Oracle/SQL Tutorial[1]

Michael Gertz
Database and Information Systems Group
Department of Computer Science
University of California, Davis
gertz@cs.ucdavis.edu
http://www.db.cs.ucdavis.edu

This Oracle/SQL tutorial provides a detailed introduction to the SQL query language and the Oracle Relational Database Management System. Further information about Oracle and SQL can be found on the web site www.db.cs.ucdavis.edu/dbs.

Comments, corrections, or additions to these notes are welcome. Many thanks to Christina Chung for comments on the previous version.

Recommended Literature

George Koch and Kevin Loney: ORACLE*8 The Complete Reference* (The Single Most Comprehensive Sourcebook for Oracle Server, Includes CD with electronic version of the book), 1299 pages, McGraw-Hill/Osborne, 1997.

Michael Abbey and Michael Corey: ORACLE*8 : A Beginner's Guide [A Thorough Introduction for First-time Users]*, 767 pages, McGraw-Hill/Osborne, 1997.

Steven Feuerstein, Bill Pribyl, Debby Russell: ORACLE *PL/SQL Programming* (2nd Edition), O'Reilly & Associates, 1028 pages, 1997.

C.J. Date and Hugh Darwen: *A Guide to the SQL Standard* (4th Edition), Addison-Wesley, 1997.

Jim Melton and Alan R. Simon: *Understanding the New SQL: A Complete Guide* (2nd Edition, Dec 2000), The Morgan Kaufmann Series in Data Management Systems, 2000.

[1]revised Version 1.01, January 2000, Michael Gertz, Copyright 2000.

Contents

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>.

2 SQL*Plus

Introduction

SQL*Plus is the interactive (low-level) user interface to the ORACLE database management system. Typically, SQL*Plus is used to issue *ad-hoc* queries and to view the query result on the screen. Some of the features of SQL*Plus are:

- A built-in command line editor can be used to edit (incorrect) SQL queries. Instead of this line editor any editor installed on the computer can be invoked.
- There are numerous commands to format the output of a query.
- SQL*Plus provides an online-help.
- Query results can be stored in files which then can be printed.

Queries that are frequently issued can be saved to a file and invoked later. Queries can be parameterized such that it is possible to invoke a saved query with a parameter.

A Minimal User Guide

Before you start SQL*Plus make sure that the following UNIX shell variables are properly set (shell variables can be checked using the **env** command, e.g., **env** | **grep** ORACLE):

- `ORACLE_HOME`, e.g., `ORACLE_HOME=/usr/pkg/oracle/734`
- `ORACLE_SID`, e.g, `ORACLE_SID=prod`

In order to invoke SQL*Plus from a UNIX shell, the command **sqlplus** has to be issued. SQL*Plus then displays some information about the product, and prompts you for your user name and password for the ORACLE system.

```
gertz(catbert)54: sqlplus

SQL*Plus: Release 3.3.4.0.1 - Production on Sun Dec 20 19:16:52 1998

Copyright (c) Oracle Corporation 1979, 1996.  All rights reserved.

Enter user-name: scott
Enter password:

Connected to:
Oracle7 Server Release 7.3.4.0.1 - Production Release
With the distributed option
PL/SQL Release 2.3.4.0.0 - Production

SQL>
```

`SQL>` is the prompt you get when you are connected to the ORACLE database system. In SQL*Plus you can divide a statement into separate lines, each continuing line is indicated by a prompt such 2>, 3> etc. An SQL statement must always be terminated by a semicolon (;). In addition to the SQL statements discussed in the previous section, SQL*Plus provides some special SQL*Plus commands. These commands need not be terminated by a semicolon. Upper and lower case letters are only important for string comparisons. An SQL query can always be interrupted by using <Control>C. To exit SQL*Plus you can either type **exit** or **quit**.

Editor Commands

The most recently issued SQL statement is stored in the *SQL buffer*, independent of whether the statement has a correct syntax or not. You can edit the buffer using the following commands:

- `l[ist]` lists all lines in the SQL buffer and sets the current line (marked with an "$*$") to the last line in the buffer.
- `l`<number> sets the actual line to <number>
- `c[hange]`/<old_string>/<new_string> replaces the first occurrence of <old_string> by <new_string> (for the actual line)
- `a[ppend]`<string> appends <string> to the current line
- `del` deletes the current line
- `r[un]` executes the current buffer contents
- `get`<file> reads the data from the file <file> into the buffer
- `save`<file> writes the current buffer into the file <file>
- `edit` invokes an editor and loads the current buffer into the editor. After exiting the editor the modified SQL statement is stored in the buffer and can be executed (command `r`).

The editor can be defined in the SQL*Plus shell by typing the command **define _editor** = <name>, where <name> can be any editor such as *emacs, vi, joe*, or *jove*.

SQL*Plus Help System and Other Useful Commands

- To get the online help in SQL*Plus just type **help** <command>, or just **help** to get information about how to use the **help** command. In ORACLE Version 7 one can get the complete list of possible commands by typing **help command**.
- To change the password, in ORACLE Version 7 the command
  alter user <user> **identified by** <new_password>;
  is used. In ORACLE Version 8 the command **passw** <user> prompts the user for the old/new password.
- The command **desc[ribe]** <table> lists all columns of the given table together with their data types and information about whether null values are allowed or not.
- You can invoke a UNIX command from the SQL*Plus shell by using **host** <UNIX_command>. For example, **host ls -la *.sql** lists all SQL files in the current directory.

- You can log your SQL*Plus session and thus queries and query results by using the command **spool** <file>. All information displayed on screen is then stored in <file> which automatically gets the extension `.lst`. The command **spool off** turns spooling off.
- The command **copy** can be used to copy a complete table. For example, the command
  copy from `scott/tiger` **create** `EMPL` **using select** $*$ **from** `EMP`;
  copies the table `EMP` of the user `scott` with password `tiger` into the relation `EMPL`. The relation `EMP` is automatically created and its structure is derived based on the attributes listed in the **select** clause.
- SQL commands saved in a file <name>.sql can be loaded into SQL*Plus and executed using the command @<name>.
- Comments are introduced by the clause **rem**[**ark**] (only allowed between SQL statements), or - - (allowed within SQL statements).

Formatting the Output

SQL*Plus provides numerous commands to format query results and to build simple reports. For this, format variables are set and these settings are only valid during the SQL*Plus session. They get lost after terminating SQL*Plus. It is, however, possible to save settings in a file named `login.sql` in your home directory. Each time you invoke SQL*Plus this file is automatically loaded.

The command **column** <column name> <option 1> <option 2> ... is used to format columns of your query result. The most frequently used options are:

- **format** A<n> For alphanumeric data, this option sets the length of <column name> to <n>. For columns having the data type **number**, the **format** command can be used to specify the format before and after the decimal point. For example, **format** 99,999.99 specifies that if a value has more than three digits in front of the decimal point, digits are separated by a colon, and only two digits are displayed after the decimal point.
- The option **heading** <text> relabels <column name> and gives it a new heading.
- **null** <text> is used to specify the output of null values (typically, null values are not displayed).
- **column** <column name> **clear** deletes the format definitions for <column name>.

The command **set linesize** <number> can be used to set the maximum length of a single line that can be displayed on screen. **set pagesize** <number> sets the total number of lines SQL*Plus displays before printing the column names and headings, respectively, of the selected rows.

Several other formatting features can be enabled by setting SQL*Plus variables. The command **show all** displays all variables and their current values. To set a variable, type **set** <variable> <value>. For example, **set timing on** causes SQL*Plus to display timing statistics for each SQL command that is executed. **set pause on** [<text>] makes SQL*Plus wait for you to press `Return` after the number of lines defined by **set pagesize** has been displayed. <text> is the message SQL*Plus will display at the bottom of the screen as it waits for you to hit `Return`.

3 Oracle Data Dictionary

The Oracle data dictionary is one of the most important components of the Oracle DBMS. It contains all information about the structures and objects of the database such as tables, columns, users, data files etc. The data stored in the data dictionary are also often called *metadata*. Although it is usually the domain of database administrators (DBAs), the data dictionary is a valuable source of information for end users and developers. The data dictionary consists of two levels: the internal level contains all base tables that are used by the various DBMS software components and they are normally not accessible by end users. The external level provides numerous views on these base tables to access information about objects and structures at different levels of detail.

3.1 Data Dictionary Tables

An installation of an Oracle database always includes the creation of three standard Oracle users:

- `SYS`: This is the owner of all data dictionary tables and views. This user has the highest privileges to manage objects and structures of an Oracle database such as creating new users.
- `SYSTEM`: is the owner of tables used by different tools such SQL*Forms, SQL*Reports etc. This user has less privileges than `SYS`.
- `PUBLIC`: This is a "dummy" user in an Oracle database. All privileges assigned to this user are automatically assigned to all users known in the database.

The tables and views provided by the data dictionary contain information about

- users and their privileges,
- tables, table columns and their data types, integrity constraints, indexes,
- statistics about tables and indexes used by the optimizer,
- privileges granted on database objects,
- storage structures of the database.

The SQL command

select $*$ **from** `DICT[IONARY]`;

lists all tables and views of the data dictionary that are accessible to the user. The selected information includes the name and a short description of each table and view. Before issuing this query, check the column definitions of `DICT[IONARY]` using **desc** `DICT[IONARY]` and set the appropriate values for **column** using the **format** command.

The query

select $*$ **from** `TAB`;

retrieves the names of all tables owned by the user who issues this command. The query

select $*$ **from** `COL`;

returns all information about the columns of one's own tables.

Each SQL query requires various internal accesses to the tables and views of the data dictionary. Since the data dictionary itself consists of tables, ORACLE has to generate numerous SQL statements to check whether the SQL command issued by a user is correct and can be executed.

Example: The SQL query

select $*$ **from** EMP
where SAL $>$ 2000;

requires a verification whether (1) the table `EMP` exists, (2) the user has the privilege to access this table, (3) the column `SAL` is defined for this table etc.

3.2 Data Dictionary Views

The external level of the data dictionary provides users a front end to access information relevant to the users. This level provides numerous views (in ORACLE7 approximately 540) that represent (a portion of the) data from the base tables in a readable and understandable manner. These views can be used in SQL queries just like normal tables.

The views provided by the data dictionary are divided into three groups: `USER`, `ALL`, and `DBA`. The group name builds the prefix for each view name. For some views, there are associated synonyms as given in brackets below.

- `USER_`: Tuples in the `USER` views contain information about objects owned by the account performing the SQL query (current user)

USER_TABLES	all tables with their name, number of columns, storage information, statistical information etc. (TABS)
USER_CATALOG	tables, views, and synonyms (CAT)
USER_COL_COMMENTS	comments on columns
USER_CONSTRAINTS	constraint definitions for tables
USER_INDEXES	all information about indexes created for tables (IND)
USER_OBJECTS	all database objects owned by the user (OBJ)
USER_TAB_COLUMNS	columns of the tables and views owned by the user (COLS)
USER_TAB_COMMENTS	comments on tables and views
USER_TRIGGERS	triggers defined by the user
USER_USERS	information about the current user
USER_VIEWS	views defined by the user

- `ALL_`: Rows in the `ALL` views include rows of the `USER` views and all information about objects that are accessible to the current user. The structure of these views is analogous to the structure of the `USER` views.

ALL_CATALOG	owner, name and type of all accessible tables, views, and synonyms
ALL_TABLES	owner and name of all accessible tables
ALL_OBJECTS	owner, type, and name of accessible database objects
ALL_TRIGGERS	...
ALL_USERS	...
ALL_VIEWS	...

- `DBA_`: The `DBA` views encompass information about all database objects, regardless of the owner. Only users with DBA privileges can access these views.

DBA_TABLES	tables of all users in the database
DBA_CATALOG	tables, views, and synonyms defined in the database
DBA_OBJECTS	object of all users
DBA_DATA_FILES	information about data files
DBA_USERS	information about all users known in the database

4 Application Programming

4.1 PL/SQL

4.1.1 Introduction

The development of database applications typically requires language constructs similar to those that can be found in programming languages such as C, C++, or Pascal. These constructs are necessary in order to implement complex data structures and algorithms. A major restriction of the database language SQL, however, is that many tasks cannot be accomplished by using only the provided language elements.

PL/SQL (Procedural Language/SQL) is a procedural extension of Oracle-SQL that offers language constructs similar to those in imperative programming languages. PL/SQL allows users and designers to develop complex database applications that require the usage of control structures and procedural elements such as procedures, functions, and modules.

The basic construct in PL/SQL is a *block*. Blocks allow designers to combine logically related (SQL-) statements into units. In a block, constants and variables can be declared, and variables can be used to store query results. Statements in a PL/SQL block include SQL statements, control structures (loops), condition statements (if-then-else), exception handling, and calls of other PL/SQL blocks.

PL/SQL blocks that specify procedures and functions can be grouped into *packages*. A package is similar to a module and has an interface and an implementation part. Oracle offers several predefined packages, for example, input/output routines, file handling, job scheduling etc. (see directory `$ORACLE_HOME/rdbms/admin`).

Another important feature of PL/SQL is that it offers a mechanism to process query results in a tuple-oriented way, that is, one tuple at a time. For this, *cursors* are used. A cursor basically is a pointer to a query result and is used to read attribute values of selected tuples into variables. A cursor typically is used in combination with a loop construct such that each tuple read by the cursor can be processed individually.

In summary, the major goals of PL/SQL are to

- increase the expressiveness of SQL,
- process query results in a tuple-oriented way,
- optimize combined SQL statements,
- develop modular database application programs,
- reuse program code, and
- reduce the cost for maintaining and changing applications.

4.1.2 Structure of PL/SQL-Blocks

PL/SQL is a block-structured language. Each block builds a (named) program unit, and blocks can be nested. Blocks that build a procedure, a function, or a package must be named. A PL/SQL block has an optional declare section, a part containing PL/SQL statements, and an optional exception-handling part. Thus the structure of a PL/SQL looks as follows (brackets [ ] enclose optional parts):

[<Block header>]
[**declare**
    <Constants>
    <Variables>
    <Cursors>
    <User defined exceptions>]
begin
    <PL/SQL statements>
    [**exception**
        <Exception handling>]
end;

The block header specifies whether the PL/SQL block is a procedure, a function, or a package. If no header is specified, the block is said to be an *anonymous* PL/SQL block. Each PL/SQL block again builds a PL/SQL statement. Thus blocks can be nested like blocks in conventional programming languages. The scope of declared variables (i.e., the part of the program in which one can refer to the variable) is analogous to the scope of variables in programming languages such as C or Pascal.

4.1.3 Declarations

Constants, variables, cursors, and exceptions used in a PL/SQL block must be declared in the declare section of that block. Variables and constants can be declared as follows:

<variable name> [**constant**] <data type> [**not null**] [:= <expression>];

Valid data types are SQL data types (see Section 1.1) and the data type **boolean**. Boolean data may only be *true*, *false*, or *null*. The **not null** clause requires that the declared variable must always have a value different from *null*. <expression> is used to initialize a variable. If no expression is specified, the value *null* is assigned to the variable. The clause **constant** states that once a value has been assigned to the variable, the value cannot be changed (thus the variable becomes a constant). Example:

```
declare
    hire_date       date;                       /* implicit initialization with null */
    job_title       varchar2(80) := 'Salesman';
    emp_found       boolean;                    /* implicit initialization with null */
    salary_incr     constant number(3,2) := 1.5;                  /* constant */
    ...
begin ... end;
```

Instead of specifying a data type, one can also refer to the data type of a table column (so-called anchored declaration). For example, `EMP.Empno%TYPE` refers to the data type of the column `Empno` in the relation `EMP`. Instead of a single variable, a record can be declared that can store a complete tuple from a given table (or query result). For example, the data type `DEPT%ROWTYPE` specifies a record suitable to store all attribute values of a complete row from the table `DEPT`. Such records are typically used in combination with a cursor. A field in a record can be accessed using <record name>.<column name>, for example, `DEPT.Deptno`.

A cursor declaration specifies a set of tuples (as a query result) such that the tuples can be processed in a tuple-oriented way (i.e., one tuple at a time) using the **fetch** statement. A cursor declaration has the form

cursor <cursor name> [(<list of parameters>)] **is** <**select** statement>;

The cursor name is an undeclared identifier, not the name of any PL/SQL variable. A parameter has the form <parameter name> <parameter type>. Possible parameter types are **char**, **varchar2**, **number**, **date** and **boolean** as well as corresponding subtypes such as **integer**. Parameters are used to assign values to the variables that are given in the **select** statement.

Example: We want to retrieve the following attribute values from the table `EMP` in a tuple-oriented way: the job title and name of those employees who have been hired after a given date, and who have a manager working in a given department.

```
cursor employee_cur (start_date date, dno number) is
    select JOB, ENAME from EMP E where HIREDATE > start_date
    and exists (select * from EMP
                where E.MGR = EMPNO and DEPTNO = dno);
```

If (some) tuples selected by the cursor will be modified in the PL/SQL block, the clause **for update**[(<column(s)>)] has to be added at the end of the cursor declaration. In this case selected tuples are locked and cannot be accessed by other users until a **commit** has been issued. Before a declared cursor can be used in PL/SQL statements, the cursor must be opened, and after processing the selected tuples the cursor must be closed. We discuss the usage of cursors in more detail below.

Exceptions are used to process errors and warnings that occur during the execution of PL/SQL statements in a controlled manner. Some exceptions are internally defined, such as ZERO_DIVIDE. Other exceptions can be specified by the user at the end of a PL/SQL block. User defined exceptions need to be declared using <name of exception> **exception**. We will discuss exception handling in more detail in Section 4.1.5

4.1.4 Language Elements

In addition to the declaration of variables, constants, and cursors, PL/SQL offers various language constructs such as variable assignments, control structures (loops, if-then-else), procedure and function calls, etc. However, PL/SQL does not allow commands of the SQL data definition language such as the **create table** statement. For this, PL/SQL provides special packages.

Furthermore, PL/SQL uses a modified **select** statement that requires each selected tuple to be assigned to a record (or a list of variables).

There are several alternatives in PL/SQL to a assign a value to a variable. The most simple way to assign a value to a variable is

declare
    counter **integer** := 0;
    ...
begin
    counter := counter + 1;

Values to assign to a variable can also be retrieved from the database using a **select** statement

select <column(s)> **into** <matching list of variables>
from <table(s)> **where** <condition>;

It is important to ensure that the **select** statement retrieves at most one tuple ! Otherwise it is not possible to assign the attribute values to the specified list of variables and a run-time error occurs. If the **select** statement retrieves more than one tuple, a cursor must be used instead. Furthermore, the data types of the specified variables must match those of the retrieved attribute values. For most data types, PL/SQL performs an automatic type conversion (e.g., from **integer** to **real**).

Instead of a list of single variables, a record can be given after the keyword **into**. Also in this case, the **select** statement must retrieve at most one tuple !

declare
    `employee_rec EMP%ROWTYPE;`
    `max_sal EMP.SAL%TYPE;`
begin
    select `EMPNO, ENAME, JOB, MGR, SAL, COMM, HIREDATE, DEPTNO`
    into `employee_rec`
    from `EMP` **where** `EMPNO` = 5698;
    select max(`SAL`) **into** `max_sal` **from** `EMP`;
    ...
end;

PL/SQL provides **while**-loops, two types of **for**-loops, and continuous loops. Latter ones are used in combination with cursors. All types of loops are used to execute a sequence of statements multiple times. The specification of loops occurs in the same way as known from imperative programming languages such as C or Pascal.

A **while**-loop has the pattern

[<< <label name> >>]
while <condition> **loop**
    <sequence of statements>;
end loop [<label name>] ;

A loop can be named. Naming a loop is useful whenever loops are nested and inner loops are completed unconditionally using the **exit** <label name>; statement.

Whereas the number of iterations through a **while** loop is unknown until the loop completes, the number of iterations through the **for** loop can be specified using two integers.

[<< <label name> >>]
for <index> **in** [**reverse**] <lower bound>..<upper bound> **loop**
    <sequence of statements>
end loop [<label name>] ;

The loop counter <index> is declared implicitly. The scope of the loop counter is only the **for** loop. It overrides the scope of any variable having the same name outside the loop. Inside the **for** loop, <index> can be referenced like a constant. <index> may appear in expressions, but one cannot assign a value to <index>. Using the keyword **reverse** causes the iteration to proceed downwards from the higher bound to the lower bound.

Processing Cursors: Before a cursor can be used, it must be opened using the **open** statement

open <cursor name> [(<list of parameters>)] ;

The associated **select** statement then is processed and the cursor references the first selected tuple. Selected tuples then can be processed one tuple at a time using the **fetch** command

fetch <cursor name> **into** <list of variables>;

The **fetch** command assigns the selected attribute values of the current tuple to the list of variables. After the **fetch** command, the cursor advances to the next tuple in the result set. Note that the variables in the list must have the same data types as the selected values. After all tuples have been processed, the **close** command is used to disable the cursor.

close <cursor name>;

The example below illustrates how a cursor is used together with a continuous loop:

```
declare
    cursor emp_cur is select * from EMP;
    emp_rec EMP%ROWTYPE;
    emp_sal EMP.SAL%TYPE;
begin
    open emp_cur;
    loop
        fetch emp_cur into emp_rec;
        exit when emp_cur%NOTFOUND;
        emp_sal := emp_rec.sal;
        <sequence of statements>
    end loop;
    close emp_cur;
    ...
end;
```

Each loop can be completed unconditionally using the **exit** clause:

exit [<block label>] [**when** <condition>]

Using **exit** without a block label causes the completion of the loop that contains the **exit** statement. A condition can be a simple comparison of values. In most cases, however, the condition refers to a cursor. In the example above, `%NOTFOUND` is a predicate that evaluates to *false* if the most recent `fetch` command has read a tuple. The value of <cursor name>`%NOTFOUND` is *null* before the first tuple is fetched. The predicate evaluates to *true* if the most recent **fetch** failed to return a tuple, and *false* otherwise. `%FOUND` is the logical opposite of `%NOTFOUND`.

Cursor **for** loops can be used to simplify the usage of a cursor:

[<< <label name> >>]
for <record name> **in** <cursor name>[(<list of parameters>)] **loop**
    <sequence of statements>
end loop [<label name>];

A record suitable to store a tuple fetched by the cursor is implicitly declared. Furthermore, this loop implicitly performs a **fetch** at each iteration as well as an **open** before the loop is entered and a **close** after the loop is left. If at an iteration no tuple has been fetched, the loop is automatically terminated without an **exit**.

It is even possible to specify a query instead of <cursor name> in a **for** loop:

for <record name> **in** (<**select** statement>) **loop**
    <sequence of statements>
end loop;

That is, a cursor needs not be specified before the loop is entered, but is defined in the **select** statement.

Example:

for `sal_rec` **in** (**select** `SAL + COMM total` **from** `EMP`) **loop**
    ...;
end loop;

`total` is an alias for the expression computed in the **select** statement. Thus, at each iteration only one tuple is fetched. The record `sal_rec`, which is implicitly defined, then contains only one entry which can be accessed using `sal_rec.total`. Aliases, of course, are not necessary if only attributes are selected, that is, if the **select** statement contains no arithmetic operators or aggregate functions.

For conditional control, PL/SQL offers **if-then-else** constructs of the pattern

if <condition> **then** <sequence of statements>
[**elsif**] <condition> **then** <sequence of statements>
...
[**else**] <sequence of statements> **end if**;

Starting with the first condition, if a condition yields *true*, its corresponding sequence of statements is executed, otherwise control is passed to the next condition. Thus the behavior of this type of PL/SQL statement is analogous to if-then-else statements in imperative programming languages.

Except data definition language commands such as **create table**, all types of SQL statements can be used in PL/SQL blocks, in particular **delete**, **insert**, **update**, and **commit**. Note that in PL/SQL only **select** statements of the type **select** <column(s)> **into** are allowed, i.e., selected attribute values can only be assigned to variables (unless the **select** statement is used in a subquery). The usage of **select** statements as in SQL leads to a syntax error. If **update** or **delete** statements are used in combination with a cursor, these commands can be restricted to currently fetched tuple. In these cases the clause **where current of**<cursor name> is added as shown in the following example.

Example: The following PL/SQL block performs the following modifications: All employees having 'KING' as their manager get a 5% salary increase.

```
declare
    manager EMP.MGR%TYPE;
    cursor emp_cur (mgr_no number) is
        select SAL from EMP
        where MGR = mgr_no
    for update of SAL;
begin
    select EMPNO into manager from EMP
    where ENAME = 'KING';
    for emp_rec in emp_cur(manager) loop
        update EMP set SAL = emp_rec.sal * 1.05
        where current of emp_cur;
    end loop;
    commit;
end;
```

Remark: Note that the record `emp_rec` is implicitly defined. We will discuss another version of this block using parameters in Section 4.1.6.

4.1.5 Exception Handling

A PL/SQL block may contain statements that specify exception handling routines. Each error or warning during the execution of a PL/SQL block raises an exception. One can distinguish between two types of exceptions:

- system defined exceptions
- user defined exceptions (which must be declared by the user in the declaration part of a block where the exception is used/implemented)

System defined exceptions are always automatically raised whenever corresponding errors or warnings occur. User defined exceptions, in contrast, must be raised explicitly in a sequence of statements using **raise** <exception name>. After the keyword **exception** at the end of a block, user defined exception handling routines are implemented. An implementation has the pattern

when <exception name> **then** <sequence of statements>;

The most common errors that can occur during the execution of PL/SQL programs are handled by system defined exceptions. The table below lists some of these exceptions with their names and a short description.

Exception name	Number	Remark
CURSOR_ALREADY_OPEN	ORA-06511	You have tried to open a cursor which is already open
INVALID_CURSOR	ORA-01001	Invalid cursor operation such as fetching from a closed cursor
NO_DATA_FOUND	ORA-01403	A **select** ... **into** or **fetch** statement returned no tuple
TOO_MANY_ROWS	ORA-01422	A **select** ... **into** statement returned more than one tuple
ZERO_DIVIDE	ORA-01476	You have tried to divide a number by 0

Example:

```
declare
    emp_sal EMP.SAL%TYPE;
    emp_no EMP.EMPNO%TYPE;
    too_high_sal exception;
begin
    select EMPNO, SAL into emp_no, emp_sal
    from EMP where ENAME = 'KING';
    if emp_sal * 1.05 > 4000 then raise too_high_sal
    else update EMP set SQL ...
    end if;
    exception
        when NO_DATA_FOUND  -- no tuple selected
            then rollback;
        when too_high_sal then insert into high_sal_emps values(emp_no);
        commit;
end;
```

After the keyword **when** a list of exception names connected with **or** can be specified. The last **when** clause in the exception part may contain the exception name **others**. This introduces the default exception handling routine, for example, a **rollback**.

If a PL/SQL program is executed from the SQL*Plus shell, exception handling routines may contain statements that display error or warning messages on the screen. For this, the procedure **raise_application_error** can be used. This procedure has two parameters <error_number> and <message_text>. <error_number> is a negative integer defined by the user and must range between -20000 and -20999. <error_message> is a string with a length up to 2048 characters. The concatenation operator "||" can be used to concatenate single strings to one string. In order to display numeric variables, these variables must be converted to strings using the function **to_char**. If the procedure **raise_application_error** is called from a PL/SQL block, processing the PL/SQL block terminates and all database modifications are undone, that is, an implicit **rollback** is performed in addition to displaying the error message.

Example:

if `emp_sal` $*$ 1.05 > 4000
then raise_application_error(-20010, 'Salary increase for employee with Id '
|| **to_char**(`Emp_no`) || ' is too high');

4.1.6 Procedures and Functions

PL/SQL provides sophisticated language constructs to program procedures and functions as stand-alone PL/SQL blocks. They can be called from other PL/SQL blocks, other procedures and functions. The syntax for a procedure definition is

create [**or replace**] **procedure** <procedure name> [(<list of parameters>)] **is**
<declarations>
begin
<sequence of statements>
[**exception**
<exception handling routines>]
end [<procedure name>];

A function can be specified in an analogous way

create [**or replace**] **function** <function name> [(<list of parameters>)]
return <data type> **is**
...

The optional clause **or replace** re-creates the procedure/function. A procedure can be deleted using the command **drop procedure** <procedure name> (**drop function** <function name>). In contrast to anonymous PL/SQL blocks, the clause **declare** may not be used in procedure/function definitions.

Valid parameters include all data types. However, for **char**, **varchar2**, and **number** no length and scale, respectively, can be specified. For example, the parameter **number**(6) results in a compile error and must be replaced by **number**. Instead of explicit data types, implicit types of the form `%TYPE` and `%ROWTYPE` can be used even if constrained declarations are referenced. A parameter is specified as follows:

<parameter name> [**IN** | **OUT** | **IN OUT**] <data type> [{ := | `DEFAULT`} <expression>]

The optional clauses **IN, OUT**, and **IN OUT** specify the way in which the parameter is used. The default mode for a parameter is **IN**. **IN** means that the parameter can be referenced inside the procedure body, but it cannot be changed. **OUT** means that a value can be assigned to the parameter in the body, but the parameter's value cannot be referenced. **IN OUT** allows both assigning values to the parameter and referencing the parameter. Typically, it is sufficient to use the default mode for parameters.

Example: The subsequent procedure is used to increase the salary of all employees who work in the department given by the procedure's parameter. The percentage of the salary increase is given by a parameter, too.

```
create procedure raise_salary(dno number, percentage number DEFAULT 0.5) is
    cursor emp_cur (dept_no number) is
        select SAL from EMP where DEPTNO = dept_no
        for update of SAL;
    empsal number(8);
begin
    open emp_cur(dno); - - Here dno is assigned to dept_no
    loop
        fetch emp_cur into empsal;
        exit when emp_cur%NOTFOUND;
        update EMP set SAL = empsal * ((100 + percentage)/100)
        where current of emp_cur;
    end loop;
    close emp_cur;
    commit;
end raise_salary;
```

This procedure can be called from the SQL*Plus shell using the command

```
execute raise_salary(10, 3);
```

If the procedure is called only with the parameter 10, the default value 0.5 is assumed as specified in the list of parameters in the procedure definition. If a procedure is called from a PL/SQL block, the keyword **execute** is omitted.

Functions have the same structure as procedures. The only difference is that a function returns a value whose data type (unconstrained) must be specified.

Example:

```
create function get_dept_salary(dno number) return number is
    all_sal number;
begin
    all_sal := 0;
    for emp_sal in (select SAL from EMP where DEPTNO = dno
                    and SAL is not null) loop
```

```
        all_sal := all_sal + emp_sal.sal;
    end loop;
    return all_sal;
end get_dept_salary;
```

In order to call a function from the SQL*Plus shell, it is necessary to first define a variable to which the return value can be assigned. In SQL*Plus a variable can be defined using the command **variable** <variable name> <data type>;, for example, **variable** `salary` **number**. The above function then can be called using the command **execute** `:salary` := `get_dept_salary`(20); Note that the colon ":" must be put in front of the variable.

Further information about procedures and functions can be obtained using the **help** command in the SQL*Plus shell, for example, **help** [**create**] **function**, **help subprograms**, **help stored subprograms**.

4.1.7 Packages

It is essential for a good programming style that logically related blocks, procedures, and functions are combined into modules, and each module provides an interface which allows users and designers to utilize the implemented functionality. PL/SQL supports the concept of modularization by which modules and other constructs can be organized into *packages*. A package consists of a package specification and a package body. The package specification defines the interface that is visible for application programmers, and the package body implements the package specification (similar to header- and source files in the programming language C).

Below a package is given that is used to combine all functions and procedures to manage information about employees.

```
create package manage_employee as -- package specification
   function hire_emp (name varchar2, job varchar2, mgr number, hiredate date,
                      sal number, comm number default 0, deptno number)
            return number;
   procedure fire_emp (emp_id number);
   procedure raise_sal (emp_id number, sal_incr number);
end manage_employee;

create package body manage_employee as
   function hire_emp (name varchar2, job varchar2, mgr number, hiredate date,
                      sal number, comm number default 0, deptno number)
            return number is
   -- Insert a new employee with a new employee Id
   new_empno number(10);
   begin
     select emp_sequence.nextval into new_empno from dual;
```

```
    insert into emp values(new_empno, name, job, mgr, hiredate,
                           sal, comm, deptno);
    return new_empno;
  end hire_emp;

  procedure fire_emp(emp_id number) is
  -- deletes an employee from the table EMP
  begin
    delete from emp where empno = emp_id;
    if SQL%NOTFOUND then -- delete statement referred to invalid emp_id
    raise_application_error(-20011, 'Employee with Id ' ||
                            to_char(emp_id) || ' does not exist.');
    end if;
  end fire_emp;

  procedure raise_sal(emp_id number, sal_incr number) is
  -- modify the salary of a given employee
  begin
    update emp set sal = sal + sal_incr
    where empno = emp_id;
    if SQL%NOTFOUND then
    raise_application_error(-20012, 'Employee with Id ' ||
                            to_char(emp_id) || ' does not exist');
    end if;
  end raise_sal;
end manage_employee;
```

Remark: In order to compile and execute the above package, it is necessary to create first the required sequence (**help sequence**):

create sequence emp_sequence **start with** 8000 **increment by** 10;

A procedure or function implemented in a package can be called from other procedures and functions using the statement <package name>.<procedure name>[(<list of parameters>)]. Calling such a procedure from the SQL*Plus shell requires a leading **execute**.

Oracle offers several predefined packages and procedures that can be used by database users and application developers. A set of very useful procedures is implemented in the package DBMS_OUTPUT. This package allows users to display information to their SQL*Plus session's screen as a PL/SQL program is executed. It is also a very useful means to debug PL/SQL programs that have been successfully compiled, but do not behave as expected. Below some of the most important procedures of this package are listed:

Procedure name	Remark
DBMS_OUTPUT.ENABLE	enables output
DBMS_OUTPUT.DISABLE	disables output
DBMS_OUTPUT.PUT(<string>)	appends (displays) <string> to output buffer
DBMS_OUTPUT.PUT_LINE(<string>)	appends <string> to output buffer and appends a new-line marker
DBMS_OUTPUT.NEW_LINE	displays a new-line marker

Before strings can be displayed on the screen, the output has to be enabled either using the procedure DBMS_OUTPUT.ENABLE or using the SQL*Plus command **set serveroutput on** (before the procedure that produces the output is called).

Further packages provided by ORACLE are UTL_FILE for reading and writing files from PL/SQL programs, DBMS_JOB for job scheduling, and DBMS_SQL to generate SQL statements dynamically, that is, during program execution. The package DBMS_SQL is typically used to create and delete tables from within PL/SQL programs. More packages can be found in the directory $ORACLE_HOME/rdbms/admin.

4.1.8 Programming in PL/SQL

Typically one uses an editor such as emacs or vi to write a PL/SQL program. Once a program has been stored in a file <name> with the extension .sql, it can be loaded into SQL*Plus using the command @<name>. It is important that the last line of the file contains a slash "/".

If the procedure, function, or package has been successfully compiled, SQL*Plus displays the message PL/SQL procedure successfully completed. If the program contains errors, these are displayed in the format ORA-n <message text>, where n is a number and <message text> is a short description of the error, for example, ORA-1001 INVALID CURSOR. The SQL*Plus command **show errors** [<function|procedure|package|package body|trigger> <name>] displays all compilation errors of the most recently created or altered function (or procedure, or package etc.) in more detail. If this command does not show any errors, try **select** $*$ **from** USER_ERRORS.

Under the UNIX shell one can also use the command **oerr** ORA n to get information of the following form:

error description
Cause: *Reason for the error*
Action: *Suggested action*

4.2 Embedded SQL and Pro*C

The query language constructs of SQL described in the previous sections are suited for formulating ad-hoc queries, data manipulation statements and simple PL/SQL blocks in simple, interactive tools such as SQL*Plus. Many data management tasks, however, occur in sophisticated engineering applications and these tasks are too complex to be handled by such an interactive tool. Typically, data are generated and manipulated in computationally complex application programs that are written in a Third-Generation-Language (3GL), and which, therefore, need an interface to the database system. Furthermore, a majority of existing data-intensive engineering applications are written previously using an imperative programming language and now want to make use of the functionality of a database system, thus requiring an easy to use programming interface to the database system. Such an interface is provided in the form of *Embedded SQL*, an embedding of SQL into various programming languages, such as C, C++, Cobol, Fortran etc. Embedded SQL provides application programmers a suitable means to combine the computing power of a programming language with the database manipulation and management capabilities of the declarative query language SQL.

Since all these interfaces exhibit comparable functionalities, in the following we describe the embedding of SQL in the programming language C. For this, we base our discussion on the Oracle interface to C, called Pro*C. The emphasis in this section is placed on the description of the interface, not on introducing the programming language C.

4.2.1 General Concepts

Programs written in Pro*C and which include SQL and/or PL/SQL statements are precompiled into regular C programs using a precompiler that typically comes with the database management software (precompiler package). In order to make SQL and PL/SQL statements in a Proc*C program (having the suffix .pc) recognizable by the precompiler, they are always preceded by the keywords **EXEC SQL** and end with a semicolon ";". The Pro*C precompiler replaces such statements with appropriate calls to functions implemented in the SQL runtime library. The resulting C program then can be compiled and linked using a normal C compiler like any other C program. The linker includes the appropriate Oracle specific libraries. Figure 1 summarizes the steps from the source code containing SQL statements to an executable program.

4.2.2 Host and Communication Variables

As it is the case for PL/SQL blocks, also the first part of a Pro*C program has a declare section. In a Pro*C program, in a declare section so-called *host variables* are specified. Host variables are the key to the communication between the host program and the database. Declarations of host variables can be placed wherever normal C variable declarations can be placed. Host variables are declared according to the C syntax. Host variables can be of the following data types:

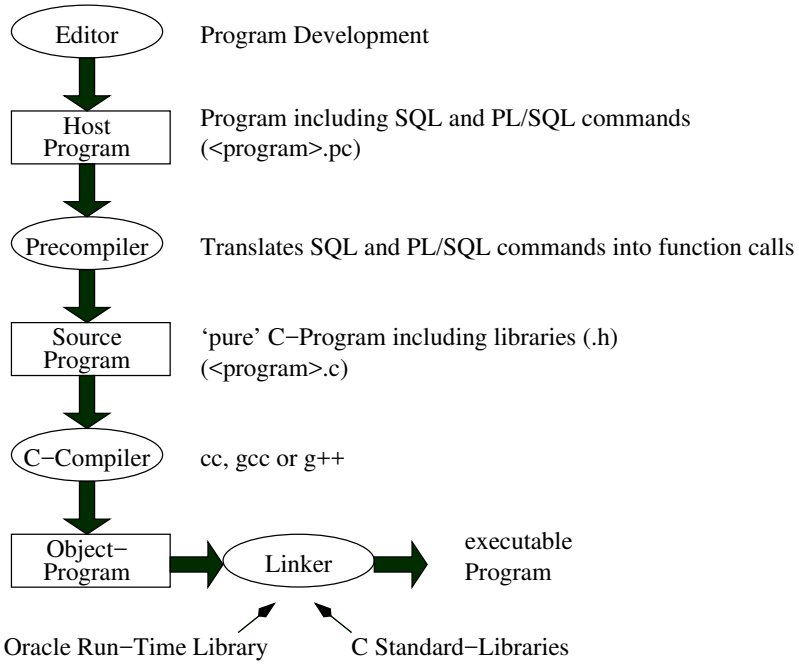

Figure 1: Translation of a Pro*C Program

char <Name>	single character
char <Name>[n]	array of n characters
int	integer
float	floating point
VARCHAR<Name>[n]	variable length strings

VARCHAR[2] is converted by the Pro*C precompiler into a structure with an n-byte character array and a 2-bytes length field. The declaration of host variables occurs in a declare section having the following pattern:

EXEC SQL BEGIN DECLARE SECTION

<Declaration of host variables>

/* e.g., **VARCHAR** `userid[20];` */

/* e.g., **char** `test_ok;` */

EXEC SQL END DECLARE SECTION

In a Pro*C program at most one such a declare section is allowed. The declaration of cursors and exceptions occurs outside of such a declare section for host variables. In a Pro*C program host variables referenced in SQL and PL/SQL statements must be prefixed with a colon ":". Note that it is not possible to use C function calls and most of the pointer expressions as host variable references.

[2]Note: all uppercase letters; **varchar2** is not allowed!

4.2.3 The Communication Area

In addition to host language variables that are needed to pass data between the database and C program (and vice versa), one needs to provide some status variables containing program runtime information. The variables are used to pass status information concerning the database access to the application program so that certain events can be handled in the program properly. The structure containing the status variables is called *SQL Communication Area* or *SQLCA*, for short, and has to be included after the declare section using the statement

EXEC SQL INCLUDE SQLCA.H

In the variables defined in this structure, information about error messages as well as program status information is maintained:

```
struct   sqlca
         {
         /* ub1 */ char    sqlcaid[8];
         /* b4  */ long    sqlabc;
         /* b4  */ long    sqlcode;
         struct
           {
           /* ub2 */ unsigned short sqlerrml;
           /* ub1 */ char           sqlerrmc[70];
           } sqlerrm;
         /* ub1 */ char    sqlerrp[8];
         /* b4  */ long    sqlerrd[6];
         /* ub1 */ char    sqlwarn[8];
         /* ub1 */ char    sqlext[8];
         };
```

The fields in this structure have the following meaning:

`sqlcaid`	Used to identify the SQLCA, set to "SQLCA"
`sqlabc`	Holds the length of the SQLCA structure
`sqlcode`	Holds the status code of the most recently executed SQL (PL/SQL) statement $0 \hat{=}$ No error, statement successfully completed $> 0 \hat{=}$ Statement executed and exception detected. Typical situations are where **fetch** or **select into** returns no rows. $< 0 \hat{=}$ Statement was not executed because of an error; transaction should be rolled back explicitly.
`sqlerrm`	Structure with two components `sqlerrml`: length of the message text in `sqlerrmc`, and `sqlerrmc`: error message text (up to 70 characters) corresponding to the error code recorded in `sqlcode`
`sqlerrp`	Not used

`sqlerrd`	Array of binary integers, has 6 elements: `sqlerrd[0],sqlerrd[1],sqlerrd[3],sqlerrd[6]` not used; `sqlerrd[2]` = number of rows processed by the most recent SQL statement; `sqlerrd[4]` = offset specifying position of most recent parse error of SQL statement.
`sqlwarn`	Array with eight elements used as warning (not error!) flags. Flag is set by assigning it the character 'W'. `sqlwarn[0]`: only set if other flag is set `sqlwarn[1]`: if truncated column value was assigned to a host variable `sqlwarn[2]`: *null* column is not used in computing an aggregate function `sqlwarn[3]`: number of columns in **select** is not equal to number of host variables specified in **into** `sqlwarn[4]`: if every tuple was processed by an **update** or **delete** statement without a **where** clause `sqlwarn[5]`: procedure/function body compilation failed because of a PL/SQL error `sqlwarn[6]` and `sqlwarn[7]`: not used
`sqlext`	not used

Components of this structure can be accessed and verified during runtime, and appropriate handling routines (e.g., exception handling) can be executed to ensure a correct behavior of the application program. If at the end of the program the variable `sqlcode` contains a 0, then the execution of the program has been successful, otherwise an error occurred.

4.2.4 Exception Handling

There are two ways to check the status of your program after executable SQL statements which may result in an error or warning: (1) either by explicitly checking respective components of the SQLCA structure, or (2) by doing automatic error checking and handling using the **WHENEVER** statement. The complete syntax of this statement is

EXEC SQL WHENEVER <condition> <action>;

By using this command, the program then automatically checks the SQLCA for <condition> and executes the given <action>. <condition> can be one of the following:

- SQLERROR: `sqlcode` has a negative value, that is, an error occurred
- SQLWARNING: In this case `sqlwarn[0]` is set due to a warning
- NOT FOUND: `sqlcode` has a positive value, meaning that no row was found that satisfies the **where** condition, or a **select into** or **fetch** statement returned no rows

<action> can be

- STOP: the program exits with an `exit()` call, and all SQL statements that have not been committed so far are rolled back

- CONTINUE: if possible, the program tries to continue with the statement following the error resulting statement
- DO <function>: the program transfers processing to an error handling function named <function>
- GOTO <label>: program execution branches to a labeled statement (see example)

4.2.5 Connecting to the Database

At the beginning of Pro*C program, more precisely, the execution of embedded SQL or PL/SQL statements, one has to connect to the database using a valid Oracle account and password. Connecting to the database occurs trough the embedded SQL statement

EXEC SQL CONNECT :<Account> **IDENTIFIED BY** :<Password>.

Both <Account> and <Password> are host variables of the type **VARCHAR** and must be specified and handled respectively (see also the sample Pro*C program in Section 4.2.7). <Account> and <Password> can be specified in the Pro*C program, but can also be entered at program runtime using, e.g., the C function **scanf**.

4.2.6 Commit and Rollback

Before a program is terminated by the c `exit` function and if no error occurred, database modifications through embedded insert, update, and delete statements must be committed. This is done by using the embedded SQL statement

EXEC SQL COMMIT WORK RELEASE;

If a program error occurred and previous non-committed database modifications need to be undone, the embedded SQL statement

EXEC SQL ROLLBACK WORK RELEASE;

has to be specified in the respective error handling routine of the Pro*C program.

4.2.7 Sample Pro*C Program

The following Pro*C program connects to the database using the database account scott/tiger. The database contains information about employees and departments (see the previous examples used in this tutorial). The user has to enter a salary which then is used to retrieve all employees (from the relation `EMP`) who earn more than the given minimum salary. Retrieving and processing individual result tuples occurs through using a PL/SQL cursor in a C while-loop.

```
/* Declarations */
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

/* Declare section for host variables */
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR userid[20];
   VARCHAR passwd[20];
   int empno;
   VARCHAR ename[15];
   float sal;
   float min_sal;
EXEC SQL END DECLARE SECTION;

/* Load SQL Communication Area */
EXEC SQL INCLUDE SQLCA.H;

main()  /* Main program */
{  int retval;
   /* Catch errors automatically and go to error handling rountine */
   EXEC SQL WHENEVER SQLERROR GOTO error;

   /* Connect to Oracle as SCOTT/TIGER; both are host variables      */
   /* of type VARCHAR; Account and Password are specified explicitly */
   strcpy(userid.arr,"SCOTT");    /* userid.arr := "SCOTT" */
   userid.len=strlen(userid.arr); /* uid.len := 5 */
   strcpy(passwd.arr,"SCOTT");    /* passwd.arr := "TIGER" */
   passwd.len=strlen(passwd.arr); /* passwd.len := 5 */

   EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;

   printf("Connected to ORACLE as: %s\n\n", userid.arr);

   /* Enter minimum salary by user */
   printf("Please enter minimum salary > ");
   retval = scanf("%f", &min_sal);

   if(retval != 1) {
      printf("Input error!!\n");
      EXEC SQL ROLLBACK WORK RELEASE;
      /* Disconnect from ORACLE */
      exit(2); /* Exit program */
   }

   /* Declare cursor; cannot occur in declare section! */
   EXEC SQL DECLARE EMP_CUR CURSOR FOR
   SELECT EMPNO,ENAME,SAL FROM EMP
```

```
   WHERE SAL>=:min_sal;

/* Print Table header, run cursor through result set */
printf("Empployee-ID      Employee-Name       Salary \n");
printf("---------------   -----------------   -------\n");
EXEC SQL OPEN EMP_CUR;
EXEC SQL FETCH EMP_CUR INTO :empno, :ename, :sal; /* Fetch 1.tuple */
while(sqlca.sqlcode==0) { /* are there more tuples ? */
   ename.arr[ename.len] = '\0';    /* "End of String" */
   printf("%15d   %-17s   %7.2f\n",empno,ename.arr,sal);
   EXEC SQL FETCH EMP_CUR INTO :empno, :ename, :sal; /* get next tuple */
}
EXEC SQL CLOSE EMP_CUR;

/* Disconnect from database and terminate program */
EXEC SQL COMMIT WORK RELEASE;
printf("\nDisconnected from ORACLE\n");
exit(0);

/* Error Handling: Print error message */
error: printf("\nError: %.70s \n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}
```

5 Integrity Constraints and Triggers

5.1 Integrity Constraints

In Section 1 we have discussed three types of integrity constraints: not null constraints, primary keys, and unique constraints. In this section we introduce two more types of constraints that can be specified within the **create table** statement: *check constraints* (to restrict possible attribute values), and *foreign key constraints* (to specify interdependencies between relations).

5.1.1 Check Constraints

Often columns in a table must have values that are within a certain range or that satisfy certain conditions. Check constraints allow users to restrict possible attribute values for a column to admissible ones. They can be specified as column constraints or table constraints. The syntax for a check constraint is

[**constraint** <name>] **check**(<condition>)

If a **check** constraint is specified as a column constraint, the condition can only refer that column.

Example: The name of an employee must consist of upper case letters only; the minimum salary of an employee is 500; department numbers must range between 10 and 100:

```
create table EMP
    ( ...,
      ENAME      varchar2(30) constraint check_name
                                check(ENAME = upper(ENAME) ),
      SAL        number(5,2) constraint check_sal  check(SAL >= 500),
      DEPTNO     number(3) constraint check_deptno
                                check(DEPTNO between 10 and 100) );
```

If a **check** constraint is specified as a table constraint, <condition> can refer to all columns of the table. Note that only simple conditions are allowed. For example, it is not allowed to refer to columns of other tables or to formulate queries as check conditions. Furthermore, the functions **sysdate** and **user** cannot be used in a condition. In principle, thus only simple attribute comparisons and logical connectives such as **and**, **or**, and **not** are allowed. A check condition, however, can include a not null constraint:

SAL **number**(5,2) **constraint** `check_sal` **check**(`SAL` **is not null and** `SAL` >= 500),

Without the **not null** condition, the value *null* for the attribute `SAL` would not cause a violation of the constraint.

Example: At least two persons must participate in a project, and the project's start date must be before the project's end date:

create table PROJECT
(...,
PERSONS **number**(5) **constraint** check_pers **check** (PERSONS > 2),
...,
constraint dates_ok **check**(PEND > PSTART));

In this table definition, check_pers is a column constraint and dates_ok is a table constraint.

The database system automatically checks the specified conditions each time a database modification is performed on this relation. For example, the insertion

insert into EMP **values**(7999,'SCOTT','CLERK',7698,'31-OCT-94',450,10);

causes a constraint violation

ORA-02290: check_constraint (CHECK_SAL) violated

and the insertion is rejected.

5.1.2 Foreign Key Constraints

A foreign key constraint (or referential integrity constraint) can be specified as a column constraint or as a table constraint:

[**constraint** <name>] [**foreign key** (<column(s)>)]
references <table>[(<column(s)>)]
[**on delete cascade**]

This constraint specifies a column or a list of columns as a foreign key of the referencing table. The referencing table is called the *child-table*, and the referenced table is called the *parent-table*. In other words, one cannot define a referential integrity constraint that refers to a table R before that table R has been created.

The clause **foreign key** has to be used in addition to the clause **references** if the foreign key includes more than one column. In this case, the constraint has to be specified as a table constraint. The clause **references** defines which columns of the parent-table are referenced. If only the name of the parent-table is given, the list of attributes that build the primary key of that table is assumed.

Example: Each employee in the table EMP must work in a department that is contained in the table DEPT:

create table EMP
(EMPNO **number**(4) **constraint** pk_emp **primary key**,
...,
DEPTNO **number**(3) **constraint** fk_deptno **references** DEPT(DEPTNO));

The column DEPTNO of the table EMP (child-table) builds the foreign key and references the primary key of the table DEPT (parent-table). The relationship between these two tables is illustrated in Figure 2. Since in the table definition above the referential integrity constraint

includes only one column, the clause **foreign key** is not used. It is very important that a foreign key must refer to the complete primary key of a parent-table, not only a subset of the attributes that build the primary key !

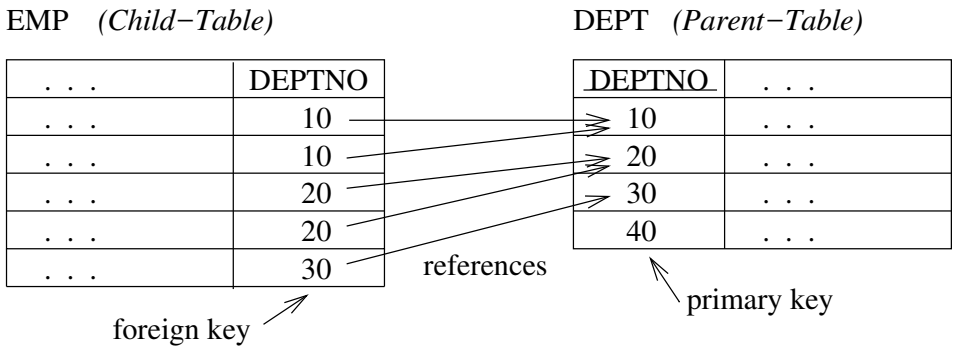

Figure 2: Foreign Key Constraint between the Tables `EMP` and `DEPT`

In order to satisfy a foreign key constraint, each row in the child-table has to satisfy one of the following two conditions:

- the attribute value (list of attribute values) of the foreign key must appear as a primary key value in the parent-table, or
- the attribute value of the foreign key is *null* (in case of a composite foreign key, at least one attribute value of the foreign key is *null*)

According to the above definition for the table `EMP`, an employee must not necessarily work in a department, i.e., for the attribute `DEPTNO` the value *null* is admissible.

Example: Each project manager must be an employee:

```
create table PROJECT
    ( PNO      number(3) constraint prj_pk primary key,
      PMGR     number(4) not null
                         constraint fk_pmgr references  EMP,
      ...);
```

Because only the name of the parent-table is given (`DEPT`), the primary key of this relation is assumed. A foreign key constraint may also refer to the same table, i.e., parent-table and child-table are identical.

Example: Each manager must be an employee:

```
create table EMP
    ( EMPNO    number(4) constraint emp_pk primary key,
      ...
      MGR      number(4) not null
                         constraint fk_mgr references EMP,
      ...
    );
```

5.1.3 More about Column- and Table Constraints

If a constraint is defined within the **create table** command or added using the **alter table** command (compare Section 1.5.5), the constraint is automatically enabled. A constraint can be disabled using the command

alter table <table> **disable**
constraint <name> | **primary key** | **unique**[<column(s)>]
[**cascade**];

To disable a primary key, one must disable all foreign key constraints that depend on this primary key. The clause **cascade** automatically disables foreign key constraints that depend on the (disabled) primary key.

Example: Disable the primary key of the table `DEPT` and disable the foreign key constraint in the table `EMP`:

alter table `DEPT` **disable primary key cascade**;

In order to enable an integrity constraint, the clause **enable** is used instead of **disable**. A constraint can only be enabled successfully if no tuple in the table violates the constraint. Otherwise an error message is displayed. Note that for enabling/disabling an integrity constraint it is important that you have named the constraints.

In order to identify those tuples that violate an integrity constraint whose activation failed, one can use the clause **exceptions into** `EXCEPTIONS` with the **alter table** statement. `EXCEPTIONS` is a table that stores information about violating tuples.[3] Each tuple in this table is identified by the attribute `ROWID`. Every tuple in a database has a pseudo-column `ROWID` that is used to identify tuples. Besides the rowid, the name of the table, the table owner as well as the name of the violated constraint are stored.

Example: Assume we want to add an integrity constraint to our table `EMP` which requires that each manager must earn more than 4000:

alter table `EMP` **add constraint** `manager_sal`
check(`JOB` != 'MANAGER' **or** `SAL` >= 4000)
exceptions into `EXCEPTIONS`;

If the table `EMP` already contains tuples that violate the constraint, the constraint cannot be activated and information about violating tuples is automatically inserted into the table `EXCEPTIONS`.

Detailed information about the violating tuples can be obtained by joining the tables `EMP` and `EXCEPTIONS`, based on the join attribute `ROWID`:

select `EMP.*, CONSTRAINT` **from** `EMP, EXCEPTIONS`
where `EMP.ROWID = EXCEPTIONS.ROW_ID`;

[3] Before this table can be used, it must be created using the SQL script `utlexcept.sql` which can be found in the directory `$ORACLE_HOME/rdbms/admin`.

Tuples contained in the query result now can be modified (e.g., by increasing the salary of managers) such that adding the constraint can be performed successfully. Note that it is important to delete "old" violations from the relation `EXCEPTIONS` before it is used again.

If a table is used as a reference of a foreign key, this table can only be dropped using the command **drop table** <table> **cascade constraints**;. All other database objects that refer to this table (e.g., triggers, see Section 5.2) remain in the database system, but they are not valid.

Information about integrity constraints, their status (enabled, disabled) etc. is stored in the data dictionary, more precisely, in the tables `USER_CONSTRAINTS` and `USER_CONS_CONSTRAINTS`.

5.2 Triggers

5.2.1 Overview

The different types of integrity constraints discussed so far provide a *declarative* mechanism to associate "simple" conditions with a table such as a primary key, foreign keys or domain constraints. Complex integrity constraints that refer to several tables and attributes (as they are known as assertions in the SQL standard) cannot be specified within table definitions. *Triggers*, in contrast, provide a procedural technique to specify and maintain integrity constraints. Triggers even allow users to specify more complex integrity constraints since a trigger essentially is a PL/SQL procedure. Such a procedure is associated with a table and is automatically called by the database system whenever a certain modification (*event*) occurs on that table. Modifications on a table may include **insert**, **update**, and **delete** operations (Oracle 7).

5.2.2 Structure of Triggers

A trigger definition consists of the following (optional) components:

- *trigger name*
  create [**or replace**] **trigger** <trigger name>
- *trigger time point*
  before | **after**
- *triggering event(s)*
  insert or update [**of** <column(s)>] **or delete on** <table>
- *trigger type* (optional)
  for each row
- *trigger restriction* (only for **for each row** triggers !)
  when (<condition>)
- *trigger body*
  <PL/SQL block>

The clause **replace** re-creates a previous trigger definition having the same <trigger name>. The name of a trigger can be chosen arbitrarily, but it is a good programming style to use

a trigger name that reflects the table and the event(s), e.g., `upd_ins_EMP`. A trigger can be invoked **before** or **after** the triggering event. The *triggering event* specifies before (after) which operations on the table <table> the trigger is executed. A single event is an **insert**, an **update**, or a **delete**; events can be combined using the logical connective **or**. If for an **update** trigger no columns are specified, the trigger is executed after (before) <table> is updated. If the trigger should only be executed when certain columns are updated, these columns must be specified after the event **update**. If a trigger is used to maintain an integrity constraint, the triggering events typically correspond to the operations that can violate the integrity constraint.

In order to program triggers efficiently (and correctly) it is essential to understand the difference between a *row level trigger* and a *statement level trigger*. A row level trigger is defined using the clause **for each row**. If this clause is not given, the trigger is assumed to be a statement trigger. A row trigger executes once for each row after (before) the event. In contrast, a statement trigger is executed once after (before) the event, independent of how many rows are affected by the event. For example, a row trigger with the event specification **after update** is executed once for each row affected by the update. Thus, if the update affects 20 tuples, the trigger is executed 20 times, for each row at a time. In contrast, a statement trigger is only executed once.

When combining the different types of triggers, there are twelve possible trigger configurations that can be defined for a table:

event	trigger time point		trigger type	
	before	**after**	statement	row
insert	X	X	X	X
update	X	X	X	X
delete	X	X	X	X

Figure 3: Trigger Types

Row triggers have some special features that are not provided by statement triggers:

Only with a row trigger it is possible to access the attribute values of a tuple before and after the modification (because the trigger is executed once for each tuple). For an **update** trigger, the old attribute value can be accessed using **:old**.<column> and the new attribute value can be accessed using **:new**.<column>. For an **insert** trigger, only **:new**.<column> can be used, and for a **delete** trigger only **:old**.<column> can be used (because there exists no old, respectively, new value of the tuple). In these cases, **:new**.<column> refers to the attribute value of <column> of the inserted tuple, and **:old**.<column> refers to the attribute value of <column> of the deleted tuple. In a row trigger thus it is possible to specify comparisons between old and new attribute values in the PL/SQL block, e.g., "**if :old**.`SAL` < **:new**.`SAL` **then** ...". If for a row trigger the trigger time point **before** is specified, it is even possible to modify the new values of the row, e.g., **:new**.`SAL` := **:new**.`SAL` $*$ 1.05 or **:new**.`SAL` := **:old**.`SAL`. Such modifications are not possible with **after** row triggers. In general, it is advisable to use a **after** row trigger if the new row is not modified in the PL/SQL block. Oracle then can process

these triggers more efficiently. Statement level triggers are in general only used in combination with the trigger time point **after**.

In a trigger definition the **when** clause can only be used in combination with a **for each row** trigger. The clause is used to further restrict when the trigger is executed. For the specification of the condition in the **when** clause, the same restrictions as for the **check** clause hold. The only exceptions are that the functions **sysdate** and **user** can be used, and that it is possible to refer to the old/new attribute values of the actual row. In the latter case, the colon ":" must not be used, i.e., only **old**.<attribute> and **new**.<attribute>.

The trigger body consists of a PL/SQL block. All SQL and PL/SQL commands except the two statements **commit** and **rollback** can be used in a trigger's PL/SQL block. Furthermore, additional **if** constructs allow to execute certain parts of the PL/SQL block depending on the triggering event. For this, the three constructs **if inserting**, **if updating**[('<column>')], and **if deleting** exist. They can be used as shown in the following example:

```
create or replace trigger emp_check
after insert or delete or update on EMP
for each row
begin
   if inserting then
      <PL/SQL block>
   end if;
   if updating then
      <PL/SQL block>
   end if;
   if deleting then
      <PL/SQL block>
   end if;
end;
```

It is important to understand that the execution of a trigger's PL/SQL block builds a part of the transaction that contains the triggering event. Thus, for example, an **insert** statement in a PL/SQL block can cause another trigger to be executed. Multiple triggers and modifications thus can lead to a cascading execution of triggers. Such a sequence of triggers terminates successfully if (1) no exception is raised within a PL/SQL block, and (2) no declaratively specified integrity constraint is violated. If a trigger raises an exception in a PL/SQL block, all modifications up to the beginning of the transaction are rolled back. In the PL/SQL block of a trigger, an exception can be raised using the statement **raise_application_error** (see Section 4.1.5). This statement causes an implicit **rollback**. In combination with a row trigger, **raise_application_error** can refer to old/new values of modified rows:

```
raise_application_error(−20020, 'Salary increase from ' || to_char(:old.SAL) || ' to '
                                    to_char(:new.SAL) || ' is too high');  or

raise_application_error(−20030, 'Employee Id ' ||
                                 to_char(:new .EMPNO) || ' does not exist.');
```

5.2.3 Example Triggers

Suppose we have to maintain the following integrity constraint: "The salary of an employee different from the president cannot be decreased and must also not be increased more than 10%. Furthermore, depending on the job title, each salary must lie within a certain salary range.

We assume a table SALGRADE that stores the minimum (MINSAL) and maximum (MAXSAL) salary for each job title (JOB). Since the above condition can be checked for each employee individually, we define the following row trigger:

trig1.sql

```
create or replace trigger check_salary_EMP
after insert or update of SAL, JOB on EMP
for each row
when (new.JOB != 'PRESIDENT') -- trigger restriction
declare
  minsal, maxsal SALGRADE.MAXSAL%TYPE;
begin
  -- retrieve minimum and maximum salary for JOB
  select MINSAL, MAXSAL into minsal, maxsal from SALGRADE
  where JOB = :new.JOB;
  -- If the new salary has been decreased or does not lie within the salary range,
  -- raise an exception
  if (:new.SAL < minsal or :new.SAL > maxsal) then
    raise_application_error(-20225, 'Salary range exceeded');
  elsif (:new.SAL < :old.SAL) then
    raise_application_error(-20230, 'Salary has been decreased');
  elsif (:new.SAL > 1.1 * :old.SAL) then
    raise_application_error(-20235, 'More than 10% salary increase');
  end if;
end;
```

We use an **after** trigger because the inserted or updated row is not changed within the PL/SQL block (e.g., in case of a constraint violation, it would be possible to restore the old attribute values).

Note that also modifications on the table SALGRADE can cause a constraint violation. In order to maintain the complete condition we define the following trigger on the table SALGRADE. In case of a violation by an **update** modification, however, we do not raise an exception, but restore the old attribute values.

trig2.sql

```
create or replace trigger check_salary_SALGRADE
before update or delete on SALGRADE
for each row
when (new.MINSAL > old.MINSAL
      or new.MAXSAL < old.MAXSAL)
      -- only restricting a salary range can cause a constraint violation
declare
  job_emps number(3) := 0;
begin
  if deleting then -- Does there still exist an employee having the deleted job ?
    select count(*) into job_emps from EMP
    where JOB = :old.JOB;
    if job_emps != 0 then
      raise_application_error(-20240, ' There still exist employees with the job ' ||
                                        :old.JOB);
    end if ;
  end if ;
  if updating then
    -- Are there employees whose salary does not lie within the modified salary range ?
    select count(*) into job_emps from EMP
    where JOB = :new.JOB
      and SAL not between :new.MINSAL and :new.MAXSAL;
    if job_emps != 0 then -- restore old salary ranges
      :new.MINSAL := :old.MINSAL;
      :new.MAXSAL := :old.MAXSAL;
    end if ;
  end if ;
end;
```

In this case a **before** trigger must be used to restore the old attribute values of an updated row.

Suppose we furthermore have a column BUDGET in our table DEPT that is used to store the budget available for each department. Assume the integrity constraint requires that the total of all salaries in a department must not exceed the department's budget. Critical operations on the relation EMP are insertions into EMP and updates on the attributes SAL or DEPTNO.

trig3.sql

```
create or replace trigger check_budget_EMP
after insert or update of SAL, DEPTNO on EMP
declare
  cursor DEPT_CUR is
    select DEPTNO, BUDGET from DEPT;
  DNO       DEPT.DEPTNO%TYPE;
  ALLSAL    DEPT.BUDGET%TYPE;
  DEPT_SAL number;
begin
  open DEPT_CUR;
  loop
    fetch DEPT_CUR into DNO, ALLSAL;
    exit when DEPT_CUR%NOTFOUND;
    select sum(SAL) into DEPT_SAL from EMP
    where DEPTNO = DNO;
    if DEPT_SAL > ALLSAL then
      raise_application_error(-20325, 'Total of salaries in the department ' ||
                              to_char(DNO) || ' exceeds budget');
    end if;
  end loop;
  close DEPT_CUR;
end;
```

In this case we use a statement trigger on the relation EMP because we have to apply an aggregate function on the salary of all employees that work in a particular department. For the relation DEPT, we also have to define a trigger which, however, can be formulated as a row trigger.

5.2.4 Programming Triggers

For programmers, row triggers are the most critical type of triggers because they include several restrictions. In order to ensure read consistency, ORACLE performs an exclusive lock on the table at the beginning of an **insert**, **update**, or **delete** statement. That is, other users cannot access this table until modifications have been successfully completed. In this case, the table currently modified is said to be a *mutating* table. The only way to access a mutating table in a trigger is to use **:old**.<column> and **:new**.<column> in connection with a row trigger.

Example of an erroneous row trigger:

```
create trigger check_sal_EMP
after update of SAL on EMP
for each row
```

```
declare
  sal_sum number;
begin
  select sum(SAL) into sal_sum from EMP;
  ...;
end;
```

For example, if an **update** statement of the form **update** EMP **set** SAL = SAL ∗ 1.1 is executed on the table EMP, the above trigger is executed once for each modified row. While the table is being modified by the update command, it is not possible to access all tuples of the table using the **select** command, because it is locked. In this case we get the error message

```
ORA-04091: table EMP is mutating, trigger may not read or modify it
ORA-06512: at line 4
ORA-04088: error during execution of trigger 'CHECK_SAL_EMP'
```

The only way to access the table, or more precisely, to access the modified tuple, is to use **:old**.<column> and **:new**.<column>.

It is recommended to follow the rules below for the definition of integrity maintaining triggers:

identify operations and tables that are critical for the integrity constraint
  for each such table check
    if constraint can be checked at row level **then**
      if checked rows are modified in trigger **then**
        use **before** row trigger
      else use **after** row trigger
    else
    use **after** statement trigger

Triggers are not exclusively used for integrity maintenance. They can also be used for

- Monitoring purposes, such as the monitoring of user accesses and modifications on certain sensitive tables.
- Logging actions, e.g., on tables:

```
create trigger LOG_EMP
after insert or update or delete on EMP
begin
  if inserting then
    insert into EMP_LOG values(user, 'INSERT', sysdate);
```

```
    end if;
    if updating then
       insert into EMP_LOG values(user, 'UPDATE', sysdate);
    end if;
    if deleting then
       insert into EMP_LOG values(user, 'DELETE', sysdate);
    end if;
end;
```

By using a row trigger, even the attribute values of the modified tuples can be stored in the table EMP_LOG.

- automatic propagation of modifications. For example, if a manager is transfered to another department, a trigger can be defined that automatically transfers the manager's employees to the new department.

5.2.5 More about Triggers

If a trigger is specified within the SQL*Plus shell, the definition must end with a point "." in the last line. Issuing the command **run** causes SQL*Plus to compile this trigger definition. A trigger definition can be loaded from a file using the command @. Note that the last line in the file must consist of a slash "/".

A trigger definition cannot be changed, it can only be re-created using the **or replace** clause. The command **drop** <trigger name> deletes a trigger.

After a trigger definition has been successfully compiled, the trigger automatically is enabled. The command **alter trigger** <trigger name> **disable** is used to deactivate a trigger. All triggers defined on a table can be (de)activated using the command

alter table <Tabelle> **enable** | **disable all trigger**;

The data dictionary stores information about triggers in the table USER_TRIGGERS. The information includes the trigger name, type, table, and the code for the PL/SQL block.

6 System Architecture

In the following sections we discuss the main components of the ORACLE DBMS (Version 7.X) architecture (Section 6.1) and the logical and physical database structures (Sections 6.2 and 6.3). We furthermore sketch how SQL statements are processed (Section 6.4) and how database objects are created (Section 6.5).

6.1 Storage Management and Processes

The ORACLE DBMS server is based on a so-called *Multi-Server Architecture*. The server is responsible for processing all database activities such as the execution of SQL statements, user and resource management, and storage management. Although there is only one copy of the program code for the DBMS server, to each user connected to the server logically a separate server is assigned. The following figure illustrates the architecture of the ORACLE DBMS consisting of storage structures, processes, and files.

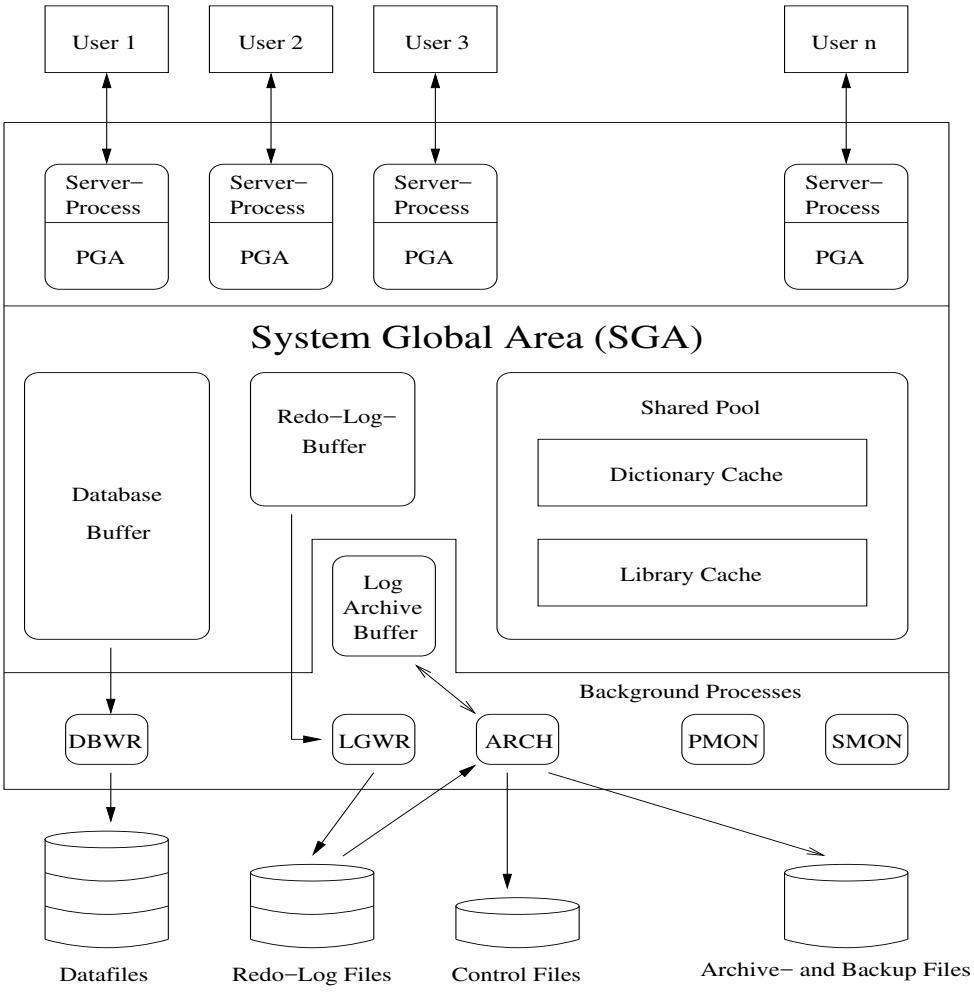

Figure 4: Oracle System Architecture

Each time a database is started on the server (*instance startup*), a portion of the computer's main memory is allocated, the so-called *System Global Area (SGA)*. The SGA consists of the *shared pool*, the *database buffer*, and the *redo-log buffer*. Furthermore, several background processes are started. The combination of SGA and processes is called *database instance*. The memory and processes associated with an instance are responsible for efficiently managing the data stored in the database, and to allow users accessing the database concurrently. The ORACLE server can manage multiple instances, typically each instance is associated with a particular application domain.

The SGA serves as that part of the memory where all database operations occur. If several users connect to an instance at the same time, they all share the SGA. The information stored in the SGA can be subdivided into the following three caches.

Database Buffer The database buffer is a cache in the SGA used to hold the data blocks that are read from data files. Blocks can contain table data, index data etc. Data blocks are modified in the database buffer. ORACLE manages the space available in the database buffer by using a least recently used (LRU) algorithm. When free space is needed in the buffer, the least recently used blocks will be written out to the data files. The size of the database buffer has a major impact on the overall performance of a database.

Redo-Log-Buffer This buffer contains information about changes of data blocks in the database buffer. While the redo-log-buffer is filled during data modifications, the log writer process writes information about the modifications to the redo-log files. These files are used after, e.g., a system crash, in order to restore the database (database recovery).

Shared Pool The shared pool is the part of the SGA that is used by all users. The main components of this pool are the *dictionary cache* and the *library cache*. Information about database objects is stored in the data dictionary tables. When information is needed by the database, for example, to check whether a table column specified in a query exists, the dictionary tables are read and the data returned is stored in the dictionary cache. Note that all SQL statements require accessing the data dictionary. Thus keeping relevant portions of the dictionary in the cache may increase the performance. The library cache contains information about the most recently issued SQL commands such as the parse tree and query execution plan. If the same SQL statement is issued several times, it need not be parsed again and all information about executing the statement can be retrieved from the library cache.

Further storage structures in the computer's main memory are the *log-archive buffer* (optional) and the *Program Global Area* (PGA). The log-archive buffer is used to temporarily cache redo-log entries that are to be archived in special files. The PGA is the area in the memory that is used by a single ORACLE user process. It contains the user's context area (cursors, variables etc.), as well as process information. The memory in the PGA is not sharable.

For each database instance, there is a set of processes. These processes maintain and enforce the relationships between the database's physical structures and memory structures. The number

of processes varies depending on the instance configuration. One can distinguish between user processes and ORACLE processes. ORACLE processes are typically background processes that perform I/O operations at database run-time.

DBWR This process is responsible for managing the contents of the database buffer and the dictionary cache. For this, DBWR writes modified data blocks to the data files. The process only writes blocks to the files if more blocks are going to be read into the buffer than free blocks exist.

LGWR This process manages writing the contents of the redo-log-buffer to the redo-log files.

SMON When a database instance is started, the system monitor process performs instance recovery as needed (e.g., after a system crash). It cleans up the database from aborted transactions and objects involved. In particular, this process is responsible for coalescing contiguous free extents to larger extents (space defragmentation, see Section 6.2).

PMON The process monitor process cleans up behind failed user processes and it also cleans up the resources used by these processes. Like SMON, PMON wakes up periodically to check whether it is needed.

ARCH (optional) The LGWR background process writes to the redo-log files in a cyclic fashion. Once the last redo-log file is filled, LGWR overwrites the contents of the first redo-log file. It is possible to run a database instance in the archive-log mode. In this case the ARCH process copies redo-log entries to archive files before the entries are overwritten by LGWR. Thus it is possible to restore the contents of the database to any time after the archive-log mode was started.

USER The task of this process is to communicate with other processes started by application programs such as SQL*Plus. The USER process then is responsible for sending respective operations and requests to the SGA or PGA. This includes, for example, reading data blocks.

6.2 Logical Database Structures

For the architecture of an ORACLE database we distinguish between logical and physical database structures that make up a database. Logical structures describe logical areas of storage (name spaces) where objects such as tables can be stored. Physical structures, in contrast, are determined by the operating system files that constitute the database.

The logical database structures include:

Database A database consists of one or more storage divisions, so-called tablespaces.

Tablespaces A tablespace is a logical division of a database. All database objects are logically stored in tablespaces. Each database has at least one tablespace, the SYSTEM tablespace, that contains the data dictionary. Other tablespaces can be created and used for different applications or tasks.

Segments If a database object (e.g., a table or a cluster) is created, automatically a portion of the tablespace is allocated. This portion is called a segment. For each table there is a table segment. For indexes so-called index segments are allocated. The segment associated with a database object belongs to exactly one tablespace.

Extent An extent is the smallest logical storage unit that can be allocated for a database object, and it consists a contiguous sequence of data blocks! If the size of a database object increases (e.g., due to insertions of tuples into a table), an additional extent is allocated for the object. Information about the extents allocated for database objects can be found in the data dictionary view `USER_EXTENTS`.

A special type of segments are *rollback segments*. They don't contain a database object, but contain a "before image" of modified data for which the modifying transaction has not yet been committed. Modifications are undone using rollback segments. ORACLE uses rollback segments in order to maintain read consistency among multiple users. Furthermore, rollback segments are used to restore the "before image" of modified tuples in the event of a rollback of the modifying transaction.

Typically, an extra tablespace (RBS) is used to store rollback segments. This tablespace can be defined during the creation of a database. The size of this tablespace and its segments depends on the type and size of transactions that are typically performed by application programs.

A database typically consists of a SYSTEM tablespace containing the data dictionary and further internal tables, procedures etc., and a tablespace for rollback segments. Additional tablespaces include a tablespace for user data (USERS), a tablespace for temporary query results and tables (TEMP), and a tablespace used by applications such as SQL*Forms (TOOLS).

6.3 Physical Database Structure

The physical database structure of an ORACLE database is determined by files and data blocks:

Data Files A tablespace consists of one or more operating system files that are stored on disk. Thus a database essentially is a collection of data files that can be stored on different storage devices (magnetic tape, optical disks etc.). Typically, only magnetic disks are used. Multiple data files for a tablespace allows the server to distribute a database object over multiple disks (depending on the size of the object).

Blocks An extent consists of one or more contiguous ORACLE data blocks. A block determines the finest level of granularity of where data can be stored. One data block corresponds to a specific number of bytes of physical database space on disk. A data block size is specified for each ORACLE database when the database is created. A database uses and allocates free database space in ORACLE data blocks. Information about data blocks can be retrieved from the data dictionary views `USER_SEGMENTS` and `USER_EXTENTS`. These views show how many blocks are allocated for a database object and how many blocks are available (free) in a segment/extent.

As mentioned in Section 6.1, aside from datafiles three further types of files are associated with a database instance:

Redo-Log Files Each database instance maintains a set of redo-log files. These files are used to record logs of all transactions. The logs are used to recover the database's transactions in their proper order in the event of a database crash (the recovering operations are called roll forward). When a transaction is executed, modifications are entered in the redo-log buffer, while the blocks affected by the transactions are not immediately written back to disk, thus allowing optimizing the performance through batch writes.

Control Files Each database instance has at least one control file. In this file the name of the database instance and the locations (disks) of the data files and redo-log files are recorded. Each time an instance is started, the data and redo-log files are determined by using the control file(s).

Archive/Backup Files If an instance is running in the archive-log mode, the ARCH process archives the modifications of the redo-log files in extra archive or backup files. In contrast to redo-log files, these files are typically not overwritten.

The following ER schema illustrates the architecture of an ORACLE database instance and the relationships between physical and logical database structures (relationships can be read as "consists of").

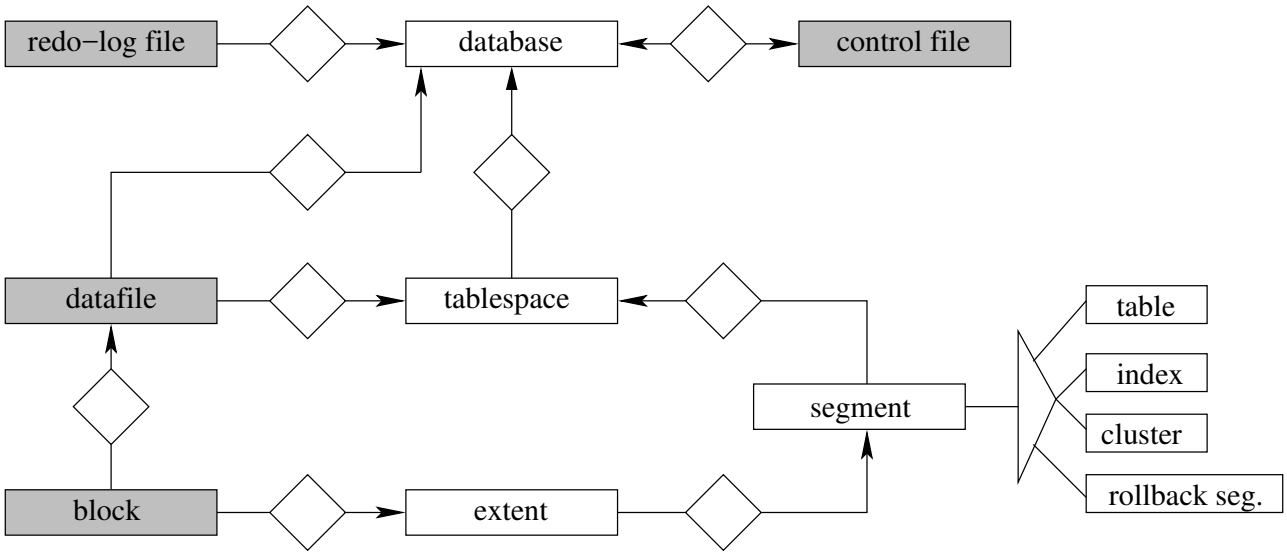

Figure 5: Relationships between logical and physical database structures

6.4 Steps in Processing an SQL Statement

In the following we sketch how an SQL statement is processed by the ORACLE server and which processes and buffers involved.

1. Assume a user (working with SQL*Plus) issues an update statement on the table `TAB` such that more than one tuple is affected by the update. The statement is passed to the server by the USER process. Then the server (or rather the query processor) checks whether this statement is already contained in the library cache such that the corresponding information (parse tree, execution plan) can be used. If the statement can not be found, it is parsed and after verifying the statement (user privileges, affected tables and columns) using data from the dictionary cache, a query execution plan is generated by the query optimizer. Together with the parse tree, this plan is stored in the library cache.

2. For the objects affected by the statement (here the table `TAB`) it is checked, whether the corresponding data blocks already exist in the database buffer. If not, the USER process reads the data blocks into the database buffer. If there is not enough space in the buffer, the least recently used blocks of other objects are written back to the disk by the DBWR process.

3. The modifications of the tuples affected by the update occurs in the database buffer. Before the data blocks are modified, the "before image" of the tuples is written to the rollback segments by the DBWR process.

4. While the redo-log buffer is filled during the data block modifications, the LGWR process writes entries from the redo-log buffer to the redo-log files.

5. After all tuples (or rather the corresponding data blocks) have been modified in the database buffer, the modifications can be committed by the user using the **commit** command.

6. As long as no **commit** has been issued by the user, modifications can be undone using the **rollback** statement. In this case, the modified data blocks in the database buffer are overwritten by the original blocks stored in the rollback segments.

7. If the user issues a **commit**, the space allocated for the blocks in the rollback segments is deallocated and can be used by other transactions. Furthermore, the modified blocks in the database buffer are unlocked such that other users now can read the modified blocks. The end of the transaction (more precisely the **commit**) is recorded in the redo-log files. The modified blocks are only written to the disk by the DBWR process if the space allocated for the blocks is needed for other blocks.

6.5 Creating Database Objects

For database objects (tables, indexes, clusters) that require their own storage area, a segment in a tablespace is allocated. Since the system typically does not know what the size of the

database object will be, some default storage parameters are used. The user, however, has the possibility to explicitly specify the storage parameters using a storage clause in, e.g., the **create table** statement. This specification then overwrites the system parameters and allows the user to specify the (expected) storage size of the object in terms of extents.

Suppose the following table definition that includes a storage clause:

```
create table STOCKS
        (ITEM       varchar2(30),
         QUANTITY number(4))
storage (initial 1M next 400k
          minextents 1 maxextents 20 pctincrease 50);
```

initial and **next** specify the size of the first and next extents, respectively. In the definition above, the initial extent has a size of 1MB, and the next extent has a size of 400KB. The parameter **minextents** specifies the total number of extents allocated when the segment is created. This parameter allows the user to allocate a large amount of space when an object is created, even if the space available is not contiguous. The default and minimum value is 1. The parameter **maxextents** specifies the admissible number of extents. The parameter **pctincrease** specifies the percent by which each extent after the second grows over the previous extent. The default value is 50, meaning that each subsequent extent is 50% larger than the preceding extent. Based on the above table definition, we thus would get the following logical database structure for the table `STOCKS` (assuming that four extents have already been allocated):

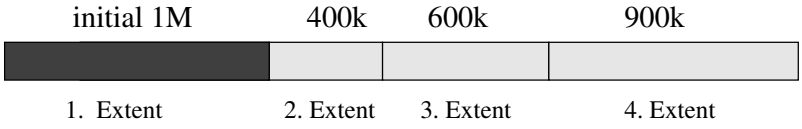

Figure 6: Logical Storage Structure of the Table `STOCKS`

If the space required for a database object is known before creation, already the initial extent should be big enough to hold the database object. In this case, the Oracle server (more precisely the resource manager) tries to allocate contiguous data blocks on disks for this object, thus the defragmentation of data blocks associated with a database object can be prevented.

For indexes a storage clause can be specified as well

```
create index STOCK_IDX on STOCKS(ITEM)
        storage (initial 200k next 100k
                 minextents 1 maxextents 5);
```