International Black Sea University

Data Base management service

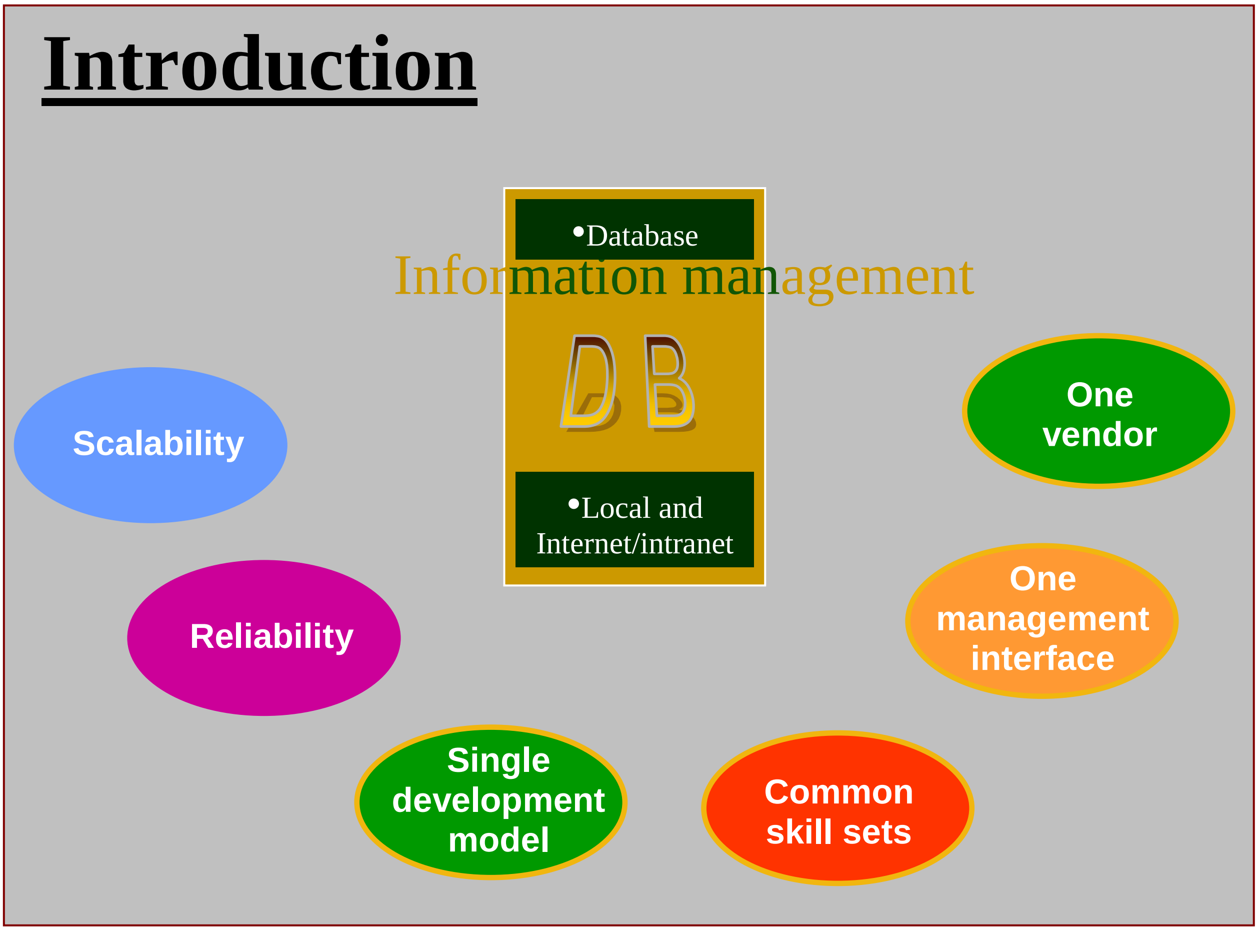
(Database I, II)

Lecture Notes

Books and sites:

- http://www.w3schools.com/sql/default.asp
- http://www.sqlcourse.com
- http://plsql-tutorial.com
- SQLP_1Z0-007(module1).pdf
- SQLP_1Z0-007(module2).pdf
- SQLP_1Z0-007(module3).pdf

Designed by Prof. Nodar Momtselidze



Introduction

Information management

- Database

- Local and Internet/intranet

Scalability

Reliability

Single development model

Common skill sets

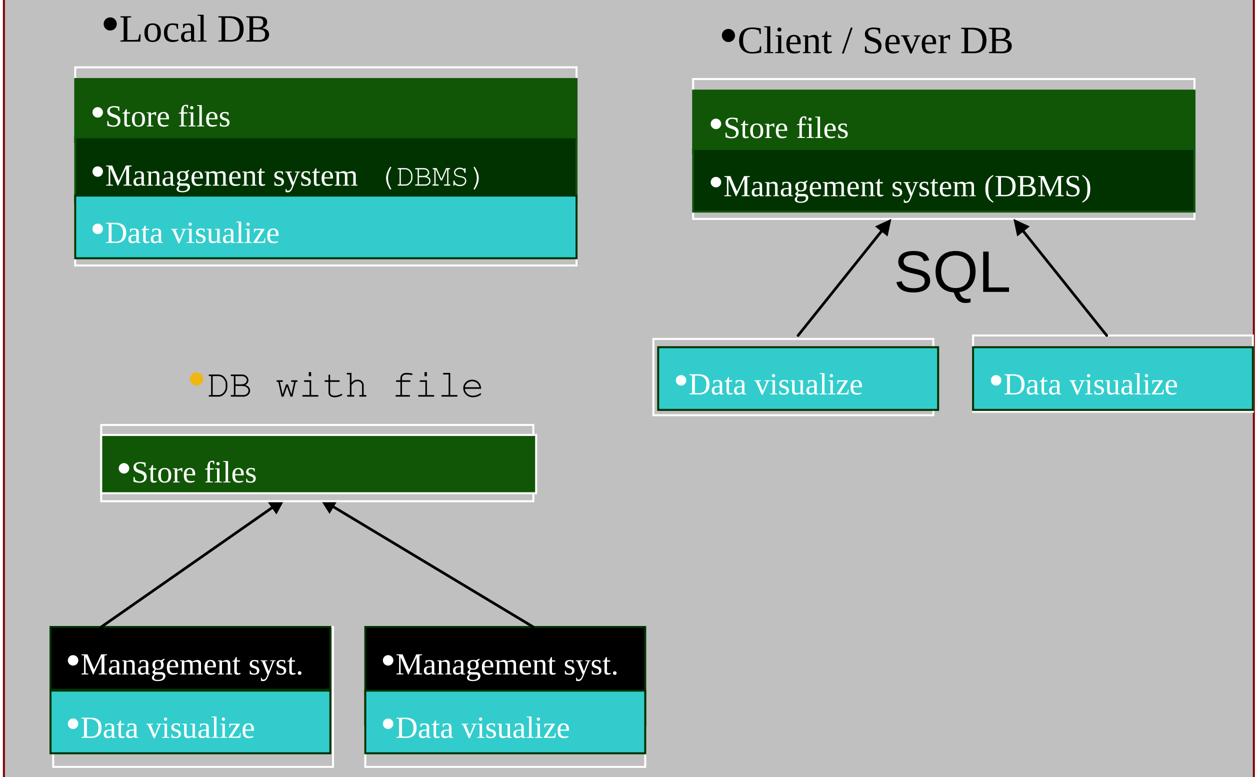
One management interface

One vendor

Data Bases

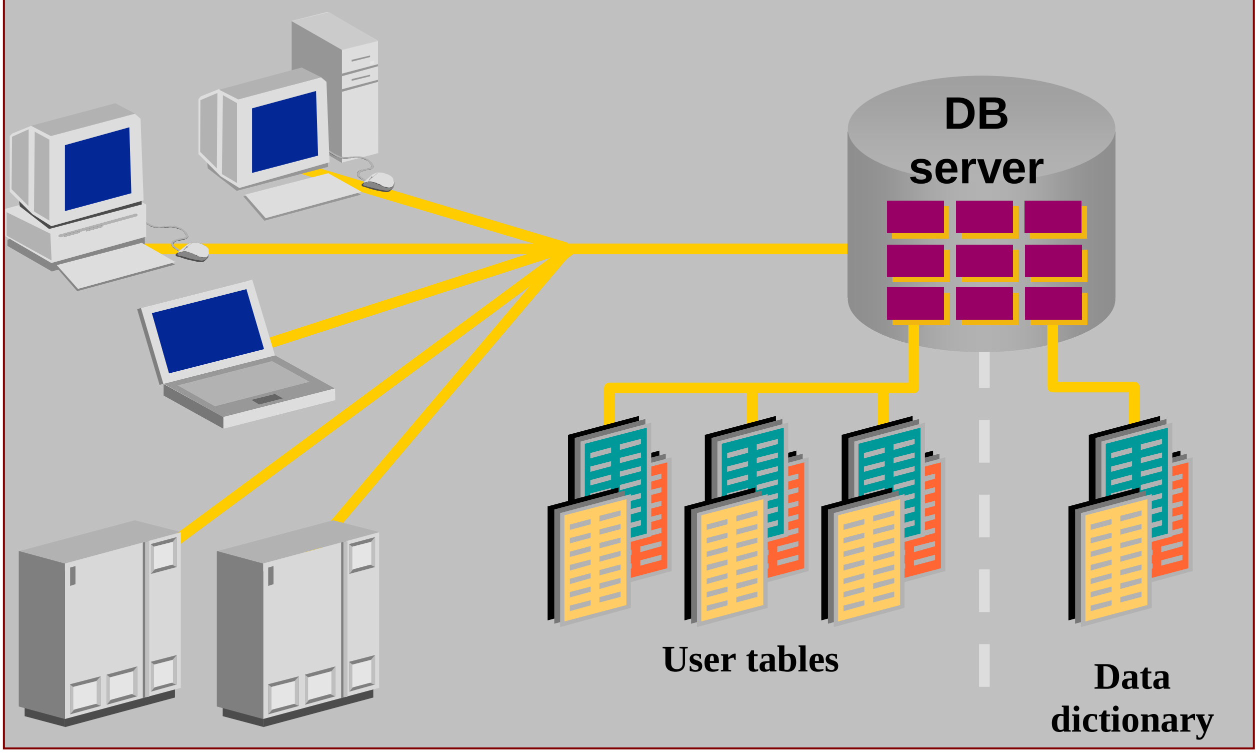
- Data base main concepts
- Data store
- Data Base Management System （ **DBMS** ）
- Network,
- Data visualize
- Local and Corporative Data Base

Data base principal types

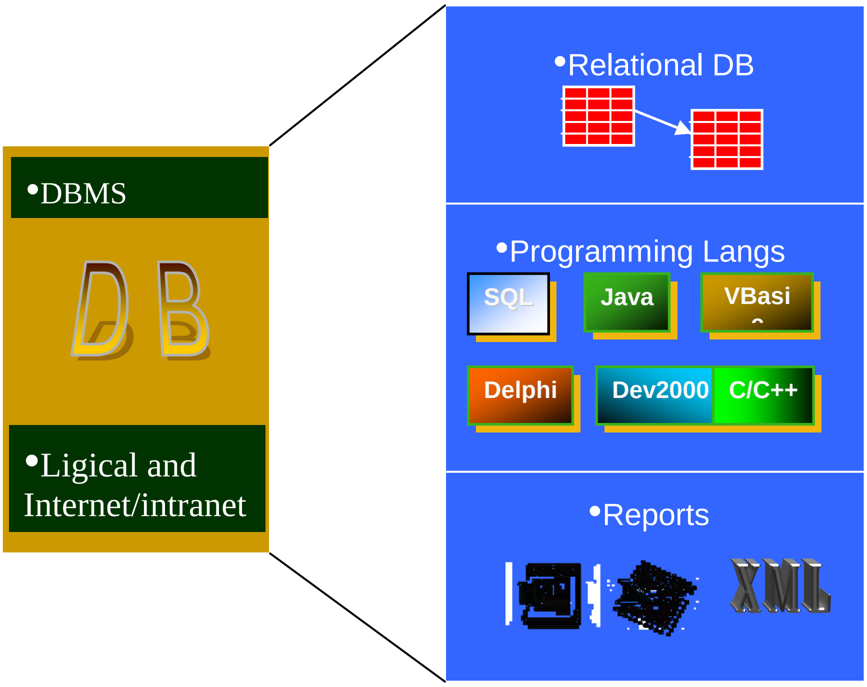


- Local DB
  - Store files
  - Management system (DBMS)
  - Data visualize
- Client / Sever DB
  - Store files
  - Management system (DBMS)
  - SQL
  - Data visualize
  - Data visualize
- DB with file
  - Store files
  - Management syst.
  - Data visualize
  - Management syst.
  - Data visualize

Relational DBMS



DBMS
Programing forms and reports



- DBMS

DB

- Ligical and Internet/intranet

- Relational DB

- Programming Langs

SQL Java VBasi

Delphi Dev2000 C/C++

- Reports

XML

- Oracle
- DB2
- Sybase
- Intermix
- MS SQL
- Interbase
- MySQL

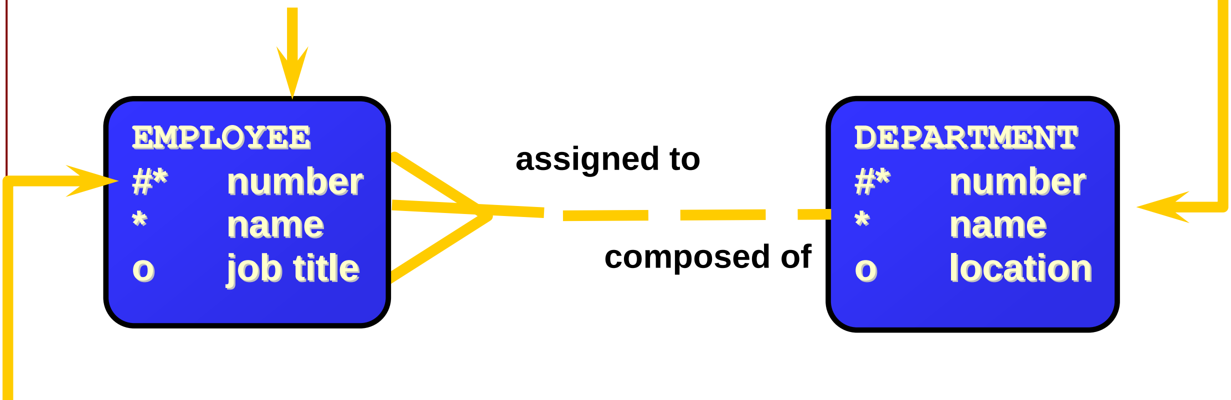
CASE - technology

- Schema creation
- Entity / Relationship structures
- Relational DB creation
- Exercises

Entity Relationship Modeling Conventions

Entity
Soft box
Singular, unique name
Uppercase
Synonym in parentheses

Attribute
Singular name
Lowercase
Mandatory marked with "*"
Optional marked with "o"

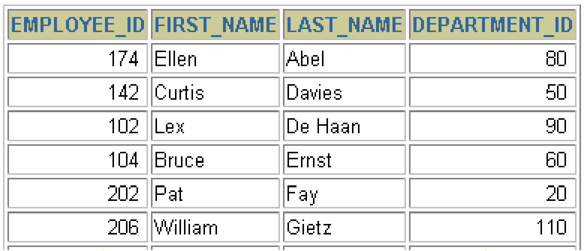


Unique Identifier (UID)
Primary marked with "#"
Secondary marked with "(#)"

Tables and relationship

- **Every string have to be unique by the help of primary key (PK)**
- **Tables could be connected logically by the help of foreign keys (FK)**

Table name: EMPLOYEES

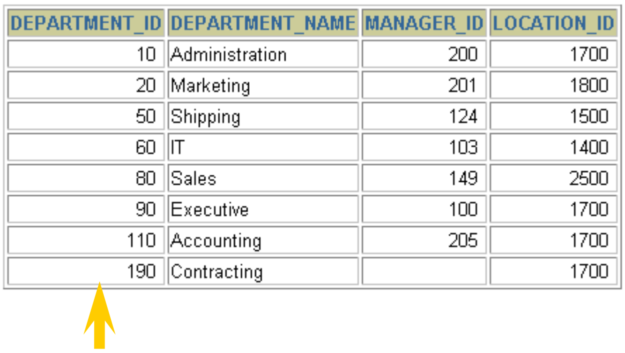


EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
174	Ellen	Abel	80
142	Curtis	Davies	50
102	Lex	De Haan	90
104	Bruce	Ernst	60
202	Pat	Fay	20
206	William	Gietz	110

Primary key

Foreign key

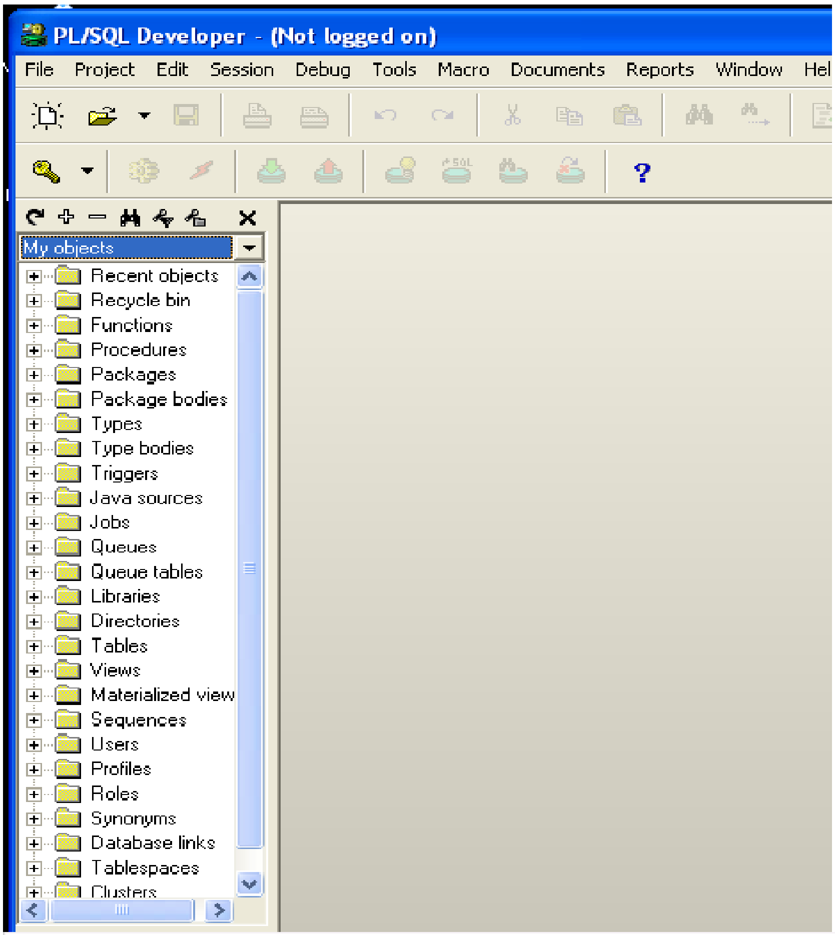
Table name: DEPARTMENTS



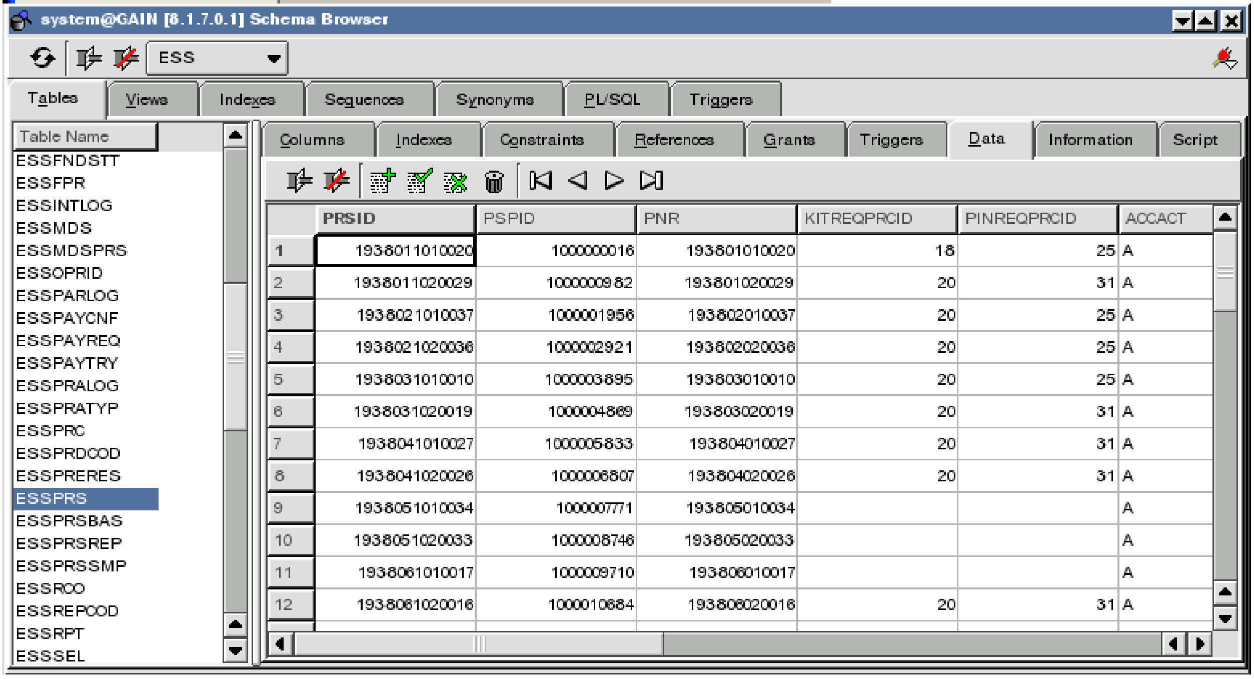
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

Primary key

SQL Developer’s applications



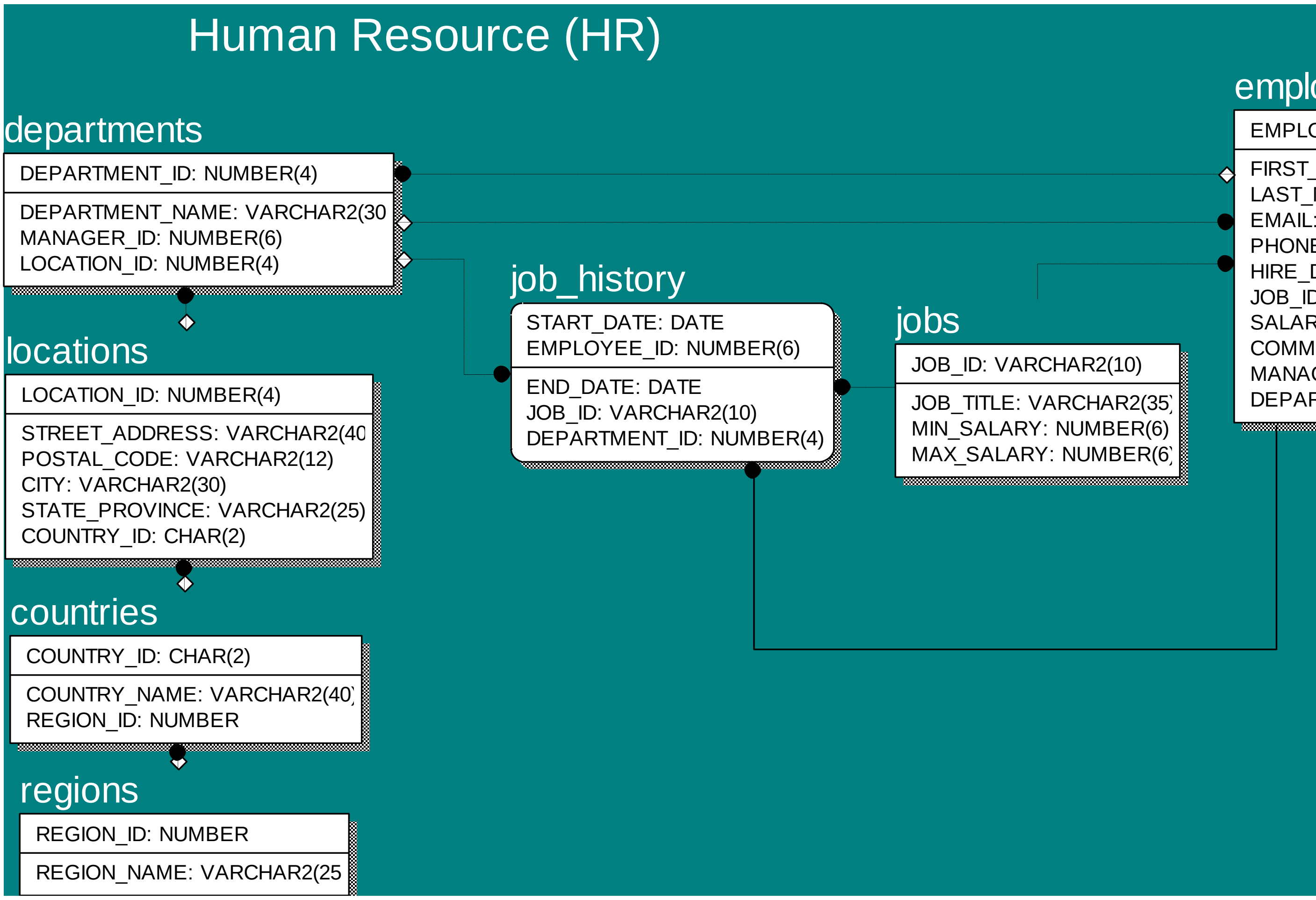
- PL/SQL Developer
- TO ADD
- TORA
- SQL Query Analyzer



	PRSID	PSPID	PNR	KITREQPRCID	PINREQPRCID	ACCACT
1	1938011010020	1000000016	193801010020	18	25	A
2	1938011020029	1000000982	193801020029	20	31	A
3	1938021010037	1000001956	193802010037	20	25	A
4	1938021020036	1000002921	193802020036	20	25	A
5	1938031010010	1000003895	193803010010	20	25	A
6	1938031020019	1000004869	193803020019	20	31	A
7	1938041010027	1000005833	193804010027	20	31	A
8	1938041020026	1000006807	193804020026	20	31	A
9	1938051010034	1000007771	193805010034			A
10	1938051020033	1000008746	193805020033			A
11	1938061010017	1000009710	193806010017			A
12	1938061020016	1000010684	193806020016	20	31	A

- TORA

Physical Entity Relationship for Homan Resource (HR)



Data manipulate language SQL
(Structured Query Language)

What is SQL?

SQL (pronounced "ess-que-el") stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database. This tutorial will provide you with the instruction on the basics of each of these commands as well as allow you to put them to practice using the SQL Interpreter.

- **Select**
- **Insert**

- **Update**
- **Delete**
- **Commit**
- **Rollback**

Note: *All next examples have to be executed in schema HR*

Select

The **select** statement is used to query the database and retrieve selected data that match the criteria that you specify. Here is the format of a simple select statement:

```
SELECT  * or field names
FROM    table names
WHERE   logical conditions
```

The column names that follow the select keyword determine which columns will be returned in the results. You can select as many column names that you'd like, or you can use a "*" to select all columns.

The table name that follows the keyword **from** specifies the table that will be queried to retrieve the desired results.

The **where** clause (optional) specifies which data values or rows will be returned or displayed, based on the criteria described after the keyword **where**.

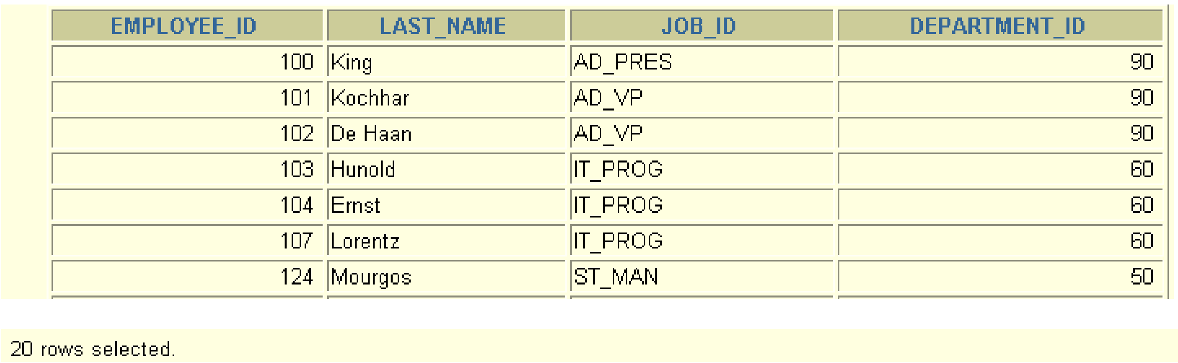
Conditional selections used in the **where** clause:

=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal to; or !=
LIKE	*See note below

Example: Find all employees in “90” department.

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees
WHERE  department_id = 90;
```

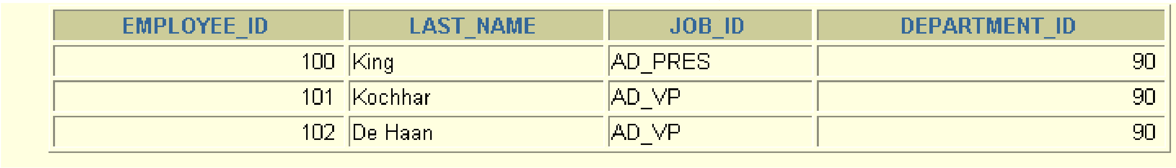
EMPLOYEES



EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
104	Ernst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50

20 rows selected.

“Find all employees In 90 department”

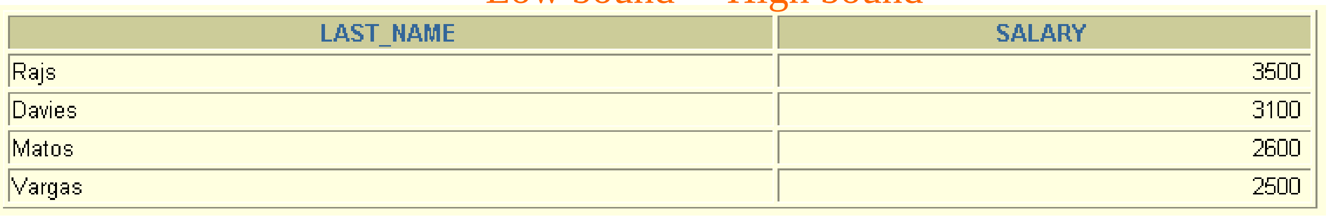


EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

Example: Find all employees, whose salary is between 2500 and 3500.
BETWEEN is equivalent to SALARY>=2500 AND SALARY <=3500

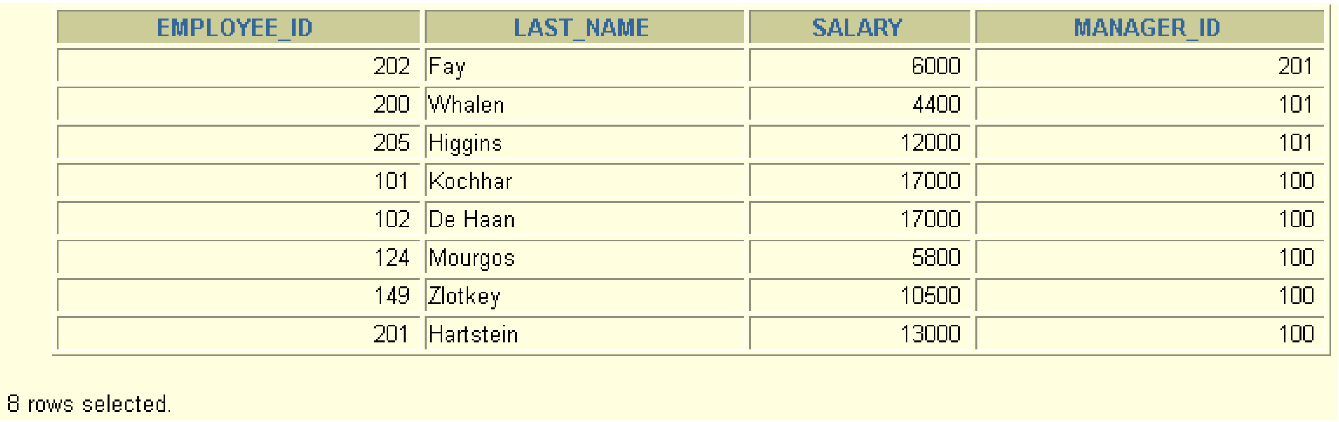
```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN  2500 AND 3500;
```

Low bound   High bound



LAST_NAME	SALARY
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201);
```

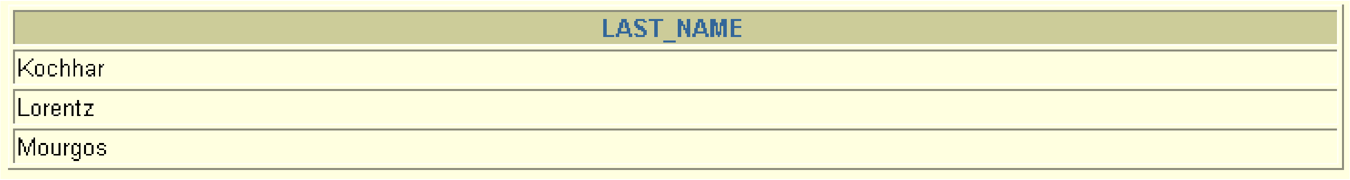


EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.

The **LIKE** pattern matching operator can also be used in the conditional selection of the where clause. Like is a very powerful operator that allows you to select only rows that are "like" what you specify. The percent sign "%" can be used as a wild card to match any possible character that might appear before or after the characters specified. Underscore "_" shows symbol position in pattern.
For example: Find employees last names where on the second position is letter "o".

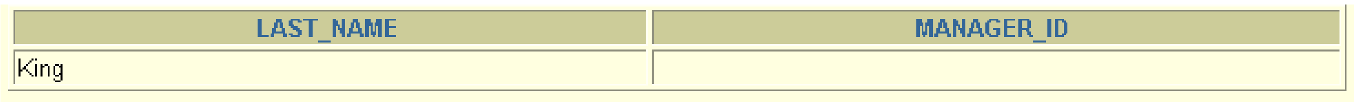
```
SELECT last_name
FROM   employees
WHERE  last_name LIKE '_o%';
```



LAST_NAME
Kochhar
Lorentz
Mourgos

NULL is record field's value where nothing is written.
Example: Find employees which have no manager.

```
SELECT last_name, manager_id
FROM   employees
WHERE  manager_id IS NULL;
```

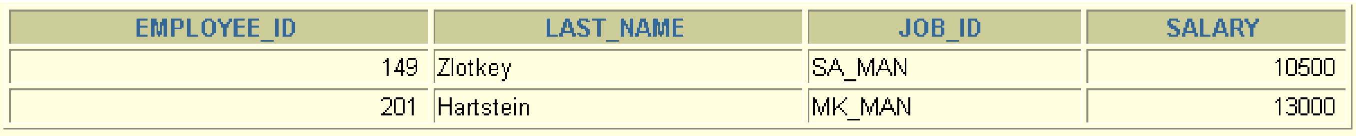


LAST_NAME	MANAGER_ID
King	

Logical operations

Operations	Description
AND	**Is true when both operands are true.**
OR	**Is true when one of the operands are true.**
NOT	**Is true when operand is false and the reverse.**
LIKE	**Is pattern constructing with the help of :** **% - substring; _ - position.**

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >=10000
AND    job_id LIKE '%MAN%';
```

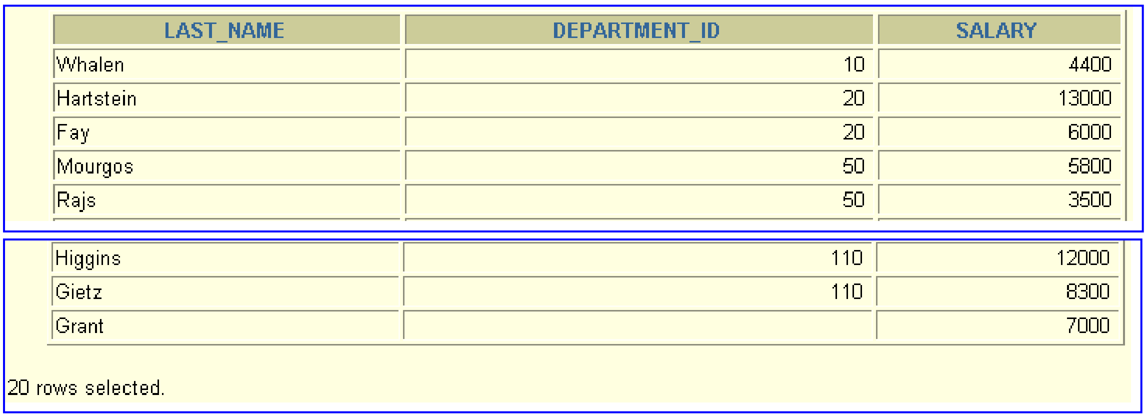


EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
149	Zlotkey	SA_MAN	10500
201	Hartstein	MK_MAN	13000

Order by

```
SELECT last_name, department_id, salary
FROM   employees
ORDER BY department_id, salary DESC;
```

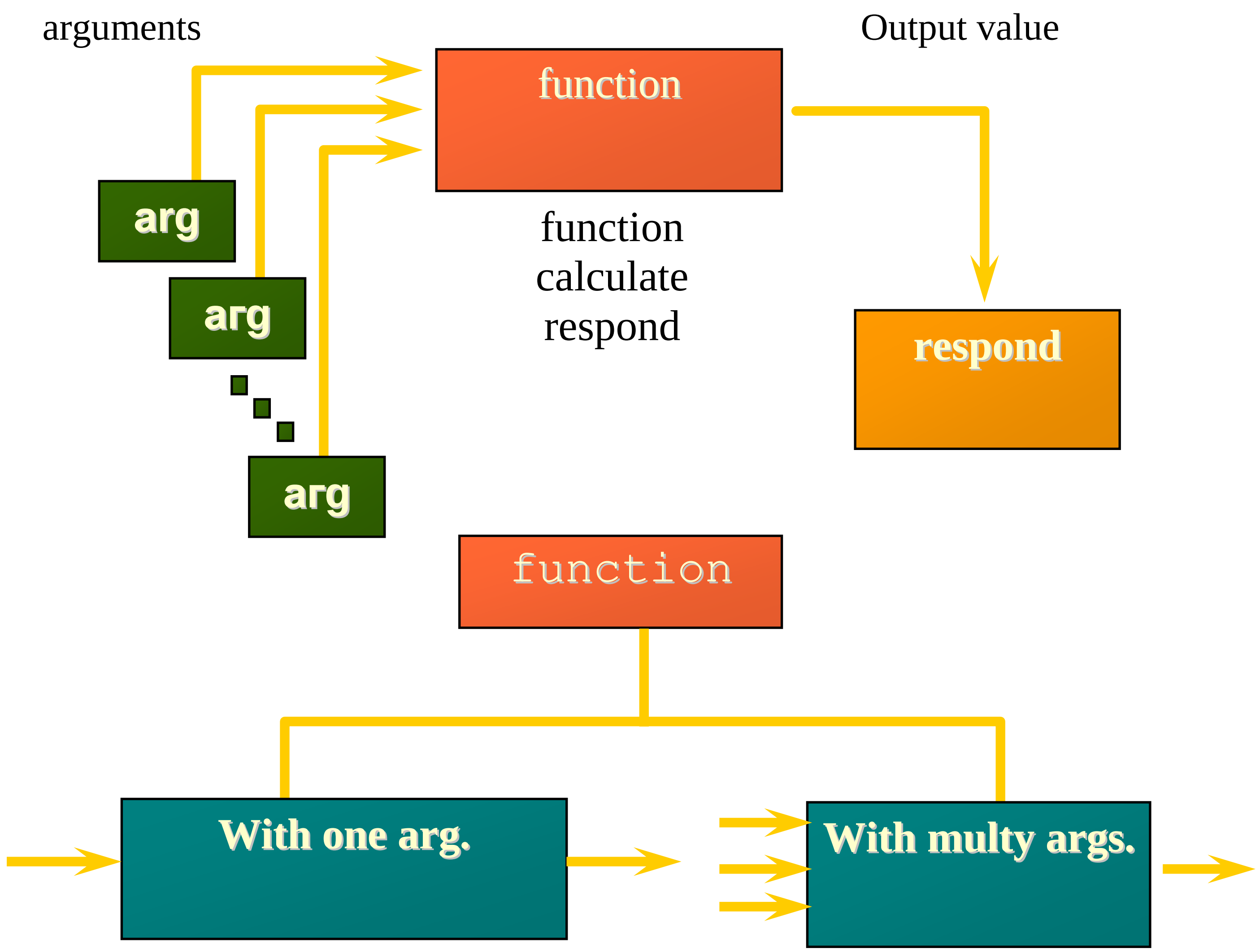
- **ASC**
- **DESC**



LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3500
Higgins	110	12000
Gietz	110	8300
Grant		7000

20 rows selected.

Funcions



arguments

Output value

function

arg

arg

arg

function
calculate
respond

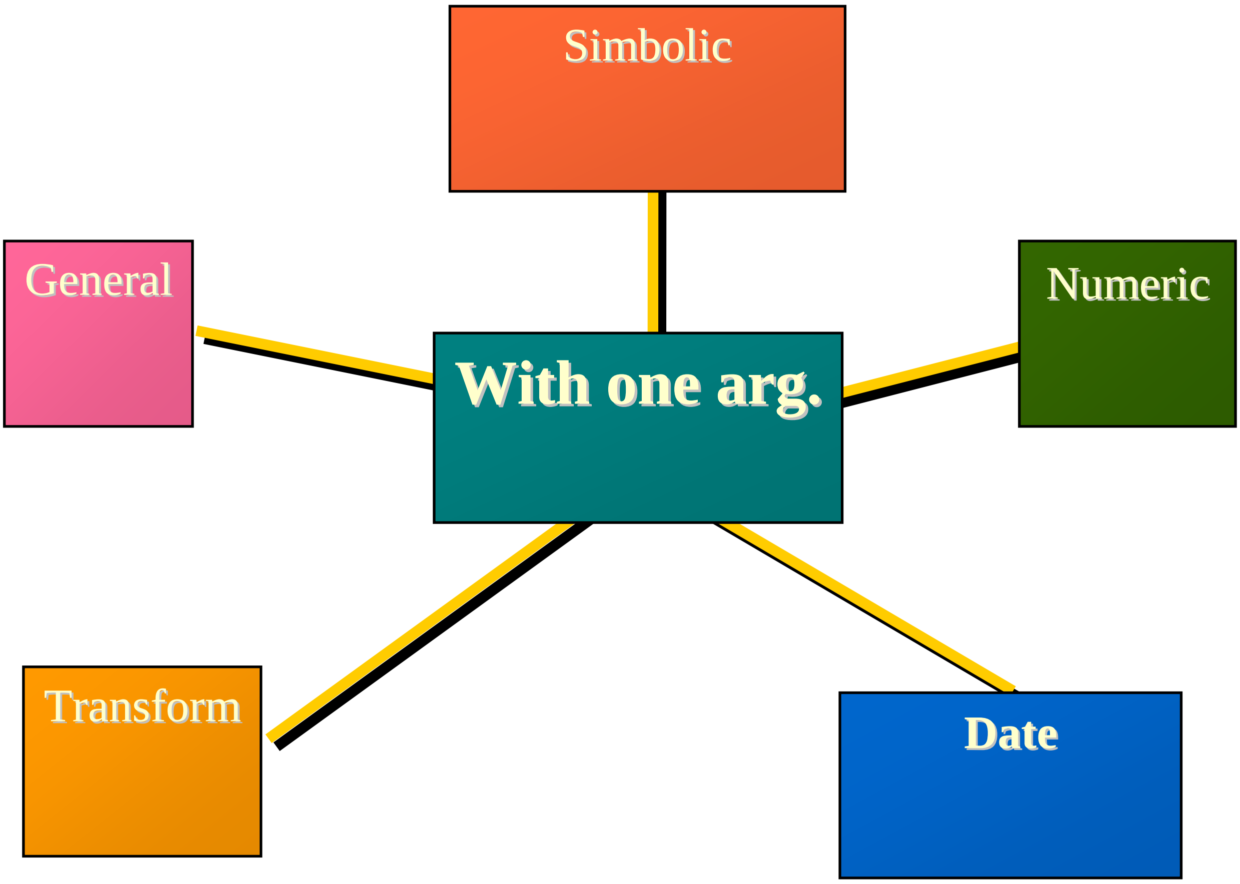
respond

function

With one arg.

With multy args.

Functions with one argument.



With one arg.

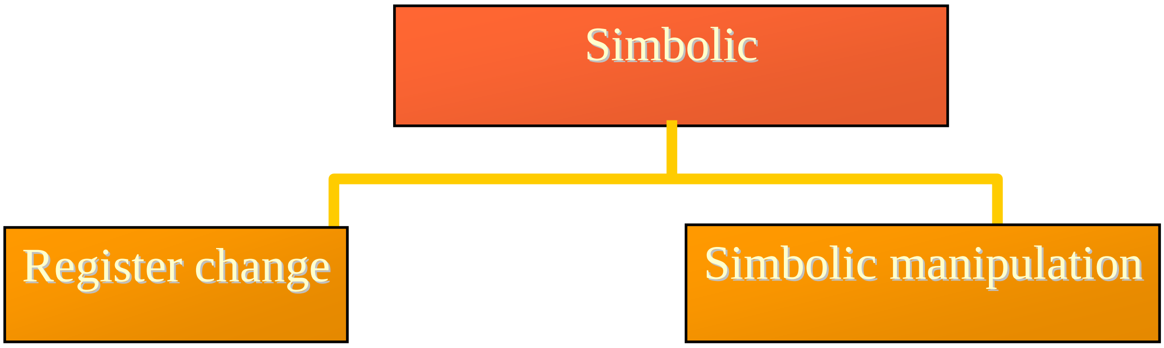
Simbolic

General

Numeric

Transform

Date



Simbolic

Register change

Simbolic manipulation

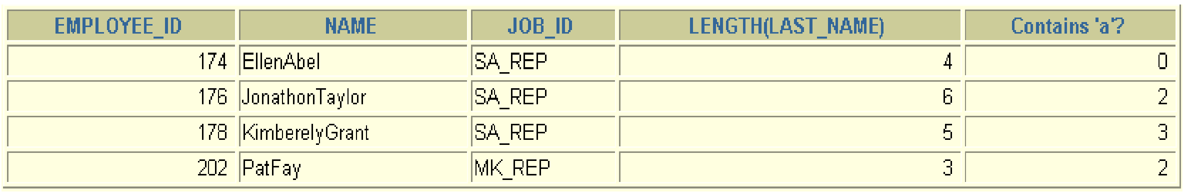
LOWER
UPPER
INITCA

CONCAT
SUBSTR
LENGTH
INSTR
LPAD | RPAD
TRIM
REPLACE

function	response
LOWER('SQLCourse')	sqlcourse
UPPER('SQLCourse')	SQLCOURSE
INITCAP('SQLCourse')	Sqlcourse

function	response
CONCAT('Hello', 'World')	HelloWorld
SUBSTR('HelloWorld',1,5)	Hello
LENGTH('HelloWorld')	10
INSTR('HelloWorld', 'W')	6
LPAD(salary,10,'*')	*****24000
RPAD(salary, 10, '*')	24000*****
TRIM('H' FROM 'HelloWorld')	elloWorld

```
SELECT employee_id, CONCAT(first_name, last_name) NAME, job_id, LENGTH
(last_name), INSTR(last_name, 'a') "Contains'a'?"
FROM   employees
WHERE  SUBSTR(job_id, 4) = 'REP';
```



EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

function	description
MONTHS_BETWEEN	Number of months between two dates.
ADD_MONTHS	Adds months to given date.
NEXT_DAY	Nearest date to given day.
LAST_DAY	The last date to given date.
ROUND	Date rounding.
TRUNC	Truncate date.
SYSDATE	The current age,year,month,day,hour,minute,secunda

MONTHS_BETWEEN('01-SEP-95','11-JAN-94')	→	19.6774194
ADD_MONTHS ('11-JAN-94',6)	→	'11-JUL-94'
NEXT_DAY ('01-SEP-95','FRIDAY')	→	'08-SEP-95'
LAST_DAY('01-FEB-95')	→	'28-FEB-95'
ROUND(SYSDATE,'MONTH')	→	'01-AUG-95'

Number Functions

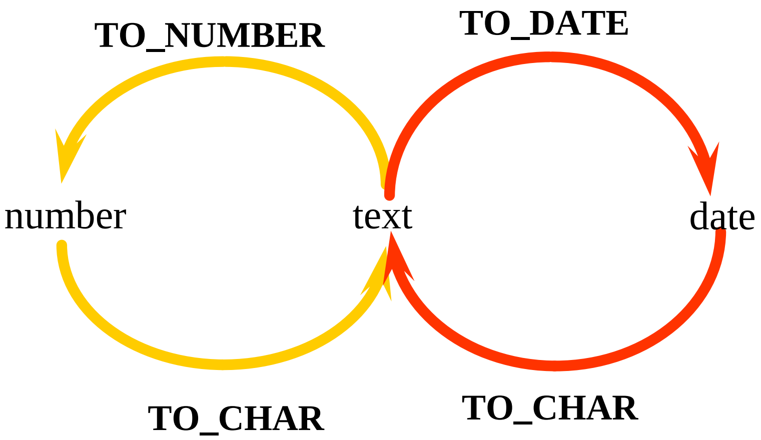
function	description
ROUND	Number rounding.
FLOOR	The largest integer value that is equal to or less than a number.
CEIL	The smallest integer value that is greater than or equal to a *number*.

ROUND(19.6774,2)	→	19.68
ROUND(19.6774,-1)	→	20
ROUND(19.6774,0)	→	20
FLOOR(19.6774)	→	19
CEIL(19.6774)	→	20

Example: Get Persons Age

```
SELECT FLOOR( MONTHS_BETWEEN(SYSDATE, ’01-SEP-95’) / 12 ) AGE FROM dual
```

Convert Date and Number



Transform

VARCHAR → NUMBER
NUMBER → VARCHAR

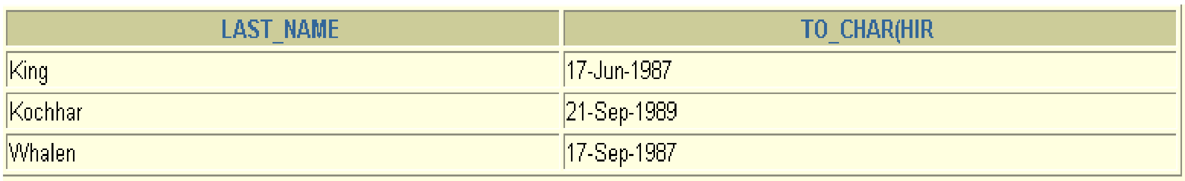
Date format

‘dd/mm/yyyy hh24:mi:ss’

‘dd.mm.yyyy hh24:mi:ss’

‘dd – mon – yyyy’

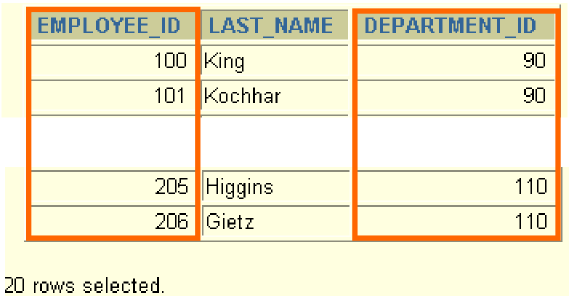
```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM   employees
WHERE  hire_date < TO_DATE('01/01/1990', 'DD/MM/YYYY');
```



LAST_NAME	TO_CHAR(HIR
King	17-Jun-1987
Kochhar	21-Sep-1989
Whalen	17-Sep-1987

Select from several tables

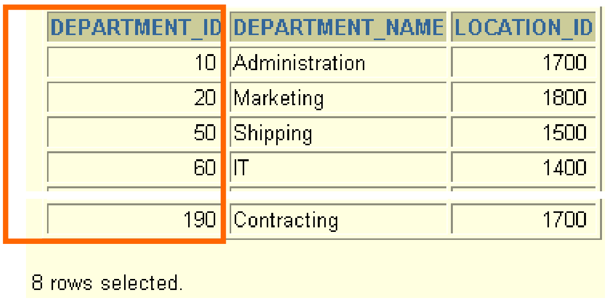
EMPLOYEES



EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
205	Higgins	110
206	Gietz	110

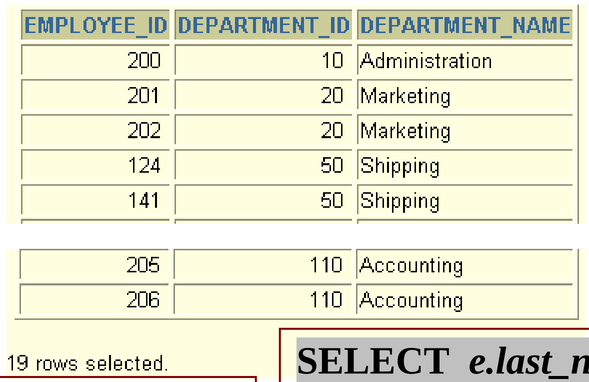
20 rows selected.

DEPARTMENTS



DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
190	Contracting	1700

8 rows selected.



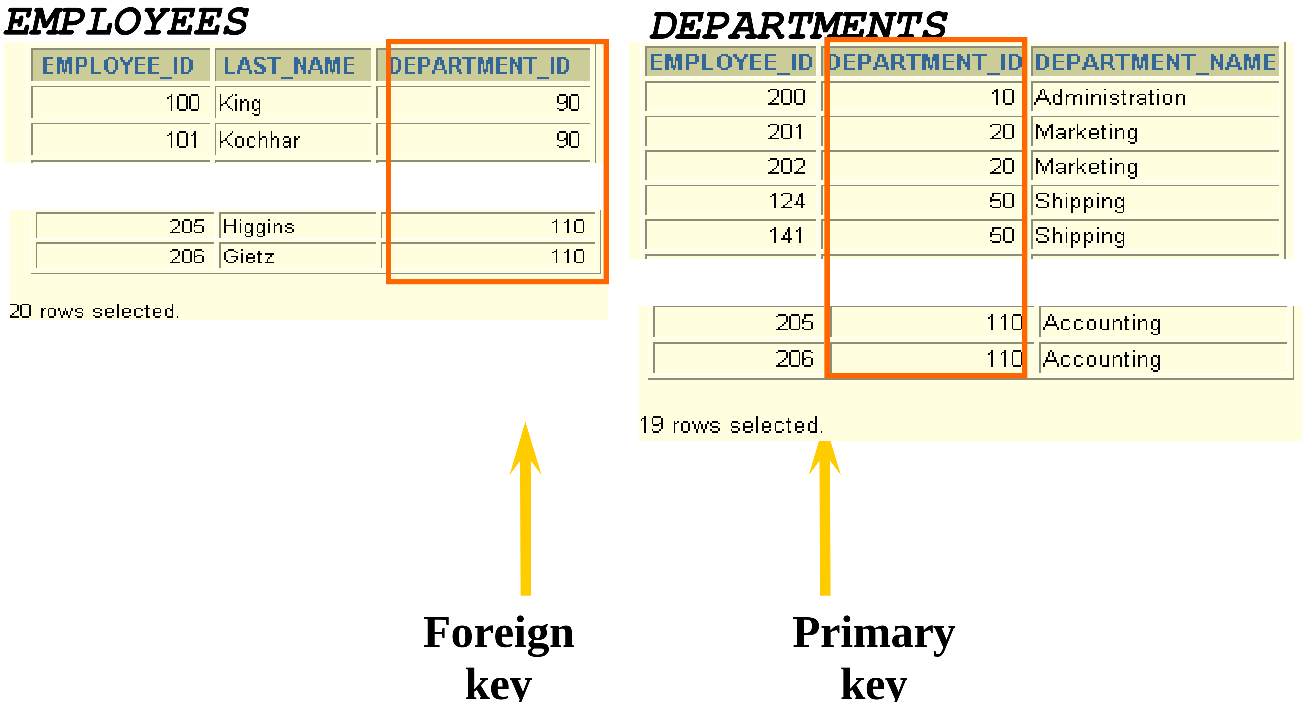
EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
124	50	Shipping
141	50	Shipping
205	110	Accounting
206	110	Accounting

19 rows selected.

```
SELECT    t.col1, t.col2, r.col1, r.col3
FROM      tab1 t, tab2 r
WHERE     t.col1= r.col2;
```

```
SELECT  e.last_name,
        d.department_name
FROM    employees e, departments d
WHERE
  e.department_id = d.department_id;
```

What is Joins (equiconnection)?



EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
124	50	Shipping
141	50	Shipping
205	110	Accounting
206	110	Accounting

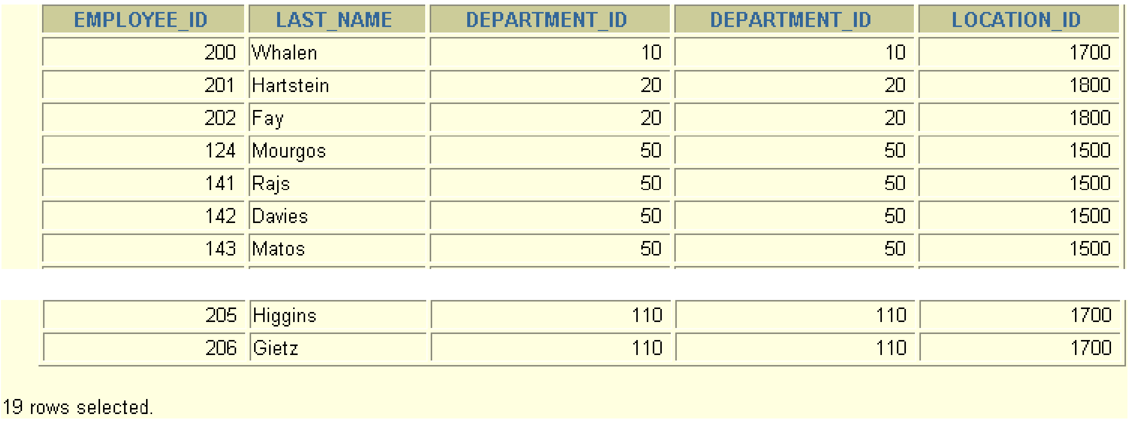
19 rows selected.

Foreign key

Primary key

Select with the help of join

```
SELECT e.employee_id, e.last_name,   e.department_id, d.department_id,     d.location_id
FROM   employees e, departments d
WHERE  e.department_id = d.department_id;
```



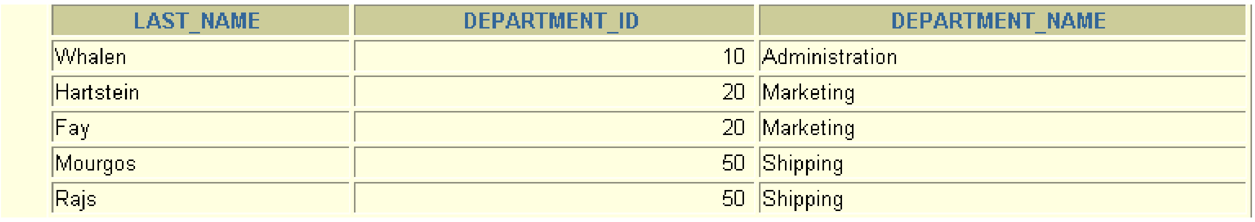
EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
205	Higgins	110	110	1700
206	Gietz	110	110	1700

19 rows selected.

SQL Inner (simple) Joins

```
SELECT   t.col1, t.col2, r.col1, r.col3
FROM     tab1 t JOIN  tab2 r
ON       t.col1= r.col2;
```

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e JOIN departments d
ON e.department_id = d.department_id;
```



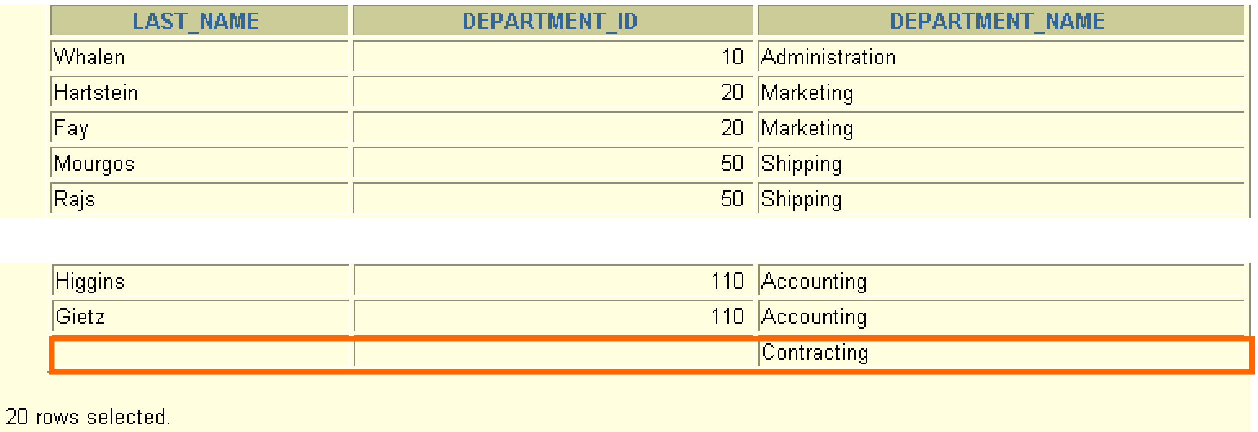
LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping

Oracle Outer Joins

```
SELECT   t.col1, t.col2, r.col1, r.col3
FROM     tab1 t, tab2 r
WHERE    t.col1(+)= r.col2;
```

```
SELECT   t.col1, t.col2, r.col1, r.col3
FROM     tab1 t, tab2 r
WHERE    t.col1= r.col2(+);
```

```
SELECT e.last_name, e.department_id, d.department_name
FROM employees e, departments d
WHERE e.department_id(+) = d.department_id;
```



LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

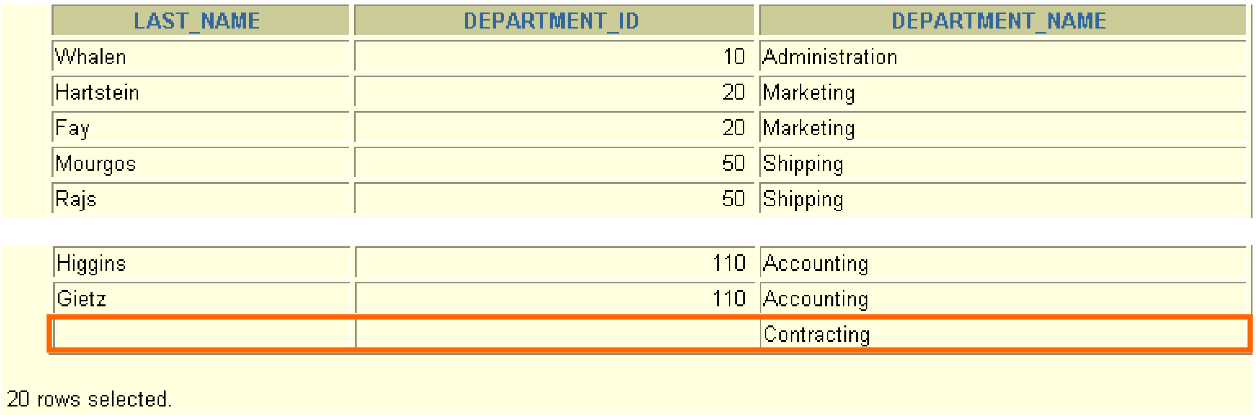
SQL Outer Joins

```
SELECT      t.col1, t.col2, r.col1, r.col3
FROM        tab1t LEFT OUTER JOIN  tab2 r
ON          t.col1= r.col2;
```

```
SELECT      t.col1, t.col2, r.col1, r.col3
FROM        tab1t RIGHT OUTER JOIN  tab2 r
ON          t.col1= r.col2;
```

```
SELECT      t.col1, t.col2, r.col1, r.col3
FROM        tab1t FULL OUTER JOIN  tab2 r
ON          t.col1= r.col2;
```

```
SELECT e.last_name, e.department_id, d.department_name
FROM    employees e FULL JOIN departments d
ON        e.department_id = d.department_id;
```



LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

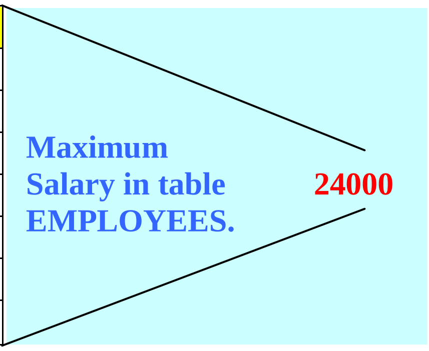
20 rows selected.

Group Functions

count
sum
max
min
avg

EMPLOYEES

Department_ID	Salary
50	2600
50	2600
10	4400
20	13000
20	6000
90	24000
70	10000


Maximum Salary in table EMPLOYEES.

24000

```
SELECT max ( e.salary )
FROM employees e;
```

Result: 24000

```
SELECT count ( * )
FROM employees e;
```

Result: 20

Group functions

Syntaxes of group functions

```
SELECT        [colon,]
group_function(colon), ...
FROM                    table
[WHERE        condition]
[GROUP BY   colon]
[ORDER BY   colon];
```

```
SELECT department_id, AVG(salary)
FROM   employees
GROUP BY department_id;
```

department_id	salary	avg
10	4400	4400
20	13000	9500
20	6000	
50	5800	
50	3500	
50	3100	3500
50	2500	
50	2600	
60	9000	
60	6000	6400
60	4200	
80	10500	
110	8300	

Average salary in table EMPLOYEES by every department

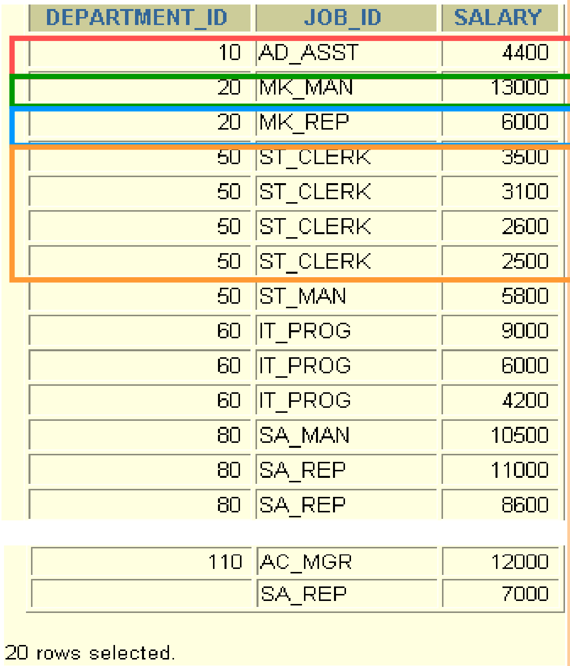
department_id	avgSal
10	4400
20	9500
50	3500
60	6400
80	10033

Group by several colons

v Summarize salaries in table EMPLOYEES by every job in every department

```
SELECT   department_id dept_id, job_id,
                  SUM(salary)
FROM     employees
GROUP BY department_id, job_id;
```

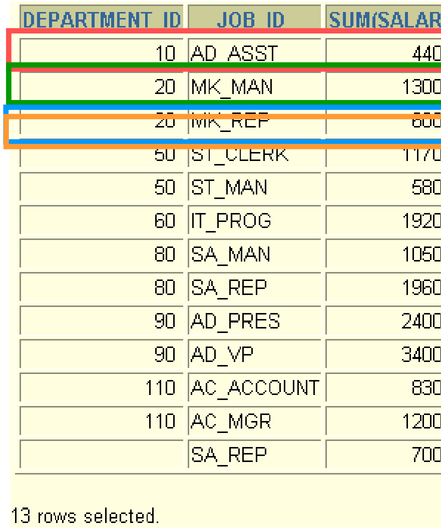
EMPLOYEES



DEPARTMENT_ID	JOB_ID	SALARY
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
50	ST_MAN	5800
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600
110	AC_MGR	12000
	SA_REP	7000

20 rows selected.

Summarize salaries in table EMPLOYEES by every job in every department



DEPARTMENT_ID	JOB_ID	SUM(SALAR
10	AD_ASST	440
20	MK_MAN	1300
20	MK_REP	600
50	ST_CLERK	1170
50	ST_MAN	580
60	IT_PROG	1920
80	SA_MAN	1050
80	SA_REP	1960
90	AD_PRES	2400
90	AD_VP	3400
110	AC_ACCOUNT	830
110	AC_MGR	1200
	SA_REP	700

13 rows selected.

Group excluding

```
SELECT department_id, AVG(salary)
FROM   employees
WHERE  AVG(salary) > 8000
GROUP BY department_id;
```

```
WHERE  AVG(salary) > 8000
       *
ERROR at line 3:
ORA-00934: group function is not allowed
```

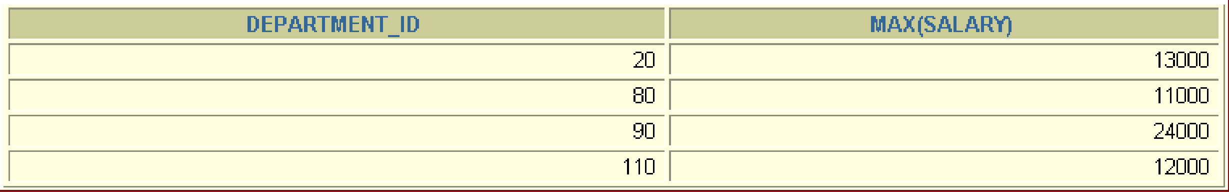
In group functions WHERE to exclude group

Is not allowed

SELECT [*colon*,] *group_function(colon)*, ...
FROM *table*
[WHERE *condition*]
[GROUP BY *colon*]
[HAVING exclusive condition]
[ORDER BY *colon*];

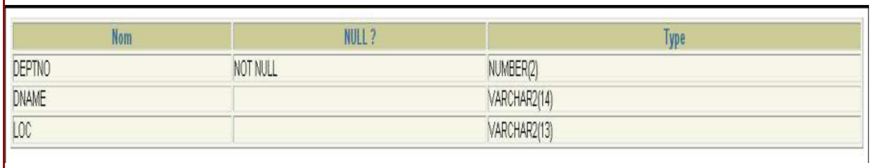
To exclude group you have to use **HAVING**

```
SELECT   department_id, MAX(salary)
FROM     employees
GROUP BY department_id
HAVING   MAX(salary)>10000;
```



DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

Insert data into table



Nom	NULL ?	Type
DEPTNO	NOT NULL	NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

INSERT INTO *table [(column[, column...])]*
VALUES *(value [, value...]);*

```
SQL> INSERT INTO dept(deptno,dname,loc)
       VALUES (90,'Sciences','Paris');
1 row created.

INSERT INTO dept(deptno,dname)
 VALUES (5,'Implicit');
1 row created.

INSERT INTO dept(deptno,dname,loc)
 VALUES (6,'Explicit',NULL);
1 row created.

INSERT INTO emp(empno,ename,hiredate)
VALUES (1,'MIKE',TO_DATE('FEV 3, 1999','MON DD,YYYY'));
1 row(s) created.
```

Inserting new rows with null values
There are two methods for inserting null values into a table:
Method Description
Implicit Omit the column from the column list.
Explicit Specify the NULL keyword in the values list,
Specify the empty string (‘ ‘) in the **VALUES** list for character strings and dates.
You must be sure that the column in which you want to insert null values don’t have the NOT NULL constraint, by issuing a **DESCRIBE** command and verifying the *NULL?* status of the column.

Update data in table

```
SQL> UPDATE emp
          SET sal=sal*1.20
WHERE ename='SMITH';
1 row updated.

SQL> UPDATE emp
SET sal = sal * 1.20,
       job = 'SALESMAN'
WHERE ename = 'SMITH';

1 row updated.
```

UPDATE *table*
SET *column = value [, column = value ...]*
[WHERE *condition];*

Delete data from table

```
SQL> DELETE FROM copy_emp;
14 row(s) deleted.

SQL> DELETE FROM copy_emp
 WHERE empno=7369;
1 row(s) deleted.
```

DELETE [FROM] *table*
[WHERE *condition***];**

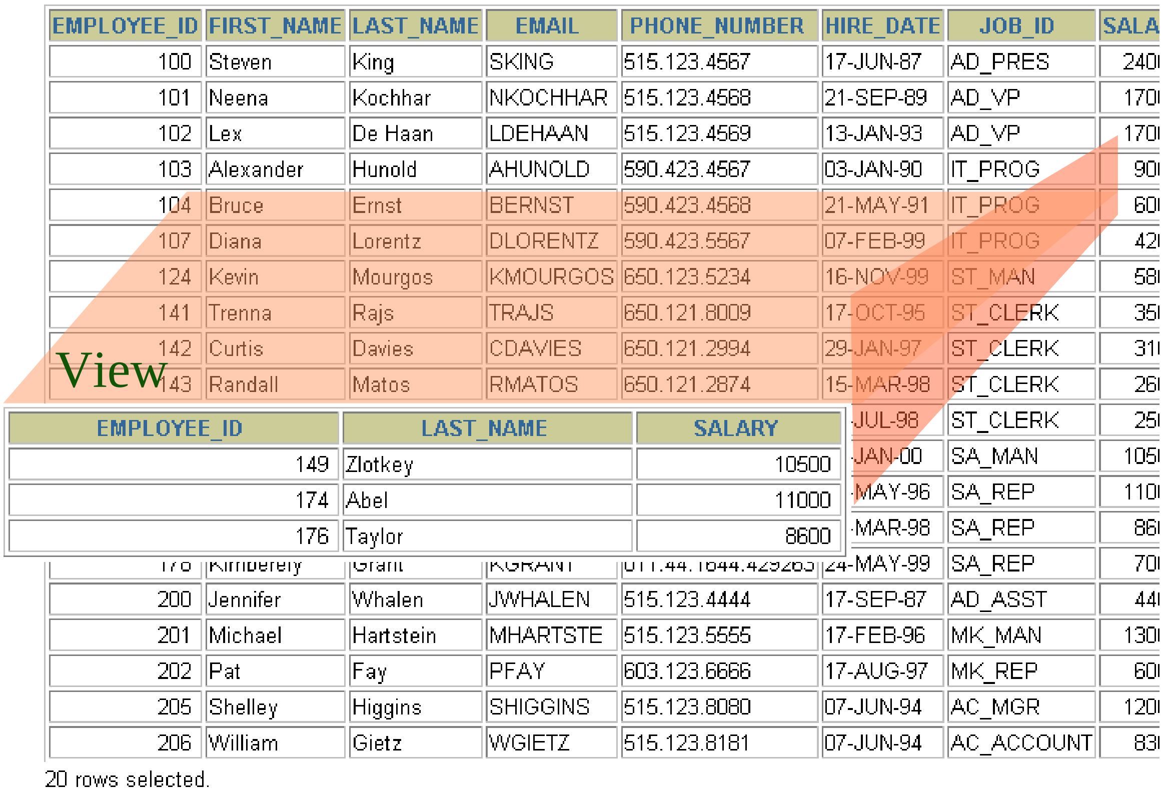
Commit;
Rollback;

COMMIT;   -- fixes changed data in DB
ROLLBACK; -- rollbacks to previous state

View objects

```
CREATE VIEW         empvu80
 AS SELECT  employee_id, last_name, salary
   FROM    employees
   WHERE   department_id = 80;
```

Table Employees

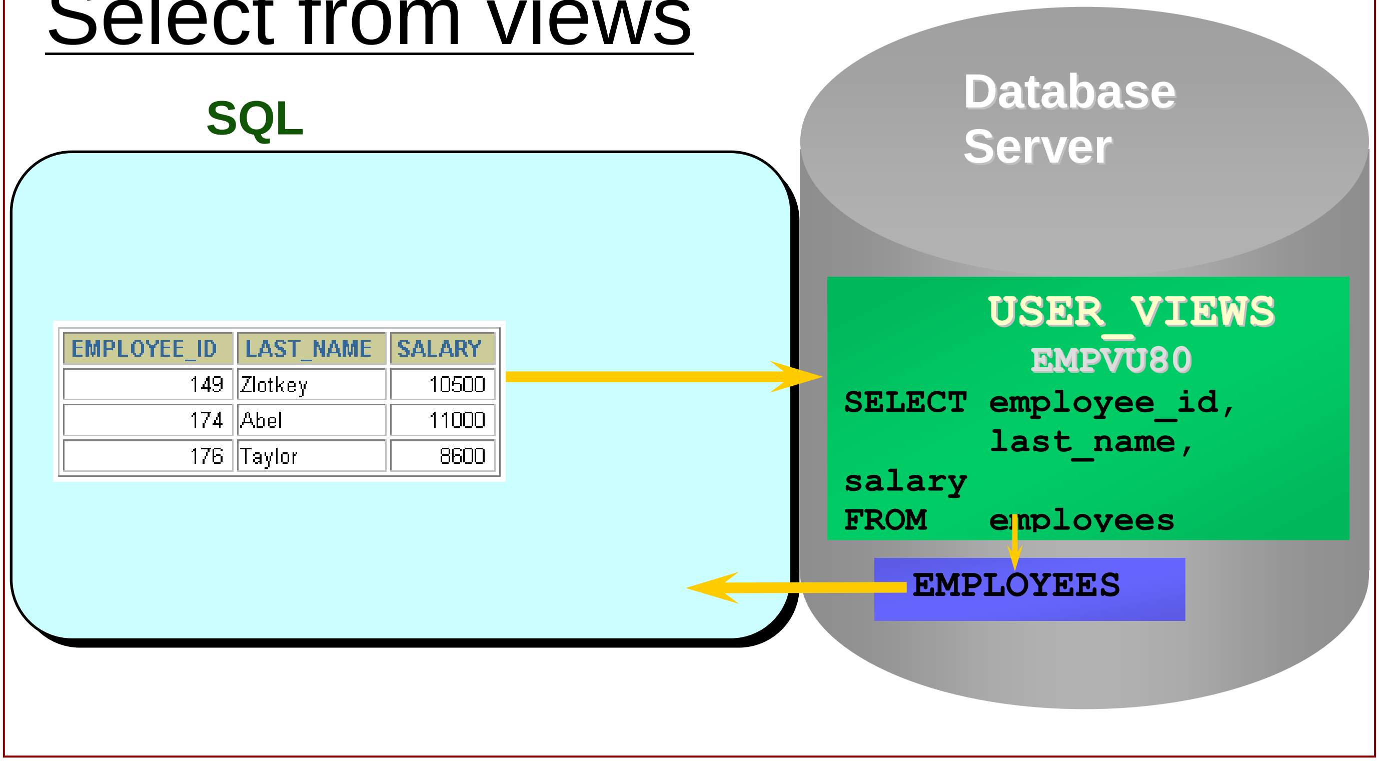


EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	26
					JUL-98	ST_CLERK	25
					JAN-00	SA_MAN	105
					MAY-96	SA_REP	110
					MAR-98	SA_REP	86
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	70
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	44
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	130
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	60
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	120
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	83

EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

20 rows selected.

Select from views



EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

```
SELECT employee_id,
       last_name,
salary
FROM   employees
```

Some creates of view

View from one table

```
CREATE OR REPLACE VIEW empvu80  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' ' || last_name,    salary, department_id
  FROM    employees
  WHERE   department_id = 80;
View created.
```

View from two tables

```
CREATE VIEW  dept_sum_vu  (name, minsal, maxsal, avgsal) AS
SELECT     d.department_name, MIN(e.salary), MAX(e.salary),AVG(e.salary)
  FROM      employees e, departments d
  WHERE     e.department_id = d.department_id
  GROUP BY  d.department_name;
View created.
```

The CREATE TABLE statement

First of all, you must have the **CREATE TABLE** privilege to use this statement, and a storage area to create objects.
The **CREATE TABLE** statement creates tables to store data. It's a DDL statement.
This statement, such as all the DDL statements, has an immediate effect on the database.
Syntax:
CREATE TABLE *[schema].table*
*(Column datatype [DEFAULT expr], [...]);**
SCHEMA: Is the schema on which you want to create the table.
TABLE: Is the name of the table.
DEFAULT expr: Specifies a default value, if a value is omitted in the **INSERT** statement.
COLUMN: Is the name of the column.
DATATYPE: Is the datatype of the column. (You must specify its length).
This SQL statement creates the EMP2 table, with three columns: EMPNO, ENAME and LOCATION.

```
CREATE   TABLE   EMP2  (
EMPNO   NUMBER(2),
ENAME   VARCHAR2(50),
LOCATION   VARCHAR2(50) DEFAULT 'Not specified');
Table created.
```

DROP TABLE *tablename;   Ex.: Drop table EMP2;*

Oracle Data types

Data type Description

VARCHAR2(*size*) Variable-length character data(a maximum size must be specified: Min= 1,Max= 4000)

CHAR[(*size*)] Fixed-length character data of length *size* bytes. (a maximum size can be specified: Min= 1, Max= 2000)

NUMBER [(*p*,*s*)] Number having precision *p* and scale *s*.(The precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point; the precision can range from 1 to 38 and the scale can range from -84 to 127).

DATE Date and Time values to the nearest second between January 1,4712 B.C., and December 31,9999 A.D.

LONG Variable-length character up to 2 gigabytes.

CLOB Character data up to 4 gigabytes.

RAW(*size*) Raw binary data of length *size*. (a max size must be specified . max=2000).

LONG RAW Raw binary data of variable length up to 2 gigabytes.

BLOB Binary data up to 4 gigabytes.

BFILE Binary data, stored in an external file; up to 4 gigabytes.

ROWID A 64 base number system representing the unique address of a row in its table.

MySQL [MS SQL] Data types

INTEGER [(length)] [UNSIGNED] [ZEROFILL] 4 byte integer

TINYINT [(length)] [UNSIGNED] [ZEROFILL] 1 byte integer

SMALLINT [(length)] [UNSIGNED] [ZEROFILL] 2 byte integer

MEDIUMINT [(length)] [UNSIGNED] [ZEROFILL] 3 byte integer

BIGINT [(length)] [UNSIGNED] [ZEROFILL] 8 byte integer (if compiler supports longlong)

REAL [(length,dec)] float (4 bytes)

FLOAT [(length,dec)] float (4 bytes)

DOUBLE [(length,dec)] double (4 or 8 bytes) A packed floating point number.

DECIMAL (length,dec) An unpacked floating point number.

CHAR(NUM) Fixed width string (1 <= NUM <= 255)

VARCHAR(NUM) Variable length string (1 <= NUM <= 255)

TINYBLOB Binary object with a maximum length of 255 BLOB Binary object with a maximum length of 65535

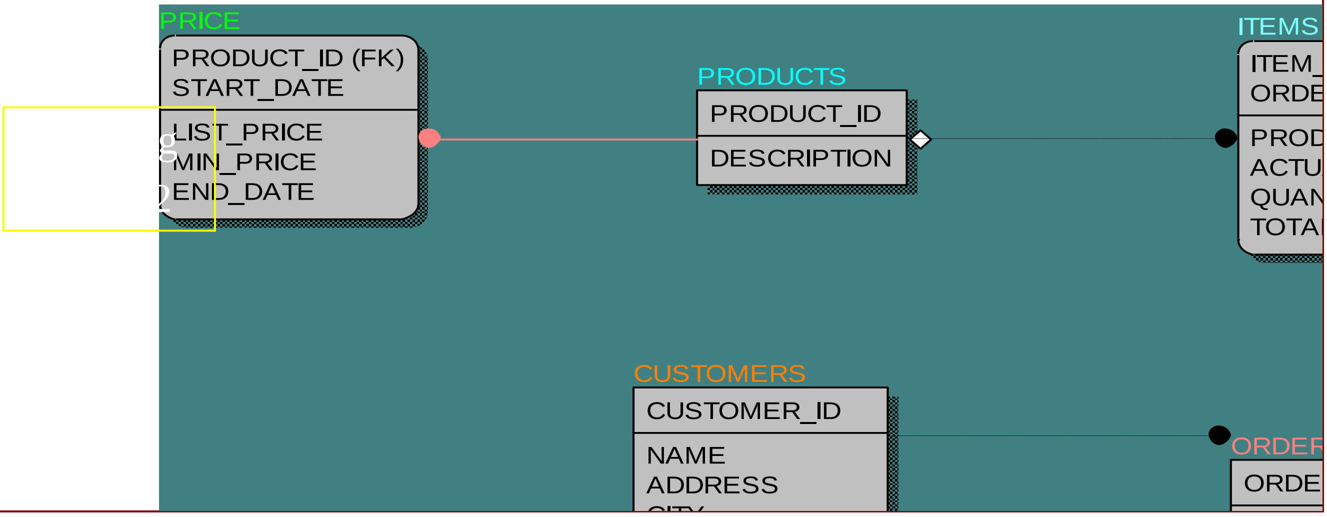
MEDIUMBLOB Binary object with a maximum length of 16777216

LONGBLOB Binary object with a maximum length of 2**32

TIMESTAMP(NUM) Changes automatically on insert/update (YYMMDDHHMMSS) The length determines how the output is formatted.

Shopping Database Structure

(logical performance)



PRICE

PRODUCT_ID (FK)
START_DATE

LIST_PRICE
MIN_PRICE
END_DATE

PRODUCTS

PRODUCT_ID

DESCRIPTION

ITEMS

ITEM_
ORDE

PROD
ACTU
QUAN
TOTA

CUSTOMERS

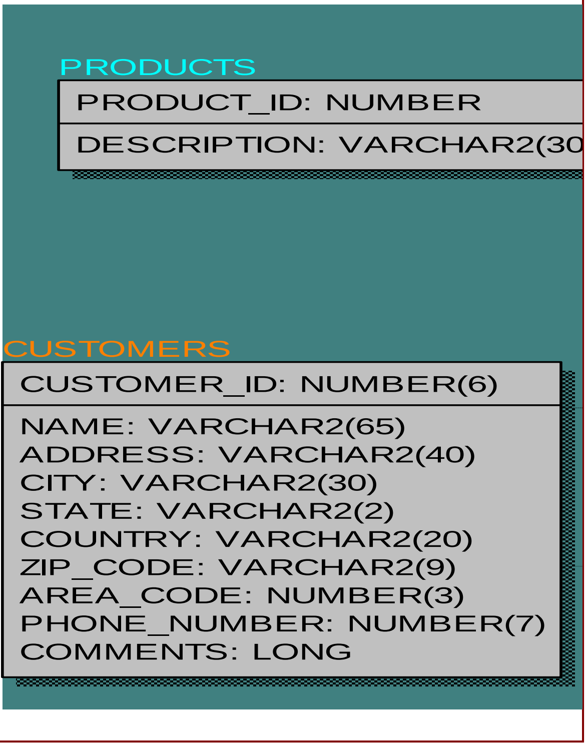
CUSTOMER_ID

NAME
ADDRESS

ORDER

ORDE

(physical performance)



PRODUCTS

PRODUCT_ID: NUMBER

DESCRIPTION: VARCHAR2(30

CUSTOMERS

CUSTOMER_ID: NUMBER(6)

NAME: VARCHAR2(65)
ADDRESS: VARCHAR2(40)
CITY: VARCHAR2(30)
STATE: VARCHAR2(2)
COUNTRY: VARCHAR2(20)
ZIP_CODE: VARCHAR2(9)
AREA_CODE: NUMBER(3)
PHONE_NUMBER: NUMBER(7)
COMMENTS: LONG

Create Shopping schema (oracle version)

```
CREATE   TABLE   PRODUCTS (
     DESCRIPTION             VARCHAR2(30)   NULL,
     PRODUCT_ID              NUMBER  NOT   NULL,
     CONSTRAINT XPKPRODUCTS
          PRIMARY KEY (PRODUCT_ID)
);
CREATE   TABLE   PRICE (
     LIST_PRICE           NUMBER(8,2)   NULL,
     PRODUCT_ID           NUMBER NOT   NULL,
     MIN_PRICE            NUMBER(8,2)   NULL,
     START_DATE           DATE   NOT   NULL,
     END_DATE             DATE   NULL,
     CONSTRAINT XPKPRICE
          PRIMARY KEY (PRODUCT_ID, START_DATE)
);
CREATE   TABLE   CUSTOMERS (
     NAME                VARCHAR2(65)   NULL,
     ADDRESS             VARCHAR2(40)   NULL,
     CITY                VARCHAR2(30)   NULL,
     STATE               VARCHAR2(2)   NULL,
     COUNTRY             VARCHAR2(20)   NULL,
     ZIP_CODE            VARCHAR2(9)   NULL,
     AREA_CODE           NUMBER(3)   NULL,
     PHONE_NUMBER        NUMBER(7)   NULL,
     COMMENTS            LONG   NULL,
     CUSTOMER_ID         NUMBER(6)   NOT   NULL,
     CONSTRAINT XPKCUSTOMERS
          PRIMARY KEY (CUSTOMER_ID)
);
```

```
CREATE   TABLE   ORDERS (
     ORDER_DATE           DATE   NULL,
     CUSTOMER_ID          NUMBER(6)   NOT   NULL,
     SHIP_DATE            DATE   NULL,
     TOTAL                NUMBER(8,2)   NULL,
     ORDER_ID             NUMBER(4)   NOT   NULL,
     CONSTRAINT XPKORDERS
          PRIMARY KEY (ORDER_ID)
);
CREATE   TABLE   ITEMS (
     ITEM_ID              NUMBER(4)   NOT   NULL,
     PRODUCT_ID           NUMBER   NULL,
     ACTUAL_PRICE         NUMBER(8,2)   NULL,
     QUANTITY             NUMBER(8)   NULL,
     TOTAL                NUMBER(8,2)   NULL,
     ORDER_ID             NUMBER(4)   NOT   NULL,
     CONSTRAINT XPKITEMS
          PRIMARY KEY (ITEM_ID, ORDER_ID)
);
```

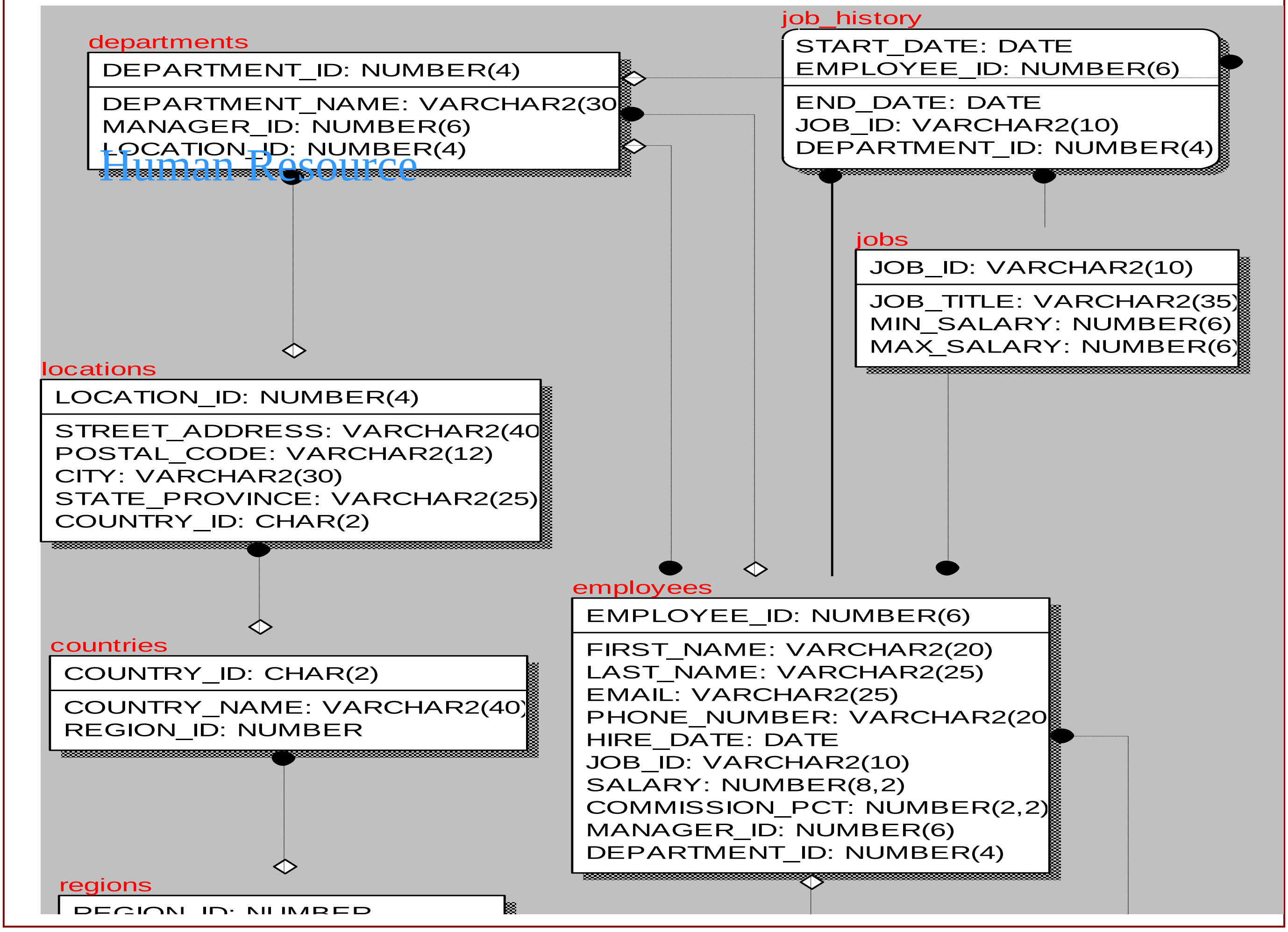
Create View

```
CREATE  OR  REPLACE   VIEW   ORDER_CUST AS
     SELECT CUSTOMERS.NAME,   CUSTOMERS.CITY,
                ORDERS.TOTAL,
ORDERS.ORDER_DATE
     FROM ORDERS,  CUSTOMERS;
```

Insert data into tables

See: MyShop.create_all_tables.rtf

Human Resource schema (HR) (physical performance)



departments

DEPARTMENT_ID: NUMBER(4)

DEPARTMENT_NAME: VARCHAR2(30
MANAGER_ID: NUMBER(6)
LOCATION_ID: NUMBER(4)

Human Resource

job_history

START_DATE: DATE
EMPLOYEE_ID: NUMBER(6)

END_DATE: DATE
JOB_ID: VARCHAR2(10)
DEPARTMENT_ID: NUMBER(4)

jobs

JOB_ID: VARCHAR2(10)

JOB_TITLE: VARCHAR2(35)
MIN_SALARY: NUMBER(6)
MAX_SALARY: NUMBER(6)

locations

LOCATION_ID: NUMBER(4)

STREET_ADDRESS: VARCHAR2(40
POSTAL_CODE: VARCHAR2(12)
CITY: VARCHAR2(30)
STATE_PROVINCE: VARCHAR2(25)
COUNTRY_ID: CHAR(2)

countries

COUNTRY_ID: CHAR(2)

COUNTRY_NAME: VARCHAR2(40)
REGION_ID: NUMBER

employees

EMPLOYEE_ID: NUMBER(6)

FIRST_NAME: VARCHAR2(20)
LAST_NAME: VARCHAR2(25)
EMAIL: VARCHAR2(25)
PHONE_NUMBER: VARCHAR2(20
HIRE_DATE: DATE
JOB_ID: VARCHAR2(10)
SALARY: NUMBER(8,2)
COMMISSION_PCT: NUMBER(2,2)
MANAGER_ID: NUMBER(6)
DEPARTMENT_ID: NUMBER(4)

regions

REGION_ID: NUMBER

DB management
PL/SQL programic language

- Block
- Procedures
- Functions
- Packages
- Triggers

Understanding the Main Features of PL/SQL

```
--  PL/SQL BLOCK

DECLARE
qty_on_hand NUMBER(5);

BEGIN

SELECT quantity INTO qty_on_hand FROM inventory
WHERE product = ’TENNIS RACKET’
FOR UPDATE OF quantity;

IF qty_on_hand > 0 THEN -- check quantity
  UPDATE inventory SET quantity = quantity - 1
  WHERE product = ’TENNIS RACKET’;

  INSERT INTO purchase_record
  VALUES (’Tennis racket purchased’, SYSDATE);
ELSE
  INSERT INTO purchase_record
  VALUES (’Out of tennis rackets’, SYSDATE);
END IF;

COMMIT;
END;
```

-- Block Structure

```
[DECLARE
-- declarations]
BEGIN
-- statements
[EXCEPTION
-- handlers]
END;
```

-- Declaring Variables

Variables can have any SQL datatype, such as CHAR, DATE, or NUMBER, or any PL/SQL datatype, such as BOOLEAN or BINARY_INTEGER. For example, assume that you want to declare a variable named part_no to hold 4-digit numbers and a variable named in_stock to hold the Boolean value TRUE or FALSE. You declare these variables as follows:

```
part_no   NUMBER(4);
in_stock   BOOLEAN;
Ship_date   DATE;
Item    I_TAB.ITEM.%TYPE;
```

The General Syntax to declare a variable and constant

variable_name datatype [NOT NULL := value]; *constant_name CONSTANT datatype := VALUE;*

- *variable_name* is the name of the variable.
- *datatype* is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable.
- *value* or DEFAULT *value*is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.
- Constant could not be changed in body.

For example: The below example declares two variables, one of which is a not null and constant.

```
DECLARE
salary number(4);
dept varchar2(10) NOT NULL := “HR Dept”;
```

```
DECLARE
salary_increase CONSTANT number (3) := 10;
```

Assigning Values to a Variable

You can assign values to a variable in three ways. The first way uses the assignment operator (:=), a colon followed by an equal sign. You place the variable to the left of the operator and an expression (which can include function calls) to the right. A few examples follow:

```
tax := price * tax_rate;
valid_id := FALSE;
bonus := current_salary * 0.10;
wages := gross_pay(emp_id, st_hrs, ot_hrs) - deductions;
```

The second way to assign values to a variable is by selecting (or fetching) database values into it. In the example below, you have Oracle compute a 10% bonus when you select the salary of an employee.

Now, you can use the variable bonus in another computation or insert its value into a database table.

```
SELECT sal * 0.10 INTO bonus FROM emp WHERE empno = emp_id;
```

The third way to assign values to a variable is by passing it as an OUT or IN OUT parameter to a subprogram. As the following example shows, an IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller:

```
DECLARE
my_sal REAL(7,2);
PROCEDURE adjust_salary (emp_id INT, salary IN OUT REAL) IS ...
BEGIN
SELECT AVG(sal) INTO my_sal FROM emp;
adjust_salary(7788, my_sal); -- assigns a new value to my_sal
```

Datatypes Built-in Oracle PL/SQL

Scalar Types

BINARY_INTEGER
DEC
DECIMAL
DOUBLE
PRECISION
FLOAT
INT
INTEGER
NATURAL
NATURALN
NUMBER
NUMERIC
PLS_INTEGER
POSITIVE
POSITIVEN
REAL
SIGNTYPE
SMALLINT

CHAR
CHARACTER
LONG
LONG RAW
NCHAR
NVARCHAR2
RAW
ROWID
STRING
UROWID
VARCHAR
VARCHAR2

BOOLEAN

DATE
INTERVAL DAY TO SECOND
INTERVAL YEAR TO MONTH
TIMESTAMP
TIMESTAMP WITH LOCAL TIME ZONE
TIMESTAMP WITH TIME ZONE

Composite Types

RECORD
TABLE
VARRAY

LOB Types

BFILE
BLOB
CLOB
NCLOB

Reference Types

REF CURSOR
REF object_type

PL/SQL Records

What are records?

Records are another type of datatypes which oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc. Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

Declaring a record:
To declare a record, you must first define a composite datatype; then declare a record for that type.

The General Syntax to define a composite datatype is:

```
TYPE record_type_name IS RECORD
(first_col_name column_datatype,
second_col_name column_datatype, ...);
```

- *record_type_name* – it is the name of the composite type you want to define.

- *first_col_name, second_col_name, etc.,- it is the* names the fields/columns within the record.
- *column_datatype* defines the scalar datatype of the fields.

There are different ways you can declare the datatype of the fields.

1) You can declare the field in the same way as you declare the fieds while creating the table.
2) If a field is based on a column from database table, you can define the field_type as follows:

col_name **table_name.column_name%type;**

By declaring the field datatype in the above method, the datatype of the column is dynamically applied to the field. This method is useful when you are altering the column specification of the table, because you do not need to change the code again.

NOTE: You can use also *%type* to declare variables and constants.

The General Syntax to declare a record of a uer-defined datatype is:
record_name record_type_name;

The following code shows how to declare a record called *employee_rec* based on a user-defined type.

```
DECLARE
TYPE employee_type IS RECORD
(employee_id number(5),
 employee_first_name varchar2(25),
 employee_last_name employee.last_name%type,
 employee_dept employee.dept%type);
 employee_salary employee.salary%type;
 employee_rec employee_type;
```

If all the fields of a record are based on the columns of a table, we can declare the record as follows:

record_name table_name%ROWTYPE;

For example, the above declaration of employee_rec can as follows:

```
DECLARE
 employee_rec employee%ROWTYPE;
```

The advantages of declaring the record as a ROWTYPE are:
1) You do not need to explicitly declare variables for all the columns in a table.
2) If you alter the column specification in the database table, you do not need to update the code.

The disadvantage of declaring the record as a ROWTYPE is:
1) When u create a record as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields. So use ROWTYPE only when you are using all the columns of the table in the program.

NOTE: When you are creating a record, you are just creating a datatype, similar to creating a variable. You need to assign values to the record to use them.

The following table consolidates the different ways in which you can define and declare a pl/sql record.

Syntax	**Usage**
TYPE record_type_name IS RECORD (column_name1 datatype, column_name2 datatype, ...);	Define a composite datatype, where each field is scalar.
col_name table_name.column_name%type;	Dynamically define the datatype of a column based on a database column.
record_name record_type_name;	Declare a record based on a user-defined type.
record_name table_name%ROWTYPE;	Dynamically declare a record based on an entire row of a table. Each column in the table corresponds to a field in the record.

Passing Values To and From a Record

When you assign values to a record, you actually assign values to the fields within it.
The General Syntax to assign a value to a column within a record direclty is:

record_name.col_name := value;

If you used %ROWTYPE to declare a record, you can assign values as shown:

record_name.column_name := value;

We can assign values to records using SELECT Statements as shown:

```
SELECT col1, col2
INTO record_name.col_name1, record_name.col_name2
FROM table_name
[WHERE clause];
```

If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each columns separately. In this case, you must SELECT all the columns from the table into the record as shown:

```
SELECT * INTO record_name
FROM table_name
[WHERE clause];
```

Lets see how we can get values from a record.
The General Syntax to retrieve a value from a specific field into another variable is:

var_name := record_name.col_name;

The following table consolidates the different ways you can assign values to and from a record:

Syntax	**Usage**
record_name.col_name := value;	To directly assign a value to a specific column of a record.
record_name.column_name := value;	To directly assign a value to a specific column of a record, if the record is declared using %ROWTYPE.
SELECT col1, col2 INTO record_name.col_name1, record_name.col_name2 FROM table_name [WHERE clause];	To assign values to each field of a record from the database table.
SELECT * INTO record_name FROM table_name [WHERE clause];	To assign a value to all fields in the record from a database table.
variable_name := record_name.col_name;	To get a value from a record column and assigning it to a variable.

Conditional Statement (operator IF)

```
IF condition THEN
sequence_of_statements
END IF;

IF condition THEN
sequence_of_statements1
ELSE
sequence_of_statements2
END IF;

IF condition1 THEN
sequence_of_statements1
ELSIF condition2 THEN
sequence_of_statements2
ELSE
sequence_of_statements3
END IF;
```

```
IF sales > quota THEN
compute_bonus(empid);
UPDATE payroll SET pay = pay + bonus WHERE empno = emp_id;
END IF;


IF trans_type = 'CR' THEN
UPDATE accounts SET balance = balance + credit WHERE ...
ELSE
UPDATE accounts SET balance = balance - debit WHERE ...
END IF;


BEGIN
...
IF sales > 50000 THEN
bonus := 1500;
ELSIF sales > 35000 THEN
bonus := 500;
ELSE
bonus := 100;
END IF;
INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

Conditional Statement (operator CASE)

```
dbms_output.put_line(text or field varchar type);
-- auxiliary library operator to print out variable values
```

```
--school grades:
IF grade = 'A' THEN
dbms_output.put_line('Excellent');
ELSIF grade = 'B' THEN
dbms_output.put_line('Very Good');
ELSIF grade = 'C' THEN
dbms_output.put_line('Good');
ELSIF grade = 'D' THEN
dbms_output. put_line('Fair');
ELSIF grade = 'F' THEN
dbms_output.put_line('Poor');
ELSE
dbms_output.put_line('No such grade');
END IF;
```

```
CASE grade
WHEN 'A' THEN dbms_output.put_line('Excellent');
WHEN 'B' THEN dbms_output.put_line('Very
Good');
WHEN 'C' THEN dbms_output.put_line('Good');
WHEN 'D' THEN dbms_output.put_line('Fair');
WHEN 'F' THEN dbms_output.put_line('Poor');
ELSE dbms_output.put_line('No such grade');
END CASE;
```

Iterative Control: LOOP and EXIT Statements

```
LOOP
...
IF credit_rating < 3 THEN
...
EXIT; -- exit loop immediately
END IF;
-- or
EXIT WHEN credit_rating < 3;
END LOOP;
-- control resumes here
```

```
-- another loop
WHILE total <= 25000 LOOP
...
SELECT sal INTO salary FROM
        emp WHERE ...
total := total + salary;
END LOOP;
```

```
FOR counter IN [REVERSE] lower_bound..higher_bound
LOOP
sequence_of_statements
END LOOP;

FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
sequence_of_statements -- executes three times
END LOOP;

FOR i IN REVERSE 1..3 LOOP -- assign the values 3,2,1 to i
sequence_of_statements -- executes three times
END LOOP;
```

```
DECLARE
result temp.col1%TYPE;
CURSOR c1 IS
SELECT n1, n2, n3 FROM data_table WHERE exper_num =
1;
BEGIN
FOR c1_rec IN c1 LOOP
/* calculate and store the results */
result := c1_rec.n2 / (c1_rec.n1 + c1_rec.n3);
INSERT INTO temp VALUES (result, NULL, NULL);
END LOOP;
COMMIT;
END;
```

What are Cursors?

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active* set.

There are two types of cursors in PL/SQL:

Implicit cursors:

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Explicit cursors:

They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

Implicit Cursors:

When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.
When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

The status of the cursor for each of these attributes are defined in the below table.

Attributes	**Return Value**	**Example**
%FOUND	The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row.	SQL%FOUND
	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row.	
%NOTFOUND	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row.	SQL%NOTFOUND
	The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row.	
%ROWCOUNT	Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT	SQL%ROWCOUNT

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

```
DECLARE  var_rows number(5);
BEGIN
  UPDATE employee
  SET salary = salary + 1000;
  IF SQL%NOTFOUND THEN
    dbms_output.put_line('None of the salaries where updated');
  ELSIF SQL%FOUND THEN
    var_rows := SQL%ROWCOUNT;
    dbms_output.put_line('Salaries for ' || var_rows || 'employees are updated');
  END IF;
END;
```

In the above PL/SQL Block, the salaries of all the employees in the ‘employee’ table are updated. If none of the employee’s salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 1000 employees are updated' if there are 1000 rows in ‘employee’ table.

Explicit Cursors

An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor.

The General Syntax for creating a cursor is as given below:

CURSOR cursor_name IS select_statement;

- *cursor_name – A suitable name for the cursor.*
- *select_statement – A select query which returns multiple rows.*

How to use Explicit Cursor?

There are four steps in using an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

1) Declaring a Cursor in the Declaration Section:

```
  DECLARE
  CURSOR emp_cur IS
  SELECT *
  FROM emp_tbl
  WHERE salary > 5000;
```

In the above example we are creating a cursor ‘emp_cur’ on a query which returns the records of all the

employees with salary greater than 5000. Here ‘emp_tbl’ in the table which contains records of all the employees.

2) Accessing the records in the cursor:

Once the cursor is created in the declaration section we can access the cursor in the execution section of the PL/SQL program.

How to access an Explicit Cursor?

These are the three steps in accessing the cursor.
1) Open the cursor.
2) Fetch the records in the cursor one at a time.
3) Close the cursor.

General Syntax to open a cursor is:

OPEN cursor_name;

General Syntax to fetch records from a cursor is:

FETCH cursor_name INTO record_name;
OR
FETCH cursor_name INTO variable_list;

General Syntax to close a cursor is:

CLOSE cursor_name;

When a cursor is opened, the first row becomes the current row. When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row. On every fetch statement, the pointer moves to the next row. If you want to fetch after the last row, the program will throw an error. When there is more than one row in a cursor we can use loops along with explicit cursor attributes to fetch all the records.

Points to remember while fetching a row:

· We can fetch the rows in a cursor to a PL/SQL Record or a list of variables created in the PL/SQL Block.
· If you are fetching a cursor to a PL/SQL Record, the record should have the same structure as the cursor.
· If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.

General Form of using an explicit cursor is:

```
DECLARE
   variables;
   records;
   create a cursor;
```

```
BEGIN
  OPEN cursor;
  FETCH cursor;
    process the records;
  CLOSE cursor;
END;
```

Lets Look at the example below

Example 1:

```
1> DECLARE
2>    emp_rec emp_tbl%rowtype;
3>    CURSOR emp_cur IS
4>    SELECT *
5>    FROM
6>    WHERE salary > 10;
7> BEGIN
8>    OPEN emp_cur;
9>    FETCH emp_cur INTO emp_rec;
10>      dbms_output.put_line (emp_rec.first_name || ' ' || emp_rec.last_name);
11>   CLOSE emp_cur;
12> END;
```

In the above example, first we are creating a record ‘emp_rec’ of the same structure as of table ‘emp_tbl’ in line no 2. We can also create a record with a cursor by replacing the table name with the cursor name. Second, we are declaring a cursor ‘emp_cur’ from a select query in line no 3 - 6. Third, we are opening the cursor in the execution section in line no 8. Fourth, we are fetching the cursor to the record in line no 9. Fifth, we are displaying the first_name and last_name of the employee in the record emp_rec in line no 10. Sixth, we are closing the cursor in line no 11.

What are Explicit Cursor Attributes?

Oracle provides some attributes known as Explicit Cursor Attributes to control the data processing while using cursors. We use these attributes to avoid errors while accessing cursors through OPEN, FETCH and CLOSE Statements.

When does an error occur while accessing an explicit cursor?

a) When we try to open a cursor which is not closed in the previous operation.
b) When we try to fetch a cursor after the last operation.

These are the attributes available to check the status of an explicit cursor.

Attributes	Return values	Example
%FOUND	TRUE, if fetch statement returns at least one row.	Cursor_name%FOUND
	FALSE, if fetch statement doesn’t return a row.	
%NOTFOUND	TRUE, , if fetch statement doesn’t return a row.	Cursor_name%NOTFOUND
	FALSE, if fetch statement returns at least one row.	

%ROWCOUNT	The number of rows fetched by the fetch statement	Cursor_name%ROWCOUNT
	If no row is returned, the PL/SQL statement returns an error.	
%ISOPEN	TRUE, if the cursor is already open in the program	Cursor_name%ISNAME
	FALSE, if the cursor is not opened in the program.	

Using Loops with Explicit Cursors:

Oracle provides three types of cursors namely SIMPLE LOOP, WHILE LOOP and FOR LOOP. These loops can be used to process multiple rows in the cursor. Here I will modify the same example for each loops to explain how to use loops with cursors.

Cursor with a Simple Loop:

```
1> DECLARE
2>   CURSOR emp_cur IS
3>   SELECT first_name, last_name, salary FROM emp_tbl;
4>   emp_rec emp_cur%rowtype;
5> BEGIN
6>   IF NOT sales_cur%ISOPEN THEN
7>      OPEN sales_cur;
8>   END IF;
9>   LOOP
10>     FETCH emp_cur INTO emp_rec;
11>     EXIT WHEN emp_cur%NOTFOUND;
12>     dbms_output.put_line(emp_cur.first_name || ' ' ||emp_cur.last_name
13>     || ' ' ||emp_cur.salary);
14>  END LOOP;
15>  END;
16>  /
```

In the above example we are using two cursor attributes %ISOPEN and %NOTFOUND.
In line no 6, we are using the cursor attribute %ISOPEN to check if the cursor is open, if the condition is true the program does not open the cursor again, it directly moves to line no 9.
In line no 11, we are using the cursor attribute %NOTFOUND to check whether the fetch returned any row. If there is no rows found the program would exit, a condition which exists when you fetch the cursor after the last row, if there is a row found the program continues.

We can use %FOUND in place of %NOTFOUND and vice versa. If we do so, we need to reverse the logic of the program. So use these attributes in appropriate instances.

Cursor with a While Loop:

Lets modify the above program to use while loop.

```
1> DECLARE
2>  CURSOR emp_cur IS
3>  SELECT first_name, last_name, salary FROM emp_tbl;
4>  emp_rec emp_cur%rowtype;
5> BEGIN
```

```
6>   IF NOT sales_cur%ISOPEN THEN
7>      OPEN sales_cur;
8>   END IF;
9>   FETCH sales_cur INTO sales_rec;
10>  WHILE sales_cur%FOUND THEN
11>  LOOP
12>    dbms_output.put_line(emp_cur.first_name || ' ' ||emp_cur.last_name
13>    || ' ' ||emp_cur.salary);
15>    FETCH sales_cur INTO sales_rec;
16>  END LOOP;
17> END;
18> /
```

In the above example, in line no 10 we are using %FOUND to evaluate if the first fetch statement in line no 9 returned a row, if true the program moves into the while loop. In the loop we use fetch statement again (line no 15) to process the next row. If the fetch statement is not executed once before the while loop the while condition will return false in the first instance and the while loop is skipped. In the loop, before fetching the record again, always process the record retrieved by the first fetch statement, else you will skip the first row.

Cursor with a FOR Loop:

When using FOR LOOP you need not declare a record or variables to store the cursor values, need not open, fetch and close the cursor. These functions are accomplished by the FOR LOOP automatically.

General Syntax for using FOR LOOP:

```
FOR record_name IN cusror_name
LOOP
   process the row...
END LOOP;
```

Let’s use the above example to learn how to use for loops in cursors.

```
DECLARE
 CURSOR emp_cur IS
 SELECT first_name, last_name, salary FROM emp_tbl;
 emp_rec emp_cur%rowtype;
BEGIN
 FOR emp_rec in sales_cur
 LOOP
 dbms_output.put_line(emp_cur.first_name || ' ' ||emp_cur.last_name
   || ' ' ||emp_cur.salary);
END LOOP;
END;
/
```

In the above example, when the FOR loop is processed a record ‘emp_rec’of structure ‘emp_cur’ gets created, the cursor is opened, the rows are fetched to the record ‘emp_rec’ and the cursor is closed after the last row is processed. By using FOR Loop in your program, you can reduce the number of lines in the program.

NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

Stored Procedures

What is a Stored Procedure?

A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages. A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

We can pass parameters to procedures in three ways.
1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]
IS
   Declaration section
BEGIN
   Execution section
EXCEPTION
  Exception section
END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

The below example creates a procedure 'employer_details' which gives the details of the employee.

```
CREATE OR REPLACE PROCEDURE employer_details
IS
 CURSOR emp_cur IS
 SELECT first_name, last_name, salary FROM emp_tbl;
 emp_rec emp_cur%rowtype;
BEGIN
 FOR emp_rec in sales_cur
 LOOP
 dbms_output.put_line(emp_cur.first_name || ' ' ||emp_cur.last_name
   || ' ' ||emp_cur.salary);
 END LOOP;
END;
 /
```

How to execute a Stored Procedure?

There are two ways to execute a procedure.

1) From the SQL prompt.

```
 EXECUTE [or EXEC] procedure_name;
```

2) Within another procedure – simply use the procedure name.

```
  procedure_name;
```

NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

PL/SQL Functions

What is a Function in PL/SQL?

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

The General Syntax to create a function is:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS
   Declaration_section
BEGIN
   Execution_section
   Return return_variable;
EXCEPTION
   exception section
   Return return_variable;
END;
```

1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a function called ''employer_details_func' similar to the one created in stored proc

```
1> CREATE OR REPLACE FUNCTION employer_details_func
2>    RETURN VARCHAR(20);
3> IS
5>    emp_name VARCHAR(20);
6> BEGIN
7>          SELECT first_name INTO emp_name
8>          FROM emp_tbl WHERE empID = '100';
9>          RETURN emp_name;
10> END;
11> /
```

In the example we are retrieving the ‘first_name’ of employee with empID 100 to variable ‘emp_name’.
The return type of the function is VARCHAR which is declared in line no 2.
The function returns the 'emp_name' which is of type VARCHAR as the return value in line no 9.

How to execute a PL/SQL Function?

A function can be executed in the following ways.

1) Since a function returns a value we can assign it to a variable.

employee_name := employer_details_func;

If ‘employee_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

SELECT employer_details_func FROM dual;

3) In a PL/SQL Statements like,

dbms_output.put_line(employer_details_func);
This line displays the value returned by the function.

Parameters in Procedure and Functions

How to pass parameters to Procedures and Functions in PL/SQL ?

In PL/SQL, we can pass parameters to procedures and functions in three ways.

1) IN type parameter: These types of parameters are used to send values to stored procedures.
2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
3) IN OUT parameter: These types of parameters are used to send values and get values from stored procedures.

NOTE: If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.

1) IN parameter:

This is similar to passing parameters in programming languages. We can pass values to the stored procedure through these parameters or variables. This type of parameter is a read only parameter. We can assign the value of IN type parameter to a variable or use it in a query, but we cannot change its value inside the procedure.

The General syntax to pass a IN parameter is

CREATE [OR REPLACE] PROCEDURE procedure_name (
param_name1 IN datatype, param_name12 IN datatype ...)

- param_name1, □param_name2... are unique parameter names.
- datatype - defines the datatype of the variable.
- IN - is optional, by default it is a IN type parameter.

2) OUT Parameter:

The OUT parameters are used to send the OUTPUT from a procedure or a function. This is a write-only parameter i.e, we cannot pass values to OUT paramters while executing the stored procedure, but we can assign values to OUT parameter inside the stored procedure and the calling program can recieve this output value.

The General syntax to create an OUT parameter is

CREATE [OR REPLACE] PROCEDURE proc2 (param_name OUT datatype)

The parameter should be explicity declared as OUT parameter.

3) IN OUT Parameter:

The IN OUT parameter allows us to pass values into a procedure and get output values from the procedure. This parameter is used if the value of the IN parameter can be changed in the calling program.

By using IN OUT parameter we can pass values into a parameter and return a value to the calling program using the same parameter. But this is possible only if the value passed to the procedure and output value have a same datatype. This parameter is used if the value of the parameter will be changed in the procedure.

The General syntax to create an IN OUT parameter is

CREATE [OR REPLACE] PROCEDURE proc3 (param_name IN OUT datatype)

The below examples show how to create stored procedures using the above three types of parameters.

Example1:

Using IN and OUT parameter:

Let’s create a procedure which gets the name of the employee when the employee id is passed.

```
1> CREATE OR REPLACE PROCEDURE emp_name (id IN NUMBER, emp_name OUT
NUMBER)
2> IS
3> BEGIN
4>    SELECT first_name INTO emp_name
5>    FROM emp_tbl WHERE empID = id;
6> END;
7> /
```

We can call the procedure ‘emp_name’ in this way from a PL/SQL Block.

```
1> DECLARE
2>  empName varchar(20);
3>  CURSOR id_cur SELECT id FROM emp_ids;
4> BEGIN
5> FOR emp_rec in id_cur
6> LOOP
7>   emp_name(emp_rec.id, empName);
8>   dbms_output.putline('The employee ' || empName || ' has id ' || emp-rec.id);
9> END LOOP;
10> END;
11> /
```

In the above PL/SQL Block
In line no 3; we are creating a cursor ‘id_cur’ which contains the employee id.
In line no 7; we are calling the procedure ‘emp_name’, we are passing the ‘id’ as IN parameter and ‘empName’ as OUT parameter.
In line no 8; we are displaying the id and the employee name which we got from the procedure ‘emp_name’.

Example 2:

Using IN OUT parameter in procedures:

```
1> CREATE OR REPLACE PROCEDURE emp_salary_increase
2> (emp_id IN emptbl.empID%type, salary_inc IN OUT emptbl.salary%type)
3> IS
4>    tmp_sal number;
5> BEGIN
6>    SELECT salary
7>    INTO tmp_sal
8>    FROM emp_tbl
9>    WHERE empID = emp_id;
10>   IF tmp_sal between 10000 and 20000 THEN
11>      salary_inout := tmp_sal * 1.2;
12>   ELSIF tmp_sal between 20000 and 30000 THEN
```

```
      salary_inout := tmp_sal * 1.3;
   ELSIF tmp_sal > 30000 THEN
      salary_inout := tmp_sal * 1.4;
   END IF;
END;
/
```

The below PL/SQL block shows how to execute the above 'emp_salary_increase' procedure.

```
DECLARE
   CURSOR updated_sal is
   SELECT empID,salary
   FROM emp_tbl;
   pre_sal number;
BEGIN
  FOR emp_rec IN updated_sal LOOP
      pre_sal := emp_rec.salary;
      emp_salary_increase(emp_rec.empID, emp_rec.salary);
       dbms_output.put_line('The salary of ' || emp_rec.empID ||
              ' increased from '|| pre_sal || ' to '||emp_rec.salary);
   END LOOP;
END;
/
```

Exception Handling

In this section we will discuss about the following,
1) What is Exception Handling.
2) Structure of Exception Handling.
3) Types of Exception Handling.

1) What is Exception Handling?

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is recieved.
PL/SQL Exception message consists of three parts.
1) Type of Exception
2) An Error Code
3) A message
By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

2) Structure of Exception Handling.

The General Syntax for coding the exception section

```
 DECLARE
   Declaration section
 BEGIN
   Exception section
 EXCEPTION
 WHEN ex_name1 THEN
    -Error handling statements
 WHEN ex_name2 THEN
    -Error handling statements
 WHEN Others THEN
   -Error handling statements
END;
```

General PL/SQL statments can be used in the Exception Block.

When an exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'ex_name1 ', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing fo the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

If there are nested PL/SQL blocks like this.

```
 DELCARE
   Declaration section
 BEGIN
    DECLARE
      Declaration section
    BEGIN
      Execution section
    EXCEPTION
      Exception section
    END;
 EXCEPTION
   Exception section
 END;
```

In the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block. If none of the blocks handle the exception the program ends abruptly with an error.

3) Types of Exception.

There are 3 types of Exceptions.
a) Named System Exceptions

b) Unnamed System Exceptions
c) User-defined Exceptions

a) Named System Exceptions

System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.

For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions.

Named system exceptions are:
1) Not Declared explicitly,
2) Raised implicitly when a predefined Oracle error occurs,
3) caught by referencing the standard name within an exception-handling routine.

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

For Example: Suppose a NO_DATA_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

```
BEGIN
  Execution section
EXCEPTION
WHEN NO_DATA_FOUND THEN
 dbms_output.put_line ('A SELECT...INTO did not return any row.');
 END;
```

b) Unnamed System Exceptions

Those system exception for which oracle does not provide a name is known as unamed system exception. These exception do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed sysyem exceptions:
1. By using the WHEN OTHERS exception handler, or
2. By associating the exception code to a name and using it as a named exception.

We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION_INIT. EXCEPTION_INIT** will associate a predefined Oracle error number to a programmer_defined exception name.

Steps to be followed to use unnamed system exceptions are
- They are raised implicitly.
- If they are not handled in WHEN Others they must be handled explicity.
- To handle the exception explicity, they must be declared using Pragma EXCEPTION_INIT as given above and handled referecing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

```
DECLARE
   exception_name EXCEPTION;
   PRAGMA
   EXCEPTION_INIT (exception_name, Err_code);
BEGIN
Execution section
EXCEPTION
  WHEN exception_name THEN
     handle the exception
END;
```

For Example: Lets consider the product table and order_items table from sql joins.

Here product_id is a primary key in product table and a foreign key in order_items table.
If we try to delete a product_id from the product table when it has child records in order_id table an exception will be thrown with oracle code number -2292.
We can provide a name to this exception and handle it in the exception section as given below.

```
 DECLARE
  Child_rec_exception EXCEPTION;
  PRAGMA
   EXCEPTION_INIT (Child_rec_exception, -2292);
BEGIN
  Delete FROM product where product_id= 104;
EXCEPTION
   WHEN Child_rec_exception
   THEN Dbms_output.put_line('Child records are present for this product_id.');
END;
/
```

c) User-defined Exceptions

Apart from sytem exceptions we can explicity define exceptions based on business rules. These are known as user-defined exceptions.

Steps to be followed to use user-defined exceptions:
- They should be explicitly declared in the declaration section.
- They should be explicitly raised in the Execution Section.
- They should be handled by referencing the user-defined exception name in the exception section.

For Example: Lets consider the product table and order_items table from sql joins to explain user-defined exception.
Lets create a business rule that if the total no of units of any particular product sold is more than 20, then it is a huge quantity and a special discount should be provided.

```
DECLARE
  huge_quantity EXCEPTION;
  CURSOR product_quantity is
  SELECT p.product_name as name, sum(o.total_units) as units
  FROM order_tems o, product p
  WHERE o.product_id = p.product_id;
  quantity order_tems.total_units%type;
  up_limit CONSTANT order_tems.total_units%type := 20;
  message VARCHAR2(50);
BEGIN
  FOR product_rec in product_quantity LOOP
    quantity := product_rec.units;
     IF quantity > up_limit THEN
      message := 'The number of units of product ' || product_rec.name ||
                 ' is more than 20. Special discounts should be provided.
                              Rest of the records are skipped. '
     RAISE huge_quantity;
     ELSIF quantity < up_limit THEN
      v_message:= 'The number of unit is below the discount limit.';
     END IF;
     dbms_output.put_line (message);
  END LOOP;
 EXCEPTION
   WHEN huge_quantity THEN
     dbms_output.put_line (message);
 END;
/
```

RAISE_APPLICATION_ERROR ( )

RAISE_APPLICATION_ERROR is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.

Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements).

RAISE_APPLICATION_ERROR raises an exception but does not handle it.

RAISE_APPLICATION_ERROR is used for the following reasons,
a) to create a unique id for an user-defined exception.
b) to make the user-defined exception look like an Oracle error.

The General Syntax to use this procedure is:

RAISE_APPLICATION_ERROR (error_number, error_message);

- The Error number must be between -20000 and -20999
- The Error_message is the message you want to display when the error occurs.

Steps to be folowed to use RAISE_APPLICATION_ERROR procedure:

1. Declare a user-defined exception in the declaration section.
2. Raise the user-defined exception based on a specific business rule in the execution section.
3. Finally, catch the exception and link the exception to a user-defined error number in RAISE_APPLICATION_ERROR.

Using the above example we can display a error message using RAISE_APPLICATION_ERROR.

```
DECLARE
  huge_quantity EXCEPTION;
  CURSOR product_quantity is
  SELECT p.product_name as name, sum(o.total_units) as units
  FROM order_tems o, product p
  WHERE o.product_id = p.product_id;
  quantity order_tems.total_units%type;
  up_limit CONSTANT order_tems.total_units%type := 20;
  message VARCHAR2(50);
BEGIN
  FOR product_rec in product_quantity LOOP
    quantity := product_rec.units;
     IF quantity > up_limit THEN
        RAISE huge_quantity;
     ELSIF quantity < up_limit THEN
      v_message:= 'The number of unit is below the discount limit.';
     END IF;
     Dbms_output.put_line (message);
  END LOOP;
 EXCEPTION
   WHEN huge_quantity THEN
               raise_application_error(-2100, 'The number of unit is above the discount limit.');
 END;
/
```

What is a Trigger?

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

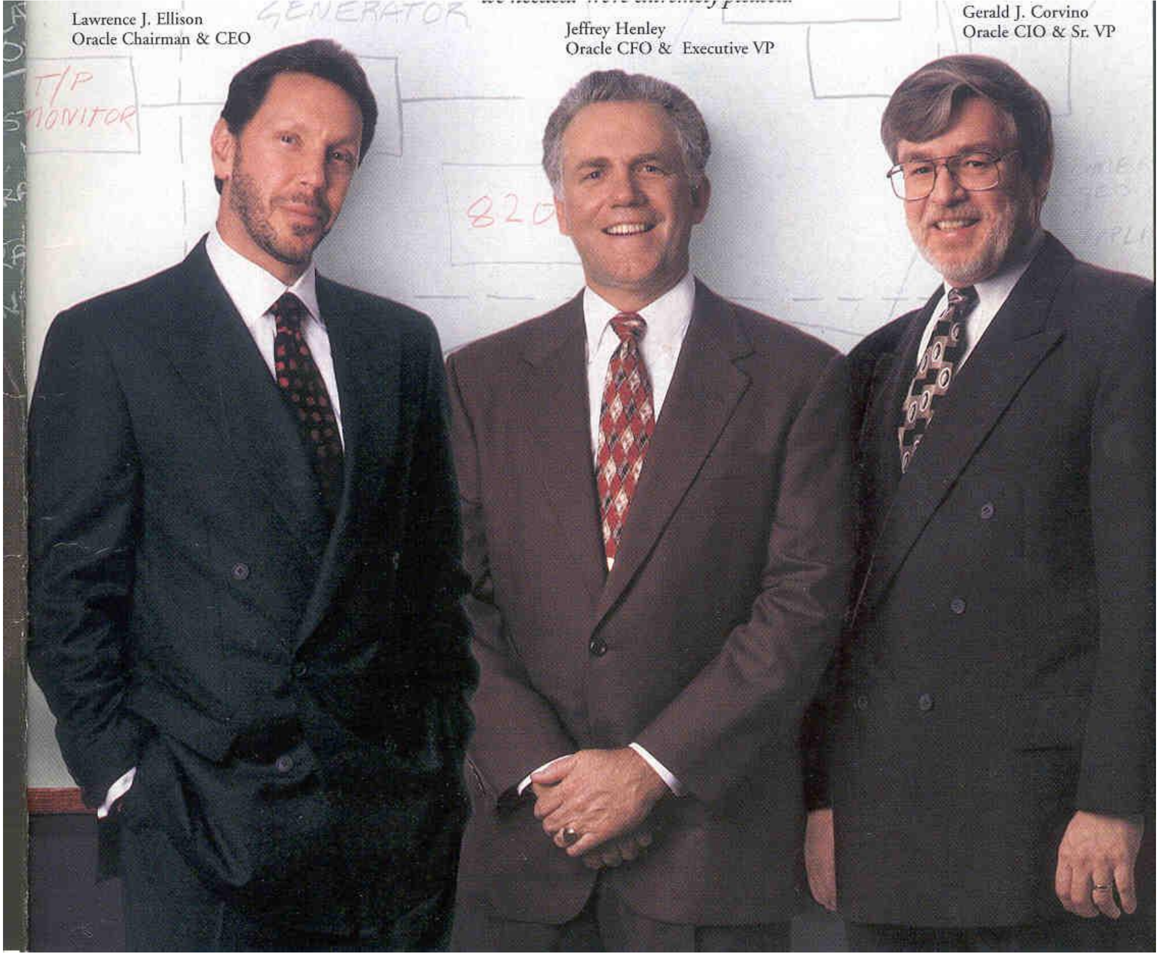
Syntax of Triggers

The Syntax for creating a trigger is:

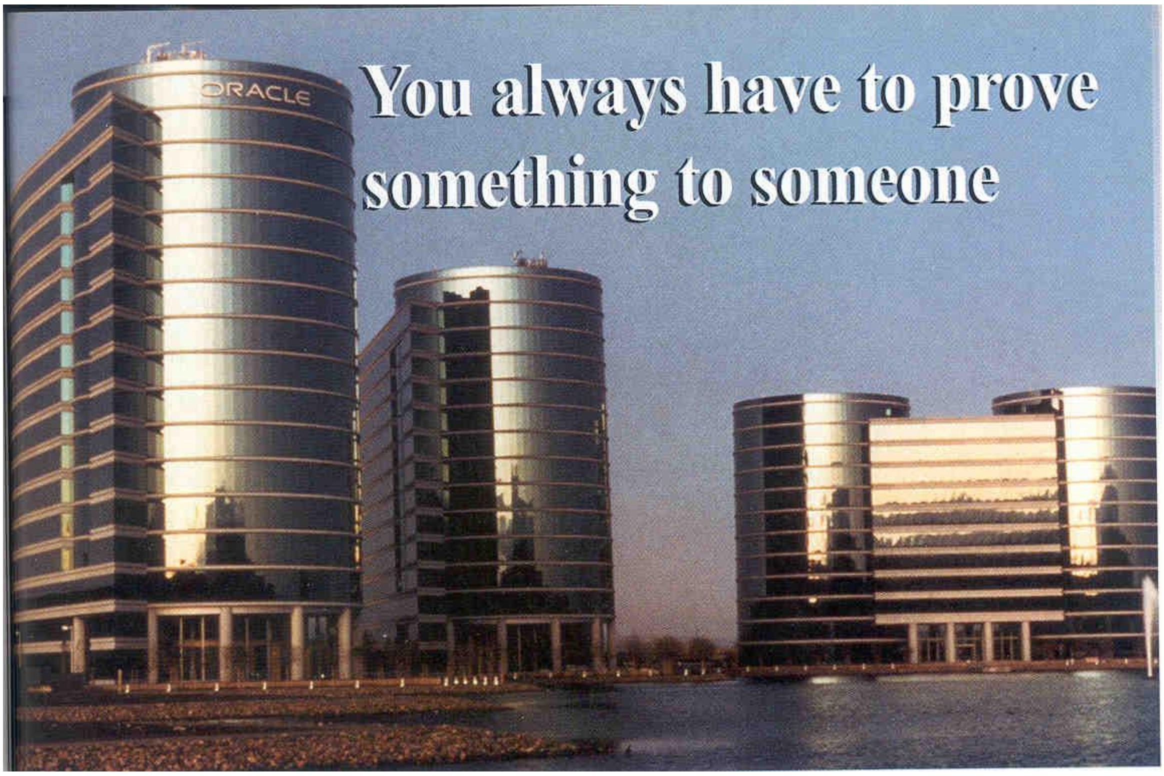
```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
  --- sql statements
END;
```

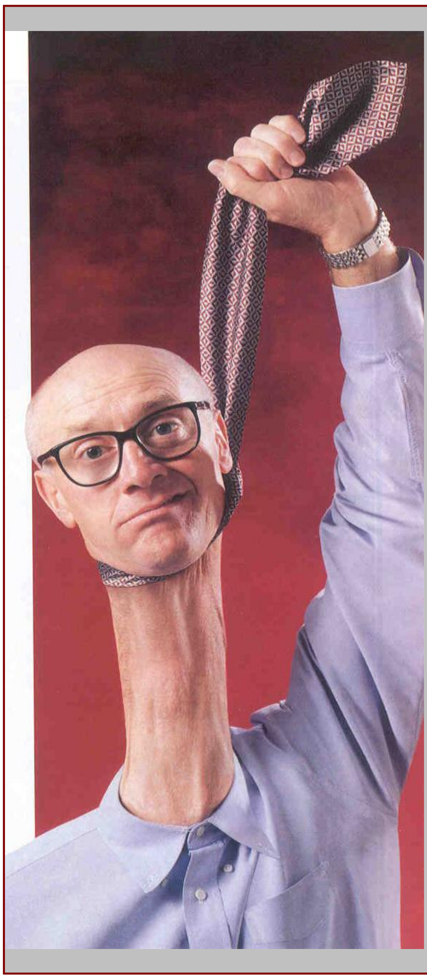
- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *{BEFORE | AFTER | INSTEAD OF }* - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- *{INSERT [OR] | UPDATE [OR] | DELETE}* - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- *[OF col_name]* - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- *CREATE [OR REPLACE] TRIGGER trigger_name* - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- *[ON table_name]* - This clause identifies the name of the table or view to which the trigger is associated.
- *[REFERENCING OLD AS o NEW AS n]* - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- *[FOR EACH ROW]* - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).
- *WHEN (condition)* - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Lawrence (Lary) Ellison: The head and author of “Oracle”.



“Oracle” – head office.





DB Administrating

(principle tasks. In Database III)

- Data base (Server/Application) Install and Upgrade
- Planning and management of system memomry
- DB prime creation
- DB Backup / Recovery
- Data archiving
- Server management
- Usernames and password management