1 SQL – Structured Query Language

1.1 Tables

In relational database systems (DBS) data are represented using *tables* (*relations*). A query issued against the DBS also results in a table. A table has the following structure:

Column 1	Column 2	...	Column n
...	...	...	...

⟵ Tuple (or Record)

A table is uniquely identified by its name and consists of *rows* that contain the stored information, each row containing exactly one *tuple* (or *record*). A table can have one or more columns. A *column* is made up of a column name and a data type, and it describes an attribute of the tuples. The structure of a table, also called *relation schema*, thus is defined by its attributes. The type of information to be stored in a table is defined by the data types of the attributes at table creation time.

SQL uses the terms *table, row*, and *column* for *relation, tuple*, and *attribute*, respectively. In this tutorial we will use the terms interchangeably.

A table can have up to 254 columns which may have different or same data types and sets of values (domains), respectively. Possible domains are alphanumeric data (strings), numbers and date formats. ORACLE offers the following basic data types:

- **char**(n): Fixed-length character data (string), n characters long. The maximum size for n is 255 bytes (2000 in ORACLE8). Note that a string of type **char** is always padded on right with blanks to full length of n. (☞ can be memory consuming).
  Example: **char**(40)
- **varchar2**(n): Variable-length character string. The maximum size for n is 2000 (4000 in ORACLE8). Only the bytes used for a string require storage. *Example:* **varchar2**(80)
- **number**(o, d): Numeric data type for integers and reals. o = overall number of digits, d = number of digits to the right of the decimal point.
  Maximum values: o =38, d= −84 to +127. *Examples:* **number**(8), **number**(5,2)
  Note that, e.g., **number**(5,2) cannot contain anything larger than 999.99 without resulting in an error. Data types derived from **number** are **int[eger]**, **dec[imal]**, **smallint** and **real**.
- **date**: Date data type for storing date and time.
  The default format for a date is: DD-MMM-YY. *Examples*: '13-OCT-94', '07-JAN-98'

- **long**: Character data up to a length of 2GB. Only one **long** column is allowed per table.

Note: In ORACLE-SQL there is no data type **boolean**. It can, however, be simulated by using either **char**(1) or **number**(1).

As long as no constraint restricts the possible values of an attribute, it may have the special value *null* (for unknown). This value is different from the number 0, and it is also different from the empty string `''`.

Further properties of tables are:

- the order in which tuples appear in a table is not relevant (unless a query requires an explicit sorting).
- a table has no duplicate tuples (depending on the query, however, duplicate tuples can appear in the query result).

A *database schema* is a set of relation schemas. The extension of a *database schema* at database run-time is called a *database instance* or *database*, for short.

1.1.1 Example Database

In the following discussions and examples we use an example database to manage information about employees, departments and salary scales. The corresponding tables can be created under the UNIX shell using the command **demobld**. The tables can be dropped by issuing the command **demodrop** under the UNIX shell.

The table `EMP` is used to store information about employees:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	30
.....						
7698	BLAKE	MANAGER		01-MAY-81	3850	30
7902	FORD	ANALYST	7566	03-DEC-81	3000	10

For the attributes, the following data types are defined:

`EMPNO`:**number**(4), `ENAME`:**varchar2**(30), `JOB`:**char**(10), `MGR`:**number**(4), `HIREDATE`:**date**, `SAL`:**number**(7,2), `DEPTNO`:**number**(2)

Each row (tuple) from the table is interpreted as follows: an employee has a number, a name, a job title and a salary. Furthermore, for each employee the number of his/her manager, the date he/she was hired, and the number of the department where he/she is working are stored.

The table DEPT stores information about departments (number, name, and location):

DEPTNO	DNAME	LOC
10	STORE	CHICAGO
20	RESEARCH	DALLAS
30	SALES	NEW YORK
40	MARKETING	BOSTON

Finally, the table SALGRADE contains all information about the salary scales, more precisely, the maximum and minimum salary of each scale.

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

1.2 Queries (Part I)

In order to retrieve the information stored in the database, the SQL query language is used. In the following we restrict our attention to simple SQL queries and defer the discussion of more complex queries to Section 1.5

In SQL a query has the following (simplified) form (components in brackets [ ] are optional):

select [**distinct**] <column(s)>
from <table>
[**where** <condition>]
[**order by** <column(s) [**asc**|**desc**]>]

1.2.1 Selecting Columns

The columns to be selected from a table are specified after the keyword **select**. This operation is also called *projection*. For example, the query

select LOC, DEPTNO **from** DEPT;

lists only the number and the location for each tuple from the relation DEPT. If all columns should be selected, the asterisk symbol "∗" can be used to denote all attributes. The query

select ∗ **from** EMP;

retrieves all tuples with all columns from the table EMP. Instead of an attribute name, the **select** clause may also contain arithmetic expressions involving arithmetic operators etc.

select ENAME, DEPTNO, SAL ∗ 1.55 **from** EMP;

For the different data types supported in ORACLE, several operators and functions are provided:

- for numbers: **abs, cos, sin, exp, log, power, mod, sqrt**, $+, -, *, /, \ldots$
- for strings: **chr**, **concat**(string1, string2), **lower, upper**, **replace**(string, search_string, replacement_string), **translate**, **substr**(string, m, n), **length**, **to_date**, ...
- for the date data type: **add_month, month_between, next_day, to_char**, ...

The usage of these operations is described in detail in the SQL*Plus help system (see also Section 2).

Consider the query

select DEPTNO **from** EMP;

which retrieves the department number for each tuple. Typically, some numbers will appear more than only once in the query result, that is, duplicate result tuples are not automatically eliminated. Inserting the keyword **distinct** after the keyword **select**, however, forces the elimination of duplicates from the query result.

It is also possible to specify a sorting order in which the result tuples of a query are displayed. For this the **order by** clause is used and which has one or more attributes listed in the **select** clause as parameter. **desc** specifies a descending order and **asc** specifies an ascending order (this is also the default order). For example, the query

select ENAME, DEPTNO, HIREDATE **from** EMP;
from EMP
order by DEPTNO [**asc**], HIREDATE **desc**;

displays the result in an ascending order by the attribute DEPTNO. If two tuples have the same attribute value for DEPTNO, the sorting criteria is a descending order by the attribute values of HIREDATE. For the above query, we would get the following output:

ENAME	DEPTNO	HIREDATE
FORD	10	03-DEC-81
SMITH	20	17-DEC-80
BLAKE	30	01-MAY-81
WARD	30	22-FEB-81
ALLEN	30	20-FEB-81
.....		

1.2.2 Selection of Tuples

Up to now we have only focused on selecting (some) attributes of all tuples from a table. If one is interested in tuples that satisfy certain conditions, the **where** clause is used. In a **where** clause simple conditions based on comparison operators can be combined using the logical connectives **and**, **or**, and **not** to form complex conditions. Conditions may also include pattern matching operations and even subqueries (Section 1.5).

Example: List the job title and the salary of those employees whose manager has the number 7698 or 7566 and who earn more than 1500:

select JOB, SAL
from EMP
where (MGR = 7698 **or** MGR = 7566) **and** SAL > 1500;

For all data types, the comparison operators =, != or <>, <, >, <=, => are allowed in the conditions of a **where** clause.

Further comparison operators are:

- *Set Conditions:* <column> [**not**] **in** (<list of values>)
  Example: **select** ∗ **from** DEPT **where** DEPTNO **in** (20,30);
- *Null value:* <column> **is** [**not**] **null**,
  i.e., for a tuple to be selected there must (not) exist a defined value for this column.
  Example: **select** ∗ **from** EMP **where** MGR **is not null**;
  Note: the operations = **null** and ! = **null** are not defined!
- *Domain conditions:* <column> [**not**] **between** <lower bound> **and** <upper bound>
  Example: • **select** EMPNO, ENAME, SAL **from** EMP
  where SAL **between** 1500 **and** 2500;
  • **select** ENAME **from** EMP
  where HIREDATE **between** '02-APR-81' **and** '08-SEP-81';

1.2.3 String Operations

In order to compare an attribute with a string, it is required to surround the string by apostrophes, e.g., **where** LOCATION = 'DALLAS'. A powerful operator for pattern matching is the **like** operator. Together with this operator, two special characters are used: the percent sign % (also called wild card), and the underline _, also called position marker. For example, if one is interested in all tuples of the table DEPT that contain two C in the name of the department, the condition would be **where** DNAME **like** '%C%C%'. The percent sign means that any (sub)string is allowed there, even the empty string. In contrast, the underline stands for exactly one character. Thus the condition **where** DNAME **like** '%C_C%' would require that exactly one character appears between the two Cs. To test for inequality, the **not** clause is used.

Further string operations are:

- **upper**(<string>) takes a string and converts any letters in it to uppercase, e.g., DNAME = **upper**(DNAME) (*The name of a department must consist only of upper case letters.*)
- **lower**(<string>) converts any letter to lowercase,
- **initcap**(<string>) converts the initial letter of every word in <string> to uppercase.
- **length**(<string>) returns the length of the string.
- **substr**(<string>, n [, m]) clips out a m character piece of <string>, starting at position n. If m is not specified, the end of the string is assumed.
  substr('DATABASE SYSTEMS', 10, 7) returns the string 'SYSTEMS'.

1.2.4 Aggregate Functions

Aggregate functions are statistical functions such as **count**, **min**, **max** etc. They are used to compute a single value from a set of attribute values of a column:

count Counting Rows
Example: How many tuples are stored in the relation EMP?
select count(∗) **from** EMP;
Example: How many different job titles are stored in the relation EMP?
select count(**distinct** JOB) **from** EMP;

max Maximum value for a column
min Minimum value for a column
Example: List the minimum and maximum salary.
select min(SAL), **max**(SAL) **from** EMP;
Example: Compute the difference between the minimum and maximum salary.
select max(SAL) - **min**(SAL) **from** EMP;

sum Computes the sum of values (only applicable to the data type **number**)
Example: Sum of all salaries of employees working in the department 30.
select sum(SAL) **from** EMP
where DEPTNO = 30;

avg Computes average value for a column (only applicable to the data type **number**)

Note: **avg, min** and **max** ignore tuples that have a null value for the specified attribute, but **count** considers null values.

1.3 Data Definition in SQL

1.3.1 Creating Tables

The SQL command for creating an empty table has the following form:

create table <table> (
<column 1> <data type> [**not null**] [**unique**] [<column constraint>],
. . . . . .
<column n> <data type> [**not null**] [**unique**] [<column constraint>],
[<table constraint(s)>]
);

For each column, a name and a data type must be specified and the column name must be unique within the table definition. Column definitions are separated by colons. There is no difference between names in lower case letters and names in upper case letters. In fact, the only place where upper and lower case letters matter are strings comparisons. A **not null**

constraint is directly specified after the data type of the column and the constraint requires defined attribute values for that column, different from *null*.

The keyword **unique** specifies that no two tuples can have the same attribute value for this column. Unless the condition **not null** is also specified for this column, the attribute value *null* is allowed and two tuples having the attribute value *null* for this column do not violate the constraint.

Example: The **create table** statement for our EMP table has the form

create table EMP (
EMPNO **number**(4) **not null**,
ENAME **varchar2**(30) **not null**,
JOB **varchar2**(10),
MGR **number**(4),
HIREDATE **date**,
SAL **number**(7,2),
DEPTNO **number**(2)
);

Remark: Except for the columns EMPNO and ENAME null values are allowed.

1.3.2 Constraints

The definition of a table may include the specification of integrity constraints. Basically two types of constraints are provided: *column constraints* are associated with a single column whereas *table constraints* are typically associated with more than one column. However, any column constraint can also be formulated as a table constraint. In this section we consider only very simple constraints. More complex constraints will be discussed in Section 5.1.

The specification of a (simple) constraint has the following form:

[**constraint** <name>] **primary key** | **unique** | **not null**

A constraint can be named. It is advisable to name a constraint in order to get more meaningful information when this constraint is violated due to, e.g., an insertion of a tuple that violates the constraint. If no name is specified for the constraint, ORACLE automatically generates a name of the pattern SYS_C<number>.

The two most simple types of constraints have already been discussed: **not null** and **unique**. Probably the most important type of integrity constraints in a database are primary key constraints. A primary key constraint enables a unique identification of each tuple in a table. Based on a primary key, the database system ensures that no duplicates appear in a table. For example, for our EMP table, the specification

create table EMP (
EMPNO **number**(4) **constraint** pk_emp **primary key**,
...);

defines the attribute EMPNO as the primary key for the table. Each value for the attribute EMPNO thus must appear only once in the table EMP. A table, of course, may only have one primary key. Note that in contrast to a **unique** constraint, null values are not allowed.

Example:

We want to create a table called PROJECT to store information about projects. For each project, we want to store the number and the name of the project, the employee number of the project's manager, the budget and the number of persons working on the project, and the start date and end date of the project. Furthermore, we have the following conditions:

- a project is identified by its project number,
- the name of a project must be unique,
- the manager and the budget must be defined.

Table definition:

create table PROJECT (
PNO **number**(3) **constraint** prj_pk **primary key**,
PNAME **varchar2**(60) **unique**,
PMGR **number**(4) **not null**,
PERSONS **number**(5),
BUDGET **number**(8,2) **not null**,
PSTART **date**,
PEND **date**);

A **unique** constraint can include more than one attribute. In this case the pattern **unique**(<column i>, ..., <column j>) is used. If it is required, for example, that no two projects have the same start and end date, we have to add the table constraint

constraint no_same_dates **unique**(PEND, PSTART)

This constraint has to be defined in the **create table** command after both columns PEND and PSTART have been defined. A primary key constraint that includes more than only one column can be specified in an analogous way.

Instead of a **not null** constraint it is sometimes useful to specify a default value for an attribute if no value is given, e.g., when a tuple is inserted. For this, we use the **default** clause.

Example:

If no start date is given when inserting a tuple into the table PROJECT, the project start date should be set to January 1st, 1995:

PSTART **date default**('01-JAN-95')

Note: Unlike integrity constraints, it is not possible to specify a name for a default.

1.3.3 Checklist for Creating Tables

The following provides a small checklist for the issues that need to be considered before creating a table.

- What are the attributes of the tuples to be stored? What are the data types of the attributes? Should **varchar2** be used instead of **char** ?
- Which columns build the primary key?
- Which columns do (not) allow null values? Which columns do (not) allow duplicates ?
- Are there default values for certain columns that allow null values ?

1.4 Data Modifications in SQL

After a table has been created using the **create table** command, tuples can be inserted into the table, or tuples can be deleted or modified.

1.4.1 Insertions

The most simple way to insert a tuple into a table is to use the **insert** statement

insert into <table> [(<column i, . . . , column j>)]
values (<value i, . . . , value j>);

For each of the listed columns, a corresponding (matching) value must be specified. Thus an insertion does not necessarily have to follow the order of the attributes as specified in the **create table** statement. If a column is omitted, the value **null** is inserted instead. If no column list is given, however, for each column as defined in the **create table** statement a value must be given.

Examples:

insert into `PROJECT(PNO, PNAME, PERSONS, BUDGET, PSTART)`
values(313, 'DBS', 4, 150000.42, '10-OCT-94');

or

insert into `PROJECT`
values(313, 'DBS', 7411, **null**, 150000.42, '10-OCT-94', **null**);

If there are already some data in other tables, these data can be used for insertions into a new table. For this, we write a query whose result is a set of tuples to be inserted. Such an **insert** statement has the form

insert into <table> [(<column i, . . . , column j>)] <query>

Example: Suppose we have defined the following table:

create table `OLDEMP` (
`ENO` **number**(4) **not null**,
`HDATE` **date**);

We now can use the table `EMP` to insert tuples into this new relation:

insert into `OLDEMP (ENO, HDATE)`
select `EMPNO, HIREDATE` **from** `EMP`
where `HIREDATE` < '31-DEC-60';

1.4.2 Updates

For modifying attribute values of (some) tuples in a table, we use the **update** statement:

update <table> **set**
<column i> = <expression i>, . . . , <column j> = <expression j>
[**where** <condition>];

An expression consists of either a constant (new value), an arithmetic or string operation, or an SQL query. Note that the new value to assign to <column i> must a the matching data type.

An **update** statement without a **where** clause results in changing respective attributes of all tuples in the specified table. Typically, however, only a (small) portion of the table requires an update.

Examples:

- The employee JONES is transfered to the department 20 as a manager and his salary is increased by 1000:

  update `EMP` **set**
  `JOB` = 'MANAGER', `DEPTNO` = 20, `SAL` = `SAL` +1000
  where `ENAME` = 'JONES';
- All employees working in the departments 10 and 30 get a 15% salary increase.

  update `EMP` **set**
  `SAL` = `SAL` ∗ 1.15 **where** `DEPTNO` **in** (10,30);

Analogous to the **insert** statement, other tables can be used to retrieve data that are used as new values. In such a case we have a <query> instead of an <expression>.

Example: All salesmen working in the department 20 get the same salary as the manager who has the lowest salary among all managers.

update `EMP` **set**
`SAL` = (**select min**(`SAL`) **from** `EMP`
where `JOB` = 'MANAGER')
where `JOB` = 'SALESMAN' **and** `DEPTNO` = 20;

Explanation: The query retrieves the minimum salary of all managers. This value then is assigned to all salesmen working in department 20.

It is also possible to specify a query that retrieves more than only one value (but still only one tuple!). In this case the **set** clause has the form **set**(<column i, ..., column j>) = <query>. It is important that the order of data types and values of the selected row exactly correspond to the list of columns in the **set** clause.

1.4.3 Deletions

All or selected tuples can be deleted from a table using the **delete** command:

delete from <table> [**where** <condition>];

If the **where** clause is omitted, all tuples are deleted from the table. An alternative command for deleting all tuples from a table is the **truncate table** <table> command. However, in this case, the deletions cannot be undone (see subsequent Section 1.4.4).

Example:

Delete all projects (tuples) that have been finished before the actual date (system date):

delete from `PROJECT` **where** `PEND` < **sysdate**;

sysdate is a function in SQL that returns the system date. Another important SQL function is **user**, which returns the name of the user logged into the current ORACLE session.

1.4.4 Commit and Rollback

A sequence of database modifications, i.e., a sequence of **insert**, **update**, and **delete** statements, is called a *transaction*. Modifications of tuples are temporarily stored in the database system. They become permanent only after the statement **commit;** has been issued.

As long as the user has not issued the **commit** statement, it is possible to undo all modifications since the last **commit**. To undo modifications, one has to issue the statement **rollback;**.

It is advisable to complete each modification of the database with a **commit** (as long as the modification has the expected effect). Note that any data definition command such as **create table** results in an internal **commit**. A **commit** is also implicitly executed when the user terminates an ORACLE session.

1.5 Queries (Part II)

In Section 1.2 we have only focused on queries that refer to exactly one table. Furthermore, conditions in a **where** were restricted to simple comparisons. A major feature of relational databases, however, is to combine (join) tuples stored in different tables in order to display more meaningful and complete information. In SQL the **select** statement is used for this kind of queries joining relations:

select [**distinct**] [<alias a_k>.]<column i>, ..., [<alias a_l>.]<column j>
from <table 1> [<alias a_1>], ..., <table n> [<alias a_n>]
[**where** <condition>]

The specification of table aliases in the **from** clause is necessary to refer to columns that have the same name in different tables. For example, the column `DEPTNO` occurs in both `EMP` and `DEPT`. If we want to refer to either of these columns in the **where** or **select** clause, a table alias has to be specified and put in the front of the column name. Instead of a table alias also the complete relation name can be put in front of the column such as `DEPT.DEPTNO`, but this sometimes can lead to rather lengthy query formulations.

1.5.1 Joining Relations

Comparisons in the **where** clause are used to combine rows from the tables listed in the **from** clause.

Example: In the table `EMP` only the numbers of the departments are stored, not their name. For each salesman, we now want to retrieve the name as well as the number and the name of the department where he is working:

```
select ENAME, E.DEPTNO, DNAME
    from EMP E, DEPT D
    where E.DEPTNO = D.DEPTNO
    and JOB = 'SALESMAN';
```

Explanation: `E` and `D` are table aliases for `EMP` and `DEPT`, respectively. The computation of the query result occurs in the following manner (without optimization):

1. Each row from the table `EMP` is combined with each row from the table `DEPT` (this operation is called *Cartesian product*). If `EMP` contains m rows and `DEPT` contains n rows, we thus get $n * m$ rows.
2. From these rows those that have the same department number are selected (**where** `E.DEPTNO = D.DEPTNO`).
3. From this result finally all rows are selected for which the condition `JOB = 'SALESMAN'` holds.

In this example the joining condition for the two tables is based on the equality operator "=". The columns compared by this operator are called *join columns* and the join operation is called an *equijoin*.

Any number of tables can be combined in a **select** statement.

Example: For each project, retrieve its name, the name of its manager, and the name of the department where the manager is working:

```
select ENAME, DNAME, PNAME
from EMP E, DEPT D, PROJECT P
where E.EMPNO = P.MGR
  and D.DEPTNO = E.DEPTNO;
```

It is even possible to join a table with itself:

Example: List the names of all employees together with the name of their manager:

```
select E1.ENAME, E2.ENAME
from EMP E1, EMP E2
where E1.MGR = E2.EMPNO;
```

Explanation: The join columns are `MGR` for the table `E1` and `EMPNO` for the table `E2`. The equijoin comparison is `E1.MGR = E2.EMPNO`.

1.5.2 Subqueries

Up to now we have only concentrated on simple comparison conditions in a **where** clause, i.e., we have compared a column with a constant or we have compared two columns. As we have already seen for the **insert** statement, queries can be used for assignments to columns. A query result can also be used in a condition of a **where** clause. In such a case the query is called a *subquery* and the complete **select** statement is called a *nested query*.

A respective condition in the **where** clause then can have one of the following forms:

1. *Set-valued subqueries*
   <expression> [**not**] **in** (<subquery>)
   <expression> <comparison operator> [**any**|**all**] (<subquery>)
   An <expression> can either be a column or a computed value.
2. *Test for (non)existence*
   [**not**] **exists** (<subquery>)

In a **where** clause conditions using subqueries can be combined arbitrarily by using the logical connectives **and** and **or**.

Example: List the name and salary of employees of the department 20 who are leading a project that started before December 31, 1990:

```
select ENAME, SAL from EMP
where EMPNO in
        (select PMGR from PROJECT
         where PSTART < '31-DEC-90')
and DEPTNO =20;
```

Explanation: The subquery retrieves the set of those employees who manage a project that started before December 31, 1990. If the employee working in department 20 is contained in this set (**in** operator), this tuple belongs to the query result set.

Example: List all employees who are working in a department located in BOSTON:


```
select * from EMP
where DEPTNO in
        (select DEPTNO from DEPT
         where LOC = 'BOSTON');
```

The subquery retrieves only one value (the number of the department located in Boston). Thus it is possible to use "=" instead of **in**. As long as the result of a subquery is not known in advance, i.e., whether it is a single value or a set, it is advisable to use the **in** operator.

A subquery may use again a subquery in its **where** clause. Thus conditions can be nested arbitrarily. An important class of subqueries are those that refer to its surrounding (sub)query and the tables listed in the **from** clause, respectively. Such type of queries is called *correlated subqueries*.

Example: List all those employees who are working in the same department as their manager (note that components in [ ] are optional:

```
select * from EMP E1
where DEPTNO in
        (select DEPTNO from EMP [E]
         where [E.]EMPNO = E1.MGR);
```

Explanation: The subquery in this example is related to its surrounding query since it refers to the column `E1.MGR`. A tuple is selected from the table `EMP` (`E1`) for the query result if the value for the column `DEPTNO` occurs in the set of values select in the subquery. One can think of the evaluation of this query as follows: For each tuple in the table `E1`, the subquery is evaluated individually. If the condition **where** `DEPTNO` **in** ... evaluates to true, this tuple is selected. Note that an alias for the table `EMP` in the subquery is not necessary since columns without a preceding alias listed there always refer to the innermost query and tables.

Conditions of the form <expression> <comparison operator> [**any**|**all**] <subquery> are used to compare a given <expression> with each value selected by <subquery>.

- For the clause **any**, the condition evaluates to true if there exists at least on row selected by the subquery for which the comparison holds. If the subquery yields an empty result set, the condition is not satisfied.
- For the clause **all**, in contrast, the condition evaluates to true if for all rows selected by the subquery the comparison holds. In this case the condition evaluates to true if the subquery does not yield any row or value.

Example: Retrieve all employees who are working in department 10 and who earn at least as much as any (i.e., at least one) employee working in department 30:

```
select * from EMP
where SAL >= any
        (select SAL from EMP
         where DEPTNO = 30)
and DEPTNO = 10;
```

Note: Also in this subquery no aliases are necessary since the columns refer to the innermost **from** clause.

Example: List all employees who are not working in department 30 and who earn more than all employees working in department 30:

```
select * from EMP
where SAL > all
      (select SAL from EMP
       where DEPTNO = 30)
and DEPTNO <> 30;
```

For **all** and **any**, the following equivalences hold:

in $\Leftrightarrow$ = **any**
not in $\Leftrightarrow$ <> **all** or != **all**

Often a query result depends on whether certain rows do (not) exist in (other) tables. Such type of queries is formulated using the **exists** operator.

Example: List all departments that have no employees:

```
select * from DEPT
where not exists
      (select * from EMP
       where DEPTNO = DEPT.DEPTNO);
```

Explanation: For each tuple from the table DEPT, the condition is checked whether there exists a tuple in the table EMP that has the same department number (DEPT.DEPTNO). In case no such tuple exists, the condition is satisfied for the tuple under consideration and it is selected. If there exists a corresponding tuple in the table EMP, the tuple is not selected.

1.5.3 Operations on Result Sets

Sometimes it is useful to combine query results from two or more queries into a single result. SQL supports three set operators which have the pattern:

<query 1> <set operator> <query 2>

The set operators are:

- **union** [**all**] returns a table consisting of all rows either appearing in the result of <query 1> or in the result of <query 2>. Duplicates are automatically eliminated unless the clause **all** is used.
- **intersect** returns all rows that appear in both results <query 1> and <query 2>.
- **minus** returns those rows that appear in the result of <query 1> but not in the result of <query 2>.

Example: Assume that we have a table EMP2 that has the same structure and columns as the table EMP:

- All employee numbers and names from both tables:
  ```
  select EMPNO, ENAME from EMP
    union
  select EMPNO, ENAME from EMP2;
  ```
- Employees who are listed in both EMP and EMP2:
  ```
  select * from EMP
    intersect
  select * from EMP2;
  ```
- Employees who are only listed in EMP:
  ```
  select * from EMP
    minus
  select * from EMP2;
  ```

Each operator requires that both tables have the same data types for the columns to which the operator is applied.

1.5.4 Grouping

In Section 1.2.4 we have seen how aggregate functions can be used to compute a single value for a column. Often applications require grouping rows that have certain properties and then applying an aggregate function on one column for each group separately. For this, SQL provides the clause **group by** <group_column(s)>. This clause appears after the **where** clause and must refer to columns of tables listed in the **from** clause.

select <column(s)>
from <table(s)>
where <condition>
group by <group_column(s)>
[**having** <group_condition(s)>];

Those rows retrieved by the **selected** clause that have the same value(s) for <group_column(s)> are grouped. Aggregations specified in the **select** clause are then applied to each group separately. It is important that only those columns that appear in the <group_column(s)> clause can be listed without an aggregate function in the **select** clause !

Example: For each department, we want to retrieve the minimum and maximum salary.

```
select DEPTNO, min(SAL), max(SAL)
from EMP
group by DEPTNO;
```

Rows from the table EMP are grouped such that all rows in a group have the same department number. The aggregate functions are then applied to each such group. We thus get the following query result:

DEPTNO	MIN(SAL)	MAX(SAL)
10	1300	5000
20	800	3000
30	950	2850

Rows to form a group can be restricted in the **where** clause. For example, if we add the condition **where** JOB = 'CLERK', only respective rows build a group. The query then would retrieve the minimum and maximum salary of all clerks for each department. Note that is not allowed to specify any other column than DEPTNO without an aggregate function in the **select** clause since this is the only column listed in the **group by** clause (is it also easy to see that other columns would not make any sense).

Once groups have been formed, certain groups can be eliminated based on their properties, e.g., if a group contains less than three rows. This type of condition is specified using the **having** clause. As for the **select** clause also in a **having** clause only <group_column(s)> and aggregations can be used.

Example: Retrieve the minimum and maximum salary of clerks for each department having more than three clerks.

select DEPTNO, **min**(SAL), **max**(SAL)
from EMP
where JOB = 'CLERK'
group by DEPTNO
having count(∗) > 3;

Note that it is even possible to specify a subquery in a **having** clause. In the above query, for example, instead of the constant 3, a subquery can be specified.

A query containing a **group by** clause is processed in the following way:

1. Select all rows that satisfy the condition specified in the **where** clause.
2. From these rows form groups according to the **group by** clause.
3. Discard all groups that do not satisfy the condition in the **having** clause.
4. Apply aggregate functions to each group.
5. Retrieve values for the columns and aggregations listed in the **select** clause.

1.5.5 Some Comments on Tables

Accessing tables of other users

Provided that a user has the privilege to access tables of other users (see also Section 3), she/he can refer to these tables in her/his queries. Let <user> be a user in the Oracle system and <table> a table of this user. This table can be accessed by other (privileged) users using the command

select ∗ **from** <user>.<table>;

In case that one often refers to tables of other users, it is useful to use a *synonym* instead of <user>.<table>. In Oracle-SQL a synonym can be created using the command

create synonym <name> **for** <user>.<table> ;

It is then possible to use simply <name> in a **from** clause. Synonyms can also be created for one's own tables.

Adding Comments to Definitions

For applications that include numerous tables, it is useful to add comments on table definitions or to add comments on columns. A comment on a table can be created using the command

comment on table <table> **is** '<text>';

A comment on a column can be created using the command

comment on column <table>.<column> **is** '<text>';

Comments on tables and columns are stored in the data dictionary. They can be accessed using the data dictionary views USER_TAB_COMMENTS and USER_COL_COMMENTS (see also Section 3).

Modifying Table- and Column Definitions

It is possible to modify the structure of a table (the relation schema) even if rows have already been inserted into this table. A column can be added using the **alter table** command

alter table <table>
add(<column> <data type> [**default** <value>] [<column constraint>]);

If more than only one column should be added at one time, respective **add** clauses need to be separated by colons. A table constraint can be added to a table using

alter table <table> **add** (<table constraint>);

Note that a column constraint is a table constraint, too. **not null** and **primary key** constraints can only be added to a table if none of the specified columns contains a null value. Table definitions can be modified in an analogous way. This is useful, e.g., when the size of strings that can be stored needs to be increased. The syntax of the command for modifying a column is

alter table <table>
modify(<column> [<data type>] [**default** <value>] [<column constraint>]);

Note: In earlier versions of Oracle it is not possible to delete single columns from a table definition. A workaround is to create a temporary table and to copy respective columns and rows into this new table. Furthermore, it is not possible to rename tables or columns. In the most recent version (9i), using the **alter table** command, it is possible to rename a table, columns, and constraints. In this version, there also exists a **drop column** clause as part of the **alter table** statement.

Deleting a Table

A table and its rows can be deleted by issuing the command **drop table** <table> [**cascade constraints**];.

1.6 Views

In ORACLE the SQL command to create a view (virtual table) has the form

create [**or replace**] **view** <view-name> [(<column(s)>)] **as**
<**select**-statement> [**with check option** [**constraint** <name>]];

The optional clause **or replace** re-creates the view if it already exists. <column(s)> names the columns of the view. If <column(s)> is not specified in the view definition, the columns of the view get the same names as the attributes listed in the **select** statement (if possible).

Example: The following view contains the name, job title and the annual salary of employees working in the department 20:

```
create view DEPT20 as
    select ENAME, JOB, SAL*12 ANNUAL_SALARY from EMP
    where DEPTNO = 20;
```

In the **select** statement the column alias `ANNUAL_SALARY` is specified for the expression `SAL*12` and this alias is taken by the view. An alternative formulation of the above view definition is

```
create view DEPT20 (ENAME, JOB, ANNUAL_SALARY) as
    select ENAME, JOB, SAL * 12 from EMP
    where DEPTNO = 20;
```

A view can be used in the same way as a table, that is, rows can be retrieved from a view (also respective rows are not physically stored, but derived on basis of the **select** statement in the view definition), or rows can even be modified. A view is evaluated again each time it is accessed. In ORACLE SQL no **insert**, **update**, or **delete** modifications on views are allowed that use one of the following constructs in the view definition:

- Joins
- Aggregate function such as **sum, min, max** etc.
- set-valued subqueries (**in, any, all**) or test for existence (**exists**)
- **group by** clause or **distinct** clause

In combination with the clause **with check option** any update or insertion of a row into the view is rejected if the new/modified row does not meet the view definition, i.e., these rows would not be selected based on the **select** statement. A **with check option** can be named using the **constraint** clause.

A view can be deleted using the command **delete** <view-name>.