



©Mahel2003

# **Tanveer Khan**

**MS Computer Science**

**Project Management Professional (PMP)**

**Oracle Certified Professional (OCP)**

**Teradata Certified Professional**

**ERP Certified**

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# Introduction

to

# Oracle SQL

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# Introduction

Copyright © Tanveer Khan, 2005 - 2016. All rights reserved.

## ➤ DATABASE:

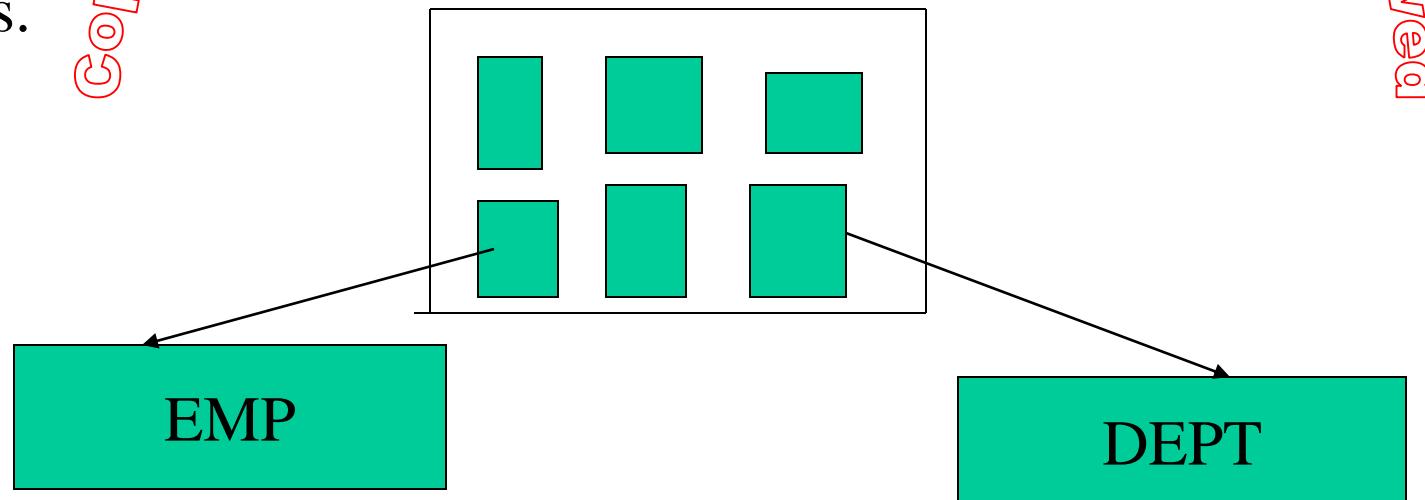
A database is an organized collection of information.

## ➤ DATABASE MANAGEMENT SYSTEM (DBMS):

To manage databases we need DBMS. A DBMS is a program that stores, retrieves, and modifies data in the databases on request.

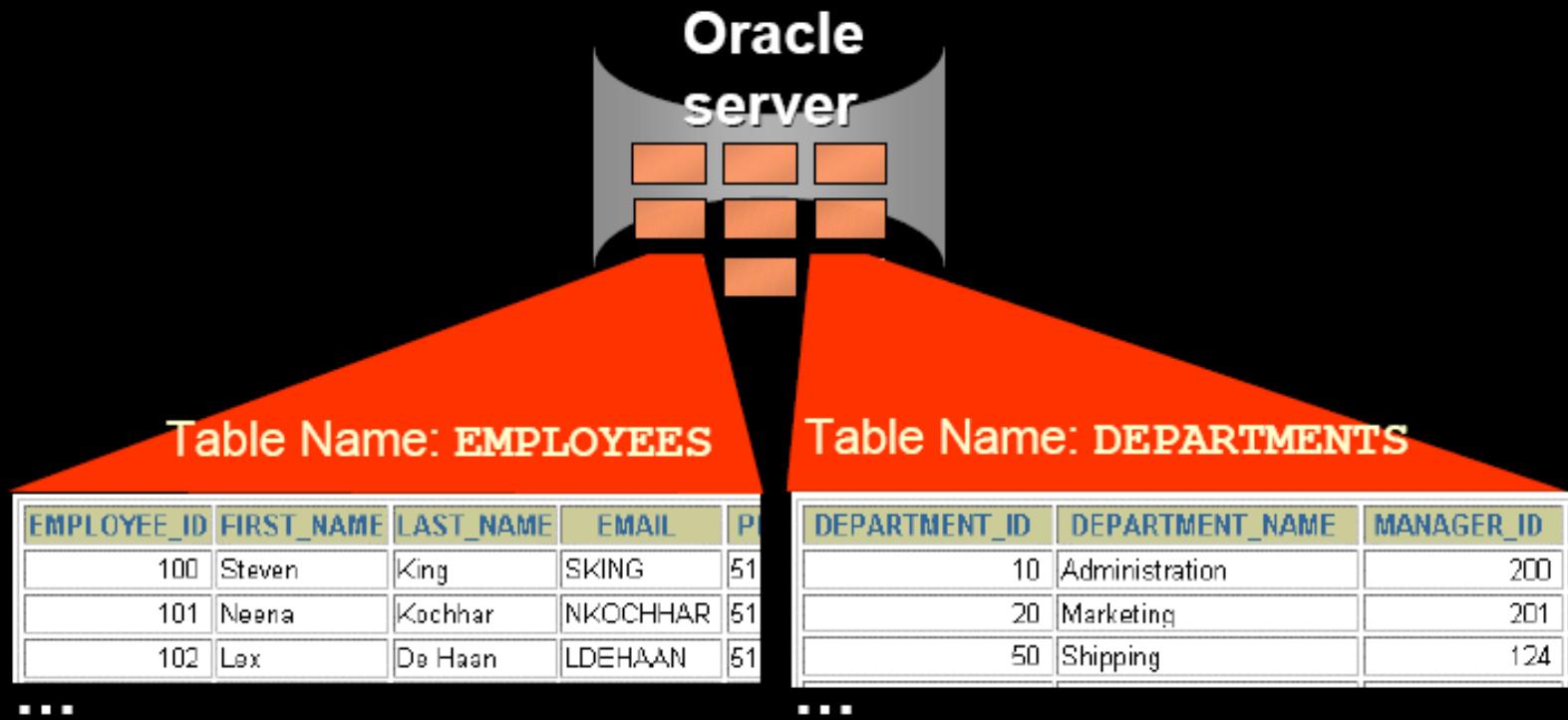
## ➤ RELATIONAL DATABASE MANAGEMENT SYSTEM:

A Relational Database is a collection of relations of two dimensional tables.



# Definition of a Relational Database

A relational database is a collection of relations or two-dimensional tables.



## ➤ TERMINOLOGIES USED IN RDBMS:

### 1) Table:

A Table is the basic storage structure of an RDBMS. A table stores all the data necessary about something in the real world.

#### Example:

Employees.

### 2) Single Row or Tuple:

Representing all data required for a particular employee. Each row in a table may be identified by a Primary Key, which allows no duplicate rows.

### 3) Column or Attribute:

Usually refers to a characteristic of an entity.

### 4) Primary Key:

A Field which uniquely identifies a row.

## 5) Foreign Key:

A Foreign Key is a column that identifies how tables relate to each other. A foreign key refers to a Primary Key in another table.

### RELATING MULTIPLE TABLES.

- Each row of data in a table is uniquely identified by a Primary Key.
- You can logically relate data from multiple tables using foreign keys.

#### Example:

**EMP** may have *empno,ename,job,deptno* where *empno* is a primary key and *deptno* is a foreign key.

**DEPT** may have *deptno,dname,loc* where *deptno* is a primary key.

## **GUIDELINES FOR PRIMARY AND FOREIGN KEYS:**

- No duplicate values are allowed in a primary key.
- Primary keys generally cannot be changed.
- Foreign keys are based on data values and are purely logical, not physical pointers.
- A foreign key must match an existing primary key or else be null.

## **RELATIONAL DATABASE PROPERTIES:**

A relational database

- can be accessed and modified by executing **Structured Query Language (SQL)**.
- Contains a collection of tables with no physical pointer
- Use a set of operators for partitioning and combining relations e.g *where clause, orderby clause*.

# Common Data types used in Databases:

datatype	size	Type of Data Hold
CHAR	255 characters	Character
VARCHAR2	> 2000 characters	Character
NUMBER	$9.99 * 10^{124}$	Integer, Float
DATE	DD-MON-YY	Date and time
LONG	65,535 characters	Binary Data.

# Common Data types used in MS Access:

<b>DATA TYPE</b>	<b>TYPE OF DATA HOLD</b>
TEXT	Character
MEMO	Character
DATE/TIME	Date & Time
NUMBER	Integer, Long Integer , Decimal etc.
CURRENCY	Dollar , Euro etc.

# **SQL (Structured Query Language):**

ANSI SQL is divided into five different sections.

## **1) Data Retrieval:**

Select.

## **2) Data Manipulation Language (DML):**

Insert, Update, Delete , Merge

## **3) Data Definition Language (DDL):**

Create, Alter, Drop, Rename, Truncate.

## **4) Data Control Language (DCL):**

Grant, Revoke.

## **5) Transaction Control:**

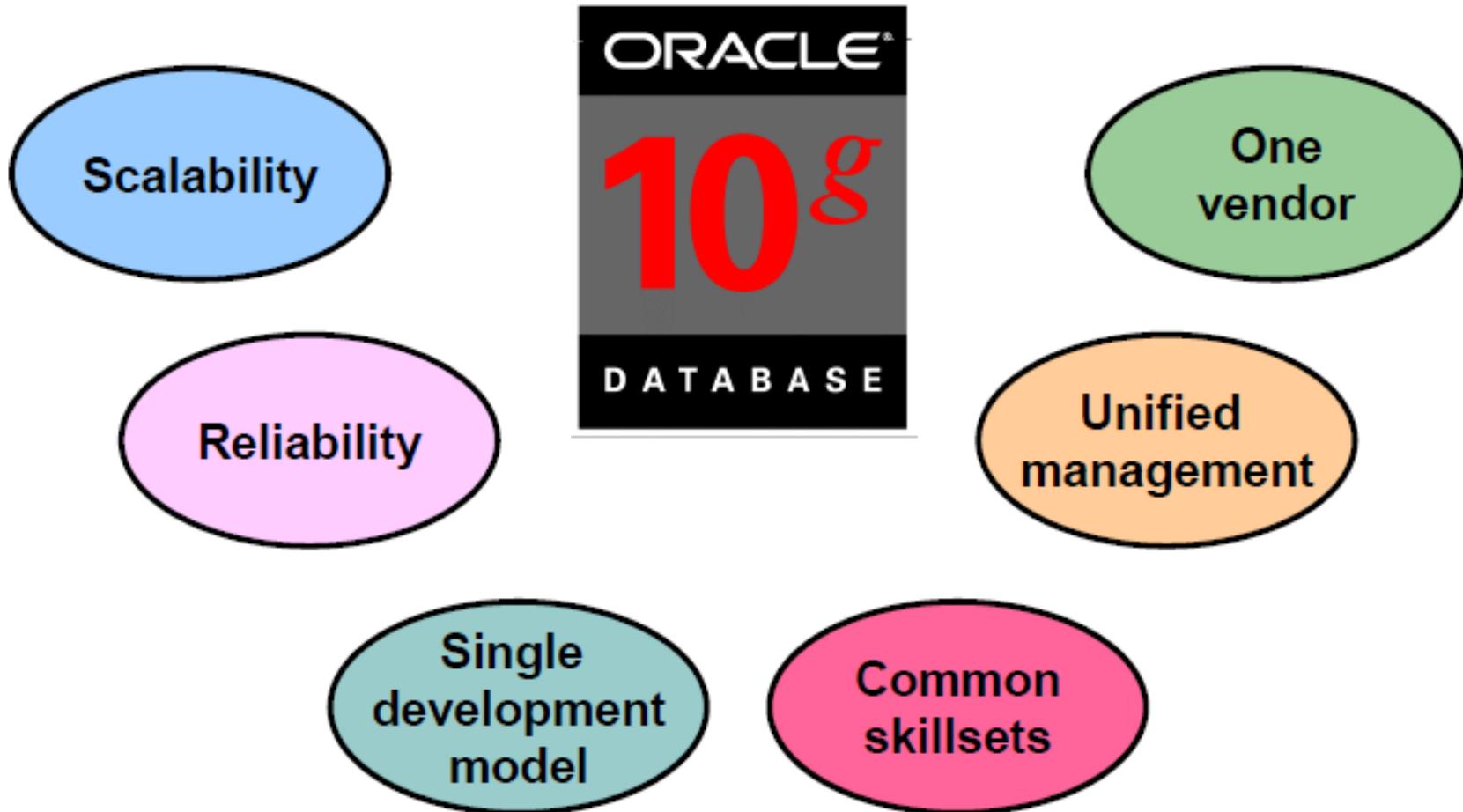
Commit, Rollback, Savepoint.

# PL/SQL:

- PL/SQL is an extension to SQL with design features of programming language.
- Data manipulation and query statements of SQL are included within procedural units of code.

Copyright Tanveer Khan, 2005 - 2016 All Rights Reserved

# Oracle10g



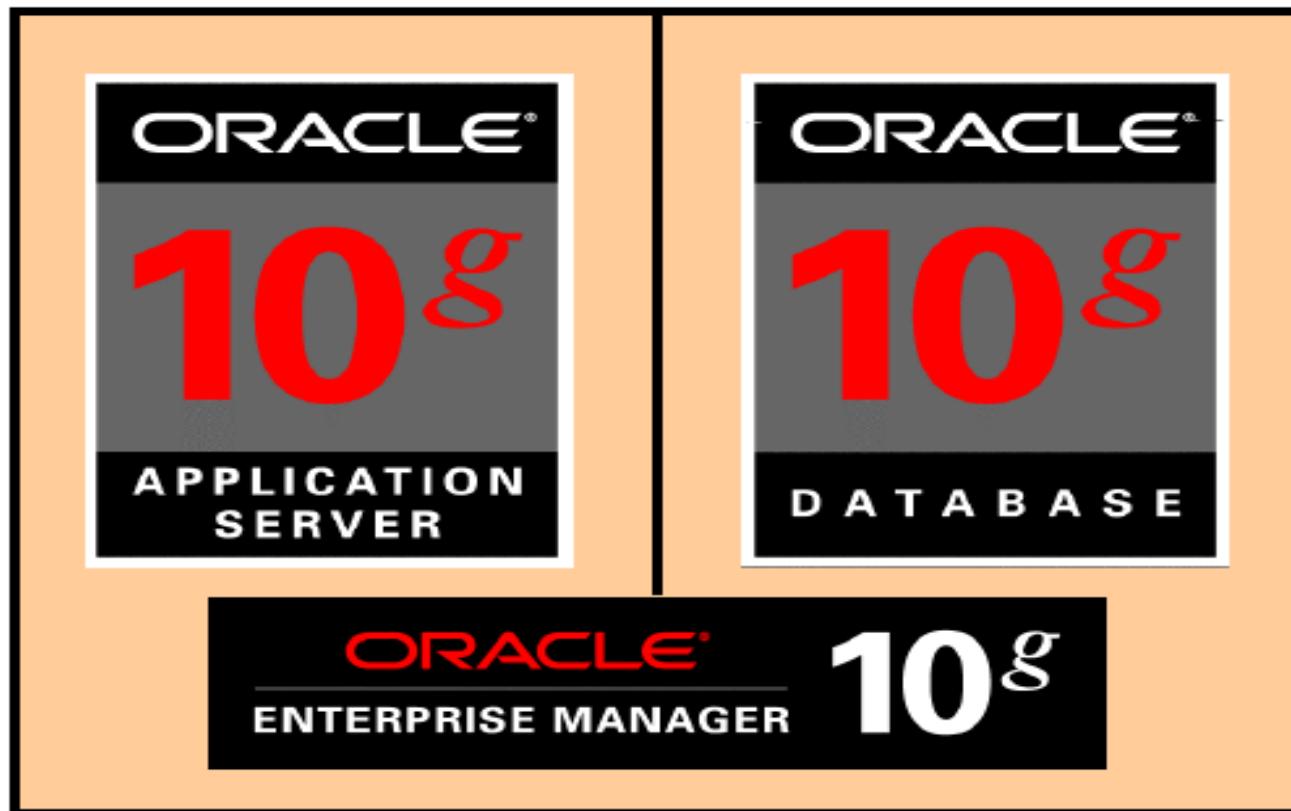
## oracle10g Features

- Oracle offers a comprehensive high-performance infrastructure for e-business. It is called oracle10g.
- oracle10g can support up to 512 Peta Bytes (1024 Tera Bytes) of data.
- oracle10g includes everything needed to develop, deploy, and manage Internet applications.

### ➤ Benefits include:

1. Scalability from departments to enterprise e-business sites
2. Robust reliable, available, secure architecture
3. One development model, easy deployment options
4. Leverage an organization's current skillset throughout the Oracle platform (including SQL, PL/SQL, Java, and XML)
5. One management interface for all applications
6. Industry standard technologies, no proprietary lock-in

# Oracle10g

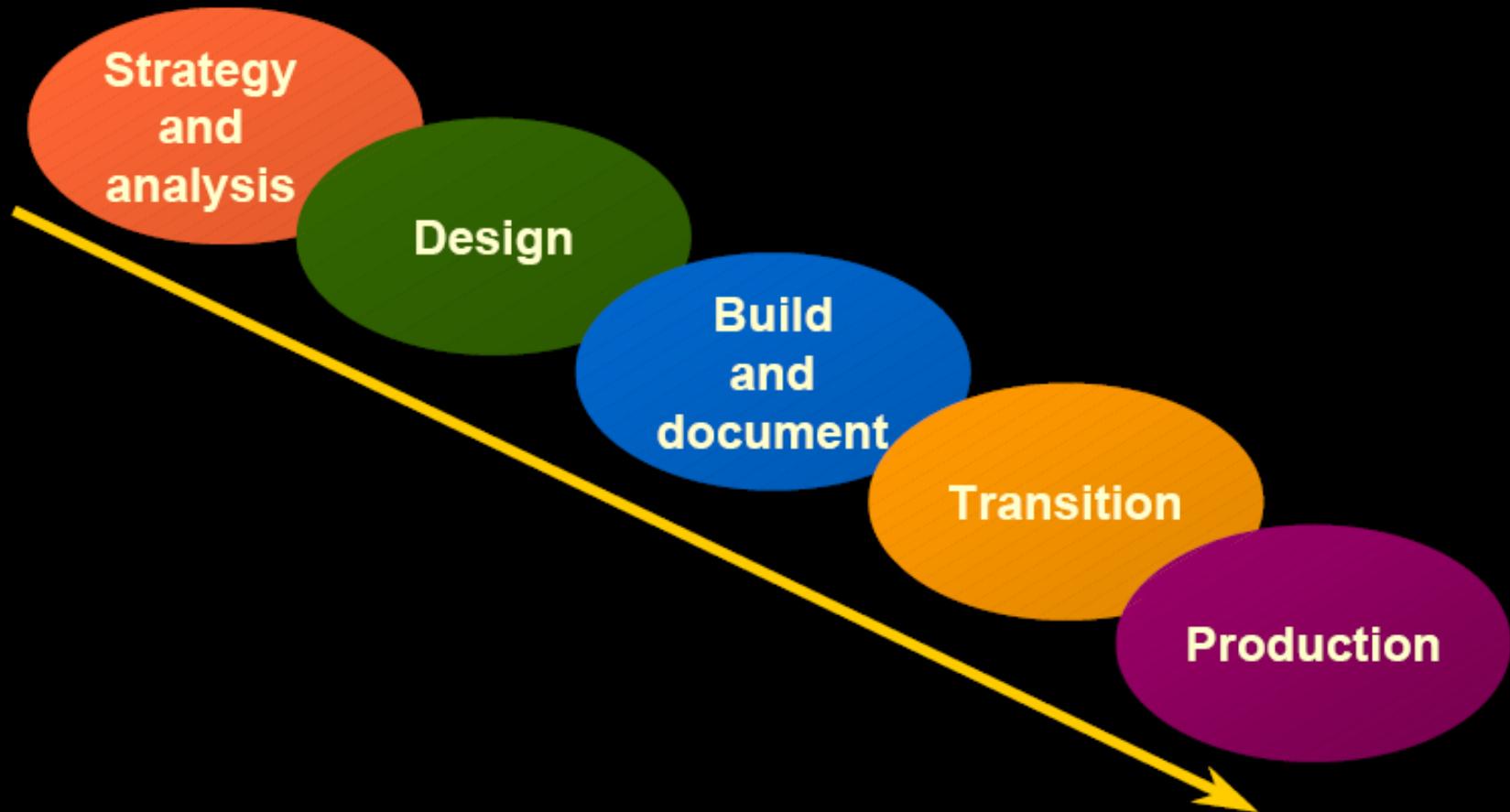


There are two products, *Oracle Application Server* and *Oracle Database Server*, that provide a complete and simple infrastructure for Internet applications.

# **Relational and Object Relational Database Management System**

- **Relational model and object relational model**
- **User-defined data types and objects**
- **Fully compatible with relational database**
- **Support of multimedia and large objects**
- **High-quality database server features**

# System Development Life Cycle



# System Development Life Cycle

- From concept to production, you can develop a database by using the system development life cycle, which contains multiple stages of development.
- This top-down, systematic approach to database development transforms business information requirements into an operational database.

## Strategy and Analysis

- Study and analyze the business requirements.
- Interview users and managers to identify the information requirements.
- Incorporate the enterprise and application mission statements as well as any future system specifications.
- Build models of the system.
- Transfer the business narrative into a graphical representation of business information needs and rules. Confirm and refine the model with the analysts and experts.

## **Design**

- Design the database based on the model developed in the strategy and analysis phase.

## **Build and Document**

- Build the prototype system.
- Write and execute the commands to create the tables and supporting objects for the database.
- Develop user documentation, Help text, and operations manuals to support the use and operation of the system.

## **Transition**

- Refine the prototype.
- Move the application into production with user acceptance testing, conversion of existing data, and parallel operations. Make any modifications required.

## **Production**

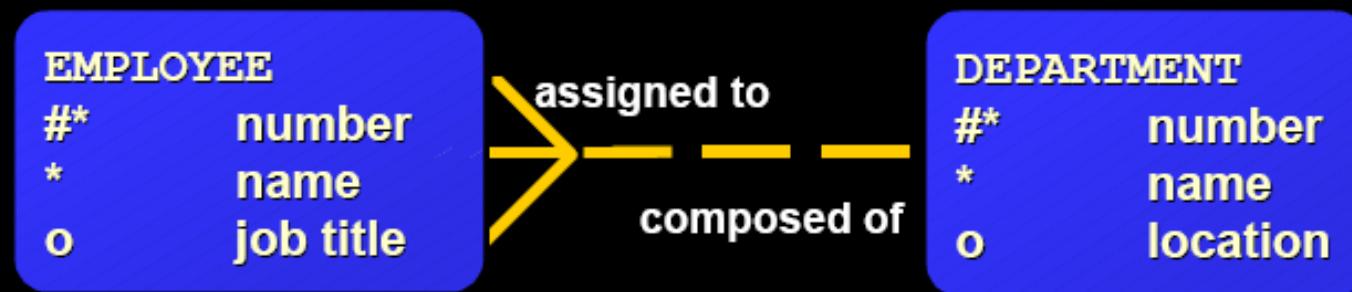
- Roll out the system to the users.
- Operate the production system. Monitor its performance, and enhance and refine the system.

# **Relational Database Concept**

- Dr. E.F. Codd proposed the relational model for database systems in 1970.
- It is the basis for the relational database management system (RDBMS).
- The relational model consists of the following:
  - Collection of objects or relations
  - Set of operators to act on the relations
  - Data integrity for accuracy and consistency

# Entity Relationship Model

- Create an entity relationship diagram from business specifications or narratives



- Scenario
  - “... Assign one or more employees to a department ...”
  - “... Some departments do not yet have assigned employees ...”

# Entity Relationship Modeling Conventions

## Entity

Soft box

Singular, unique name

Uppercase

Synonym in parentheses

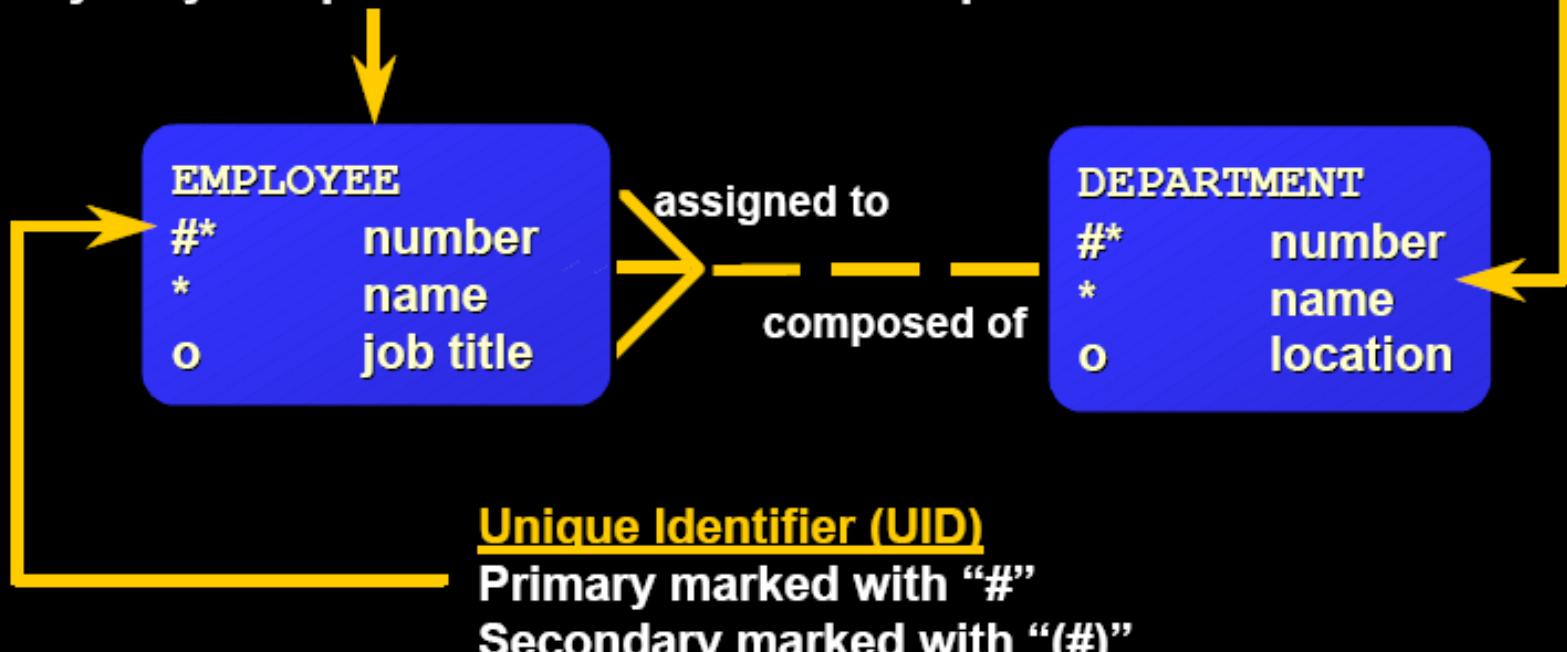
## Attribute

Singular name

Lowercase

Mandatory marked with “\*”

Optional marked with “o”



# Relating Multiple Tables

- Each row of data in a table is uniquely identified by a primary key (PK).
- You can logically relate data from multiple tables using 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	Emst	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

# Communicating with a RDBMS Using SQL

**SQL statement  
is entered.**

```
SELECT department_name  
FROM   departments;
```

**Statement is sent to  
Oracle Server.**

Oracle

DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

# Tables Used in the Course

## EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200
124	Kevin	Mourgos	KMOURCOS	650.123.5234	16-NOV-99	ST_MAN	5800
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600

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

DEPARTMENTS

GRADE	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

JOB\_GRADES

# Summary

- **The Oracle9*i* Server is the database for Internet computing.**
- **Oracle9*i* is based on the object relational database management system.**
- **Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.**
- **With the Oracle Server, you can store and manage information by using the SQL language and PL/SQL engine.**

# *LESSON 1*

# **Retrieving Data Using SQL Select Statements**

# Capabilities of SQL SELECT Statements

**Projection**


**Table 1**

**Selection**


**Table 1**


**Table 1**

**Join**


**Table 2**

# Capabilities of SQL SELECT Statements

- A SELECT statement retrieves information from the database.
  - Using a SELECT statement, you can do the following:
    1. **Projection:**
      - You can use the projection capability in SQL to choose the columns in a table that you want returned by your query.
      - You can choose as few or as many columns of the table as you require.
    2. **Selection:**
      - You can use the selection capability in SQL to choose the rows in a table that you want returned by a query.
      - You can use various criteria to restrict the rows that you see.
    3. **Joining:**
      - You can use the join capability in SQL to bring together data that is stored in different tables by creating a link between them.
- Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved.*

# Basic SELECT Statement

```
SELECT      * | { [DISTINCT] column|expression [alias], ... }  
FROM        table;
```

- **SELECT identifies *what columns***
- **FROM identifies *which table***

# Selecting All Columns

```
SELECT *  
FROM 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

8 rows selected.

# Selecting Specific Columns

```
SELECT department_id, location_id  
FROM departments;
```

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

# Writing SQL Statements

- **SQL statements are not case sensitive.**
- **SQL statements can be on one or more lines.**
- **Keywords cannot be abbreviated or split across lines.**
- **Clauses are usually placed on separate lines.**
- **Indents are used to enhance readability.**

# Arithmetic Expressions

**Create expressions with number and date data by using arithmetic operators.**

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

# Using Arithmetic Operators

```
SELECT last_name, salary, salary + 300  
FROM employees;
```

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300
...		
Hartstein	13000	13300
Fay	6000	6300
Higgins	12000	12300
Gietz	8300	8600

20 rows selected.

# Operator Precedence

\* / + -

- **Multiplication and division take priority over addition and subtraction.**
- **Operators of the same priority are evaluated from left to right.**
- **Parentheses are used to force prioritized evaluation and to clarify statements.**

# Operator Precedence

```
SELECT last_name, salary, 12*salary+100  
FROM employees;
```

LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100
Hunold	9000	108100
Ernst	6000	72100
...		

Hartstein	13000	156100
Fay	6000	72100
Higgins	12000	144100
Gietz	8300	99700

20 rows selected.

# Using Parentheses

```
SELECT last_name, salary, 12*(salary+100)  
FROM employees;
```

LAST_NAME	SALARY	12*(SALARY+100)
King	24000	289200
Kochhar	17000	205200
De Haan	17000	205200
Hunold	9000	109200
Ernst	6000	73200
...		

Hartstein	13000	157200
Fay	6000	73200
Higgins	12000	145200
Gietz	8300	100800

20 rows selected.

# Defining a Null Value

- A null is a value that is unavailable, unassigned, unknown, or inapplicable.
- A null is not the same as zero or a blank space.

```
SELECT last_name, job_id, salary, commission_pct  
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	

...

Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2

...

Gietz	AC_ACCOUNT	8300	
-------	------------	------	--

20 rows selected.

# Null Values in Arithmetic Expressions

**Arithmetic expressions containing a null value evaluate to null.**

```
SELECT last_name, 12*salary*commission_pct  
FROM employees;
```

LAST_NAME	12*SALARY*COMMISSION_PCT
King	
Kochhar	
...	
Zlotkey	25200
Abel	39600
Taylor	20640
...	
Gietz	
20 rows selected.	

# Defining a Column Alias

**A column alias:**

- **Renames a column heading**
- **Is useful with calculations**
- **Immediately follows the column name - there can also be the optional AS keyword between the column name and alias**
- **Requires double quotation marks if it contains spaces or special characters or is case sensitive**

# Using Column Aliases

```
SELECT last_name AS name, commission_pct comm  
FROM employees;
```

NAME	COMM
King	
Kochhar	
De Haan	
...	

20 rows selected.

```
SELECT last_name "Name", salary*12 "Annual Salary"  
FROM employees;
```

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000
...	

20 rows selected.

# Concatenation Operator

**A concatenation operator:**

- **Concatenates columns or character strings to other columns**
- **Is represented by two vertical bars (||)**
- **Creates a resultant column that is a character expression**

# Using the Concatenation Operator

```
SELECT      last_name || job_id AS "Employees"  
FROM        employees;
```

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP
HunoldIT_PROG
ErnstIT_PROG
LorentzIT_PROG
MourgosST_MAN
RajsST_CLERK
...

20 rows selected.

# Literal Character Strings

- A literal is a character, a number, or a date included in the SELECT list.
- Date and character literal values must be enclosed within single quotation marks.
- Each character string is output once for each row returned.

# Using Literal Character Strings

```
SELECT last_name || ' is a ' || job_id  
      AS "Employee Details"  
FROM   employees;
```

Employee Details
King is a AD_PRES
Kochhar is a AD_VP
De Haan is a AD_VP
Hunold is a IT_PROG
Ernst is a IT_PROG
Lorentz is a IT_PROG
Mourgos is a ST_MAN
Rajs is a ST_CLERK
...

20 rows selected.

# Duplicate Rows

The default display of queries is all rows, including duplicate rows.

```
SELECT department_id  
FROM employees;
```

DEPARTMENT_ID
90
90
90
80
80
80
50
50
50
...

20 rows selected.

# Eliminating Duplicate Rows

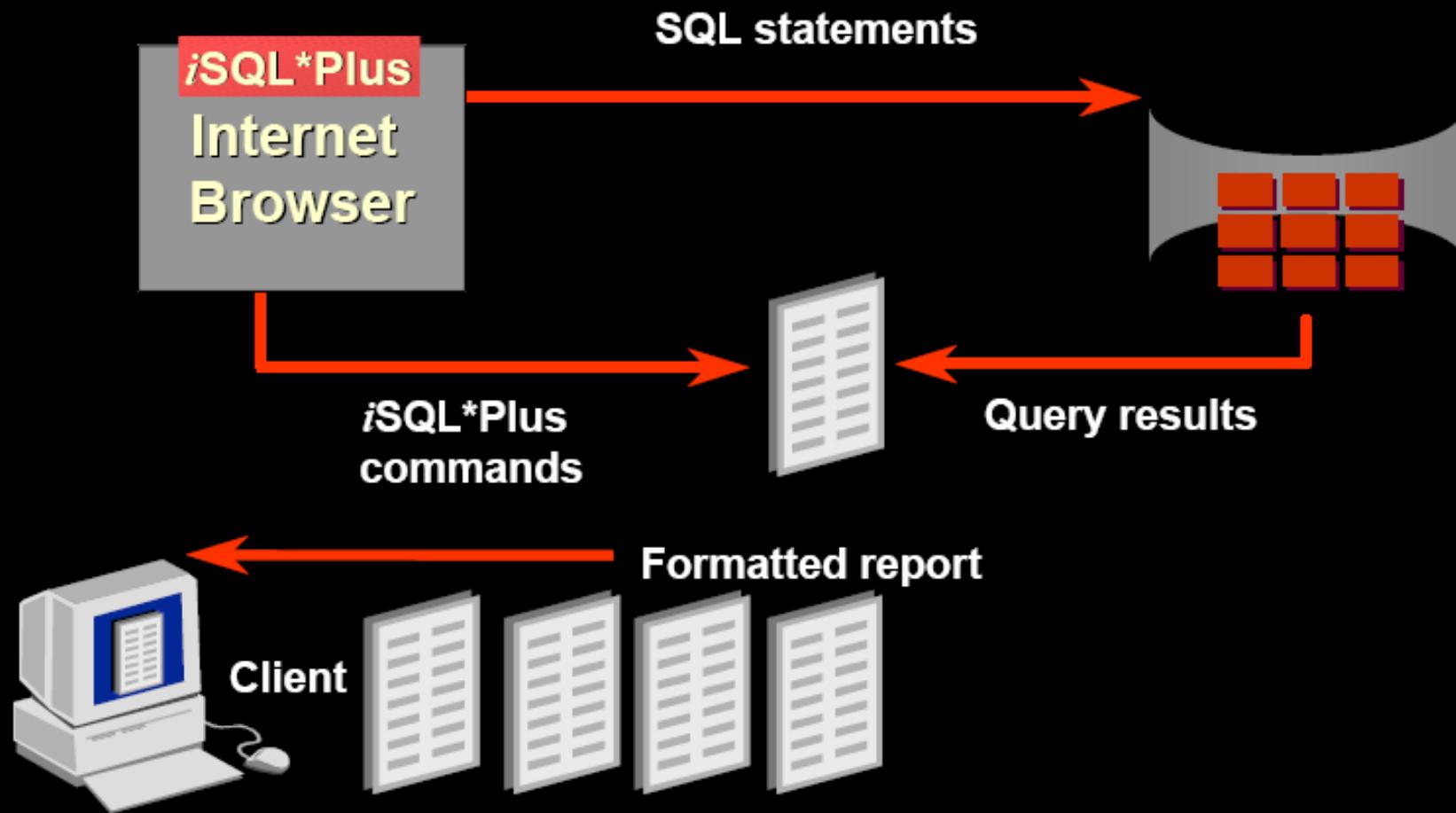
**Eliminate duplicate rows by using the DISTINCT keyword in the SELECT clause.**

```
SELECT DISTINCT department_id  
FROM employees;
```

DEPARTMENT_ID
10
20
30
40
50
60
70
80
90
100
110
120

8 rows selected

# SQL and *i*SQL\*Plus Interaction



## SQL and iSQL\*Plus

- SQL is a command language for communication with the Oracle server from any tool or application.
- Oracle SQL contains many extensions.
- iSQL\*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle server for execution and contains its own command language.

Copyright

Tanveer Khan, 2005 - 2016. All Rights Reserved

# **SQL Statements Versus *i*SQL\*Plus Commands**

## **SQL**

- A language
- ANSI standard
- Keyword cannot be abbreviated
- Statements manipulate data and table definitions in the database

**SQL  
statements**

## ***i*SQL\*Plus**

- An environment
- Oracle proprietary
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database
- Runs on a browser
- Centrally loaded, does not have to be implemented on each machine

***i*SQL\*Plus  
commands**

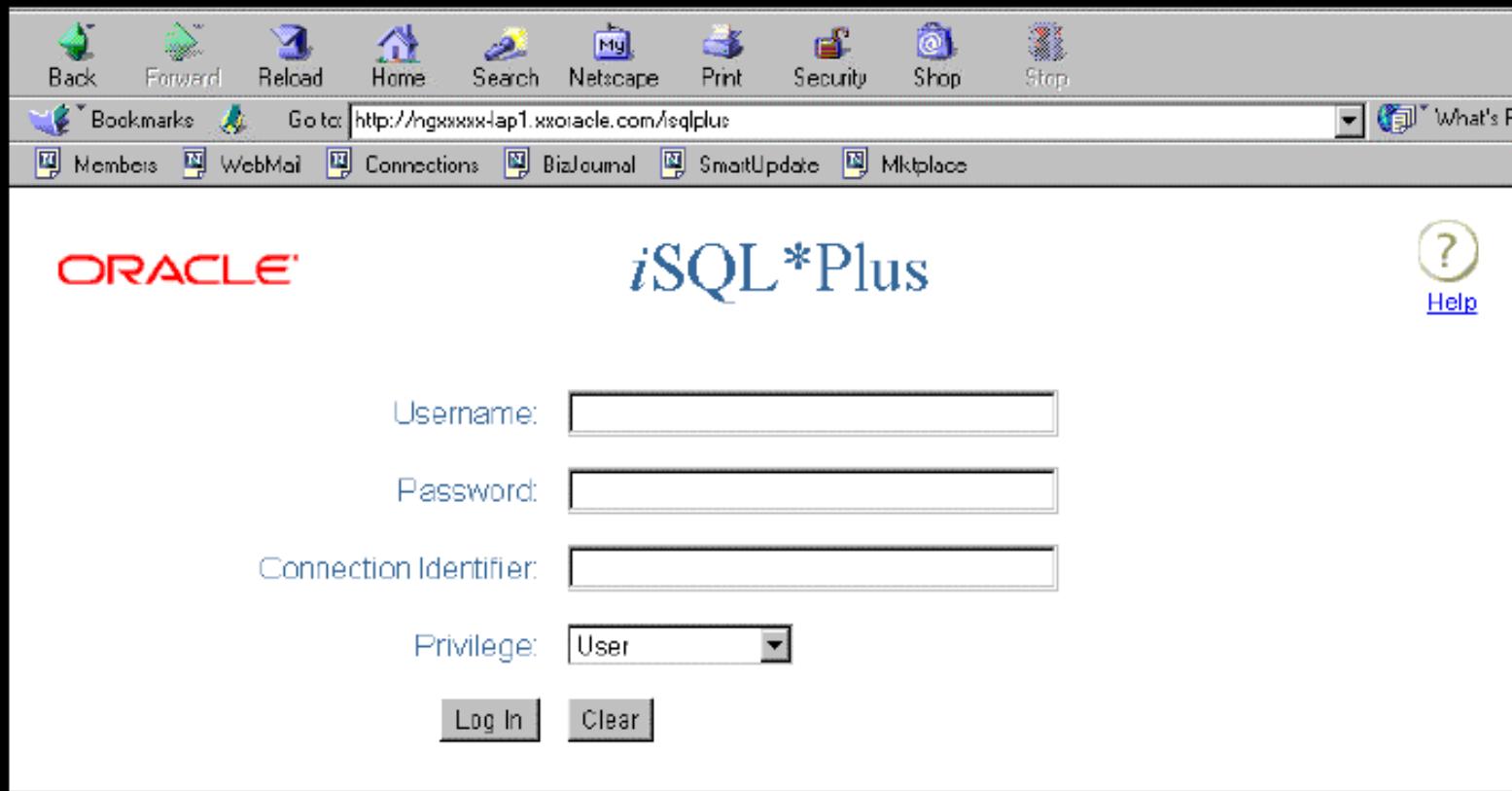
# **Overview of *i*SQL\*Plus**

**After you log into *i*SQL\*Plus, you can:**

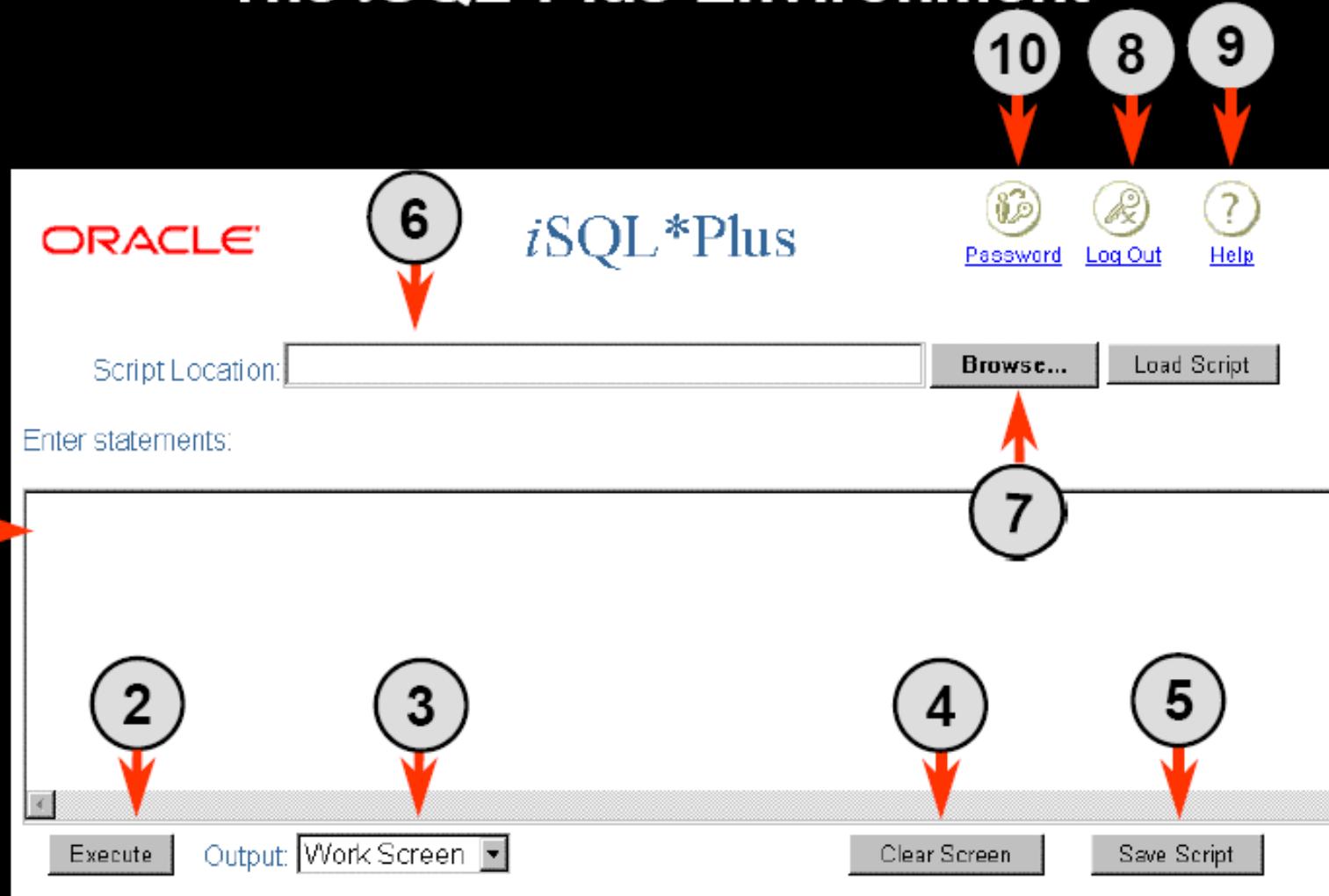
- Describe the table structure**
- Edit your SQL statement**
- Execute SQL from *i*SQL\*Plus**
- Save SQL statements to files and append SQL statements to files**
- Execute statements stored in saved files**
- Load commands from a text file into the *i*SQL\*Plus Edit window**

# Logging In to *iSQL\*Plus*

From your Windows browser environment:



# The *i*SQL\*Plus Environment



# Displaying Table Structure

**Use the *i*SQL\*Plus DESCRIBE command to display the structure of a table.**

```
DESC [RIBE] tablename
```

# Displaying Table Structure

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

# Review Lesson

1

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

## BASIC SELECT STATEMENT:

Select [distinct] {\*,column [alias],.....} from table;

- SELECT identifies *What* columns.
- FROM identifies *Which* table.

### Examples:

- 1) Select \* from EMP;
- 2) Select distinct ename from EMP;
- 3) Select ename “Employee Name” from EMP;

## ➤ Arithmetic Expressions:

Arithmetic expression can be created by applying the arithmetic operators on NUMBER and DATE data types.

### Example:

*Tanveer Khan, 2005 - 2016*

Select ename, sal, sal+300 “Commision” from Emp;

Arithmetic expressions containing a null values evaluates to null.

Select ename,12\*sal+comm from emp where ename = ‘KING’;

If any column value in an arithmetic expression is null, the result is null.

## Using Column Aliases:

- Renames a column heading.
- Is useful with calculations.
- Immediately follows column name; optional **AS keyword** b/w column name and alias.
- Requires double quotation marks if it contains spaces or special characters ~~or is case sensitive~~.
- An Alias cannot be used in the where clause.

### Example:

1) Select ename as NAME,sal SALARY,dname “Dept Name”;

### OUTPUT:

NAME	SALARY	Dept Name
-----	-----	-----

## Literal Character String:

```
Select ename || ' ' || 'is a' || ' ' || job AS "Employee Details" from emp;
```

### OUTPUT:

Employee Details

---

King is a President

Blake is a Manager.

## ➤ Eliminating Duplicates Rows:

The default display of queries is all rows including duplicate rows.

### Example:

```
select deptno from emp;
```

### OUTPUT:

DEPTNO
-----
10
30
10
20

10  
30  
10  
20

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

To eliminate duplicate rows in the result, include **DISTINCT** keywords in the SELECT clause immediately after the SELECT keyword.

## Example:

```
select distinct deptno from emp;
```

OUTPUT:

DEPTNO
-----
10
20
30

10  
20  
30

We can specify multiple columns after the DISTINCT qualifier. The distinct qualifier affects the selected columns, and the result represents a distinct combination of the columns.

## Example:

```
select distinct deptno, job from emp;
```

## OUTPUT:

DEPTNO	JOB
10	CLERK
10	MANAGER
20	ANALYST.

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# LESSON 2

# Restricting and Sorting Data

Tanveer Khan, 2005 - 2016. All Rights Reserved.

Copyright © Tanveer Khan

Restrictions

## Limiting Rows Selected:

- Restrict the rows returned by using the WHERE clause.
- Character String and Date values are enclosed in a single quotation marks.
- Character Values are case sensitive and Date values are format sensitive.
- The default date format is DD-MON-YY.

### Example:

- 1) select ename,job,deptno from emp where ename = 'KHALID';
- 2) select ename from emp where hiredate = '01-JAN-00';

# Limiting the Rows Selected

- Restrict the rows returned by using the WHERE clause.

```
SELECT      * | { [DISTINCT] column|expression [alias], ... }  
FROM        table  
[WHERE      condition(s)];
```

- The WHERE clause follows the FROM clause.

# Limiting Rows Using a Selection

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	Einst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50

...

20 rows selected.

**“retrieve all  
employees  
in department 90”**



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

# Using the WHERE Clause

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

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

# Character Strings and Dates

- Character strings and date values are enclosed in single quotation marks.
- Character values are case sensitive, and date values are format sensitive.
- The default date format is DD-MON-RR.

```
SELECT last_name, job_id, department_id  
FROM   employees  
WHERE  last_name = 'Whalen';
```

# Comparison Conditions

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
!= , <> , ^=	Not equal to

## ➤ Comparison Operators:

In addition to      = , > , >= , < , <= ,  $\diamond$  ;

we have five more operators.

OPERATOR	DESCRIPTION
BETWEEN...AND...	Between two values (inclusive)
IN(list)	Match any of a list of values
LIKE	Match a character pattern.
IS NULL	Is a null value.

## Examples:

- 1) Select ename, sal from emp where sal **BETWEEN** 1000 **AND** 1500;
- 2) Select empno,ename,sal from emp where ename **IN** ('Asif','Khuram');

- Use the **LIKE** operator to perform wildcard searches of valid search string values.
- Search condition can contain either literal characters or numbers
  - **%** denotes zero or many characters.
  - **\_** denotes one character.

- 3) Select ename from emp where ename **LIKE** 'S%';
- 4) Select ename from emp where ename **LIKE** '\_A%';
- 5) Select empno,ename from emp where ename **IS NULL**;

# Using Comparison Conditions

```
SELECT last_name, salary  
FROM employees  
WHERE salary <= 3000;
```

LAST_NAME	SALARY
Matos	2600
Vargas	2500

# Other Comparison Conditions

Operator	Meaning
BETWEEN . . . AND . . .	Between two values (inclusive), . . . AND . . .
IN (set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

# Using the BETWEEN Condition

Use the BETWEEN condition to display rows based on a range of values.

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



Lower limit      Upper limit

LAST_NAME	SALARY
Rajs	3600
Davies	3100
Matos	2600
Vargas	2500

# Using the IN Condition

**Use the IN membership condition to test for values in a list.**

```
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	5000	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.

# Using the LIKE Condition

- Use the **LIKE condition to perform wildcard searches of valid search string values.**
- Search conditions can contain either literal characters or numbers:
  - % denotes zero or many characters.
  - \_ denotes one character.

```
SELECT      first_name
FROM        employees
WHERE       first_name LIKE 'S%';
```

# Using the LIKE Condition

- You can combine pattern-matching characters.

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

LAST_NAME
Kochhar
Lorentz
Mourgos

- You can use the ESCAPE identifier to search for the actual % and \_ symbols.

# Using the NULL Conditions

Test for nulls with the IS NULL operator.

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

LAST_NAME	MANAGER_ID
King	

# Logical Conditions

Operator	Meaning
AND	Returns TRUE if <i>both</i> component conditions are true
OR	Returns TRUE if <i>either</i> component condition is true
NOT	Returns TRUE if the following condition is false

# Using the AND Operator

**AND requires both conditions to be true.**

```
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

# Using the OR Operator

OR requires either condition to be true.

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

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
124	Mourgos	ST_MAN	5800
149	Zlotkey	SA_MAN	10500
174	Abel	SA_REP	11000
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

8 rows selected.

# ➤ LOGICAL OPERATORS:

OPERATORS	DESCRIPTION
AND	Returns TRUE if both component conditions are TRUE.
OR	Returns TRUE if either component condition is TRUE.
NOT	Returns TRUE if the following component condition is TRUE.

## Examples:

➤ **AND** requires both conditions to be true.

1) Select empno,ename,sal from emp where ename = ‘KHALID’ **and** job = ‘MANAGER’;

➤ **OR** requires either condition to be true.

2) Select empno,ename from emp where ename = ‘KALID’ **or** job = ‘MANAGER’;

➤ **NOT** negate the condition

3)Select job from emp where name **NOT IN** (‘KHALID’,’ARSALAN’);

4)Select ename from emp where job **NOT LIKE** ‘CL%’;

# Using the NOT Operator

```
SELECT last_name, job_id  
FROM   employees  
WHERE  job_id  
      NOT IN ('IT_PROG', 'ST_CLERK', 'SA REP');
```

LAST_NAME	JOB_ID
King	AD_PRES
Kochhar	AD_VP
De Haan	AD_VP
Mourgos	ST_MAN
Zlotkey	SA_MAN
Whalen	AD_ASST
Hartstein	MK_MAN
Fay	MK_REP
Higgins	AC_MGR
Gietz	AC_ACCOUNT

10 rows selected

## ➤ Rules for Precedence:

ORDER EVALUATED	OPERATOR
1	NOT
2	AND
3	OR

We can override the rules of precedence by using parenthesis.

## Example:

1) Select ename,job,sal from emp where job = ‘SALESMAN’ OR job = ‘MANAGER’ AND sal > 40000;

Above Select statement will be executed as follows

“Select the row if an employee is a MANAGER and earns more than 40000      or      if the employee is a SALESMAN”.

➤ Use parenthesis to force priority

2) Select ename,job,sal from emp where (job = ‘SALESMAN’ OR job = ‘MANAGER’) AND sal > 40000;

Above Select statement will be executed as follows

“Select the row if an employee is a SALESMAN or MANAGER and earns more than 40000.

# Rules of Precedence

```
SELECT last_name, job_id, salary  
FROM   employees  
WHERE  job_id = 'SA_REP'  
OR     job_id = 'AD_PRES'  
AND    salary > 15000;
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Abel	SA_REP	11000
Taylor	SA_REP	8600
Grant	SA_REP	7000

# Rules of Precedence

Use parentheses to force priority.

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  (job_id = 'SA_REP'
OR      job_id = 'AD_PRES')
AND    salary > 15000;
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

## Sorting:

- Sort rows with the **ORDER BY** clause
  - **ASC**; ascending order, default.
  - **DESC**; descending order.
- **ORDER BY** clause comes last in the select statement.
- **NULL values** are displayed last for ascending sequences and first for descending sequences.

### Example:

Select ename,job,hiredate from emp order by hiredate desc;

- We can use the column alias in the ORDER BY clause

### Example:

Select ename, sal\*12 SALARY from emp order by SALARY;

- We can sort query results by more than one column

### Example:

Select ename,deptno,sal from emp ORDER BY deptno desc,sal desc;

# ORDER BY Clause

- Sort rows with the ORDER BY clause
  - ASC: ascending order, default
  - DESC: descending order
- The ORDER BY clause comes last in the SELECT statement.

```
SELECT    last_name, job_id, department_id, hire_date  
FROM      employees  
ORDER BY  hire_date ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Emst	IT_PROG	60	21-MAY-91

...

20 rows selected.

# Sorting in Descending Order

```
SELECT      last_name, job_id, department_id, hire_date  
FROM        employees  
ORDER BY    hire_date DESC ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
Zlotkey	SA_MAN	80	29-JAN-00
Mourgos	ST_MAN	50	16-NOV-99
Grant	SA_REP		24-MAY-99
Lorentz	IT_PROG	60	07-FEB-99
Vargas	ST_CLERK	50	09-JUL-98
Taylor	SA_REP	80	24-MAR-98
Matos	ST_CLERK	50	15-MAR-98
Fay	MK_REP	20	17-AUG-97
Davies	ST_CLERK	50	29-JAN-97
...			

20 rows selected.

# Sorting by Column Alias

```
SELECT employee_id, last_name, salary*12 annsal  
FROM employees  
ORDER BY annsal;
```

EMPLOYEE_ID	LAST_NAME	ANNSAL
144	Vargas	30000
143	Matos	31200
142	Davies	37200
141	Rajs	42000
107	Lorentz	50400
200	Whalen	52800
124	Mourgos	69600
104	Eust	72000
202	Fay	72000
178	Grant	84000
...		

20 rows selected.

# Sorting by Multiple Columns

- The order of ORDER BY list is the order of sort.

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

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3600
Davies	50	3100
Matos	50	2600
Vargas	50	2500
...		

20 rows selected.

- You can sort by a column that is not in the SELECT list.

## ➤ ORDER OF EXECUTION OF SELECT STATEMENT:

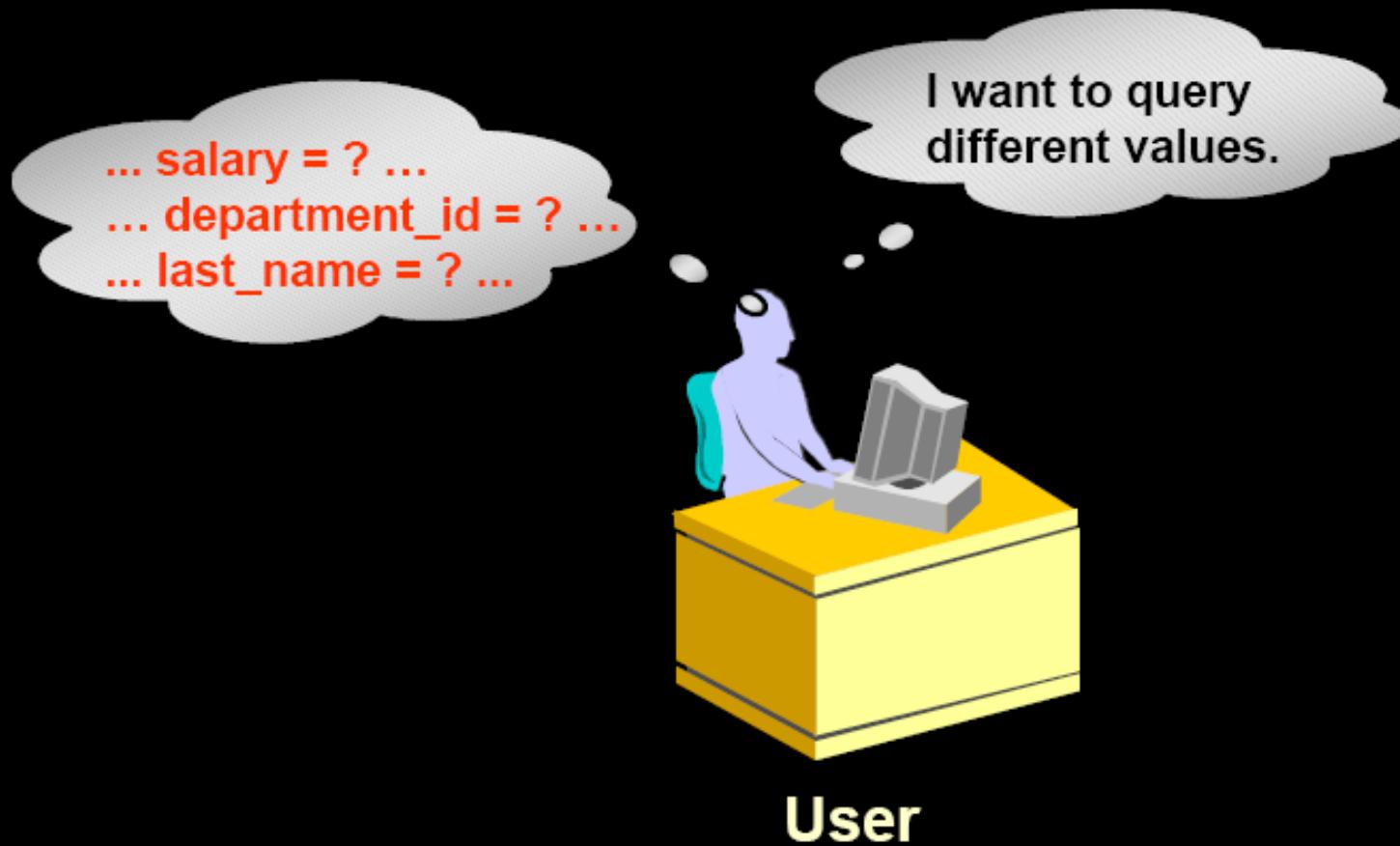
internally, the order of execution for a SELECT statement is as follows:

1. **FROM clause.**
2. **WHERE clause.**
3. **SELECT clause.**
4. **ORDER BY clause.**

Copyright

Tanveer Khan, 2005 - 2016. All Rights Reserved

# Substitution Variables



## Substitution Variables:

- The examples so far have been hard-coded, the user would trigger the report, and the report would run without further prompting.
- Using iSQLPlus, you can create reports that prompt the user to supply their own values to restrict the range of data returned by using substitution variables.
- You can embed substitution variables in a command file or in a single SQL statement.
- A substitution variable can be thought of as a container in which the values are temporarily stored. When the statement is run, the value is substituted.

# Substitution Variables

**Use iSQL\*Plus substitution variables to:**

- **Temporarily store values**
  - Single ampersand (&)
  - Double ampersand (&&)
  - DEFINE command
- **Pass variable values between SQL statements**
- **Dynamically alter headers and footers**

## 1) Single Ampersand(&)

To prompt the user every time at the execution of command for a value.

### Example 1:

```
Select empno,ename,sal  
from emp  
where empno = &employee_number;
```

### OUTPUT:

Enter value for employee\_number:

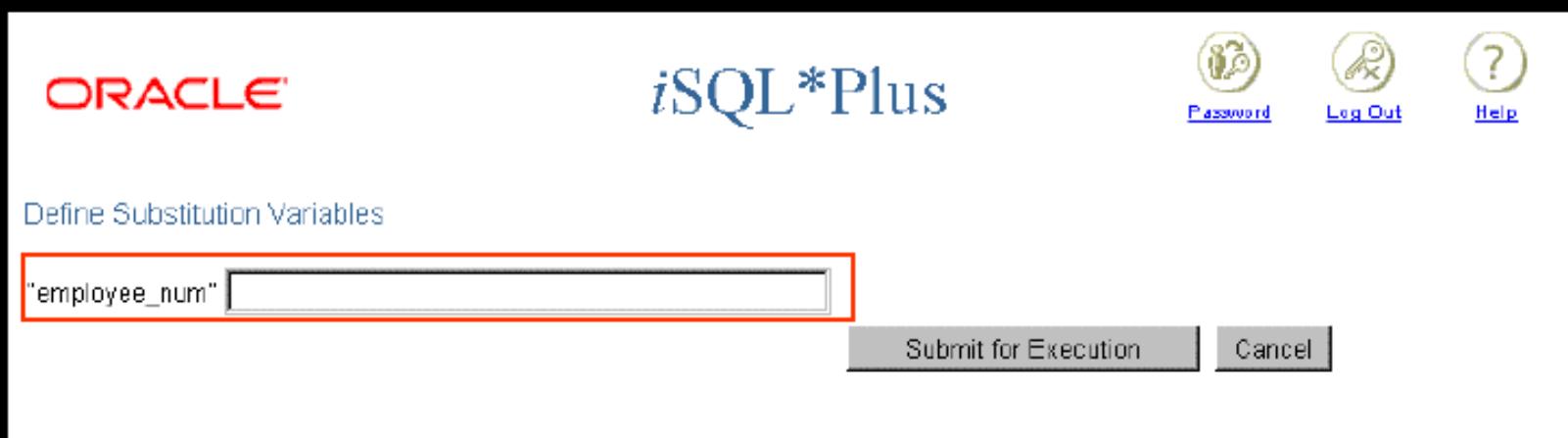
### Example 2:

```
Select empno,ename,sal  
from emp  
where job = upper('&emp_job');
```

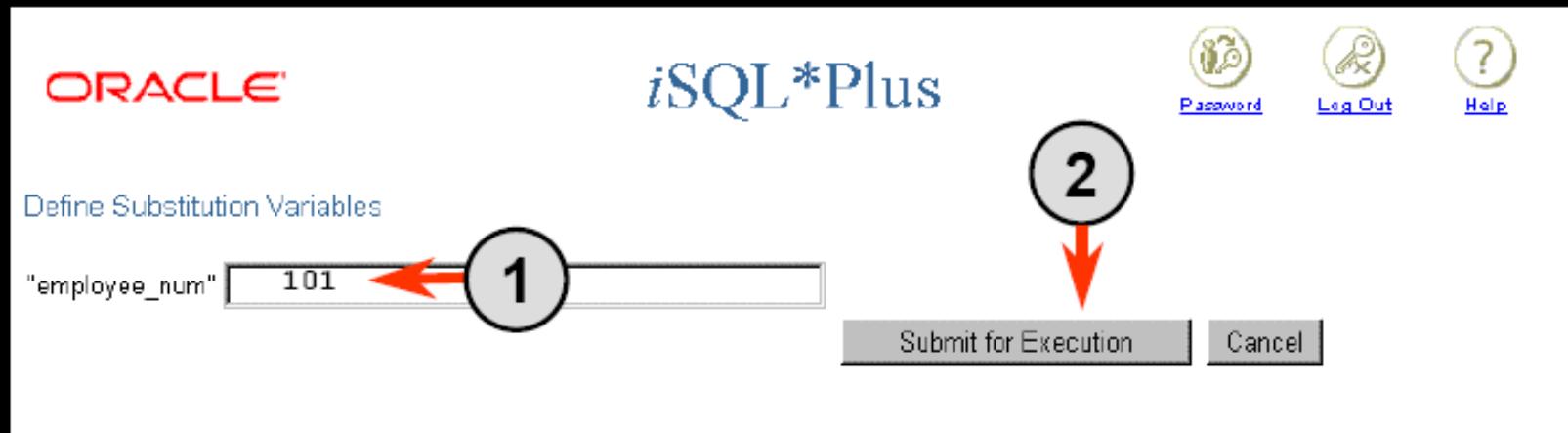
# Using the & Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value.

```
SELECT      employee_id, last_name, salary, department_id  
FROM        employees  
WHERE       employee_id = &employee_num ;
```



# Using the & Substitution Variable



```
old 3: WHERE employee_id = &employee_num  
new 3: WHERE employee_id = 101
```

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
101	Kochhar	17000	90

# Character and Date Values with Substitution Variables

Use single quotation marks for date and character values.

```
SELECT last_name, department_id, salary*12  
FROM employees  
WHERE job_id = '&job_title' ;
```

Define Substitution Variables

'job\_title'

LAST_NAME	DEPARTMENT_ID	SALARY*12
Hunold	60	108000
Ernst	60	72000
Lorentz	60	50400

Substitution variables are used to supplement the following

- WHERE condition
- ORDER BY clause
- Column expression
- Table name

**Example:**

```
SELECT empno, &column  
FROM &table  
WHERE &condition  
ORDER BY &order_by_column;
```

## 2) Double Ampersand (&&)

Use (&&), if you want to reuse the variable value without prompting the user each time.

### Example:

Select empno, ename, &&job From emp Order By &job;

## 3) DEFINE

It creates a CHAR datatype user variable. Use UNDEFINE command to delete it.

### SYNTAX:

DEFINE *variable* = *value*

### Example:

DEFINE column = job;

## Example:

Define col1 = empno;

Define col2=ename;

Define table=emp;

Define condition=1000;

SELECT &col1,&col2

FROM &table

WHERE &sal > &condition;

# Defining Substitution Variables

- You can **predefine variables using the iSQL\*Plus DEFINE command.**  
`DEFINE variable = value` creates a user variable with the CHAR data type.
- If you need to **predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the DEFINE command.**
- A **defined variable is available for the session**

# DEFINE and UNDEFINE Commands

- A variable remains defined until you either:
  - Use the UNDEFINE command to clear it
  - Exit iSQL\*Plus
- You can verify your changes with the DEFINE command.

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE          = "IT_PROG" (CHAR)
```

```
UNDEFINE job_title
DEFINE job_title
SP2-0135: symbol job_title is UNDEFINED
```

# Using the VERIFY Command

**Use the VERIFY command to toggle the display of the substitution variable, before and after iSQL\*Plus replaces substitution variables with values.**

```
SET VERIFY ON
SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num;
```

"employee\_num"

```
old    3: WHERE employee_id = &employee_num
new    3: WHERE employee_id = 200
```

# Summary

In this lesson, you should have learned how to:

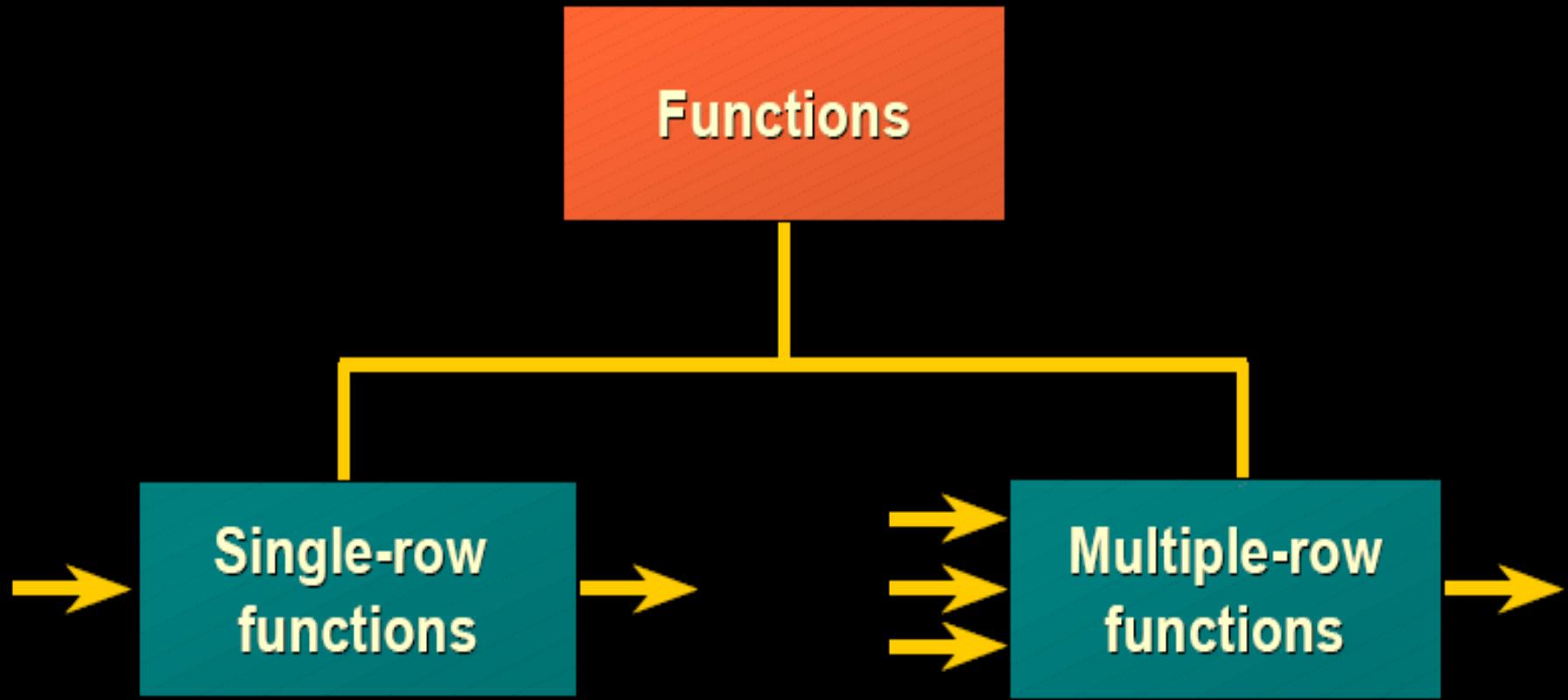
- Use the WHERE clause to restrict rows of output
  - Use the comparison conditions
  - Use the BETWEEN, IN, LIKE, and NULL conditions
  - Apply the logical AND, OR, and NOT operators
- Use the ORDER BY clause to sort rows of output

```
SELECT      * | { [DISTINCT] column|expression [alias],... }
FROM        table
[WHERE      condition(s)]
[ORDER BY   {column, expr, alias} [ASC|DESC]];
```

# **LESSON 3**

# **Using Single- Row Functions to Customize Output**

# Two Types of SQL Functions



# SQL Functions:

There are two types of SQL functions.

## 1) Single Row Functions:

- These functions operate of single rows only and return one result per row.
- Can be used in SELECT, WHERE and ORDER BY clause.

### Example:

```
select empno, lower(ename) from emp;
```

EMPNO	LOWER(ENAME)
7369	smith
7499	allen
7521	ward
7566	jones

## 2) Multiple Row Functions:

These functions manipulate group of rows to give one and only one result per group of rows.

### Example:

Select deptno,**sum(sal)** from emp **group by** deptno;

DEPTNO	SUM(SAL)
10	8750
20	10875
30	9400

# Types of Single Row Functions:

## 1) Character Functions:

Accept character input and can return both character and number values.

## 2) Number Functions:

Accept numeric input and returns numeric values.

## 3) Date Functions:

Operate on values of date data type. It always return a date value except the **MONTHS\_BETWEEN** function, which returns a number.

## 4) Conversion Functions:

Converts a value from one data type to another.

## 5) General Functions:

- NVL function.
- DECODE function etc.

# Character Functions

## Character functions

### Case-manipulation functions

LOWER  
UPPER  
INITCAP

### Character-manipulation functions

CONCAT  
SUBSTR  
LENGTH  
INSTR  
LPAD | RPAD  
TRIM  
REPLACE

## Character Function:

### 1) LOWER(column | expression):

Converts mixed case or upper case character string to lower case.

#### Example:

Select lower(job) from emp;

### 2) UPPER(Column | Expression):

Converts mixed case or lower case character string to upper case.

#### Example:

Select upper(job) from emp;

### 3) INITCAP(column | expression):

Converts first letter of each word to upper case and remaining letters to lower case.

#### Example:

Select initcap(job) from emp;

#### **4) CONCAT(col 1 | exp 1, col 2 | exp 2):**

Concatenate the first character value to the second character value.

#### **5) SUBSTR(col | exp, m , [n]):**

Returns specified character from the character value starting at character position **m** up to **n** characters.

#### **6) LENGTH(col | exp):**

Returns the number of characters in value.

#### **7) INSTR(col | exp, m):**

Return the numeric position of a named character.

### **Example:**

Select concat(ename,job), length(ename), instr(ename,'A') from emp  
where substr(job,1,5) = 'SALES';

## **8) LPAD(column|expression,n,'string'):**

Pads the character value right-justified to a total width of ***n*** character positions.

## **9) RPAD(column|expression,n,'string'):**

Pads the character value left-justified to a total width of ***n*** character positions.

## **10) TRIM(leading|trailing|both trim\_character from trim\_source):**

Enables you to trim heading or trailing characters (or both) from a character string. If trim\_source is character literal, you must enclose it in single quotes.

## **11) REPLACE(text,search\_string,replacement\_string):**

Searches a text expression for a character string and, if found, replaces it with a specified replacement string.

# Using Case Manipulation Functions

**Display the employee number, name, and department number for employee Higgins:**

```
SELECT employee_id, last_name, department_id  
FROM   employees  
WHERE  last_name = 'higgins';  
no rows selected
```

```
SELECT employee_id, last_name, department_id  
FROM   employees  
WHERE  LOWER(last_name) = 'higgins';
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

# Character-Manipulation Functions

These functions manipulate character strings:

Function	Result
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 TRIM (0 FROM 0009872348900) "TRIM Example" FROM DUAL;
- SELECT TRIM(leading 'A' from 'AMJADA') FROM DUAL;
- SELECT TRIM(TRAILING 'A' FROM 'AMJADA') FROM DUAL;

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

## NUMBER FUNCTION:

### 1) MOD(m,n):

Returns the remainder of  $m$  divided by  $n$ .

*Example:*

Select ename,sal,mod(sal,comm) from emp where job = 'SALESMAN';

### 2) ROUND(m,n):

Rounds the value  $m$  to specified  $n$  decimal places.

### 3) TRUNC(m,n):

Truncates the value  $m$  to specified  $n$  decimal places.

# Number Functions

- ROUND: Rounds value to specified decimal

ROUND (45.926, 2)  45.93

- TRUNC: Truncates value to specified decimal

TRUNC (45.926, 2)  45.92

- MOD: Returns remainder of division

MOD (1600, 300)  100

1. select round(45.269,2),trunc(45.269,2) from dual;

OUTPUT:

ROUND(45.269,2)	TRUNC(45.269,2)
-----------------	-----------------

-----	-----
45.27	45.26

2. select round(45.265,2),trunc(45.265,2) from dual;

OUTPUT:

ROUND(45.265,2)	TRUNC(45.265,2)
-----------------	-----------------

-----	-----
45.27	45.26

3. select round(45.263,2),trunc(45.263,2) from dual;

OUTPUT:

ROUND(45.263,2)	TRUNC(45.263,2)
-----------------	-----------------

-----	-----
45.26	45.26

4. select round(45.263),trunc(45.263) from dual;

OUTPUT:

ROUND(45.263)	TRUNC(45.263)
---------------	---------------

-----	-----
45	45

## Arithmetic with Dates:

- Add or subtract a number to or from a date for a resultant date value.
- SYSDATE is a function that returns:
  - Date
  - Time

### Example:

- 1) Select sysdate+10 from dual;
- 2) Select sysdate-10 from dual;

Dual is a dummy table you can use to view results from functions and calculations.

- Subtract two dates to find the number of days b/w those dates.
- Example:

Select sysdate – hiredate from emp;

## Date Functions:

1) **Months\_Between(date1,date2):**

Number of months b/w two months.

2) **Add\_Months(date,n):**

Add calendar months to date.

3) **Next\_day(date,'char'):**

Date of the next specified day of the week('char');

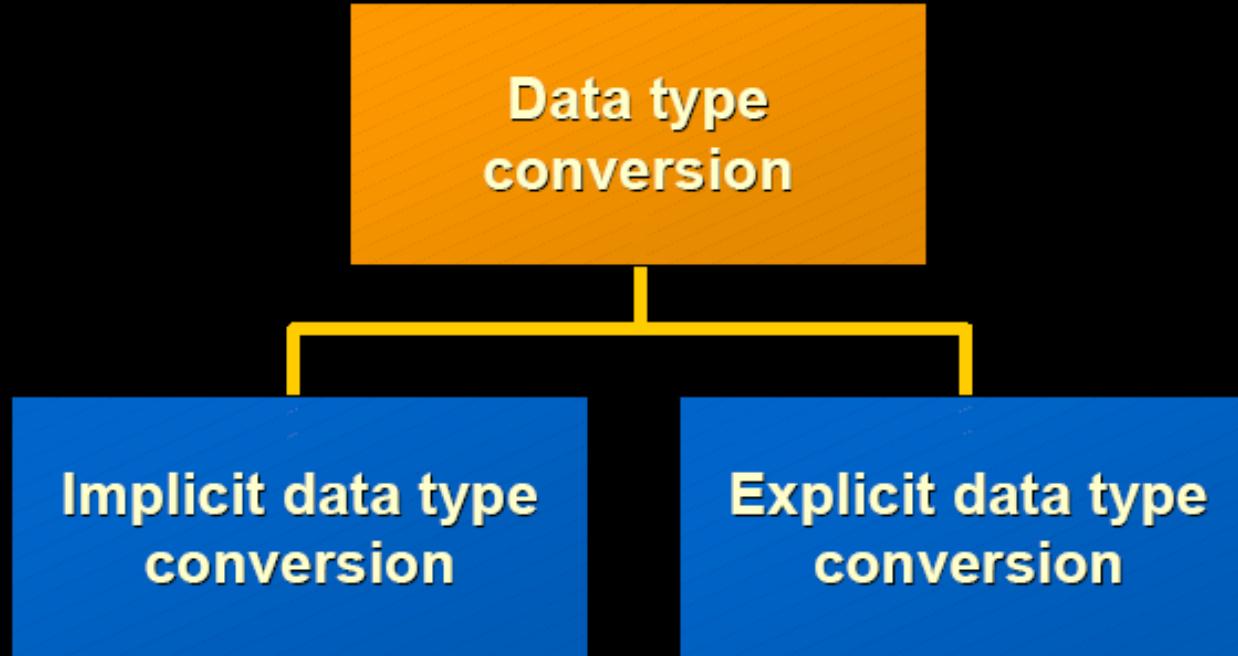
4) **Last\_day(date):**

Finds the date of the last date of the month that contains date.

**Example:**

```
select empno,hiredate,  
months_between(sysdate,hiredate),  
add_months(hiredate,6),  
next_day(hiredate,'FRIDAY'),  
last_day(hire_date)  
from emp;
```

# Conversion Functions



## Data Type Conversion Functions:

- In some cases, oracle server uses data of one data type where it expects data of a different data type.
- When this happens, oracle server can automatically convert the data to the expected data type.
- This data type conversion can be done *implicitly by oracle server*, or *explicitly by the user*.

### Implicit Data Type Conversion:

- Implicit data type conversions work according to the rules explained in the next two slides.

### Explicit Data Type Conversion:

- Explicit data type conversions are done by using the conversion functions.
- Conversion functions convert a value from one data type to another.
- Generally the form of function names follows the conversion *data type* to *data type*. This *first data type is the input data type*; the *last data type is the output*.

# Implicit Data Type Conversion

**For assignments, the Oracle server can automatically convert the following:**

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

## Example for Implicit Conversion:

Step-1: Create a table having fields of NUMBER, DATE and VARCHAR2 data types.

```
create table dummy(a number,b date,c varchar2(10));
```

Step-2: Now insert the values into the table which does not have same data types as that of table's attributes data types but must be compatible so that they may be converted implicitly.

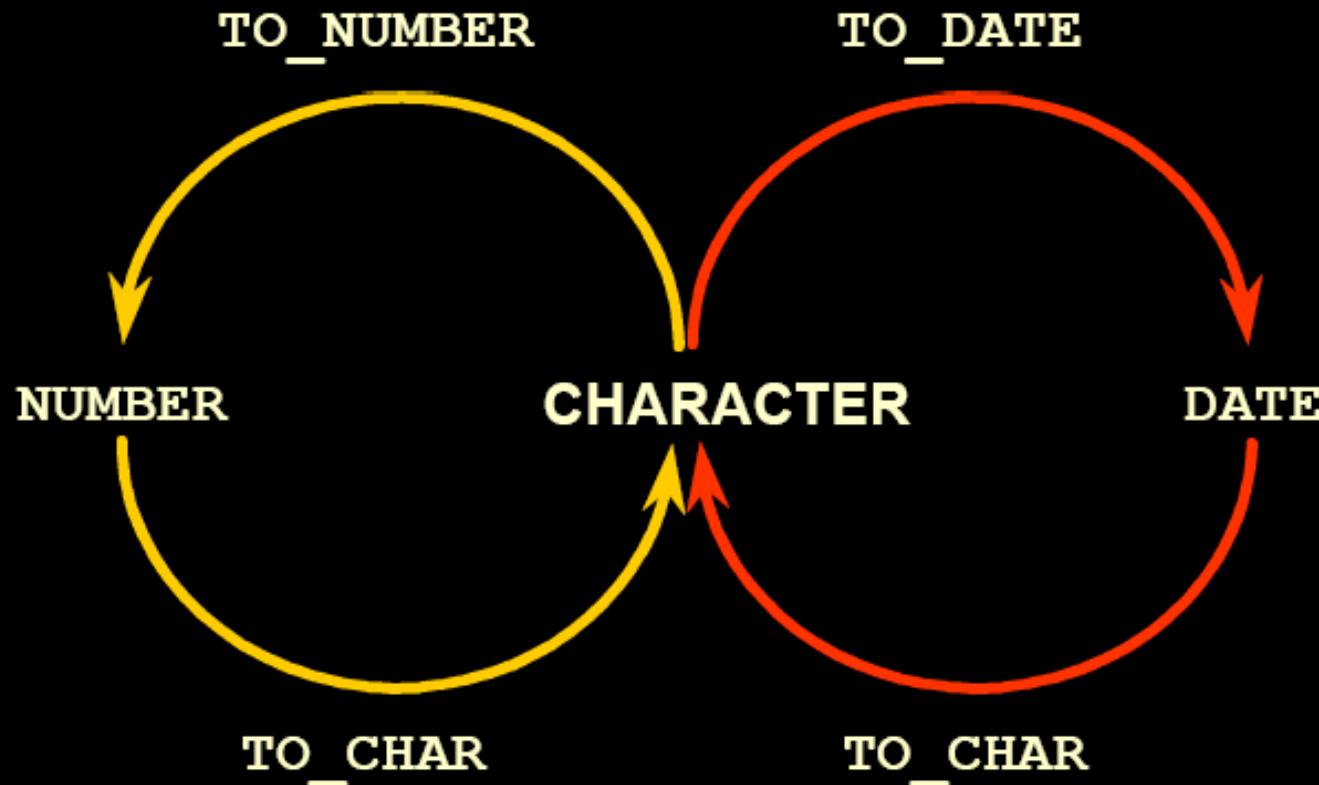
```
insert into dummy values('12','12-jun-2006',25);
```

Step-3: Now retrieve the values from the table.

```
select * from dummy;
```

**CONCLUSION:** Now, you will notice that besides 12 is inserted as a string value into the table attribute having data type NUMBER but still, we get no error. Similarly, for DATE and VARCHAR2 data types.

# Explicit Data Type Conversion



# Explicit Data Type Conversion Functions:

## 1) TO\_CHAR(number|date,'fmt'):

Converts a number or date value to a varchar2 character string with format 'fmt'.

### Example:

- Select empno,to\_char(hiredate,'mm/yy') Month\_hired from emp;
- Select to\_char(hiredate,'dd-month-yyyy') from emp;
- Select to\_char(sal,'\$99,999') from emp;
- Select tochar(17145,'\$099,999') from emp;

# Elements of the Date Format Model

YYYY	Full year in numbers
YEAR	Year spelled out
MM	Two-digit value for month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

# Elements of the Date Format Model

- **Time elements format the time portion of the date.**

HH24 :MI :SS AM

15:45:32 PM

- **Add character strings by enclosing them in double quotation marks.**

DD "of" MONTH

12 of OCTOBER

- **Number suffixes spell out numbers.**

ddsspth

fourteenth

## Example:

```
select to_char(sysdate,'fmddspth "of" month yyyy fmhh:mi:ss AM')  
from dual;
```

## Output:

```
TO_CHAR(SYSDATE,'FMDDSPTH"OF"MONTHYYYFMHH:M
```

---

seventh of june 2006 12:52:06 PM

- To remove padded blanks or to suppress leading zeros, use the fill mode ***fm*** element.

# Using the TO\_CHAR Function with Dates

```
SELECT last_name,  
       TO_CHAR(hire_date, 'fmDD Month YYYY')  
          AS HIREDATE  
FROM   employees;
```

LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999
...	

20 rows selected.

# Using the TO\_CHAR Function with Numbers

```
TO_CHAR(number, 'format_model')
```

**These are some of the format elements you can use with the TO\_CHAR function to display a number value as a character:**

9	Represents a number
0	Forces a zero to be displayed
\$	Places a floating dollar sign
.	Prints a decimal point
,	Prints a thousand indicator

# Using the TO\_CHAR Function with Numbers

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY  
FROM employees  
WHERE last_name = 'Ernst';
```

SALARY
\$6,000.00

## **2)TO\_NUMBER(char,’fmt’):**

Convert a character string to a number format.

### **Example:**

Select sal+to\_number(substr('\$100',2,3)) from emp;

## **3)TO\_DATE(char,’fmt’):**

Convert a character string to a date format.

### **Example:**

Select ename,hiredate from emp where hiredate = to\_date('Feb 22, 1981', 'Mon dd, yyyy');

## RR Date Format:

- The RR date format is similar to the YY element, but you can use it to specify different centuries.
- You can use the RR date format element instead of YY, so that the century of the return value varies according to the specified two digit year and the last two digits of the current year.

Current Year	Given Date	Interpreted (RR)	Interpreted (YY)
1994	27-OCT-95	1995	1995
1994	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017

# RR Date Format

Current Year	Specified Date	RR Format	YY Format
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

If two digits of the current year are:	If the specified two-digit year is:		
	0–49	50–99	
If two digits of the current year are:	0–49	The return date is in the current century	The return date is in the century before the current one
	50–99	The return date is in the century after the current one	The return date is in the current century

## Example of RR Date Format

To find employees hired prior to 1990, use the RR format, which produces the same results whether the command is run in 1999 or now:

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM   employees
WHERE  hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
```

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

- To find the employees who were hired prior to 1990, the RR format can be used.
- Since the year is now greater than 1990, the RR format interprets the year portion of the date from 1950 to 1999.

## Example:

- The following command, on the other hand, results in no rows being selected because the YY format interprets the year portion of the date in the current century.

```
SELECT ename, TO_CHAR(hiredate, 'DD-Mon-YY')  
FROM emp  
WHERE TO_DATE(hiredate, 'DD-Mon-yy') < '01-Jan-90';
```

## Example:

Suppose we have the following data in the table EMP

```
SQL> SELECT empno,to_char(hiredate,'dd-mon-yyyy')  
      FROM emp;
```

### OUTPUT

EMPNO	TO_CHAR(HIR)
7369	17-dec-1980
7499	20-feb-1981
7788	19-apr-1987
101	01-oct-2007

**Step-1:** Suppose we want to find out all the employees hired before 99

SQL>

```
SELECT empno,hiredate  
FROM emp  
WHERE hiredate < '01-dec-99';
```

## OUTPUT

EMPNO	TO_CHAR(HIR
7369	17-dec-1980
7499	20-feb-1981
7788	19-apr-1987

**Note:** All the records will be displayed except the last record which belongs to 2007.

**Reason:** Because according to the table, if the current year is 0-49 and specified year is 50-99, then century will be before the current one.

**Step-2:** This problem can be resolved by using the RR format or by mentioning the complete hiredate even with century.

## Option-1:

SQL>

```
SELECT empno, hiredate  
FROM emp  
WHERE hiredate < '01-dec-2099';
```

## OUTPUT

EMPNO	TO_CHAR(HIR
7369	17-dec-1980
7499	20-feb-1981
7788	19-apr-1987
101	01-oct-2007

## Option-2:

SQL> *SELECT empno,hiredate  
FROM emp  
WHERE to\_date(hiredate,'dd-mon-rr') < '01-dec-09';*

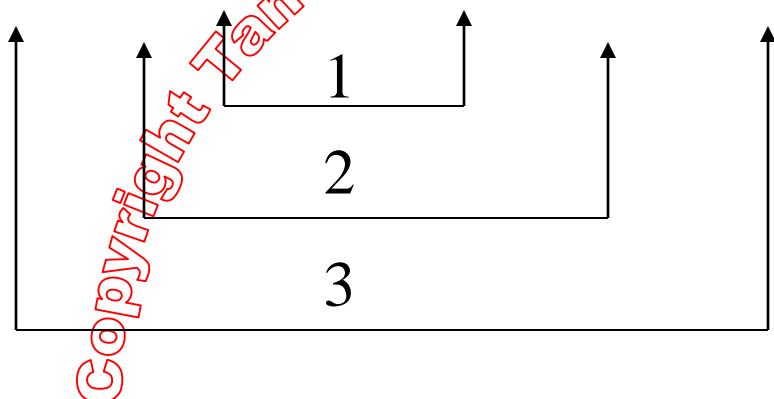
### OUTPUT

EMPNO	TO_CHAR(HIR
7369	17-dec-1980
7499	20-feb-1981
7788	19-apr-1987
101	01-oct-2007

## Nested Functions:

- Single row function can be nested to any level.
- Nested functions are evaluated from innermost level to the outermost level.

$F3(F2(F1(colour,arg1),arg2),arg3)$



Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# General Functions

**These functions work with any data type and pertain to using nulls.**

- NVL (expr1, expr2)
- NVL2 (expr1, expr2, expr3)
- NULLIF (expr1, expr2)
- COALESCE (expr1, expr2, . . . , exprn)

## NVL Function :

### Syntax:

NVL(expr1, expr2)

- If *expr1* is null, NVL returns *expr2*.
- The argument *expr1* can have any data type.
- Datatypes of *expr1* and *expr2* must match.

### Examples:

- 1) Select ename,sal,comm, (sal \* 12) + NVL(Comm,0) from emp;
- 2) Select nvl(job,'No Job Yet') from emp;
- 3) Select ename,NVL(to\_char(manager\_id),'No Manager')  
from employees where mgr is NULL;

## NVL2 Function :

### Syntax:

NVL2(expr1, expr2, expr3)

- If *expr1* is not null, NVL2 returns *expr2*. If *expr1* is null, NVL2 returns *expr3*.
- The argument *expr1* can have any data type.
- Datatypes of *expr1*, *expr2* and *expr3* must match.

### Examples:

- 1) Select ename,sal,comm, NVL2(Comm,sal+comm,sal) from emp;

## **NULLIF Function:**

### **Syntax:**

**NULLIF(expr1,expr2)**

- NULLIF function compares two expressions. If they are equal, the function returns **NULL** else function returns the first expression.

Copyright Tanveer Khan, 2005 - 2016

All Rights Reserved

# Using the NULLIF Function

```
SELECT first_name, LENGTH(first_name) "expr1",
       last_name, LENGTH(last_name) "expr2",
       NULLIF(LENGTH(first name), LENGTH(last name)) result
FROM   employees;
```

FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
Steven	6	King	4	6
Neena	5	Kochhar	7	5
Lex	3	De Haan	7	3
Alexander	9	Hunold	6	9
Bruce	5	Ernst	5	
Diana	5	Lorentz	7	5
Kevin	5	Mourgos	7	5
Trenna	6	Rajs	4	6
Curtis	6	Davies	6	
...				

20 rows selected.



## COALESCE Function:

- COALESCE is just like NVL function but this function can take multiples alternative values.
- If the first expression is not null, it returns that expression; otherwise, it does a COALESCE of the remaining expressions.

### Syntax:

COALESCE(expr1,expr2,.....,exprn)

In the syntax:

Expr1

returns this expression if it is not null.

Expr2

returns this expression if the first expression is null and this expression is not null.

Exprn

returns this expression if the preceding expressions are null.

# Using the COALESCE Function

```
SELECT      last_name,  
            COALESCE(commission_pct, salary, 10) comm  
FROM        employees  
ORDER BY    commission_pct;
```

LAST_NAME	COMM
Grant	.15
Zlotkey	.2
Taylor	.2
Abel	.3
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
...	

20 rows selected.

## Conditional Expressions:

- Two methods used to implement conditional processing (if-then-else logic) within the SQL statements are:
  - 1) CASE expression. (it complies with ANSI SQL)
  - 2) DECODE function. (it is specific to Oracle Syntax)

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# The CASE Expression

**Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:**

```
CASE expr WHEN comparison_expr1 THEN return_expr1  
          [WHEN comparison_expr2 THEN return_expr2  
          WHEN comparison_exprn THEN return_exprn  
          ELSE else_expr]  
END
```

# Using the CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
SELECT last_name, job_id, salary,  
       CASE job_id WHEN 'IT_PROG' THEN 1.10*salary  
                     WHEN 'ST_CLERK' THEN 1.15*salary  
                     WHEN 'SA REP' THEN 1.20*salary  
                     ELSE salary END      "REVISED SALARY"  
FROM employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3600	4020
...			
Gietz	AC_ACCOUNT	8300	8300
20 rows selected.			

## **DECODE Function:**

DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic used in various languages.

### **Syntax:**

```
DECODE(col/expression, search1, result1,  
           [search2,result2,],....  
           [default])
```

### **Example:**

```
select job,sal, decode(job, 'Analyst',sal * 10,  
           'Clerk',sal * 20,  
           sal) "Revised Salary" from emp;
```

# **LESSON 4**

# **Reporting**

# **Aggregated**

# **Data Using the**

# **Group Functions**

## Group Functions:

- Group Functions operate on set of rows to give one result per group.
- All group functions except COUNT(\*) ignore null values in the column.
- WHERE clause comes before GROUP BY clause.
- HAVING clause can come after GROUP BY clause.
- Alias names cannot be used in GROUP BY clause.
- GROUP BY columns are not necessarily be in the SELECT clause (example 3).
- GROUP BY clause can be applied on more than one columns (example 3).

## Syntax:

Select [column,] group\_function(column) from table  
[where condition] [group by column] [order by column];

## Example:

- 1) Select avg(sal),max(sal),sum(sal) from emp where job like 'SALES%';
- 2) Select deptno, sum(sal) from emp group by deptno;
- 3) Select sum(sal) from emp group by deptno,job;

## 1) AVG( [Distinct | All] n):

Average of  $n$  values, ignoring null values.

### Example:

Select avg(sal) from emp;

## 2) COUNT({\* | [distinct | all] exp }):

- Count the number of rows, where the expression evaluates to something other than null.
- Count(\*) returns the total number of row including duplicate and rows with null.

### Example:

- 1) Select count(\*) from emp where deptno = 30;
- 2) Select count(comm) from emp where deptno = 30;
- 3) Select count(distinct(deptno)) from emp;

### 3) MAX ([distinct | all] exp):

Maximum value of expression, ignoring null values.

### 4) MIN ([distinct | all] exp):

Minimum value of expression, ignoring null values.

### 5) STDDEV( [distinct | all] n):

2005 - 2016. All Rights Reserved

Standard deviation of n, ignoring null values.

### 6) SUM( [Distinct] | All] n):

Sum values of n, ignoring null values.

### 7) VARIANCE ( [Distinct | All] n):

Variance of n, ignoring null values.

Copyright  
Tanveer Khan

Copyright  
Tanveer Khan

## Guidelines:

- **DISTINCT** returns the nonduplicate values.
- **All** returns the duplicate values.
- All group functions except **COUNT(\*)** ignore null values. To substitute a value for null values, use the NVL,NVL2 or COALESCE functions.

## NVL Function with Group Functions:

NVL function forces group functions to include null values

### Example:

Select avg(nvl(comm,0)) from emp;

## Grouping by more than one column:

**Group By** clause can be applied on more than one columns. Lets say, we want to display the total salary being paid to each job title, within each department.

**Group By** clause can be used without using a group function in the Select list.

### Example:

Select deptno,job,sum(sal) from emp group by deptno,job;

## HAVING Clause:

- You cannot use the **WHERE** clause to restrict groups.
- You use the **HAVING** clause to restrict groups.

## Syntax:

Select column,group\_function **From** table  
[**Where** condition] [**Group By** column]  
[**Having** group condition] [**Order By** column];

## Example:

### WRONG:

```
SELECT deptno, AVG(sal)
FROM emp
WHERE AVG(sal) > 2000
GROUP BY deptno;
```

### CORRECT:

```
SELECT deptno, AVG(sal)
FROM emp
GROUP BY deptno
HAVING AVG(sal) > 2000
```

## Nested Group Functions:

**Group Functions** can be nested to a depth of two.

### Example:

Select max(avg(sal)) from emp group by deptno;

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# **LESSON 5**

# **Displaying Data From Multiple Tables**

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved.

# Objectives

**After completing this lesson, you should be able to do the following:**

- Write SELECT statements to access data from more than one table using equality and nonequality joins
- View data that generally does not meet a join condition by using outer joins
- Join a table to itself by using a self join

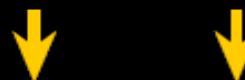
# Obtaining Data from Multiple Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700



EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
206	110	Accounting

➤ Sometimes you need to use data from more than one table. In the slide example, the report displays data from two separate tables.

✓ **EMPLOYEE\_ID** exists in the employee table.

✓ **DEPARTMENT\_ID** exists in both EMPLOYEES and DEPARTMENT tables.

✓ **LOCATION\_ID** exists in the department table.

➤ To produce the report, you need to link the EMPLOYEES and DEPARTMENTS tables and access data from both of them.

## Cartesian Product:

The Cartesian product operation defines a relation that is the concatenation of every tuple of relation **R** with every tuple of relation **S**.

The diagram shows two vertical rectangles representing relations. The left rectangle is labeled 'R' at the top and contains two rows labeled 'A' and 'B'. The right rectangle is labeled 'S' at the top and contains three rows labeled '1', '2', and '3'. A red diagonal watermark across the middle of the slide reads 'Copyright © Tanveer Khan, 2005 \* 2016. All Rights Reserved'.

$$\begin{array}{|c|} \hline R \\ \hline A \\ \hline B \\ \hline \end{array} \quad \begin{array}{|c|} \hline S \\ \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} = \quad \begin{array}{|c|c|} \hline & 1 \\ \hline A & 1 \\ \hline A & 2 \\ \hline A & 3 \\ \hline B & 1 \\ \hline B & 2 \\ \hline B & 3 \\ \hline \end{array}$$

- Cartesian product operation multiplies the two relations to defines another relation consisting of all possible pairs of tuples from two relations, therefore, if
  - one relation has **I tuples** and **N attributes**
  - another has **J tuples** and **M attributes**
  - the cartesian product relation will contain **(i\*j) tuples** with **(N+M) attributes.**

- A Cartesian product is formed when:
    - A join condition is omitted.
    - A join condition is invalid.
    - All rows in the first table are joined to all rows in the second table.
  - To avoid the Cartesian product, always include a valid join condition in a WHERE clause.
  - Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.
- Tanveer Khan, 2005 - 2016. All Rights Reserved*

## Example:

```
SELECT ename,dname  
FROM emp, dept;
```

# Generating a Cartesian Product

EMPLOYEES (20 rows)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS (8 rows)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
60	Shipping	1600
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

Cartesian  
product: →  
 $20 \times 8 = 160$  rows

EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700
...		

160 rows selected.

```
SELECT employee_id,department_id,location_id
FROM employees,departments
```

# Types of Joins

## Oracle Proprietary Joins (8i and prior):

- Equijoin
- Non-equijoin
- Outer join
- Self join

## SQL: 1999

## Compliant Joins:

- Cross joins
- Natural joins
- Using clause
- Full or two sided outer joins
- Arbitrary join conditions for outer joins

• oracle 10g database offers JOIN syntax that SQL:1999 compliant.

# Joining Tables Using Oracle Syntax

**Use a join to query data from more than one table.**

```
SELECT      table1.column,  table2.column  
FROM        table1,  table2  
WHERE       table1.column1 = table2.column2;
```

- **Write the join condition in the WHERE clause.**
- **Prefix the column name with the table name when the same column name appears in more than one table.**

## Defining Join

- When data from more than one table in the database is required, a join condition is used.
- Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, usually, **PRIMARY** and **FOREIGN** key columns.

## ➤ Guidelines:

- ✓ When writing a select statement that joins tables, precede the column name with the table name for clarity and to enhance database name.
- ✓ If the same column name appears in more than one table, the column name must be prefixed with the table name.
- ✓ To join  $n$  tables together, you need a minimum of  $n-1$  join conditions. For example, to join four tables, a minimum of three joins is required.

# What is an Equijoin?

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

Foreign key      Primary key

## Equi Join.

Equi Join are also called **Simple Join** or **Inner Join**. This type of join involves primary and foreign key compliments.

### Example:

Display the empno, ename and department location of all employees.

```
SELECT e.empno,e.deptno,d.loc  
FROM emp e,dept d  
WHERE e.deptno = d.deptno;
```

OR

```
SELECT e.empno,e.deptno,d.loc  
FROM  
emp e  
JOIN  
dept d  
ON e.deptno = d.deptno;
```

(or **INNER JOIN**)

# Additional Search Conditions Using the AND Operator

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Emst	60

...

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
60	IT
60	IT

...

```
SELECT last_name,E.department_id,department_name  
FROM employees E, departments D  
WHERE E.department_id = D.department_id  
AND last_name = 'MATOS';
```

## Qualifying Ambiguous Column Names:

- You need to qualify the names of the columns in the **WHERE clause** with the table name to avoid ambiguity.
- Without the table prefixes, the **DEPTNO column** could be from either the **DEPT table** or the **EMP table**. Therefore, it is necessary to add the table prefix to execute your query.
- If there are no common column names between the two tables, there is no need to qualify the columns.
- However using the table prefix improves performance, because you tell oracle server exactly where to find the columns.

```
SELECT empno,ename,deptno,dname  
FROM emp,dept  
WHERE emp.deptno =dept.deptno;
```

ERROR at line 1:

ORA-00918: column ambiguously defined

Joining after removing error:

```
SELECT empno,ename,dept.deptno,dname  
FROM emp,dept  
WHERE emp.deptno =dept.deptno;
```

# Using Table Aliases

- Simplify queries by using table aliases.
- Improve performance by using table prefixes.

```
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;
```

## Table Aliases

- Qualifying column names with tables names can be very time consuming, particularly if table names are lengthy.
- You can use table aliases instead of tables names.
- Just as *column aliases* gives a column another name, a *table alias* gives a table another name.

### Guidelines:

- Table aliases can be up to 30 characters in length, but shorter is better.
- The table alias is valid only for the current SELECT statement.

# Joining More than Two Tables

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90
Hunold	60
Ernst	60
Lorentz	60
Mourgos	60
Rajs	50
Davies	50
Matos	50
Vargas	50
Zlotkey	80
Abel	80
Taylor	80

20 rows selected.

DEPARTMENTS

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

LOCATIONS

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

- To join  $n$  tables together, you need a minimum of  $n-1$  join conditions. For example, to join three tables, a minimum of two joins is required.

```
SELECT e.last_name, d.department_name, l.city  
FROM employees e, departments d, locations l  
WHERE e.department_id = d.department_id  
AND d.location_id = l.location_id;
```

# Non-Equijoins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3600
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600
...	

20 rows selected.

JOB\_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000



**Salary in the EMPLOYEES table must be between lowest salary and highest salary in the JOB\_GRADES table.**

- A non-equijoin is a join condition containing something other than an equality (=) operator.

<b>EMPNO</b>	<b>ENAME</b>	<b>SAL</b>
101	A	1000
102	B	2000
103	C	3000

<b>GRADE</b>	<b>LOSALE</b>	<b>HISALE</b>
1	700	1500
2	1501	2500
3	2501	3500

### Example:

Select e.ename, e.sal, s.grade  
 from emp e, salgrade s  
 where e.sal between s.loosal and s.hisal;

### OUTPUT:

<b>ENAME</b>	<b>SAL</b>	<b>GRADE</b>
A	1000	1
B	2000	2
C	3000	3

## Outer Join.

- Missing rows can be returned if an outer join operator is used in the join condition.
- The operator is a plus sign enclosed in parenthesis (+).
- It is placed **on the side of the join that is deficient** in information.

Empno	Deptno	Ename
101	10	Khalid
102	20	Rashid

Deptno	Dname
10	Accounts
20	Marketing
30	Operations

Empno	Deptno	Dname
101	10	Accounts
102	20	Marketing
	30	Operations

## Example:

Suppose DEPT table has deptno = 30 but there is no employee in EMP table who belong to deptno = 30 but we want to display that dept also then

```
SELECT e.empno,e.deptno,d.dname  
FROM emp e,dept d  
WHERE e.deptno(+) = d.deptno;
```

OR

```
SELECT e.empno,e.deptno,d.dname  
FROM  
emp e  
RIGHT OUTER JOIN  
dept d  
ON e.deptno = d.deptno;
```

## Self Join:

EMPNO	ENAME	MGR
101	SAJID	
102	KHALID	101
103	OMER	104
104	WASEEM	102

Suppose that we want to find out the **KHALID's Manager name**, then

- First we have to find out the KHALID record.
- Now we will see the **KHALID's Manager number** in the MGR Column which is 101.
- Now we will find the name of the manager with  $\text{EMPNO} = 101$ .

In this process, we have to look in the table twice.

Now, from query point of view, we have to join the table to itself.  
This is called **Self Join**.

```
SELECT worker.ename "Worker Name", manager.ename "Manager  
Name"  
FROM emp worker , emp manager  
WHERE worker.mgr = manager.empno;
```

Copyright

Tanveer Khan, 2005 - 2016 All Rights Reserved

# Joining Tables Using SQL: 1999 Syntax

**Use a join to query data from more than one table.**

```
SELECT      table1.column, table2.column
FROM        table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
    ON (table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
    ON (table1.column_name = table2.column_name)];
```

CROSS JOIN	Returns a Cartesian product from two tables.
NATURAL JOIN	Join two tables on the same column name.
JOIN table USING col_name	Performs an EQUI JOIN based on the column name
JOIN table ON table1.col_name = table2.col_name	Performs an EQUI JOIN based on the condition in the ON clause.
LEFT/RIGHT/FULL OUTER	

# Creating Cross Joins

- The **CROSS JOIN clause produces the cross-product of two tables.**
- This is the same as a **Cartesian product between the two tables.**

```
SELECT last_name, department_name  
FROM employees  
CROSS JOIN departments ;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
...	

160 rows selected.

# Creating Natural Joins

- **The NATURAL JOIN clause is based on all columns in the two tables that have the same name.**
- **It selects rows from the two tables that have equal values in all matched columns.**
- **If the columns having the same names have different data types, an error is returned.**

## Creating Natural Joins:

- It was not possible to do a join without explicitly specifying the columns in the corresponding tables in prior releases of Oracle.
- In oracle 10g,it is possible to let the join be completed automatically based on columns in the two tables which have matching data types and names, using the key word NATURAL JOIN.
- If the columns have same name, but different data types, the NATURAL JOIN syntax causes an error.

# Retrieving Records with Natural Joins

```
SELECT department_id, department_name,  
       location_id, city  
FROM   departments  
NATURAL JOIN locations ;
```

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
30	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

8 rows selected.

# Exercise:

Step1: create table emp1(empno number primary key);

Step2: create table emp2(empno varchar2(5) primary key);

Step3: create table emp3(eno number primary key);

Step4: create table emp4(empno number primary key);

Step5: insert into emp1 values(101);

Step6: insert into emp1 values(102);

Step7: insert into emp2 values('101');

Step8: insert into emp3 values(101);

Step9: insert into emp4 values(101);

Step10: insert into emp4 values(103);

Step11: select \* from emp1,emp2 where emp1.empno = emp2.empno;

EMPNO	EMPNO
-----	-----
101	101

Step12: select \* from emp1,emp3 where emp1.empno = emp3.eno;

EMPNO	ENO
-----	-----
101	101

Step13: select \* from emp1 **natural join** emp3;

EMPNO	ENO
-----	-----
101	101
102	101

Step14: select \* from emp1 natural join emp4

EMPNO
-----
101

# Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, the NATURAL JOIN clause can be modified with the USING clause to specify the columns that should be used for an equijoin.
- Use the USING clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The NATURAL JOIN and USING clauses are mutually exclusive.

## The USING clause:

- Natural Joins use all columns with matching names and data types to join the tables.
- The **USING clause** can be used to specify only those columns that should be used for an equi join.
- The columns referenced in the USING clause should not have a qualifier (table name or alias) anywhere in the SQL statement.
- This statement is invalid because the LOCATION\_ID is qualified with the table alias name in the where clause:

```
SELECT l.city,d.department_name
```

```
FROM locations l JOIN departments d USING (location_id)  
WHERE d.location_id = 1400
```

# Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- To specify arbitrary conditions or specify columns to join, the ON clause is used.
- The join condition is separated from other search conditions.
- The ON clause makes code easy to understand.

## Example:

```
SELECT empno,ename,dname  
FROM emp JOIN dept  
ON emp.deptno = dept.deptno  
AND emp.deptno > 10  
WHERE ename like '%S%'
```

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# Creating Three-Way Joins with the ON Clause

```
SELECT employee_id, city, department_name  
FROM   employees e  
JOIN   departments d  
ON     d.department_id = e.department_id  
JOIN   locations l  
ON     d.location id = l.location id;
```

EMPLOYEE_ID	CITY	DEPARTMENT_NAME
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping
...		

19 rows selected.

## THREE-WAY JOINS:

- A three-way join is a join of three tables.
- In SQL:1999 compliant syntax, joins are performed from left to right so the first join to be performed is EMPLOYEES JOIN DEPARTMENTS
- The first join condition can reference columns in EMPLOYEES and DEPARTMENTS but cannot reference columns in LOCATIONS.
- The second join condition can reference columns from all three tables.
- In Three-way equijoin:

```
SELECT empno,city,dname  
FROM employees e,departments d,locations l  
WHERE e.deptno = d.deptno  
AND d.location_id = l.location_id;
```

- In USING clause:

```
SELECT empno,city,dname  
FROM employees e  
JOIN  
departments d  
USING (deptno)  
JOIN  
location l  
USING (location_id);
```

## 1. INNER JOIN:

The join of two table returning only matched rows is called INNER JOIN.

## 2. LEFT OUTER JOIN:

A join between two tables that returns the results of the inner join as well as unmatched rows of the left table is called LEFT OUTER JOIN.

## 3. RIGHT OUTER JOIN:

A join between two tables that returns the results of the inner join as well as unmatched rows of the right table is called RIGHT OUTER JOIN.

## 4. FULL OUTER JOIN:

A join between two tables that returns the results of an inner join as well as the results of a left and right outer joins is called FULL OUTER JOIN.

## LEFT OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
---		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

This query retrieves all rows in the *EMPLOYEES table*, which is the **LEFT TABLE** if there is no match in the *DEPARTMENTS table*.

## RIGHT OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
King	90	Executive
Kochhar	90	Executive
...		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.



This query retrieves all rows in the *DEPARTMENTS table*, which is the **RIGHT TABLE** if there is no match in the *EMPLOYEES table*.

## FULL OUTER JOIN

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

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
		Contracting

21 rows selected.

This query retrieves all rows in the **DEPARTMENTS table** even there is no match in the **EMPLOYEES table**.

It also retrieves all rows in the **EMPLOYEES table** even there is no match in the **DEPARTMENTS table**.

# Additional Conditions

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e JOIN departments d  
ON     (e.department_id = d.department_id)  
AND    e.manager_id = 149 ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
174	Abel	00	00	2500
176	Taylor	80	80	2500

# **LESSON 6**

# **Using Sub Queries to Solve Queries**

Copyright Tanveer Khan, 2005 - 2016. All rights reserved.

# Objectives

**After completing this lesson, you should be able to do the following:**

- **Describe the types of problem that subqueries can solve**
- **Define subqueries**
- **List the types of subqueries**
- **Write single-row and multiple-row subqueries**

# Using a Subquery to Solve a Problem

**Who has a salary greater than Abel's?**

Main Query:



Which employees have salaries greater  
than Abel's salary?



Subquery



What is Abel's salary?

## Sub Query:

Suppose you want to find out “Which employee have a salary greater than Jones Salary?”.

To solve this problem, you need two queries

- 1) What is the salary of Jones?
- 2) Who earns more than that amount?

This problem can be solved by sub query.

### Syntax:

```
SELECT column FROM table WHERE expression operator  
(SELECT column FROM table);
```

- The subquery (inner query) executes once before main query.
- The result of the subquery is used by the main query (outer query).
- Subquery can be placed in a number of SQL clauses:
  - ✓ WHERE clause.
  - ✓ HAVING clause.
  - ✓ FROM clause.

# Using a Subquery

```
SELECT last_name  
FROM   employees 11000  
WHERE  salary >  
       (SELECT salary  
        FROM   employees  
        WHERE  last_name = 'Abel');
```

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

## Example:

Select ename from emp where sal > (select sal from emp where ename = ‘JONES’);

## Guidelines:

- Enclose subqueries in parenthesis.
- Place subqueries on the right of the comparison operator.
- Use single row operators ( $>$ ,  $<$ ,  $\geq$ ,  $\leq$ ,  $\neq$ ) with single row subqueries.
- Use multiple row operators (IN, ANY, ALL) with multiple row subqueries.

## Same Query Through SELF JOIN:

- We can do the same job of sub queries through some other query techniques.
- Like in this scenario, we can get the same result through SELF JOIN.

### Example:

```
SELECT a.ename,a.sal,j.ename,j.sal  
FROM emp j,emp a  
WHERE a.sal > j.sal  
AND j.ename = 'JONES';
```

# Types of Subqueries

- Single-row subquery



- Multiple-row subquery



## Types of Subqueries:

### 1) Single-Row Subquery:

Queries that return only one row from the inner select statement.

#### Example:

Display the names of employees whose job title is the same as that of employee 7369.

Select ename,job from emp where job = (select job from emp where empno = 7369);

# Single-Row Subqueries

- **Return only one row**
- **Use single-row comparison operators**

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

## Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM employees
WHERE job_id = ST_CLERK
      (SELECT job_id
       FROM employees
       WHERE employee_id = 141)
AND salary > 2600
      (SELECT salary
       FROM employees
       WHERE employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

- A SELECT statement can be considered as a query block.
- The above example displays employees whose JOB\_ID is the same as that of employee 141 and whose salary is greater than that of employee 143.
- The example consists of three query blocks: one outer query and two inner queries.
- The inner query blocks are executed first, producing the query results ST\_CLERK and 2600, respectively. The out query is then processed and uses the values returned by the inner queries to complete its search condition.

# Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary  
FROM   employees  
WHERE  salary =  
       (SELECT MIN(salary)  
        FROM   employees);
```

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

- Group functions can be used in sub queries

# The HAVING Clause with Subqueries

- The Oracle server executes subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

```
SELECT      department_id, MIN(salary)
FROM        employees
GROUP BY    department_id
HAVING      MIN(salary) > 2500
(SELECT MIN(salary)
  FROM   employees
  WHERE  department_id = 50);
```

- You can use subqueries not only in WHERE clause, but also in HAVING clause.

## Trace out the error:

```
Select empno,ename from emp where sal =  
(select min(sal) from emp group by deptno)
```

Query returns more than one rows which is not possible with single row queries.

## Correct:

```
Select empno,ename from emp where sal IN  
(select min(sal) from emp group by deptno)
```

## 2) Multiple-Row Subquery:

- Queries that return more than one row from the inner select statement.

### Example:

MANAGER

CLERK

- Use multiple-row comparison operators

OPERATOR	MEANING
IN	Equals to any member in the list.
ANY	Compare value to each value by the subquery.
ALL	Compare value to every value returned by the subquery.

## Example:

### ➤ Wrong:

Select empno , ename , job  
from emp

where sal < **IN** (Select sal from emp where job = ‘CLERK’);

### ➤ Correct:

Select empno , ename , job  
from emp

where sal **IN** (Select sal from emp where job = ‘CLERK’);

### ➤ Correct:

Select empno , ename , job  
from emp

where sal < **ANY** (Select sal from emp where job = ‘CLERK’);

➤ **IN** operator **cannot** be used along with a **single row** operators.

➤ **ANY** , **ALL** operators can be used along with a **single row** operators.

# Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees      9000, 6000, 4200
WHERE  salary < ANY
       (SELECT salary
        FROM   employees
        WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500
...			

10 rows selected.

# Using the ALL Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL
        (SELECT salary
         FROM   employees
         WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

9000, 6000, 4200

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2800
144	Vargas	ST_CLERK	2500

## Multiple-Column Subquery:

Queries that return more than one columns from the inner select statement.

### Syntax:

```
SELECT column,column, ..... FROM table  
WHERE (column,column,...) IN  
(SELECT column,column,.... FROM table WHERE condition);
```

### Example:

Display the order number, product number and quantity of any item in which the product number and quantity match both the product number and quantity of an item in order 605 but the order number should not be 605.

```
SELECT ordid,prodid,qty  
FROM item  
WHERE (prodid, qty)  
IN  
(SELECT prodid,qty FROM item WHERE ordid = 605)  
AND  
ordid <> 605;
```

## NULL Values in Subqueries:

```
SELECT worker.ename FROM emp worker  
WHERE worker.empno NOT IN (SELECT manager.mgr FROM emp  
manager);
```

- In the above SQL statement, we want to display all the employees who do not have any subordinate or they are not managers. Logically this is a correct statement but it returns no row.
- The reason is that there are many employees who have no manager so the query will return null values for several rows and if any of the result of subquery is null it will make the whole query wrong.

## CORRECTION:

```
select e.ename from emp e where e.empno not in  
(select m.mgr from emp m where mgr is not null);
```

## CONCLUSION:

- Whenever null values are likely to be part of the resultant set of a subquery, don't use the **NOT IN** operator. The **NOT IN** operator is equivalent to **!=ALL**.
- NULL value as part of the resultant set of a subquery will not be a problem if you are using the **IN** operator. The **IN** operator is equivalent to **=ANY**.

### Example:

To display the employee names who have subordinates.

```
SELECT worker.ename FROM emp worker
```

```
WHERE worker.empno IN (SELECT manager.mgr FROM emp  
manager);
```

## **Subquery in the FROM clause:**

- Subquery in the **FROM** clause is just like the **VIEW**.

### **Example:**

Display the employees' name , deptno and salaries of all the employees who make more than the average salary of their department.

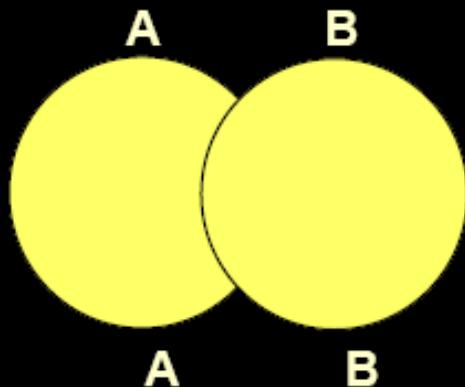
```
SELECT a.ename,a.sal,a.deptno,b.salavg  
FROM  
emp a,  
(SELECT deptno,avg(sal) salavg FROM emp GROUP BY deptno) b  
WHERE  
a.deptno = b.deptno AND a.sal > b.salavg;
```

# *LESSON 7*

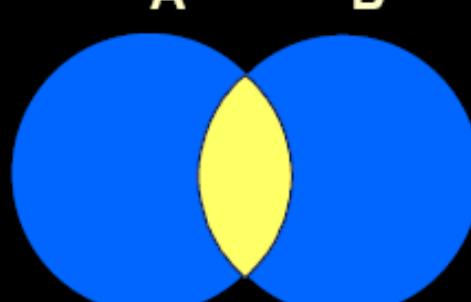
## **Using Set Operators**

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

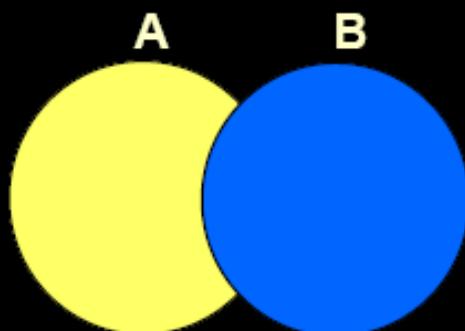
# The SET Operators



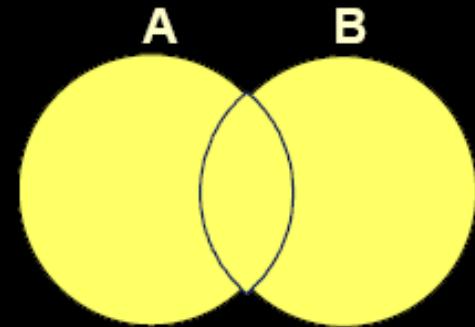
**UNION/UNION ALL**



**INTERSECT**



**MINUS**



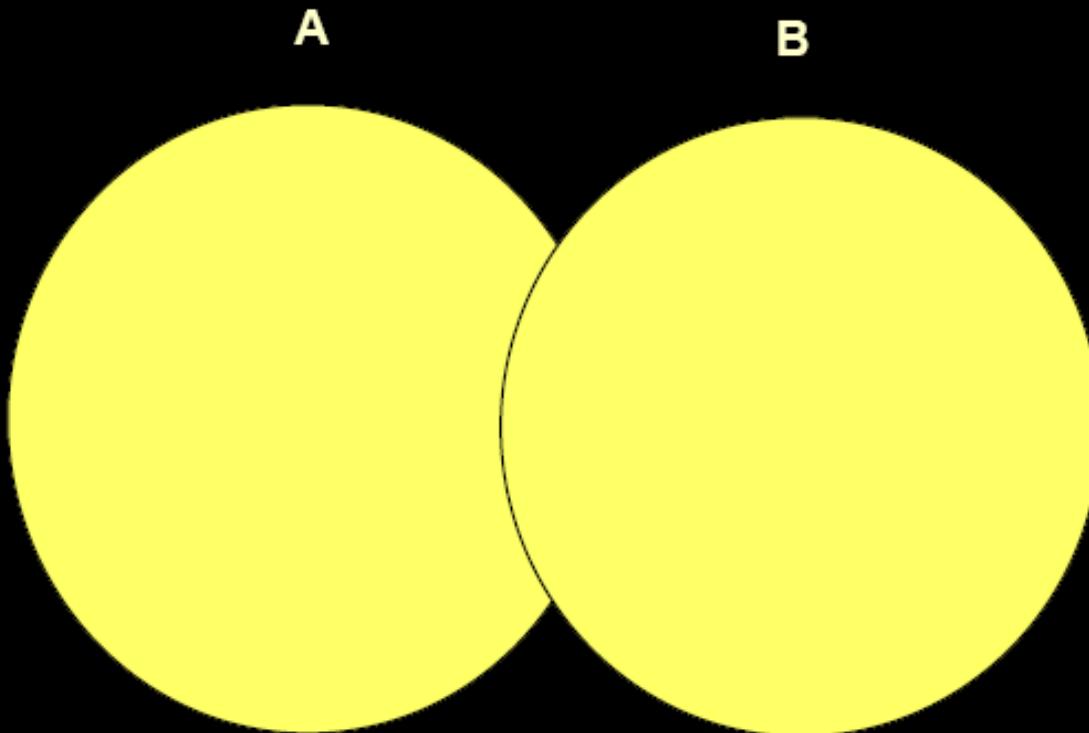
*Yellow color represents the result*

## The SET operator

- The SET operators combine the results of two or more component queries into one result.
- Queries containing the SET operators are called **Compound Queries**.
- All set operators have equal precedence.
- INTERSECT and MINUS operators are not ANSI SQL compliant. They are Oracle Specific.

Operator	Returns
<b>UNION</b>	All distinct rows selected by EITHER query, <b>eliminating duplicate values</b>
<b>UNION ALL</b>	All rows selected by EITHER query, <b>including all duplicates</b> .
<b>INTERSECT</b>	All distinct rows selected by BOTH queries.
<b>MINUS</b>	All distinct rows that are selected by the FIRST SELECT statement and not selected in the SECOND SELECT statement.

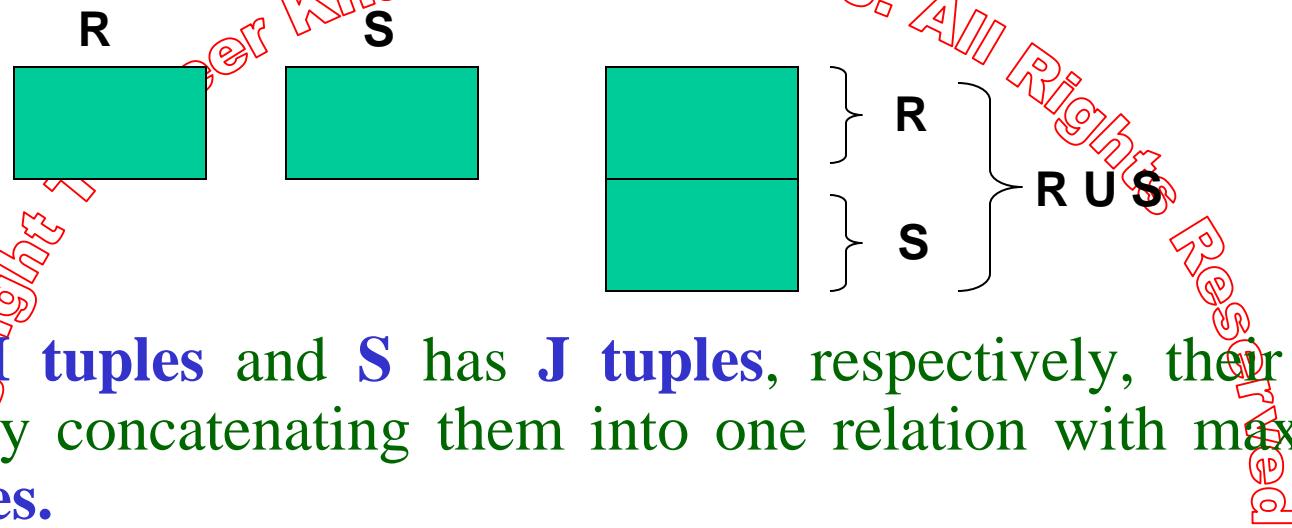
# The UNION Operator



**The UNION operator returns results from both queries after eliminating duplications.**

## 1) Union

The union of two relations **R** and **S** defines a relation that contains all the tuples of **R**, or **S**, or both (**R and S**), duplicates tuples being eliminated.



- If **R** has **I tuples** and **S** has **J tuples**, respectively, their union is obtained by concatenating them into one relation with maximum of **(I+J) tuples**.
- R and S must be union compatible.

## Guidelines for Union

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query.
- The names of the columns need not be identical.
- The output is sorted in ascending order of the first column of the SELECT clause.
- NULL values are not ignored during duplicate checking.

Copyright © Tanveer Khan, 2005 - 2016. All rights reserved

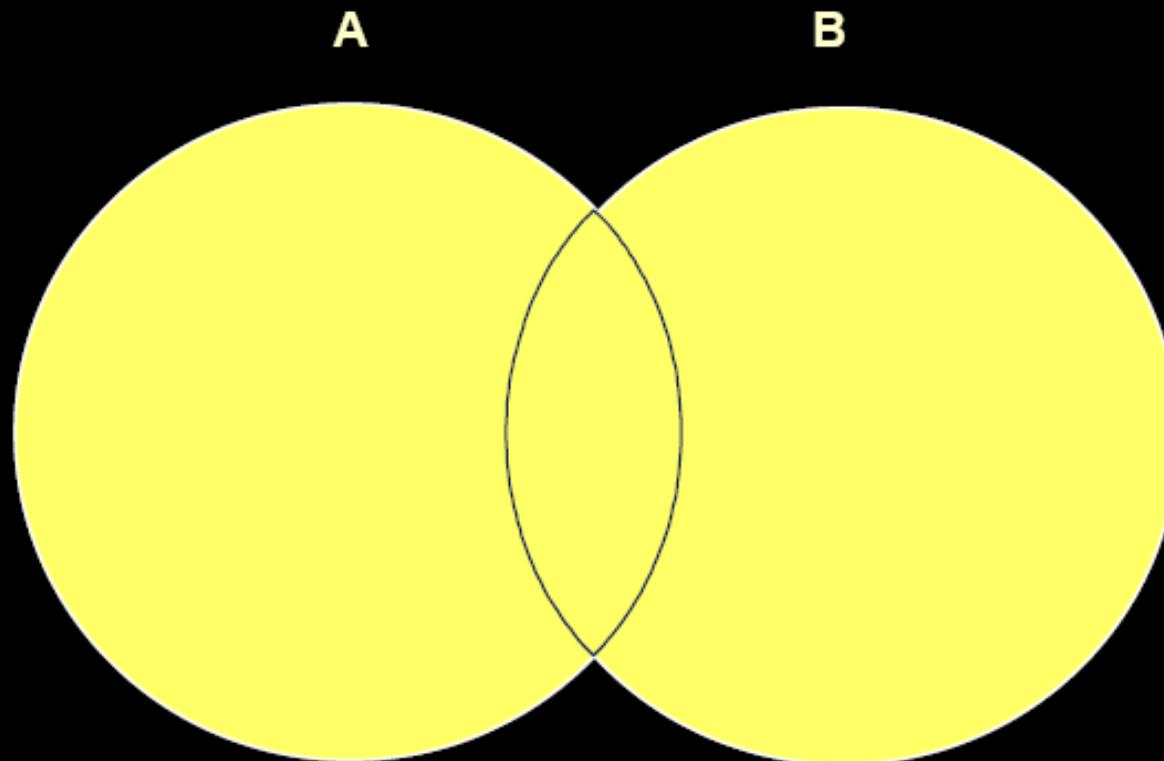
# Using the UNION Operator

Display the current and previous job details of all employees. Display each employee only once.

```
SELECT employee_id, job_id  
FROM   employees  
UNION  
SELECT employee_id, job_id  
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AC_ACCOUNT
...	
200	AC_ACCOUNT
200	AD_ASST
...	
205	AC_MGR
206	AC_ACCOUNT

# The UNION ALL Operator



**The UNION ALL operator returns results from both queries, including all duplications.**

## 2) Union All:

- UNION ALL returns all rows from multiple queries.

### GUIDELINES:

- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- DISTINCT keyword cannot be used.
- With the exceptions of the above, the guidelines for UNION and UNION ALL are the same.

# Using the UNION ALL Operator

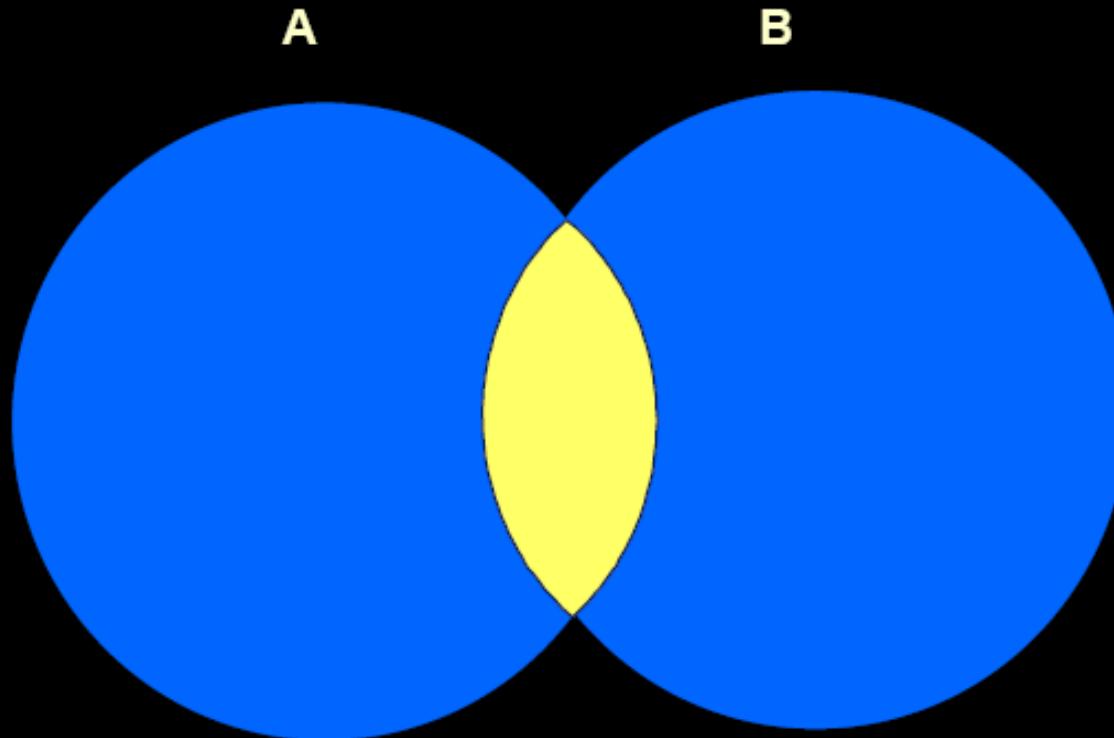
Display the current and previous departments of all employees.

```
SELECT employee_id, job_id, department_id  
FROM   employees  
UNION ALL  
SELECT employee_id, job_id, department_id  
FROM   job_history  
ORDER BY employee_id;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
100	AD_PRES	90
101	AD_VP	90
...		
200	AD_ASST	10
200	AD_ASST	90
200	AC_ACCOUNT	90
...		
205	AC_MGR	110
206	AC_ACCOUNT	110

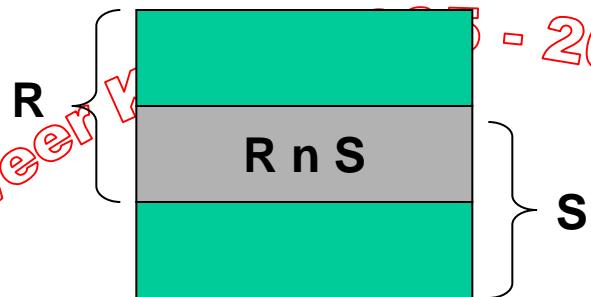
30 rows selected.

# The INTERSECT Operator



### 3) Intersection:

The intersection operation defines a relation consisting of the set of all tuples that are common in both R and S.



#### Guidelines:

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query.
- The names of the columns need not be identical.
- INTERSECT does not ignore NULL values.

# Using the INTERSECT Operator

**Display the employee IDs and job IDs of employees who currently have a job title that they held before beginning their tenure with the company.**

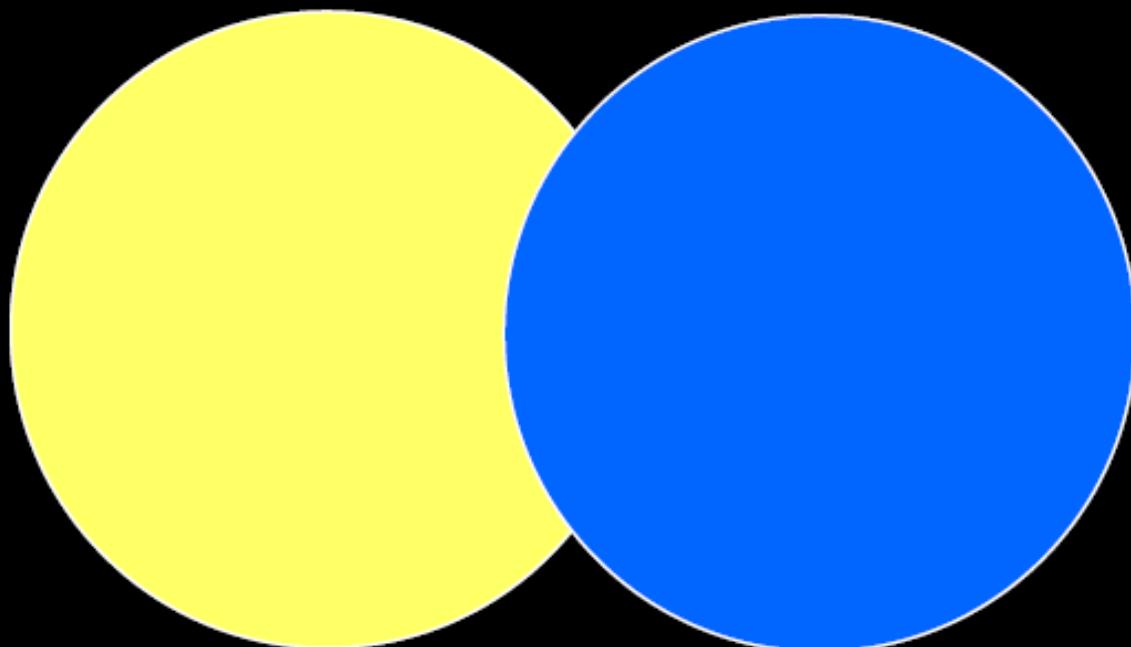
```
SELECT employee_id, job_id  
FROM   employees  
INTERSECT  
SELECT employee_id, job_id  
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

# The MINUS Operator

A

B



$A - B$

## 4) Minus:

Use the MINUS operator to return rows returned by the first query that are not present in the second query.



### Guidelines:

- The number of columns and the datatypes of the columns being selected must be identical in all the SELECT statements used in the query.
- The names of the columns need not be identical.
- All of the columns in the WHERE clause must be in the SELECT clause for the MINUS operator to work.

## Example:

- Suppose PRODUCT\_MASTER is the table having all the items inventory.
- Suppose ORDER\_DETAILS is the table having all the ordered items inventory.

Display all the product\_no of non-moving items in the product master table.

```
select product_no from product_master  
minus  
select product_no from order_details;
```

- The output will be the records only in the query one.

# The Oracle Server and SET Operators

- **Duplicate rows are automatically eliminated except in UNION ALL.**
- **Column names from the first query appear in the result.**
- **The output is sorted in ascending order by default except in UNION ALL.**
- The ORDER BY clause:
  - Can appear at the very end of the statement
  - Will accept the column name, aliases from the first SELECT statement, or the positional notation.

```
select empno,ename  
from emp  
union  
select deptno,dname  
from dept;
```

**NOTE:**

- Column names from the first query will be displayed in the result.
- By default the result is in ascending order of the first column of the first query.

EMPNO	ENAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS
50	MARKETING
401	khalid
1000	TANVEER
1001	WASEEM
1002	SAJID
7369	SMITH

```
1. select empno,ename  
   from emp  
   union  
   select deptno,dname  
   from dept  
   order by deptno;
```

#### NOTE

- Will accept the column name, aliases from the first SELECT statement, or the positional notation.

#### OUTPUT:

ORA-00904: "DEPTNO": invalid identifier.

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

2. select empno,ename  
from emp  
union  
select deptno,dname  
from dept  
order by 2;

**NOTE**

- Use of positional notation in the order by clause.

**OUTPUT:**

EMPNO	ENAME
10	ACCOUNTING
7876	ADAMS
7499	ALLEN
7698	BLAKE
7782	CLARK
7902	FORD
7900	JAMES
7566	JONES
7839	KING
50	MARKETING
7654	MARTIN

# Matching the SELECT Statements

**Using the UNION operator, display the department ID, location, and hire date for all employees.**

```
SELECT department_id, TO_NUMBER(null)
      location, hire_date
FROM   employees
UNION
SELECT department_id, location_id,  TO_DATE(null)
FROM   departments;
```

DEPARTMENT_ID	LOCATION	HIRE_DATE
10	1700	
10		17-SEP-07
20	1800	
20		17-FEB-96
...		
110	1700	
110		07-JUN-94
190	1700	
		24-MAY-99

27 rows selected.

# Matching the SELECT Statement

- Using the UNION operator, display the employee ID, job ID, and salary of all employees.

```
SELECT employee_id, job_id, salary  
FROM   employees  
UNION  
SELECT employee_id, job_id, 0  
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID	SALARY
100	AD_PRES	24000
101	AC_ACCOUNT	0
101	AC_MGR	0
...		
205	AC_MGR	12000
206	AC_ACCOUNT	8300

30 rows selected.

# Controlling the Order of Rows

Produce an English sentence using two UNION operators.

```
COLUMN a_dummy NOPRINT
SELECT 'sing' AS "My dream", 3 a_dummy
FROM dual
UNION
SELECT 'I''d like to teach', 1
FROM dual
UNION
SELECT 'the world to', 2
FROM dual
ORDER BY 2;
```

My dream
I'd like to teach
the world to
sing

**Note:** By default, the output is sorted in ascending order on the first column. You can use the ORDER BY clause to change hits.

## **2) TTITLE [text | ON | OFF]**

Specifies a header to appear at the top of each page.

**Example:**

TTITLE 'IBA'

## **3) BTITLE [text | ON | OFF]**

Specifies a footer to appear at the bottom of each page.

**Example:**

BTITLE 'GARDEN EAST'

## **4) BREAK [ON report\_element]:**

Suppresses duplicate values and sections rows of data with line feed.

**Example:**

BREAK ON job skip 2.

# *LESSON 8*

# **Manipulating Data**

**(DML)**

Copyright Tanveer Khan  
2005 - 2016  
All Rights Reserved

# Objectives

**After completing this lesson, you should be able to do the following:**

- **Describe each DML statement**
- **Insert rows into a table**
- **Update rows in a table**
- **Delete rows from a table**
- **Merge rows in a table**
- **Control transactions**

# Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

## **INSERTION:**

- INSERT statement add new rows to the table.
- Only one row is inserted in a row.

## **SYNTAX:**

```
INSERT INTO table [(column, column,...)] VALUES  
(values,values,....);
```

### **Example:**

```
INSERT INTO emp (empno,ename,sal,deptno,hiredate)  
VALUES (102,'RASHID',5000,30,'30-jan-01');
```

# Inserting New Rows

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.
- Optionally, list the columns in the INSERT clause.

```
INSERT INTO departments(department_id, department_name,
                      manager_id, location_id)
VALUES      (70, 'Public Relations', 100, 1700);
1 row created.
```

- Enclose character and date values within single quotation marks.

# INSERTING NULL VALUES:

## 1) IMPLICIT METHOD:

Omit the column from the column list

### Example:

- ~~Copyright © Tanveer Khan, 2005 - 2016. All rights reserved.~~ INSERT INTO dept(deptno,dname) VALUES(50,'ACCOUNTS');
- ~~Copyright © Tanveer Khan, 2005 - 2016. All rights reserved.~~ INSERT INTO dept VALUES (50,'ACCOUNTS');

## 2) EXPLICIT METHOD:

Specify the **NULL** keyword or the empty string '' in the values list.

### Example:

INSERT INTO dept VALUES (50,'ACCOUNTS', **NULL**);

## INSERTING DATES:

- 1) `INSERT INTO emp (empno,ename,deptno,hiredate) VALUES (102,'SAJID',40,SYSDATE);`
- 2) `INSERT INTO emp (empno,ename,deptno,hiredate) VALUES (102,'SAJID',40,'04-AUG-01');`
- 3) `INSERT INTO emp (empno,ename,deptno,hiredate) VALUES (102,'SAJID',40,to_date('AUG 04, 2001','MON dd,yyyy'));`

# INSERTING VALUES BY SUBSTITUTION VARIABLES:

- 1) INSERT INTO emp (empno,&col,deptno,hiredate) VALUES (102,'&value',40,'04-AUG-01');
- 2) ACCEPT col PROMPT 'Enter column name? '  
ACCEPT val PROMPT 'Enter value for column? '  
INSERT INTO emp (empno,&col,deptno,hiredate)  
VALUES(102,'&val',40,'04-AUG-01');

Copyright  
Tanveer Khan

Tanveer Khan, 2005 - 2016. All Rights Reserved

## COPYINGS ROW FROM ANOTHER TABLE:

- Write your INSERT statement with a subquery.
- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause to those in the subquery.

### Example:

```
INSERT INTO employee(id, name, salary)  
SELECT empno, ename, sal FROM emp WHERE job = 'CLERK';
```

## UPDATION:

- Modify existing rows with the UPDATE statement.
- More than one rows can be modified.
- Specify row or rows are modified when you specify the WHERE clause.
- All the rows in the table are modified if you omit the WHERE clause.

## SYNTAX:

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

## EXAMPLE:

UPDATE emp SET sal = 5000 WHERE empno = 102;

UPDATE emp SET ename = 'YASIR', sal = 5000 WHERE empno = 102;

# Updating Two Columns with a Subquery

**Update employee 114's job and salary to match that of employee 205.**

```
UPDATE      employees
SET          job_id   =  (SELECT    job_id
                         FROM      employees
                         WHERE     employee_id = 205),
            salary   =  (SELECT    salary
                         FROM      employees
                         WHERE     employee_id = 205)
WHERE        employee_id      =  114;
1 row updated.
```

# UPDATING WITH MULTIPLE COLUMN SUBQUERY:

## EXAMPLE:

```
UPDATE emp  
SET (ename,sal) = (SELECT ename,sal FROM emp where empno = 102)  
WHERE empno = 4421;
```

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# Updating Rows Based on Another Table

**Use subqueries in UPDATE statements to update rows in a table based on values from another table.**

```
UPDATE copy_emp
SET    department_id = (SELECT department_id
                        FROM employees
                        WHERE employee_id = 100)
WHERE   job_id          = (SELECT job_id
                           FROM employees
                           WHERE employee_id = 200);
1 row updated.
```

# Updating Rows: Integrity Constraint Error

```
UPDATE employees  
SET department_id = 55  
WHERE department_id = 110;
```

```
UPDATE employees  
*  
ERROR at line 1:  
ORA-02291: integrity constraint (HR.EMP_DEPT_FK)  
violated - parent key not found
```

**Department number 55 does not exist**

## **DELETION:**

- You can remove existing rows from the table by using the **DELETE** statement.
- Specific rows are deleted when you specify the WHERE clause.
- All rows in the table are deleted if you omit the WHERE clause.

### **Examples:**

- 1) `DELETE FROM dept WHERE deptno = 30;`
- 2) `DELETE FROM dept;`

## **DELETING ROWS BASED ON ANOTHER TABLE:**

Use subqueries in DELETE statements to remove rows from a table based on values from another table.

### **Example:**

```
DELETE FROM emp  
WHERE deptno  
IN  
(SELECT deptno FROM emp HAVING avg(sal) > 2000 GROUP BY deptno);
```

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# Deleting Rows: Integrity Constraint Error

```
DELETE FROM departments  
WHERE      department_id = 60;
```

```
DELETE FROM departments  
          *  
ERROR at line 1:  
ORA-02292: integrity constraint (HR.EMP_DEPT_FK)  
violated - child record found
```

**You cannot delete a row that contains a primary key that is used as a foreign key in another table.**

# The MERGE Statement

- **Provides the ability to conditionally update or insert data into a database table**
- **Performs an UPDATE if the row exists, and an INSERT if it is a new row:**
  - **Avoids separate updates**
  - **Increases performance and ease of use**
  - **Is useful in data warehousing applications**

# The MERGE Statement Syntax

You can conditionally insert or update rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
      col1 = col_val1,
      col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

# Merging Rows

**Insert or update rows in the COPY\_EMP table to match the EMPLOYEES table.**

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
    c.first_name      = e.first_name,
    c.last_name       = e.last_name,
    ...
    c.department_id  = e.department_id
WHEN NOT MATCHED THEN
  INSERT VALUES(e.employee_id, e.first_name, e.last_name,
                e.email, e.phone_number, e.hire_date, e.job_id,
                e.salary, e.commission_pct, e.manager_id,
                e.department_id);
```

# Merging Rows

```
SELECT *
FROM COPY_EMP;
```

no rows selected

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
    UPDATE SET
      ...
WHEN NOT MATCHED THEN
    INSERT VALUES...;
```

```
SELECT *
FROM COPY_EMP;
```

20 rows selected.

## Example:

**Step-1:** Create a copy table of EMP table having no record.

SQL> *CREATE TABLE employee*

*AS*

*SELECT \**

*FROM emp*

*WHERE 1 = 2;*

**Step-2:** Now insert a record into the copy table which also exists in the EMP table.

SQL> *INSERT INTO employee(empno,ename,sal)*

*VALUES(7369,'SAJID',5000);*

SQL> *COMMIT;*

### Step-3: Now apply MERGE on EMPLOYEE

SQL> *MERGE INTO EMPLOYEE A*

*USING EMP B*

*ON (A.EMPNO = B.EMPNO)*

*WHEN MATCHED*

*THEN UPDATE SET*

*A.ENAME = B.ENAME,*

*A.SAL = B.SAL*

*WHEN NOT MATCHED THEN*

*INSERT(A.EMPNO,A.ENAME,A.SAL)*

*VALUES(B.EMPNO,B.ENAME,B.SAL);*

## **DATABASE TRANSACTION:**

- A **TRANSACTION** consists of a collection of DML statements that form a logical unit of work.
- A **DATABASE TRANSACTION** consists of one of the following statements:
  - 1) DML statements that make up one consistent change to the data.
  - 2) One DDL statement.
  - 3) One DCL statement.
- Transactions consists of DML statements that make up one consistent change to the data.

### **Example:**

A transfer of funds between two accounts should include the debit to one account and credit to another account in the same amount. Both actions should either failed or succeed together; the credit should not be committed without the debit.

- Database Transaction begins when the first DML SQL statement is executed.
  
- Ends with one of the following events
  - COMMIT or ROLLBACK is issued.
  - DDL or DCL statement executes (Automatic commit).
  - User normally exits.

Copyright

Tanveer Khan, 2005 - 2016. All Rights Reserved

# **Advantages of COMMIT and ROLLBACK Statements**

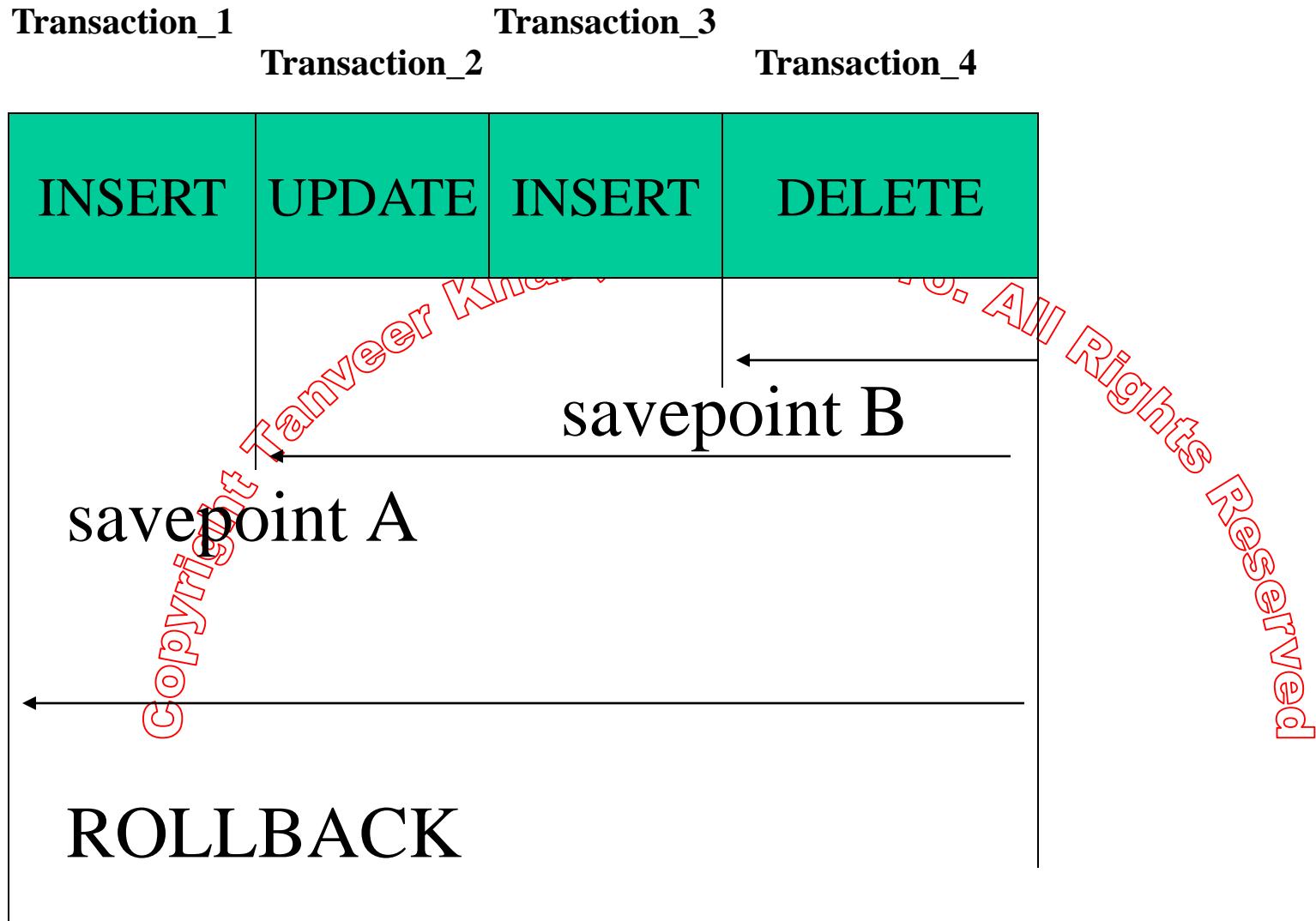
**With COMMIT and ROLLBACK statements, you can:**

- Ensure data consistency**
- Preview data changes before making changes permanent**
- Group logically related operations**

## CONTROLLING TRANSACTIONS:

We can control the logic of transactions by using the

COMMAND	EXPLANATION
COMMIT	Ends the current transaction by making all pending data changes permanent.
SAVEPOINT name	Marks the savepoint within the current location.
ROLLBACK [TO <i>Savepoint name</i> ]	ROLLBACK ends the current transaction by discarding all pending data changes. ROLLBACK TO <i>savepoint name</i> discards the savepoint and all subsequent changes.



# **Rolling Back Changes to a Marker**

- **Create a marker in a current transaction by using the SAVEPOINT statement.**
- **Roll back to that marker by using the ROLLBACK TO SAVEPOINT statement.**

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```

## Examples:

- 1) UPDATE emp SET ename = 'RASHID' WHERE empno = 101;
- ROLLBACK;
- 2) UPDATE emp SET ename = 'RASHID' WHERE empno = 101;
- COMMIT;
- 3) UPDATE emp SET ename = 'RASHID' WHERE empno = 101;
- SAVEPOINT updation;
- 4) INSERT INTO emp (empno,ename,deptno) VALUES (104,'WAJID',30);
- ROLLBACK TO updation;

# Implicit Transaction Processing

➤ **Automatic COMMIT** occurs under the following conditions

- **DDL statement** is issued
- **DCL statement** is issued.
- Normal Exit with command EXIT from SQL\*PLUS.

➤ **Automatic ROLLBACK** occurs under the following conditions

- Abnormal Exit or Termination of SQL \* PLUS.
- System Failure.

## Data Before COMMIT:

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using SELECT.
- Other users can not view the results of the DML operations made by the user.
- The affected rows are locked; other users cannot change the data within the affected rows.

## Data After COMMIT:

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released.
- All savepoints are erased.

# Committing Data

- Make the changes.

```
DELETE FROM employees  
WHERE employee_id = 99999;  
1 row deleted.
```

```
INSERT INTO departments  
VALUES (290, 'Corporate Tax', NULL, 1700);  
1 row inserted.
```

- Commit the changes.

```
COMMIT;
```

```
Commit complete.
```

# State of the Data After ROLLBACK

**Discard all pending changes by using the ROLLBACK statement:**

- **Data changes are undone.**
- **Previous state of the data is restored.**
- **Locks on the affected rows are released.**

```
DELETE FROM copy_emp;  
22 rows deleted.  
ROLLBACK;  
Rollback complete.
```

## Read Consistency:

- Database users access the database in two ways:
  - ✓ Read Operations (SELECT statement)
  - ✓ Write operations (INSERT,UPDATE,DELETE statements)
- You need read consistency so that the following occur:
  - ✓ The database reader and writer are assured a consistent view of the data.
  - ✓ Readers do not view data that is in the process of being changed.
  - ✓ Writers are ensured that the changes to the database are done in a consistent way.
  - ✓ Changes made by one writer do not disrupt or conflict with changes another writer is making.
- The purpose of read consistency is to ensure that user sees data as it existed at the last commit, before a DML operation is started.

## Implementation of Read Consistency:

- Read consistency is an automatic implementation.
- It keeps a partial copy of the database in **UNDO/ROLLBACK SEGMENT**.
  
- When an insert, update or delete operation is made to the database, the oracle server takes a *copy of the data before it is changed* and writes it to an **UNDO/ROLLBACK SEGMENT**.
  
- All users, except the one who issued the change, still see the database as it existed before the changes started; they view the **UNDO/ROLLBACK SEGMENT's** “snapshot” of the data.
  
- *Before changes are committed* to the database, only the user who is modifying the data sees the database with the alterations; everyone else sees the snapshot in the **UNDO/ROLLBACK SEGMENT**. This guarantees that readers of the data read consistent data that is not currently undergoing change.
  
- *When a DML statement is committed*, the change made to the database becomes visible to anyone executing a SELECT statement. The space occupied by the old data in the undo segment file is freed for reuse.
  
- *When a DML statement is rolled Back*, the changes are undone:
  - The original older version, of the data in the **UNDO/ROLLBACK SEGMENT**, is written back to the table.
  - All users see the database as it existed before the transaction began.

# **Locking**

**In an Oracle database, locks:**

- **Prevent destructive interaction between concurrent transactions**
- **Require no user action**
- **Automatically use the lowest level of restrictiveness**
- **Are held for the duration of the transaction**
- **Are of two types: explicit locking and implicit locking**

## Implicit Locking:

- Implicit locks are maintained by the Oracle itself and requires no user interaction.
- There are two types of implicit lock modes:
  - **Exclusive:**
    - ✓ A share lock is automatically obtained at the row level during DML operations e.g UPDATE
    - ✓ Exclusive locks prevent the row from being changed by other transactions until the transaction is committed or rolled back.
    - ✓ This lock ensures that no other user can modify the same row at the same time and overwrite changes not yet committed by another user.
  - **Share Lock:**
    - ✓ Allows other users to access.
    - ✓ A share lock is automatically obtained at the table level during DML operations e.g INSERT
    - ✓ With share lock mode, several transactions can acquire share locks on the same resource.

## Explicit Locking:

- Explicit Locking can be achieved by the user explicitly.
- SELECT .... FOR UPDATE statement is an explicit lock used.

### Example:

- LOCK TABLE table1,table2,table3 IN ROW EXCLUSIVE MODE;
- LOCK TABLE emp IN EXCLUSIVE MODE;
- SELECT empno,ename FROM EMP FOR UPDATE;

# **LESSON 9**

## **Using DDL**

### **Statements to Create and Manage Tables**

Copyright © Tanveer Khan 2005 - 2016. All rights reserved.

# **Database Objects:**

## **1) TABLE:**

Basic unit of storage; composed of rows and columns.

## **2) VIEW:**

Logically represents subsets of data from one or more tables.

## **3) SEQUENCE:**

Generate primary key values or numeric values generator.

## **4) INDEX:**

Improve the performance of some queries.

## **5) SYNONYM:**

Gives alternate names to objects.

## **6) PROCEDURE or FUNCTION:**

Small modules of precompiled programs.

# Naming Rules

## Table names and column names:

- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, \_, \$, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server reserved word

# The CREATE TABLE Statement

- You must have:
  - CREATE TABLE privilege
  - A storage area

```
CREATE TABLE [schema.]table  
          (column datatype [DEFAULT expr] [, . . .]);
```

- You specify:
  - Table name
  - Column name, column data type, and column size

## CREATE table:

### Syntax:

```
CREATE TABLE table_name (column datatype [DEFAULT expr]  
[.....]);
```

### Example:

```
CREATE TABLE employee  
(empno NUMBER(5),  
ename VARCHAR2(25),  
hiredate DATE DEFAULT sysdate);
```

# The DEFAULT Option

- Specify a default value for a column during an insert.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- Literal values, expressions, or SQL functions are legal values.
- Another column's name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

# TABLES IN ORACLE:

## ➤ USER TABLES:

- tables created and maintained by the user.
- contain user information.

## ➤ DATA DICTIONARY:

- tables created and maintained by the Oracle server.
- contain database information.
- All data dictionary tables owned by the **SYS** user.
- Information Stored in data dictionary includes:

- Names of Oracle Server Users
- Privileges granted to users.
- Database object names.
- Table Constraints.
- Auditing information

## Types of Data dictionary View:

There are four categories of data dictionary views; each category has a distinct prefix that reflects its intended use.

2005 - 2016

PREFIX	DESCRIPTION
USER_	These views contain information about object owned by the user.
ALL_	These views contains information about all the tables accessible to the users.
DBA_	These views are restricted views, which can be accessed only by the people who have been assigned the DBA role.
V\$	These views are dynamic performance views, database server performance, memory and locking.

# Querying the Data Dictionary

- See the names of tables owned by the user.

```
SELECT table_name  
FROM   user_tables ;
```

- View distinct object types owned by the user.

```
SELECT DISTINCT object_type  
FROM   user_objects ;
```

- View tables, views, synonyms, and sequences owned by the user.

```
SELECT *  
FROM   user_catalog ;
```

## Creating Tables Using Subqueries:

- With the help of subqueries, we can create a table and insert the rows into it.
- We can assign the new names to the columns.

### SYNTAX:

```
CREATE TABLE table_name [(column, column, ...)]  
AS subquery;
```

### Examples:

1) CREATE TABLE employee

AS

```
SELECT empno,ename,sal FROM emp WHERE empno = 40;
```

2) CREATE TABLE employee

(employee\_num,employee\_name,salary)

AS

```
SELECT empno,ename,sal FROM emp WHERE empno = 40;
```

- To create a table with the same structure as an existing table, but without the data from the existing table, use a sub query with a where clause that always evaluates as FALSE.

### Example:

```
CREATE TABLE copy_table AS  
(  
SELECT *  
FROM emp  
WHERE 1=2  
);
```

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# Creating a Table by Using a Subquery

```
CREATE TABLE dept80
AS
SELECT employee_id, last_name,
       salary*12 ANNSAL,
       hire_date
  FROM employees
 WHERE department_id = 80;
```

Table created.

```
DESCRIBE dept80
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

## ALTER TABLE:

ALTER TABLE statement is used to

- **Add** new column.
- **Modify** an existing column.
- **Drop** a column. (only one column at a time)
- **Rename** a column.

## SYNTAX:

- 1) ALTER TABLE table\_name  
**ADD** (column datatype, column datatype, ....);
- 2) ALTER TABLE table\_name  
**MODIFY** (column datatype, column datatype, ....);
- 3) ALTER TABLE table\_name  
**DROP COLUMN**(column\_name);
- 4) ALTER TABLE table\_name  
**RENAME COLUMN** <old\_col\_name> TO <new\_col\_name>;

## Example:

- 1) ALTER TABLE dept **ADD** (job VARCHAR2(9));
- 2) ALTER TABLE dept **MODIFY**(ename CHAR(9));
- 3) ALTER TABLE dept **DROP COLUMN** ename;  
ALTER TABLE dept **DROP** (ename);
- 4) ALTER TABLE emp01 **RENAME COLUMN** empno TO emp\_id;

*Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved*

# Adding a Column

- You use the ADD clause to add columns.

```
ALTER TABLE dept80
ADD          (job_id VARCHAR2(9));
Table altered.
```

- The new column becomes the last column.

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOD_ID
149	Zlotkey	128000	29-JAN-00	
174	Abel	132000	11-MAY-96	
176	Taylor	103200	24-MAR-98	

Note:

If a table already contains rows when a column is added, then the new column is initially NULL for all rows.

# Modifying a Column

- You can change a column's data type, size, and default value.

```
ALTER TABLE    dept80
MODIFY          (last_name VARCHAR2(30)) ;
Table altered.
```

- A change to the default value affects only subsequent insertions to the table.

## Guidelines:

- You can increase the width and precision of a numeric column.
- You can increase the width of numeric or character columns.
- You can decrease the width of a column only if the column contains only null values or if the table has no rows.
- You can change the data type only if the column contains null values.
- You can convert a CHAR column to VARCHAR2 data type or convert a VARCHAR2 column to CHAR data type only if the column contains null values or if you don't change the size.
- A change to the default value of a column affects only subsequent insertions to the table.

# Dropping a Column

**Use the `DROP COLUMN` clause to drop columns you no longer need from the table.**

```
ALTER TABLE dept80
DROP COLUMN job_id;
Table altered.
```

## Guidelines:

- The column may or may not contain data.
- Using the `ALTER TABLE` statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- Once a column is dropped, it cannot be recovered.

# The SET UNUSED Option

- You use the SET UNUSED option to mark one or more columns as unused.
- You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.

```
ALTER TABLE    table  
SET    UNUSED  (column);
```

OR

```
ALTER TABLE    table  
SET    UNUSED COLUMN column;
```

```
ALTER TABLE table  
DROP  UNUSED COLUMNS;
```

## Guidelines:

- After a column has been marked as unused, you have no access to that column.
- A SELECT \* query will not retrieve data from unused columns.
- The names and types of the columns marked unused will not be displayed during a DESCRIBE, and you can add to the table a new column with the same name as an unused column.
- SET UNUSED information is stored in the USER\_UNUSED\_COL\_TABS dictionary view.

## **DROP table:**

- All data and structure in the table is deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- Statement cannot be roll back.

### **SYNTAX:**

```
DROP TABLE table_name;
```

### **Example:**

```
DROP TABLE employee;
```

## **RENAME table:**

- To change the name of a table, view, sequence or synonym.
- You must be the owner of the object.

### **Syntax:**

```
RENAME old_name TO new_name;
```

### **Example:**

```
RENAME emp TO employee;
```

## TRUNCATE table:

- Remove all rows from a table.
- Releases all storage space used by the table.
- Truncated rows cannot be roll back.
- **DELETE** is a **DML** command while **TRUNCATE** is a **DDL** command so it cannot be roll back.

### SYNTAX:

TRUNCATE TABLE table\_name;

### Example:

TRUNCATE TABLE employee;

## COMMENTS:

- Comments can be added to a TABLE or a COLUMN.

## SYNTAX:

```
COMMENT ON TABLE table_name | COLUMN table.column IS  
'text';
```

### Example:

```
COMMENT ON COLUMN emp.empno IS 'employee number';  
COMMENT ON TABLE emp IS 'EMPLOYEES INFORMATION';
```

- Comments can be dropped

```
COMMENT ON COLUMN emp.empno IS '';
```

- Comments can be viewed through the data dictionary views:

- ALL\_COL\_COMMENTS
- USER\_COL\_COMMENTS
- ALL\_TAB\_COMMENTS
- USER\_COL\_COMMENTS

➤ **To check the comments:**

select table\_name, column\_name, comments

From all\_col\_comments

where table\_name = 'EMP';

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# **Tables Constraints**

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

## CONSTRAINTS:

- Constraints enforces rules at the table level.
- Constraints prevents the deletion of a table if there are dependencies.
- Constraints can be **added,dropped,enabled** or **disabled**.

## CONSTRAINTS TYPES:

- 1) NOT NULL.
- 2) UNIQUE.
- 3) PRIMARY KEY.
- 4) FOREIGN KEY.
- 5) CHECK.

## Constraints Guidelines:

- Name a constraint or the Oracle server generates a name by using the **SYS\_Cn** format where **n** is an integer so that the constraint name is unique.
- Create a constraint either:
  - At the same time as the table is created, or
  - After the table has been created
- Define a constraint at the column or table level.
- You can view the constraints defined for a specific table by looking at the **USER\_CONSTRAINTS** data dictionary table.

# Defining Constraints

```
CREATE TABLE [schema.]table  
  (column datatype [DEFAULT expr]  
   [column_constraint],  
   ...  
   [table_constraint] [, . . .]);
```

```
CREATE TABLE employees(  
    employee_id  NUMBER(6),  
    first_name    VARCHAR2(20),  
    ...  
    job_id        VARCHAR2(10) NOT NULL,  
    CONSTRAINT emp_emp_id_pk  
                PRIMARY KEY (EMPLOYEE_ID));
```

## Levels of Constraints:

### 1) COLUMN LEVEL constraints:

- References a single column.
- Defined with the definition of column.

#### Example:

```
CREATE TABLE employee  
(empno number(5) PRIMARY KEY,  
ename varchar2(25));
```

### 2) TABLELEVEL constraints:

- References one or more columns
- Defined separately from the definition of the column in the table.
- Can define any constraint except **NOT NULL**.

#### Example:

```
CREATE TABLE employee  
(empno number(5),  
ename varchar2(25),  
PRIMARY KEY(empno));
```

# The NOT NULL Constraint

**Ensures that null values are not permitted for the column:**

EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

20 rows selected.

  
**NOT NULL constraint**  
**(No row can contain a null value for this column.)**

  
**NOT NULL constraint**

  
**Absence of NOT NULL constraint**  
**(Any row can contain null for this column.)**

# The NOT NULL Constraint

Is defined at the column level:

```
CREATE TABLE employees (
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,           ← System named
    salary            NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date         DATE
        CONSTRAINT emp_hire_date_nn
        NOT NULL,
    ...
)
```

System  
named

User  
named

## 1) NOT NULL Constraint:

- Defined at the column level, not at the table level.

### Example:

```
CREATE TABLE employee  
(empno number(4),  
ename varchar2(25) NOT NULL)
```

## 2) UNIQUE Key Constraint:

- Defined at either the table level or the column level.
- If the UNIQUE constraint comprises more than one column, that group of columns is called the *Composite Unique Key*.
- Null values can be inserted.

### Example:

1) Create table employee

```
(ename varchar2(25) NOT NULL UNIQUE);
```

2) Create table employee

```
(empno number(3),
```

```
ename varchar2(25),
```

```
CONSTRAINT unique_cons UNIQUE (ename,empno))
```

### 3) PRIMARY KEY constraint:

- Defined at either the table or the column level.
- Composite key is created by using the table level definition.
- A table can have only one PRIMARY KEY but can have several UNIQUE constraints.
- PRIMARY KEY cannot be null.

#### Example:

- 1) CREATE TABLE employee  
(empno number(5),  
ename varchar2(25),  
CONSTRAINT emp\_cons PRIMARY KEY(empno));
- 2) CREATE TABLE employee  
(empno number(5) Primary Key,  
ename varchar2(25));

#### 4) FOREIGN KEY constraint:

- Defined at either the **table** or the **column** level.
- Composite Foreign Key is created by using the table level definition.
- **Foreign key:** Defines the column in the child table at the table level constraint.
- **References:** Identifies the table and column in the parent table.
- **On delete cascade:** allows deletion in the parent table and deletion on the dependent rows in the child table.
- **On delete set null:** converts dependent foreign key values to NULL when the parent value is removed.
- Without the **ON DELETE CASCADE** or the **ON DELETE SET NULL** options, the row in the parent table cannot be deleted if it is referenced in the child table.

## Example:

### AT TABLE LEVEL

```
CREATE TABLE employee  
(empno number(5) PRIMARY KEY,  
ename varchar2(25),  
deptno number(5),  
CONSTRAINT emp_cons FOREIGN KEY(deptno) REFERENCES  
dept (deptno) ON DELETE CASCADE);
```

### AT COLUMN LEVEL

```
CREATE TABLE employee  
(empno number(5) PRIMARY KEY,  
ename varchar2(25),  
deptno number(5) references (dept));
```

## 5) CHECK constraint:

- Defined at either the table level or the column level.
- Defines a condition that each row must satisfy.

### Example:

```
CREATE TABLE employee  
(empno number(5),  
ename varchar2(25),  
CONSTRAINT empdept_cons CHECK(empno BETWEEN 1000 AND 5000));
```

# Adding a Constraint Syntax

**Use the ALTER TABLE statement to:**

- **Add or drop a constraint, but not modify its structure**
- **Enable or disable constraints**
- **Add a NOT NULL constraint by using the MODIFY clause**

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

- You can define a NOT NULL column only if the table is empty or if the column has the value for each row.

## ADDING a constraint:

Example:

```
ALTER TABLE employee  
ADD CONSTRAINT  
emp_cons PRIMARY KEY(empno);
```

## DROPPING a constraint:

Example:

```
ALTER TABLE employee  
DROP CONSTRAINT emp_cons;
```

## DISABLING a constraint:

Example:

```
ALTER TABLE employee  
DISABLE CONSTRAINT emp_cons;
```

## ENABLING a constraint:

Example:

```
ALTER TABLE employee  
ENABLE CONSTRAINT emp_cons;
```

# Adding a Constraint

**Add a FOREIGN KEY constraint to the EMPLOYEES table indicating that a manager must already exist as a valid employee in the EMPLOYEES table.**

```
ALTER TABLE      employees
ADD CONSTRAINT  emp_manager_fk
    FOREIGN KEY(manager_id)
    REFERENCES employees(employee_id) ;
```

**Table altered.**

## Dropping a constraint:

- To drop a constraints, you can identify the constraint name from the ***USER\_CONSTRAINTS*** and ***USER\_CONS\_COLUMNS*** data dictionary views.

### Syntax:

```
ALTER TABLE table_name  
DROP PRIMARY KEY | UNIQUE (col_name) |  
CONSTRAINT constraints [CASCADE];
```

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# Dropping a Constraint

- Remove the manager constraint from the EMPLOYEES table.

```
ALTER TABLE      employees
DROP CONSTRAINT emp_manager_fk;
Table altered.
```

- Remove the PRIMARY KEY constraint on the DEPARTMENTS table and drop the associated FOREIGN KEY constraint on the EMPLOYEES . DEPARTMENT\_ID column.

```
ALTER TABLE departments
DROP PRIMARY KEY CASCADE;
Table altered.
```

# Disabling Constraints

- Execute the **DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint.**
- **Apply the CASCADE option to disable dependent integrity constraints.**

```
ALTER TABLE          employees
DISABLE CONSTRAINT  emp_emp_id_pk CASCADE;
Table altered.
```

# Enabling Constraints

- Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.

```
ALTER TABLE employees  
ENABLE CONSTRAINT emp_emp_id_pk;  
Table altered.
```

- A UNIQUE or PRIMARY KEY index is automatically created if you enable a UNIQUE key or PRIMARY KEY constraint.

# Cascading Constraints

- The **CASCADE CONSTRAINTS** clause is used along with the **DROP COLUMN** clause.
- The **CASCADE CONSTRAINTS** clause drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns.
- The **CASCADE CONSTRAINTS** clause also drops all multicolumn constraints defined on the dropped columns.

## **Example-1:**

Step-1: Create a dummy table DEPT2 table.

```
Create table dept2( deptno number constraint dept2_pk primary key,  
dname varchar2(10));
```

Step-2: Create a dummy table EMP2 table.

```
Create table emp2( empno number constraint emp2_pk primary key,  
ename varchar2(25),  
deptno number constraint emp2_fk references dept2(deptno) on delete set  
null);
```

Step-3: Now insert the values in both the tables.

```
Insert into dept2 values(10, 'MKT');
```

```
Insert into emp2 values(101, 'A', 10);
```

Step-4: Now try to drop the DEPTNO column from the DEPT table.

```
Alter table dept2  
drop column deptno;
```

**You will get the error.**

Step-5: Now try to drop the DEPTNO column from the DEPT table with the following command.

```
Alter table dept2  
drop column deptno  
cascade constraints;
```

The above command will drop the column by deleting the constraint applied on both EMP2 and DEPT2.

## Example:

```
create table test1
(
    pk number PRIMARY KEY,
    fk number,
    col1 number,
    col2 number,
    CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,
    CONSTRAINT ck1 CHECK(pk > 0 and col1 > 0),
    CONSTRAINT ck2 CHECK(col2 > 0)
);
```

*Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved*

Now executing the following commands will generate error

```
ALTER TABLE test1 DROP (pk);      -- pk is a parent key
ALTER TABLE test1 DROP (col1);    -- col1 is referenced by multicolumn constraint ck1
```

## To remove the errors:

- Submitting the following statement drops column PK, the primary key constraint, the fk\_constraint foreign key constraint, and the check constraints CK1.

```
ALTER TABLE test1 DROP (pk) CASCADE CONSTRAINTS;
```

- If all the columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required.

```
ALTER TABLE test1 DROP (pk,fk,col1);
```

## Viewing Constraints:

- The only constraint you can verify thorough the DESCRIBE command is NOT NULL.
- To view all the constraints on your table , query the USER\_CONSTRAINTS table.

### Example:

```
SELECT constraint_name , constraint_type, search_condition  
FROM USER_CONSTRAINTS  
WHERE table_name = 'EMP';
```

CONSTRAINT_TYPE	SYMBOL
CHECK	C
PRIMARY KEY	P
FOREIGN KEY	R
UNIQUE	U
NOT NULL	C

```

SELECT      constraint_name, constraint_type,
            search_condition
FROM        user_constraints
WHERE       table_name = 'EMPLOYEES';

```

CONSTRAINT_NAME	C	SEARCH_CONDITION
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL
EMP_SALARY_MIN	C	salary > 0
EMP_EMAIL_UK	U	

# Viewing the Columns Associated with Constraints

**View the columns associated with the constraint names in the USER\_CONS\_COLUMNS view.**

```
SELECT      constraint_name, column_name
FROM        user_cons_columns
WHERE       table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID
...	

# **LESSON 11**

# **Creating**

# **Other**

# **Schema**

# **Objects**

Copyright Tanveer Khan 2005-2016 All rights reserved

# Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

## What is a View?

- You can present logical subsets or combination of data by creating views of tables.
- A view is a logical table based on a table or another view.
- A view contains no data of its own but is like a window through which data from tables can be viewed or changed.
- The tables on which a view is based are called **BASE TABLES**.
- The view is stored as a SELECT statement in the data dictionary.

# What is a View?

## EMPLOYEES Table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000
104	Pat	Randall	PRANDALL	590.423.4568	01-JAN-95	SA_MAN	60000
105	SG	Bergman	SGMURRAY	590.423.4569	01-JAN-95	SA_CLERK	42000
106	Elis	Martins	EMARTINS	590.423.4570	01-JAN-95	SA_CLERK	58000
107	Teena	Montgomery	TMONAGHTON	590.423.4571	01-JAN-95	SA_CLERK	35000
108	Mark	Frederick	MFRDERICK	590.423.4572	01-JAN-95	SA_CLERK	31000
109	Adam	Khurana	AKHURANA	590.423.4573	01-JAN-95	SA_CLERK	26000
110	Shelli	Guiseppe	SGUISEPPE	590.423.4574	01-JAN-95	SA_CLERK	25000
111	Howard	Elbel	HELBEL	590.423.4575	01-JAN-95	SA_CLERK	25000
112	Patricia	Abel	PABEL	590.423.4576	01-JAN-95	SA_CLERK	25000
113	John	Taylor	JTAYLOR	590.423.4577	01-JAN-95	SA_CLERK	25000
114	David	Zlotkey	DZLOTKEY	590.423.4578	01-JAN-95	SA_CLERK	25000
115	Michael	Ernst	MEERNST	590.423.4579	01-JAN-95	SA_CLERK	25000
116	Victoria	Smith	VSMITH	590.423.4580	01-JAN-95	SA_CLERK	25000
117	Robert	Scott	RSCOTT	590.423.4581	01-JAN-95	SA_CLERK	25000
118	Patricia	Montgomery	PMONTGOMERY	590.423.4582	01-JAN-95	SA_CLERK	25000
119	James	Frederick	JFRDERICK	590.423.4583	01-JAN-95	SA_CLERK	25000
120	Patricia	Khurana	PKHURANA	590.423.4584	01-JAN-95	SA_CLERK	25000
121	James	Elbel	JELBEL	590.423.4585	01-JAN-95	SA_CLERK	25000
122	Patricia	Abel	PABEL	590.423.4586	01-JAN-95	SA_CLERK	25000
123	James	Taylor	JTAYLOR	590.423.4587	01-JAN-95	SA_CLERK	25000
124	Patricia	Zlotkey	PZLOTKEY	590.423.4588	01-JAN-95	SA_CLERK	25000
125	James	Ernst	JEERNST	590.423.4589	01-JAN-95	SA_CLERK	25000
126	Patricia	Smith	PSMITH	590.423.4590	01-JAN-95	SA_CLERK	25000
127	James	Montgomery	JMONTGOMERY	590.423.4591	01-JAN-95	SA_CLERK	25000
128	Patricia	Frederick	PFREDERICK	590.423.4592	01-JAN-95	SA_CLERK	25000
129	James	Khurana	JKHURANA	590.423.4593	01-JAN-95	SA_CLERK	25000
130	Patricia	Elbel	PELBEL	590.423.4594	01-JAN-95	SA_CLERK	25000
131	James	Abel	JABEL	590.423.4595	01-JAN-95	SA_CLERK	25000
132	Patricia	Taylor	PTAYLOR	590.423.4596	01-JAN-95	SA_CLERK	25000
133	James	Zlotkey	JZLOTKEY	590.423.4597	01-JAN-95	SA_CLERK	25000
134	Patricia	Ernst	PERNST	590.423.4598	01-JAN-95	SA_CLERK	25000
135	James	Smith	JSMITH	590.423.4599	01-JAN-95	SA_CLERK	25000
136	Patricia	Montgomery	PMONTGOMERY	590.423.4600	01-JAN-95	SA_CLERK	25000
137	James	Frederick	JFRDERICK	590.423.4601	01-JAN-95	SA_CLERK	25000
138	Patricia	Khurana	PKHURANA	590.423.4602	01-JAN-95	SA_CLERK	25000
139	James	Elbel	JELBEL	590.423.4603	01-JAN-95	SA_CLERK	25000
140	Patricia	Abel	PABEL	590.423.4604	01-JAN-95	SA_CLERK	25000
141	James	Taylor	JTAYLOR	590.423.4605	01-JAN-95	SA_CLERK	25000
142	Patricia	Zlotkey	PZLOTKEY	590.423.4606	01-JAN-95	SA_CLERK	25000
143	James	Ernst	JEERNST	590.423.4607	01-JAN-95	SA_CLERK	25000
144	Patricia	Smith	PSMITH	590.423.4608	01-JAN-95	SA_CLERK	25000
145	James	Montgomery	JMONTGOMERY	590.423.4609	01-JAN-95	SA_CLERK	25000
146	Patricia	Frederick	PFREDERICK	590.423.4610	01-JAN-95	SA_CLERK	25000
147	James	Khurana	JKHURANA	590.423.4611	01-JAN-95	SA_CLERK	25000
148	Patricia	Elbel	PELBEL	590.423.4612	01-JAN-95	SA_CLERK	25000
149	James	Abel	JABEL	590.423.4613	01-JAN-95	SA_CLERK	25000
150	Patricia	Taylor	PTAYLOR	590.423.4614	01-JAN-95	SA_CLERK	25000
151	James	Zlotkey	JZLOTKEY	590.423.4615	01-JAN-95	SA_CLERK	25000
152	Patricia	Ernst	PERNST	590.423.4616	01-JAN-95	SA_CLERK	25000
153	James	Smith	JSMITH	590.423.4617	01-JAN-95	SA_CLERK	25000
154	Patricia	Montgomery	PMONTGOMERY	590.423.4618	01-JAN-95	SA_CLERK	25000
155	James	Frederick	JFRDERICK	590.423.4619	01-JAN-95	SA_CLERK	25000
156	Patricia	Khurana	JKHURANA	590.423.4620	01-JAN-95	SA_CLERK	25000
157	James	Elbel	JELBEL	590.423.4621	01-JAN-95	SA_CLERK	25000
158	Patricia	Abel	PABEL	590.423.4622	01-JAN-95	SA_CLERK	25000
159	James	Taylor	JTAYLOR	590.423.4623	01-JAN-95	SA_CLERK	25000
160	Patricia	Zlotkey	PZLOTKEY	590.423.4624	01-JAN-95	SA_CLERK	25000
161	James	Ernst	JEERNST	590.423.4625	01-JAN-95	SA_CLERK	25000
162	Patricia	Smith	PSMITH	590.423.4626	01-JAN-95	SA_CLERK	25000
163	James	Montgomery	JMONTGOMERY	590.423.4627	01-JAN-95	SA_CLERK	25000
164	Patricia	Frederick	PFREDERICK	590.423.4628	01-JAN-95	SA_CLERK	25000
165	James	Khurana	JKHURANA	590.423.4629	01-JAN-95	SA_CLERK	25000
166	Patricia	Elbel	PELBEL	590.423.4630	01-JAN-95	SA_CLERK	25000
167	James	Abel	JABEL	590.423.4631	01-JAN-95	SA_CLERK	25000
168	Patricia	Taylor	PTAYLOR	590.423.4632	01-JAN-95	SA_CLERK	25000
169	James	Zlotkey	JZLOTKEY	590.423.4633	01-JAN-95	SA_CLERK	25000
170	Patricia	Ernst	PERNST	590.423.4634	01-JAN-95	SA_CLERK	25000
171	James	Smith	JSMITH	590.423.4635	01-JAN-95	SA_CLERK	25000
172	Patricia	Montgomery	PMONTGOMERY	590.423.4636	01-JAN-95	SA_CLERK	25000
173	James	Frederick	JFRDERICK	590.423.4637	01-JAN-95	SA_CLERK	25000
174	Patricia	Khurana	JKHURANA	590.423.4638	01-JAN-95	SA_CLERK	25000
175	James	Elbel	JELBEL	590.423.4639	01-JAN-95	SA_CLERK	25000
176	Patricia	Abel	PABEL	590.423.4640	01-JAN-95	SA_CLERK	25000
177	James	Taylor	JTAYLOR	590.423.4641	01-JAN-95	SA_CLERK	25000
178	Patricia	Zlotkey	PZLOTKEY	590.423.4642	01-JAN-95	SA_CLERK	25000
179	James	Ernst	JEERNST	590.423.4643	01-JAN-95	SA_CLERK	25000
180	Patricia	Smith	PSMITH	590.423.4644	01-JAN-95	SA_CLERK	25000
181	James	Montgomery	JMONTGOMERY	590.423.4645	01-JAN-95	SA_CLERK	25000
182	Patricia	Frederick	PFREDERICK	590.423.4646	01-JAN-95	SA_CLERK	25000
183	James	Khurana	JKHURANA	590.423.4647	01-JAN-95	SA_CLERK	25000
184	Patricia	Elbel	PELBEL	590.423.4648	01-JAN-95	SA_CLERK	25000
185	James	Abel	JABEL	590.423.4649	01-JAN-95	SA_CLERK	25000
186	Patricia	Taylor	PTAYLOR	590.423.4650	01-JAN-95	SA_CLERK	25000
187	James	Zlotkey	JZLOTKEY	590.423.4651	01-JAN-95	SA_CLERK	25000
188	Patricia	Ernst	PERNST	590.423.4652	01-JAN-95	SA_CLERK	25000
189	James	Smith	JSMITH	590.423.4653	01-JAN-95	SA_CLERK	25000
190	Patricia	Montgomery	PMONTGOMERY	590.423.4654	01-JAN-95	SA_CLERK	25000
191	James	Frederick	JFRDERICK	590.423.4655	01-JAN-95	SA_CLERK	25000
192	Patricia	Khurana	JKHURANA	590.423.4656	01-JAN-95	SA_CLERK	25000
193	James	Elbel	JELBEL	590.423.4657	01-JAN-95	SA_CLERK	25000
194	Patricia	Abel	PABEL	590.423.4658	01-JAN-95	SA_CLERK	25000
195	James	Taylor	JTAYLOR	590.423.4659	01-JAN-95	SA_CLERK	25000
196	Patricia	Zlotkey	PZLOTKEY	590.423.4660	01-JAN-95	SA_CLERK	25000
197	James	Ernst	JEERNST	590.423.4661	01-JAN-95	SA_CLERK	25000
198	Patricia	Smith	PSMITH	590.423.4662	01-JAN-95	SA_CLERK	25000
199	James	Montgomery	JMONTGOMERY	590.423.4663	01-JAN-95	SA_CLERK	25000
200	Patricia	Frederick	PFREDERICK	590.423.4664	01-JAN-95	SA_CLERK	25000
201	James	Khurana	JKHURANA	590.423.4665	01-JAN-95	SA_CLERK	25000
202	Patricia	Elbel	PELBEL	590.423.4666	01-JAN-95	SA_CLERK	25000
203	James	Abel	JABEL	590.423.4667	01-JAN-95	SA_CLERK	25000
204	Patricia	Taylor	PTAYLOR	590.423.4668	01-JAN-95	SA_CLERK	25000
205	James	Zlotkey	JZLOTKEY	590.423.4669	01-JAN-95	SA_CLERK	25000
206	Patricia	Ernst	PERNST	590.423.4670	01-JAN-95	SA_CLERK	25000
207	James	Smith	JSMITH	590.423.4671	01-JAN-95	SA_CLERK	25000
208	Patricia	Montgomery	PMONTGOMERY	590.423.4672	01-JAN-95	SA_CLERK	25000
209	James	Frederick	JFRDERICK	590.423.4673	01-JAN-95	SA_CLERK	25000
210	Patricia	Khurana	JKHURANA	590.423.4674	01-JAN-95	SA_CLERK	25000
211	James	Elbel	JELBEL	590.423.4675	01-JAN-95	SA_CLERK	25000
212	Patricia	Abel	PABEL	590.423.4676	01-JAN-95	SA_CLERK	25000
213	James	Taylor	JTAYLOR	590.423.4677	01-JAN-95	SA_CLERK	25000
214	Patricia	Zlotkey	PZLOTKEY	590.423.4678	01-JAN-95	SA_CLERK	25000
215	James	Ernst	JEERNST	590.423.4679	01-JAN-95	SA_CLERK	25000
216	Patricia	Smith	PSMITH	590.423.4680	01-JAN-95	SA_CLERK	25000
217	James	Montgomery	JMONTGOMERY	590.423.4681	01-JAN-95	SA_CLERK	25000
218	Patricia	Frederick	PFREDERICK	590.423.4682	01-JAN-95	SA_CLERK	25000
219	James	Khurana	JKHURANA	590.423.4683	01-JAN-95	SA_CLERK	25000
220	Patricia	Elbel	PELBEL	590.423.4684	01-JAN-95	SA_CLERK	25000
221	James	Abel	JABEL	590.423.4685	01-JAN-95	SA_CLERK	25000
222	Patricia	Taylor	PTAYLOR	590.423.4686	01-JAN-95	SA_CLERK	25000
223	James	Zlotkey	JZLOTKEY	590.423.4687	01-JAN-95	SA_CLERK	25000
224	Patricia	Ernst	PERNST	590.423.4688	01-JAN-95	SA_CLERK	25000
225	James	Smith	JSMITH	590.423.4689	01-JAN-95	SA_CLERK	25000
226	Patricia	Montgomery	PMONTGOMERY	590.423.4690	01-JAN-95	SA_CLERK	25000
227	James	Frederick	JFRDERICK	590.423.4691	01-JAN-95	SA_CLERK	25000
228	Patricia	Khurana	JKHURANA	590.423.4692	01-JAN-95	SA_CLERK	25000
229	James	Elbel	JELBEL	590.423.4693	01-JAN-95	SA_CLERK	25000
230	Patricia	Abel	PABEL	590.423.4694	01-JAN-95	SA_CLERK	25000
231	James	Taylor	JTAYLOR	590.423.4695	01-JAN-95	SA_CLERK	25000
232	Patricia	Zlotkey	PZLOTKEY	590.423.4696	01-JAN-95	SA_CLERK	25000
233	James	Ernst	JEERNST	590.423.4697	01-JAN-95	SA_CLERK	25000
234	Patricia	Smith	PSMITH	590.423.4698	01-JAN-95	SA_CLERK	25000
235	James	Montgomery	JMONTGOMERY	590.423.4699	01-JAN-95	SA_CLERK	25000
236	Patricia	Frederick	PFREDERICK	590.423.4700	01-JAN-95	SA_CLERK	25000
237	James	Khurana	JKHURANA	590.423.4701	01-JAN-95	SA_CLERK	25000
238	Patricia	Elbel	PELBEL	590.423.4702	01-JAN-95	SA_CLERK	25000
239	James	Abel	JABEL	590.423.4703	01-JAN-95	SA_CLERK	25000
240	Patricia	Taylor	PTAYLOR	590.423.4704	01-JAN-95	SA_CLERK	25000
241	James	Zlotkey	JZLOTKEY	590.423.4705	01-JAN-95	SA_CLERK	25000
242	Patricia	Ernst	PERNST	590.423.4706	01-JAN-95	SA_CLERK	25000
243	James	Smith	JSMITH	590.423.4707	01-JAN-95	SA_CLERK	25000
244	Patricia	Montgomery	PMONTGOMERY	590.423.4708</td			

## WHY USE VIEWS:

- To restrict database access.
- To make complex queries easy.
- To allow data independence. (one view can be used to retrieve data from several tables.)
- To present different views of the same data.

### Types of views:

#### i) Simple View:

- Derives data from only one table.
- Contains no functions or groups of data.
- Can perform DML through the view.

#### ii) Complex View:

- Derives data from many tables.
- Contains functions or groups of data.
- Does not always allow DML through view.

# **Simple Views and Complex Views**

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

# Creating a View

- You embed a subquery within the CREATE VIEW statement.

```
CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW view  
[(alias[, alias] . . .)]  
AS subquery  
[WITH CHECK OPTION [CONSTRAINT constraint]]  
[WITH READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex SELECT syntax.

OR REPLACE	Re-creates the view if it already exists. 
FORCE	Creates the view regardless of whether or not the base table exists.
NO FORCE	Creates the view only if the base table exists (this is default).
WITH CHECK OPTION	Specifies that only rows accessible to the view can be inserted or updated.
WITH READ ONLY	Ensures that no DML operation can be performed on this view.

## CREATE VIEW :

### ➤ Syntax:

CREATE VIEW <views name>

AS

<select query>;

### ➤ Example:

CREATE VIEW emp01

AS

SELECT empno, ename FROM emp WHERE deptno = 10;

## DESCRIBE VIEW:

Describe the structure of the view by using the SQL \* PLUS DESCRIBE Command.

### ➤ DESCRIBE emp01;

## Guidelines for creating a view:

- The subquery that defines a view can contain complex SELECT syntax, including joins, groups and subqueries.

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

## CREATING VIEWS USING ALIASES:

### ➤ Example:

```
CREATE VIEW emp01  
AS  
SELECT empno EMPLOYEE_NUMBER, ename NAME  
FROM emp  
WHERE deptno = 10;
```

OR

```
CREATE VIEW emp01 (EMPLOYEE_NUMBER, NAME)  
AS  
SELECT empno, ename  
FROM emp  
WHERE deptno = 10;
```

## RETRIEVING DATA FROM VIEWS:

### ➤ Example-1:

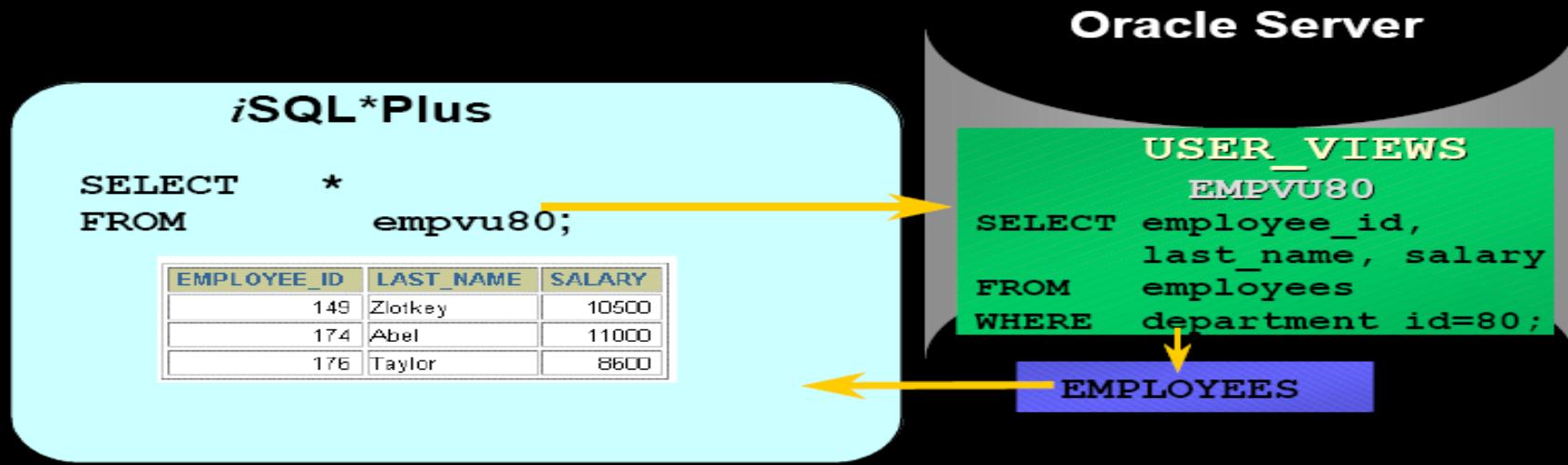
```
SELECT * FROM emp01;
```

### ➤ Example-2:

select the columns from the view by their alias names if the alias names are given at the time of definition of view.

```
SELECT employee_number, name  
FROM emp01;
```

# Querying a View



## Data Access Using Views



When you access data using a view, the Oracle server performs the following operations

1. It retrieves the view definition from the data dictionary table **USER\_VIEWS**.
2. It checks access privilege for the view base table.
3. It converts the view query into an equivalent operation on the underlying base table or tables.

## Views in the Data Dictionary:

- Once your view has been created, you can query the data dictionary view called **USER\_VIEWS** to see the name of the view and the view definition.

### Example:

```
select view_name, text  
from user_views  
where upper(view_name) = 'EMP_VIEW'
```

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

## MODIFYING A VIEW:

- With the ***OR REPLACE*** option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner.

### ➤ Example:

```
CREATE OR REPLACE VIEW emp01  
(EMPLOYEE_NUMBER , NAME)
```

AS

```
SELECT empno, ename FROM emp WHERE deptno = 10;
```

- Column aliases in the CREATE VIEW clause are listed in the same order as the columns in the query.

## CREATING A COMPLEX VIEW:

Create a complex view that contains group functions to display values from two tables.

### Example:

```
CREATE VIEW dept_sum_vu (dname , minsal, maxsal, avgsal)
AS
SELECT d.dname , MIN(e.sal) , MAX(e.sal),AVG(e.sal)
FROM emp e , dept d
WHERE e.deptno = d.deptno
GROUP BY d.dname;
```

## **Rules for Performing DML Operations on a View**

- You can perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A GROUP BY clause
  - The DISTINCT keyword
  - The pseudocolumn ROWNUM keyword

# **Rules for Performing DML Operations on a View**

**You cannot modify data in a view if it contains:**

- Group functions**
- A GROUP BY clause**
- The DISTINCT keyword**
- The pseudocolumn ROWNUM keyword**
- Columns defined by expressions**

# **Rules for Performing DML Operations on a View**

**You cannot add data through a view if the view includes:**

- **Group functions**
- **A GROUP BY clause**
- **The DISTINCT keyword**
- **The pseudocolumn ROWNUM keyword**
- **Columns defined by expressions**
- **NOT NULL columns in the base tables that are not selected by the view**

# Using the WITH CHECK OPTION Clause

- You can ensure that DML operations performed on the view stay within the domain of the view by using the WITH CHECK OPTION clause.

```
CREATE OR REPLACE VIEW empvu20
AS SELECT *
      FROM employees
     WHERE department_id = 20
          WITH CHECK OPTION CONSTRAINT empvu20_ck ;
View created.
```

- Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

## Using the WITH CHECK OPTION Clause:

- The WITH CHECK OPTION clause specifies that INSERT and UPDATES performed through the view cannot create rows which the view cannot select, and therefore it allows integrity constraints and data validation checks to be enforced on data being inserted or updated.

### Example:

```
CREATE OR REPLACE VIEW empv20  
AS  
SELECT * FROM emp  
WHERE deptno = 20  
WITH CHECK OPTION CONSTRAINT empv20_ck;
```

Now, if there is an attempt to perform DML operations on rows that the views has not selected, an error is displayed, with the constraint name if that has been specified.

```
UPDATE empv20  
SET deptno = 10  
WHERE empno = 201;
```

Error: ORA-01402: view WITH CHECK OPTION where\_clause violation.

## DENYING DML OPERATINOS IN VIEWS:

You can ensure that no DML operations occur by adding the WITH READ ONLY option to your view definition.

### Example:

```
CREATE OR REPLACE VIEW emp01
```

```
(EMPLOYEE_NUMBER , NAME)
```

```
AS
```

```
SELECT empno, ename FROM emp WHERE deptno = 10
```

```
WITH READ ONLY;
```

## REMOVING A VIEW:

Remove a view without losing data because view is based on underlying tables in the database.

### Syntax:

```
Drop view <view_name>;
```

### Example:

```
DROP VIEW emp_01;
```

## INLINE VIEW:

- An inline view is created by placing a subquery in the FROM clause and giving that subquery an alias.
- The subquery defines a data source that can be referenced in the main query.

### Example:

Display the employees name , deptno and salaries of all the employees who make more than the average salary of their department.

**SELECT** a.ename,a.sal,a.deptno,b.salavg

**FROM**

emp a,

(SELECT deptno,avg(sal) salavg FROM emp GROUP BY deptno) b

**WHERE**

a.deptno = b.deptno AND a.sal > b.salavg;

## Example:

you can get the same result by a simple sub-query

**SELECT**  
**FROM**  
**WHERE**  
**GROUP BY**

e.ename,e.sal,e.deptno,avg(sal)  
emp e  
e.sal > (select avg(sal)  
from emp f  
where e.deptno = f.deptno  
group by deptno)  
ename,sal,deptno

# Top-N Analysis

- **Top-N queries ask for the  $n$  largest or smallest values of a column. For example:**
  - What are the ten best selling products?
  - What are the ten worst selling products?
- **Both largest values and smallest values sets are considered Top-N queries.**

# Performing Top-N Analysis

The high-level structure of a Top-N analysis query is:

```
SELECT [column_list], ROWNUM
FROM   (SELECT [column_list]
        FROM table
        ORDER BY Top-N_column)
WHERE  ROWNUM <= N;
```

# Example of Top-N Analysis

To display the top three earner names and salaries from the EMPLOYEES table:

```
1          2          3  
SELECT ROWNUM as RANK, last_name, salary  
FROM (SELECT last_name,salary FROM employees  
      ORDER BY salary DESC)  
WHERE ROWNUM <= 3;
```

RANK	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000

1

2

3

# Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

# What Is a Sequence?

## A sequence:

- Automatically generates unique numbers
- Is a sharable object i.e. same sequence can be used for multiple tables
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

# The CREATE SEQUENCE Statement Syntax

**Define a sequence to generate sequential numbers automatically:**

```
CREATE SEQUENCE sequence
    [INCREMENT BY n]
    [START WITH n]
    [{MAXVALUE n | NOMAXVALUE}]
    [{MINVALUE n | NOMINVALUE}]
    [{CYCLE | NOCYCLE}]
    [{CACHE n | NOCACHE}];
```

<b>INCREMENT BY n</b>	Specifies the interval between sequence number (by default incremented by 1)
<b>START WITH n</b>	Specifies the first sequence number to be generated (by default starts with 1)
<b>MAX VALUE n</b>	Specifies the max. value the sequence can generate.
<b>NOMAXVALUE</b>	Specifies the max. value of $10^{27}$ for an ascending sequence and -1 for a descending sequence.
<b>MIN VALUE n</b>	Specifies the minimum sequence value.
<b>NOMINVALUE</b>	Specifies the min. value of 1 for an ascending sequence and $-10^{26}$ for a descending sequence.
<b>CYCLE/NO CYCLE</b>	Specifies whether the sequence continues to generate values after reaching its max. or min. value (NOCYCLE is the default option)
<b>CACHE n   NOCACHE</b>	Specifies how many values the ORACLE server preallocates and keep in memory. (By deafault Oracle server preallocates 20 values.)

# Creating a Sequence

- Create a sequence named DEPT\_DEPTID\_SEQ to be used for the primary key of the DEPARTMENTS table.
- Do not use the CYCLE option.

```
CREATE SEQUENCE dept_deptid_seq
    INCREMENT BY 10
    START WITH 120
    MAXVALUE 9999
    NOCACHE
    NOCYCLE;
```

Sequence created.

# Confirming Sequences

- Verify your sequence values in the USER\_SEQUENCES data dictionary table.

```
SELECT    sequence_name, min_value, max_value,  
          increment_by, last_number  
FROM      user_sequences;
```

- The LAST\_NUMBER column displays the next available sequence number if NOCACHE is specified.

# **NEXTVAL and CURRVAL Pseudocolumns**

- **NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.**
- **CURRVAL obtains the current sequence value.**
- **NEXTVAL must be issued for that sequence before CURRVAL contains a value.**

# Using a Sequence

- Insert a new department named “Support” in location ID 2500.

```
INSERT INTO departments(department_id,  
                      department_name, location_id)  
VALUES      (dept_deptid_seq.NEXTVAL,  
                      'Support', 2500);
```

1 row created.

- View the current value for the DEPT\_DEPTID\_SEQ sequence.

```
SELECT      dept_deptid_seq.CURRVAL  
FROM        dual;
```

## Example:

### Step-1: Create a table.

```
CREATE TABLE emp01  
(empno number, ename varchar2(25));
```

### Step-2: Insert values into the table.

```
INSERT INTO emp01(ename) VALUES('a');  
INSERT INTO emp01(ename) VALUES('b');  
INSERT INTO emp01(ename) VALUES('c');  
INSERT INTO emp01(ename) VALUES('d');  
INSERT INTO emp01(ename) VALUES('e');  
INSERT INTO emp01(ename) VALUES('f');  
INSERT INTO emp01(ename) VALUES('g');  
INSERT INTO emp01(ename) VALUES('h');
```

### Step-3: SELECT \* FROM emp01;

EMPNO	ENAME
a	
b	
c	
d	
e	
f	
g	
h	

### Step-4: Create a sequence.

```
CREATE SEQUENCE emp_seq  
START WITH 10  
INCREMENT BY 10  
MAXVALUE 40
```

CYCLE

CACHE 4

### Step-5: Update the table

```
UPDATE emp01  
SET empno = emp_seq.nextval;
```

### Step-6: SELECT \* FROM emp01;

EMPNO	ENAME
10	a
20	b
30	c
40	d
1	e
11	f
21	g
31	h

Values must be  
10,20,30,40 but since  
no MINVALUE is defined  
in the sequence so we get  
the values in the next  
cycle start with 1

## Step-7: Update the table

```
UPDATE emp01  
SET empno = NULL;
```

## Step-8: Alter the sequence

```
ALTER SEQUENCE emp_seq  
MINVALUE 10  
CACHE 3;
```

## Step-9: Update the table

```
UPDATE emp01  
SET empno = emp_seq.nextval;
```

## Step-10: SELECT \* FROM emp01;

EMPNO	ENAME
10	a
20	b
30	c
40	d
<b>10</b>	e
<b>20</b>	f
<b>30</b>	g
<b>40</b>	h

## Step-11 Alter the sequence

```
ALTER SEQUENCE emp_seq  
NOCYCLE;
```

## Step-12: Update the table

```
UPDATE emp01  
SET empno = emp_seq.nextval;
```

### ERROR:

sequence EMP\_SEQ.NEXTVAL exceeds MAXVALUE and cannot be instantiated

# Using a Sequence

- **Caching sequence values in memory gives faster access to those values.**
- **Gaps in sequence values can occur when:**
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table
- **If the sequence was created with NOCACHE, view the next available value, by querying the `USER_SEQUENCES` table.**

# Modifying a Sequence

**Change the increment value, maximum value, minimum value, cycle option, or cache option.**

```
ALTER SEQUENCE dept_deptid_seq  
    INCREMENT BY 20  
    MAXVALUE 999999  
    NOCACHE  
    NOCYCLE;
```

**Sequence altered.**

# **Guidelines for Modifying a Sequence**

- You must be the owner or have the ALTER privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.  
Example: new MAXVALUE that is less than the current sequence number cannot be performed.

# Removing a Sequence

- Remove a sequence from the data dictionary by using the **DROP SEQUENCE** statement.
- Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptid_seq;  
Sequence dropped.
```

# **What is an Index?**

**An index:**

- **Is a schema object**
- **Is used by the Oracle server to speed up the retrieval of rows by using a pointer**
- **Can reduce disk I/O by using a rapid path access method to locate data quickly**
- **Is independent of the table it indexes**
- **Is used and maintained automatically by the Oracle server**

# How Are Indexes Created?

- **Automatically:** A unique index is created automatically when you define a PRIMARY KEY or UNIQUE constraint in a table definition.
- **Manually:** Users can create nonunique indexes on columns to speed up access to the rows.

## **CREATING INDEX:**

### **Syntax:**

- `CREATE INDEX <index_name>  
ON TABLE (column, column, ....)`

### **Example of Non-Unique Index:**

```
CREATE INDEX emp_ename_idx  
ON emp(ename);
```

### **Example of Unique Index:**

```
CREATE UNIQUE INDEX unique_ind  
ON dummy(empno);
```

## **REMOVING AN INDEX:**

Remove an index from the data dictionary.

```
DROP INDEX <index_name>;
```

## **ALTERING TABLE:**

```
ALTER TABLE sales ADD CONSTRAINT sales_unique UNIQUE (sales_id)  
DISABLE VALIDATE;
```

# When to Create an Index

**You should create an index if:**

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a WHERE clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4 percent of the rows

**Note:** The more indexes you have associated with a table, the more effort the oracle server must make to update all the indexes after a DML operation.

# When Not to Create an Index

**It is usually not worth creating an index if:**

- **The table is small**
- **The columns are not often used as a condition in the query**
- **Most queries are expected to retrieve more than 2 to 4 percent of the rows in the table**
- **The table is updated frequently**
- **The indexed columns are referenced as part of an expression**

# Confirming Indexes

- The **USER\_INDEXES** data dictionary view contains the name of the index and its uniqueness.
- The **USER\_IND\_COLUMNS** view contains the index name, the table name, and the column name.

```
SELECT      ic.index_name, ic.column_name,  
            ic.column position col_pos, ix.uniqueness  
FROM        user indexes ix, user ind columns ic  
WHERE       ic.index_name = ix.index_name  
AND         ic.table_name = 'EMPLOYEES';
```

# Function-Based Indexes

- A function-based index is an index based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```
CREATE INDEX upper_dept_name_idx  
ON departments(UPPER(department_name)) ;
```

Index created.

```
SELECT *  
FROM   departments  
WHERE  UPPER(department_name) = 'SALES' ;
```

# Synonyms

**Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:**

- Ease referring to a table owned by another user
- Shorten lengthy object names

```
CREATE [PUBLIC] SYNONYM synonym
FOR      object;
```

- PUBLIC synonym is available to all users

# Creating and Removing Synonyms

- Create a shortened name for the DEPT\_SUM\_VU view.

```
CREATE SYNONYM d_sum  
FOR dept_sum_vu;  
Synonym Created.
```

- Drop a synonym.

```
DROP SYNONYM d_sum;  
Synonym dropped.
```

- If any other user wants to SELECT data from the EMP table, the syntax they must use it:

```
SELECT *
```

```
FROM scott.emp;
```

- Alternatively they can create synonyms for the table and SELECT from the synonyms:

- i. CREATE SYNONYM employees FOR scott.emp;
- ii. SELECT \* from employees;

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# **FLASHBACK TABLE Statement**

- Repair tool for accidental table modifications
  - Restores a table to an earlier point in time
  - Benefits: Ease of use, availability, fast execution
  - Performed in place
- Syntax:

```
FLASHBACK TABLE [schema.] table [,  
[ schema.] table ]...  
TO { TIMESTAMP | SCN } expr  
[ { ENABLE | DISABLE } TRIGGERS ];
```

## FLASHBACK TABLE Statement

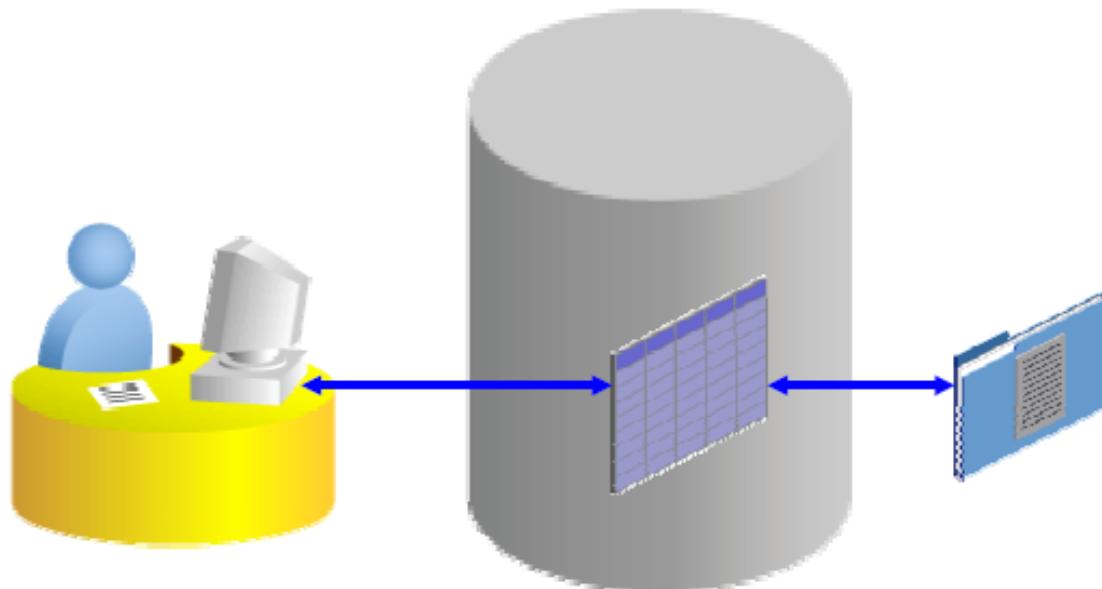
```
DROP TABLE emp2;  
DROP TABLE succeeded.
```

```
SELECT original_name, operation, droptime  
FROM recyclebin;
```

ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2008-11-13:08:44:39

```
FLASHBACK TABLE emp2 TO BEFORE DROP;  
FLASHBACK TABLE succeeded.
```

# External Tables



## Creating a Directory for the External Table

Create a DIRECTORY object that corresponds to the directory on the file system where the external data source resides.

```
CREATE OR REPLACE DIRECTORY emp_dir  
AS '/.../emp_dir';  
  
GRANT READ ON DIRECTORY emp_dir TO hr;
```

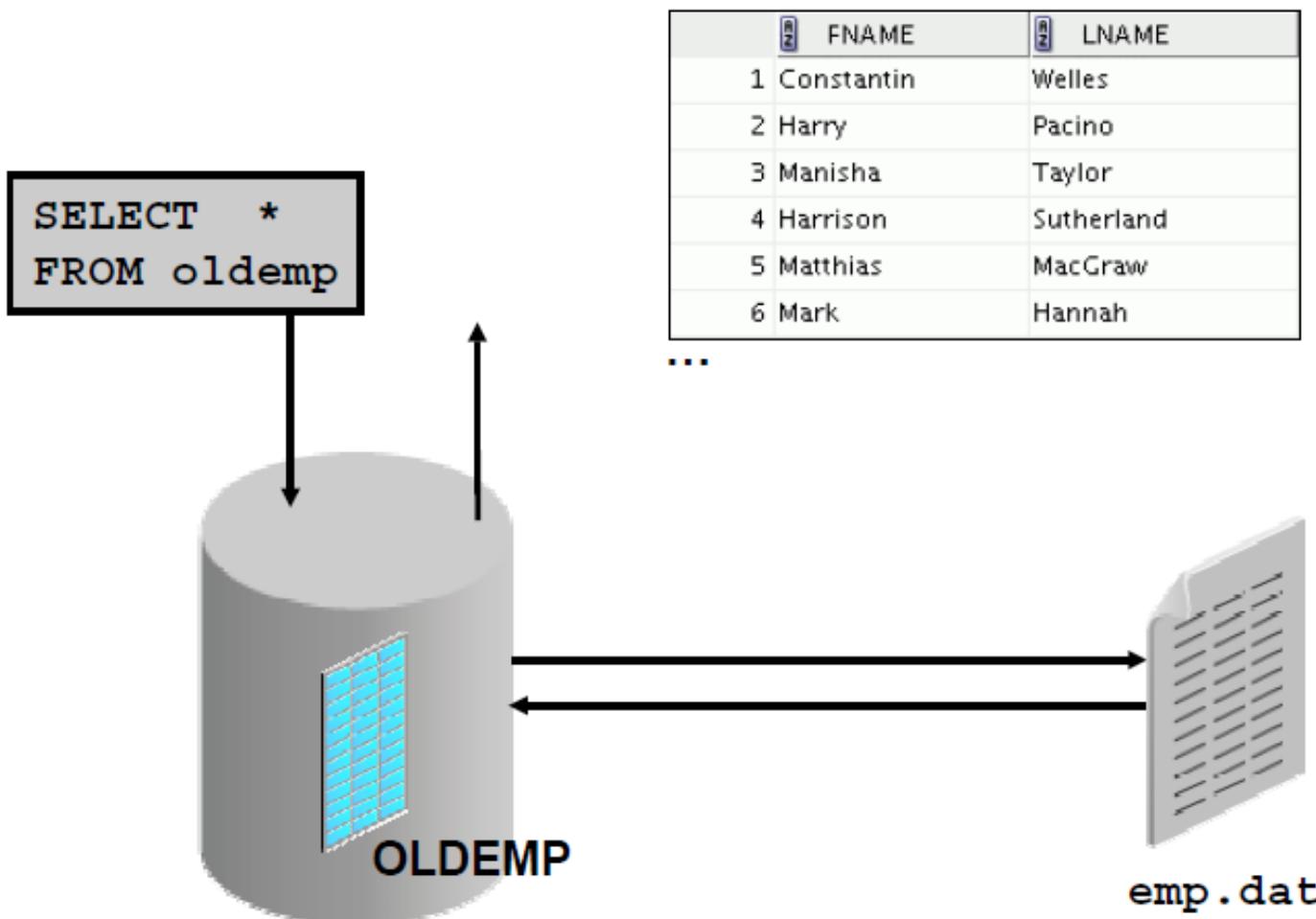
# Creating an External Table

```
CREATE TABLE <table_name>
  ( <col_name> <datatype>, ... )
ORGANIZATION EXTERNAL
  (TYPE <access_driver_type>
  DEFAULT DIRECTORY <directory_name>
  ACCESS PARAMETERS
    (... ) )
  LOCATION ('<locationSpecifier>') )
REJECT LIMIT [0 | <number> | UNLIMITED];
```

# Creating an External Table by Using ORACLE\_LOADER

```
CREATE TABLE oldemp (
    fname char(25), lname CHAR(25))
ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
  DEFAULT DIRECTORY emp_dir
  ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE
    NOBADFILE
    NOLOGFILE
    FIELDS TERMINATED BY ','
      (fname POSITION ( 1:20) CHAR,
       lname POSITION (22:41) CHAR))
  LOCATION ('emp.dat'))
  PARALLEL 5
  REJECT LIMIT 200;
CREATE TABLE succeeded.
```

# Querying External Tables



## External Tables:

- Oracle 10g external table feature enables you to use external data as a ***virtual table*** that can be queried and joined directly and in parallel without requiring the external data to be first loaded in the database.
- External tables enable the ***pipelining*** of the loading phase with the transformation phase.
- The ***transformation process can be merged with the loading process*** without any interruption of the data streaming.
- It is ***no longer necessary to stage the data*** inside the database for further processing inside the database, such as comparison or transformation.
- ***For example***, the conversion functionality of a conventional load can be used for a direct-path INSERT AS SELECT statement in conjunction with the SELECT from an external table.

## Difference between REGULAR and EXTERNAL TABLES:

- The main difference between external tables and regular tables is that externally organized tables are read-only. No DML operations (UPDATE/INSERT/DELETE) are possible and no indexes can be created on them.
- It contains no data while regular table contains the data.

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

## Steps of External Table:

### Step-1: Accessing the External Data

- To access external files from within oracle, you must first use the ***create directory*** command to define a directory object pointing to the external file location.
- Users who will access the external files must have the READ privilege on the directory.
- External directory to which CREATE DIRECTORY command points out should exists.

Create directory EMP\_EXT as ‘c:\dump’;

Grant read on directory EMP\_EXT to SCOTT;

Grant write on directory EMP\_EXT to SCOTT; //write privileges for log and bad files

## Step-2: Source Data

- The following listing generates a file for sample data from the EMP.

```
Set pagesize 0 linesize 120 newpage 0 feedback off  
select empno||'-'||ename||' ' from emp;  
spool c:\dump\employee.dat  
/  
spool off
```

## Step-3: Creating an External Table

- Now that the external data is available and accessible, we can create a table structure that accesses it.
- To do so, you need to use the *organization external* clause of the *create table* command.
- Within the *organization external* clause, you can specify the data structure much as you would for a SQL\*Loader control file.

## Example:

```
create table EMP_EXT
```

```
(  
    empno varchar2(4),  
    ename varchar2(10)  
)
```

*organization external*

```
(  
    type oracle_loader  
    default directory employee  
    access parameters  
        (  
            records delimited by newline  
            fields terminated by "-"  
                (empno char(4),  
                 ename char(10))  
        )  
    location ('employee.dat')  
)
```

- The EMP\_EXT table points to the EMPLOYEE.DAT file.
- If you alter the data in the file, the data in EMP\_EXT will change.
- We can query external tables the same way we query standard tables i.e. in join, as part of views and so on.
- Functions can also be performed on external tables columns with in the SQL just like the standard tables.

### Queries about External Tables:

- We can see the external table information through USER\_EXTERNAL\_TABLES data dictionary view.

```
Select * from USER_EXTERNAL_TABLES  
Where table_name = 'EMP_EXT';
```

- USER\_EXTERNAL\_TABLES does not show the name of the external file the able references. To see that information, query USER\_EXTERNAL\_LOCATIONS.

```
Select * from USER_EXTERNAL_LOCATIONS;
```

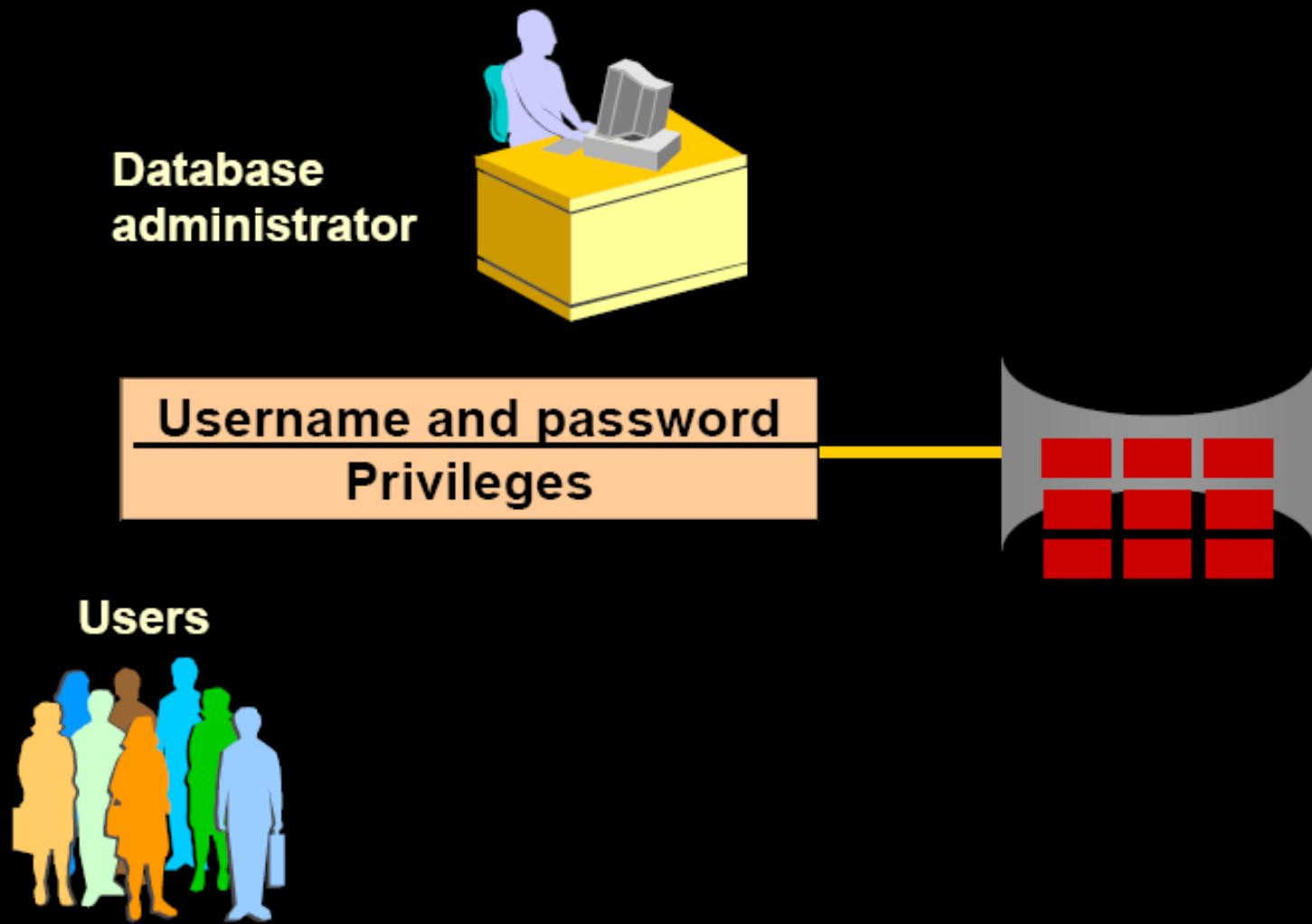
- The external table can now be used from within the database, accessing some columns of the external data only, grouping the data, and inserting it into the fact table.

```
INSERT INTO COSTS          // cost is a fact table
(
TIME_ID,
PROD_ID,
UNIT_COST,
UNIT_PRICE
)
SELECT
TIME_ID,PROD_ID,SUM(UNIT_COST),SUM(UNIT_PRICE)
FROM sales_transactions_ext // sales_transactions_ext is an
                           // external table
GROUP BY time_id, prod_id;
```

# LESSON 13

# Controlling User Access

# Controlling User Access



## Controlling User Access:

- In a multi-user environment, you want to maintain security of the database access and use.
- With Oracle Server database security, you can do the following:
  - ✓ Control database access.
  - ✓ Give access to specific objects in the database.
  - ✓ Confirm given and received privileges with the Oracle Data Dictionary.
  - ✓ Create synonyms for database objects.

## PRIVILEGES:

- Privileges are the rights to execute particular SQL statements.
- **DBA** is a high level user with the ability to grant access to the database and its objects.
- User requires **System Privileges** to gain access to the database and **Object Privileges** to manipulate the contents of the database objects.
- Users can also be given the privilege to grant additional privileges to other users or to roles, which are named groups of related privileges.

# Privileges

- **Database security:**
  - **System security**
  - **Data security**
- **System privileges: Gaining access to the database**
- **Object privileges: Manipulating the content of the database objects**
- **Schemas: Collections of objects, such as tables, views, and sequences**

# Database Security:

Database security can be classified into two categories:

1. **System Security:** It covers access and use of the database at the system level, such as username and password, the disk space allocated to users and the system operations that users can perform.
2. **Data Security:** It covers access and use of the database objects and the actions that those users can have on the objects.

# System Privileges

- **More than 100 privileges are available.** (Approx. 157)
- **The database administrator has high-level system privileges for tasks such as:**
  - Creating new users
  - Removing users
  - Removing tables
  - Backing up tables

# TYPICAL DBA PRIVILEGES

SYSTEM PRIVILEGE	OPERATIONS AUTHORIZED
CREATE USER	Grantee can create other Oracle users (a privilege required for a DBA role).
DROP USER	Grantee can drop another user.
DROP ANY TABLE	Grantee can drop a table in a schema.
BACKUP ANY TABLE	Grantee can backup any table in any schema with the export utility.
SELECT ANY TABLE	Grantee can query tables, views, or snapshots in any schema.
CREATE ANY TABLE	Grantee can create tables in any schema.



©

# Creating Users

**The DBA creates users by using the CREATE USER statement.**

```
CREATE USER user  
IDENTIFIED BY password;
```

```
CREATE USER scott  
IDENTIFIED BY tiger;  
User created.
```

- The newly created user does not have any privileges by default. The DBA can grant privileges to that user.

# User System Privileges

- Once a user is created, the DBA can grant specific system privileges to a user.

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];
```

- An application developer, for example, may have the following system privileges:
  - CREATE SESSION
  - CREATE TABLE
  - CREATE SEQUENCE
  - CREATE VIEW
  - CREATE PROCEDURE

# TYPICAL SYSTEM / USER PRIVILEGES

SYSTEM PRIVILEGE	OPERATIONS AUTHORIZED
CREATE SESSION	Connect to the database
CREATE TABLE	Create tables in the user's schema
CREATE SEQUENCE	Create a sequence in the user's schema
CREATE VIEW	Create a view in the user's schema
UNLIMITED TABLESPACE	Assign table space to create object and apply DML operations.
CREATE PROCEDURE	Create a stored procedure, function, or package in the user's schema

©

2005 - 2016. All Rights Reserved

## Note:

- Current system privileges can be found in the data dictionary view **SESSION\_PRIVS**.

# Granting System Privileges

The DBA can grant a user specific system privileges.

```
GRANT  create session, create table,  
       create sequence, create view  
TO     scott;  
Grant succeeded.
```

## 2) Object Privileges:

- Manipulate the **contents of the objects** in the database.
- Object privileges vary from object to object.
- An owner of the object has all the privileges on the object by default.
- An object owner can grant any object privilege on the object to any other user or role of the database.

### Syntax:

```
GRANT      {object_privilege|ALL}    [ (column) ]  
ON object  
TO {user | role |PUBLIC}  
[WITH GRANT OPTION];
```

### Example:

```
GRANT UPDATE (dname , loc) ON dept TO scott , manager;
```

### Note:

- DBA generally allocates system privileges.
- Any user who owns an object can grant object privileges.

# Granting Object Privileges

- Grant query privileges on the EMPLOYEES table.

```
GRANT select  
ON employees  
TO sue, rich;  
Grant succeeded.
```

- Grant privileges to update specific columns to users and roles.

```
GRANT update (department_name, location_id)  
ON departments  
TO scott, manager;  
Grant succeeded.
```

# Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	✓		✓	
DELETE	✓	✓		
EXECUTE				✓
INDEX	✓			
INSERT	✓	✓		
REFERENCES	✓	✓		
SELECT	✓	✓	✓	
UPDATE	✓	✓		

# Using the WITH GRANT OPTION and PUBLIC Keywords

- Give a user authority to pass along privileges.

```
GRANT select, insert  
ON departments  
TO scott  
WITH GRANT OPTION;  
Grant succeeded.
```

- Allow all users on the system to query data from Alice's DEPARTMENTS table.

```
GRANT select  
ON alice.departments  
TO PUBLIC;  
Grant succeeded.
```

# Confirming Privileges Granted

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_REC'D	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_REC'D	Object privileges granted to the user on specific columns
USER_SYS_PRIVS	Lists system privileges granted to the user

# HOW TO REVOKE OBJECT PRIVILEGES:

- REVOKE statement is used to revoke privileges granted to the other users.
- Privileges granted to others through the WITH GRANT OPTION will also be revoked.

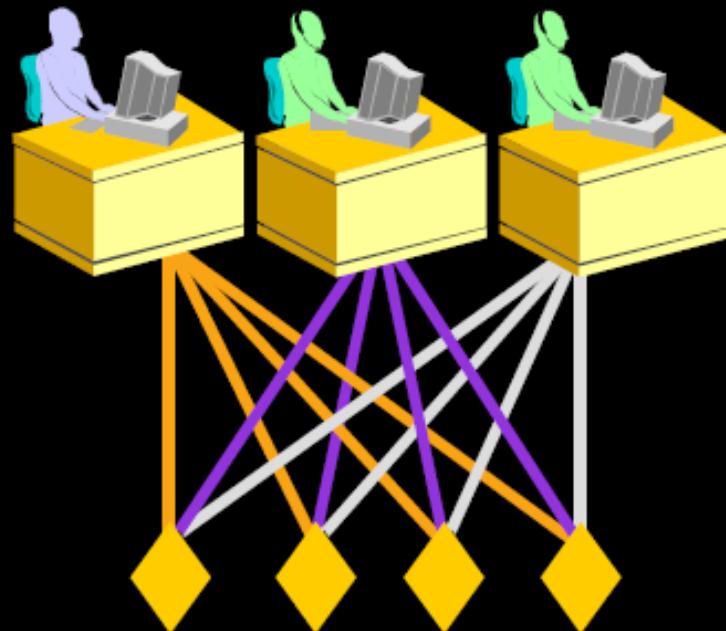
## Syntax:

```
REVOKE {privilege , privilege , .... | ALL }  
ON object  
FROM {user , user , ....., | role | PUBLIC}
```

## Example:

```
REVOKE select , insert  
ON dept  
FROM scott;
```

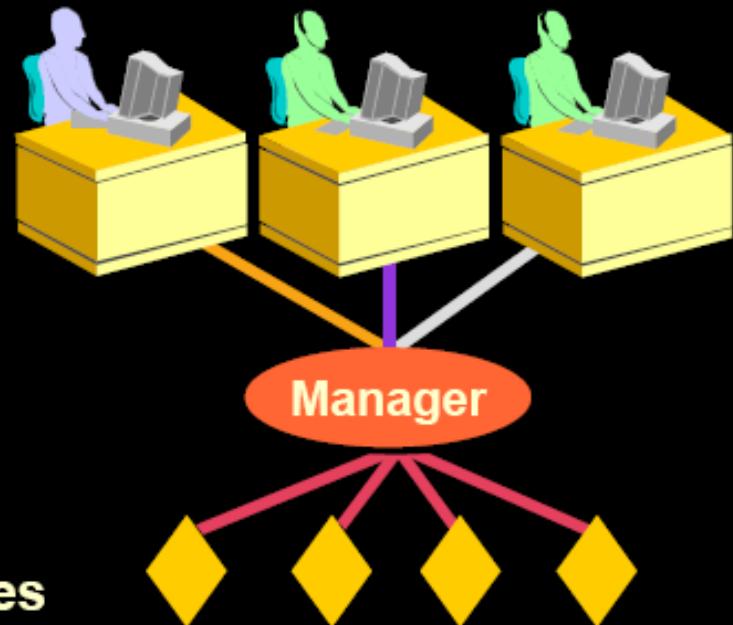
# What is a Role?



**Allocating privileges  
without a role**

Users

Privileges



**Allocating privileges  
with a role**

## **ROLE:**

- A role is a named **group of related privileges** that can be granted to the user.
- The method makes granting and revoking privileges easier to perform and maintain.
- A user can have access to several roles.
- Several users can be assigned to the same role.

## **CREATING AND ASSIGNING A ROLE:**

**Step-1:** First, the DBA must create the role.

**Step-2:** Second, the DBA can assign privileges to the role.

**Step-3:** Third, the role is assigned to the users.

# Creating and Granting Privileges to a Role

- **Create a role**

```
CREATE ROLE manager;  
Role created.
```

- **Grant privileges to a role**

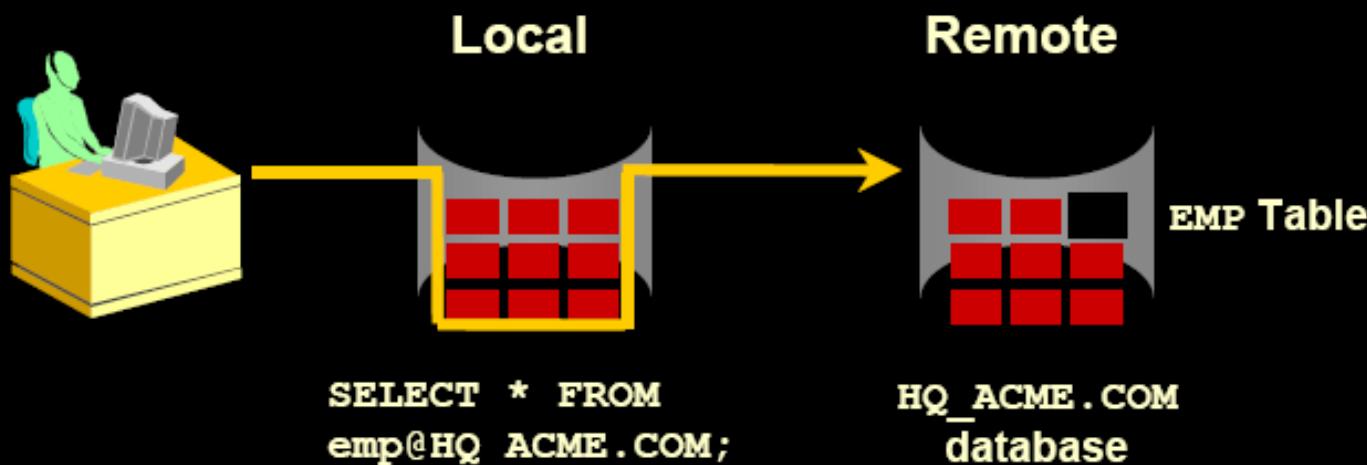
```
GRANT create table, create view  
TO manager;  
Grant succeeded.
```

- **Grant a role to users**

```
GRANT manager TO DEHAAN, KOCHHAR;  
Grant succeeded.
```

# Database Links

A database link connection allows local users to access data on a remote database.



## What Are Database Links?

- A database link connection gives local users access to data on a remote database.
- A database link is a pointer that defines a one-way communication path from an Oracle database server to another database server.
- The link pointer is actually defined as an entry in a data dictionary table.
- To access the link, you must be connected to the local database that contains the data dictionary entry.
- A database link connection is **one-way** in the sense that a client connected to local database **A** can use a link stored in database **A** to access information in remote database **B**, but users connected to database **B** cannot use the same link to access data in database **A**.  
If local users on database **B** want to access data on database **A**, then they must define a link that is stored in the data dictionary of database **B**.
- The dictionary view **USER\_DB\_LINKS** contains information on links to which a user has access.

## Step of Linking:

- 1) Create a database service of the remote database on the local server through **NET CONFIGURATION ASSISTANT**.
- 2) CREATE DATABASE LINK db\_link1  
CONNECT TO scott IDENTIFIED BY tiger  
USING 'almighty';
- 3) CREATE SYNONYM employee FOR emp@db\_link1;
- 4) select \* from employee;
- 5) DROP DATABASE LINK db\_link1;

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# **END OF COURSE FOR PAPERS**

<b>Exam Number</b>	:	1Z0-007
<b>Exam Name</b>	:	Introduction to oracle10g: SQL
<b>Duration</b>	:	120 Minutes.
<b>Number of Questions</b>	:	52
<b>Passing Score</b>	:	71%
<b>Exam Price</b>	:	95 U.S. Dollar

# LESSON 17

## Enhancement to the GROUP BY Clause

# Review of Group Functions

**Group functions operate on sets of rows to give one result per group.**

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

## Example:

```
SELECT AVG(salary), STDDEV(salary),
       COUNT(commission_pct), MAX(hire_date)
FROM employees
WHERE job_id LIKE 'SA%';
```

# Review of the GROUP BY Clause

## Syntax:

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

## Example:

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

# Review of the HAVING Clause

```
SELECT      [column,] group_function(column)...
FROM        table
[WHERE       condition]
[GROUP BY   group by expression]
[HAVING     having expression]
[ORDER BY   column];
```

- Use the HAVING clause to specify which groups are to be displayed.
- You further restrict the groups on the basis of a limiting condition.

## **GROUP BY with ROLLUP and CUBE Operators**

- **Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.**
- **ROLLUP grouping produces a results set containing the regular grouped rows and the subtotal values.**
- **CUBE grouping produces a results set containing the rows from ROLLUP and cross-tabulation rows.**

# ROLLUP Operator

```
SELECT      [column,] group_function(column) . . .
FROM        table
[WHERE       condition]
[GROUP BY   [ROLLUP] group_by_expression]
[HAVING     having_expression];
[ORDER BY   column];
```

- ROLLUP is an extension to the GROUP BY clause.
- Use the ROLLUP operation to produce cumulative aggregates, such as subtotals.

# ROLLUP Operator Example

```
SELECT      department_id, job_id, SUM(salary)
FROM        employees
WHERE       department_id < 60
GROUP BY   ROLLUP(department_id, job_id);
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
		4400
20	MK_MAN	13000
	MK_REP	6000
50	ST_CLERK	11700
	ST_MAN	5800
		17500
		40900

1      2      3

9 rows selected.

➤ select job,deptno,sum(sal) from emp group by rollup(job,deptno)

JOB	DEPTNO	SUM(SAL)	
CLERK	10	1300	(Total of Clerks of Deptno = 10)
CLERK	20	1900	(Total of Clerks of Deptno = 20)
CLERK	30	950	(Total of Clerks of Deptno = 30)
CLERK		4150	(Total of Clerks)
ANALYST	20	6000	
ANALYST		6000	
MANAGER	10	2450	
MANAGER	20	2975	
MANAGER	30	2850	
MANAGER		8275	
SALESMAN	30	5600	
SALESMAN		5600	
PRESIDENT	10	5000	
PRESIDENT		5000	
		29025	(Grand Total)

➤ select job,deptno,sum(sal) from emp group by rollup(deptno,job)

<b>JOB</b>	<b>DEPTNO</b>	<b>SUM(SAL)</b>	
CLERK	10	1300	(Total of Clerks of Deptno = 10)
MANAGER	10	2450	(Total of Managers of Deptno = 10)
PRESIDENT	10	5000	(Total of Presidents of Deptno = 10)
	10	8750	(Total of Deptno = 10)
CLERK	20	1900	
ANALYST	20	6000	
MANAGER	20	2975	
	20	10875	
CLERK	30	950	
MANAGER	30	2850	
SALESMAN	30	5600	
	30	9400	
		29025	(Grand Total)

**Note:**

There is no sub-total of every job  
like clerk,manager,president and salesman



- The action of ROLLUP is straightforward: it creates subtotals that roll up from the most detailed level to a grand total, following a grouping list specified in the ROLLUP clause.
- ROLLUP creates *subtotals at  $n+1$  levels*, where *n* is the **number of grouping columns**.
- ROLLUP takes as its argument an ordered list of grouping columns.
- *First*, it calculates the standard aggregate values specified in the GROUP BY clause.
- *Then*, it creates progressively higher-level subtotals, *moving from right to left* through the list of grouping columns.
- *Finally*, it creates a grand total.

```
select job,deptno,sum(sal)
from emp
group by rollup(job,deptno)
order by job
```

JOB	DEPTNO	SUM(SAL)
ANALYST	20	6000
ANALYST		6000
CLERK	10	1300
CLERK	20	9900
CLERK	30	950
CLERK		12150
MANAGER	10	2450
MANAGER	20	2975
MANAGER	30	2850
MANAGER		8275
MGR	10	6000
MGR		6000
PRESIDENT	10	5000
PRESIDENT		5000
SALESMAN	30	5600
SALESMAN		5600

## Same query without using ROLLUP:

Through the following query, you will get the same result what you get through the ROLLUP in the previous slide.

```
select job,deptno,sum(sal)  
from emp  
group by (job,deptno)
```

```
union all
```

```
select job,to_number(null),sum(sal)  
from emp  
group by job
```

```
union all
```

```
select to_char(null),to_number(null),sum(sal)  
from emp
```

```
order by job
```

# CUBE Operator

```
SELECT      [column,] group_function(column)...
FROM        table
[WHERE       condition]
[GROUP BY   [CUBE] group_by_expression]
[HAVING      having_expression]
[ORDER BY   column];
```

- CUBE is an extension to the GROUP BY clause.
- You can use the CUBE operator to produce cross-tabulation values with a single SELECT statement.

## The CUBE operator:

- CUBE operator is an additional switch in the GROUP BY clause in a select statement.
- CUBE is used to produce result sets that are typically used for cross-tabular reports.
- ROLLUP produces only a fraction of possible subtotal combinations, while CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause and a grand total.
- If you have  $n$  columns or expressions in the GROUP BY clause, there will be  $2^n$  possible superaggregate combinations.

Channel	Country		
	UK	US	Total
Direct Sales	1,378,126	2,835,557	4,213,683
Internet	911,739	1,732,240	2,643,979
Total	2,289,865	4,567,797	6,857,662
	Sub Total for UK	Sub Total for US	

**Table 16–1 Simple Cross-Tabular Report With Subtotals**

# CUBE Operator: Example

```
SELECT      department_id, job_id, SUM(salary)
FROM        employees
WHERE       department_id < 60
GROUP BY    CUBE (department_id, job_id) ;
```

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
		4400
20	MK_MAN	13000
	MK_REP	6000
20		19000
50	ST_CLERK	11700
	ST_MAN	5800
50		17500
AD_ASST		4400
	MK_MAN	13000
	MK_REP	6000
	ST_CLERK	11700
	ST_MAN	5800
		40900

14 rows selected.

select job,deptno,sum(sal) from emp group by cube(job,deptno)

<b>JOB</b>	<b>DEPTNO</b>	<b>SUM(SAL)</b>	
		29025	(Grand Total)
	10	8750	(Total of deptno = 10)
	20	10875	(Total of deptno = 20)
	30	9400	(Total of deptno = 30)
CLERK		4150	(Total of Clerks)
CLERK	10	1300	(Total of Clerks of deptno = 10)
CLERK	20	1900	(Total of Clerks of deptno = 20)
CLERK	30	950	(Total of Clerks of deptno = 30)
ANALYST		6000	(Total of Analysts)
ANALYST	20	6000	(Total of Analyst of deptno = 20)
MANAGER		8275	(Total of Managers)
MANAGER	10	2450	(Total of Managers of deptno = 10)
MANAGER	20	2975	(Total of Managers of deptno = 20)
MANAGER	30	2850	(Total of Managers of deptno = 30)
SALESMAN		5600	(Total of Salesmans)
SALESMAN	30	5600	(Total of Salesman of deptno = 30)
PRESIDENT		5000	(Total of President)
PRESIDENT	10	5000	(Total of President of deptno = 30)

## Same query without using CUBE:

Through the following query, you will get the same result what you get through the CUBE in the previous slide.

```
select to_char(null) job,to_number(null) deptno,sum(sal),1  
from emp
```

```
union all
```

```
select to_char(null),deptno,sum(sal),2  
from emp  
group by deptno
```

```
union all
```

```
select job,to_number(null),sum(sal),3  
from emp  
group by job
```

```
union all
```

```
select job,deptno,sum(sal),4  
from emp  
group by (job,deptno)  
order by 4
```

# GROUPING Function

```
SELECT      [column,] group_function(column) . ,  
            GROUPING(expr)  
FROM        table  
[WHERE      condition]  
[GROUP BY  [ROLLUP] [CUBE] group_by_expression]  
[HAVING    having_expression]  
[ORDER BY  column];
```

- The GROUPING function can be used with either the CUBE or ROLLUP operator.
- Using the GROUPING function, you can find the groups forming the subtotal in a row.
- Using the GROUPING function, you can differentiate stored NULL values from NULL values created by ROLLUP or CUBE.
- The GROUPING function returns 0 or 1.

# GROUPING Functions

Two challenges arise with the use of ROLLUP and CUBE.

- First,
  - ✓ How can you programmatically determine which result set rows are subtotals?
- Second,
  - ✓ What happens if query results contain both stored NULL values and "NULL" values created by a ROLLUP or CUBE?

## GROUPING Function

- GROUPING handles these problems.
- Using a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the group by clause.
- Grouping function returns only 0 or 1.
- The values returned by the GROUPING function are useful to:
  - ✓ Determine the level of aggregation of a given subtotal, that is, the group or groups on which subtotal is based.
  - ✓ Identify whether a NULL value in the expression column of a row of the result set indicates:
    - ❖ A NULL value from the base table (stored NULL value)
    - ❖ A NULL value created by ROLLUP/CUBE.

- It ***returns 0*** when:
  - ✓ The expression has been used to calculate the aggregate value.
  - ✓ The NULL value in the expression column is a stored null value.
  
- It ***returns 1*** when:
  - ✓ The expression has not been used to calculate the aggregate value.
  - ✓ The NULL value in the expression column is created by ROLLUP or CUBE as a result of grouping.

*Tanveer Khan, 2005 - 2016. All Rights Reserved*

➤ select job,deptno,sum(sal) ,  
grouping (job) as j,  
grouping (deptno) as dno  
from emp group by rollup(deptno,job)

**NOTE:**

This is example of the table having *an* employee  
with a job = NULL

JOB	DEPTNO	SUM(SAL)	J	DNO
CLERK	10	6000	0	0
MANAGER	10	1300	0	0
PRESIDENT	10	2450	0	0
	10	5000	0	0
	10	<b>14750</b>	<b>1</b>	<b>0</b>
				(JOB = NULL)
CLERK	20	1900	0	0
ANALYST	20	6000	0	0
MANAGER	20	2975	0	0
	20	10875	1	0
				(SUB-TOTAL)
CLERK	30	950	0	0
MANAGER	30	2850	0	0
SALESMAN	30	5600	0	0
	30	9400	1	0
		35025	1	1

➤ select job,deptno,sum(sal) ,  
grouping (job) as j,  
grouping (deptno) as dno  
from emp group by rollup(deptno,job);

**NOTE:**

This is example of the table having **no** employee  
with a job = NULL

JOB	DEPTNO	SUM(SAL)	J	DNO
CLERK	10	1300	0	0
MANAGER	10	2450	0	0
PRESIDENT	10	5000	0	0
	10	8750	1	0
CLERK	20	1900	0	0
ANALYST	20	6000	0	0
MANAGER	20	2975	0	0
	20	10875	1	0
CLERK	30	950	0	0
MANAGER	30	2850	0	0
SALESMAN	30	5600	0	0
	30	9400	1	0
		29025	1	1

## **GROUPING SETS**

- GROUPING SETS are a further extension of the GROUP BY clause.
- You can use GROUPING SETS to define multiple groupings in the same query.
- The Oracle Server computes all groupings specified in the GROUPING SETS clause and combines the results of individual groupings with a UNION ALL operation.
- Grouping set efficiency:
  - Only one pass over the base table is required.
  - There is no need to write complex UNION statements.
  - The more elements the GROUPING SETS have, the greater the performance benefit.

# GROUPING SETS: Example

```
SELECT      department_id, job_id,  
            manager_id, avg(salary)  
FROM        employees  
GROUP BY   GROUPING SETS  
((department_id,job_id), (job_id,manager_id));
```

DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
10	AD_ASST		4400
20	MK_MAN		13000
20	MK_REP		6000
60	ST_CLERK		2925
...			
	SA_MAN	100	10500
	SA_REP	149	8866.666667
	ST_CLERK	124	2925
	ST_MAN	100	5800

26 rows selected.

1

2

```
select deptno,job,sum(sal)
from emp
group by grouping sets ((deptno,job),(job));
```

DEPTNO	JOB	SUM(SAL)
10	MGR	6000
	MGR	6000
10	CLERK	1300
20	CLERK	9900
30	CLERK	950
	CLERK	12150
20	ANALYST	6000
	ANALYST	6000
10	MANAGER	2450
20	MANAGER	2975
30	MANAGER	2850
	MANAGER	8275
30	SALESMAN	5600
	SALESMAN	5600
10	PRESIDENT	5000
	PRESIDENT	5000

```
select deptno,job,sum(sal)  
from emp  
group by grouping sets ((deptno,job))
```

DEPTNO	JOB	SUM(SAL)
10	MGR	6000
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	CLERK	9900
20	ANALYST	6000
20	MANAGER	2975
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

```
select deptno,job,sum(sal)  
from emp  
group by grouping sets (deptno,job)
```

DEPTNO	JOB	SUM(SAL)
10		19750
20		18875
30		15400
	ANALYST	6000
	CLERK	12150
	MANAGER	8275
	MGR	6000
	PRESIDENT	5000
	SALESMAN	5600
		11000

## Equivalence of CUBE and ROLLUP to GROUPING SETS

2005 - 2016

	GROUPING SETS
CUBE(a,b,c)	(a,b,c),(a,b),(a,c),(b,c),(a),(b),(c ),()
ROLLUP(a,b,c)	(a,b,c),(a,b),(a),()

- GROUP BY ( ) is typically a SELECT statement with NULL values for the column **a** and **b** and only the aggregate function.
- This is generally used for generating the grand totals.

e.g

```
select null,null,sum(sal)
from emp
group by ( );
```

# Composite Columns

- A composite column is a collection of columns that are treated as a unit.  
`ROLLUP (a, (b, c), d)`
- To specify composite columns, use the GROUP BY clause to group columns within parentheses so that the Oracle server treats them as a unit while computing ROLLUP or CUBE operations.
- When used with ROLLUP or CUBE, composite columns would mean skipping aggregation across certain levels.

```

select deptno,job,hiredate,sum(sal)
from emp
group by rollup((deptno,job),hiredate)

```

DEPTNO	JOB	HIREDATE	SUM(SAL)
10	CLERK	23-JAN-82	1300
10	CLERK		1300
10	MANAGER	09-JUN-81	2450
10	MANAGER		2450
10	PRESIDENT	17-NOV-81	5000
10	PRESIDENT		5000
20	CLERK		8000
20	CLERK	17-DEC-80	800
20	CLERK	23-MAY-87	1100
20	CLERK		9900
20	ANALYST	03-DEC-81	3000
20	ANALYST	19-APR-87	3000
20	ANALYST		6000
20	MANAGER	02-APR-81	2975
20	MANAGER		2975
30	CLERK	03-DEC-81	950
30	CLERK		950
30	MANAGER	01-MAY-81	2850
30	MANAGER		2850
30	SALESMAN	20-FEB-81	1600
30	SALESMAN	22-FEB-81	1250
30	SALESMAN	08-SEP-81	1500
30	SALESMAN	28-SEP-81	1250
30	SALESMAN		5600
			54025

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# Equivalence of CUBE and ROLLUP with composite columns to GROUPING SETS

	GROUPING SETS
CUBE(a,b,c)	(a,b,c),(a,b),(a,c),(b,c),(a),(b),(c ),( )
ROLLUP(a,b,c)	(a,b,c),(a,b),(a),( )
CUBE(a,(b,c))	(a,b,c),(a),(b,c),( )
ROLLUP(a,(b,c))	(a,b,c),(a),( )

- GROUP BY ( ) is typically a SELECT statement with NULL values for the column a and b and only the aggregate function.
- This is generally used for generating the grand totals.

e.g

```
select null,null,sum(sal)  
from emp  
group by ( );
```

# LESSON 18

## Advanced Subqueries

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved.

# What Is a Subquery?

A **subquery** is a **SELECT statement embedded in a clause of another SQL statement**.

Main query →

```
SELECT ...  
FROM   ...  
WHERE  ...
```

```
(SELECT ...  
  FROM   ...  
 WHERE  ....)
```

← Subquery

# Subqueries

```
SELECT select_list  
FROM   table  
WHERE  expr operator (SELECT select_list  
                      FROM   table);
```

- **The subquery (inner query) executes once before the main query.**
- **The result of the subquery is used by the main query (outer query).**

# Using a Subquery

```
SELECT last_name
FROM   employees
WHERE  salary > 10500
       (SELECT salary
        FROM   employees
        WHERE  employee_id = 149) ;
```

LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

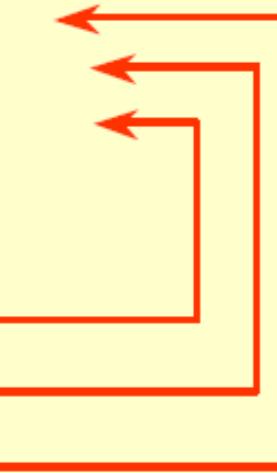
# Multiple-Column Subqueries

## Main query

WHERE (MANAGER\_ID, DEPARTMENT\_ID) IN

## Subquery

100	90
102	60
124	50



Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

## Multiple-Column Subquery:

Queries that return more than one columns from the inner select statement.

### Syntax:

```
SELECT column,column, ..... FROM table  
WHERE (column,column,...) IN  
(SELECT column,column,.... FROM table WHERE condition);
```

### Example:

Display the order number, product number and quantity of any item in which the product number and quantity match both the product number and quantity of an item in order 605 but the order number should not be 605.

```
SELECT ordid,prodid,qty  
FROM item  
WHERE (prodid, qty)  
IN  
(SELECT prodid,qty FROM item WHERE ordid = 605)  
AND  
ordid <> 605;
```

# Column Comparisons

**Column comparisons in a multiple-column subquery can be:**

- **Pairwise comparisons**
- **Nonpairwise comparisons**

# Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager *and* work in the same department as the employees with EMPLOYEE\_ID 178 or 174.

```
SELECT employee_id, manager_id, department_id  
FROM   employees  
WHERE  (manager_id, department_id) IN  
        (SELECT manager_id, department_id  
         FROM   employees  
         WHERE  employee_id IN (178,174))  
AND    employee_id NOT IN (178,174);
```

# Nonpairwise Comparison Subquery

**Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE\_ID 174 or 141 and work in the same department as the employees with EMPLOYEE\_ID 174 or 141.**

```
SELECT employee_id, manager_id, department_id
FROM employees
WHERE manager_id IN
      (SELECT manager_id
       FROM employees
       WHERE employee_id IN (174,141))
AND department_id IN
      (SELECT department_id
       FROM employees
       WHERE employee_id IN (174,141))
AND employee_id NOT IN(174,141);
```

# Using a Subquery in the FROM Clause

```
SELECT a.last_name, a.salary,  
       a.department_id, b.salavg  
FROM   employees a, (SELECT department_id,  
                      AVG(salary) salavg  
                     FROM employees  
                     GROUP BY department_id) b  
WHERE  a.department_id = b.department_id  
AND    a.salary > b.salavg;
```

LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
Hartstein	13000	20	9500
Mourgos	5800	50	3600
Hunold	9000	60	6400
Zlotkey	10500	80	10033.3333
Abel	11000	80	10033.3333
King	24000	90	19333.3333
Higgins	12000	110	10150

7 rows selected.

# Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries were supported in Oracle8*i* only in a limited set of cases, For example:
  - SELECT statement (FROM and WHERE clauses)
  - VALUES list of an INSERT statement
- In Oracle9*i*, scalar subqueries can be used in:
  - Condition and expression part of DECODE and CASE
  - All clauses of SELECT except GROUP BY

SELECT

(select max(sal) from emp)

ename

(select avg(sal) from emp)

dname

FROM

emp,

(select dname from dept where deptno = 10);

highest\_salary,  
employee\_name,  
avg\_salary,

HIGHEST_SALARY	EMPLOYEE_N	AVG_SALARY	DNAME
8000	SMITH	3001.38889	ACCOUNTING
8000	ALLEN	3001.38889	ACCOUNTING
8000	WARD	3001.38889	ACCOUNTING
8000	JONES	3001.38889	ACCOUNTING
8000	MARTIN	3001.38889	ACCOUNTING
8000	BLAKE	3001.38889	ACCOUNTING
8000	CLARK	3001.38889	ACCOUNTING
8000	SCOTT	3001.38889	ACCOUNTING
8000	KING	3001.38889	ACCOUNTING
8000	TURNER	3001.38889	ACCOUNTING
8000	ADAMS	3001.38889	ACCOUNTING

## **Example-1**

```
insert into max_credit  
(name, credit)  
Values  
(‘Bill’, select max(credit) from credit_table where name = ‘BILL’);
```

## **Example-2**

```
insert into emp_salary_summary  
( sum_salaries , max_salary, min_salary, avg_salary)  
Values  
(  
    (select sum(salary) from emp),  
    (select max(salary) from emp),  
    (select min(salary) from emp),  
    (select avg(salary) from emp)  
);
```

### Example-3

Display all empno,ename,manager no and manager names of every employee.

```
SELECT e.empno,e.ename,e.mgr mgr_no,  
(select ename mgr_name from emp where empno = e.mgr)  
from emp e;
```

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# Scalar Subqueries: Examples

## Scalar Subqueries in CASE Expressions

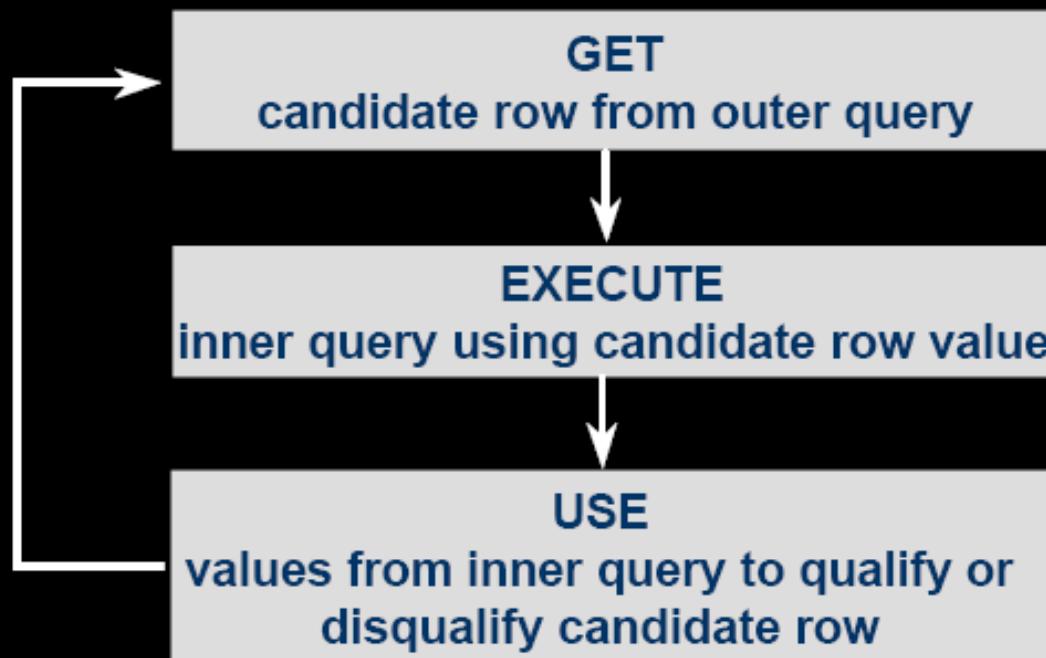
```
SELECT employee_id, last_name,  
       (CASE  
           WHEN department_id = 20 (SELECT department_id FROM departments  
                WHERE location_id = 1800)  
               THEN 'Canada' ELSE 'USA' END) location  
FROM   employees;
```

## Scalar Subqueries in ORDER BY Clause

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

# Correlated Subqueries

**Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.**



## Correlated Subqueries:

- The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement.

### Example:

```
SELECT ename,sal,deptno  
FROM emp e  
WHERE sal > (    SELECT avg(sal)  
                  FROM emp  
                  WHERE deptno = e.deptno)
```

referenced column of parent table

- Parent statement can be a **SELECT,UPDATE or DELETE** statements.

Copyright  
Tanveer Khan, 2005 - 2016. All Rights Reserved

# NORMAL SUBQUERIES VESUS CORRELATED SUBQUERIES:

- With a *normal subquery*, the inner SELECT query runs first and execute once, returning values to be used by the main query.
- With a *correlated subquery*, the inner SELECT statement executes once for each candidate row considered by the outer query.

## Execution of NORMAL (NESTED) SUBQUERIES

- The inner query executes first and finds values.
- The outer query executes once, using the value from the inner query.

## Execution of CORRELATED SUBQUERIES

- Get a candidate row (fetched by the outer query)
- Execute the inner query using the value of the candidate row
- Repeat until no candidate row remains.

# Correlated Subqueries

```
SELECT column1, column2, ...
FROM   table1 [outer]
WHERE  column1 operator
          (SELECT column1, column2
           FROM   table2
           WHERE  expr1 =
                  [outer.expr2]);
```

The subquery references a column from a table in the parent query.

# Using Correlated Subqueries

Display details of those employees who have switched jobs at least twice.

```
SELECT e.employee_id, last_name, e.job_id  
FROM   employees e  
WHERE  2 <= (SELECT COUNT(*)  
              FROM   job_history  
              WHERE  employee_id = e.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID
101	Kochhar	AD_VP
176	Taylor	SA_REP
200	Whalen	AD_ASST

## Correlated Update:

- Correlated update can be used to update rows in one table based on rows from another table.

### Example:

```
UPDATE employee  
SET dname = (SELECT dname  
              FROM dept d  
              WHERE e.deptno = d.deptno);
```

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

## Correlated Delete:

- Correlated delete can be used to delete rows in one table based on rows from another table.

### Example:

```
DELETE FROM product_master pm  
WHERE product_no = (SELECT product_no  
                      FROM sales_master sm  
                      WHERE pm.product_no = sm.product_no)
```

Copyright  
Tanveer Khan, 2005 - 2016. All Rights Reserved

## **EXISTS Operator:**

- EXISTS operator is used to test whether a value retrieved by the outer query exists in the result set of the values retrieved by the inner query.
- If the subquery returns atleast one row, the operator returns TRUE.
- If the value does not exists, it returns FALSE.

## **NOT EXISTS Operator:**

- NOT EXISTS works just opposite to EXISTS.

Copyright

Tanveer Khan, 2005 - 2016. All Rights Reserved

# Using the EXISTS Operator

**Find employees who have at least one person reporting to them.**

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                  FROM   employees
                  WHERE  manager_id =
                         outer.employee_id) ;
```

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
124	Mourgos	ST_MAN	50
149	Zlotkey	SA_MAN	80
201	Hartstein	MK_MAN	20
205	Higgins	AC_MGR	110

8 rows selected.

- EXISTS operator ensures that the search in the inner query does not continue when atleast one match is found for the MANAGER and EMPLOYEE NO. by the condition

*WHERE manager\_id = outer.employee\_id*

- Since the inner SELECT query does not need to return any specific value, so a constant can be selected.
- From a performance point of view, it is faster to select a constant than a column.

# Using the NOT EXISTS Operator

Find all departments that do not have any employees.

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                   FROM employees
                   WHERE department_id
                         = d.department_id);
```

DEPARTMENT_ID	DEPARTMENT_NAME
190	Contracting

## WITH Clause:

- Using the WITH clause, you can use the same query block in a SELECT statement when it occurs more than once within a complex query.
- The WITH clause retrieves the results of a query block and stores it in the temporary space.
- WITH clause improves performance.
- Makes the query easy to read.
- It is used only with SELECT statement.

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

**WITH**

**dept\_total as**

**(SELECT deptno,sum(sal) dept\_sum**

**FROM emp**

**GROUP BY deptno) ,**

**avg\_sal as**

**(SELECT avg(sal) dept\_avg**

**FROM emp**

**)**

**SELECT \***

**FROM dept\_total**

**WHERE dept\_sum > (select dept\_avg from avg\_sal);**

# *LESSON*

# **Producing Readable Output with iSQL\*Plus**

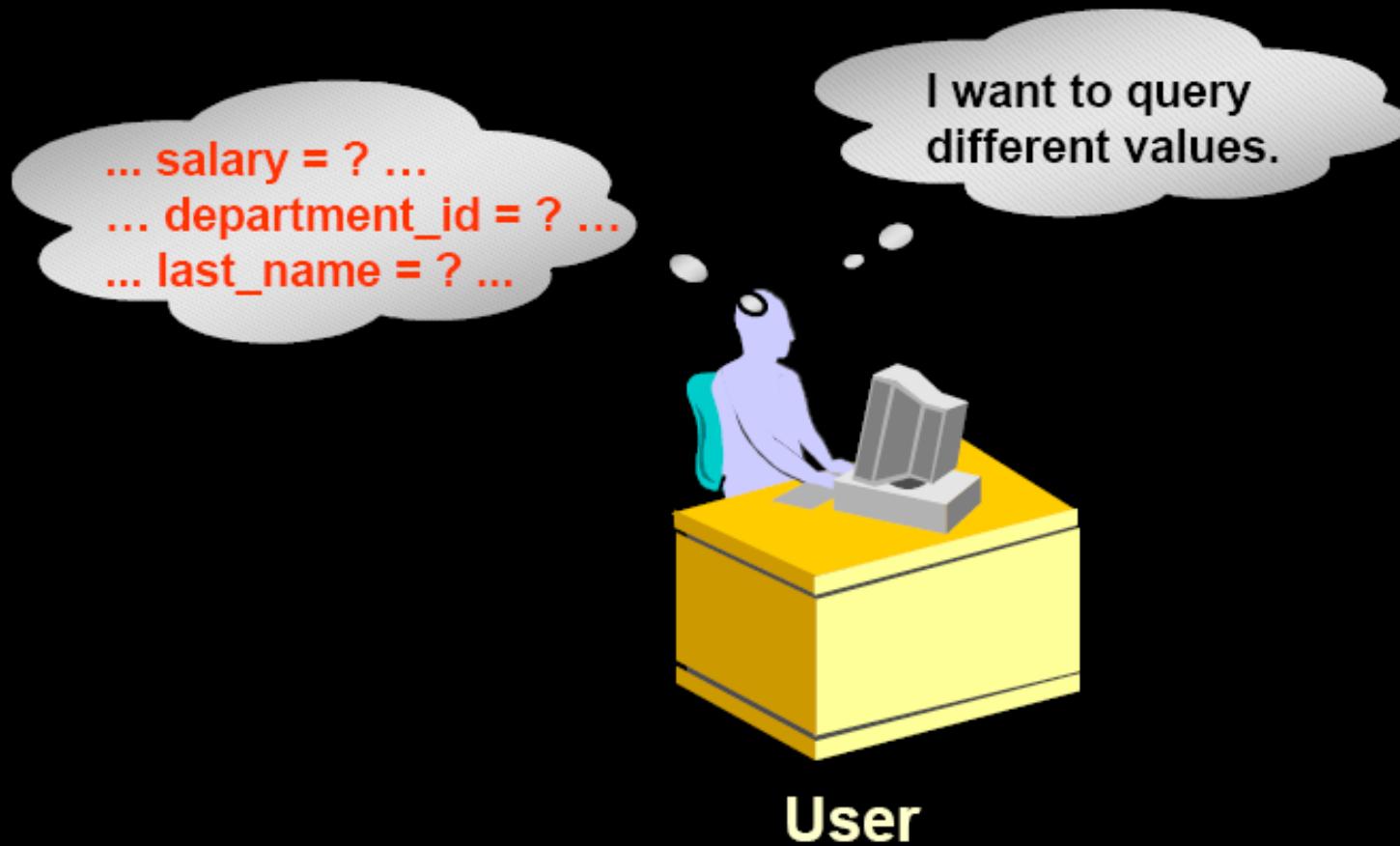
Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved.

# Objectives

**After completing this lesson, you should be able to do the following:**

- **Produce queries that require a substitution variable**
- **Customize the *i*SQL\*Plus environment**
- **Produce more readable output**
- **Create and execute script files**

# Substitution Variables



## Substitution Variables:

- The examples so far have been hard-coded, the user would trigger the report, and the report would run without further prompting.
- Using iSQLPlus, you can create reports that prompt the user to supply their own values to restrict the range of data returned by using substitution variables.
- You can embed substitution variables in a command file or in a single SQL statement.
- A substitution variable can be thought of as a container in which the values are temporarily stored. When the statement is run, the value is substituted.

# Substitution Variables

**Use iSQL\*Plus substitution variables to:**

- **Temporarily store values**
  - Single ampersand (&)
  - Double ampersand (&&)
  - DEFINE command
- **Pass variable values between SQL statements**
- **Dynamically alter headers and footers**

## 1) Single Ampersand(&)

To prompt the user every time at the execution of command for a value.

### Example 1:

```
Select empno,ename,sal  
from emp  
where empno = &employee_number;
```

### OUTPUT:

Enter value for employee\_number:

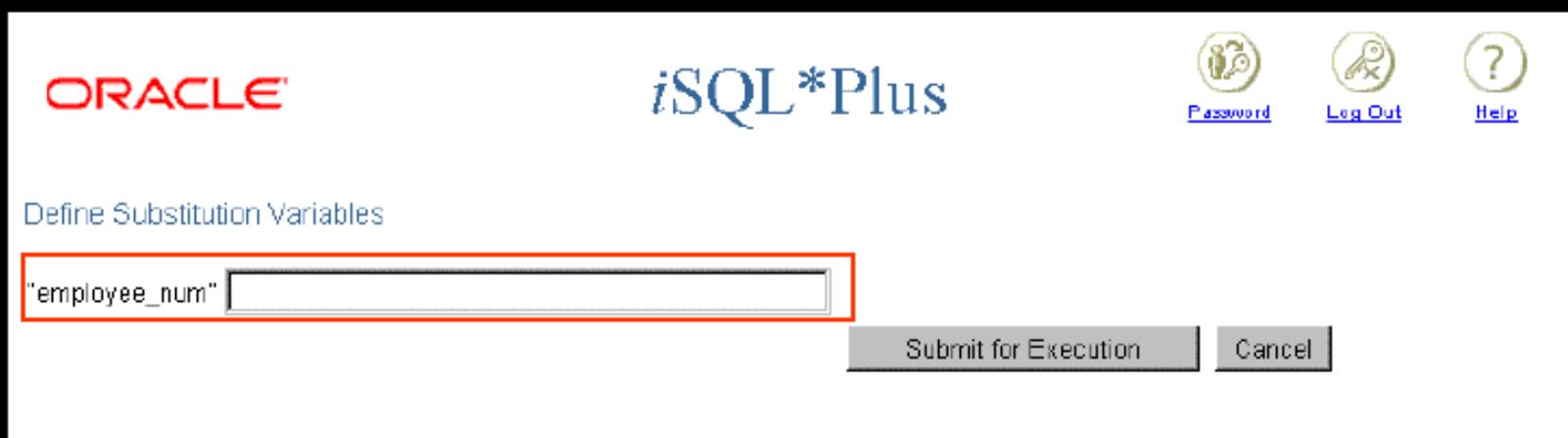
### Example 2:

```
Select empno,ename,sal  
from emp  
where job = upper('&emp_job');
```

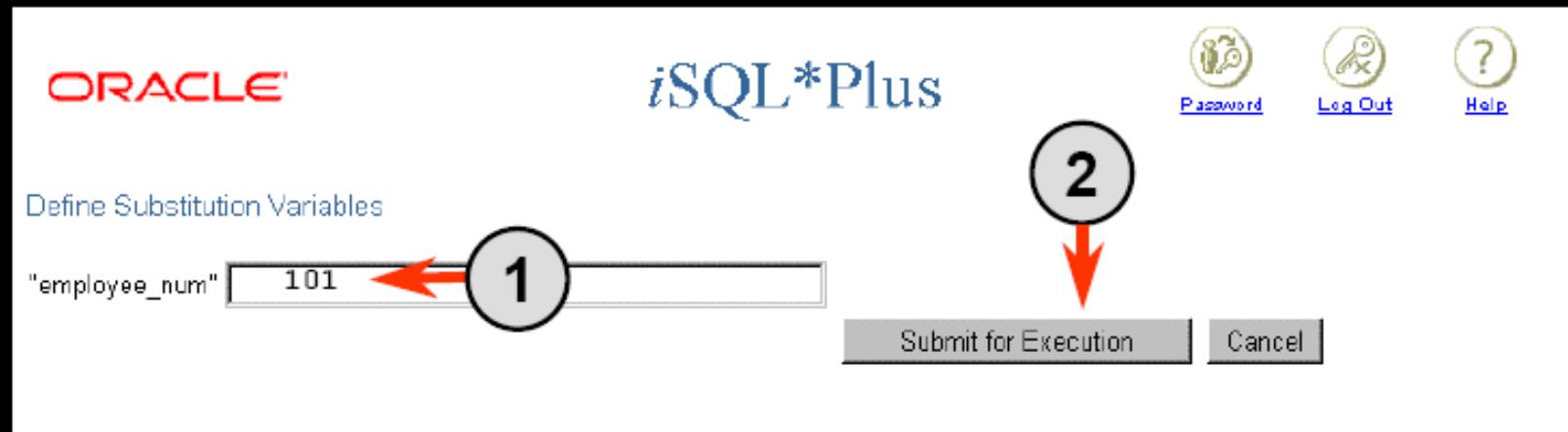
# Using the & Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value.

```
SELECT      employee_id, last_name, salary, department_id  
FROM        employees  
WHERE       employee_id = &employee_num ;
```



# Using the & Substitution Variable



EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
101	Kochhar	17000	90

# Character and Date Values with Substitution Variables

Use single quotation marks for date and character values.

```
SELECT last_name, department_id, salary*12  
FROM employees  
WHERE job_id = '&job_title' ;
```

Define Substitution Variables

'job\_title'

LAST_NAME	DEPARTMENT_ID	SALARY*12
Hunold	60	108000
Ernst	60	72000
Lorentz	60	50400

Substitution variables are used to supplement the following

- WHERE condition
- ORDER BY clause
- Column expression
- Table name

**Example:**

```
SELECT empno, &column  
FROM &table  
WHERE &condition  
ORDER BY &order_by_column;
```

## 2) Double Ampersand (&&)

Use (&&), if you want to reuse the variable value without prompting the user each time.

### Example:

Select empno, ename, &&job From emp Order By &job;

## 3) DEFINE

It creates a CHAR datatype user variable. Use UNDEFINE command to delete it.

### SYNTAX:

DEFINE *variable* = *value*

### Example:

DEFINE column = job;

## Example:

Define col1 = empno;

Define col2=ename;

Define table=emp;

Define condition=1000;

SELECT &col1,&col2

FROM &table

WHERE &sal > &condition;

# Defining Substitution Variables

- You can **predefine variables using the iSQL\*Plus DEFINE command.**  
`DEFINE variable = value` creates a user variable with the CHAR data type.
- If you need to **predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the DEFINE command.**
- A **defined variable is available for the session**

# DEFINE and UNDEFINE Commands

- A variable remains defined until you either:
  - Use the UNDEFINE command to clear it
  - Exit iSQL\*Plus
- You can verify your changes with the DEFINE command.

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE          = "IT_PROG" (CHAR)
```

```
UNDEFINE job_title
DEFINE job_title
SP2-0135: symbol job_title is UNDEFINED
```

# Using the VERIFY Command

**Use the VERIFY command to toggle the display of the substitution variable, before and after iSQL\*Plus replaces substitution variables with values.**

```
SET VERIFY ON
SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num;
```

"employee\_num"

```
old    3: WHERE employee_id = &employee_num
new    3: WHERE employee_id = 200
```

# Customizing the iSQL\*Plus Environment

- **Use SET commands to control current session.**

```
SET system_variable value
```

- **Verify what you have set by using the SHOW command.**

```
SET ECHO ON
```

```
SHOW ECHO  
echo ON
```

<b>SET Commands</b>	<b>Description</b>
COLSEP {_ text}	Sets text to be printed between columns (the default is single column)
FEEDBACK { <u>6</u>  n OFF ON}	Displays the numbers of records returned by a query when the query selects at least <u>n</u> records.
HEADING {OFF ON}	Determines whether column headings are displayed in reports.
LINESIZE {80 n}	Sets the numbers of characters per line to <u>n</u> for reports.
LONG {80 n}	Sets the max. width for displaying LONG values.
PAGESIZE {24 n}	Specifies the numbers of lines per page of output.
PAUSE {OFF ON text}	Allows you to control scrolling of your terminal (you must press [return] after seeing each pause.)
BTITLE {TEXT OFF ON}	Specifies a footer to appear at the bottom of each page of the report
TTITLE {TEXT OFF ON}	Specifies a header to appear at the top of each page of the report

## Note:

The value ***n*** represents a numeric value. The underlined values indicate default values. If you enter no value, SQL\*PLUS assumed the default value.

In order to display the records properly, the most common commands are:

1. Set linesize 120 (from 80)
2. Set pause on (from off)

## Guidelines:

- All format commands remain in effect until the end of iSQL\*Plus session or until the format setting is overwritten or cleared.
- Remember to reset your iSQL\*Plus settings to the default values after every report.
- There is no specific command to reset the environment setting; you must know the specific value or log out and log in again.

To verify, what you have SET, use the SHOW command.

### Example:

- Show linesize
- Show pause
- Show All

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved

# SQL \* PLUS Format Commands

## 1) COLUMN [col | alias] [options]

Example:

COLUMN ename HEADING 'Employee | Name' FORMAT A15

COLUMN sal JUSTIFY LEFT FORMAT \$99,990.00

COLUMN mgr FORMAT 9999 NULL 'NO MANAGER'

OPTIONS	DESCRIPTION
CLEAR	Clear any column format.
FORMAT <i>format</i>	Changes the display of the column data.
HEADING <i>text</i>	Sets the column heading.
JUSTIFY <i>align}</i>	Justifies the column heading.
NOPRINT	Hides the column.
NULL <i>text</i>	Specifies text to be displayed for null values.

# Using the COLUMN Command

- **Create column headings.**

```
COLUMN last_name HEADING 'Employee|Name'  
COLUMN salary JUSTIFY LEFT FORMAT $99,990.00  
COLUMN manager FORMAT 999999999 NULL 'No manager'
```

- **Display the current setting for the LAST\_NAME column.**

```
COLUMN last_name
```

- **Clear settings for the LAST\_NAME column.**

```
COLUMN last_name CLEAR
```

## CREATING A SCRIPT FILE TO RUN A REPORT:

```
SET PAGESIZE 37                                //37 records per screen
SET LINESIZE 60                                 // 60 characters per line
SET FEEDBACK OFF                               // command echo off
TTITLE 'EMPLOYEE | REPORT'                     // Header of Report
BTITLE 'CONFIDENTIAL'                          // Footer of Report
BREAK ON job                                    // suppress duplicates
COLUMN job HEADING 'JOB | CATEGORY' -          // heading of attribute JOB
FORMAT A15                                      // format for JOB for 15 characters
COLUMN ename HEADING 'EMPLOYEE | NAME' -        // Remarks
FORMAT A15
COLUMN sal HEADING 'SALARY' -                  // Remarks
FORMAT $99,9999
REM *** select statement ***
SELECT job,ename,sal FROM emp ORDER BY job
/
SET FEEDBACK ON
```

Copyright Tanveer Khan, 2005 - 2016. All rights Reserved

# Sample Report

Fri Sep 28

Employee  
Report

page 1

Job Category	Employee	Salary
AC_ACCOUNT	Gietz	\$8,300.00
AC_MGR	Higgins	\$12,000.00
AD_ASST	Whalen	\$4,400.00
IT_PROG	Einst	\$6,000.00
	Hunold	\$9,000.00
	Lorentz	\$4,200.00
MK_MAN	Hartstein	\$13,000.00
MK_REP	Fay	\$6,000.00
SA_MAN	Zlotkey	\$10,500.00
SA_REP	Abel	\$11,000.00
	Grant	\$7,000.00
	Taylor	\$8,600.00

Confidential

...

# **LESSON**

# **Tables Constraints**

Copyright Tanveer Khan, 2005 - 2016. All Rights Reserved.

## CONSTRAINTS:

- Constraints enforces rules at the table level.
- Constraints prevents the deletion of a table if there are dependencies.
- Constraints can be **added,dropped,enabled** or **disabled**.

## CONSTRAINTS TYPES:

- 1) NOT NULL.
- 2) UNIQUE.
- 3) PRIMARY KEY.
- 4) FOREIGN KEY.
- 5) CHECK.

## Constraints Guidelines:

- Name a constraint or the Oracle server generates a name by using the **SYS\_Cn** format where **n** is an integer so that the constraint name is unique.
- Create a constraint either:
  - At the same time as the table is created, or
  - After the table has been created
- Define a constraint at the column or table level.
- You can view the constraints defined for a specific table by looking at the **USER\_CONSTRAINTS** data dictionary table.

# Defining Constraints

```
CREATE TABLE [schema.]table  
  (column datatype [DEFAULT expr]  
   [column_constraint],  
   ...  
   [table_constraint] [, . . .]);
```

```
CREATE TABLE employees(  
    employee_id  NUMBER(6),  
    first_name   VARCHAR2(20),  
    ...  
    job_id       VARCHAR2(10) NOT NULL,  
    CONSTRAINT emp_emp_id_pk  
                PRIMARY KEY (EMPLOYEE_ID));
```

## Levels of Constraints:

### 1) COLUMN LEVEL constraints:

- References a single column.
- Defined with the definition of column.

#### Example:

```
CREATE TABLE employee  
(empno number(5) PRIMARY KEY,  
ename varchar2(25));
```

### 2) TABLELEVEL constraints:

- References one or more columns
- Defined separately from the definition of the column in the table.
- Can define any constraint except **NOT NULL**.

#### Example:

```
CREATE TABLE employee  
(empno number(5),  
ename varchar2(25),  
PRIMARY KEY(empno));
```

# The NOT NULL Constraint

**Ensures that null values are not permitted for the column:**

EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

20 rows selected.

  
**NOT NULL constraint**  
**(No row can contain a null value for this column.)**

  
**NOT NULL constraint**

  
**Absence of NOT NULL constraint**  
**(Any row can contain null for this column.)**

# The NOT NULL Constraint

Is defined at the column level:

```
CREATE TABLE employees (
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,           ← System named
    salary            NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date         DATE
        CONSTRAINT emp_hire_date_nn
        NOT NULL,
    ...
)
```

System  
named

User  
named

## 1) NOT NULL Constraint:

- Defined at the column level, not at the table level.

### Example:

```
CREATE TABLE employee  
(empno number(4),  
ename varchar2(25) NOT NULL)
```

## 2) UNIQUE Key Constraint:

- Defined at either the table level or the column level.
- If the UNIQUE constraint comprises more than one column, that group of columns is called the *Composite Unique Key*.
- Null values can be inserted.

### Example:

1) Create table employee

```
(ename varchar2(25) NOT NULL UNIQUE);
```

2) Create table employee

```
(empno number(3),
```

```
ename varchar2(25),
```

```
CONSTRAINT unique_cons UNIQUE (ename,empno))
```

### 3) PRIMARY KEY constraint:

- Defined at either the table or the column level.
- Composite key is created by using the table level definition.
- A table can have only one PRIMARY KEY but can have several UNIQUE constraints.
- PRIMARY KEY cannot be null.

#### Example:

- 1) CREATE TABLE employee  
(empno number(5),  
ename varchar2(25),  
CONSTRAINT emp\_cons PRIMARY KEY(empno));
- 2) CREATE TABLE employee  
(empno number(5) Primary Key,  
ename varchar2(25));

#### 4) FOREIGN KEY constraint:

- Defined at either the **table** or the **column** level.
- Composite Foreign Key is created by using the table level definition.
- **Foreign key:** Defines the column in the child table at the table level constraint.
- **References:** Identifies the table and column in the parent table.
- **On delete cascade:** allows deletion in the parent table and deletion on the dependent rows in the child table.
- **On delete set null:** converts dependent foreign key values to NULL when the parent value is removed.
- Without the **ON DELETE CASCADE** or the **ON DELETE SET NULL** options, the row in the parent table cannot be deleted if it is referenced in the child table.

## Example:

### AT TABLE LEVEL

```
CREATE TABLE employee  
(empno number(5) PRIMARY KEY,  
ename varchar2(25),  
deptno number(5),  
CONSTRAINT emp_cons FOREIGN KEY(deptno) REFERENCES  
dept (deptno) ON DELETE CASCADE);
```

### AT COLUMN LEVEL

```
CREATE TABLE employee  
(empno number(5) PRIMARY KEY,  
ename varchar2(25),  
deptno number(5) references (dept));
```

## 5) CHECK constraint:

- Defined at either the table level or the column level.
- Defines a condition that each row must satisfy.

### Example:

```
CREATE TABLE employee  
(empno number(5),  
ename varchar2(25),  
CONSTRAINT empdept_cons CHECK(empno BETWEEN 1000 AND 5000));
```

# Adding a Constraint Syntax

**Use the ALTER TABLE statement to:**

- **Add or drop a constraint, but not modify its structure**
- **Enable or disable constraints**
- **Add a NOT NULL constraint by using the MODIFY clause**

```
ALTER TABLE table
ADD [CONSTRAINT constraint] type (column);
```

- You can define a NOT NULL column only if the table is empty or if the column has the value for each row.

## ADDING a constraint:

Example:

```
ALTER TABLE employee  
ADD CONSTRAINT  
emp_cons PRIMARY KEY(empno);
```

## DROPPING a constraint:

Example:

```
ALTER TABLE employee  
DROP CONSTRAINT emp_cons;
```

## DISABLING a constraint:

Example:

```
ALTER TABLE employee  
DISABLE CONSTRAINT emp_cons;
```

## ENABLING a constraint:

Example:

```
ALTER TABLE employee  
ENABLE CONSTRAINT emp_cons;
```

# Adding a Constraint

**Add a FOREIGN KEY constraint to the EMPLOYEES table indicating that a manager must already exist as a valid employee in the EMPLOYEES table.**

```
ALTER TABLE      employees
ADD CONSTRAINT  emp_manager_fk
    FOREIGN KEY(manager_id)
    REFERENCES employees(employee_id) ;
```

**Table altered.**

## Dropping a constraint:

- To drop a constraints, you can identify the constraint name from the ***USER\_CONSTRAINTS*** and ***USER\_CONS\_COLUMNS*** data dictionary views.

### Syntax:

```
ALTER TABLE table_name  
DROP PRIMARY KEY | UNIQUE (col_name) |  
CONSTRAINT constraints [CASCADE];
```

Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved

# Dropping a Constraint

- Remove the manager constraint from the EMPLOYEES table.

```
ALTER TABLE      employees
DROP CONSTRAINT emp_manager_fk;
Table altered.
```

- Remove the PRIMARY KEY constraint on the DEPARTMENTS table and drop the associated FOREIGN KEY constraint on the EMPLOYEES . DEPARTMENT\_ID column.

```
ALTER TABLE    departments
DROP PRIMARY KEY CASCADE;
Table altered.
```

# Disabling Constraints

- Execute the **DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint.**
- **Apply the CASCADE option to disable dependent integrity constraints.**

```
ALTER TABLE          employees
DISABLE CONSTRAINT  emp_emp_id_pk CASCADE;
Table altered.
```

# Enabling Constraints

- Activate an integrity constraint currently disabled in the table definition by using the ENABLE clause.

```
ALTER TABLE employees  
ENABLE CONSTRAINT emp_emp_id_pk;  
Table altered.
```

- A UNIQUE or PRIMARY KEY index is automatically created if you enable a UNIQUE key or PRIMARY KEY constraint.

# Cascading Constraints

- The **CASCADE CONSTRAINTS** clause is used along with the **DROP COLUMN** clause.
- The **CASCADE CONSTRAINTS** clause drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns.
- The **CASCADE CONSTRAINTS** clause also drops all multicolumn constraints defined on the dropped columns.

## **Example-1:**

Step-1: Create a dummy table DEPT2 table.

```
Create table dept2( deptno number constraint dept2_pk primary key,  
dname varchar2(10));
```

Step-2: Create a dummy table EMP2 table.

```
Create table emp2( empno number constraint emp2_pk primary key,  
ename varchar2(25),  
deptno number constraint emp2_fk references dept2(deptno) on delete set  
null);
```

Step-3: Now insert the values in both the tables.

```
Insert into dept2 values(10, 'MKT');
```

```
Insert into emp2 values(101, 'A', 10);
```

Step-4: Now try to drop the DEPTNO column from the DEPT table.

```
Alter table dept2  
drop column deptno;
```

**You will get the error.**

Step-5: Now try to drop the DEPTNO column from the DEPT table with the following command.

```
Alter table dept2  
drop column deptno  
cascade constraints;
```

The above command will drop the column by deleting the constraint applied on both EMP2 and DEPT2.

## Example:

```
create table test1
(
    pk number PRIMARY KEY,
    fk number,
    col1 number,
    col2 number,
    CONSTRAINT fk_constraint FOREIGN KEY (fk) REFERENCES test1,
    CONSTRAINT ck1 CHECK(pk > 0 and col1 > 0),
    CONSTRAINT ck2 CHECK(col2 > 0)
);
```

*Copyright © Tanveer Khan, 2005 - 2016. All Rights Reserved*

Now executing the following commands will generate error

```
ALTER TABLE test1 DROP (pk);      -- pk is a parent key
ALTER TABLE test1 DROP (col1);    -- col1 is referenced by multicolumn constraint ck1
```

## To remove the errors:

- Submitting the following statement drops column PK, the primary key constraint, the fk\_constraint foreign key constraint, and the check constraints CK1.

```
ALTER TABLE test1 DROP (pk) CASCADE CONSTRAINTS;
```

- If all the columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required.

```
ALTER TABLE test1 DROP (pk,fk,col1);
```

## Viewing Constraints:

- The only constraint you can verify thorough the DESCRIBE command is NOT NULL.
- To view all the constraints on your table , query the USER\_CONSTRAINTS table.

### Example:

```
SELECT constraint_name , constraint_type, search_condition  
FROM USER_CONSTRAINTS  
WHERE table_name = 'EMP';
```

CONSTRAINT_TYPE	SYMBOL
CHECK	C
PRIMARY KEY	P
FOREIGN KEY	R
UNIQUE	U
NOT NULL	C

```

SELECT      constraint_name, constraint_type,
            search_condition
FROM        user_constraints
WHERE       table_name = 'EMPLOYEES';

```

CONSTRAINT_NAME	C	SEARCH_CONDITION
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL
EMP_SALARY_MIN	C	salary > 0
EMP_EMAIL_UK	U	

# Viewing the Columns Associated with Constraints

**View the columns associated with the constraint names in the USER\_CONS\_COLUMNS view.**

```
SELECT      constraint_name, column_name
FROM        user_cons_columns
WHERE       table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID
...	