looks like previous, but is different
FROM   person p
JOIN   contact c1 ON p.id = c1.person_id
JOIN   contact c2 ON p.id = c2.person_id  -- it's a second (virtual) incarnation of contact table
WHERE  c1.contact_type = 'icq'            -- from first incarnation
AND    c2.contact_type = 'fixed line';    -- from second incarnation

-- In this example of an INNER JOIN we can convert the WHERE part to an additional JOIN criterion.
-- This may clarify the intention of the command. But be careful: This shifting in combination with
-- one of the OUTER JOINs may lead to different results.
SELECT p.id, p.firstname, p.lastname, c1.contact_value as icq, c2.contact_value as "fixed line"
FROM   person p
JOIN   contact c1 ON p.id = c1.person_id AND c1.contact_type = 'icq'
JOIN   contact c2 ON p.id = c2.person_id AND c2.contact_type = 'fixed line';
```

Show first- and lastname plus (if present) the ICQ number for ALL persons

Click to see solution

```
-- To retrieve ALL persons it's necessary to use a LEFT join.
-- But the first approach is not what we expect! In this example the LEFT JOIN is evaluated first
-- and creates an intermediate table with null-values in contact_type (eliminate the
-- WHERE clause to see this intermediate result). These rows and all other except the
-- one with 'ICQ' are then thrown away by evaluating the WHERE clause.
SELECT    p.id, p.firstname, p.lastname, c.contact_value
FROM      person p
LEFT JOIN contact c ON p.id = c.person_id
WHERE     c.contact_type = 'icq';
-- It's necessary to formulate the search criterion as part of the JOIN. Unlike with
-- the INNER JOIN in the previous example with (LEFT/FULL/RIGHT) OUTER JOINs it is not possible
-- to shift it to the WHERE clause.
SELECT    p.id, p.firstname, p.lastname, c.contact_value
FROM      person p
LEFT JOIN contact c ON p.id = c.person_id AND c.contact_type = 'icq';
```

Create a list which contains ALL hobbies plus according persons (if present)

Click to see solution

```
SELECT p.id p_id, p.firstname, p.lastname, h.hobbyname, h.id h_id
FROM   person              p
RIGHT  JOIN   person_hobby ph ON p.id = ph.person_id
RIGHT  JOIN   hobby        h  ON ph.hobby_id = h.id
ORDER BY h.hobbyname, p.lastname, p.firstname;
```

Is it possible that one of the three outer joins contains fewer rows than the corresponding inner join?

Click to see solution

```
No.

All four join types contain the same rows with column-matching-values. In addition
outer joins contain rows where column values do not match - if such a situation exists.
```

In this chapter we will leave the level of individual rows. We strive to find informations and statements that refer to **groups of rows** - at the expense of information about individual rows. In the context of SQL such 'row-groups' (or sets of rows) are build by the GROUP BY clause and further processed by the HAVING clause.

Constitute Groups

First we must establish criteria according to which the rows are assigned to groups. To do so we use the content of one or more columns of the involved table(s). If the values are identical, the rows belong to the same group. Consider the *lastname* in table *person*. In our small example we can insinuate that persons with same lastname form a family. So if we strive for informations about families we should use this column as the grouping criterion. This grouping allows us to ask questions concerning whole families, such as 'Which families are there?', 'How many families exists?', 'How many persons are in each family?'. Please note that all of them are questions about the whole group (which means the family), not about single rows (which means the person).

In the SQL syntax the criterion is specified after the key word GROUP BY and consists of one or more columnnames.

```
SELECT ...              -- as usual
FROM   ...              -- as usual (optionally plus JOINs)
GROUP BY <columnname>   -- optionally more columnnames
...                     -- optionally other elements of SELECT command
;
```

Our concrete example about families looks like this:

```
SELECT lastname
FROM   person
GROUP BY lastname;
```

The query retrieves seven 'family names' out of the 10 rows. There are several persons with lastname 'Goldstein' or 'de Winter'.

We can retrieve the same seven 'family names' by applying the key word DISTINCT in a SELECT without GROUP BY.

```
SELECT DISTINCT lastname
FROM   person;
-- no GROUP BY clause
```

What makes the difference? The DISTINCT key word is limited to remove duplicate values. It can not initiate computations on other rows and columns of the result set. In contrast, the GROUP BY additionally arranges the intermediate received rows as a number of groups and offers the possibility to get informations about each of these groups. It is even the case that within these groups **all** columns are available, not only the 'criterion'-column. To confirm this statement about 'all' columns we use *weight* which is not the 'criterion'-column.

```
SELECT lastname, avg(weight)  -- avg() is a function to compute the arithmetic mean of numerical values
FROM   person
GROUP BY lastname;
```

The result shows the seven family names - as seen before - plus the average weight of every family. The weight of individual persons is not shown. (In groups with exactly one person the average weight of the group is of course identical to the single persons weight.)

Grouping over multiple columns

If necessary we can define the grouping over more than one column. In this case we can imagine the concatenation of the colums as the grouping rule.

```
-- Group over one column: place_of_birth leads to 6 resulting rows
SELECT place_of_birth, count(*)
FROM   person
GROUP BY place_of_birth;
-- Group over two columns: place_of_birth plus lastname leads to 8 resulting rows with Richland and SF shown twice
SELECT place_of_birth, lastname, count(*)
FROM   person
GROUP BY place_of_birth, lastname;
```

Inspect Groups

After we have defined groups with the GROUP BY key word, we can select more informations about each of them, e.g.: how much persons (rows) exist within each family (group of rows)?

```
SELECT lastname, count(*)  -- count() is a function which counts values or rows
FROM   person
GROUP BY lastname;
```

We see that in our small example database there is one family with 3 members, another with 2 members and all others consist of exactly 1 member.

What is going on behind the scene during the execution of the command?

1. All ten rows of table *person* are retrieved (in the above command there is no WHERE clause).
2. The rows are arranged into seven groups according to the value of column *lastname*.
3. Every group with all of its rows is passed to the SELECT clause.
4. The SELECT builds one resulting row for every received group (in 'real world' databases each of the groups may contain thousands of rows).

In step 4 **exactly one** resulting row is generated per group. Because the SELECT creates only one resulting row per group, it is not possible to show values of such columns which may differ from row to row, e.g. the firstname. The SELECT can only show such values of which it is ensured that they are identical within all rows of the group: the 'criterion'-column.

```
-- It is not possible to show the 'firstname' of a group! 'firstname' is an attribute of single person.
-- Within a group 'firstname' varies from row to row.
-- The DBMS should recognise this problem and should issue an error message.
SELECT lastname, firstname
FROM   person
GROUP BY lastname;
-- A hint to users of MySQL:
-- To receive correct results (the error message) you must deactivate a special performance feature by issuing the command
-- set sql_mode = 'ONLY_FULL_GROUP_BY'; or set it in the workbench or in the ini-file.
```

Nevertheless we can get information about the non-criterion-columns. But this information is more generalized. The DBMS offers a special group of functions which builds one value out of a set of rows. Consider the avg() function, which computes the arithmetic mean of numerical values. This function receives a column name and operates on a set of rows. If our command in question contains a GROUP BY clause, the avg() function does compute one value per group - not one value per all rows as usual. So it is possible to show the result of such functions together with the values of the 'criterion'-column.

Here is an - incomplete - list of such functions: count(), max(), min(), sum(), avg(). Not all functions are of that kind, e.g. the function concat(), which concatenates two strings, operates on single rows and creates one value per row.

```
-- compute avg() by your own formula
SELECT lastname, sum(weight) / count(weight) as "Mean weight 1", avg(weight) as "Mean weight 2"
FROM   person
GROUP BY lastname;
```

Focus on Desired Groups

You know the WHERE clause. It defines which rows of a table will be part of the result set. The HAVING clause has the same meaning at the group-level. It defines which groups will be part of the result set.

```
-- The HAVING complements the GROUP BY
SELECT ...
FROM   ...
GROUP BY <columnname>
HAVING <having clause>; -- specify a criterion which can be applied to groups
```

We retrieve exclusively families with more than 1 members:

```
SELECT lastname
FROM   person
GROUP BY lastname    -- grouping over lastname
HAVING count(*) > 1; -- more than one person within the group
```

All families with one member are no longer part of the result.

In a second example we focus on such groups which satifies a criterion on column *firstname*. Consider that *firstname* is not the grouping-column.

```
-- Groups containing a person whose firstname has more than 4 characters: 5 resulting rows
SELECT lastname
```

```
FROM   person
GROUP BY lastname
HAVING max(length(firstname)) > 4; -- max() returns ONE value (the highest one) for all rows of each 'lastname'-group
```

The result shows the 5 families Baker, de Winter, Goldstein, Rich and Stefanos (, but not the row(s) with the long *firstname*).

Please note that this result is very different from the similar question to persons whose firstname has more than 4 characters:

```
-- Persons whose firstname has more than 4 characters: 6 resulting rows!!
SELECT lastname, firstname
FROM   person
WHERE  length(firstname) > 4;
-- no GROUP BY and no HAVING. The WHERE isn't an equivalent replacement for the HAVING!!
```

Where is the additional row coming from? In the family de Winter there are two persons with a firstname longer than 4 characters: James and Victor. Because in the command without GROUP BY we select for persons and not for families, both rows are displayed individually.

In summary we can say that the HAVING clause decides, which groups are part of the result set and which are not.

The Overall Picture

The GROUP BY and HAVING clauses are part of the SELECT comand and we can combine them with any other clauses of the SELECT as desired. Only the order of the clauses is obligatory.

```
-- This is the obligatory order of clauses
SELECT ...
FROM   ...
WHERE  ...
GROUP BY ...
HAVING   ...
ORDER BY ...
;
```

As mentioned the WHERE clause works on the row-level whereas the HAVING clause works on the group-level. First the WHERE is evaluated, next the GROUP BY, next the HAVING, next the ORDER BY and at the end the SELECT. Every step is based on the results of the previous one.

Finally we offer two additional examples:

```
-- Are there persons born on the same day?
SELECT date_of_birth  -- In a later chapter you will learn how to select the name of this persons.
FROM   person
GROUP BY date_of_birth
HAVING count(date_of_birth) > 1 -- more than one on the same day?
ORDER BY date_of_birth;

-- Families with long first- and lastname. Comment out some lines to see differences to the original query.
SELECT lastname, count(*) as cnt
FROM   person
WHERE  length(firstname) > 4
GROUP BY lastname
HAVING length(lastname) > 4
ORDER BY cnt desc, lastname
;
```

Exercises

Are there persons born on the same day in the same city? Hint: group over both criteria

Click to see solution

```
SELECT date_of_birth, place_of_birth
FROM   person
GROUP BY date_of_birth, place_of_birth
HAVING   count(*) > 1;
```

Categorise persons according to the formula: 'round (weight / 10)': 10 to 19 kg --> 1, 20 to 29 kg --> 2, ...
How much persons exist in each category?

Click to see solution

```
SELECT round (weight / 10), count(*)
FROM   person
GROUP BY round (weight / 10)
-- ORDER BY round (weight / 10)  -- order by category
ORDER BY count(*)                -- order by frequency
```

```
;
```

Which contact type is used in which frequency in table contact?

Click to see solution

```
SELECT contact_type, count(*)
FROM   contact
GROUP BY contact_type
-- ORDER BY contact_type  -- order by contact_type
ORDER BY count(*)         -- order by frequency
;
```

Restrict previous result to contact types which occurs more than once.

Click to see solution

```
SELECT contact_type, count(*)
FROM   contact
GROUP BY contact_type
HAVING   count(*) > 1
-- order by contact_type  -- order by contact_type
ORDER BY count(*)         -- order by frequency
;
```

Are there persons performing more than 2 hobies? Hint: check table person_hobby.

Click to see solution

```
SELECT person_id, count(*)
FROM   person_hobby
GROUP BY person_id
HAVING   count(*) > 2
;
```

Are there persons performing only one hobby?

Click to see solution

```
SELECT person_id, count(*)
FROM   person_hobby
GROUP BY person_id
HAVING   count(*) = 1
;
```

Are there persons performing no hobby?

Click to see solution

```
There are persons, which do not perform a hobby. But the nearby formulation 'count(*) = 0'
will not lead to the expected result because for such persons there are no rows
in table person_contact, so the DBMS cannot create any group and hence cannot display anything.

Looking for something that does NOT exist is often more difficult than looking for the
existence of something. In such cases you usually have to use one of: NOT EXISTS, NOT IN,
a combination of OUTER JOIN and IS NULL, a combination of OUTER JOIN and MINUS together
with INNER JOIN.
```

When creating new rows it may occur that we don't know the value of one or more columns.

Let's assume that we want to store informations about banking accounts and for one of those accounts we don't know the balance. What can we do? There are several possibilities:

- **Reject** the whole row with all other informations like account number, dispositional credit, interest rate, ... Not very attractive.
- Store a **default value** instead of the value we acually don't know. But there are cases where it is impossible to define a default value because every value is possible, e.g. a bank account of '0' or '-1' is not unusual.
- Store a **flag** that signals that no value is stored. This approach is similar to the *Not-a-Number* technique.

Relational DBMS uses the last mentioned technique and the sense of the flag is 'there is no value stored'. Sometimes people say 'The **NULL** value is stored' or 'The **NULL** special marker is stored'.

Extention of Boolean Logic

Assume there is a table for banking accounts and some of its rows hold the NULL special marker in the column *balance*. Does those rows fullfill at least one of the two WHERE conditions 'balance >= 0' or 'balance <= 0'? No. It is not possible to decide whether these conditions are true or false! Honestly we must admit that we don't know an answer in our usual true/false logic because we don't know a value for *balance*. We are forced to extend the range of boolean values with a third one, which we call **unkown**. The two conditions above evaluate neither true nor false, both evaluate to 'unknown' for rows where *balance* holds the NULL special marker.

In a later stage we need definitions for the boolean operators NOT, AND, OR and EQUAL when true/false interact with unknown. You find the definitions here.

Retrieve the NULL Special Marker

Within every SELECT command such rows become part of the resulting rows, in which the WHERE condition evaluates to true. If it evaluates to false or unknown, the row will be rejected. As **all** WHERE conditions like the above 'balance >= 0' - and also their negation - evaluates to unknown for missing *balance* values, there is preliminary no way to retrieve them.

To overcome this lack, SQL contains the special phrase 'IS NULL'. The wording 'balance IS NULL' evaluates to true for exactly the rows with a missing value in *balance*.

```
SELECT ...
FROM   ...
WHERE  <columnname> IS NULL
...
;
```

We must use exactly this wording. The use of any arithmetic operator like >, <=, !=, ... will not retrieve rows with the NULL special marker. The same holds true even for the condition '(balance = 0) OR NOT (balance = 0)', which is a tautology in conventional true/false logic. Beside this IS NULL predicate there is no other way to retrieve the NULL special marker - without one simple but not helpfull exception: if you omit the WHERE condition, all rows of the table are retrieved, with and without NULL special marker in any column.

That's all! Dealing with NULL special marker and the 3-value-logic might sound strange if you first met this topic. But as the IS NULL predicate evaluates always to true or false everything works as usual afterwards. We can use all other elements of the SELECT command (boolean logic, join, having, order by, ...) in the same way we have done so far.

Some Examples

Our test database does not contain the NULL special marker. Nevertheless we have met the situation during the explanation of OUTER joins. OUTER joins create resulting rows where some columns contain the NULL special marker. We must consider this possibility, if we deal with the results of such subselects.

There are two other ways to generate the NULL special marker.

- INSERT or UPDATE command with the explicit notion of the NULL special marker. In this case the SQL key word **null** is used as a representative for the NULL special marker.
- INSERT command without using all columns. The omitted columns will get the NULL special marker - or a default, if one is defined.

To demonstrate this and to create some examples for the following excercises, we put one row into the *person* table with some columns left empty.

```
-- Insert a new row for testing purpose
INSERT INTO person (id, firstname, lastname)  VALUES (51, 'Half man', 'Uncomplete');
COMMIT;

-- Retrieve the row. As defined in CREATE TABLE statement the weight has a default value of integer 0.
-- Date_of_birth and place_of_birth contain the NULL special marker.
SELECT * FROM person WHERE  id = 51;

-- use the IS NULL predicate within WHERE clause. The result contains 1 row.
SELECT * FROM person WHERE ssn IS NULL;

-- weight has a value!! We expect to retrieve no rows when we use the IS NULL predicate.
SELECT * FROM person WHERE weight IS NULL;
-- or, to say it the other way round, the number of rows is 0
SELECT count(*) FROM person WHERE weight IS NULL;
-- but in the next statement the number of rows is 1
SELECT count(*) FROM person WHERE weight = 0;

-- Negate the IS NULL predicate
SELECT count(*) FROM person WHERE ssn IS NULL;     -- IS NULL
SELECT count(*) FROM person WHERE ssn IS NOT NULL; -- Negation of IS NULL

SELECT count(*)
FROM   person
WHERE  ssn IS NULL
OR     ssn IS NOT NULL; -- A tautology, which always retrieves ALL rows of a table
-- Same as above
```

```
SELECT count(*)
FROM   person
WHERE  ssn IS NULL
OR NOT ssn IS NULL; -- A tautology, which always retrieves ALL rows of a table
```

Next we show the use of the UPDATE command in combination with the key word NULL

```
--
-- Insert a new row for testing purpose with all columns filled with a usefull value
INSERT INTO person (id, firstname, lastname,  date_of_birth,    place_of_birth, ssn,           weight)
VALUES             (52, 'Lyn',     'Mutable', DATE'1951-05-13', 'Anchorage',    '078-05-1152', 69);
COMMIT;
SELECT * FROM person WHERE id = 52;

-- Delete a single column value (not the complete row)
UPDATE person SET ssn = null WHERE id = 52;
COMMIT;
SELECT * FROM person WHERE id = 52;      -- one row
SELECT * FROM person WHERE ssn IS NULL;  -- two rows: 51 + 52
```

Restore the original state of the example database.

```
DELETE FROM person WHERE id > 50;
COMMIT;
```

Coalesce() and Similar Functions

In the context of the NULL special marker it is often the case that we have to retrieve rows with no value (the NULL special marker) **or** a default value such as 0 or blank. In such cases, the WHERE condition looks something like this: "... WHERE (col IS NULL OR col = 0) ...". To keep source code simpler, the SQL standard defines a function **coalesce(<expression_1>, <expression_2>)**. If the first argument, which normaly is the name of a column, is not NULL, the function evaluates to this argument - else to the second argument.

Example:

```
-- Retrieve rows without ssn or with ssn equal to blank.
SELECT *
FROM   person
WHERE  coalesce(ssn, ' ') = ' ';
-- equivalent:
-- WHERE (ssn IS NULL
-- OR     ssn = ' ');
```

The function name *coalesce* results from the fact that the function accepts an arbitrary number of parameters and evaluates them in a recursive manner. If parameter *n* results in a real value, it evaluates to this parameter, else the function calls itself without the n-th parameter. coalesce(expression_1, expression_2, expression_3) evaluates to expression_1, if expression_1 is not NULL, else to expression_2, if expression_2 is not NULL, else to expression_3.

The SQL standard defines another function **nullif(<expression_1>, <expression_2>)**. It evaluates to NULL, if the two expressions are equal - and it evaluates to the first expression, if they differ from each other.

Different vendors offers some more functions like **isnull()**, **ifnull()** or **nvl()** to support handling of NULL values. The meaning of this functions is vendor specific.

Exercises

Insert a new hobby 'Snowshoeing' without a remark.

Click to see solution

```
INSERT INTO hobby (id, hobbyname,     remark)
VALUES            (10, 'Snowshoeing', null);
COMMIT;
```

Find a second solution for the above question without using the key word 'null'. (First delete row 10.)

Click to see solution

```
DELETE FROM hobby WHERE id = 10;
INSERT INTO hobby (id, hobbyname)
VALUES            (10, 'Snowshoeing');
COMMIT;
```

Retrieve all hobbies without a remark.

Click to see solution

```
-- 1 row
SELECT * FROM hobby WHERE remark IS NULL;
```

How many hobbies are exemplified with a remark?

Click to see solution

```
-- 9 rows
SELECT count(*) FROM hobby WHERE remark IS NOT NULL;
```

Change row 10 of hobby in the way that the hobbname contains the string 'NULL' and the remark 'Name of hobby not known'.

Click to see solution

```
-- Consider the two apostrophes surrounding the string 'NULL', which consists of the 4 characters N, U, L and L !!
UPDATE hobby SET hobbyname = 'NULL', remark = 'Name of hobby not known' WHERE id = 10;
COMMIT;
```

a) Retrieve the row where hobbyname is 'NULL'.
b) Retrieve the row where remark is 'Name of hobby not known'.

Click to see solution

```
-- This may be a pitfall question. There is no relation to the IS NULL predicate
SELECT * FROM hobby WHERE hobbyname = 'NULL';
SELECT * FROM hobby WHERE remark = 'Name of hobby not known';
```

How many hobbies have a hobby name?

Click to see solution

```
-- All 10 rows contains a hobby name, even the row with the hobbyname 'NULL'
SELECT count(*) FROM hobby WHERE hobbyname IS NOT NULL;
```

There are two groups of predefined functions:

- **aggregate functions**. They work on a set of rows, which means they receive one value for each row of a set of rows and returns one value for the whole set. If they are called in the context of a GROUP BY clause, they are called once per group, else once for all rows.
- **scalar functions**. They work on single rows, which means they receive one value of a single row and returns one value for each of them.

Aggregate functions

They work on a set of rows and return one single value like the number of rows, the highest or lowest value, the standard deviation, etc. The most important aggregate functions are:

Signatur	Semantic
`COUNT(*)`	The number of rows
`COUNT(<column name>)`	The number of rows where <column name> contains a value (IS NOT NULL). The elimination of rows with the NULL special marker in the considered column applies to all aggregate functions.
`MIN(<column name>)`	Lowest value. In the case of strings according to the sequence of characters.
`MAX(<column name>)`	Highest value. In the case of strings according to the sequence of characters.
`SUM(<column name>)`	Sum of all values
`AVG(<column name>)`	Arithmetic mean

As an example we retrieve the maximum weight of all persons:

```
SELECT MAX(weight)
FROM   person;
```

A Word of Caution
Aggregate functions result in one value for a set of rows. Therefore it is not possible to use them together with 'normal' columns in the projection (the part behind SELECT key word). If we specify, for example,

```
SELECT lastname, SUM(weight)
FROM   person;
```

we try to instruct the DBMS to show a **lot of rows** containing the *lastname* simultaneously with **one** value. This is a contradiction and the system will throw an exception. We can use a lot of aggregate functions within one projection but we are not allowed to use them together with 'normal' columns.

```
-- Multiple aggregate functions. No 'normal' columns.
SELECT SUM(weight)/COUNT(weight) as average_1, AVG(weight) as average_2
FROM   person;
```

Grouping
If we use aggregate functions in the context of commands containing a GROUP BY, the aggregate functions are called once per group.

```
-- Not only one resulting row, but one resulting row per lastname together with the average weight of all rows with this lastname.
SELECT AVG(weight)
FROM   person
GROUP BY lastname;
```

In such cases the GROUP BY column(s) may be displayed as it is impossible that they change within the group.

```
-- The lastname may be shown as it is the GROUP BY criteria
SELECT lastname, AVG(weight)
FROM   person
GROUP BY lastname;
```

The NULL special marker

If a row contains no value (it holds the NULL special marker) in the named column, the row is not part of the computation.

```
-- If ssn is NULL, this row will not count.
SELECT COUNT(ssn)
FROM   person;
```

ALL vs. DISTINCT

The complete signatures of the functions are a little more detailed. We can prepend the column name with one of the two key words ALL or DISTINCT. If we specify ALL, which is the default, every value is part of the computation, else only those, which are distinct from each other.

```
function_name ([ALL|DISTINCT]<column name>)
COUNT (DISTINCT weight) -- as an example
```

Hint

The standard defines some more aggregate functions to compute statistical messures. Also the keywords ANY, EVERY and SOME formally are defined as aggregate functions. We will discuss them on a separate page.

Scalar functions

Scalar functions act on a 'per row basis'. They are called once per row and they return one value per call. Often they are grouped according to the data types they act on:

- String functions
  SUBSTRING(<column name> FROM <pos> FOR <len>) returns a string starting at position <pos> (first character counts '1') in the length of <len>.
  UPPER(<column name>) returns the uppercase equivalent of the column value.

LOWER(<column name>) returns the lowercase equivalent of the column value.
CHARACTER_LENGTH(<column name>) returns the length of the column value.
TRIM(<column name>) returns the column value without leading and trailing spaces.
TRIM(LEADING FROM <column name>) returns the column value without leading spaces.
TRIM(TRAILING FROM <column name>) returns the column value without trailing spaces.

- Numeric functions
  SQRT(<column name>) returns the square root of the column value.
  ABS(<column name>) returns the absolute value of the column value.
  MOD(<column name>, <divisor>) returns the remaining of column value divided by divisor.
  others: FLOOR, CEIL, POWER, EXP, LN.

- Date, Time & Interval functions
  EXTRACT(month FROM date_of_birth) returns the month of column date_of_birth.

- build-in functions. They do not have any input parameter.
  CURRENT_DATE() returns the currente date.
  CURRENT_TIME() returns the currente time.

There is another wikibook where those functions are shown in detail. The data type of the return value is not always identical to the type of the input, e.g. 'character_length()' receives a string and returns a number.

Here is an example with some scalar functions:

```
SELECT LOWER(firstname), UPPER(lastname), CONCAT('today is: ', CURRENT_DATE)
FROM   person;
```

Exercises

What is the hightest id used so far in the hobby table?

Click to see solution

```
SELECT max(id)
FROM   hobby;
```

Which lastname will occur first in an ordered list?

Click to see solution

```
SELECT min(lastname)
FROM   person;
```

Are there aggregate functions where it makes no difference to use the ALL or the DISTINCT key word?

Click to see solution

```
Yes. min(ALL <column name>) leads to the same result as min(DISTINCT <column name>) as
it makes no difference whether the smallest value occurs one or more times. The same is true for max().
```

Show persons with a short firstname (up to 4 characters).

Click to see solution

```
-- We can use functions as part of the WHERE clause.
SELECT *
FROM   person
WHERE  character_length(firstname) <= 4; -- Hint: Some implementations use a different function name: length() or len().
```

Show firstname, lastname and the number of characters for the concatenated string. Find two different solutions. You may use the character_length() function to compute the length of strings and the concat() function to concatenate strings.

Click to see solution

```
-- Addition of the computed length. Hint: Some implementations use a different function name: length() or len().
SELECT firstname, lastname, character_length(firstname) + character_length(lastname)
FROM   person;
-- length of the concatenated string
SELECT firstname, lastname, character_length(concat (firstname, lastname))
```

```
FROM   person;
-- show both solutions together
SELECT firstname, lastname,
       character_length(firstname) + character_length(lastname) as L1,
       character_length(concat (firstname, lastname)) as L2
FROM   person;
```

Tables, views and results of SELECT commands are in somewhat similar to sets of set theory. In this comparision the elements of sets correspond to rows of tables, views and SELECT results. The differences between set theory and the itemized SQL constructs are:

- Sets of set theory do not allow duplicates whereas SQL allows duplicates. (Even different rows of one table may be identical as there is no duty to use the concept of primary keys.) In the following we use the term *multiset* when we speak about sets in SQL where duplicates are possible.
- Sets of set theory and multisets are not ordered. But for the result of a SELECT command we can enforce an ordering by means of the optional ORDER BY clause.

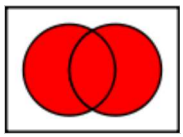
The comparision between set theory and SQL goes even further. In SQL we have operations which acts on *multisets* in the sense of set theory: The SQL operations UNION, INTERSECT and EXCEPT (some name it MINUS) process intermediate *multisets* generated by differents SELECT commands. The operations expect the *multisets* are of the same type. This means mainly that they **must** have the same number of columns. Also their data type should correlate, but this is not mandatory. If they do not, the DBMS will cast them to a common data type - if possible.

UNION

The UNION operation pushs the result of several SELECT commands together. The result of the UNION contains those values, which are in the first or in the second intermediate result.

```
-- Please consider that this is only one command (only ONE semicolon at the very end)
SELECT firstname  -- first SELECT command
FROM   person
  UNION           -- push both intermediate results together to one result
SELECT lastname   -- second SELECT command
FROM   person;
```

This is a single SQL command. It consists of two SELECTs and one UNION operation. The SELECTs are evaluated first. Afterwards their results are pushed together to one single result. In our example the result contains all lastnames and firstnames in a single column (our example may be of limited help in praxis, it's only a demonstration for the UNION).



The **UNION** of two intermediate results

DISTINCT / ALL

If we examine the result closely, we will notice that it consists only of 17 values. The table *person* contains ten rows so that we probably expect twenty values in the result. If we perform the 'SELECT firstname ...' and 'SELECT lastname ...' as separate commands without the UNION, we receive for both commands 10 values. The explanation for the 3 missing values is the UNION command. It behaves by default that it removes duplicates. Therefore some of the intermediate values are skipped. If we want to obtain this duplicate values we have to extent the UNION. It can be widen with one of the two key words DISTINCT or ALL. DISTINCT is the default and its behaviour is the removal of duplicate values which we have seen before. ALL leads to the retention of all values, independent whether they appeared before or not.

```
-- remove (that's the default) or keep duplicates
SELECT ...
  UNION [DISTINCT | ALL]
SELECT ...
[ -- it is possible to 'UNION' more than 2 intermediate results
  UNION [DISTINCT | ALL]
SELECT ...
];
```

A hint for Oracle users: The use of the key word DISTINCT, which is the default, is not accepted by Oracle. Omit it.

General hint

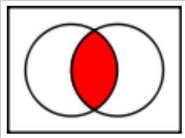
In most cases the UNION combines SELECT commands on different tables or on different columns of the same table. SELECT commands on the same column of a single table usually use the WHERE clause in combination with boolean logic.

```
-- A very unusual example. People apply such querys on the same table only in combination with very complex WHERE conditions.
-- This example would normally expressed with a phrasing similar to: WHERE lastname IN ('de Winter', 'Goldstein');
SELECT *
FROM   person
WHERE  lastname = 'de Winter'
  UNION ALL
SELECT *
```

```
FROM   person
WHERE  lastname = 'Goldstein';
```

INTERSECT

The INTERSECT operation evaluates to those values, which are in both intermediate results, in the first as well as in the second.



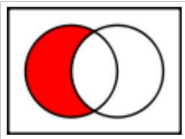
INTERSECT

```
-- As in our example database there is no example for the INTERSECT we insert a new person.
-- This person has the same lastname 'Victor' as the first name of another person.
INSERT INTO person VALUES (21, 'Paul', 'Victor', DATE'1966-04-02', 'Washington', '078-05-1121', 66);
COMMIT;
-- All firstnames which are used as lastname.
SELECT firstname  -- first SELECT command
FROM   person
  INTERSECT       -- looking for common values
SELECT lastname   -- second SELECT command
FROM   person;
```

A hint to MySQL users: MySQL (5.5) does not support INTERSECT operation. But as it is not an elementary operation, there are workarrounds.

EXCEPT

The EXCEPT operation evaluates to those values, which are in the first intermediate result but not in the second.



EXCEPT

```
-- All firstname with the exception of 'Victor' because there is a lastname with this value.
SELECT firstname  -- first SELECT command
FROM   person
  EXCEPT          -- are there values in the result of first SELECT but not of second?
SELECT lastname   -- second SELECT command
FROM   person;
```

A hint to MySQL users: MySQL (5.5) does not support the EXCEPT operation. But as it is not an elementary operation, there are workarrounds.
A hint to Oracle users: Oracle use the key word MINUS instead of EXCEPT.

```
-- Clean up the example database
DELETE FROM person WHERE id > 10;
COMMIT;
```

Order By

We can combine set operations with all other elements of SELECT command, in particular with ORDER BY and GROUP BY. But this may lead to some uncertainties. Therefore, we would like to explain some of the details below.

```
SELECT firstname  -- first SELECT command
FROM   person
  UNION           -- push both intermediate results together to one result
SELECT lastname   -- second SELECT command
FROM   person
ORDER BY firstname;
```

To which part of the command belongs the ORDER BY? To the first SELECT, to the second SELECT or to the result of the UNION? The SQL rules determine, that set operations are evaluated before ORDER BY clauses (as always parenthesis can change the order of evaluation). Therefore the ORDER BY sorts the final result and not any of the intermediate results.

We rearrange the example in the hope that things get clear.

```
-- Equivalent semantic
```

```
SELECT * FROM
  (SELECT firstname  -- first SELECT command
   FROM   person
     UNION           -- push both intermediate (unnamed) results together to the next intermediate result 't'
   SELECT lastname   -- second SELECT command
   FROM   person
  ) t                -- 't' is the name for the intermediate result generated by UNION
ORDER BY t.firstname;
```

First the two SELECTS are evaluated, afterwareds the UNION. This intermediate result gets the name 't'. 't' is ordered.

Often one would like to achieve that the rows from the first SELECT are ordered independent from those of the second SELECT. We can do this by adding a virtuel column to the result of both SELECTs.

```
SELECT '1' as dummy, firstname
FROM   person
  UNION
SELECT '2', lastname
FROM   person
ORDER BY dummy, firstname;
```

Group By

With the GROUP BY clause things are little more complicated than with ORDER BY. The GROUP BY refers to the last SELECT or - to say it the other way round - to the SELECT of its direct level.

```
-- Will not work because the GROUP BY belongs to the second SELECT and not to the UNION!
SELECT firstname
FROM   person
  UNION
SELECT lastname
FROM   person
GROUP BY firstname;
--
-- Works, but possibly not what you want to do.
-- The alias name for the (only) column of the UNION is 'firstname'.
SELECT firstname
FROM   person
  UNION
-- We group over the (only) column of the second SELECT, which is 'lastname' and results in 7 values
SELECT lastname
FROM   person
GROUP BY lastname;
--
-- Make things clear: rearrange the query to group over the final result
SELECT * FROM
  (SELECT firstname  -- columnnames of the first SELECT determins the columnnames of the UNION
   FROM   person
     UNION
   SELECT lastname
   FROM   person
  ) t
GROUP BY t.firstname; -- now we can group over the complete result
```

Exercises

Show the lowest, highest and mean weight as a) 3 values of 1 row and b) 1 value in 3 rows.

Click to see solution

```
-- 1 row
SELECT min(weight), max(weight), avg(weight)
FROM   person;

-- 3 rows
SELECT min(weight)
FROM   person
  UNION
SELECT max(weight)
FROM   person
  UNION
SELECT avg(weight)
FROM   person;
```

Extend the previous 3-rows-solution to meet two additional criteria: a) consider only persons born in San Francisco and b) add a virtual column to show 'Min', 'Max' and 'Avg' according to the correlating numeric values.

Click to see solution

```
SELECT 'Min', min(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
```

```
  UNION
SELECT 'Max', max(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
  UNION
SELECT 'Avg', avg(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco';
```

Extend the previous solution to order the result: the minimum value first, followed by the average and then the highest value.

Click to see solution

```
-- 'ugly' solution
SELECT '1 Min' AS note, min(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
  UNION
SELECT '3 Max' AS note, max(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
  UNION
SELECT '2 Avg' AS note, avg(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
ORDER BY note;

-- 'clean' solution
SELECT 1 AS note, 'Min', min(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
  UNION
SELECT 3 AS note, 'Max', max(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
  UNION
SELECT 2 AS note, 'Avg', avg(weight)
FROM   person
WHERE  place_of_birth = 'San Francisco'
ORDER BY note;
```

Create a list of lastnames for persons with a weight greater than 70 kg together with
all e-mail values (one value per row). There is no concordance between lastnames and e-mails.
(This example is not very helpfull for praxis, but instructive.)

Click to see solution

```
SELECT lastname
FROM   person
WHERE  weight > 70
  UNION
SELECT contact_value
FROM   contact
WHERE  contact_type = 'email';
```

In the previous example the lastname 'de Winter' is shown only once. But there are more than one persons of the family with a weight greater than 70 kg.
Why?
Extend the previous solution to show as much resulting rows as hits to the criteria.

Click to see solution

```
-- Extend 'UNION' to 'UNION ALL'. The default is 'UNION DISTINCT'
SELECT lastname
FROM   person
WHERE  weight > 70
  UNION ALL
SELECT contact_value
FROM   contact
WHERE  contact_type = 'email';
```

Sometimes it's necessary to translate stored values (or values to be stored) from one representation to another. Suppose there is a column *status* with legal values from 0 to 9 but the end-users should receive strings which explain the meaning of the numeric values in short, eg.: 'ordered', 'delivered', 'back delivery', 'out of stock', ... The recommended way to do this is a separate table where the numeric values maps to the explanatory strings. Notwithstanding this, application developers may favor a solution within an application server.

The CASE expression, which is shown on this page, is a technique for solving the described situation as part of a SELECT, INSERT or UPDATE command as well as solving additional problems. As part of the language it's a powerful term which can be applied at plenty places within SQL commands. On this page we focus on its use together with the SELECT command. The strategy and syntax for CASE within INSERT and UPDATE are equivalent and are presented over there. In comparison with the recommended technique of a separate

table for the translation the CASE expression is much more flexible (which is not an advantage in all cases).

Two Examples

```
-- Technical term: "simple case"
-- Select id, contact_type in a translated version and contact_value
SELECT id,
       CASE contact_type
         WHEN 'fixed line' THEN 'Phone'
         WHEN 'mobile'     THEN 'Phone'
         ELSE                  'Not a telephone number'
       END,
       contact_value
FROM   contact;
```

The CASE expression is introduced with its key word CASE and runs up to the END key word. In this first example it specifies a column name and a series of WHEN/THEN clauses with an optional ELSE clause. The WHEN/THEN clauses are compared and evaluated against the values of the named column, one after the other. If non of them hits, the ELSE clause applies. If there is no ELSE clause and non of the WHEN/THEN clauses hit, the NULL special marker will be applied.

The comparison between the values of the column and the fixed values within the WHEN/THEN clause is done solely by "=" (equals). This is a good starting point, but real applications need more than that. Therefore there is a variant of the CASE.

```
-- Technical term: "searched case"
-- Select persons name, weight and a denomination of the weight
SELECT firstname,
       lastname,
       weight,
       CASE
         WHEN (weight IS NULL OR weight = 0) THEN 'weight is unknown'
         WHEN weight < 40                    THEN 'lightweight'
         WHEN weight BETWEEN 40 AND 85       THEN 'medium'
         ELSE                                     'heavyweight'
       END
FROM   person;
```

The crucial point is the direct succession of the two key words CASE and WHEN. There is **no** column name between them. In this variant there must be a complete expression, which evaluates to one of the 3-value-logic terms *true*, *false* or *unknown*, between each WHEN and THEN. Now it is possible to use all the comparisons and boolean operators as they are known by the WHERE clause. It is even possible to compare different columns or function calls with each other.

Syntax

There are the two variants *simple case* and *searched case*.

```
-- "simple case" performs successive comparisons using the equal operator: <column_name> = <expression_x>
CASE <column_name>
  WHEN <expression_1> THEN <result_1>
  WHEN <expression_2> THEN <result_2>
  ...
  ELSE                     <default_result>  -- optional
END

-- "searched case" is recognised by 'nothing' between CASE and first WHEN
CASE
  WHEN <condition_1> THEN <result_1>
  WHEN <condition_2> THEN <result_2>
  ...
  ELSE                    <default_result>  -- optional
END
```

The *simple case* is limited to one column and the use of the equal operator whereas the *searched case* may evaluate arbitrary columns of the (intermediate) result with arbitrary operators, functions or predicates.

Typical Use Cases

The use of CASE expressions is not limited to projections (the column list between SELECT and FROM). As the clause evaluates to a value, it can be applied as a substitution for values at several places within SQL commands. In the following, we offer some examples.

ORDER BY clause

Sort contact values in the order: all fixed lines, all mobile phones, all emails, all icq's. Within each group sort over the contact values.

```
SELECT *
```

```
FROM   contact
ORDER  BY
       -- a "simple case" construct as substitution for a column name
       CASE contact_type
         WHEN 'fixed line' THEN 0
         WHEN 'mobile'     THEN 1
         WHEN 'email'      THEN 2
         WHEN 'icq'        THEN 3
         ELSE                   4
       END,
       contact_value;
```

In the next example persons are ordered by weight classes, within the classes by their name.

```
-- order by weight classes
SELECT firstname, lastname, weight,
       CASE
         WHEN (weight IS NULL OR weight = 0) THEN 'weight is unknown'
         WHEN weight < 40                    THEN 'lightweight'
         WHEN weight BETWEEN 40 AND 85       THEN 'medium'
         ELSE                                     'heavyweight'
       END
FROM   person
ORDER  BY
       -- a "searched case" construct with IS NULL, BETWEEN and 'less than'.
       CASE
         WHEN (weight IS NULL OR weight = 0) THEN 0
         WHEN weight < 40                    THEN 1
         WHEN weight BETWEEN 40 AND 85       THEN 2
         ELSE                                     3
       END, lastname, firstname;
```

WHERE clause

Within the WHERE clauses there may occur fixed values or column names. CASE expressions can be used as a substitution for them. In the example persons receive a discount on their weight depending on their place of birth (consider it as a theoretical example). Thus Mr. Goldstein with its 95 kg counts only with 76 kg and is not part of the result set.

```
SELECT *
FROM   person
WHERE  CASE
         -- Modify weight depending on place of birth.
         WHEN place_of_birth = 'Dallas'   THEN weight * 0.8
         WHEN place_of_birth = 'Richland' THEN weight * 0.9
         ELSE                                  weight
       END > 80
OR     weight < 20; -- any other condition
```

Exercises

Show firstname, lastname and the gender of all persons. Consider Larry, Tom, James, John, Elias, Yorgos, Victor as 'male', Lisa as 'female' and all others as 'unknown gender'. Use a *simple case* expression.

Click to see solution

```
SELECT firstname, lastname,
       CASE firstname
         WHEN 'Larry'   THEN 'male'
         WHEN 'Tom'     THEN 'male'
         WHEN 'James'   THEN 'male'
         WHEN 'John'    THEN 'male'
         WHEN 'Elias'   THEN 'male'
         WHEN 'Yorgos'  THEN 'male'
         WHEN 'Victor'  THEN 'male'
         WHEN 'Lisa'    THEN 'female'
         ELSE                'unknown gender'
       END
FROM   person;
```

Use a *searched case* expression to solve the previous question.

Click to see solution

```
SELECT firstname, lastname,
       CASE
         WHEN firstname in ('Larry', 'Tom', 'James', 'John', 'Elias', 'Yorgos', 'Victor')
                                THEN 'male'
         WHEN firstname = 'Lisa'     THEN 'female'
```

```
        ELSE                            'unknown gender'
      END
FROM   person;
```

Show firstname, lastname and a classification of all persons. Classify persons according to the length of their firstname. Call the class 'short name' if character_length(firstname) < 4, 'medium length' if < 6, 'long name' else.

Click to see solution

```
-- Hint: Some implementations use a different function name: length() or len().
SELECT firstname, lastname,
       CASE
         WHEN CHARACTER_LENGTH(firstname) < 4 THEN 'short name'
         WHEN CHARACTER_LENGTH(firstname) < 6 THEN 'medium length'
         ELSE                                     'long name'
       END
FROM   person;
```

Count the number of short, medium and long names of the above exercise.
Click to see solution

```
-- Hint: Some implementations use a different function name: length() or len().
SELECT SUM(CASE
             WHEN CHARACTER_LENGTH(firstname) < 4 THEN 1
             ELSE                                     0
           END) as short_names,
       SUM(CASE
             WHEN CHARACTER_LENGTH(firstname) between 4 and 5 THEN 1
             ELSE                                                  0
           END) as medium,
       SUM(CASE
             WHEN CHARACTER_LENGTH(firstname) > 5 THEN 1
             ELSE                                     0
           END) as long_names
FROM   person;
```

A subquery is a complete SELECT command which is used within another SELECT, UPDATE, INSERT or DELETE command. The only difference to a simple SELECT is, that it is enclosed in parenthesis.

Classification

Depending on the type of the created result there are three classes of subqueries:

- *Scalar Value Subquery*: The subquery returns one single value, e.g: `(SELECT max(weight) FROM person)`.
- *Row Subquery*: The subquery returns one single row of one or more values, e.g: `(SELECT min(weight), max(weight) FROM person)`.
- *Table Subquery*: The subquery returns a list of rows, which is a table, e.g: `(SELECT lastname, weight FROM person)`. For the classification it makes no difference whether the resulting list contains zero, one or more rows. The demarcation between a table subquery and a row subquery is that **potentially** more than one row may occur.

Every type can be used on all positions where the type it stands for may occur: the scalar value subquery where a single value may occur, the row subquery where a single row may occur and the table subquery where a table may occur. Additionally table subqueries may occur as an argument of an EXISTS, IN, SOME, ANY or ALL predicate.

Independent from this classification subqueries may be *correlated subqueries* or *non-correlated subqueries*. Correlated subqueries have a correlation to the surrounding query by the fact that they use values from the surrounding query within the subquery. Non-correlated subqueries are independent from the surrounding query. This distinction is shown in detail in the next chapter but applies also to the other two subquery classes.

Because correlated subqueries use values, which are determined by the surrounding query and may change from row to row, the subquery is executed - conceptional - as often as resulting rows of the surrounding query exist. This might lead to performance problems. Nevertheless correlated subqueries are an often used construct. In many cases exist equivalent constructs which use a JOIN. Which one shows the better performance depends highly on the DBMS, and the number of involved rows, the existence of indices and a lot more variables.

Scalar Value Subquery

The first example creates a list of lastnames, weights and the average weight of all persons.

```
SELECT id,
```

```
       lastname,
       weight,
       (SELECT avg(weight) FROM person)    -- this is the subquery
FROM   person
ORDER BY lastname;
```

Because the subquery uses the `avg()` function, the SQL compiler knows that it will return exactly one single value. Therefore it's type is *Scalar Value Subquery* and can be used on positions where scalar values may occur, e.g. in the list between SELECT and FROM.

In the next example the subquery is used as a deputy for a value within the WHERE clause.

```
-- Persons who weigh more than the average of all persons
SELECT id, lastname, weight
FROM   person
WHERE  weight >= (SELECT avg(weight) FROM person)   -- another position for the subquery
ORDER BY lastname;
```

Both examples use the table *person* twice. Just as well one can use different tables. There is no dependency between the table name in the subquery and in the surrounding query. This applies to all classes of correlated and non-correlated subqueries. The subqueries may retrieve any value from any other table, e.g. the number of contacts.

This first two examples show non-correlated subqueries, which means, that the subqueries are independent from the queries in which they are embedded. They are executed only once.

But often an application faces a situation, where the subquery must use values from the outside query (similar to subroutines which uses parameters). This kind of subquery is called a correlated subquery. As an example the next query lists persons together with the average weight of their family.

```
SELECT id, firstname, lastname, weight,
       (SELECT avg(weight)
        FROM   person sq                -- 'sq' is an arbitrary alias name for the table in the subquery
        WHERE  sq.lastname = p.lastname -- identify the inner and outer table by its alias names
       ) family_average                 -- an arbitrary alias name for the computed family average
FROM   person p                         -- 'p' is an arbitrary alias name for the table in the surrounding query
ORDER BY lastname, weight;
```

The subselect gets one row of the surrounding SELECT after the next as an parameter with the name 'p'. Within the subselect all columns of the row 'p' are known and may be used. Here the family name from the outside row is used in the subquery to find all persons within the family and the average weight of the family members.

Be careful: Correlated subqueries are executed once per row of the surrounding query. Therefore they are much more costly than non-correlated subqueries. There might exist an equivalent solution using JOIN or GROUP BY which works with better performance. The query optimizer of the DBMS internally might rearrange the given subquery into one of the equivalent forms. But this does not work in all cases.

The distinction between correlated and non-correlated subqueries is universal. It applies also to the other subquery classes.

Row Subquery

This example retrieves one or more persons, whose firstname is the lowest (in the sense of the lexical order) of all firstnames and whose lastname is the lowest of all lastnames. Because of the AND condition it might be the case that no person is found.

```
-- One resulting row: Elias Baker
SELECT *
FROM   person
WHERE  (firstname, lastname) = (SELECT MIN(firstname), MIN(lastname) FROM person);
```

Within the subquery the lowest first- and lastnames are retrieved. The use of the `min()` function garanties that not more than one row with two columns will arise - therefore it is a row subquery. In the surrounding query this intermediate result is compared with each row of the complete table *person* or - if present - an index is used.

It's a fortune that the command retrieves a row. In most cases the lowest first- and lastname results from different persons. But also in those cases the command is syntactically correct and will not throw any exception.

In the next example persons with the lowest first- and lastnames within every family are retrieved. To do so, it is neccessary to use a correlated row subquery.

```
-- 7 rows, one per family
SELECT *
FROM   person p
WHERE (firstname, lastname) =
  (SELECT MIN(firstname), MIN(lastname) FROM person sq where p.lastname = sq.lastname);
```

Again, there are the two incarnations of table *person*, one with the alias name 'p' in the surrounding query and one with the alias name 'sq' in the subquery. The subquery is called once per resulting row of the surrounding query, because the 'p.lastname' may change with every row of 'p'.

Within every family there is as least one person which achievs the condition - it is also conceivable that several persons achiev the condition.

Table Subquery

The next example retrievs persons who have a hobby. The class of the subquery is: non-correlated table subquery (used as a condition in the IN predicate).

```
SELECT *
FROM   person
WHERE  id IN
  (SELECT person_id FROM contact);  -- the subquery
```

The subquery creates multiple rows with one column for each of them. This constitutes a new, intermediate table. Therefore this example is a table subquery.

The IN operator is able to act on this intermediate table. In contrast, it is not possible to use operators like '=' or '>' on this kind of intermediate result. In this cases the SQL compiler will recognize an syntax error.

The next example is an extention of the first one. It adds a correlation criterion between the query and the subquery by requesting the lastname within an email-address.

```
-- A correlated table subquery, looking for lastnames within e-mail-addresses
SELECT *
FROM   person p
WHERE  id IN
  (SELECT person_id
   FROM   contact c
   WHERE  c.contact_type = 'email'
   AND    UPPER(c.contact_value) LIKE CONCAT(CONCAT('%', UPPER(p.lastname)), '%'));
```

The last comparision after the AND is a little bit complex. It uses the functions CONCAT() and UPPER() as well as the predicate LIKE, but this is not of interest for the actual topic 'subquery'. The important part is that the subquery refers to 'p.lastname' of the surrounding query. Only Mr. Goldstein meets the criterion that his e-mail address contains his lastname when the two columns are compared case-insensitive.

Remark: CONCAT() concatenates two strings. UPPER() converts a string to upper-case. LIKE in combination with the '%' sign looks for one string within another.

Next, there is an example where a non-correlated table subquery is object to a join operation.

```
-- Persons plus maximum weight of their family
SELECT *
FROM   person p
JOIN   (SELECT lastname, max(weight) max_fam_weight
        FROM   person
        GROUP BY lastname
       ) AS sq ON p.lastname = sq.lastname   -- join criterion between subquery table 'sq' and table 'p'
;
```

Another Example

The example shows a solution for a common problem. Sometimes there are rows describing an outdated stage of entities. Those rows - for one logical entity - differ from each other in some columns and there is an additional column *version* to track the time flow.

Here is the example table *booking* and its data.

```
-- The table holds actual and historical values
CREATE TABLE booking (
  -- identifying columns
  id             DECIMAL      NOT NULL,
  booking_number DECIMAL      NOT NULL,
  version        DECIMAL      NOT NULL,
  -- describing columns
  state          CHAR(10)     NOT NULL,
  enter_ts       TIMESTAMP    NOT NULL,
  enter_by       CHAR(20)     NOT NULL,
  -- ...
  -- select one of the defined columns as the Primary Key
```

```
  CONSTRAINT booking_pk PRIMARY KEY (id),
  -- forbid duplicate recordings
  CONSTRAINT booking_unique UNIQUE (booking_number, version)
);

-- Add data
INSERT INTO booking VALUES (1, 4711, 1, 'created',   TIMESTAMP'2014-02-02 10:01:01', 'Emily');
INSERT INTO booking VALUES (2, 4711, 2, 'modified',  TIMESTAMP'2014-02-03 11:10:01', 'Emily');
INSERT INTO booking VALUES (3, 4711, 3, 'canceled',  TIMESTAMP'2014-02-10 09:01:01', 'John');
--
INSERT INTO booking VALUES (4, 4712, 1, 'created',   TIMESTAMP'2014-03-10 12:12:12', 'Emily');
INSERT INTO booking VALUES (5, 4712, 2, 'delivered', TIMESTAMP'2014-03-12 06:01:00', 'Charles');
--
INSERT INTO booking VALUES (6, 4713, 1, 'created',   TIMESTAMP'2014-03-11 08:50:02', 'Emily');
INSERT INTO booking VALUES (7, 4713, 2, 'canceled',  TIMESTAMP'2014-03-12 08:40:12', 'Emily');
INSERT INTO booking VALUES (8, 4713, 3, 'reopend',   TIMESTAMP'2014-03-13 10:04:32', 'Jack');
INSERT INTO booking VALUES (9, 4713, 4, 'delivered', TIMESTAMP'2014-03-15 06:40:12', 'Jack');
--
COMMIT;
```

The problem is to retrieve all **actual** rows, which are those with the highest version number within each booking. Bookings are considered to be the same, if they have the same booking_number.

The first solution uses a non-correlated table subquery.

```
SELECT *
FROM   booking b
WHERE  (booking_number, version) IN
  (SELECT booking_number, MAX(version) FROM booking sq GROUP BY booking_number) -- the subquery
ORDER BY booking_number;
```

The subquery creates a list of booking numbers together with their highest version. This list is used by the surrounding query to retrieve the required rows with all its columns.

The second solution uses a correlated scalar value subquery.

```
SELECT *
FROM   booking b
WHERE  version =
  (SELECT max(version) FROM booking sq WHERE sq.booking_number = b.booking_number)
ORDER BY booking_number;
```

The surrounding query retrievs all rows of the table. For each of them it calls the subquery, which retrieves the highest version within this booking_number. In most cases this highest version differs from the version of the actual row and because of the '=' operator those rows are not part of the result. Only those, whose version is equal to the value determined in the subquery (and whose booking_number is the same as those used in the subquery) are part of the final result.

A variation of the introducing question may be to retrieve only historical rows (all versions except the highest one) for one special booking.

```
SELECT *
FROM   booking b
WHERE  version !=
  (SELECT max(version) FROM booking sq WHERE sq.booking_number = b.booking_number)
AND    booking_number = 4711
ORDER BY version;
```

The surrounding query restricts the rows to those of one special booking. The subquery is called only for those rows.

It's easy to run into pitfalls:

```
-- Unexpected result!
SELECT *
FROM   booking b
WHERE  version != (SELECT max(version) FROM booking)
AND    booking_number = 4711
ORDER BY version;
```

The above query returns all versions of booking 4711 including the actual one! To get the expected result, it's necessary to 'link' the surrounding query and the subquery together.

Exercises

Find the booking with the most versions.

Click to see solution

```
-- The subselect return exactly ONE single value. Therefor it's a (non-correlated) single value subquery.
-- But this is only a intermediate result. The final result may contain several rows, which is not the case in our example database!
SELECT *
FROM   booking
WHERE  version = (SELECT MAX(version) FROM booking);
```

Find all bookings with are canceled (in the latest version).

Click to see solution

```
-- It's necessary to link the subquery with the surrounding query.
SELECT *
FROM   booking b
WHERE  version =
  (SELECT MAX(version) FROM booking sq WHERE sq.booking_number = b.booking_number)
AND    state = 'canceled';

-- Additionally within the resulting rows there must be a correlation between the version and the state.
-- This is accomplished with the AND key word at the level of the surrounding query. If the AND works within
-- the subquery, the result does not meet the expectations.
SELECT *
FROM   booking b
WHERE  version =
  (SELECT MAX(version) FROM booking sq WHERE sq.booking_number = b.booking_number AND state = 'canceled');
```

Create a list of all persons together with the number of persons which are born in the same city as they itself.

Click to see solution

```
-- The subselect uses the place_of_birth of the outside row. Therefore it's a correlated subquery.
SELECT firstname,
       lastname,
       place_of_birth,
       (SELECT COUNT(*) FROM person sq WHERE p.place_of_birth = sq.place_of_birth) cnt -- an arbitrary name for the additional colum
FROM   person p;
```

Create a list of all persons together with the number of their contact information.

Click to see solution

```
-- The subselect uses the ID of the outside row. Therefore it's a correlated subquery.
SELECT firstname,
       lastname,
       (SELECT COUNT(*) FROM contact c WHERE p.id = c.person_id) cnt -- an arbitrary name for the additional column
FROM   person p;
```

Create a list of all persons together with the number of their e-mail-addresses.

Click to see solution

```
SELECT firstname,
       lastname,
       (SELECT COUNT(*)
        FROM   contact c
        WHERE  p.id = c.person_id
        AND    contact_type = 'email'  -- The subselect is a complete SELECT. Therefor all elements of
                                       -- a 'regular' SELECT may be used: Join, functions, ... and: SUBSELECT
       ) cnt                           -- an arbitrary name of the additional column
FROM   person p;
```

Create a list of all persons together with the number of their contact information. (Same question as above.)
Replace the subquery by a JOIN construct.

Click to see solution

```
-- Step 1 (for demonstration purpose only): To retrieve ALL persons, it's necessary to use an OUTER JOIN
SELECT firstname,
       lastname,
       c.contact_type
FROM   person p
LEFT OUTER JOIN contact c ON p.id = c.person_id;
--
-- Step 2 (complete solution): Add the counter. To do so, the result must be grouped.
SELECT firstname,
       lastname,
       count(c.contact_type)
FROM   person p
LEFT OUTER JOIN contact c ON p.id = c.person_id
```

```
GROUP BY firstname, lastname;
```

For which persons there are NO contact information?

Click to see solution

```
-- The subquery returns more than one row. Therefore it's a table subquery.
SELECT firstname, lastname
FROM   person
WHERE  id NOT IN (SELECT person_id FROM contact);  -- the subquery
```

Often users and applications request information in a form which differs from the structure of existing tables. To achieve those requests the SELECT command offers plenty possibilities: projections, joins, group by clause and so on. If there are always the same requests, what is the case in particular for applications, or if the table structure intentionally should be hidden from the application-level, views can be defined. Furthermore the access rights to views may be different from those to tables.

Views look like a table. They have columns of a certain data type, which can be retrieved in the same way as columns of a table. But views are only definitions, they don't have data of its own! Their data is always the data of a table or is based on another view. A view is a **different sight** to the stored data or somewhat like a **predefined SELECT**.

Create a View

One creates a view by specify its name, column names - which is optionally - and especially the SELECT command on which the view is based. Within this SELECT all elements are allowed in the same way as in a standalone SELECT command. If no column names are specified the column names of the SELECT are used.

```
CREATE VIEW <view_name> [(column_name, ...)] AS
  SELECT ...  -- as usual
;
```

Examples and Explanations

Example 1: Hide Columns

As a first example here is the view *person_view_1* which contains all but *id* and *ssn* columns of table *person*. Users which have the right to read from this view but not from the table *person* doesn't have access to *id* and *ssn*.

```
CREATE VIEW person_view_1 AS
  SELECT firstname, lastname, date_of_birth, place_of_birth, weight
  FROM   person;

-- SELECTs on views have identical syntax as SELECTs on tables
SELECT *
FROM   person_view_1
ORDER BY lastname;

-- The column 'id' is not part of the view. Therefore it is not seen and cannot be used
-- anywhere in SELECTs to person_view_1.
-- This SELECT will generate an error message because of missing 'id' column:
SELECT *
FROM   person_view_1
WHERE  id = 5;
```

As indicated in the above 'order by' example it is possible to use all columns of the view (but not all of the table!) within any part of SELECTs to the view: in the projection, the WHERE, ORDER BY, GROUP BY and HAVING clauses, in function calls and so on.

```
-- SELECTs on views have identical syntax as SELECTs on tables
SELECT count(lastname), lastname
FROM   person_view_1
GROUP BY lastname
ORDER BY lastname;
```

Example 2: Rename Columns

Next there is a renaming of a column. The column name *lastname* of the table will be *familyname* in the view.

```
-- first technique: list the desired column names within parenthesis after the view name
CREATE VIEW person_view_2a (firstname, familyname, date_of_birth, place_of_birth, weight) AS
```

```
  SELECT                    firstname, lastname,   date_of_birth, place_of_birth, weight
  FROM   person;

-- second technique: rename the column in the SELECT part
CREATE VIEW person_view_2b AS
  SELECT firstname, lastname AS familyname, date_of_birth, place_of_birth, weight
  FROM   person;
-- Hint: technique 1 overwrites technique 2

-- Access to person.lastname is possible via person_view_2a.familyname or person_view_2b.familyname.
-- The objects person.familyname, person_view_2a.lastname or person_view_2b.lastname does not exist!
```

Example 3: Apply WHERE Condition

Not only columns can be hidden in a view. It's also possible to hid complete rows, because the view definition may contain a WHERE clause.

```
-- Restrict access to few rows
CREATE VIEW person_view_3 AS
  SELECT firstname, lastname, date_of_birth, place_of_birth, weight
  FROM   person
  WHERE  place_of_birth in ('San Francisco', 'Richland');

-- Verify result:
SELECT *
FROM   person_view_3;
```

This view contains only persons born in San Francisco or Richland. All other persons are hidden. Therefore the following SELECT retrieves nothing although there are persons in the table which fulfil the condition.

```
-- No hit
SELECT *
FROM   person_view_3
WHERE  place_of_birth = 'Dallas';

-- One hit
SELECT *
FROM   person
WHERE  place_of_birth = 'Dallas';
```

Example 4: Use Functions

This example usues the sum() function.

```
--
CREATE VIEW person_view_4 AS
  -- General hint: Please consider that not all columns are availabe in a SELECT containing a GROUP BY clause
  SELECT lastname, count(lastname) AS count_of_members
  FROM   person
  GROUP BY lastname
  HAVING count(*) > 1;

-- Verify result: 2 rows
SELECT *
FROM   person_view_4;

-- The computed column 'count_of_members' may be part of a WHERE condition.
-- This SELECT results in 1 row
SELECT *
FROM   person_view_4
WHERE  count_of_members > 2;
```

In this example the elaborated construct 'GROUP BY / HAVING' is hidden from users and applications.

Example 5: Join

Next, there is an example where a view contains columns out of serveral tables. To do so a JOIN is neccessary. The view contains the name of persons in combination with the available contact information. As an INNER JOIN is used, some persons occur multiple, others not at all.

```
-- Persons and contacts
CREATE VIEW person_view_5 AS
  SELECT p.firstname, p.lastname, c.contact_type, c.contact_value
  FROM   person p
  JOIN   contact c ON p.id = c.person_id;

-- Verify result
SELECT *
FROM   person_view_5;

SELECT *
FROM   person_view_5
```

```
WHERE  lastname = 'Goldstein';
```

The columns *person.id* an *contact.person_id* are used during the definition of the view. But they are not part of the projection and hence not available for SELECTs to the view.

Hint: The syntax and semantic of join operations is explained on a separate page.

Some more Hints

Within a CREATE VIEW statement one may use more elements of the regular SELECT statement than it is shown on this page, especially: SET operations, recursive definitions, CASE expressions, ORDER BY and so on.

If there is an ORDER BY clause within the CREATE VIEW and another one in a SELECT to this view, the later one overwrites the former.

Write Access via Views

In some cases, but not in general, it should be possible to change data (UPDATE, INSERT or DELETE command) in a table by accessing it via a view. Assume, as an counterexample, that one wants to change the column *count_of_members* of *person_view_4* to a different value. What shall the DBMS do? The column is subject to an aggregate function which counts the number of existing rows in the underlying table. Shall it add some more random values into new rows respectively shall it delete random rows to satisfy the new value of *count_of_members*? Of course not!

On the other hand a very simple view like 'CREATE VIEW person_0 AS SELECT * from person;', which is an 1:1 copy of the original table, should be manageable by the DBMS. Where is the borderline between updateable and non updateable views? The SQL standard does not define it. But the concrete SQL implementations offer limited write-access to views based on their own rules. These rules sometimes are very fix, in other cases they consists of flexible techniques like 'INSTEAD OF' triggers to give programmers the chance to implement their own rules.

Here are some general rules which **may be part** of the implementors fixed rules to define, which views are updateable in his sens:

- The view definition is based on one and only one table. It includes the Primary Key of this underlying table.
- The view definition must not use any aggregate function.
- The view definition must not have any DISTINCT-, GROUP BY- or HAVING-clause.
- The view definition must not have any JOIN, SUBQUERY, SET operation, EXISTS or NOT EXISTS predicate.

If it is possible to use the UPDATE, INSERT or DELETE command to a view, the syntax is the same as with tables.

Clean up the Example Database

The DROP VIEW statement deletes a view definition. In doing so the data of the underlying table(s) is not affected.

Don't confuse the DROP command (definitions) with the DELETE command (data)!

```
DROP VIEW person_view_1;
DROP VIEW person_view_2a;
DROP VIEW person_view_2b;
DROP VIEW person_view_3;
DROP VIEW person_view_4;
DROP VIEW person_view_5;
```

Exercises

Create a view 'hobby_view_1' which contains all columns of table 'hobby' except 'id'.
Rename column 'remark' to 'explanation'. Create two different solutions.

Click to see solution

```
CREATE VIEW hobby_view_1a AS
  SELECT hobbyname, remark AS explanation
  FROM   hobby;
-- Verification
SELECT * FROM hobby_view_1a;

CREATE VIEW hobby_view_1b (hobbyname, explanation) AS
  SELECT hobbyname, remark
  FROM   hobby;
-- Verification
SELECT * FROM hobby_view_1b;
```

Create a view 'hobby_view_2' with the same criteria as in the previous example. The only difference is that the length of the explanation column is limited to 30 character. Hint: use the function substr(<column name>, 1, 30) to determine the first 30 characters - she is not part of the SQL standard but works in plenty implementation.

Click to see solution

```
CREATE VIEW hobby_view_2 AS
  SELECT hobbyname, substr(remark, 1, 30) AS explanation
  FROM   hobby;
-- Verification
SELECT * FROM hobby_view_2;
```

Create a view 'contact_view_3' which contains all rows of table contact with the exception of the 'icq' rows. Count the number of the view rows and compare it with the number of rows in the table 'contact'.

Click to see solution

```
CREATE VIEW contact_view_3 AS
  SELECT *
  FROM   contact
  WHERE  contact_type != 'icq';  -- an alternate operator with the same semantic as '!=' is '<>'

-- Verification
SELECT 'view',  count(*) FROM contact_view_3
  UNION
SELECT 'table', count(*) FROM contact;
```

Create a view 'contact_view_4' which contains one row per contact type with its notation and the number of occurences. Afterwords select those which occur more than once.

Click to see solution

```
CREATE VIEW contact_view_4 AS
  SELECT contact_type, count(*) AS cnt
  FROM   contact
  GROUP BY contact_type;

-- Verification
SELECT *
FROM   contact_view_4;

-- Use columns of a view with the same syntax as column of a table.
SELECT *
FROM   contact_view_4
WHERE  cnt > 2;
```

Create a view 'person_view_6' which contains first- and lastname of persons plus the number of persons with the same name as the person itself (family name). Hint: the solution uses a correlated subquery.

Click to see solution

```
CREATE VIEW person_view_6 AS
  SELECT firstname, lastname, (SELECT count(*) FROM person sq WHERE sq.lastname = p.lastname) AS cnt_family
  FROM   person p;

-- Verification
SELECT *
FROM   person_view_6;
```

Clean up the example database.

Click to see solution

```
DROP VIEW hobby_view_1a;
DROP VIEW hobby_view_1b;
DROP VIEW hobby_view_2;
DROP VIEW contact_view_3;
DROP VIEW contact_view_4;
DROP VIEW person_view_6;
```

Hint: Be carefull and deactivate AUTOCOMMIT.

The basic syntax and semantic of the INSERT command is described on the page INSERT. There are examples how to insert single rows with fixed values into a table. The present page describes how to dynamise the command by the use of subqueries.

Evaluate Values at Runtime

First, the values to be inserted may be evaluated in a relative strict way by reading the system time or other (quasi) constants.

```
-- Use the key word CURRENT_DATE to determine the actual day.
INSERT INTO person ( id,  firstname,        lastname,    date_of_birth,     place_of_birth, ssn,           weight)
VALUES             (101,  'Larry, no. 101', 'Goldstein', CURRENT_DATE,      'Dallas',       '078-05-1120', 95);
COMMIT;
```

Next, the values to be inserted may be evaluated by a scalar value subquery. This means, that single values may be computed at runtime based on the rows of the same or another table.

```
-- Count the number of rows to determine the next ID. Caution: This handling of IDs is absolutly NOT recommended for real applicatio
INSERT INTO person ( id,   firstname,        lastname,    date_of_birth,     place_of_birth, ssn,           weight)
VALUES             ((SELECT COUNT(*) + 1000 FROM person),  -- The scalar value subquery. It computes one single value, in this case
-- VALUES          ((Select * FROM (SELECT COUNT(*) + 1000 FROM person) tmp), -- MySQL insists in using an intermediate table
                           'Larry, no. ?',   'Goldstein', CURRENT_DATE,      'Dallas',       '078-05-1120', 95);
COMMIT;
```

Evaluate Rows at Runtime

Similar to the above shown evaluation of a single scalar value through a scalar value subquery one can use a table subquery to get several rows and insert them into the specified table within one INSERT command. This version is able to insert thousands of rows within one single statement. In addition to its dynamic nature it saves all but one round-trips between the application and the DBMS and therefore is much faster than a lot of single row-based INSERTs.

```
-- The statement doubles the number of rows within the table. It omits in the table subquery the WHERE clause and therefore
-- it reads all existing rows. Caution: This handling of IDs is absolutly NOT recommended for real applications!
INSERT INTO person (id,          firstname, lastname, date_of_birth, place_of_birth, ssn, weight)
  SELECT            id + 1100,   firstname, lastname, date_of_birth, place_of_birth, ssn, weight
  FROM   person;
COMMIT;
```

The syntax has change in such a way that the key word 'VALUES' with its list of values is replaced by a complete subquery (often named 'subselect') which starts with the key word 'SELECT'. Of course the number and type of the selected columns must correlate with the number and type of the columns of the specified column list behind the 'INSERT INTO' key word. Within the subquery the complete power of the SELECT statement may be used: JOIN, WHERE, GROUP BY, ORDER BY and especially other subqueries in a recursive manner. Therefore there is a wide range of use cases: create rows with increased version numbers, with percentage increased salary, with the actual timestamp, fixed values from rows of the same or another table, ...

```
-- The next two statements compute different weights depending on the old weight
INSERT INTO person (id,          firstname, lastname, date_of_birth, place_of_birth, ssn, weight)
  -- the subquery starts here
  SELECT            id + 1200,   firstname, lastname, date_of_birth, place_of_birth, ssn,
                    CASE WHEN weight < 40 THEN weight + 10
                         ELSE                  weight +  5
                         END
  FROM   person
  WHERE  id <= 10;                          -- only the original 10 rows from the example database
COMMIT;

-- The same semantic with a more complex syntax (to demonstrate the power of subselect)
INSERT INTO person (id,          firstname, lastname, date_of_birth, place_of_birth, ssn, weight)
  -- the first subquery starts here
  SELECT            id + 1300,   firstname, lastname, date_of_birth, place_of_birth, ssn,
                     -- here starts a subquery of the first subquery. The CASE construct evaluates different
                     -- weights depending on the old weight.
                    (SELECT CASE WHEN weight < 40 THEN weight + 10
                                 ELSE                  weight +  5
                                 END
                     FROM   person ssq      -- alias for the table name in sub-subquery
                     WHERE  sq.id = ssq.id  -- link the rows together
                    )
  FROM   person sq                          -- alias for the table name in subquery
  WHERE  id <= 10;                          -- only the original 10 rows from the example database
COMMIT;
```

The technique shown at Structured Query Language/Example_Database_Data#Grow_up which multiplies existing data, e.g. for testing purpose, is based on such table subqueries.

Clean up Your Database

```
DELETE FROM person WHERE id > 100;
COMMIT;
```

Exercises

Insert a new person with id 1301, firstname 'Mr. Mean', lastname is the lowest lastname (in the sense of the character encoding, use min() function). Its weight is the average weight of all persons (use avg() function).

Click to see solution

```
-- Two columns are computed during runtime
INSERT INTO person (id,   firstname,  lastname, weight)
VALUES             (1301,
                    'Mr. Mean',
                    (SELECT MIN(lastname) FROM person),
                    (SELECT AVG(weight)   FROM person)
-- the MySQL version with its intermediate tables
--                  (SELECT * FROM (SELECT MIN(lastname) FROM person) tmp1),
--                  (SELECT * FROM (SELECT AVG(weight)   FROM person) tmp2)
                   );
COMMIT;
-- Check your result
SELECT * FROM person WHERE id = 1301;
```

Insert one additional person per family (=lastname) with firstname 'An extraordinary family member', lastname is the family name. Incorporate only the rows from the original example database with id <= 10.

Click to see solution

```
-- Two columns are computed during runtime. The number of involved rows is delimited by the WHERE clause.
INSERT INTO person (id,   firstname,  lastname)
  -- here starts the subquery
  SELECT MAX(id) + 1310,  -- in this case the max() function works per group
         'An extraordinary family member',
         lastname
  FROM   person
  WHERE  id <= 10
  GROUP BY lastname;
COMMIT;
-- Check your result
SELECT * FROM person WHERE id > 1310;
```

Clean up your database.

Click to see solution

```
DELETE FROM person WHERE id > 1300;
COMMIT;
```

Hint: Be carefull and deactivate AUTOCOMMIT.

The page in hand offers two additional technics as an extention to the UPDATE command shown on one of the previous pages:

- Computing values, which are assigned to a column, at runtime.
- Using complex subqueries as search conditions in the WHERE clause.

Evaluate Values at Runtime

The values which are assigned to a column may be computed by a correlated or non-correlated scalar value subquery on the involved table or another one. There are many use cases where this technic is utilised: Increase values linear or in percentage, use values from the same or another table, ... The situation is similar to that described on the page about the INSERT command.

```
-- The average weight of all persons is stored in column 'weight' of the first four persons.
UPDATE person SET
  -- dynamic computation of a value
  weight = (SELECT AVG(weight) FROM person)
  --  weight = (SELECT * FROM (SELECT AVG(weight) FROM person) tmp) -- MySQL insists on using an intermediate table
WHERE  id < 5;

-- Check the result
SELECT * FROM person;
```

```
-- revoke the changes
ROLLBACK;
```

The subquery may use values of the row, which is actually updated. In the next example persons receive the mean weight of their family. To compute this mean weight, it is necessary to use the column 'lastname' of the actual processed row.

```
-- The subquery is a 'correlated' scalar value subquery.
UPDATE person p SET
  -- 'p.lastname' refers to the lastname of the actual row. The subquery bears all rows in mind, not only such with 'id >= 5'.
  weight = (SELECT AVG(weight) FROM person sq WHERE sq.lastname = p.lastname)
  -- A hint to MySQL users: MySQL does not support UPDATE in combination with a correlated subquery
  -- to the same table. Different tables work. MySQL has a different, non-standard concept: multi-table update.
WHERE  id >= 5;

-- Check the result
SELECT * FROM person;

-- revoke the changes
ROLLBACK;
```

Subqueries in WHERE Clause

The WHERE clause determines which rows of a table are involved by the UPDATE command. This WHERE clause has the same syntax and semantic as the WHERE clause of the SELECT or DELETE command. It may contain complex combinations of boolean operators, predicates like ANY, ALL or EXISTS and - in a recusive manner - subquerys as described in SELECT: Subquery.

```
-- UPDATE rows in the table 'person'. The decision which rows are affected is made by consulting the table 'contact'.
-- In the example persons with more than 2 contact information are affected.
UPDATE person
SET    firstname = 'Has many buddies'
WHERE  id IN
  (SELECT person_id
   FROM   contact
   GROUP BY person_id
   HAVING count(*) > 2
  );

-- Check the result
SELECT * FROM person;

-- revoke the changes
ROLLBACK;
```

The command performs an UPDATE in the table *person*, but the affected rows are identified by a subquery in table *contact*. This technique of grabbing information from other tables offers very flexible strategies to modify the data.

It is no error to select 0 rows in the subquery. In this case the DBMS executes the UPDATE command as usual and throws no exception. (The same holds true for subqueries in SELECT or DELETE statements.)

Exercises

Assign the firstname 'Short firstname' to all persons which have a firstname with less than 5 characters.

Click to see solution

```
-- Hint: Some implementations use a different function name: length() or len().
UPDATE person
SET    firstname = 'Short firstname'
WHERE  character_length(firstname) < 5;

-- Check the result
SELECT * FROM person;

-- revoke the changes
ROLLBACK;
```

Assign the firstname 'No hobby' to all persons which have no hobby.

Click to see solution

```
UPDATE person
SET    firstname = 'No hobby'
WHERE  id NOT IN
  (SELECT person_id
```

```
   FROM   person_hobby
  );

-- Check the result
SELECT * FROM person;

-- revoke the changes
ROLLBACK;
```

Assign the firstname 'Sportsman' to all persons performing one of the hobbies 'Underwater Diving' or 'Yoga'.

Click to see solution

```
UPDATE person
SET    firstname = 'Sportsman'
WHERE  id IN
  -- The subquery must join to the table 'hobby' to see their column 'hobbyname'.
  (SELECT ph.person_id
   FROM   person_hobby ph
   JOIN   hobby        h  ON ph.hobby_id = h.id
   AND    h.hobbyname IN ('Underwater Diving', 'Yoga')
  );

-- Check the result
SELECT * FROM person;

-- revoke the changes
ROLLBACK;
```

Hint: Be carefull and deactivate AUTOCOMMIT.

In many cases applications want to store rows in the database without knowing whether this rows previously exist in the database or not. If the rows exist, they must use the UPDATE command, if not, the INSERT command. To do so the following construct is often used:

```
-- pseudocode
IF (SELECT COUNT(*) = 0 ...) THEN
  INSERT ...
ELSE
  UPDATE ...
;
```

This situation is unpleasant in many ways:

- There are two roundtrips between application and DBMS, either SELECT + INSERT or SELECT + UPDATE.
- The application must transfer one row after the other. A 'bulk storing' is not possible because the evaluation of the criterion which decides between INSERT and UPDATE may lead to different results from row to row.
- The syntax is spread across three SQL statements. This is error-prone.

To overcome the disadvantages the SQL standard defines a MERGE command, which contains the complete code shown above in one single statement. The MERGE performs an INSERT or an UPDATE depending on the existence of individual rows at the target table.

```
-- Define target, source, match criterion, INSERT and UPDATE within one single command
MERGE INTO  <target_table>      <target_table_alias>  -- denote the target table
      USING <source_table>      <source_table_alias>  -- denote the source table
      ON    (<match_criterion>)                       -- define the 'match criterion' which compares the source and
                                                      -- target rows with the same syntax as in any WHERE clause
  WHEN MATCHED THEN
UPDATE SET column1 = value1 [, column2 = value2 ...]  -- a variant of the regular UPDATE command
  WHEN NOT MATCHED THEN
INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...]) -- a variant of the regular INSERT command
;
```

Description

The target table is named after the MERGE INTO key word, the source table after the USING key word.

The comparision between target rows and source rows, which is necessary to decide between INSERT and UPDATE, is specified after the ON key word with a syntax, which is identical to the syntax of a WHERE clause. If this comparision matches, the UPDATE will be performed, else the INSERT. In simple cases the comparision compares Primary Key or Foreign Key columns. But it is also possible to use very sophisticated conditions on any column.

In the 'MATCHED' case a variant of the UPDATE follows. It differs from the regular UPDATE command in that it has no table name (the

table name is already denoted after the MERGE INTO) and no WHERE clause (it uses the match criterion after the ON key word).

In the 'NOT MATCHED' case a variant of the INSERT follows. For the same reason as before the target table is not named within the INSERT.

Example

Create a table 'hobby_shadow' to store some of the 'hobby' rows. The subsequent MERGE command shall perform an INSERT or an UPDATE depending on the existence of correlating rows.

```
-- store every second row in a new table 'hobby_shadow'
CREATE TABLE hobby_shadow AS SELECT * FROM hobby where MOD(id, 2) = 0;
SELECT * FROM hobby_shadow;

-- INSERT / UPDATE depending on the column 'id'.
MERGE INTO  hobby_shadow                  t   -- the target
      USING (SELECT id, hobbyname, remark
             FROM   hobby)                s   -- the source
      ON    (t.id = s.id)                     -- the 'match criterion'
  WHEN MATCHED THEN
UPDATE SET remark = concat(s.remark, ' Merge / Update')
  WHEN NOT MATCHED THEN
INSERT (id, hobbyname, remark) VALUES (s.id, s.hobbyname, concat(s.remark, ' Merge / Insert'))
;
COMMIT;

-- Check the result
SELECT * FROM hobby_shadow;
```

The MERGE command handls all rows, but there is only 1 roundtrip between the application and the DBMS. Some of the rows are handled by the INSERT part of MERGE, others by its UPDATE part. This distinction may be observed by the last part of the column 'remark'.

Use Case

Typical use cases for the MERGE command are ETL processes. Often those processes have to aggregate some values for a grouping criterion (eg: a product line) over a time period. The first access per product line and period has to insert new rows with given values, subsequent accesses have to update them by increasing values.

Extentions

The SQL standard defines some more features within the MERGE command.

WHEN clause
The WHEN MATCHED and WHEN NOT MATCHED clauses may be extended by an optional query expression like `AND (place_of_birth = 'Dallas')`. As a consequence, it's possible to use a series of WHEN MATCHED / WHEN NOT MATCHED clauses.

```
...
  WHEN MATCHED AND (t.hobby_name IN ('Fishing', 'Underwater Diving')) THEN
UPDATE SET remark = concat('Water sports: ', t.remark)
  WHEN MATCHED AND (t.hobby_name IN ('Astronomy', 'Microscopy', 'Literature')) THEN
UPDATE SET remark = concat('Semi-professional leisure activity: ', t.remark)
  WHEN MATCHED THEN
UPDATE SET remark = concat('Leisure activity: ', t.remark)
...
-- The same is possible with WHEN NOT MATCHED in combination with INSERT
```

DELETE
Within a WHEN MATCHED clause it is possible to use a DELETE command instead of an UPDATE to remove the matched row. This feature may be combined with the previous presented extension by an optional query expression. In the SQL standard the DELETE command is not applicable to the WHEN NOT MATCHED clause.

```
-- Update 'Fishing' and 'Underwater Diving'. Delete all others which have a match between source and target.
...
  WHEN MATCHED AND (t.hobby_name IN ('Fishing', 'Underwater Diving')) THEN
UPDATE SET remark = concat('Water sports: ', t.remark)
  WHEN MATCHED THEN
DELETE
...
```

Caveat

The MERGE command is clearly defined by standard SQL. The command itself as well as the extensions described before are

implemented by a lot of DBMS. Deviating from the standard most implementations unfortunatelly use different and/or additional keywords and - sometimes - different concepts. Even the introductive key words MERGE INTO may differ from the standard.

Exercises

A) Create a new table 'contact_merge' with the same structure as 'contact'.
B) Copy row number 3 from 'contact' to 'contact_merge'.
C) Use the MERGE command to insert/update all E-Mail-adresses from 'contact' to 'contact_merge' and add the e-mail-protocol name to the contact values (prepend column contact_value by the string 'mailto:').

Click to see solution

```
-- Create table and copy one row
CREATE TABLE contact_merge AS SELECT * FROM contact WHERE id = 3;
SELECT * FROM contact_merge;

-- INSERT / UPDATE depending on the column 'id'.
MERGE INTO  contact_merge                              t   -- the target
      USING (SELECT id, person_id, contact_type, contact_value
             FROM   contact
             WHERE  contact_type = 'email')            s   -- the source
      ON    (t.id = s.id)                                  -- the 'match criterion'
  WHEN MATCHED THEN
UPDATE SET contact_value = concat('mailto:', t.contact_value)
  WHEN NOT MATCHED THEN
INSERT (id, person_id, contact_type, contact_value) VALUES (s.id, s.person_id, s.contact_type, concat('mailto:', s.contact_value))
;
COMMIT;

-- Check the result
SELECT * FROM contact_merge;
```

Hint: Be carefull and deactivate AUTOCOMMIT.

Because the DELETE command deletes rows as a whole and not partly, the syntax is very simple. Its structure was shown on a previous page. The page on hand offers only one addition: The WHERE clause isn't limited to simple conditions like 'id = 10' but may contain a subquery. This gives the command much more flexibility.

The use of subqueries as part of a DELETE command is identical to its use within an UPDATE or SELECT command.

There is another command for the deletion of rows. The TRUNCATE command is very similar to DELETE. TRUNCATE deletes **all** rows of a table and shows better performance. But it has no mechanism to choose individual rows.

Example

The example command deletes contact information from persons which are born in San Francisco.

```
-- Delete rows depending on a criteria which resides in a different table.
DELETE FROM contact
WHERE  person_id IN
  (SELECT id
   FROM   person
   WHERE  place_of_birth = 'San Francisco'
  );

-- It's only a test. Restore the rows.
ROLLBACK;
```

Correlated subqueries in combination with DELETE commands are not supported by all implementations.

It often happens that the DBMS rejects DELETE commands because Foreign Key constraints will be violated during its execution. E.g.: if the command tries to delete a person to whom a contact or hobby information is known, the command fails (as a whole). To overcame such situations there are different strategies:

- Delete all dependent rows prior to the intended row.
- Define the Foreing Key constraint as DEFERRED (it will be check not before COMMIT) and delete the depending rows before or after the intended one.
- Define the Foreing Key constraint as CASCADE. In this case the depending rows will be deleted automatically.

Exercise

Delete hobby information for family Goldstein.

Click to see solution

```
DELETE FROM person_hobby
WHERE  person_id IN
  (SELECT id
   FROM   person
   WHERE  lastname = 'Goldstein'
  );

-- Refrain from deleting the hobby itself - because:
--   a) The hobby may be allocated to a different person.
--   b) After the information in person_hobby is deleted, there is no longer the possibility to get
--      to old assignment between person and hobby.

-- It's only a test. Restore the rows.
ROLLBACK;
```

The TRUNCATE TABLE command deletes **all** rows of a table without causing any triggered action. Unlike the DELETE command it contains no WHERE clause to specify individual rows.

With respect to the TRUNCATE TABLE command most DBMS show significant better performance than with DELETE command. This results from the facts the DBMS can empty the table (and its indexes) as a whole. It's not necessary to access individual rows.

- There is - per definition - no WHERE clause.
- No trigger action will be launched - per definition.
- The transaction locks the complete table.
- If there is an FK-constraint from table *t1* to *t2*, the command `'TRUNCATE TABLE t2'` will fail. This holds true independent from the question whether any row of *t1* refers actually to one of the rows of *t2* or not. The DBMS checks only the existence of the FK-constraint definition.

The syntax of the TRUNCATE TABLE command is very simple.

```
TRUNCATE TABLE <tablename>;
```

Example

```
-- Delete ALL rows of the table 'myTable'
TRUNCATE TABLE myTable;
-- In most DBMS ROLLBACK is not possible - in opposite to situations with a DELETE command.
```

An Analogy

To illustrate the difference between the TRUNCATE TABLE command and the DELETE command (without a WHERE clause) one can imagine a trucker, who wants to empty a trailer full of sand at a construction site. To do so he has two possibilities. Either he empties the trailer in that he tilts him - this corresponds to the TRUNCATE TABLE command. Or he climbs onto the trailer and throws down one grain of sand after the next - this corresponds to the DELETE command.

Exercises

Delete all rows of table 'person_hobby' using the DELETE command.
Verify that there are no rows left in 'person_hobby'.
Delete all rows of table 'hobby' using the TRUNCATE TABLE command.
What will happen? (Consider that there is an FK constraint from the table empty 'person_hobby' to 'hobby'.)

Click to see solution

```
-- Delete all rows of 'person_hobby' with a DELETE command
DELETE FROM person_hobby;
COMMIT;

-- Are there any rows?
SELECT count(*) FROM person_hobby;

-- Try TRUNCATE TABLE command:
TRUNCATE TABLE hobby;
-- An exception will be thrown. Although there is no row in 'person_hobby' refering a row in 'hobby',
-- the definition of the FK constraint exists. This is the reason for the exception.
```

What will happen in the above example, if the TRUNCATE TABLE command is replaced by a DELETE command?

Click to see solution

```
-- As there is no row in 'person_hobby' refering to 'hobby', the DELETE command deletes all rows in 'hobby'.
DELETE FROM hobby;
COMMIT;
```

The original data of the example database can be reconstructed as shown on the example database data page.

Advanced Topics

One of the basic steps during database development cycles is the fixing of decisions about the table structure. To do so there is the CREATE TABLE statement with which developers define tables together with their columns and constraints.

Because a lot of features may be activated by the command, its syntax is a little bit complex. This page shows the most important parts. The syntax is not straight forward. At some points it is possible to use alternative formulations to express the same purpose, e.g. the Primary Key may be defined within the column definition as a column constraint, at the end of the command as a table constraint or as a separate stand-alone command 'ALTER TABLE ADD CONSTRAINT ...;' .

```
CREATE TABLE <tablename> (
  <column_name> <data_type> <default_value> <identity_specification> <column_constraint>,
  <column_name> <data_type> <default_value>                          <column_constraint>,
  ...,
  <table_constraint>,
  <table_constraint>,
  ...
);
```

General Description

After the introductory key words CREATE TABLE the tablename is specified. Within a pair of parentheses a list of column definitions follows. Each column is defined by its name, data type, an optional default value and optional constraints for this individual column.

After the list of column definitions developers can specifiy table constraints like Primary and Foreign Keys, Unique conditions and general column conditions.

An first example was shown at the page Create a simple Table and a second one here:

```
CREATE TABLE test_table (
  -- define columns (name / type / default value / column constraint
  id              DECIMAL                          PRIMARY KEY,
  part_number     CHAR(10)          DEFAULT 'n/a'  NOT NULL,
  part_name       VARCHAR(500),
  state           DECIMAL           DEFAULT -1,
  -- define table constraints (eg: 'n/a' shall correlate with NULL)
  CONSTRAINT test_check CHECK ((part_number  = 'n/a' AND part_name IS     NULL) OR
                               (part_number != 'n/a' AND part_name IS NOT NULL))
);
```

The table consists of 4 columns. All of them have a data type and some a default value. The column *id* acts as the Primary Key. The table constraint *test_check* guarantees that *part_name* is mandatory if *part_number* is recorded.

Column Definition

Data Type

The standard defines a lot of predefined data types: character strings of fixed and variable size, character large objects (CLOB), binary strings of fixed and variable size, binary large objects (BLOB), numeric, boolean, datetime, interval, xml. Beyond there are complex types like: ROW, REF(erence), ARRAY, MULTISET and user-definded types (UDT). The predefined data types are explained on the next page. To keep things simple we use on this page only CHAR, VARCHAR and DECIMAL.

Default Value

A column can have a default value. Its data type corresponds to the type of the column. It may be a constant value like the number -1 or the string 'n/a', or it is a system variable or a function call to determine dynamic values like the username or the actual timestamp.

The default clause affects those INSERT and MERGE commands, which do not specify the column. In our example database the *person*

table has the column *weight* with the default value 0. If we omit this column in an INSERT command, the DBMS will store the value 0.

```
-- This INSERT command omits the 'weight' column. Therefore the value '0' (which is different from
-- the NULL value) is stored in the weight column.
INSERT INTO person (id, firstname, lastname,    date_of_birth,     place_of_birth, ssn)
VALUES             (11, 'Larry',   'Goldstein', date'1970-11-20', 'Dallas',        '078-05-1120');
COMMIT;

-- This SELECT retrieves the row ...
SELECT *
FROM   person
WHERE  id = 11
AND    weight = 0;

-- ... but not this one:
SELECT *
FROM   person
WHERE  id = 11
AND    weight IS NULL;
```

Identity Specification

The *identity specification* serves for the generation of a series of unique values which acts as the Primary Key to the tables rows. The standard defines the syntax as: "GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY". Unfortunatelly most DBMS vendors do not support this formulation. Instead they offer different syntaxes and even different concepts to generate primary key values. Some use a combination of generators/sequences and triggers, others a special data type or different key words.

An overview about the wide spread of implementations is available in the wikibook SQL Dialects Reference: Auto-increment_column.

Column Constraint

The column constraint clause specifies conditions which all values must meet. There are different column constraint types:

NOT NULL
Primary Key
Unique
Foreign Key
Check values

The NOT NULL phrase defines, that it is not allowed to store the NULL value in the column.

```
-- The column col_1 is per definition not allowed to hold the NULL value
CREATE TABLE t1 (col_1 DECIMAL NOT NULL);

-- This INSERT command will fail
INSERT INTO t1(col_1) values(NULL);

-- The same applies to the following UPDATE command
INSERT INTO t1(col_1) values(5);
UPDATE t1 SET col_1 = NULL;
```

The PRIMARY KEY phrase defines that the column acts as the Primary Key of the table. This implies that the column is not allowed to store a NULL value and that the values of all rows are distinct from each other.

```
CREATE TABLE t2 (col_1 DECIMAL PRIMARY KEY);

-- This INSERT will fail because a primary key column is not allowed to store the NULL value.
INSERT INTO t2(col_1) VALUES(NULL);

-- This INSERT works
INSERT INTO t2(col_1) VALUES(5);

-- But the next INSERT will fail, because only one row with the value '5' is allowed.
INSERT INTO t2(col_1) VALUES(5);
```

The UNIQUE constraint has a similar meaning as the PRIMARY KEY phrase. But there are two slight differences.

First, the values of different rows of a UNIQUE column are not allowed to be equal, which is the same as with PK. But they are allowed to hold the NULL value, which is different from PK. The existence of NULL values has an implication. As the term *null* = *null* never evaluates to *true* (it evaluates to *unknown*) there may exist multiple rows with the NULL value in a column which is defined to be UNIQUE.

Second, only one PK definition per table is allowed. In contrast, there may be many UNIQUE constraints (on different columns).

```
CREATE TABLE t3 (col_1 DECIMAL UNIQUE);

-- works well
INSERT INTO t3(col_1) VALUES(5);
-- fails because there is another row with value 5
INSERT INTO t3(col_1) VALUES(5);

-- works well
INSERT INTO t3(col_1) VALUES(null);
-- works also
INSERT INTO t3(col_1) VALUES(null);

-- check the results
SELECT * FROM t3;
```

The FOREIGN KEY condition defines that the column can hold only those values, which are also stored in a different column of (the same or) another table. This different column has to be UNIQUE or a Primary Key, whereas the values of the foreign key column itself may hold identical values for multiple rows. The consequence is that one cannot create a row with a certain value in this column before there is a row with exactly this certain value in the referred table. In our example database we have a *contact* table whose column *person_id* refers to the id of persons. It makes sense that one cannot store contact values before storing the appropriate person.

Foreign Keys are the technique to realise 1:m relationships.

```
-- A table with a column which referes to the 'id' column of table 'person'
CREATE TABLE t4 (col_1 DECIMAL REFERENCES person(id));

-- This INSERT works as in table 'person' of our example database there is a row with id = 3.
INSERT INTO t4(col_1) VALUES(3);

-- This statement will fail as in 'person' there is no row with id = 99.
INSERT INTO t4(col_1) VALUES(99);
```

Column checks inspect the values of the column to see whether they meet the defined criterion. Within such column checks only the actual column is visible. If a condition covers two or more colums (eg.: col_1 > col_2) a table check must be used.

```
-- 'col_1' shall contain only values from 1 to 10.
-- A hint to MySQL users: MySQL accepts the syntax of column checks - but it ignores them silently.
CREATE TABLE t5 (col_1 DECIMAL CHECK (col_1 BETWEEN 1 AND 10));

-- This INSERT works:
INSERT INTO t5(col_1) VALUES(3);

-- This statement will fail:
INSERT INTO t5(col_1) VALUES(99);
```

Table Constraint

Table constraints defines rules which are mandatory for the table as a whole. Their sematic and syntax overlaps partially with the previous shown column constraints.

Table constraints are defined after the definition of all columns. The syntax starts with the key word CONSTRAINT and includes the possibility to denominate them with a meaningful name, *t6_pk*, *t6_uk* and *t6_fk* in the next example. In the case of any exception most DBMS shows this name as part of the error message - and if you havn't defined one it uses its internal naming conventions which may be very cryptic.

Primary Key, UNIQUE and Foreign Key

In the same manner as shown in the column constraints part Primary Key, UNIQUE and Foreign Key conditions can be expressed as table constraints. The syntax differs slightly from the column constraint syntax, the semantic is identical.

```
-- A table with a PK column, one UNIQUE column and a FK column.
CREATE TABLE t6 (
  col_1 DECIMAL,
  col_2 CHAR(10),
  col_3 DECIMAL,
  CONSTRAINT t6_pk PRIMARY KEY (col_1), -- 't6_pk' is the name of the constraint
  CONSTRAINT t6_uk UNIQUE      (col_2),
  CONSTRAINT t6_fk FOREIGN KEY (col_3) REFERENCES person(id)
);
```

NOT NULL and Simple Column Checks

In a similar way as shown in the column constraints part NOT NULL conditions and simple column checks can be expressed as table expressions.

```
CREATE TABLE t7 (
  col_1 DECIMAL,
  col_2 DECIMAL,
  CONSTRAINT t7_col_1_nn    CHECK (col_1 IS NOT NULL),
  CONSTRAINT t7_col_2_check CHECK (col_2 BETWEEN 1 and 10)
);
```

General Column Checks

If a condition affects more than one column it must be expressed as a table constraint.

```
CREATE TABLE t8 (
  col_1 DECIMAL,
  col_2 DECIMAL,
  col_3 DECIMAL,
  col_4 DECIMAL,
  -- col_1 can hold only those values which are greater than col_2
  CONSTRAINT t8_check_1 CHECK (col_1 > col_2),
  -- If col_3 is NULL, col_4 must be NULL also
  CONSTRAINT t8_check_2 CHECK ((col_3 IS     NULL AND col_4 IS     NULL) OR
                               (col_3 IS NOT NULL AND col_4 IS NOT NULL))
);

-- This two INSERTs work as they meet all conditions
INSERT INTO t8 VALUES(1, 0, null, null);
INSERT INTO t8 VALUES(2, 0, 5, 5);

-- Again: MySQL ignores check conditions silently

-- This INSERT fails because col_1 is not greater than col_2
INSERT INTO t8 VALUES(3, 6, null, null);

-- This INSERT fails because col_3 is not null and col_4 is null
INSERT INTO t8 VALUES(4, 0, 5, null);
```

Column Constraints vs. Table Constraints

As you have seen some constraints may be defined as part of the column definition, which is called a *column constraint*, or as a separate *table constraint*. Table constraints have two advantages. First, they are a little bit more powerful.

Second, they do have their own name! This helps to understand system messages. Furthermore it opens the possibility to manage constraints after the table exists and contains data. The ALTER TABLE statement can deactivate, activate or delete constraints. To do so, you have to know their name.

Clean Up

```
DROP TABLE t1;
DROP TABLE t2;
DROP TABLE t3;
DROP TABLE t4;
DROP TABLE t5;
DROP TABLE t6;
DROP TABLE t7;
DROP TABLE t8;
```

Exercises

Create a table 'company' with columns 'id' (numeric, primary key), 'name' (strings of variable size up to 200), 'isin' (strings of length 12, not nullable, unique values).
Create a solution with column constraints only and another one with table constraints only.

Click to see solution

```
-- column constraints only
CREATE TABLE company_1 (
  id   DECIMAL PRIMARY KEY,
  name VARCHAR(200),
  isin CHAR(12) NOT NULL UNIQUE
);
-- table constraints only
CREATE TABLE company_2 (
  id   DECIMAL,
  name VARCHAR(200),
  isin CHAR(5),
  CONSTRAINT company_2_pk PRIMARY KEY (id),
  CONSTRAINT company_2_uk UNIQUE      (isin),
  CONSTRAINT company_2_check_isin CHECK (isin IS NOT NULL)
);
```

Create a table 'accessory' with columns 'id' (numeric, primary key), 'name' (strings of variable size up to 200, unique), 'hobby_id' (decimal, not nullable, foreing key to column 'id' of table 'hobby').
Create a solution with column constraints only and another one with table constraints only.

Click to see solution

```
-- column constraints only
CREATE TABLE accessory_1 (
  id       DECIMAL PRIMARY KEY,
  name     VARCHAR(200) UNIQUE,
  hobby_id DECIMAL NOT NULL REFERENCES hobby(id)
);
-- table constraints only
CREATE TABLE accessory_2 (
  id       DECIMAL,
  name     VARCHAR(200),
  hobby_id DECIMAL,
  CONSTRAINT accessory_2_pk PRIMARY KEY (id),
  CONSTRAINT accessory_2_uk UNIQUE      (name),
  CONSTRAINT accessory_2_check_1  CHECK (hobby_id IS NOT NULL),
  CONSTRAINT accessory_2_fk FOREIGN KEY (hobby_id) REFERENCES hobby(id)
);

-- Test some legal and illegal values
INSERT INTO accessory_1 VALUES (1, 'Fishing-rod', 2);
COMMIT;
-- ...
```

The SQL standard knows three kinds of data types

- predefined data types
- constructed types
- user-defined types.

This page presents only the ***predefined data types***. *Constructed types* are one of ARRAY, MULTISET, REF(erence) or ROW. *User-defined types* are comparable to classes in object-oriented language with their own constructors, observers, mutators, methods, inheritance, overloading, overwriting, interfaces and so on.

Overview

The standard groups predefined data types into types with similar characteristics.

- Character Types
  - Character (CHAR)
  - Character Varying (VARCHAR)
  - Character Large Object (CLOB)
- Binary Types
  - Binary (BINARY)
  - Binary Varying (VARBINARY)
  - Binary Large Object (BLOB)
- Numeric Types
  - Exact Numeric Types (NUMERIC, DECIMAL, SMALLINT, INTEGER, BIGINT)
  - Approximate Numeric Types (FLOAT, REAL, DOUBLE PRECISION)
- Datetime Types (DATE, TIME, TIMESTAMP)
- Interval Type (INTERVAL)
- Boolean
- XML

Character types hold printable characters, binary types any binary data. Both may have a fixed or variable size with an upper limit. If the upper limit exeeds a certain value the type is a 'large object' with special methods and functions.

Exact numeric types hold numeric values without digits after the decimal or with a firm number of digits after the decimal. Please note that the standard does not define a separate **data type** 'auto-increment' for generating primary keys. Instead he defines the phrase 'GENERATED ALWAYS AS IDENTITY' as part of the CREATE TABLE statement, see CREATE TABLE statement or auto-increment-columns.

Approximate numeric types hold numeric values with an implementation defined precision (after the decimal).

Temporal types hold values for INTERVAL (a certain range on the time bar), DATE (year, month, day), TIME with and without TIMEZONE (name of timezone, hour, minute, second including fraction) and TIMESTAMP with and without TIMEZONE (name of timezone, year to second including fraction).

The boolean data type holds the two values *true* and *false*.

Part 14 of the SQL standard extends the list of predefined data types by introducing the data type XML (Oracle calls it XMLType) together with a bunch of particular functions. Columns of this type hold XML instances.

In the outdated SQL-2 standard there was a data type 'BIT'. This data type is no longer part of the standard.

Most DBMS implement the majority of predefined data types, but there are some exceptions. Also the naming differs slightly. An overview about the major implementations is available in the wikibook SQL_Dialects_Reference.

Data types are used within the CREATE TABLE statement as part of column definitions - or during CAST operations.

```
CREATE TABLE <tablename> (
  <column_name> <data_type> ... ,
  <column_name> <data_type> ... ,
  ...
);
```

Character

A series of printable characters - which is a string - can be stored within *character string types*. If all rows of a table use the same fixed size for the strings, the data type is `CHAR(<n>)` where <n> is the size of the strings. If the size varies from row to row, the data type `VARCHAR(<n>)` defines that **up to** <n> characters can be stored in the column. So <n> defines the upper limit for this column. The maximum value for <n> depends on the used DBMS implementation. If applications need to store longer strings than it is allowed by this upper system limit, the data type `CLOB` must be used. Also `CLOB` has its own upper limit, but this is significantly greater than the upper limit of `VARCHAR`.

```
-- A table with columns of fixed and variable size strings and a CLOB string
CREATE TABLE datatypes_1 (
  id    DECIMAL PRIMARY KEY,
  col_1 CHAR(10),        -- exactly 10 characters
  col_2 VARCHAR(150),    -- up to 150 characters
  col_3 CLOB             -- very large strings (MySQL denotes this data type: 'LONGTEXT')
);
```

Hint: Unlike other programming languages SQL does not distinguish between a *character data type* and a *string data type*. It knows only the *character string data types* CHAR, VARCHAR and CLOB.

Binary

Binary data types are similar to character data types. They differ in that they accept a different range of bytes. Binary data types accept all values.

```
-- A table with columns of fixed and variable size binary data and a BLOB
CREATE TABLE datatypes_2 (
  id    DECIMAL PRIMARY KEY,
  col_1 BINARY(10),      -- exactly 10 byte
  col_2 VARBINARY(150), -- up to 150 byte
  col_3 BLOB             -- very large data: jpeg, mp3, ...
);
```

A hint to Oracle users: The data type `BINARY` is not supported, the data type `VARBINARY` is denoted as `RAW` and is deprecated. Oracle recommends the use of `BLOB`.

Exact Numeric

Exact numeric types hold numeric values without digits after the decimal or with a firm number of digits after the decimal. All exact numeric types are signed.

`NUMERIC(<p>,<s>)` and `DECIMAL(<p>,<s>)` denotes two types which are nearly the same. <p> (precision) defines a fix number of all digits within the type and <s> (scale) defines how much of them resides behind the decimal place. Numeric values with more than (p - s) digits before the decimal place cannot be stored and numeric values with more than s digits after the decimal place are truncated to s digits after the decimal place. p and s are optional. It must always be: $p \geq s \geq 0$ and $p > 0$.

SMALLINT, INTEGER and BIGINT denotes data types without a decimal place. The SQL standard did not define their size, but the size of SMALLINT shall be smaller than the size of INTEGER and the size of INTEGER shall be smaller than the size of BIGINT.

```
-- A table using five exact numeric data types
CREATE TABLE datatypes_3 (
  id    DECIMAL PRIMARY KEY,
  col_1 DECIMAL(5,2),    -- three digits before the decimal and two behind
  col_2 SMALLINT,        -- no decimal point
  col_3 INTEGER,         -- no decimal point
  col_4 BIGINT           -- no decimal point. (Not supported by Oracle.)
);
```

Approximate Numeric

Approximate numeric types hold numeric values with an implementation defined precision (after the decimal). All approximate numeric types are signed. Their primary use cases are scientific computations.

There are three types: `FLOAT (<p>)`, `REAL` and `DOUBLE PRECISION`, where p denotes the guaranteed precision of the `FLOAT` data type. The precision of `REAL` and `DOUBLE PRECISION` is implementation defined.

```
-- A table using the approximate numeric data types
CREATE TABLE datatypes_4 (
  id    DECIMAL PRIMARY KEY,
  col_1 FLOAT(2),     -- two or more digits after the decimal place
  col_2 REAL,
  col_3 DOUBLE PRECISION
);
```

Temporal

Data types with respect to temporal aspects are: `DATE, TIME, TIMESTAMP` and `INTERVAL`.

`DATE` stores year, month and day. `TIME` stores hour, minute and second. `TIMESTAMP` stores year, month, day, hour, minute and second. Seconds can contain digits after the decimal. `TIME` and `TIMESTAMP` can contain the name of a TIME ZONE.

The SQL standard defines two kinds of INTERVALs. The first one is an interval with year and month, the second one is an interval with day, hour, minute and second.

```
-- A table using temporal data types
CREATE TABLE datatypes_5 (
  id    DECIMAL PRIMARY KEY,
  col_1 DATE,                       -- store year, month and day (Oracle: plus hour, minute and seconds)
  col_2 TIME,
  col_3 TIMESTAMP(9),               -- a timestamp with 9 digits after the decimal of seconds
  col_4 TIMESTAMP WITH TIME ZONE,   -- a timestamp including the name of a timezone
  col_5 INTERVAL YEAR TO MONTH,
  col_6 INTERVAL DAY TO SECOND(6)   -- an intervall with 6 digits after the decimal of seconds
);
```

A hint to Oracle users: The data type `TIME` is not supported. Use `DATE` instead.

A hint to MySQL users: The use of TIME ZONE as part of data types is not supported. MySQL implements a different concept to handle time zones. Fractions of seconds are not supported. The data type `INTERVAL` is not supported, but there is a data value `INTERVAL`.

Boolean

SQL has a 3-value-logic. It knows the boolean values `true`, `false` and `unknown`. Columns of the boolean data type can store one of the two values `true` or `false`. `unknown` is represented by storing no value, which is the NULL indicator.

```
-- A table with one column of boolean
CREATE TABLE datatypes_6 (
  id    DECIMAL PRIMARY KEY,
  col_1 BOOLEAN     -- not supported by Oracle
);
```

XML

Part 14 of the SQL standard extends the list of predefined data types by introducing the data type XML. The standard also defines a wide range of particular functions for this data type.

```
-- A table with one column of data type XML
CREATE TABLE datatypes_7 (
  id    DECIMAL PRIMARY KEY,
  col_1 XML
);
```

A hint to Oracle users: The data type `XML` is denoted as `XMLType`.

A hint to MySQL users: The data type `XML` is not supported.

Domains

In the context of data types the standard knows *domains*. The purpose of domains is to constrain the set of valid values that can be stored in a column. The domain-concept is a very early predecessor of user-defined types and may be outdated.

Clean Up

```
DROP TABLE datatypes_1;
DROP TABLE datatypes_2;
DROP TABLE datatypes_3;
DROP TABLE datatypes_4;
DROP TABLE datatypes_5;
DROP TABLE datatypes_6;
DROP TABLE datatypes_7;
```

Exercises

Create a table 'company' with columns 'id' (numeric, primary key), 'name' (strings of variable size up to 200), 'isin' (strings of length 12), 'stock_price' (numeric with 2 digits before and 2 after the decimal), 'description_text' (a very large string) and description_doc (any binary format).

Click to see solution

```
CREATE TABLE company (
  id               DECIMAL PRIMARY KEY,
  name             VARCHAR(200),
  isin             CHAR(12),
  stock_price      DECIMAL(4,2),
  description_text CLOB,
  description_doc  BLOB
);
```

Foreign Keys define a directed reference from one table (the child) to another table (the parent). This reference acts as long as the involved columns of the two tables contain identical values. It couples one row of the child table to a single row of the parent table - a row of the parent table may be coupled by many rows of the child table.

E.g.: You may have the table *department* with column *id* and the table *employee* with column *dept_id*. If you want to assign an employee to a distinct department, you store the department-id in its column *dept_id*. This can be done in every case - independent from any Foreign Key definition. But in such cases people often have two additional requirements: First, employees shall only be assigned to departments which really exist. Second, as long as employees are assigned to a distinct department, it shall be impossible to delete this department. The main purpose of Foreign Keys is to guarantee these two requirements.

In other words: Foreign Keys guarantee that **no orphans** will arise.

Foreign Key vs. Join

Within RDBMs identical values are used to link rows of different - and sometimes of the same - table together. Because this linking works on the basis of values and not of any link or special reference it has no direction. In general we call this technique a **JOIN**. **Foreign Keys** have a very similar concept because they also link rows with identical values together. But there are important differences:

- Foreign Keys have a direction. It is important to know which one of the two affected tables is the child table and which one is the parent table.
- Joins must be expressed within every DML statement which is interested in this join (with the exception of views). In contrast Foreign Keys are part of table definitions. All DML commands bear them in mind without expressing them within a DML statement.

Syntax

```
-- As part of CREATE TABLE command
CREATE TABLE <table_name> (
  ...
  CONSTRAINT <constraint_name> FOREIGN KEY (<column_name>) REFERENCES <parent_table_name> (<other_column_name>)
);

-- As part of ALTER TABLE command
ALTER TABLE <table_name> ADD  CONSTRAINT <constraint_name> ... ; -- same as above
ALTER TABLE <table_name> DROP CONSTRAINT <constraint_name>;      -- throw the definition away
```

Rules:

- FK-constraints can be defined during table definition (CREATE TABLE) or afterwards (ALTER TABLE). On this page we focus on the CREATE TABLE statement. The syntax of the ALTER TABLE statement is very similar.
- FK-constraints belong to the child table definition.
- Despite an existing FK-constraint it is possible that rows of the child table don't belong to any parent row. This occurs if the column value of the child row is NULL. If you want to avoid such situations, define the column as 'NOT NULL'.
- Although the FK-constraints belong to the child table, they also have consequences for the parent table such that rows of the parent table, which have existing rows in the child table, can not be deleted.
- The denoted parent table must exist.
- The denoted column of the parent table must be its Primary Key or a column which is UNIQUE.
- It is perfectly all right to use the same table as parent and child table within one FK-constraint, see: Exercises.
- One table may be subject of a lot of FK-contraints.

Example

The example defines the tables *department* and *employee*. The Foreign Key definition of *employee* declares *department* as the parent table of *employee*.

```
--
-- The parent table: DEPARTMENT
CREATE TABLE department (
  id          DECIMAL,
  dept_no     CHAR(10),
  dept_name   VARCHAR(100),
  CONSTRAINT dept_pk PRIMARY KEY (id)
);

-- The child table: EMPLOYEE
CREATE TABLE employee (
  id          DECIMAL,
  emp_name    VARCHAR(100),
  dept_id     DECIMAL,
  CONSTRAINT emp_pk PRIMARY KEY (id),
  CONSTRAINT emp_dept_fk FOREIGN KEY (dept_id) REFERENCES department(id)
);

-- This INSERT will fail because actually there is no department with id 10.
INSERT INTO employee   (id, emp_name, dept_id)  VALUES (1, 'Mike Baker', 10);
COMMIT;

-- It's necessary to store the department first.
INSERT INTO department (id, dept_no, dept_name) VALUES (10,'D10', 'E-Bike Development');
INSERT INTO employee   (id, emp_name, dept_id)  VALUES (1, 'Mike Baker', 10);
COMMIT;

-- The department may have a lot of employees
INSERT INTO employee   (id, emp_name, dept_id)  VALUES (2, 'Elenore McNeal', 10);
INSERT INTO employee   (id, emp_name, dept_id)  VALUES (3, 'Ted Walker', 10);
COMMIT;

-- This DELETE will fail because actually there are employees within the department.
DELETE FROM department WHERE dept_name = 'E-Bike Development';
COMMIT;
```

This kind of modelling allows the representation of hierarchical tree structures. One or many child nodes (rows) belong to a single parent node (row). In the context of DBMS this kind of association is called a 1:m relationship.

n:m Relationship

In the real world there are more association types than 1:m relationships. Often there are so called n:m relationships where objects (rows) belong to more than 1 other object (row). Thereby the meaning of parent/child tables gets lost. In our example database there is a table *hobby* and another table *person*. One person may pursue multiple hobbies. At the same time multiple persons may pursue the same hobby. This can be designed by creating a third table between the two original tables. The third table holds the id's of the first and second table. So one can decide which person pursues which hobby.

The technique to realize this n:m situation is the same as shown in the previous chapter with its 1:m association - it is only used twice. We define two Foreign Keys which start from the 'table-in-the-middle' and refers to the two other tables. In a technical sense we can say, that the 'table-in-the-middle' is the child table for the two parent tables *person* and *hobby*. *person* and *hobby* are at the same logical level.

```
--
CREATE TABLE t1 (
  id               DECIMAL,
  name             VARCHAR(50),
  -- ...
  CONSTRAINT t1_pk               PRIMARY KEY (id)
);
CREATE TABLE t2 (
  id               DECIMAL,
  name             VARCHAR(50),
  -- ...
  CONSTRAINT t2_pk               PRIMARY KEY (id)
);
CREATE TABLE t1_t2 (
  id               DECIMAL,
  t1_id            DECIMAL,
  t2_id            DECIMAL,
  CONSTRAINT t1_t2_pk            PRIMARY KEY (id),       -- also this table should have its own Primary Key
  CONSTRAINT t1_t2_unique        UNIQUE (t1_id, t2_id),  -- every link should occur only once
  CONSTRAINT t1_t2_fk_1          FOREIGN KEY (t1_id) REFERENCES t1(id),
  CONSTRAINT t1_t2_fk_2          FOREIGN KEY (t2_id) REFERENCES t2(id)
);
```

ON DELETE / ON UPDATE

So far we have assumed that rows of the parent table cannot be deleted if a row in the child table exists which refers to this parent row. This is the default, but all in all the SQL standard defines five options to handle this parent/child situation in various ways. The options extend the constraint definition. They are:

- ON DELETE CASCADE: If a row of the parent table is deleted, then all matching rows in the referencing table are deleted.
- ON DELETE SET NULL: If a row of the parent table is deleted, then all referencing columns in all matching rows of the child table are set to NULL.
- ON DELETE SET DEFAULT: If a row of the parent table is deleted, then all referencing columns in all matching rows of the child table are set to the column's default value.
- ON DELETE RESTRICT: It is prohibited to delete a row of the parent table if that row has any matching rows in the child table. The point in time when checking occures can be deferred until COMMIT.
- ON DELETE NO ACTION (the default): It is prohibited to delete a row of the parent table if that row has any matching rows in the child table. This holds true in ALL cases, even if checking is deferred (see next chapter).

Analog to the ON DELETE option there is an ON UPDATE option. It defines the same five options for the case of changing a column in the parent table which is referred by the column of a child table.

- ON UPDATE CASCADE: Any change to a referenced column in the parent table causes the same change to the corresponding referencing column in matching rows of the child table.
- ON UPDATE SET NULL: Any change to a referenced column in the parent table causes the corresponding referencing column in matching rows of the child table to be set to null.
- ON UPDATE SET DEFAULT: Any change to a referenced column in the referenced table causes the corresponding referencing column in matching rows of the referencing table to be set to its default value.
- ON UPDATE RESTRICT: It is prohibited to change a row of the parent table if that row has any matching rows in the child table. The point in time when checking occures can be deferred until COMMIT.
- ON UPDATE NO ACTION (the default): It is prohibited to change a row of the parent table if that row has any matching rows in the child table. This holds true in ALL cases, even if checking is deferred (see next chapter).

If ON DELETE or ON UPDATE are not specified, the default action NO ACTION will occur. In some systems the NO ACTION is implemented in the sense of the RESTRICT option.

An Example:

```
--
CREATE TABLE t1_t2 (
  ...
  CONSTRAINT t1_t2_fk_1 FOREIGN KEY (t1_id) REFERENCES t1(id)
                        ON UPDATE CASCADE ON DELETE RESTRICT,
  ...
);
```

Hint 1: The concept of updating Primary Keys is controversial.

Hint 2: Not all DBMS support all options.

IMMEDIATE / DEFERRED

There is an additional option to decide at what point in time the evaluation of the Foreign Key definition shall occur. The default behaviour is to check it with each UPDATE and DELETE command. The second possibility is deferring the check until the end of the transaction, which is the COMMIT command. The purpose of this deferring is to put applications in the position to modify parent tables **before** child tables (which may be helpful if they utilize Hibernate).

To define this option the constraint definition must be extended by the key words [NOT] DEFERABLE, which are pre- or postfixed by INITIALLY IMMEDIATE (the default) or INITIALLY DEFERRED to specify the initial state after the CREATE TABLE point in time.

```
--
CREATE TABLE t1_t2 (
  ...
  CONSTRAINT t1_t2_fk_1 FOREIGN KEY (t1_id) REFERENCES t1(id)
                        ON UPDATE CASCADE  DEFERRABLE INITIALLY IMMEDIATE
                        ON DELETE RESTRICT DEFERRABLE INITIALLY DEFERRED,
  ...
);
```

Hint: MySQL does not support the DEFERRABLE option, but the Foreign Key checking can be activated and deactived dynamically by 'SET foreign_key_checks = 0/1;'

The Chicken-Egg Problem

Sometimes applications run into cyclic dependencies: Table A contains a reference to table B and vice versa, e.g.: A table *team* contains the columns *id*, *team_name* and *team_leader* (which is an id to a player) and the table *player* contains the columns *id*, *player_name* and *team_id*.

```
--
CREATE TABLE team (
  id               DECIMAL,
  team_name        VARCHAR(50),
  team_leader      DECIMAL,      -- ID of a player
  CONSTRAINT team_pk             PRIMARY KEY (id)
);

CREATE TABLE player (
  id               DECIMAL,
  player_name      VARCHAR(50),
  team_id          DECIMAL,
  CONSTRAINT player_pk           PRIMARY KEY (id)
);

ALTER TABLE team   ADD CONSTRAINT team_fk   FOREIGN KEY (team_leader) REFERENCES player(id);
ALTER TABLE player ADD CONSTRAINT player_fk FOREIGN KEY (team_id)     REFERENCES team(id);
```

So far, so bad. When the first team-row shall be inserted, the player-row is missed. When the player-row is inserted first, the team-row is missed.

As we have seen above, there is a DEFER option. Using this option the FK-constraints must be defined such that they are not evaluated immediate with the INSERT commands. They shall be evaluated after all INSERTs at the COMMIT point in time.

```
-- Throw the above definitions away ...
ALTER TABLE team   DROP CONSTRAINT team_fk;
ALTER TABLE player DROP CONSTRAINT player_fk;
-- ... and use DEFERRABLE
ALTER TABLE team   ADD  CONSTRAINT team_fk
                        FOREIGN KEY (team_leader) REFERENCES player(id) DEFERRABLE INITIALLY DEFERRED;
ALTER TABLE player ADD  CONSTRAINT player_fk
                        FOREIGN KEY (team_id)     REFERENCES team(id)   DEFERRABLE INITIALLY DEFERRED;
```

Now we can insert data in any sequence (don't miss to deactivate AUTOCOMMIT).

```
--
INSERT INTO team   (id, team_name,   team_leader) VALUES (1, 'Wild Tigers',  1);
INSERT INTO player (id, player_name, team_id)     VALUES (1, 'Johnny Crash', 1);
-- No checking of Foreign Keys up to here
COMMIT;   -- Commit includes the check of Foreign Keys
```

DROP TABLE / TRUNCATE TABLE

Foreign Keys have implications to DROP TABLE and TRUNCATE TABLE commands. As long as a Foreign Key refers a parent table, this table cannot be dropped (remove structure and data) or truncated (remove data only). This holds true even if there is no actual row refering any row in the parent table - the existence of the Foreign Key is sufficient to refuse DROP and TRUNCATE.

To use DROP or TRUNCATE it is necessary to drop the constraint first.

Hint: Some implementations offer a DISABLE/ENABLE command to deactivate constraints temporarily.

Exercises

Is it possible that the parent table of a FK-constraint contains 1 row and the child table is empty?

Click to see solution

```
Yes. Parents without children are absolutly normal.
```

Is it possible that the child table of a FK-constraint contains 1 row and the parent table is empty?

Click to see solution

```
Yes. Although the main purpose of FK-constraints is the prevention of children without parents (orphans), this situation may occur.
If the column of the child row contains the NULL value, this row relates to no parent row
because 'null = <any value>' evaluates always to UNKNOWN and never to TRUE, even if that <any value> is the NULL value.
```

Create a table *genealogy* which stores information about people and their anchestors. The columns are: id, first_name, last_name, birth_name, father_id, mother_id.

Click to see solution

```
CREATE TABLE genealogy (
  id             DECIMAL                         PRIMARY KEY,
  first_name     VARCHAR(100),
  last_name      VARCHAR(100),
  birth_name     VARCHAR(100),
  father_id      DECIMAL,
  mother_id      DECIMAL
);
```

Extend the table *genealogy* by two FK-contraints such that the columns 'father_id' and 'mother_id' refer to other rows of this table.

Click to see solution

```
ALTER TABLE genealogy ADD CONSTRAINT gen_fk_1 FOREIGN KEY (father_id) REFERENCES genealogy(id);
ALTER TABLE genealogy ADD CONSTRAINT gen_fk_2 FOREIGN KEY (mother_id) REFERENCES genealogy(id);
```

Insert some data into 'genealogy', e.g.: data from your personal family.

Click to see solution

```
-- For the first rows store NULL in 'father_id' and 'mother_id'!
INSERT INTO genealogy (id, first_name, last_name, birth_name, father_id, mother_id)
               VALUES (1,  'Mike',     'Miller',  'Miller',   null,      null);
INSERT INTO genealogy (id, first_name, last_name, birth_name, father_id, mother_id)
               VALUES (2,  'Eve',      'Miller',  'Summer',   null,      null);
INSERT INTO genealogy (id, first_name, last_name, birth_name, father_id, mother_id)
               VALUES (3,  'Marry',    'Dylan',   'Miller',   1,       2);
INSERT INTO genealogy (id, first_name, last_name, birth_name, father_id, mother_id)
               VALUES (4,  'Henry',    'Dylan',   'Dylan',    null,      3);
COMMIT;
```

The ALTER TABLE command modifies column definitions and table constraints 'on the fly'. This means existing definitions are extended, changed or deleted or existing data is casted to a different type or existing data is evaluated against the new definitions.

```
-- change column definitions
ALTER TABLE <table_name> { ADD | ALTER } [ COLUMN ] <column_name> <column_definition>;
ALTER TABLE <table_name> { DROP        } [ COLUMN ] <column_name>;

-- change table constraints
ALTER TABLE <table_name> { ADD | ALTER } CONSTRAINT <constraint_name> <constraint_definition>;
ALTER TABLE <table_name> { DROP        } CONSTRAINT <constraint_name>;
```

The following examples are based on the test table *t1*.

```
CREATE TABLE t1 (
  id          NUMERIC  PRIMARY KEY,
  col_1       CHAR(4)
);
```

Columns

The syntax of the ADD COLUMN and ALTER COLUMN phrases are similar to the one shown in the create table page.

Add a Column

Existing tables can be extended by additional columns with the ADD COLUMN phrase. Within this phrase all options of the original Create Table statement are available: data type, default value, NOT NULL, Primary Key, Unique, Foreign Key, Check.

```
-- add a new column with any characteristic
ALTER TABLE t1 ADD COLUMN col_2 VARCHAR(100) CHECK (length(col_2) > 5);  -- Oracle: The key word 'COLUMN' is not allowed.
```

Alter the Characteristic of a Column

With the ALTER COLUMN phrase some characteristics of an existing column can be changed

- data type
- DEFAULT clause
- NOT NULL clause.

The new definitions must be compatible with the old existing data. If you change for example the data type from VARCHAR to NUMERIC this action can only be successful if it is possible to cast **all** existing VARCHAR data to NUMERIC - the casting of 'xyz' will fail. Casting in the direction from NUMERIC to VARCHAR will be successful as long as the width of the VARCHAR is large enought to store the result.

Hint: Concerning the change of the characteristics of columns some implementations ignore the syntax of the SQL standard and use other keywords like 'MODIFY'.

Change the Data Type

```
ALTER TABLE t1 ALTER COLUMN col_1 SET DATA TYPE NUMERIC;
```

Change the DEFAULT Clause

```
ALTER TABLE t1 ALTER COLUMN col_1 SET DEFAULT 'n/a';
```

Change the NOT NULL Clause

```
ALTER TABLE t1 ALTER COLUMN col_1 SET  NOT NULL;
ALTER TABLE t1 ALTER COLUMN col_1 DROP NOT NULL;
```

Drop a Column

Columns can be dropped from existing tables.

```
ALTER TABLE t1 DROP COLUMN col_2;    -- Oracle: The key word 'COLUMN' is mandatory.
```

Hint: As an extention to the SQL standard some implementations offer a RENAME or SET INVISIBLE command.

Table Constraints

Table constraints can be added, modified or dropped. The syntax is similar to that shown on the create table page.

Add a Table Constraint

```
ALTER TABLE t1 ADD CONSTRAINT t1_col_1_unique UNIQUE (col_1);
```

Alter a Table Constraint

```
ALTER TABLE t1 ALTER CONSTRAINT t1_col_1_unique UNIQUE (col_1);
```

Hint: Concerning the change of table constraints some implementations ignore the syntax of the SQL standard and use other keywords like 'MODIFY'.

Drop a Table Constraint

```
ALTER TABLE t1 DROP CONSTRAINT t1_col_1_unique;  -- MySQL: Not supported. There is only a 'DROP FOREIGN KEY'.
```

Hint: As an extention to the SQL standard some implementations offer an ENABLE / DISABLE command for constraints.

Exercises

Add a column 'col_3' to the table 't1': numeric, not null.

Click to see solution

```
ALTER TABLE t1 ADD COLUMN col_3 NUMERIC NOT NULL;
```

Add a Foreign Key from table 't1' column 'col_3' to table 'person' column 'id'.

Click to see solution

```
ALTER TABLE t1 ADD CONSTRAINT t1_col_3_fk FOREIGN KEY (col_3) REFERENCES person (id);
```

The DROP TABLE command removes the definition and all data of the named table from the database.

```
DROP TABLE <t1>;
```

The command handles the table as a whole. It will not fire any trigger. But it considers Foreign Key definitions. If any other table referes to the table to be dropped, the DROP TABLE command will fail. The Foreign Key definition must be dropped first.

As the DROP TABLE command handles the table as a whole, it is very fast.

Regular tables are containers to store data for a shorter or longer time periode and to offer those data to a lot of processes. In contrast, sometimes there is the requirement to handle data for a short time and only for local purposes. This is accomplished by the provision of TEMPORARY TABLES. They are subject to the SQL syntax in the same way as regular tables.

The common characteristic of all temporary tables is, that every session (connection) gets its **own incarnation** of the temporary table without any side effect to other sessions. This leads to the situation that every session sees only those data, which it has inserted previously. The data is **not shared** between different sessions, even if they use the same table name at the same time. Every session works on a different incarnation. A second common characteristic is that with the termination of the session all data of the temporary table is thrown away automatically. An explicit DELETE or DROP TABLE is not necessary.

The concept of temporary tables is similar to the concept of arrays of records within programming languages. The advantage over arrays is the availability of all DML statements known in SQL, eg.: if you need some intermediate data, you can retrieve it from a regular table and store it in a temporary table with one single Insert+Subselect command. The advantage over regular tables is that the overhead of logging and locking might be saved.

There are three slightly different types of temporary tables:

- Global temporary tables
- Local temporary tables
- Declared local temporary tables.

```
CREATE  GLOBAL TEMPORARY TABLE <table_name> (...) [ ON COMMIT { PRESERVE | DELETE } ROWS ];
CREATE  LOCAL  TEMPORARY TABLE <table_name> (...) [ ON COMMIT { PRESERVE | DELETE } ROWS ];
```

```
DECLARE LOCAL  TEMPORARY TABLE <table_name> (...) [ ON COMMIT { PRESERVE | DELETE } ROWS ];
```

If the phrase 'ON COMMIT DELETE ROWS' is used, the data is automatically thrown away with every COMMIT command, else at the end of the session (or with a DELETE command).

Global Temporary Tables (GTT)

If a GTT is created, this **definition** keeps alive beyond the end of the defining session. Even other session within this schema sees the definition. Therefore the GTT can be defined simultaneous with regular tables and applications don't need to create GTTs by itself (but they can do it). Up to this point GTTs don't differ from regular tables. The distinction relates to the **data**. As with all temporary tables every session gets its own incarnation of the table and cannot access data from any other session. If the session terminates, all data from the table is thrown away automatically.

A typical use case is an application which needs a temporary protocoll about its own activities like successful actions, exceptions, ... to perform recovery activities later on. This information is not of interest for other sessions. Moreover it may be deleted at the end of a transaction or at the end of the session.

Another use case is an application which want to store an intermediate result set and iterate about its single rows to performs actions depending on the columns values.

```
-- The table may be defined by a different session long time before.
CREATE GLOBAL TEMPORARY TABLE temp1 (
  ts       TIMESTAMP,
  action   CHAR(100),
  state    CHAR(50)
)
ON COMMIT PRESERVE ROWS;
--
-- Insert some data
INSERT INTO temp1 VALUES (current_timestamp, 'node-1-request sended.', 'OK');
INSERT INTO temp1 VALUES (current_timestamp, 'node-2-request sended.', 'OK');
INSERT INTO temp1 VALUES (current_timestamp, 'node-1-answer received.', 'Failed');
INSERT INTO temp1 VALUES (current_timestamp, 'node-2-answer received.', 'OK');
SELECT count(*) FROM temp1 WHERE state = 'OK';
...
COMMIT;
SELECT count(*) FROM temp1; -- In this example all rows should have survived the COMMIT command
-- After a disconnect from the database and establishing of a new session the table exists and is empty.
```

Local Temporary Tables (LTT)

The **definition** of a LTT will never survive the duration of a session. The same applies to its **data**, which accords to the behaviour of all temporary tables. In consequence every session must define its own LTT before it can store anything into it. Multiple sessions can use the same table name simultaneously without affecting each other, which - again - accords to the behaviour of all temporary tables.

```
-- The table must be defined by the same session (connection) which stores data into it.
CREATE LOCAL TEMPORARY TABLE temp2 (
  ts       TIMESTAMP,
  action   CHAR(100),
  state    CHAR(50)
)
ON COMMIT PRESERVE ROWS;
-- After a disconnect from the database and establishing of a new session the table will not exist.
```

The SQL-standard distinguishs between SQL-sessions and modules within SQL-sessions. It postulates that LTTs are visible only within that module, which has actually created the table. The tables are not shared between different modules of the same SQL-session. But the LTTs definition occurs in the information schema of the DBMS.

Declared Local Temporary Tables (DLTT)

The main concept of DLTT is very similar to that of LTT. The difference is that in opposite to the definition of a LTT the definition of a DLTT will not occur in the information schema of the DBMS. It is known only by the module where it is defined. You can imagine a DLTT as some kind of a module-local variable.

```
-- The declaration must be defined by the same module which stores data into the table.
DECLARE LOCAL TEMPORARY TABLE temp3 (
  ts       TIMESTAMP,
  action   CHAR(100),
  state    CHAR(50)
)
ON COMMIT PRESERVE ROWS;
-- After a disconnect from the module and entering the module again the declaration will not exist.
```

Implementation Hints

MySQL:

- Omit the key words LOCAL/GLOBAL and the ON COMMIT phrase. Temporary tables are always LOCAL and the ON COMMIT acts always in the sense of PRESERVE ROWS.
- GTT and DLTT are not supported.

Oracle:

- LTT and DLTT are not supported.

Indexes are a key feature of all SQL databases. They provide quick access to the data. Therefore almost all implementations support a CREATE INDEX statement.

Nevertheless the CREATE INDEX statement is **not part of the SQL standard**! The reason for this is unknown. Possibly it is a deliberate decision against all implemenation issues. Or it results from the wide range of different syntaxes realized by vendors and the lack of finding a compromise.

On this page we offer some basic ideas concerning indexes and the syntax which is common to a great number of implementations.

```
CREATE [UNIQUE] INDEX <index_name> ON <table_name> (<column_name> [, <column_name>]);
```

The Concept of Indexes

DBMSs offer quick access to data stored in their tables. One might think that such high-speed access is due to fast hardware of modern computers: millions of CPU cycles per second, I/O rates in the range of milliseconds, access to RAM within micro- or nanoseconds, etc. That is true, but only partly so. Instead, the use of intelligent software algorithms, especially in the case of handling large amounts of data, is the dominant factor.

Consider a request to the DBMS to determine, whether or not a person with a certain name can be found in a table with 1 million entries. With a primitive, linear algorithm the system has to read 500,000 rows (on average) to decide the question. The binary search algorithm implements a more sophisticated strategy which answers the question after reading 20 rows or less. In this case this choice of algorithm leads to a factor of 25,000 in performance. In order to really grasp the magnitude of this improvement you may want to multiply your salary by 25,000.

Admittedly this comparision between the linear access and the binary search algorithm is a little bit simple. First, DBMS usually read blocks containing multiple rows and not single rows. But this didn't change the situation. If a block contains 100 rows, modify the above example from 1 million to 100 million rows. Second, the binary seach algorithm assumes that the data is ordered. This means that during data entry there is an additional step for sorting the actual input into the existing data. This applies only once and is independent from the number of read accesses. In summary there is additional work during data entry and much less work during data access. It depends on the typical use of the data whether the additional work is worthwhile.

The index is an additional storage holding data which is copied or deducted from the original data in the table. He consists only of **redundant data**. What parts make up the index? In the case of the binary search stategy the index holds the original values of the tables column plus a backreference to the original row. In most cases he is organized as a balanced tree with the columns value as the trees key and the backreference as additional information for each key.

The binary search algorithm is one of a lot of methods for building indexes. The common characteristics of indexes are that they consists only of redundant information and use additional resouces in sense of CPU cycles, RAM or disc space and offer better performance for queries on large data amounts. If they are used on small tables or there are too much indexes for the same table it is possible that the disadvantages outweighs the benefits.

Basic Index

If an application use to retrieve data by a certain criterion - e.g. a person name for a phone book application - and this criterion consists of a tables column, this column should have an index.

```
CREATE INDEX person_lastname_idx ON person(lastname);
```

The index has its own freely selectable name - *person_lastname_idx* in this example - and is build on a certain column of a certain table. The index may be defined and created directly after the CREATE TABLE statement (when there is no data in the table) or after some or a huge number of INSERT commands. After it is created the DBMS should be in the state to answer questions like the following quicker than before.

```
SELECT count(*)
FROM   person
WHERE  lastname = 'Miller';
```

The index is used during the evaluation of the WHERE clause. But it is not sure that the index is used. The DBMS has the choice between on the one hand reading all *person* rows and counting such where the lastname is 'Miller' or on the other hand reading the index (possibly with binary search) and counting all nodes with value 'Miller'. Which strategy is used depends on a lot of decisions. If, for example, the DBMS knows that about 30% of all rows contains 'Miller' it may choose a different strategy than if it knows that only 0.3% contains 'Miller'.

A table may have more than one index.

```
CREATE INDEX person_firstname_idx ON person(firstname);
```

What will happen in such a situation to a query like the following one?

```
SELECT count(*)
FROM   person
WHERE  lastname = 'Miller'
AND    firstname = 'Henry';
```

Again, the DBMS has more than one choice to retrieve the expected result. It may use only one of the two indexes, read the resulting rows and look for the missing other value. Or it reads both indexes and count the common backreferences. Or it ignores both indexes, reads the data and counts such rows where both criterias apply. As mentioned it depends on a lot of decisions.

Multiple Columns

If an application typically searchs in two columns within **one** query, e.g. for first- and lastname, it can be usefull to build one index for both columns. This strategy is very different from the above example where we build two independent indexes, one per column.

```
CREATE INDEX person_fullname_idx ON person(lastname, firstname);
```

In this case the key of the balanced tree is the concatenation of last- and firstname. The DBMS can use this index for queries which ask for last- and firstname. It can also use the index for queries for lastname only. But it cannot use the index for queries for firstname only. The firstname can occur at different places within the balanced tree. Therefore it is worthless for such queries.

Functional Index

In some cases an existing index cannot be used for queries on the underlying column. Suppose the query to person names should be case-insensitive. To do so the application converts all user-input to upper-case and use the UPPER() function to the column in scope.

```
-- Original user input was: 'miller'
SELECT count(*)
FROM   person
WHERE  UPPER(lastname) = 'MILLER';
```

As the criterion in the WHERE clause looks only for uppercase characters and the index is build in a case-sensitive way, the key in the balanced tree is worthless: 'miller' is sorted at a very different place than 'Miller'. To overcome the problem one can define an index, which uses exactly the same strategy as the WHERE criterion.

```
CREATE INDEX person_uppername_idx ON person(UPPER(lastname)); -- not supported by MySQL
```

Now the 'UPPER()' query can use this so-called functional index.

Unique Index

The Primary Key of every table is unique, which means that no two columns can contain the same value. Sometimes one column or the concatenation of some columns is also unique. To ensure this criterion you can define a UNIQUE CONSTRAINT or you can define an index with the additional UNIQUE criterion. (Often UNIQUE CONSTRAINTS silently use UNIQUE INDEX in the background.)

```
CREATE UNIQUE INDEX person_lastname_unique_idx ON person(lastname);
```

Unique indexes can only be created on existing data, if the column in scope really has nothing but unique values (which is not the case in our database example).

Drop an Index

Indexes can be dropped by the command:

```
DROP INDEX <index_name>;
```

For multiuser systems like DBMSs it is necessary to grant and revoke rights for manipulating its objects. The GRANT command defines which user can manipulate (create, read, change, drop, ...) which object (tables, views, indexes, sequences, triggers, ...).

```
GRANT <privilege_name>
ON    <object_name>
TO    [ <user_name> | <role_name> | PUBLIC ]
[WITH GRANT OPTION];
```

The REVOKE statement deprives the granted rights.

```
REVOKE <privilege_name>
ON     <object_name>
FROM   [ <user_name> | <role_name> | PUBLIC ];
```

The example statement grants SELECT and INSERT on table *person* to the user *hibernate*. The second statement removes the granted rights.

```
GRANT  SELECT, INSERT ON person TO   hibernate;
REVOKE SELECT, INSERT ON person FROM hibernate;
```

Privileges

Privileges are actions which users can perform. The SQL standard supports only a limited list of privileges whereas real implementations offer a great bunch of different privileges. The list consists of: SELECT, INSERT, UPDATE, DELETE, CREATE <object_type>, DROP <object_type>, EXECUTE, ...

Object Types

The list of object types to which privileges may be granted is short in the SQL standard and long for real implementations. It consists of tables, views, indexes, sequences, triggers, procedures, ...

Roles / Public

If there is a great number of users connecting to the DBMS, it is helpful to group users with identical rights to a role and grant privileges not to the individuell users but to the role. To do so, the role must be created by a CREATE ROLE statement. Afterwards users are joined with this role.

```
-- Create a role
-- (MySQL supports only predefinded roles with special semantics).
CREATE ROLE department_human_resouces;

-- Enrich the role with rights
GRANT  SELECT, INSERT, UPDATE, DELETE ON person        TO department_human_resouces;
GRANT  SELECT, INSERT                 ON hobby         TO department_human_resouces;
GRANT  SELECT, INSERT, UPDATE, DELETE ON person_hobby  TO department_human_resouces;

-- Join users with the role
GRANT  department_human_resouces TO user_1;
GRANT  department_human_resouces TO user_2;
```

Instead of individuell usernames the key word PUBLIC denotes **all** known users.

```
-- Everybody shall be allowed to read the rows of 'person' table.
GRANT SELECT ON person TO PUBLIC;
```

Grant Option

If a DBA wants to delegate the managing of rights to special users, he can grant privileges to them and extend the statement with the

phrase 'WITH GRANT OPTION'. This enables the users to grant the received privileges to any other user.

```
-- User 'hibernate' gets the right to pass the SELECT privilege on table 'person' to any other user.
GRANT SELECT ON person TO hibernate WITH GRANT OPTION;
```

Structured Query Language/Like Predikate

There are use cases in which an application wants to compare rows or columns not with a fix value - e.g.: 'WHERE status = 5' - but with a result of a query which is evaluated at runtime. A first example of such dynamic queries are subqueries which results in exactly **one** value: '... WHERE version = (SELECT MAX(version) ...)'. Additionally sometimes there is the need to compare against a set, which contains **multiple** values: '... WHERE version <comparision> (SELECT version FROM t1 WHERE status > 2 ...)'.

To do so, SQL offers some special comparision methods between the table to be queried and the result of the subquery: IN, ALL, ANY/SOME and EXISTS. They belong to the group of so called *predicates*.

- The IN predicate retrievs rows which correlate to the resulting values of the subquery.
- The ALL predicate (in combination with <, <=, =, >=, > or <>) retrievs rows which correlate to **all** values of the subquery (boolean AND operation).
- The ANY predicate (in combination with <, <=, =, >=, > or <>) retrievs rows which correlate to **any** value of the subquery (boolean OR operation). The key word SOME can be used as a synonym for ANY, so you can exchange one against the other.
- The EXISTS predicate retrievs rows, if the subquery retrieves one or more rows.

IN

The IN predicate - as descripted in a previous chapter - accepts a set of values or rows.

```
SELECT *
FROM   person
WHERE  id IN
  (SELECT person_id FROM contact); -- Subquery with potentially a lot of rows.
```

The subquery selects a lot of values. Therefore it is not possible to use operators like '=' or '>'. They would merely compare single values. But the IN predicate handles the situation and compares *person.id* with every value of *contact.person_id* regardless of the number of *contact.person_id* values. This comparisons are mutally linked in the sense of boolean OR operations.

The IN predicate can be negated by adding the key word NOT.

```
...
WHERE  id NOT IN
...
```

ALL

The ALL predicate compares every row in the sense of a boolean AND with every value of the subquery. It needs - in contrast to the IN predicate - an additional operator, which is one of: <, <=, =, >=, > or <>.

```
SELECT *
FROM   person
WHERE  weight > ALL
  (SELECT weight FROM person where lastname = 'de Winter');
```

Common hint: If there is no NULL special marker in the subquery it is possible to replace the ALL predicate by equivalent (and more intuitive) operations:

<op> ALL	Substitution
< ALL	< (SELECT MIN() ...)
<= ALL	<= (SELECT MIN() ...)
= ALL	'=' or 'IN', if subselect retrieves 1 value.
	Else: A single value cannot be equal to different values at the same time. (x = a AND x = b) evaluates to 'false' in all cases.
>= ALL	>= (SELECT MAX() ...)
> ALL	> (SELECT MAX() ...)
<> ALL	'<>' or 'NOT IN', if subselect retrieves 1 value.
	Else: 'NOT IN'. (x <> a AND x <> b).

MySQL hint: Because of query rewrite issues the ONLY_FULL_GROUP_BY mode shall be disabled, e.g. by the command: set sql_mode='ANSI'.

ANY/SOME

The key words ANY and SOME are synonyms, their meaning is the same. Within this wikibook we prefer the use of ANY.

The ANY predicate compares every row in the sense of a boolean OR with every value of the subquery. It needs - in contrast to the IN predicate - an additional operator, which is one of: <, <=, =, >=, > or <>.

```
SELECT *
FROM   person
WHERE  weight > ANY
  (SELECT weight FROM person where lastname = 'de Winter');
```

Common hint: If there is no NULL special marker in the subquery it is possible to replace the ANY predicate by equivalent (and more intuitive) operations:

<op> ANY	Substitution
< ANY	< (SELECT MAX() ...)
<= ANY	<= (SELECT MAX() ...)
= ANY	'=' or 'IN', if subselect retrieves 1 value.
	Else: 'IN'. (x = a OR x = b).
>= ANY	>= (SELECT MIN() ...)
> ANY	> (SELECT MIN() ...)
<> ANY	'<>' or 'NOT IN', if subselect retrieves 1 value.
	Else: A single value is always different from two or more different values under an OR conjunction. (x <> a OR x <> b) evaluates to 'true' in all cases.

MySQL hint: Because of query rewrite issues the ONLY_FULL_GROUP_BY mode shall be disabled, e.g. by the command: set sql_mode='ANSI'.

EXISTS

The EXISTS predicate retrievs rows, if the subquery retrieves one or more rows. Meaningful examples typically use a correlated subquery.

```
SELECT *
FROM   contact c1
WHERE EXISTS
  (SELECT  *
   FROM    contact c2
   WHERE   c2.person_id = c1.person_id  -- correlation criterion between query and subquery
   AND     c2.contact_type = 'icq');
```

The example retrieves all contacts for such persons, which have an ICQ-contact.

The EXISTS predicate can be negated by adding the key word NOT.

```
...
WHERE NOT EXISTS
...
```

In the chapter Grouping we have seen that the key word GROUP BY creates groups of rows within a result set. Additionally aggregat functions like `SUM()` computes condensed values for each of those groups.

As GROUP BY can work for more than one single column there is often the requirement to compute such condensed values also for 'super-groups', which arise by omitting successive one column after the next from the GROUP BY specification.

Example Table

To illustrate the situation we offer an example table and typical questions to such kind of tables.

```
CREATE TABLE car_pool (
  -- define columns (name / type / default value / nullable)
  id           DECIMAL      NOT NULL,
  producer     VARCHAR(50)  NOT NULL,
  model        VARCHAR(50)  NOT NULL,
  yyyy         DECIMAL      NOT NULL CHECK (yyyy BETWEEN 1970 AND 2020),
  counter      DECIMAL      NOT NULL CHECK (counter >= 0),
  CONSTRAINT   car_pool_pk PRIMARY KEY (id)
);
--
INSERT INTO car_pool VALUES ( 1, 'VW',     'Golf',    2005, 5);
INSERT INTO car_pool VALUES ( 2, 'VW',     'Golf',    2006, 2);
INSERT INTO car_pool VALUES ( 3, 'VW',     'Golf',    2007, 3);
INSERT INTO car_pool VALUES ( 4, 'VW',     'Golf',    2008, 3);
INSERT INTO car_pool VALUES ( 5, 'VW',     'Passat',  2005, 5);
INSERT INTO car_pool VALUES ( 6, 'VW',     'Passat',  2006, 1);
INSERT INTO car_pool VALUES ( 7, 'VW',     'Beetle',  2005, 1);
INSERT INTO car_pool VALUES ( 8, 'VW',     'Beetle',  2006, 2);
INSERT INTO car_pool VALUES ( 9, 'VW',     'Beetle',  2008, 4);
INSERT INTO car_pool VALUES (10, 'Toyota', 'Corolla', 2005, 4);
INSERT INTO car_pool VALUES (11, 'Toyota', 'Corolla', 2006, 3);
INSERT INTO car_pool VALUES (12, 'Toyota', 'Corolla', 2007, 2);
INSERT INTO car_pool VALUES (13, 'Toyota', 'Corolla', 2008, 4);
INSERT INTO car_pool VALUES (14, 'Toyota', 'Prius',   2005, 1);
INSERT INTO car_pool VALUES (15, 'Toyota', 'Prius',   2006, 1);
INSERT INTO car_pool VALUES (16, 'Toyota', 'Hilux',   2005, 1);
INSERT INTO car_pool VALUES (17, 'Toyota', 'Hilux',   2006, 1);
INSERT INTO car_pool VALUES (18, 'Toyota', 'Hilux',   2008, 1);
--
COMMIT;
```

In the table there are two different car producer, 6 models and 4 years. Typical questions to such tables are:

- Number of cars per producer or per model.
- Number of cars per combination of some criterias like: producer plus model or producer plus year.
- Total number of cars (without any criteria).

ROLLUP

As we have seen, the key word GROUP BY offers condensed data for exactly one grouping level, *producer* plus *model* in this case.

```
SELECT producer, model, sum(counter) as cnt
FROM   car_pool
GROUP BY producer, model
ORDER BY producer, cnt desc;
--
Toyota  Corolla 13
Toyota  Hilux   3
Toyota  Prius   2
VW  Golf    13
VW  Beetle  7
VW  Passat  6
```

In such situations one would like to know also the corresponding values for upper groups: per *producer* or for the whole table. This can be achieved by submitting slightly different SELECTs.

```
SELECT producer, sum(counter) as cnt
FROM   car_pool
GROUP BY producer
ORDER BY producer, cnt desc;
--
Toyota  18
VW  26
--
--
SELECT sum(counter) as cnt
FROM   car_pool;
```

```
--
44
```

In principle it is possible, to combine such SELECTs via UNION or to submit them sequentially. But because this is a standard requirement SQL offers a more elegant solution, namely the extention of the GROUP BY with the ROLLUP key word. Based on the results of the GROUP BY it offers additional rows for every superordinate group, which arises by omitting the grouping criterias one after the other.

```
SELECT producer, model, sum(counter) as cnt
FROM   car_pool
GROUP BY ROLLUP (producer, model);  -- the MySQL syntax is: GROUP BY producer, model WITH ROLLUP
--
Toyota  Corolla 13
Toyota  Hilux   3
Toyota  Prius   2
Toyota      18   <-- the additional row per first producer
VW  Beetle  7
VW  Golf    13
VW  Passat  6
VW      26   <-- the additional row per next producer
        44   <-- the additional row per all producers
```

The simple GROUP BY clause creates rows at the level of *producer* plus *model*. The ROLLUP key word leads to additional rows where first the *model* and then *model* and *producer* are omitted.

CUBE

The ROLLUP key word offers solutions where a hierarchical point of view is adequate. But in data warehouse applications one likes to navigate freely through the aggregated data, not only from top to bottom. To support this requirement, the SQL standard offers the key word CUBE. It is an extention of ROLLUP and offers additional rows for **all possible combinations** of the GROUP BY columns.

In the case of our above example with the two columns *producer* and *modell* the ROLLUP has created rows for '*producer*-only' and 'no criteria' (= complete table). Additional to that, CUBE creates rows for '*model*-only'. (If different *producer* would use the same *model*-name, such rows will lead to only 1 additional row.)

```
SELECT producer, model, SUM(counter) AS cnt
FROM   car_pool
GROUP BY CUBE (producer, model);  -- not supported by MySQL
--
Toyota  Corolla 13
Toyota  Hilux   3
Toyota  Prius   2
Toyota  -   18
VW  Beetle  7
VW  Golf    13
VW  Passat  6
VW  -   26
-   Beetle  7        <--
-   Corolla 13       <--
-   Golf    13       <-- additional rows for 'model-only'
-   Hilux   3        <--
-   Passat  6        <--
-   Prius   2        <--
-   -   44
```

If there are tree grouping columns c1, c2 and c3, the key words leads to the following grouping.

GROUP BY: (c1, c2, c3)
GROUP BY ROLLUP: (c1, c2, c3), (c1, c2), (c1) and ()
GROUP BY CUBE: (c1, c2, c3), (c1, c2), (c1, c3), (c2, c3), (c1), (c2), (c3) and ()

The *window functions* discussed on this page are a special and very powerful extension to 'traditional' functions. They compute their result not on a single row but on a set of rows (similar to aggregate functions acting in correlation with a `GROUP BY clause`). This set of rows - and this is the crucial point - 'moves' or 'slides' over all rows, which are determined by the `WHERE clause`. This 'sliding window' is called a **frame** or - in terms of the official SQL standard - the 'window frame'.

Here are some examples:

- A very easy example is a 'sliding window' consisting of the previous, the current and the next row.
- One typical area for the use of *window functions* are evaluations about arbitrary time series. If you have the time series of market prices of a share, you can easily compute the Moving Average of the last n days.
- *Window functions* are often used in data warehouse and other OLAP applications. If you have data about sales of all products over a lot of periods within a lot of regions you can compute statistical indicators about the revenues. This evaluations are more powerful than simple `GROUP BY clauses`.

In opposite to `GROUP BY clauses`, where only one output row per group exists, with *window functions* all rows of the result set retain

their identity and are shown.

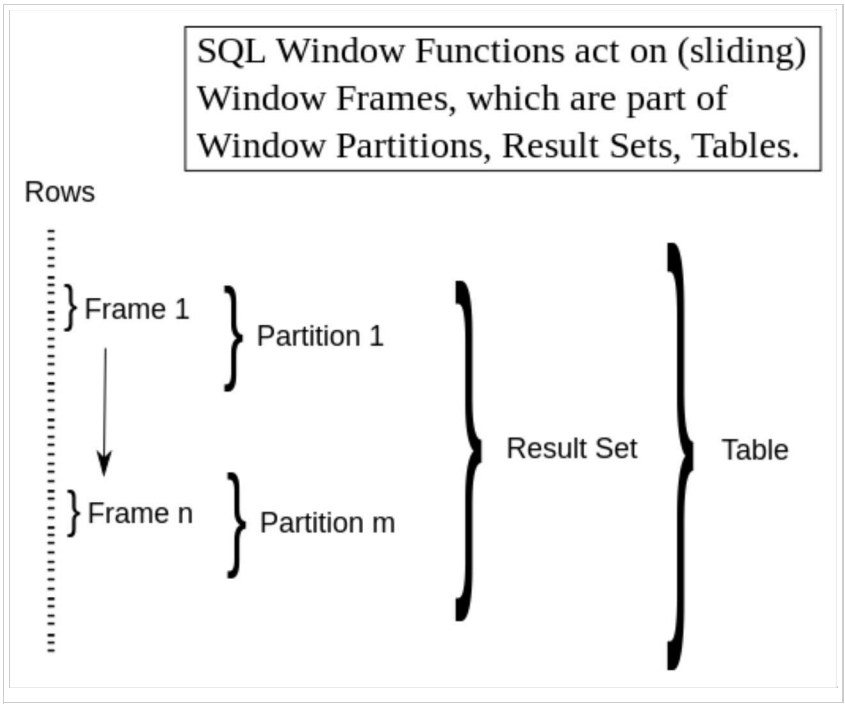
Syntax

Window functions are listed between the two key words `SELECT` and `FROM` at the same place where usual functions and columns are listed. They contain the key word OVER.

```
-- Window functions appear between the key words SELECT and FROM
SELECT   ...,
         <window_function>,
         ...
FROM     <tablename>
...
;

-- They consist of three main parts:
--   1. function type (which is the name of the function)
--   2. key word 'OVER'
--   3. specification, which rows constitute the 'sliding window' (partition, order and frame)
<window_function>      := <window_function_type> OVER <window_specification>

<window_function_type> := ROW_NUMBER() | RANK() | LEAD(<column>) | LAG(<column>) |
                          FIRST_VALUE(<column>) | LAST_VALUE(<column>) | NTH_VALUE(<column>, <n>) |
                          SUM(<column>) |  MIN(<column>) | MAX(<column>) | AVG(<column> | COUNT(<column>)

<window_specification> := [ <window_partition> ] [ <window_order> ] [ <window_frame> ]

<window_partition>     := PARTITION BY <column>
<window_order>         := ORDER BY <column>
<window_frame>         := see below
```

Overall Description

Concerning window functions there are some similar concepts. In order to be able to significantly differ the concepts from each other, it is necessary to use an exact terminology. This terminology is introduced in the next 8 paragraphs, which also - roughly - reflect the order of execution. The goal of the first seven steps is the determination of the actual frame and the eighth step acts on it.



1. The `WHERE clause` returns a certain number of rows. They constitutes the **result set**.
2. The `ORDER BY clause` (syntactically behind the `WHERE clause`) re-orders the **result set** into a certain sequence.
3. This sequence determines the order in which the rows are passed to the `SELECT clause`. The row, which is actually given to the `SELECT clause`, is called the **current row**.
4. The `WINDOW PARTITION clause` divides the **result set** into **window partitions** (We will use the shorter term **partition** as in the context of our site there is no danger of confusion). If there is no `WINDOW PARTITION clause`, all rows of the **result set** constitutes one **partition**. (This **partitions** are equivalent to groups created by the `GROUP BY clause`.) **Partitions** are distinct from each other: there is no overlapping as every row of the **result set** belongs to one and only one **partition**.
5. The `WINDOW ORDER clause` orders the rows of each **partition** (which may differ from the `ORDER BY clause`).
6. The `WINDOW FRAME clause` defines which rows of the actual **partition** belong to the actual **window frame** (We will use the shorter term **frame**). The clause defines one **frame** for every row of the **result set**. This is done by determine the lower and upper boundary of affected rows. In consequence there are as many (mostly different) frames as number of rows in the result set. The upper and lower boundaries are newly determinded with every row of the result set! Single rows may be part of more than one **frame**. The actual **frame** is the instanciation of the 'sliding window'. Its rows are ordered according to the `WINDOW ORDER clause`.
7. If there is no `WINDOW FRAME clause`, the rows of the actual **partition** constitute **frame**s with the following default boundaries: The first row of the actual **partition** is their lower boundary and the **current row** is their upper boundary. If there is no `WINDOW FRAME clause` and no `WINDOW ORDER clause`, the upper boundary switches to the last row of the actual **partition**. Below we will explain how to change this default behaviour.
8. The <window_function_type>s act on the rows of the actual **frame**.

Example Table

We use the following table to demonstrate window functions.

```
CREATE TABLE employee (
  -- define columns (name / type / default value / column constraint)
  id            DECIMAL                       PRIMARY KEY,
  emp_name      VARCHAR(20)                   NOT NULL,
  dep_name      VARCHAR(20)                   NOT NULL,
  salary        DECIMAL(7,2)                  NOT NULL,
  age           DECIMAL(3,0)                  NOT NULL,
  -- define table constraints (it's merely an example table)
  CONSTRAINT empoyee_uk UNIQUE (emp_name, dep_name)
);

INSERT INTO employee VALUES ( 1,  'Matthew', 'Management',  4500, 55);
INSERT INTO employee VALUES ( 2,  'Olivia',  'Management',  4400, 61);
INSERT INTO employee VALUES ( 3,  'Grace',   'Management',  4000, 42);
INSERT INTO employee VALUES ( 4,  'Jim',     'Production',  3700, 35);
INSERT INTO employee VALUES ( 5,  'Alice',   'Production',  3500, 24);
INSERT INTO employee VALUES ( 6,  'Michael', 'Production',  3600, 28);
INSERT INTO employee VALUES ( 7,  'Tom',     'Production',  3800, 35);
INSERT INTO employee VALUES ( 8,  'Kevin',   'Production',  4000, 52);
INSERT INTO employee VALUES ( 9,  'Elvis',   'Service',     4100, 40);
INSERT INTO employee VALUES (10,  'Sophia',  'Sales',       4300, 36);
INSERT INTO employee VALUES (11,  'Samantha','Sales',       4100, 38);
COMMIT;
```

A First Query

The example demonstrates how the boundaries 'slides' over the result set. Doing so, they create one frame after the next, one per row of the result set. These frames are part of partitions, the partitions are part of the result set and the result set is part of the table.

```
SELECT id,
       emp_name,
       dep_name,
       -- The functions FIRST_VALUE() and LAST_VALUE() explain itself by their name. They act within the actual frame.
       FIRST_VALUE(id) OVER (PARTITION BY dep_name ORDER BY id) AS frame_first_row,
       LAST_VALUE(id)  OVER (PARTITION BY dep_name ORDER BY id) AS frame_last_row,
       COUNT(*)        OVER (PARTITION BY dep_name ORDER BY id) AS frame_count,
       -- The functions LAG() and LEAD() explain itself by their name. They act within the actual partition.
       LAG(id)         OVER (PARTITION BY dep_name ORDER BY id) AS prev_row,
       LEAD(id)        OVER (PARTITION BY dep_name ORDER BY id) AS next_row
FROM   employee;
-- For simplification we use the same PARTITION and ORDER definitions for all window functions.
-- This not necessary. You can use divergent definitions!
```

Please notice how the lower boundary (FRAME_FIRST_ROW) and the upper boundary (FRAME_LAST_ROW) changes from row to row.

ID	EMP_NAME	DEP_NAME	FRAME_FIRST_ROW	FRAME_LAST_ROW	FRAME_COUNT	PREV_ROW	NEXT_ROW
1	Matthew	Management	1	1	1	-	2
2	Olivia	Management	1	2	2	1	3
3	Grace	Management	1	3	3	2	-
4	Jim	Production	4	4	1	-	5
5	Alice	Production	4	5	2	4	6
6	Michael	Production	4	6	3	5	7
7	Tom	Production	4	7	4	6	8
8	Kevin	Production	4	8	5	7	-
10	Sophia	Sales	10	10	1	-	11
11	Samantha	Sales	10	11	2	10	-
9	Elvis	Service	9	9	1	-	-

The query has no `WHERE clause`. Therefore all rows of the table are part of the result set. According to the `WINDOW PARTITION clause`, which is 'PARTITION BY dep_name', the result set is divided into the 4 partitions: 'Management', 'Production', 'Sales' and 'Service'. The frames run within these partions. As there is no `WINDOW FRAME clause` the frames start at the first row of the actual partition and runs up to the current row.

You can see that the actual number of rows within a frame (column FRAME_COUNT) grows from 1 up to the sum of all rows within the partition. When the partition switches to the next one, the number starts again with 1.

The columns PREV_ROW and NEXT_ROW shows the ids of the previous and next row within the actual partition. As the first row has

no predecessor, the `NULL indicator` is shown. This applies correspondingly to the last row and its successor.

Basic Window Functions

We present some of the `<window_function_type>` functions and their meaning. The standard as well as most implementations knows a lot of additional functions and overloaded variants.

Signature	Scope	Meaning / Return Value
FIRST_VALUE(<column>)	Actual Frame	The column value of the first row within the frame.
LAST_VALUE(<column>)	Actual Frame	The column value of the last row within the frame.
LAG(<column>)	Actual Partition	The column value of the predecessor row (the row which is before the current row).
LAG(<column>, <n>)	Actual Partition	The column value of the n.-th row before the current row.
LEAD(<column>)	Actual Partition	The column value of the successor row (the row which is after the current row).
LEAD(<column>, <n>)	Actual Partition	The column value of the n.-th row after the current row.
ROW_NUMBER()	Actual Frame	A numeric sequence of the row within the frame.
RANK()	Actual Frame	A numeric sequence of the row within the frame. Identical values in the specified order evaluate to the same number.
NTH_VALUE(<column>, <n>)	Actual Frame	The column value of the n.-th row within the frame.
SUM(<column>) MIN(<column>) MAX(<column>) AVG(<column>) COUNT(<column>)	Actual Frame	As usual.

Here are some examples:

```
SELECT id,
       emp_name,
       dep_name,
       ROW_NUMBER()           OVER (PARTITION BY dep_name ORDER BY id) AS row_number_in_frame,
       NTH_VALUE(emp_name, 2) OVER (PARTITION BY dep_name ORDER BY id) AS second_row_in_frame,
       LEAD(emp_name, 2)      OVER (PARTITION BY dep_name ORDER BY id) AS two_rows_ahead
FROM   employee;
```

ID	EMP_NAME	DEP_NAME	ROW_NUMBER_IN_FRAME	SECOND_ROW_IN_FRAME	TWO_ROWS_AHEAD
1	Matthew	Management	1	-	Grace
2	Olivia	Management	2	Olivia	-
3	Grace	Management	3	Olivia	-
4	Jim	Production	1	-	Michael
5	Alice	Production	2	Alice	Tom
6	Michael	Production	3	Alice	Kevin
7	Tom	Production	4	Alice	-
8	Kevin	Production	5	Alice	-
10	Sophia	Sales	1	-	-
11	Samantha	Sales	2	Samantha	-
9	Elvis	Service	1	-	-

The three example shows:

- The row number within the actual frame.
- The employee name of the second row within the actual frame. This is not possible in all cases. a) Every first frame within the series of frames of a partition consists of only 1 row. b) The last partition and its one and only frame has only one row.
- The employee name of the row which is two rows 'ahead' of the current row. Similar as in the previous column this not possible in

all cases.
- Please notice the difference in the last two columns of the first row. The SECOND_ROW_IN_FRAME-column contains the NULL indicator. The frame which is associated with this row contains only 1 row (from the first to the current row) - and the scope of the nth_value() function is 'frame'. In contrast, the TWO_ROW_AHEAD-column contains the value 'Grace'. This value is evaluated by the lead() function, whose scope is the partition! The partition contains 3 rows: all rows within the department 'Management'. Only with the second and third row it becomes impossible to go 2 rows 'ahead'.

Determine Partition and Sequence

As shown in the above examples, the `WINDOW PARTITION clause` defines the partitions by using the key words PARTITION BY and the `WINDOW ORDER clause` defines the sequence of rows within the partition by using the key words ORDER BY.

Determine the Frame

The frames are defined by the `WINDOW FRAME clause`, which optionally follows the `WINDOW PARTITION clause` and the `WINDOW ORDER clause`.

With the exception of the lead() and lag() functions, whose scope is the actual partition, all other window functions act on the actual frame. Therefore it is an elementary decision, which rows shall constitute the frame. This is done by establishing the lower and upper boundary (in the sense of the `WINDOW ORDER clause`). All rows within this two bounds constitute the actual frame. Therefore the `WINDOW FRAME clause` consists mainly of the definition of the two boundaries - in one of four ways:

- Define a certain number of **rows** before and after the current row. This leads to a constant number of rows within the series of frames - with some exceptions near the lower and upper boundary and the exception of the use of the 'UNBOUNDED' key word.
- Define a certain number of **groups** before and after the current row. Such groups are build by the unique values of the preceding and following rows - in the same way as a `SELECT DISTINCT ...` or `GROUP BY`. The resulting frame covers all rows, whose values fall into one of the groups. As every group may be build out of multiple rows (with the same value), the number of rows per frame is not constant.
- Define a **range** for the values of a certain column by denoting a fix numerical value, eg: 1.000 (for a salary) or 30 days (for a time series). The thereby defined range runs from the differenz of the current value and the defined value up to the current value (the FOLLOWING-case builds the sum, not the differenz). All rows of the partition, whose column values fall into this range, constitute the frame. Accordingly the number of rows within the frame may differ from frame to frame - in opposite to the **rows** technic.
- Omit the clause and use default values.

In accordance with this different strategies there are three key words 'ROWS', 'GROUPS' and 'RANGE' which leads to the different behaviour.

Terminology

The `WINDOW FRAME clause` uses some key words whose semantic hopefully gets clear in the following block, where the ordered rows of a partition are visualised.

```
Rows in a partition and the according key words
-     <--   UNBOUNDED PRECEDING (first row)
...
-     <-- 2 PRECEDING
-     <-- 1 PRECEDING
-     <--   CURRENT ROW
-     <-- 1 FOLLOWING
-     <-- 2 FOLLOWING
...
-     <--   UNBOUNDED FOLLOWING (last row)
```

The term UNBOUNDED PRECEDING denotes the first row in a partition and UNBOUNDED FOLLOWING the last row. Counting from the CURRENT ROW there are <n> PRECEDING and <n> FOLLOWING rows. Obviously this PRECEDING/FOLLOWING terminology works only, if there is a `WINDOW ORDER clause` which creates an unambiguous sequence.

The (simplified) syntax of the `WINDOW FRAME clause` is:

```
<window_frame>  := [ROWS | GROUPS | RANGE ] BETWEEN
                   [ UNBOUNDED PRECEDING | <n> PRECEDING | CURRENT ROW ] AND
                   [ UNBOUNDED FOLLOWING | <n> FOLLOWING | CURRENT ROW ]
```

An example of a complete window function with its `WINDOW FRAME clause` is:

```
 ...
 SUM(salary)  OVER  (PARTITION BY dep_name ORDER BY salary
                     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) as growing_sum,
```

```
  ...
```

In this case the `WINDOW FRAME clause` starts with the key word 'ROWS'. It defines the lower boundary to the very first row of the partition and the upper boundary to the actual row. This means that the series of frames grows from frame to frame by one additional row until all rows of the partition are handled. Afterwards the next partition starts with an 1-row-frame and repeats the growing.

ROWS

The ROWS syntax defines a certain number of rows to process.

```
SELECT id, dep_name, salary,
       SUM(salary)  OVER  (PARTITION BY dep_name ORDER BY salary
                           ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS sum_over_1or2or3_rows
FROM   employee;
```

The example acts on a certain number of rows, namely the two rows before the current row (if existing within the partition) and the current row. There is no situation where more than three rows exists in one of the frames. The window function computes the sum of the salary over these maximal three rows.

The sum is reset to zero with every new partition, which is the department in this case. This holds true also for the GROUPS and RANGE syntax.

The ROWS syntax is often used when one is interested in the average about a certain number of rows or in the distance between two rows.

GROUPS

The GROUPS syntax has a similar semantic as the ROWS syntax - with one exception: rows with equal values within the column of the `WINDOW ORDER clause` count as 1 row. With other words, the GROUPS sytax counts the number of distinct values, not the number of rows.

```
-- Hint: The syntax 'GROUPS' (Feature T620) is not supported by Oracle 11
SELECT id, dep_name, salary,
       SUM(salary)  OVER  (PARTITION BY dep_name ORDER BY salary
                           GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS sum_over_groups
FROM   employee;
```

The example starts with the key word GROUPS and defines that it wants to work on 3 distinct values of the column 'salary'. Possibly there are more than three rows satisfying this criteria - in opposite to the equivalent ROWS strategy.

The GROUPS syntax is the appropriate strategy, if one has a varying number of rows within the time period under review, eg.: one has a varying number of measurement values per day and is interested in the average or the variance over a week or month.

RANGE

At a first glance the RANGE syntax is similar to the ROWS and GROUPS syntax. But the semantic is very different! Numbers <n> given in this syntax did not specify any counter. They specify the **distance** from the value in the current row to the lower or upper boundary. Therefor the ORDER BY column shall be of type NUMERIC, DATE or INTERVAL.

```
SELECT id, dep_name, salary,
       SUM(salary)  OVER  (PARTITION BY dep_name ORDER BY salary
                           RANGE BETWEEN 100 PRECEDING AND 50 FOLLOWING) AS sum_over_range
FROM   employee;
```

This definition leads to the sum over all rows which have a salary from 100 below and 50 over the actual row. In our example table this criteria appies in some rare cases to more than 1 row.

Typical use cases for the RANGE strategy are situations where someone analyzes a wide numeric range and expects to meet only few rows within this range, e.g.: a sparse matrix.

Defaults

If the `WINDOW FRAME clause` is omitted, its default value is: 'RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW'. This leads to a range from the very first row of the partition up the current row plus all rows with the same value as the current row - because the RANGE syntax applies.

If the `WINDOW ORDER clause` is omitted, the `WINDOW FRAME clause` is not allowed and all rows of the partition constitute the frame.

If the `PARTITION BY clause` is omitted, all rows of the result set constitues the one and only partition.

A Word of Caution

Although the SQL standard 2003 and his successors define very clear rules concerning window functions, several implementations did not follow them. Some implement only parts of the standard - which is their own responsibility -, but others seems to interpret the standard in a fanciful fashion.

As far we know, the ROWS syntax is implemented standard conform - if it is implemented. But it seems that the RANGE syntax sometimes implements what the GROUPS syntax of the SQL standard requires. (Perhaps this is a misrepresentation and only the public available descriptions of various implementations does not reflect the details.) So: be carefull, test your system and give us a feedback on the discussion page.

Exercises

Show id, emp_name, dep_name, salary and the average salary within the department.

Click to see solution

```
--
-- To get the average of the department, every frame must be build by ALL rows of the department.
--
SELECT id, emp_name, dep_name, salary,
       avg(salary)  OVER  (PARTITION BY dep_name ORDER BY dep_name
                           -- all rows of partition (=department)
                           ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as avg_salary
FROM   employee;
--
-- It's possible to omit the 'window order' clause. Thereby the frames include ALL rows of the actual partition.
-- See: 'Defaults' above.
--
SELECT id, emp_name, dep_name, salary,
       avg(salary)  OVER  (PARTITION BY dep_name) as avg_salary
FROM   employee;
--
-- The following statements leads to different results as the frames are composed by a growing number of rows.
--
SELECT id, emp_name, dep_name, salary,
       avg(salary)  OVER  (PARTITION BY dep_name ORDER BY salary) as avg_salary
FROM   employee;
--
-- It's possible to sort the result set by arbitrary rows (test the emp_name, it's interesting)
--
SELECT id, emp_name, dep_name, salary,
       avg(salary)  OVER  (PARTITION BY dep_name) as avg_salary
FROM   employee
ORDER BY dep_name, salary;
```

Does older persons earn more money than younger?
To give an answer show id, emp_name, salary, age and the average salary of 3 (or 5) persons, which are in a similar age.

Click to see solution

```
SELECT id, emp_name, salary, age,
       AVG(salary)  OVER (                     ORDER BY age ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS mean_over_3,
       AVG(salary)  OVER (                     ORDER BY age ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) AS mean_over_5
FROM   employee;
-- As there is no restriction to any other criterion than the age (department or something else), there is
-- no need for any PARTITION definition. Averages are computed without any interruption.
```

Extend the above question and its solution to show the results within the four departments.

Click to see solution

```
SELECT id, emp_name, salary, age, dep_name,
       AVG(salary)  OVER (PARTITION BY dep_name ORDER BY age ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS mean_over_3,
       AVG(salary)  OVER (PARTITION BY dep_name ORDER BY age ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) AS mean_over_5
FROM   employee;
-- Averages are computed WITHIN departments.
```

Show id, emp_name, salary and the difference to the salary of the previous person (in ID-order).

Click to see solution

```
-- For mathematician: This is a very first approximation to first derivate.
SELECT id, emp_name, salary,
       salary - LAG(salary)  OVER  (ORDER BY id) as diff_salary
FROM   employee;
-- And the difference of differences:
SELECT id, emp_name, salary,
       (LAG(salary)    OVER (ORDER BY id) - salary)                        AS diff_salary_1,
       (LAG(salary)    OVER (ORDER BY id) - salary) -
       (LAG(salary, 2) OVER (ORDER BY id) - LAG(salary) OVER (ORDER BY id)) AS diff_salary_2
```

```
FROM   employee;
```

Show the 'surrounding' of a value: id and emp_name of all persons ordered by emp_name. Supplement each row with the two emp_names before and the two after the actual emp_name (in the usual alphabetical order).

Click to see solution

```
SELECT id,
       LAG(emp_name, 2)  OVER (ORDER BY emp_name)    AS before_prev,
       LAG(emp_name)     OVER (ORDER BY emp_name)    AS prev,
       emp_name                                      AS act,
       LEAD(emp_name)    OVER (ORDER BY emp_name)    AS follower,
       LEAD(emp_name, 2) OVER (ORDER BY emp_name)    AS behind_follower
FROM   employee
ORDER BY emp_name;
```

Sometimes the rows of **one** table are structured in such a way that they represent a hierarchy or a network **within** this table. Typical use cases are management structures, bill of materials (a machine consists of a number of smaller machines, …) or network structures (e.g.: flight plans).

To retrieve particular rows and all rows that correlate to them, one can use set operations in combination with subqueries to merge them together to one result set. But this technique is limited as one must exactly know the number of levels. Apart from the fact that the number of levels changes from case to case, the subselect syntax differs from level to level. To overcome this restrictions SQL offers a syntax to express queries in a **recursive** manner. They retrieve the rows of all affected levels, independent from their number.

Syntax

The SQL standard uses a special form of its `WITH clause`, which is explained on the previous page, to define recursive queries. The clause occurs before a SELECT, INSERT, UPDATE or DELETE key word and is part of the appropriate command.

Hint: The `WITH clause` (with or without the 'RECURSIVE' key word) is often referred to as a 'common table expression (CTE)'.

Hint: Oracle supports the syntax of the SQL standard since version 11.2. . MySQL does not support recursions at all and recommend procedural workarounds.

```
-- The command starts with a 'with clause', which contains the optional 'RECURSIVE' key word.
WITH [RECURSIVE] intermediate_table (temp_column_name [,...]) AS
  (SELECT ... FROM real_table          -- initial query to a real table                    (1)
     UNION ALL                                                                           (3)
   SELECT ... FROM intermediate_table  -- repetitive query using the intermediate table    (2)
  )
-- The 'with clause' is part of a regular SELECT.
-- This SELECT refers to the final result of the 'with clause'.                          (4)
SELECT ... FROM intermediate_table
; -- consider the semicolon: the command runs from the 'WITH' up to here.
```

The evaluation sequence is as follows:

1. The initial query to a real table or a view is executed and creates the start point for step 2.
2. Usually the repetitive query consists of a join between the real table or view and the result set build up so far. This step is repeated until no new rows are found.
3. The result sets from step 1. and 2. are merged together.
4. The final SELECT acts on the result of step 3.

Example Table

To demonstrate recursive queries we define an example table. It holds information about persons and their ancestors. Because ancestors are always persons, everything is stored in the same table. *father_id* and *mother_id* acts as references to the rows where father's and mother's information is stored. The combination of *father_id*, *mother_id* and *firstname* acts as a criterion, which uniquely identifies rows according to those three values (we supose, that parents give their children different names).

```
CREATE TABLE family_tree (
  -- define columns (name / type / default value / nullable)
  id             DECIMAL      NOT NULL,
  firstname      VARCHAR(50)  NOT NULL,
  lastname       VARCHAR(50)  NOT NULL,
  year_of_birth  DECIMAL      NOT NULL,
  year_of_death  DECIMAL,
  father_id      DECIMAL,
  mother_id      DECIMAL,
  -- the primary key
  CONSTRAINT family_tree_pk   PRIMARY KEY (id),
  -- an additional criterion to uniquely distinguish rows from each other
```

```
  CONSTRAINT family_tree_uniq UNIQUE (father_id, mother_id, firstname),
  -- two foreign keys (to the same table in this special case) to ensure that no broken links arise
  CONSTRAINT family_tree_fk1  FOREIGN KEY (father_id) REFERENCES family_tree(id),
  CONSTRAINT family_tree_fk2  FOREIGN KEY (mother_id) REFERENCES family_tree(id),
  -- plausibility checks
  CONSTRAINT family_tree_check1 CHECK ( year_of_birth >= 1800 AND year_of_birth < 2100),
  CONSTRAINT family_tree_check2 CHECK ((year_of_death >= 1800 AND year_of_death < 2100) OR year_of_death IS NULL)
);

-- a fictional couple
INSERT INTO family_tree VALUES ( 1, 'Karl',   'Miller', 1855, 1905, null, null);
INSERT INTO family_tree VALUES ( 2, 'Lisa',   'Miller', 1851, 1912, null, null);
-- their children
INSERT INTO family_tree VALUES ( 3, 'Ruth',   'Miller', 1878, 1888, 1,    2);
INSERT INTO family_tree VALUES ( 4, 'Helen',  'Miller', 1880, 1884, 1,    2);
INSERT INTO family_tree VALUES ( 5, 'Carl',   'Miller', 1882, 1935, 1,    2);
INSERT INTO family_tree VALUES ( 6, 'John',   'Miller', 1883, 1900, 1,    2);
-- some more people; some of them are descendants of the Millers
INSERT INTO family_tree VALUES ( 7, 'Emily',  'Newton', 1880, 1940, null, null);
INSERT INTO family_tree VALUES ( 8, 'Charly', 'Miller', 1908, 1978, 5,    7);
INSERT INTO family_tree VALUES ( 9, 'Deborah','Brown',  1910, 1980, null, null);
INSERT INTO family_tree VALUES (10, 'Chess',  'Miller', 1948, null, 8,    9);
COMMIT;
```

Basic Queries

As a first example we retrieve Mr. Karl Miller and all his descendants. To do so, we must retrieve his own row and define a rule, how to 'navigate' from level to level within the family tree.

```
-- Choose a name for the intermediate table and its columns. The column names may differ from the names in the real table.
WITH intermediate_table (id, firstname, lastname) AS
(
  -- Retrieve the starting row (or rows)
  SELECT id, firstname, lastname
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  -- Define the rule for querying the next level. In most cases this is done with a join operation.
  SELECT f.id, f.firstname, f.lastname        -- the alias 'f' refers to the real table
  FROM   intermediate_table i                 -- the alias 'i' refers to the intermediate table
  JOIN   family_tree f ON f.father_id = i.id  -- the join operation defines, how to reach the next level
)
-- The final SELECT
SELECT * FROM intermediate_table
;

-- You can use all language features of SQL to further process the intermediate table. (It isn't a real table,
-- it is only an intermediate result with the structure of a table)
-- Example: count the number of descendants.

-- The 'with clause' keeps unchanged
...
-- The final SELECT
SELECT count(*) FROM intermediate_table
;
```

To demonstrate the problems in situations where no recursive SELECT is available, we show a syntax with subqueries.

```
-- This query retrieves only Mr. Karl Miller ...
SELECT *
FROM   family_tree
WHERE  firstname = 'Karl'
AND    lastname  = 'Miller'
  UNION ALL
-- ... and his children
SELECT *
FROM   family_tree
WHERE  father_id IN (SELECT id
                     FROM   family_tree
                     WHERE  firstname = 'Karl'
                     AND    lastname  = 'Miller'
                    )
;
```

Every level has its own syntax, e.g. to retrieve grandchildren we need a subquery within a subquery.

As a second example we traverse the hierarchie in the opposite direction: from a person to its male ancestors. In comparision to the first example two things changes. The start point of the query is no longer Mr. Karl Miller, as he has no ancestor in our example table. And we have to change the join condition by swapping id and father_id.

```
-- Retrieve ancestors
WITH intermediate_table (id, father_id, firstname, lastname) AS
(
  -- Retrieve the starting row (or rows)
  SELECT id, father_id, firstname, lastname  -- now we need the 'father_id'
  FROM   family_tree
  WHERE  firstname = 'Chess'
```

```
  AND    lastname  = 'Miller'
    UNION ALL
  -- Define the rule for querying the next level.
  SELECT f.id, f.father_id, f.firstname, f.lastname
  FROM   intermediate_table i
  JOIN   family_tree f ON f.id = i.father_id  -- compared with the first example this join operation defines the opposite direction
)
-- The final SELECT
SELECT * FROM intermediate_table
;
```

Notice the Level

Sometimes we need to know to which level within the hierarchy or network a row belongs to. To display this level we include a pseudo-column with an arbitrary name into the query. We choose the name *hier_level* (as *level* is a reserved word in the context of savepoints).

```
-- We extent the above example to show the hierarchy level
WITH intermediate_table (id, firstname, lastname, hier_level) AS
( SELECT id, firstname, lastname, 0 as hier_level         -- set the level of the start point to a fix number
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.firstname, f.lastname, i.hier_level + 1  -- increment the level
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
)
SELECT * FROM intermediate_table;
```

The level is now available and we can use it as an addition condition, eg. for a restriction to the first two levels.

```
-- The with clause remains unchanged
...
SELECT * FROM intermediate_table WHERE hier_level < 2; -- restrict the result to the first two levels

-- or, as with the above solution the intermediate result set is computed over ALL levels and later restricted to the first two:
WITH intermediate_table (id, firstname, lastname, hier_level) AS
( SELECT id, firstname, lastname, 0 as hier_level
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.firstname, f.lastname, i.hier_level + 1
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
  WHERE  hier_level < 1   -- restrict the join to the expected result
)
SELECT * FROM intermediate_table;
```

Create Paths

Sometimes we want to build a path from the starting point of the hierarchy or network to the actual row, eg. for a faceted classification like *1.5.3* or for a simple numbering of the visited nodes. This can be achieved in a similar way as the computing of the level. We need a pseudo-column with an arbitrary name and append actual values to those that have already been formed.

```
-- Save the path from person to person in an additional column. We choose the name 'hier_path' as its name.
WITH intermediate_table (id, firstname, lastname, hier_level, hier_path) AS
( SELECT id, firstname, lastname, 0 as hier_level, firstname as hier_path -- we collect the given names
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  -- The SQL standard knows only a two-parameter function concat(). We us it twice.
  SELECT f.id, f.firstname, f.lastname, i.hier_level + 1, concat (concat (i.hier_path, ' / ' ), f.firstname)
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
)
SELECT * FROM intermediate_table;
```

Depth First / Breads First

There are two ways to traverse hierarchies and networks. You must decide which kind of nodes you want to process first: child nodes (nodes of the next level) or sibling nodes (nodes of the same level). The two methodes are called *depth first* and *breath first*. With the key words `DEPTH FIRST` and `BREADTH FIRST` (the default) you can decide between the two variants.

```
<with_clause>
SEARCH [DEPTH FIRST|BREADTH FIRST] BY <column_name> SET <sequence_number>
<select_clause>
```

The key words occur between the `WITH clause` and the `SELECT clause`. As - in opposite to a tree in a programming language like JAVA or C++ or like an XML instance - rows of a table have no implicit order, you must define an order for the nodes within their level. This is done behind the `BY` key word. At last you have to define - after the `SET` key word - the name of an additional pseudo-column, where a numbering over all rows is stored automatically.

```
WITH intermediate_table (id, firstname, lastname, hier_level) AS
( SELECT id, firstname, lastname, 0 AS hier_level
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.firstname, f.lastname, i.hier_level + 1
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
)
-- SEARCH BREADTH FIRST BY firstname SET sequence_number
SEARCH DEPTH FIRST BY firstname SET sequence_number
SELECT * FROM intermediate_table;
```

There are some notable remarks to the above query:

1. In opposite to the other queries on this page (where we implicitely have used the default `BREADTH FIRST`), the family tree is traversed in such a way that after every row its 'child' rows are processed. This is significant at level 1.
2. If there is more than one row per level, the rows are ordered according to the `BY` definition: *firstname* in this case.
3. The rows have a sequence number: *sequence_number* in this case. You may use this number for any additional processing.

Exercises

Retrieve Chess Miller and all its **female** ancestors.

Click to see solution

```
WITH intermediate_table (id, mother_id, firstname, lastname) AS
(
  SELECT id, mother_id, firstname, lastname
  FROM   family_tree
  WHERE  firstname = 'Chess'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.mother_id, f.firstname, f.lastname
  FROM   intermediate_table i
  JOIN   family_tree f ON f.id = i.mother_id
)
SELECT * FROM intermediate_table;
```

Retrieve Chess Miller and all its ancestors: male and female.

Click to see solution

```
WITH intermediate_table (id, father_id, mother_id, firstname, lastname) AS
(
  SELECT id, father_id, mother_id, firstname, lastname
  FROM   family_tree
  WHERE  firstname = 'Chess'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.father_id, f.mother_id, f.firstname, f.lastname
  FROM   intermediate_table i
  -- extend the JOIN condition!
  JOIN   family_tree f ON (f.id = i.mother_id OR f.id = i.father_id)
)
SELECT * FROM intermediate_table;
```

To make the situation a little bit more transparent add a number to the previous query which shows the actual level.

Click to see solution

```
WITH intermediate_table (id, father_id, mother_id, firstname, lastname, hier_level) AS
(
  SELECT id, father_id, mother_id, firstname, lastname, 0  -- we start with '0'
  FROM   family_tree
  WHERE  firstname = 'Chess'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.father_id, f.mother_id, f.firstname, f.lastname, i.hier_level + 1
  FROM   intermediate_table i
  JOIN   family_tree f ON (f.id = i.mother_id OR f.id = i.father_id)
)
SELECT * FROM intermediate_table;
```

To make the situation absolutely transparent replace the level by some kind of path (child / parent / grandparent / ...).

Click to see solution

```
WITH intermediate_table (id, father_id, mother_id, firstname, lastname, ancestry) AS
(
  SELECT id, father_id, mother_id, firstname, lastname, firstname
  FROM   family_tree
  WHERE  firstname = 'Chess'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.father_id, f.mother_id, f.firstname, f.lastname, concat (concat (i.ancestry, ' / '), f.firstname)
  FROM   intermediate_table i
  JOIN   family_tree f ON (f.id = i.mother_id OR f.id = i.father_id)
)
SELECT * FROM intermediate_table;
```

Retrieve all grandchildren of Karl Miller.

Click to see solution

```
WITH intermediate_table (id, father_id, mother_id, firstname, lastname, hier_level) AS
(
  SELECT id, father_id, mother_id, firstname, lastname, 0   -- we start with '0'
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.father_id, f.mother_id, f.firstname, f.lastname, i.hier_level + 1
  FROM   intermediate_table i
  JOIN   family_tree f ON (f.father_id = i.id AND hier_level < 2) -- performance: abort joining after the second level
)
SELECT * FROM intermediate_table WHERE hier_level = 2; -- this is the restriction to the grandchildren
```

Retrieve every person in the table *family_tree* and show its firstname and the firstname of its very first known ancestor in the male line.

Click to see solution

```
WITH intermediate_table (id, father_id, firstname, lastname, initial_row, hier_level) AS
( -- The starting points are persons (more than one in our example table) for which no father is known.
  SELECT id, father_id, firstname, lastname, firstname, 0
  FROM   family_tree
  WHERE  father_id IS NULL
    UNION ALL
  -- The start name is preserved from level to level
  SELECT f.id, f.father_id, f.firstname, f.lastname, i.initial_row, i.hier_level + 1
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
)
SELECT * FROM intermediate_table;
-- or:
... unchanged 'with clause'
SELECT id, firstname, '-->', initial_row, 'in ', hier_level, 'generation(s)' FROM intermediate_table;
```

a) How much descandants of Carl Miller are stored in the example table?
b) Same question as before, but differentiated per level.

Click to see solution

```
-- a)
WITH intermediate_table (id, firstname, lastname, hier_level) AS
( SELECT id, firstname, lastname, 0 AS hier_level
  FROM   family_tree
  WHERE  firstname = 'Karl'
  AND    lastname  = 'Miller'
    UNION ALL
  SELECT f.id, f.firstname, f.lastname, i.hier_level + 1
  FROM   intermediate_table i
  JOIN   family_tree f ON f.father_id = i.id
)
SELECT count(*) FROM intermediate_table where hier_level > 0;
-- b) Use the same WITH clause. Only the final SELECT changes.
...
SELECT hier_level, count(hier_level) FROM intermediate_table WHERE hier_level > 0 GROUP BY hier_level;
```

The Problem

As mentioned in a previous chapter of this wikibook and in wikipedia sometimes there is no value in a column of a row, or - to say it the other way round - the column stores the *NULL marker* (a flag to indicate the absence of any data), or - to use the notion of the SQL standard - the column stores the *NULL value*. This NULL marker is very different from the numeric value zero or a string with a length of zero characters! Typically it occurs when an application yet hasn't stored anything in the column of this row.

(A hint to Oracle users: For Oracle the NULL marker is identical to a string of zero characters.)

The existence of the NULL marker introduces a **new fundamental problem**. In the usual boolean logic there are the two logical values

TRUE and FALSE. Every comparision evaluates to one of the two - and the comparisions negation evaluates to the opposite one. If a comparision evaluates to TRUE, its negation evaluates to FALSE and vice versa. As an example, in the usual boolean logic one of the following two comparisions is TRUE and the other one is FALSE: 'x < 5', 'x >= 5'.

Imagine now the new situation that x holds the NULL marker. It is not feasible that 'NULL < 5' is true (1). But if we say, 'NULL < 5' is false (2), its negation 'NULL >= 5' is true (3)! Is (3) more feasible than (1)? Of course not. (1) and (3) have the same 'degree of truth', so they shall evaluate to the same value. And this value must be different from TRUE and FALSE.

Therefore the usual boolean logic is extended by a third logic value. It is named **UNKOWN**. All comparisions to the NULL marker results per definition in this new value. And the well known statement 'if a statement is true, its negation is false' gets lost because there is a third option.

SQLs logic is an implementation of this so called trivalent, ternary or three-valued logic (3VL). The existence of the NULL marker in SQL is not without controversy. But if NULLs are accepted, the 3VL is a necessity.

This page proceeds in two stages: First it explains the handling of NULLs concerning comparsions, grouping, etc. . Second it explains the boolean logic for the cases where the new value UNKOWN interacts with any other boolean value - including itself.

Step 1: Evaluation of NULLs

Comparision Predicates, IS NULL Predicate

SQL knows the six comparision predicates <, <=, =, >=, > and <> (unequal). Their main purpose is the arithmetic comparision of numeric values. Each of them needs two variables or constants (infix notation). This implies that it is possible that one or even both operants hold the NULL marker. As stated before the common and very simple rule is: "All comparisions to the NULL marker results per definition in this new value (**unkown**).". Here are some examples:

- NULL = 5 evaluates to UNKNOWN.
- 5 = NULL evaluates to UNKNOWN.
- NULL <= 5 evaluates to UNKNOWN.
- col_1 = 5 evaluates to UNKNOWN for rows where col_1 holds the NULL marker.
- col_1 = col_2 evaluates to UNKNOWN for rows where col_1 or col_2 holds the NULL marker.
- NULL = NULL evaluates to UNKNOWN.
- col_1 = col_2 evaluates to UNKNOWN for rows where col_1 and col_2 holds the NULL marker.

The WHERE clause returns such rows where it evaluates to TRUE. It does not return rows where it evaluates to FALSE or to UNKNOWN. In consequence it is not garanteed that the following SELECT will return the complete table *t1*:

```
-- This SELECT will not return such rows where col_1 holds the NULL marker.
SELECT *
FROM   t1
WHERE  col_1 > 5
OR     col_1 = 5
OR     col_1 < 5;
```

Of course there are use cases where rows with the NULL marker must be retrieved. Because the arithmetic comparisions are not able to do so, another language construct must do the job. It is the *IS NULL predicate*.

```
-- This SELECT will return exactly these rows where col_1 holds the NULL marker.
SELECT *
FROM   t1
WHERE  col_1 IS NULL;
```

Other Predicates

For the other predicates there is no simple rule of thumb. They must be explained one after the other.

The IN predicate is a shortcut for a sequence of OR operations:

```
-- Shortcut for: col_1 = 3 OR col_1 = 18 OR col_1 = NULL
SELECT *
FROM   t1
WHERE  col_1 IN (3, 18, NULL);  -- the NULL case will never hit with the IN predicate!
-- a second example which is a little more complex
-- WHERE  col_1 IN (SELECT col_x FROM t2 WHERE id < 10);
```

Only the two comparisions 'col_1 = 3' and 'col_1 = 18' are able to retrieve rows (possibly many rows). The comparision 'col_1 = NULL' will never evaluate to TRUE. It's alway UNKNOW, even if col_1 holds the NULL marker. To retrieve those rows it's necessary - as shown above - to use the 'IS NULL' predicate.

The subselect of an EXISTS predicate evaluates to TRUE if the cardinality of the retrieved rows is greater than 0, and to FALSE if the cardinality is 0. It is not possible that the UNKNOWN value occurs.

```
-- The subselect to t2 can hit some rows - or not. If there are hits in the subselect, ALL rows of t1
-- are returned, else no rows of t1 are returned.
SELECT *                                -- The select to table t1
FROM   t1
WHERE  EXISTS
       (SELECT * FROM t2 WHERE id < 10); -- The subselect to table t2
```

The LIKE predicate compares a column with a regular expression. If the column contains the NULL marker, the LIKE predicate returns the UNKOWN value, what means that the row is not retrieved.

```
-- The LIKE retrieves NO rows if col_2 contains the NULL marker.
SELECT *
FROM   t1
WHERE  col_2 LIKE 'Hello %';
```

Predefined Functions

The aggregate functions `COUNT(<column_name>)`, `MIN(<column_name>)`, `MAX(<column_name>)`, `SUM(<column_name>)` and `AVG(<column_name>)` ignores such rows where <column_name> contains the NULL marker. On the other hand `COUNT(*)` includes all rows.

If a parameter of one of the scalar functions like `UPPER()`, `TRIM()`, `CONCAT()`, `ABS()`, `SQRT()`, `...` contains the NULL marker the resulting value is - in the most cases - the NULL marker.

Grouping

There are some situations where column values are compared to each other to answer the question, whether they are distinct. For usual numbers and strings the result of such decisions is obvious. But how shall the DBMS handle NULL markers? Are they distinct from each other, are they equal to each other or is there no answer to this question at all? To get results, which are expected by (nearly) every end user, the standard defines "Two null values are not distinct.", they build a single group.

`SELECT DISTINCT col_1 FROM t1;` retrieves one and only row for all rows where *col_1* holds the NULL marker.

`... GROUP BY col_1 ...;` builds one and only one group for all rows where *col_1* holds the NULL marker.

Step 2: Boolean Operations within 3VL

After we have seen how various comparisions and predicates on the NULL marker produces TRUE, FALSE and UNKNOWN it's necessary to explain the rules for the new logic value UNKNOWN.

Inspection

A first elementary operation is the inspection of a truth value: is it TRUE, FALSE or UNKNOWN? Analogous to the *IS NULL predicate* there are three additional predicates:

- IS [NOT] TRUE
- IS [NOT] FALSE
- IS [NOT] UNKNOWN

```
-- Check for 'UNKNOWN'
SELECT *
FROM   t1
WHERE  (col_1 = col_2) IS UNKNOWN;  -- parenthesis are not necessary

-- ... is semantically equivalent to
SELECT *
FROM   t1
WHERE  col_1 IS NULL
OR     col_2 IS NULL;
```

In the abstract syntax of logical systems *p* shall represent any of its truth values. Herein the new predicates evaluate according to the following table:

p	IS TRUE	IS FALSE	IS UNKNOWN		IS NOT TRUE	IS NOT FALSE	IS NOT UNKNOWN
TRUE	TRUE	FALSE	FALSE		FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE		TRUE	FALSE	TRUE
UNKNOWN	FALSE	FALSE	TRUE		TRUE	TRUE	FALSE

Please notice that all predicates leads to TRUE or FALSE and never to UNKNOWN.

NOT

The next operation is the negation of the new value. To which value evaluates 'NOT UNKNOWN'? The UNKNOWN value represents the impossibility to decide between TRUE and FALSE. It is not feasible that the negation of this impossibility leads to TRUE or FALSE. Likewise it is UNKNOWN.

```
-- Which rows will match?  (1)
SELECT *
FROM   t1
WHERE  NOT col_2 = NULL; -- 'col_2 = NULL' evaluates to UNKNOWN in all cases, see above.

-- Is this SELECT equivalent to the first one?  (2)
SELECT *
FROM   t1
  EXCEPT
SELECT *
FROM   t1
WHERE  col_2 = NULL;

-- No, it's different!! Independent from NULL markers in col_2, (1) retrieves
-- absolutely NO row and (2) retrieves ALL rows.
```

The above SELECT (1) will retrieve no rows as 'NOT col_2 = NULL' evaluates to the same as 'col_2 = NULL', namely UNKNOWN. And the SELECT (2) will retrieve all rows, as the part after EXCEPT will retrieve no rows, hence only the part before EXCEPT is relevant.

In the abstract syntax of logical systems *p* shall represent any of its truth values and NOT *p* its negation. Herein the following table applies:

p	NOT *p*
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

AND, OR

There are the two binary operations AND and OR. They evaluate as follows:

p	*q*	*p* AND *q*	*p* OR *q*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	UNKNOWN	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE
UNKNOWN	FALSE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

The precedence of the operations is defined as usual: IS predicate, NOT, AND, OR.

Some Examples

```
--
-- Add a new row to the test data base
INSERT INTO person (id, firstname, lastname)  -- Omit some columns to generate NULL markers
VALUES             (99, 'Tommy',   'Test');
```

```
COMMIT;

SELECT *
FROM   person
-- focus all tests to the new row
WHERE  id = 99                  -- (1): TRUE
AND                             -- (3): (1) AND (2) ==> TRUE AND UNKNOWN ==> UNKNOWN
(
       date_of_birth = NULL     -- (2): UNKNOWN
);                              -- no hit

SELECT *
FROM   person
WHERE  id = 99                  -- (1): TRUE
AND                             -- (3): (1) AND (2) ==> TRUE AND TRUE ==> TRUE
(
       date_of_birth IS NULL    -- (2): TRUE
);                              -- hit

SELECT *
FROM   person
WHERE  id = 99                  -- (1): TRUE
OR                              -- (3): (1) OR (2) ==> TRUE OR UNKNOWN ==> TRUE
(
       date_of_birth = NULL     -- (2): UNKNOWN
);                              -- hit

SELECT *
FROM   person
WHERE  id = 99                  -- (1): TRUE
AND                             -- (4): (1) AND (3) ==> TRUE AND FALSE ==> FALSE
(
NOT                             -- (3): NOT (2) ==> NOT TRUE ==> FALSE
       date_of_birth IS NULL    -- (2): TRUE

);                              -- no hit (same as AND date_of_birth IS NOT NULL)

-- Clean up the test database
DELETE FROM person WHERE id = 99;
COMMIT;
```

A transaction is an embracing of **one or more** SQL statements - especially of such statements, which write to the database such as INSERT, UPDATE or DELETE, but also the SELECT command can be part of a transaction. All writing statements must be part of a transaction. The purpose of transactions is the guarantee that the database changes only from one consistent state to another consistent state fading out all intermediate situations. This holds true also in critical situations such as parallel processing, disc crash, power failure, ... Transactions ensure the **database integrity**.

To do so they support four basic properties, which all in all are called the *ACID paradigm*.

Atomic	All SQL statements of the transaction take place or none.
Consistent	The sum of all data changes of a transaction transforms the database from one consistent state to another consistent state.
Isolated	The isolation level defines, which parts of uncommited transactions are visible to other sessions.
Durable	The database retains committed changes even if the system crashes afterwards.

Transaction Boundaries

As every SQL statement which writes to the database must be part of a transaction, the DBMS silently starts a transaction for every of them, if actually there is no transaction started. An alternative is that the application/session starts a transaction explicitly by the command `START TRANSACTION`.

All subsequent SQL commands are part of this transaction. The transaction remains until it is confirmed or rejected. The confirmation takes place with the command `COMMIT`, the rejection with the command `ROLLBACK`. Before the COMMIT or ROLLBACK command is submitted, the DBMS stores the results of every writing statement into an intermediate area where it is not visible to other sessions (see: Isolation Levels). Simultaneously with the COMMIT command all changes of this transaction ends up in the common database, are visible to every other session and the transaction terminates. If the COMMIT fails for any reason, it happens the same as when the session submits a ROLLBACK command: all changes of this transaction are discarded and the transaction terminates. Please notice, that a session can revert its complete writing actions, which are part of the actual transaction, by submitting the single command ROLLBACK.

An Example:

```
-- Begin the transaction with an explicit command (In general not necessary. Not supported by Oracle.)
START TRANSACTION;
-- Insert some rows
INSERT ... ;
-- Modify those rows or some other rows
UPDATE ... ;
-- Delete some rows
DELETE ... ;
-- If the COMMIT succeeds, the results of the above 3 commands have been transfered to the 'common'
-- database and thus 'published' to all other sessions.
COMMIT;
```

```
--
--
START TRANSACTION;
INSERT ... ;
UPDATE ... ;
DELETE ... ;
-- Discard INSERT, UPDATE and DELETE
ROLLBACK;
```

Savepoints

As transactions can cover a lot of statements, it is likely that runtime errors or logical errors arise. In some of such cases applications want to rollback only parts of the actual transaction and commit the rest or resume the processing a second time. To do so, it is possible to define internal transaction boundaries which reflects all processing from the start of the transaction up to this point in time. Such intermediate boundaries are called **savepoints**. COMMIT and ROLLBACK statements terminate the complete transaction including its savepoints.

```
-- Begin the transaction with an explicit command
START TRANSACTION;
--
INSERT ... ;
-- Define a savepoint
SAVEPOINT step_1;
--
UPDATE ... ;
-- Discard only the UPDATE. The INSERT remains.
ROLLBACK TO SAVEPOINT step_1;
-- try again (or do any other action)
UPDATE ... ;
-- confirm INSERT and the second UPDATE
COMMIT;
```

During the lifetime of a transaction a savepoint can be released if it's no longer needed. (At the end of the transaction it's implicitly released.)

```
-- ...
-- ...
RELEASE SAVEPOINT <savepoint_name>;
-- This has no effect to the results of previous INSERT, UPDATE or DELETE commands. It only eliminates the
-- possiblity to ROLLBACK TO SAVEPOINT <savepoint_name>.
```

Atomicity

Transactions guarantees that the results of **all** of its statements are handled on a logical level as **one single** operation. All writing statements have a temporary nature until the COMMIT command terminates successful.

This behaviour helps to ensure the logical integrity of bussiness logic. Eg: If one wants to transfer some amount of money from one account to another, at least two rows of the database must be modified. The first modification decreases the amount in one row and the second one increases it on a different row. If there is a disc crash or power failure between this two write-operations, the application has a problem. But the *atomicity property* of transactions guaranties that none of the write-operations reaches the database (in the case of any failure or a ROLLBACK) or all of them (in the case of a successfull COMMIT).

There are more detailed informations about the atomicity property at Wikipedia.

Consistency

Transactions guarantees that the database is in a consistent state after they terminate. This consistency occurrs at different levels:

- The data and all derived index entries are syncronized. In most cases data and index entries are stored at different areas within the database. Nevertheless after the end of a transaction both areas are updated (or none).
- Table constraints and column constraints may be violated during a transaction (by use of the DEFERRABLE key word) but not after its termination.
- There may be Primary and Foreign Keys. During a transaction the rules for Foreign Keys may be violated (by use of the DEFERRABLE key word) but not after its termination.
- The **logical** integrity of the database is **not** guaranteed! If in the above example of a bank account the application forgets to update the second row, problems will arise.

Isolation

In most situations there are a lot of sessions working simultaneously on the DBMS. They compete for their resources, especially for the data. As long as the data is not modified, this is no problem. The DBMS can deliver the data to all of them.

But if multiple sessions try to modify data at the same point in time, conflicts are inescapable. Here is the timeline of an example with two sessions working on a flight reservation system. Session S1 reads the number of free seats for a flight: 1 free seat. S2 reads the number of free seats for the same flight: 1 free seat. S1 reserves the last seat. S2 reserves the last seat.

The central result of the analysis of such conflicts is that all of them are avoidable, if all transactions (concerning the same data) run sequentially: one after the other. But it's obvious that such a behavior is less efficent. The overall performance is increased if the DBMS does as much work as possible in parallel. The SQL standard offers a systematic of such conflicts and the command `SET TRANSACTION ...` to resolve them with the aim to allow parallel operations as much as possible.

Classification of Isolation Problems

The standard identifies three problematic situations:

- P1 (**Dirty read**): "SQL-transaction T1 modifies a row. SQL-transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed and that may thus be considered to have never existed." [1]
- P2 (**Non-repeatable read**): "SQL-transaction T1 reads a row. SQL-transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted." [1] Non-repeatable reads concern single rows.
- P3 (**Phantom**): "SQL-transaction T1 reads the set of rows N that satisfy some search condition. SQL transaction T2 then executes SQL-statements that generate one or more rows that satisfy the search condition used by SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same search condition, it obtains a different collection of rows." [1] Phantoms concern result sets.

Avoidance of Isolation Problems

Depending on the requirements and access strategy of an application some of the above problems may be tolerable - others not. The standard offers the `SET TRANSACTION ...` command to define, which are allowed to occur within a transaction and which not. The `SET TRANSACTION ...` command must be the first statement within a transaction.

```
-- define (un)tolerable conflict situations (Oracle does not support all of them)
SET TRANSACTION ISOLATION LEVEL [READ UNCOMMITTED |
                                 READ COMMITTED   |
                                 REPEATABLE READ  |
                                 SERIALIZABLE];
```

The following table shows which problems may occur within each level.

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

At Wikipedia there are more detailed informatiton and examples about isolation levels and concurrency control.

Durability

Transactions guarantees that every confirmed write-operation will survive (almost) every following disaster. To do so, in most cases the DBMS writes the changes not only to the database but additionally to logfiles, which shall reside on different devices. So it is possible - after a disc crash - to restore all changes from a database backup plus these logfiles.

There are more detailed informations about the durability property at Wikipedia.

Autocommit

Some DBMS offers - outside of the standard - an AUTOCOMMIT feature. If it is activated, the feature submits automatically a COMMIT command after every writing statement with the consequence that you cannot ROLLBACK a logical unit-of-work consisting of a lot of SQL statements. Furthermore the use of the SAVEPOINT feature is not possible.

In much cases the feature is activated by default.

References

1. "ISO/IEC 9075-2:2011: Information technology -- Database languages -- SQL -- Part 2: Foundation (SQL/Foundation)". http://www.iso.org/iso/catalogue_detail.htm?csnumber=53682.

Appendices

ACID	An acronym for the 4 properties *atomicity*, *consistency*, *isolation* and *durability*. Any transaction must conform to them. *Atomicity* means that either all or no data modification will take place. *Consistency* ensures that transactions transforms the database from one valid state to another valid state. *Isolation* requires that transactions will not affect each other, even if they run at the same time. *Durability* means that the modifications will keep into the database even if the system crashs immediately after the transaction. q.v.: ACID
Attribute	A set of properties (name, datatype, size, ...) used to characterize the data items of entities. A group of attributes constructs an entity-type (or table), i.e.: all values of a certain column must conform to the same attributes. Attributes are optionally complemented by constraints.
Block	Aggregation of one or more physical blocks of a mass device. Usually a block contains numerous rows of one or more tables. Sometimes one row is distributed across several blocks. q.v.: dirty block
Clause	A certain language element as part of a statement. E.g.: the *WHERE clause* defines seach criterias.
Column	A set of values of a single table which resides on the same position within its rows.
Constraint	Similar to attributes constraints define rules at a higher level, data items must conform to. E.g.: nullability, primary and foreign key, uniqueness, default value, user-defined-criterias like `STATUS < 10`.
Cursor	A cursor is a mechanism by which the rows of a table may be acted on (e.g., returned to a host programming language) one at a time.
Database	A set of tables. Those tables contain user data and the Data Dictionary.
Database Management System (DBMS)	A set of computer programs that controls the creation, maintenance and usage of the database. q.v.: DBMS
Data Dictionary (DD)	A set of predefined tables where the DBMS stores information about all user defined objects (tables, views, constraints, ...).
Data Control Language (DCL)	A class of statements which defines the access rights to data, e.g: `GRANT ..., REVOKE, ...` .
Data Definition Language (DDL)	A class of statements which defines logical and physical design of a database, e.g.: `CREATE TABLE ...` .
Data Manipulation Language (DML)	A class of statements which retrieves and manipulates data, e.g.: `SELECT ..., INSERT ..., UPDATE ..., DELETE ..., COMMIT, ROLLBACK`.
Dirty Block	A block whose content has changed in memory, but is still not written to disc.
Entity	An identifiable object like an *employee* or a *department*. An entity is an instance of an entity-type. Usually there are many instances of a certain entity-type. Every entity is stored in one row. Entities of same entity-type are stored in rows of the same table. So entities are a logical construct and rows a physical implementation.
Entity-type	A group of attributes describing the structure of entities. As entities of same entity-type are stored in rows of the same table it can be said, that an entity-type describes a table. (Many people tend to use the term entity as a synonym for entity-type.)
Expression	A certain language element as part of a statement. It can produce either scalar values or a table.
Foreign key	A value used to reference a primary key. It can point to any primary key in the database, whether in its own table (eg: bill of materials) or another table. It can point to its own row.
Index	An index is a construct containing copies of original values and backreferences to their original rows. It's purpose is the provision of a fast access to the original data. To achieve this, an index contains some kind of collocation. Remark: Indexes are not part of the SQL standard. Nevertheless they are part of nearly every DBMS.
Junction table	If more than one row of table T1 refers to more than one row of table T2 (many-to-many relationship) you need an intermediate table to store this relationship. The rows of the intermediate table contains the primary keys of T1 and T2 as values. q.v.: Junction_table
Normalization	Tables should conform to special rules - namely *First-*, *Second-* and *Third-Normal Form*. The process of rearranging columns over tables is called *normalization*.
NULL	If **no** value is stored in the column of a row, the standard says, that the *null value* is stored. As this *null value* is a flag and not a real value we use the term *null marker* within this wikibook. The *null marker* is used to indicate the absence of any data. For example it makes a difference whether a temperature is measured and stored as 0 degrees or whether the temperature is not measured and hence not stored. One consequence of the existence of the *null marker* is that SQL must know not only the boolean values TRUE and FALSE but also a third one: UNKNOWN.

Predicate	A language element which specifies a non arithmetic condition. E.g: [NOT] IN, [NOT] LIKE, IS [NOT] NULL, [NOT] EXISTS, ANY, ...
Primary key	A value or a set of values used to identify a single row uniquely.
Query	An often used statement which retrieves data from the database. It is introduced by the keyword SELECT and usually contains a predicate.
Relationship	A reference between two different or the same entity. References are not implemented as links. They base upon the values of the entities.
Relational Model	A method (and a mathematical theory) to model data as tables (relations), the relationships among each other and all operations on the data.
Row	One record in a table containing information about one single entity. A row has exactly one value for each of its columns - in accordance with *First Normal Form*. This value may be NULL.
Statement	A single command which is executed by the DBMS. There are 3 main classes of statements: DML, DDL and DCL.
Table (=Relation)	A set of rows of a certain entity-type, i.e. all rows of a certain table have the same structure.
Three Valued Logic (3VL)	SQL knows three boolean values: TRUE, FALSE and UNKNOWN. See: NULL. q.v.: trivalent, ternary or three-valued logic (3VL).
Transaction	A logical unit of work consisting of one or more modifications to the database. The ACID criterium must be achieved. A transaction is either saved by the `COMMIT` statement or completely canceled by the `ROLLBACK` statement.
Value	Implementation of a single data item within a certain column of a certain row. (You can think of a cell within a spreadsheet.)
View	A virtual table containing only its definition and no real data. The definition consists of a query to one or more real tables or views. Queries to the view are processed as queries to the underlying real tables.

Some of the above terms correlate to each other at the logical and implemention level.

Logical Design	**Implementation**
entity-type	table
entity	row
?	column
data item	value

First versions of the SQL standard used a variable called SQLCODE to flag special processing situations like exceptions, warnings or regular termination. SQLCODE is no longer part of the standard and is replaced by SQLSTATE.

SQLSTATE values consist of 5 characters where the first two denotes a **class** and the following three a **subclass**. The following table lists such values of SQLSTATE which are part of the standard. Implementations usually use much more values than those defined by the standard.

SQLSTATE values belong to one of four **categories**: "S" denotes "Success" (class 00), "W" denotes "Warning" (class 01), "N" denotes "No data" (class 02) and "X" denotes "Exception" (other classes).

Cat.	Class	Class Text	Subclass	Subclass Text
S	00	successful completion	000	(no subclass)
W	01	warning	000	(no subclass)
W	01	warning	001	cursor operation conflict
W	01	warning	002	disconnect error
W	01	warning	003	null value eliminated in set function
W	01	warning	004	string data, right truncation
W	01	warning	005	insufficient item descriptor areas
W	01	warning	006	privilege not revoked
W	01	warning	007	privilege not granted
W	01	warning	009	search condition too long for information schema
W	01	warning	00A	query expression too long for information schema
W	01	warning	00B	default value too long for information schema
W	01	warning	00C	result sets returned
W	01	warning	00D	additional result sets returned
W	01	warning	00E	attempt to return too many result sets
W	01	warning	00F	statement too long for information schema
W	01	warning	012	invalid number of conditions
W	01	warning	02F	array data, right truncation
N	02	no data	000	(no subclass)
N	02	no data	001	no additional result sets returned
X	07	dynamic SQL error	000	(no subclass)
X	07	dynamic SQL error	001	using clause does not match dynamic parameter specifications
X	07	dynamic SQL error	002	using clause does not match target specifications
X	07	dynamic SQL error	003	cursor specification cannot be executed
X	07	dynamic SQL error	004	using clause required for dynamic parameters
X	07	dynamic SQL error	005	prepared statement not a cursor specification
X	07	dynamic SQL error	006	restricted data type attribute violation
X	07	dynamic SQL error	007	using clause required for result fields
X	07	dynamic SQL error	008	invalid descriptor count
X	07	dynamic SQL error	009	invalid descriptor index
X	07	dynamic SQL error	00B	data type transform function violation
X	07	dynamic SQL error	00C	undefined DATA value
X	07	dynamic SQL error	00D	invalid DATA target
X	07	dynamic SQL error	00E	invalid LEVEL value
X	07	dynamic SQL error	00F	invalid DATETIME_INTERVAL_CODE
X	08	connection exception	000	(no subclass)
X	08	connection exception	001	SQL-client unable to establish SQL-connection
X	08	connection exception	002	connection name in use
X	08	connection exception	003	connection does not exist
X	08	connection exception	004	SQL-server rejected establishment of SQL-connection
X	08	connection exception	006	connection failure
X	08	connection exception	007	transaction resolution unknown
X	09	triggered action exception	000	(no subclass)

X	0A	feature not supported	000	(no subclass)
X	0A	feature not supported	001	multiple server transactions
X	0D	invalid target type specification	000	(no subclass)
X	0E	invalid schema name list specification	000	(no subclass)
X	0F	locator exception	000	(no subclass)
X	0F	locator exception	001	invalid specification
X	0L	invalid grantor	000	(no subclass)
X	0M	invalid SQL-invoked procedure reference	000	(no subclass)
X	0P	invalid role specification	000	(no subclass)
X	0S	invalid transform group name specification	000	(no subclass)
X	0T	target table disagrees with cursor specification	000	(no subclass)
X	0U	attempt to assign to non-updatable column	000	(no subclass)
X	0V	attempt to assign to ordering column	000	(no subclass)
X	0W	prohibited statement encountered during trigger execution	000	(no subclass)
X	0W	prohibited statement encountered during trigger execution	001	modify table modified by data change delta table
X	0Z	diagnostics exception	000	(no subclass)
X	0Z	diagnostics exception	001	maximum number of stacked diagnostics areas exceeded
X	21	cardinality violation	000	(no subclass)
X	22	data exception	000	(no subclass)
X	22	data exception	001	string data, right truncation
X	22	data exception	002	null value, no indicator parameter
X	22	data exception	003	numeric value out of range
X	22	data exception	004	null value not allowed
X	22	data exception	005	error in assignment
X	22	data exception	006	invalid interval format
X	22	data exception	007	invalid datetime format
X	22	data exception	008	datetime field overflow
X	22	data exception	009	invalid time zone displacement value
X	22	data exception	00B	escape character conflict
X	22	data exception	00C	invalid use of escape character
X	22	data exception	00D	invalid escape octet
X	22	data exception	00E	null value in array target
X	22	data exception	00F	zero-length character string
X	22	data exception	00G	most specific type mismatch
X	22	data exception	00H	sequence generator limit exceeded
X	22	data exception	00P	interval value out of range
X	22	data exception	00Q	multiset value overflow
X	22	data exception	010	invalid indicator parameter value
X	22	data exception	011	substring error
X	22	data exception	012	division by zero
X	22	data exception	013	invalid preceding or following size in window function
X	22	data exception	014	invalid argument for NTILE function
X	22	data exception	015	interval field overflow
X	22	data exception	016	invalid argument for NTH_VALUE function
X	22	data exception	018	invalid character value for cast

X	22	data exception	019	invalid escape character
X	22	data exception	01B	invalid regular expression
X	22	data exception	01C	null row not permitted in table
X	22	data exception	01E	invalid argument for natural logarithm
X	22	data exception	01F	invalid argument for power function
X	22	data exception	01G	invalid argument for width bucket function
X	22	data exception	01H	invalid row version
X	22	data exception	01S	invalid Query regular expression
X	22	data exception	01T	invalid Query option flag
X	22	data exception	01U	attempt to replace a zero-length string
X	22	data exception	01V	invalid Query replacement string
X	22	data exception	01W	invalid row count in fetch first clause
X	22	data exception	01X	invalid row count in result offset clause
X	22	data exception	021	character not in repertoire
X	22	data exception	022	indicator overflow
X	22	data exception	023	invalid parameter value
X	22	data exception	024	unterminated C string
X	22	data exception	025	invalid escape sequence
X	22	data exception	026	string data, length mismatch
X	22	data exception	027	trim error
X	22	data exception	029	noncharacter in UCS string
X	22	data exception	02D	null value substituted for mutator subject parameter
X	22	data exception	02E	array element error
X	22	data exception	02F	array data, right truncation
X	22	data exception	02G	invalid repeat argument in a sample clause
X	22	data exception	02H	invalid sample size
X	23	integrity constraint violation	000	(no subclass)
X	23	integrity constraint violation	001	restrict violation
X	24	invalid cursor state	000	(no subclass)
X	25	invalid transaction state	000	(no subclass)
X	25	invalid transaction state	001	active SQL-transaction
X	25	invalid transaction state	002	branch transaction already active
X	25	invalid transaction state	003	inappropriate access mode for branch transaction
X	25	invalid transaction state	004	inappropriate isolation level for branch transaction
X	25	invalid transaction state	005	no active SQL-transaction for branch transaction
X	25	invalid transaction state	006	read-only SQL-transaction
X	25	invalid transaction state	007	schema and data statement mixing not supported
X	25	invalid transaction state	008	held cursor requires same isolation level
X	26	invalid SQL statement name	000	(no subclass)
X	27	triggered data change violation	000	(no subclass)
X	27	triggered data change violation	001	modify table modified by data change delta table
X	28	invalid authorization specification	000	(no subclass)
X	2B	dependent privilege descriptors still exist	000	(no subclass)
X	2C	invalid character set name	000	(no subclass)
X	2D	invalid transaction termination	000	(no subclass)
X	2E	invalid connection name	000	(no subclass)
X	2F	SQL routine exception	000	(no subclass)

X	2F	SQL routine exception	002	modifying SQL-data not permitted
X	2F	SQL routine exception	003	prohibited SQL-statement attempted
X	2F	SQL routine exception	004	reading SQL-data not permitted
X	2F	SQL routine exception	005	function executed no return statement
X	2H	invalid collation name	000	(no subclass)
X	30	invalid SQL statement identifier	000	(no subclass)
X	33	invalid SQL descriptor name	000	(no subclass)
X	34	invalid cursor name	000	(no subclass)
X	35	invalid condition number	000	(no subclass)
X	36	cursor sensitivity exception	000	(no subclass)
X	36	cursor sensitivity exception	001	request rejected
X	36	cursor sensitivity exception	002	request failed
X	38	external routine exception	000	(no subclass)
X	38	external routine exception	001	containing SQL not permitted
X	38	external routine exception	002	modifying SQL-data not permitted
X	38	external routine exception	003	prohibited SQL-statement attempted
X	38	external routine exception	004	reading SQL-data not permitted
X	39	external routine invocation exception	000	(no subclass)
X	39	external routine invocation exception	004	null value not allowed
X	3B	savepoint exception	000	(no subclass)
X	3B	savepoint exception	001	invalid specification
X	3B	savepoint exception	002	too many
X	3C	ambiguous cursor name	000	(no subclass)
X	3D	invalid catalog name	000	(no subclass)
X	3F	invalid schema name	000	(no subclass)
X	40	transaction rollback	000	(no subclass)
X	40	transaction rollback	001	serialization failure
X	40	transaction rollback	002	integrity constraint violation
X	40	transaction rollback	003	statement completion unknown
X	40	transaction rollback	004	triggered action exception
X	42	syntax error or access rule violation	000	(no subclass)
X	44	with check option violation	000	(no subclass)

License

GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software

manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.3
> or any later version published by the Free Software Foundation;
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.