

# *Introduction to Database Programming*

*By*  
*Iraj Sabzevary*



## Table of Contents

Chapter 1(Getting started).....	1
1.1    Database .....	2
1.2    Relational model .....	2
1.3    DBMS .....	3
1.4    RDBMS.....	4
1.5    DBAvs. DA.....	5
1.6    Some database vendors .....	5
1.7    SQL.....	5
1.8    SQL*Plus .....	6
1.9    SQL vs. SQL*Plus .....	7
1.10   Some SQL*Plus commands.....	7
1.11   Additional SQL*Plus commands .....	11
1.12   PL/SQL .....	13
✓      CHECK 1A .....	13
SummaryExamples.....	14
Chapter 2 (Database Design) .....	16
2.1    Poor Table Design.....	17
2.2    Normalization.....	17
2.3    Normal Forms .....	17
2.4    Primary key, Foreign key, Candidate or Alternate key .....	18
2.5    First Normal Form (1NF).....	19
2.6    Second Normal Form (2NF) .....	20
2.7    Third Normal Form (3NF) .....	20
2.8    Entity Relationship Diagrams (ER).....	20
2.9    One to One .....	21
2.10   One to Many.....	22
2.11   Many to Many .....	23
2.12   Normalization example .....	24
2.13   Normalize the following .....	25
✓      CHECK 2A .....	26
Chapter 3 (DDL commands).....	27

3.1	Creating and dropping tables.....	28
✓	<i>CHECK 3A</i> .....	35
3.2	Adding columns using ALTER command.....	35
3.3	Modifying using ALTER command .....	37
3.4	Dropping columns using ALTER command.....	39
3.5	Renaming column using ALTER command .....	41
✓	<i>CHECK 3B</i> .....	41
3.6	Constraints .....	41
3.7	Primary key constraint .....	43
✓	<i>CHECK 3C</i> .....	47
3.8	Unique constraint .....	48
3.9	Check constraint.....	52
3.10	Not NULL constraint .....	55
✓	<i>CHECK 3D</i> .....	57
3.11	Foreign key constraint.....	58
✓	<i>CHECK 3E</i> .....	67
3.12	Disabling/Enabling/Dropping constraints .....	68
✓	<i>CHECK 3F</i> .....	71
3.13	Indexes .....	71
✓	<i>CHECK 3G</i> .....	73
	<i>Summary example</i> .....	74
	<b>Chapter 4 (Data Manipulation (inserts))</b> .....	76
4.1	Inserting text .....	78
4.2	Inserting numbers.....	81
4.3	Inserting Dates .....	87
✓	<i>CHECK 4A</i> .....	91
4.4	Sequences.....	92
✓	<i>CHECK 4B</i> .....	96
	<i>Summary Examples</i> .....	97
	<b>Chapter 5 (SELECT Statements)</b> .....	98
5.1	What is a SELECT statement.....	99
5.2	What is a function .....	100
5.3	Simple Select clause.....	100

5.4	Alias .....	101
5.5	Concatenation.....	101
5.6	LTRIM/RTRIM/TRIM .....	103
5.7	DISTINCT or UNIQUE.....	104
✓	<i>CHECK 5A</i> .....	105
5.8	DUAL table.....	106
5.9	INITCAP.....	107
5.10	SUBSTR, INSTR and REPLACE.....	108
✓	<i>CHECK 5B</i> .....	112
5.11	LPAD, RPAD.....	113
5.12	Arithmetic .....	115
5.13	GREATEST, LEAST.....	117
5.14	Date functions .....	118
✓	<i>CHECK 5C</i> .....	121
5.15	NVL and NVL2 Functions.....	121
5.16	DECODE .....	123
5.17	SIGN .....	124
5.18	CASE .....	125
5.19	TO_NUMBER .....	125
✓	<i>CHECK 5D</i> .....	126
	<i>Summary Examples</i> .....	127
	Chapter 6 (Restricting) .....	128
6.1	Numeric versus Text .....	129
6.2	Dates.....	131
6.3	LOWER, UPPER .....	132
6.4	AND, BETWEEN .....	132
6.5	OR, IN.....	134
6.6	ANY, ALL operator.....	136
6.7	Like Clause .....	137
6.8	NULL.....	138
✓	<i>CHECK 6A</i> .....	140
6.9	Creating tables with select statements.....	140
6.10	Updating tables using the update statement .....	142

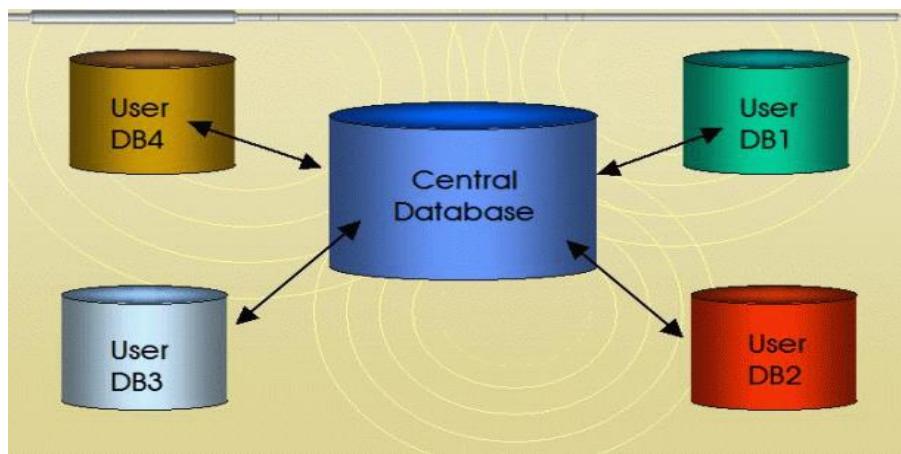
✓ <i>CHECK 6B</i> .....	143
<i>Summary examples</i> .....	144
<b>Chapter 7 (SORTING)</b> .....	<b>145</b>
✓ <i>CHECK 7A</i> .....	152
<i>Summary Examples</i> .....	152
<b>Chapter 8 (GROUP BY)</b> .....	<b>153</b>
8.1    Group by examples .....	154
8.2    SUM.....	156
8.3    DISTINCT .....	159
8.4    AVG.....	160
8.5    COUNT.....	162
8.6    MAX .....	164
8.7    MIN.....	166
8.8    Dates and group functions.....	167
✓ <i>CHECK 8A</i> .....	168
<i>Summary Examples</i> .....	168
<b>Chapter 9(SUBQUERIES)</b> .....	<b>169</b>
✓ <i>CHECK 9A</i> .....	181
<i>Summary Examples</i> .....	181
<b>Chapter 10 (Joins)</b> .....	<b>182</b>
10.1    Cartesian/Cross Join.....	186
10.2    Inner Join.....	190
✓ <i>CHECK 10A</i> .....	200
10.3    Self Join.....	200
10.4    Outer Join.....	202
✓ <i>CHECK 10B</i> .....	214
10.5    Set Operators.....	215
✓ <i>CHECK 10C</i> .....	223
<i>Summary Examples</i> .....	224
<b>Chapter 11 (Views)</b> .....	<b>226</b>
✓ <i>CHECK 11A</i> .....	238
<i>Summary Examples</i> .....	239
<b>Chapter 12 (System Tables)</b> .....	<b>240</b>

✓ <i>CHECK 12A</i> .....	248
<i>Summary Examples</i> .....	249
<b>Chapter 13 (PL/SQL).....</b>	<b>250</b>
13.1    What is a PL/SQL .....	251
13.2    Advantages of PL/SQL .....	251
13.3    Anatomy of PL/SQL .....	253
13.4    What is a PL/SQL block made up of.....	253
13.5    Variable declaration .....	255
13.6    Scope of variables .....	259
13.7    Constants.....	260
✓ <i>CHECK 13A</i> .....	261
13.8    PL/SQL records.....	261
13.9    Conditional Statements .....	264
✓ <i>CHECK 13B</i> .....	268
13.10    Loops .....	269
✓ <i>CHECK 13C</i> .....	279
13.11    Cursors .....	279
✓ <i>CHECK 13D</i> .....	285
13.12    Stored procedures.....	286
13.13    Functions.....	291
13.14    Parameters in functions and procedures.....	299
✓ <i>CHECK 13E</i> .....	304
13.15    Exceptions.....	304
<b>Chapter 14 (Triggers) .....</b>	<b>311</b>
✓ <i>CHECK 14A</i> .....	317
✓ <i>CHECK 14B</i> .....	322
<b>Appendix A (Commit, Rollback,Savepoint)</b> .....	<b>323</b>
<b>Appendix B (Substitution)</b> .....	<b>327</b>
<b>Appendix C (Import/Export)</b> .....	<b>330</b>
<b>Appendix D (SQL-LOADER)</b> .....	<b>331</b>



*Lawrence Joseph "Larry" Ellison (born August 17, 1944) is an American business magnate, co-founder and chief executive officer of Oracle Corporation, a major enterprise software company. As of 2011 he is the fifth richest person in the world, with a personal wealth of \$39.5 billion.*

## Chapter 1 (Getting started)



*"A compromise is an agreement whereby both parties get what neither of them wanted"*

# Objectives for this book

- Understand relational database fundamentals
- Create databases
- Understand the normalization process
- Recognize poor table design
- Understand the principles of adding, deleting, updating, and sorting records within a table
- Create queries
- Import/Export data

## 1.1 Database

A database consists of an organized collection of data. There are various models that are used to organize data such as:

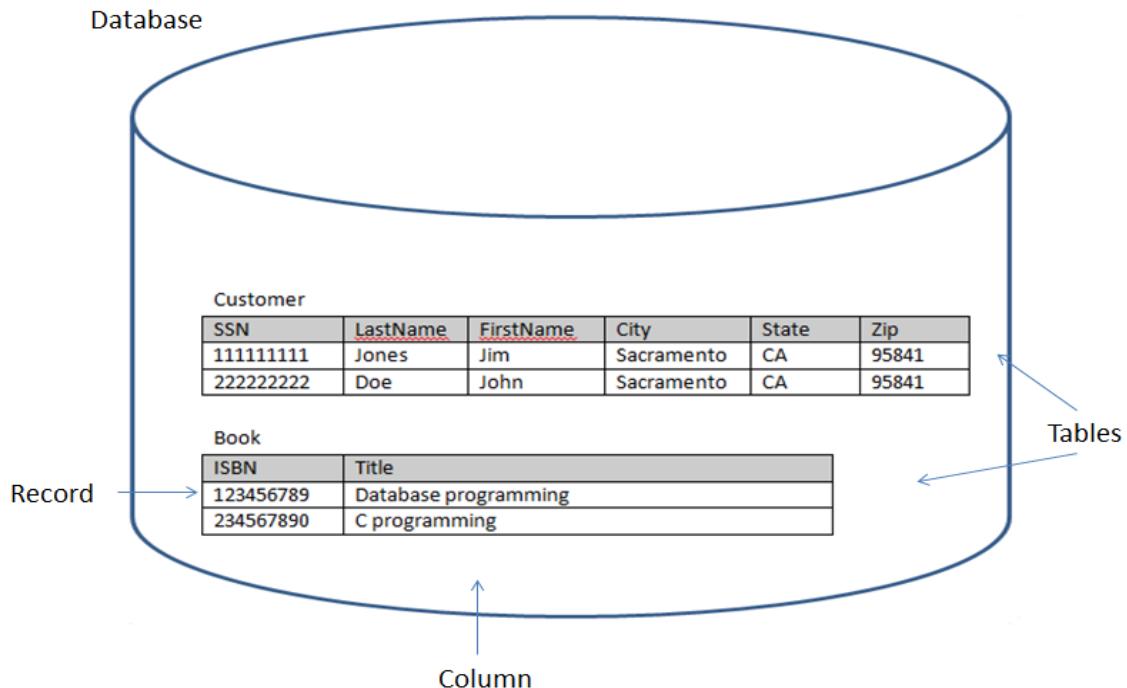
- Hierarchical model
- Network model
- Relational model
- Object Relational model
- Object Oriented Relational model.

The hierarchical data model organizes data in a tree structure. There is a hierarchy of parent and child data segments. The Network model allows for more than one parent to be associated with a child. An object database (also object-oriented database) is a database model in which information is represented in the form of objects as used in object-oriented programming. An object-relational database (ORD) is similar to a relational database, but with an object-oriented database model and can be said to provide a middle ground between relational databases and object-oriented databases. In object-relational databases, the approach is essentially that of relational databases. Our focus is on the relational model.

## 1.2 Relational model

A relational database matches data by using common characteristics found within the data set. The resulting groups of data are organized and are much easier for many people to understand.

- **Data hierarchy:** ordering of data types by size
  - **Field:** group of characters forming a single data item
    - “John”
  - **Record, row, tuple:** a group of related fields
    - An individual’s record containing ssn, lastname, firstname, city, state and zip
  - **Column:** is a set of data values of a particular type. Field value is used to refer specifically to the single item that exists at the intersection between one row and one column such as “Title”
- **Table, Entity:** a group of related records
  - Table “Customer” contains all the information about the various experiments
- **Database:** collection of related files, called tables.



## 1.3 DBMS

*“That is what learning is. You suddenly understand something you’ve understood all your life, but in a new way.”*

A Database Management System (DBMS) is a set of computer programs that controls the creation, maintenance, and the use of a database.

**■ Database management software**

- Create table descriptions
- Identify keys
- Add, delete, and update records within a table
- Sort records by different fields
- Write queries to select specific records for viewing
- Write queries to combine information from multiple tables
- Create reports
- Secure the data
  - Providing data integrity
  - Recovering lost data
  - Avoiding concurrent update problems
    - Two users make changes to the same record
    - Lock: mechanism to prevent changes to a record for some period of time
  - Providing authentication and permissions
    - Storing and verifying passwords
    - Using biometric data to identify users
    - settings that determine what actions a user is allowed to perform
  - Encryption (For data security)
    - Prevents use of the data by unauthorized users

## 1.4 RDBMS

The software used to do this grouping is called a relational database management system (RDBMS).

## 1.5 DBA vs. DA

*“A bad beginning makes a bad ending”*

A database administrator (DBA) is a person responsible for the implementation, maintenance and repair of an organization's database.

A data analyst is a person responsible for analyzing data requirements within an organization and modeling the data.

## 1.6 Some database vendors

- IBM DB2
- Microsoft SQL Server
- Informix
- Sybase Inc.
- Oracle
  - Based in Redwood, California
  - Leader in the worldwide relational and object-relational database management systems software market.

Our focus will be on the Oracle product.

## 1.7 SQL

It is referred to as Structured Query Language. It is a database computer language designed for managing data in relational database management systems (RDBMS). This language was originally based upon relational algebra and calculus. It is comprised of several sub-languages:

**Data Definition Language (DDL) statements are used to define the database structure or schema. Some examples:**

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- RENAME - rename an object

**Data Manipulation Language (DML) statements are used for managing data within schema objects. Some examples:**

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain
- MERGE - UPSERT operation (insert or update)
- CALL - call a PL/SQL or Java subprogram
- EXPLAIN PLAN - explain access path to data
- LOCK TABLE - control concurrency

**Data Control Language (DCL) statements. Some examples:**

- GRANT - gives user's access privileges to database
- REVOKE - withdraw access privileges given with the GRANT command

**Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.**

- COMMIT - save work done
- SAVEPOINT - identify a point in a transaction to which you can later roll back
- ROLLBACK - restore database to original since the last COMMIT
- SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

□ Example of a SQL statement

```
SELECT * FROM customer WHERE city='Paris';
```

## 1.8 SQL\*Plus

SQL\*Plus is an Oracle command-line utility program that can run SQL and PL/SQL commands interactively or from a script.

```
Select Command Prompt - sqlplus
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
SQL>
```

## 1.9 SQL vs. SQL\*Plus

SQL commands end with a semi-colon. They work directly with the database. SQLPlus commands do not need a semi-colon. SQLPlus is a text based environment that is used to type in SQL commands It allows you to interact with the database.

## 1.10 Some SQL\*Plus commands

**All these commands are oracle specific. They are not SQL commands. These commands are not case sensitive. Unlike SQL commands, SQLPlus commands do not end with semi-colon. The SQLPlus buffer will only store the last SQL command. The SQLPlus buffer does not store any SQLPlus commands.**

**/** Runs the SQL statement in the buffer.

**\$** Execute operating system command from within SQLPlus  
Example: \$ del c:\temp\test.SQL

**/\* \*/** These are multi-line comment symbols.  
Example : /\* some comments  
              This is my name. \*/

**--** This is also a comment but only for one line.  
Example: -- This is a comment

**APPEND** Appends to the end of your SQL buffer.  
Example: append user\_tables  
Appends user\_tables to the end of the same line.

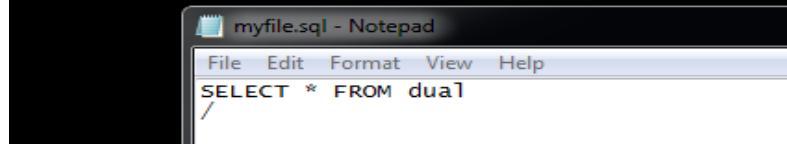
**CHANGE /from /to** Correct any spelling errors.  
**Note:** List the line number you want to change first and then issue the change command.  
Example: change /stff /stuff        or     c/stff/stuff  
Will change stff to stuff.

**clear buffer** Clears the buffer and lets you start over.

**CLEAR SCREEN** Clears the screen.

<b>CONNECT</b>	Logs you in as a different user. Example: connect po7 Logs you in as po7 and then prompts you for a password.
<b>DEL line number</b>	Will delete the line. Example: del 3 Deletes line 3 from the buffer.
<b>DISCONNECT</b>	Ends oracle session.
<b>EDIT filename</b>	Launches notepad with either a new file or an existing file. Example: edit c:\temp\test Note: The file extension will automatically be .SQL.
<b>HOST</b>	Execute operating system command from within SQLPlus. Example: host del c:\temp\test.sql NOTE: Have to identify both the file name and the file extension because you are issuing a command from the operating system which does not know what extension you have.
<b>INPUT</b>	Will add whatever text you want to the next line in the buffer. Example: input user_tables Will add the word user_tables to the next line in your buffer.
<b>LIST or L</b>	Will list everything in the SQLPlus buffer. Example: list or l
<b>LIST line number</b>	Will list the statement associated with the line number. Example: list 2 or l2 Lists only statement in line 2.
<b>REM</b>	Remark or comment. Example: REM This is a comment
<b>Run or R</b>	Runs whatever is in the buffer.
<b>SAVE Filename</b>	Saves the contents of the buffer into a new file. Example: save c:\temp\myfile Note: If file already exists then save with replace option Example: save c:\temp\test replace

```
SQL> SELECT * FROM dual;
D
-
X
SQL> l
  1* SELECT * FROM dual
SQL> SAVE c:\temp\myfile
Created file c:\temp\myfile.sql
SQL> edit c:\temp\myfile
```

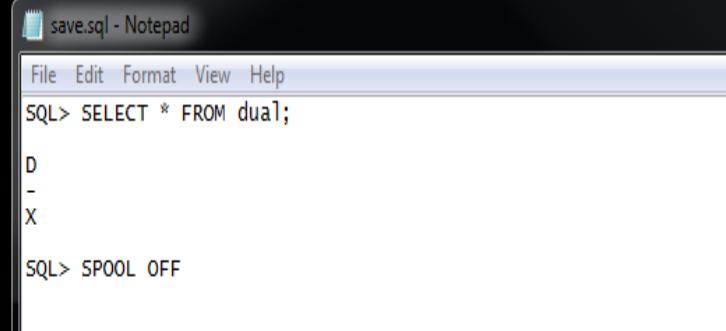


**SPOOL filename** Will save everything that you see on the SQLPlus screen to a file.  
 Example: spool c:\temp\save.sql  
 Writes to a file called c:\temp\save.sql. Make sure the directory exists and that you have permission to write to directory, otherwise you will get an error message as shown.

**SPOOL OFF** Stops writing to the file.

```
SQL> SPPOOL c:\temp1\save.sql
SP2-0606: Cannot create SPPOOL file "c:\temp1\save.sql"
SQL> SPPOOL c:\temp\save.sql
SQL> SELECT * FROM dual;
```

```
D
-
X
SQL> SPPOOL OFF
SQL> EDIT c:\temp\save
```



**START filename** Runs the contents of the file.  
 Example:Start c:\temp\test  
 Note: If the file does not contain a semi-colon or / then you will have to type in / to run the contents of the file.

**CAUTION: SPOOL vs (SAVE or EDIT)**

*When you edit or save a file, only the contents of the buffer, which contains SQL statements, will be in the file. However, the spool command will save everything that you see on your output screen to a file regardless of whether it is SQL or not.*

**Example 1.10a**

```
04. Command Prompt - sqlplus

SQL> SELECT (*
  2  FRM
  3  usr
  4  ;
SELECT (*
*
ERROR at line 1:
ORA-00936: missing expression

SQL> l1
  1* SELECT (*
SQL> c/*/* *
  1* SELECT *
SQL> l2
  2* FRM
SQL> c/FRM/FROM
  2* FROM
SQL> l3
  3* usr
SQL> c/usr/user_tables
  3* user_tables
SQL> l
  1  SELECT *
  2  FROM
  3  user_tables
  4*
SQL> r
```

## 1.11 Additional SQL\*Plus commands

<b>SHOW ALL</b>	Shows the status of all SQLPlus variables.
<b>SHOW ECHO</b>	Shows the status of the variable echo. Example: SET ECHO OFF Example: SET ECHO ON <i>Use the set command to assign a new value to a variable.</i>
<b>SHOW LINESIZE</b>	Shows the status of linesize which is the width of screen. Example: SET LINESIZE 50
<b>SHOW PAGESIZE</b>	Shows the status of pagesize which is the length of screen. Sets the number of lines that make up a page. <i>The header and the footer appear on top and bottom of the report, respectively. The number of lines per page determine when the header, footer and also the column headings appear in a report.</i>
<b>COLUMN</b>	Shows the status of all columns. All changes to columns will last for the oracle session.
<b>CLEAR COLUMNS</b>	Reset all column formatting to default.
<b>COLUMN original_column_name HEADING new_column_name</b>	Modify the column name.
<b>COLUMN original_column_name FORMAT A5</b>	Format text columns to display five characters.
<b>COLUMN salary FORMAT99999.99</b>	Format numeric columns. Make sure you consider the largest number in your table.
<b>COLUMN original_column_name TRUNCATED</b>	Truncate if column size is beyond the width you set with the format command.
<b>COLUMN original_column_name WORD_WRAPPED</b>	Word_wrapped is the default

***Example 1.11a***

```

SQL> DESC dual;
      Name          Null?    Type
-----  -----
DUMMY                           VARCHAR2(1)

SQL> SELECT * FROM dual;
D
-
X

SQL> COLUMN dummy FORMAT a10
SQL> select * from dual;

DUMMY
-----
X

SQL> COLUMN dummy HEADING stupid
SQL> SELECT * FROM dual;

stupid
-----
X

SQL> COLUMN dummy HEADING dummy
SQL> SELECT * FROM dual;

dummy
-----
X

SQL>

```

**FEEDBACK**

SHOW FEEDBACK

Reports on the number of rows retrieved.

SET FEEDBACK 2

Show the status of feedback variable.

Show number of rows if it is 2 or greater.

**TTITLE**

TTITLE CENTER "my report"

Header

Puts title on top of every page.

**BTITLE**

Footer

BTITLE LEFT SQL.PNO "my report"

Puts footer on the bottom of every page.

The footer is left justified.

Displays the page number using sql.pno

***Example 1.11b***

```

SQL> SET linesize 70
SQL> set pagesize 5
SQL> TTITLE left SQL.PNO center "Student Table"
SQL> BTITLE center "My report"
SQL> SELECT * FROM student;

          1           Student Table
FNAME    LNAME        AGE
-----  -----
john      Doe          20
                    My report

          2           Student Table
FNAME    LNAME        AGE
-----  -----
jill     Jones         25
                    My report

SQL>

```

## 1.12 PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle's Procedural extension language. PL/SQL's general syntax resembles that of Ada.

### ✓ CHECK 1A

1. What is the difference between SQL\*Plus, SQL and PL/SQL?
2. What is the SQL\*Plus buffer?
3. What are the SQL sub-languages?
4. What are some examples of SQL\*Plus variables?
5. What are some examples of SQL\*Plus commands?
6. How do you delete a file from within SQL\*Plus?

*“A man of words and not of deeds is like a garden full of weeds.”*

## Summary Examples

```
-- or REM or /* */ can be used as comment symbols.
-- An erroneous SQL statement that will be corrected using SQLPlus commands.
SELECT * FROM
user
table;

-- List contents of the buffer, which is the very last SQL statement (not SQLPlus).
L or LIST

-- Point to the first line.
L1

-- Change from SLECT to SELECT (Like find and replace).
C/SLECT/SELECT    or CHANGE/slect/select

--Get rid of the second line. All other lines are renumbered.
DEL 2

--Run contents of the buffer
RUN, R,   or /

--Save the contents of buffer to a file called hi.sql.
SAVE c:\temp\hi

--Edit the contents of the saved file.
EDIT c:\temp\hi

--Execute the contents of a file.
START c:\temp\hi

--Clear screen.
CLEAR SCREEN

--Clear the buffer.
CLEAR BUFFER

-- $ or host gets access to operating system commands.
$ del c:\temp\hi.sql    or HOST del c:\temp\hi.sql

--Show what the SQLPlus variable, pagesize, is set to.
SHOW PAGESIZE

--Set pagesize to 10
SET PAGESIZE 10

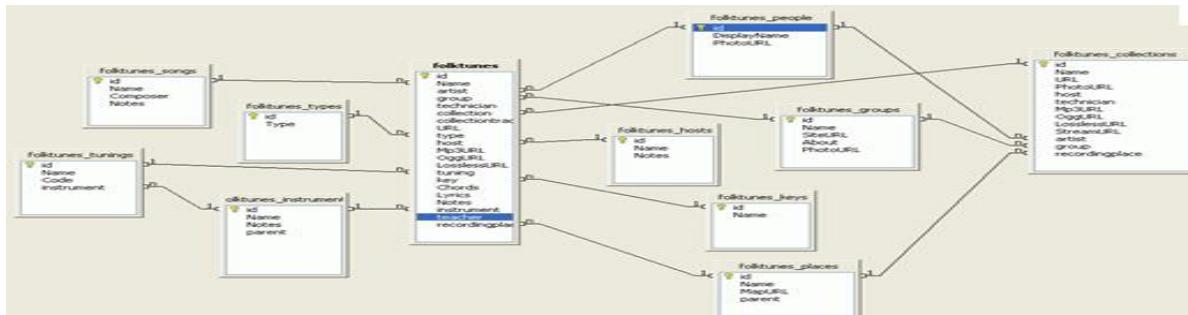
--Set linesize to 100
SET LINESIZE 100
```

```
--Set top title to (hello). Center it and include the page number.  
TTITLE CENTER "hello" SQL.PNO  
  
--Set footer to (goodbye) and include the page number.  
BTITLE LEFT "goodbye" SQL.PNO  
  
--Change the size of the column to 40 characters.  
COLUMN table_name A40  
  
--Display original column name, lname to lastname for the Oracle session.  
COLUMN lname HEADING lastname  
  
--Direct contents of the SQLPlus screen to a file.  
SPOOL c:\temp\redirect.txt  
  
--Turn off spooling.  
SPOOL OFF  
  
--Disconnect from database.  
DISCONNECT  
  
--Connect as a different user.  
CONNECT username/password or connect username
```



*In 1990, Oracle laid off 10% (about 400 people) of its work force because it was losing money. This crisis, which almost resulted in Oracle's bankruptcy, came about because of Oracle's "up-front" marketing strategy, in which sales people urged potential customers to buy the largest possible amount of software all at once. The sales people then booked the value of future license sales in the current quarter, thereby increasing their bonuses. This became a problem when the future sales subsequently failed to materialize. Oracle eventually had to restate its earnings twice, and also to settle out of court class action lawsuits arising from its having overstated its earnings.*

## Chapter 2 (Database Design)



*"Middle age is when you are warned to slow down by a doctor instead of a policeman."*

## 2.1 Poor Table Design

studentId	name	address	city	state	zip	class	classTitle
1	Rodriguez	123 Oak	Schaumburg	IL	60193	CIS101 PHI150 BIO200	Computer Literacy Ethics Genetics
2	Jones	234 Elm	Wild Rose	WI	54984	CHM100 MTH200	Chemistry Calculus
3	Mason	456 Pine	Dubuque	IA	52004	HIS202	World History

## 2.2 Normalization

The process of refining tables, keys, columns, and relationships to create an efficient database is called *normalization*. Normalization usually involves:

- ❑ Dividing a database into two or more tables
- ❑ Defining relationships between the tables
  
- Advantages
  - ❑ Reduce duplication of data
  - ❑ Avoiding irregularities Irregularities which can cause insert, update and delete issues.

## 2.3 Normal Forms

- First normal form
 

Each row and column intersection must contain one and only one value. Must be atomic. Eliminate repeating groups.
- Second normal form
 

Every non-key column must depend on the entire primary key. Eliminate partial key dependencies.
- Third normal form
 

No non-key column depends on another non-key column. Eliminate transitive dependencies.

- Fourth normal form

Forbids independent relationships between primary key columns and non-key columns.

- Fifth normal form

Breaks tables into the smallest possible pieces in order to eliminate redundancy.

Most designs implement up to the third normal form.

## 2.4 Primary key, Foreign key, Candidate or Alternate key

*“A drowning man will catch at a straw”*

Three attributes are the heart of data normalization:

- Primary key

- A field whose values are unique for each record in a table
- The Primary Key ensures that no two records in a database contain the same value for that field
- Creating relationships between tables
- May be composed of one or multiple columns

■ **Called a Compound or a composite primary key key**

- Unique/Candidate/Alternate key

A candidate key is a combination of attributes that can be uniquely used to identify a database record without any extraneous data. Each table may have one or more candidate keys. One of these candidate keys is selected as the table primary key.

- Alternate key used strictly for data retrieval purposes.
- May be composed of one or multiple columns

□ **Called a Compound or a composite candidate key**

- Foreign key

- A key field that identifies records in a different table. The foreign key is used to establish a relationship with another table or tables.

**Identify primary key:**

hall	room	bed	lastName	firstName	major
Adams	101	A	Fredricks	Madison	Chemistry
Adams	101	B	Garza	Lupe	Psychology
Adams	102	A	Liu	Jennifer	CIS
Adams	102	B	Smith	Crystal	CIS
Browning	101	A	Patel	Sarita	CIS
Browning	101	B	Smith	Margaret	Biology
Browning	102	A	Jefferson	Martha	Psychology
Browning	102	B	Bartlett	Donna	Spanish
Churchill	101	A	Wong	Cheryl	CIS
Churchill	101	B	Smith	Madison	Chemistry
Churchill	102	A	Patel	Jennifer	Psychology
Churchill	102	B	Jones	Elizabeth	CIS

## 2.5 First Normal Form (1NF)

- Unnormalized: table contains repeating groups
- Repeating group: subset of rows in a table all depend on the same key
- Table in 1NF contains no repeating groups of data
- Primary key attributes are defined
- Atomic attributes: columns contain undividable pieces of data
- In 1NF, all values for intersecting row and column must be atomic

studentId	name	address	city	state	zip	class	classTitle
1	Rodriguez	123 Oak	Schaumburg	IL	60193	CIS101 PHI150 BIO200	Computer Literacy Ethics Genetics
2	Jones	234 Elm	Wild Rose	WI	54984	CHM100 MTH200	Chemistry Calculus
3	Mason	456 Pine	Dubuque	IA	52004	HIS202	World History

studentId	name	address	city	state	zip	class	classTitle
1	Rodriguez	123 Oak	Schaumburg	IL	60193	CIS101	Computer Literacy
1	Rodriguez	123 Oak	Schaumburg	IL	60193	PHI150	Ethics
1	Rodriguez	123 Oak	Schaumburg	IL	60193	BIO200	Genetics
2	Jones	234 Elm	Wild Rose	WI	54984	CHM100	Chemistry
2	Jones	234 Elm	Wild Rose	WI	54984	MTH200	Calculus
3	Mason	456 Pine	Dubuque	IA	52004	HIS202	World History

## 2.6 Second Normal Form (2NF)

- Partial key dependencies: column depends on only part of the key
- For 2NF:
  - Database must already be in 1NF
  - All non-key fields must be dependent on the entire primary key
- Eliminate partial key dependencies by creating multiple tables
- Improvements over 1NF:
  - Eliminate update anomalies
  - Eliminate redundancies
  - Eliminate insert anomalies
  - Eliminate delete anomalies

## 2.7 Third Normal Form (3NF)

- All redundancies and anomalies are removed
- Normalization summary:
  - 1NF: no repeating groups
  - 2NF: 1NF plus no partial key dependencies
  - 3NF: 2NF plus no transitive dependencies

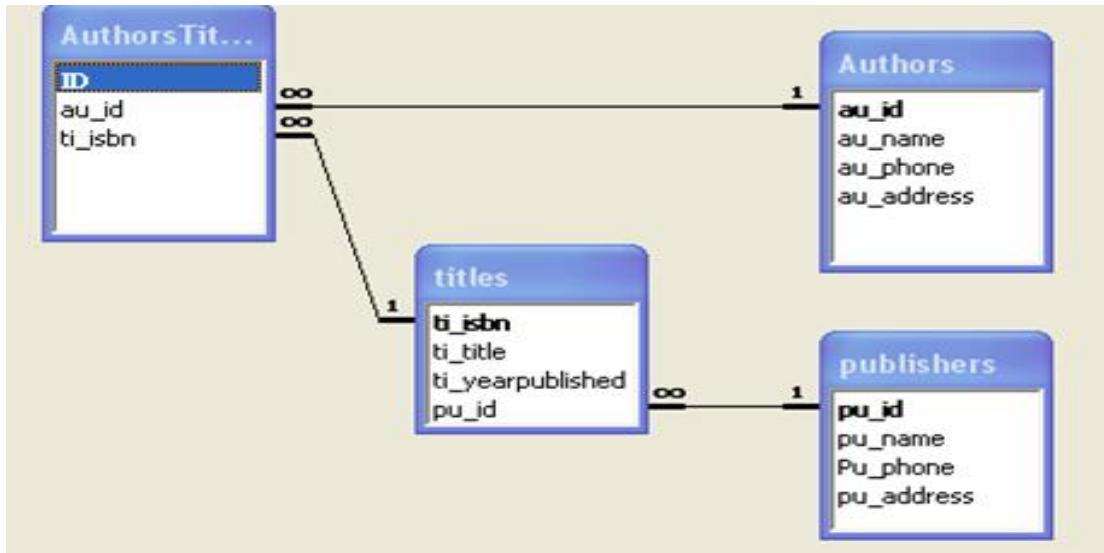
## 2.8 Entity Relationship Diagrams (ER)

Entity Relationship (ER) is a conceptual model that shows the structural organization of entities, relationships, and attributes. Three types of relationships:

- One-to-one: one instance of an entity (A) is associated with one other instance of another entity (B).
  - Row in one table corresponds to exactly one row in another table
  - Least frequently encountered relationship
  - Tables could be combined into a single table
  - Keep tables separate for security purposes
- One-to-many: one instance of an entity (A) is associated with zero, one or many instances of another entity (B), but for one instance of entity B there is only one instance of entity A.
  - Most common type of table relationship
  - Row in one table related to one or more rows in another table
  - “One” side is the base table, “Many” side is the related table
  - Primary key is used for the join
  - Foreign key: Field in one table which is primary key in another table

- Many-to-many: one instance of an entity (A) is associated with one, zero or many instances of another entity (B), and vice versa.

- Multiple rows in each table can correspond to multiple rows in the other table.
- Many to many relationships should be eliminated through the addition of a bridge or an association table which results in one to many relationships.



## 2.9 One to One

tblEmployees					tblSalaries	
empId	empLast	empFirst	empDept	empHireDate	empId	empSalary
101	Parker	Laura	3	4/07/2000	101	\$42,500
102	Walters	David	4	1/19/2001	102	\$28,800
103	Shannon	Ewa	3	2/28/2005	103	\$36,000

## 2.10 One to Many

<b>tblItems</b>				
<b>itemNumber</b>	<b>itemName</b>	<b>itemPurchaseDate</b>	<b>itemPurchasePrice</b>	<b>itemcategoryId</b>
1	Sofa	1/13/2003	\$6,500	5
2	Stereo	2/10/2005	\$1,200	6
3	Refrigerator	5/12/2005	\$750	1
4	Diamond ring	2/12/2006	\$42,000	2
5	TV	7/11/2006	\$285	6
6	Rectangular pine coffee table	4/21/2007	\$300	5
7	Round pine end table	4/21/2007	\$200	5

<b>tblCategories</b>		
<b>categoryId</b>	<b>categoryName</b>	<b>categoryInsuredAmount</b>
1	Appliance	\$30,000
2	Jewelry	\$15,000
3	Antique	\$10,000
4	Clothing	\$25,000
5	Furniture	\$5,000
6	Electronics	\$2,500
7	Miscellaneous	\$5,000

<b>tblCustomers</b>		<b>tblOrders</b>				
<b>customerNumber</b>	<b>customerName</b>	<b>orderNumber</b>	<b>customerNumber</b>	<b>orderQuantity</b>	<b>orderItem</b>	<b>orderDate</b>
214	Kowalski	10467	215	2	HP203	10/15/2009
215	Jackson	10468	218	1	JK109	10/15/2009
216	Lopez	10469	215	4	HP203	10/16/2009
217	Thompson	10470	216	12	ML318	10/16/2009
218	Vitale	10471	214	4	JK109	10/16/2009
		10472	215	1	HP203	10/16/2009
		10473	217	10	JK109	10/17/2009

## 2.11 Many to Many

tblItems

itemNumber	itemName	itemPurchaseDate	itemPurchasePrice
1	Sofa	1/13/2003	\$6,500
2	Stereo	2/10/2005	\$1,200
3	Sofa with CD player	5/24/2007	\$8,500
4	Table with DVD player	6/24/2007	\$12,000
5	Grandpa's pocket watch	12/24/1929	\$100

tblItemsCategories

itemNumber	categoryId
1	5
2	6
3	5
3	6
4	5
4	6
5	2
5	3

tblCategories

categoryId	categoryName	categoryInsuredAmount
1	Appliance	\$30,000
2	Jewelry	\$15,000
3	Antique	\$10,000
4	Clothing	\$25,000
5	Furniture	\$5,000
6	Electronics	\$2,500
7	Miscellaneous	\$5,000

## 2.12 Normalization example

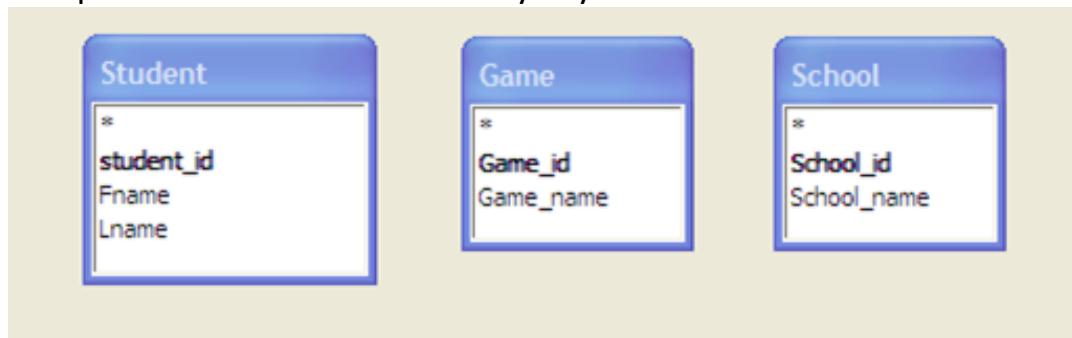
### Statement of the Project:

"We want to store information about a bunch of **students** who play different kinds of **games**. In addition, we want to store the name of the **school** that they attend."

We make the following assumptions:

- Each student can play many games.
- Each student attends only one school.

First Step: Create Tables with Primary keys



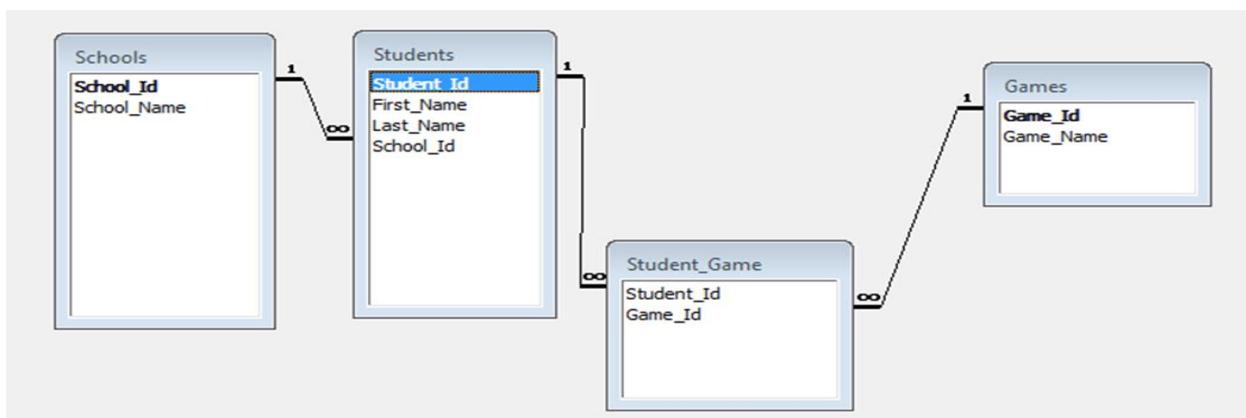
Second Step: Add Bridge table between Student and Game

- To construct a bridge between the Students and Games table, we must answer the following:
  - Can a Student play many games? Can a game be played by several students? YES
  - If the answer is yes, we have a many to many relationship.
  - This means that a bridge table is required between the Student and Game tables.

Third Step: Relationship between Student and School

- Do we need a bridge between the School and Student tables?
  - Can a school be attended by several students? Yes
  - Can a student attend different Schools? No, by assumption
  - Here we have one to man relationship. No need for bridge table

## Normalized database



## Primary and Foreign Keys:

- The primary keys:
  - School table –School\_id
  - Student table – Student\_id
  - Game table – Game\_id
- Foreign Keys:
  - Students table – School\_id
  - Student\_game - Student\_id, Game\_id
- Candidate keys
 

A possible candidate key can be a combination of last name and first name in the student table, assuming no two students have the same first name and last name. This is a bad assumption and is only used to illustrate a composite candidate key.

## 2.13 Normalize the following

### Problem 1:

Name of table: Disease

Name	Age	Address	Disease1	Disease2	Disease3
Bob Smith	25	111 J street	malaria	yellow fever	Bird flu
Jack Jones	35	111 k street	malria	AIDS	
Julie smith	45	111 J street	ADS	Brd flu	

### Problem 2

Name of table: Biography

Student	Religion	Ethn1	Ethn2
John Doe	Buddhist	White	
Jack smith	Christian	Black	white
Tiger Woods	Moslem	Asian	

Note: Tiger woods identifies himself as Cabinasian (Caucasian, Black, Indian, Asian)

✓ **CHECK 2A**

1. What is normalization?
2. What are the different normal forms?
3. What are the different types of relationships between tables?
4. What is the difference between a primary key, unique key and foreign key?
5. How many primary keys and unique keys can a table have?
6. What is a bridge table?
7. Create a normalized database that can store different people's personality type. Here is a sample of the denormalized data:

SSN	Fname	Lname	Salary	DOB	Personality type
111	John	Germs	100000	1/1/1990	Good
112	Jill	Fumbles	50000	2/1/91	Bad
113	James	Grapes	200000	3/1/88	Bad
114	Jack	Fickle	900000	3/2/77	Ugly

*“True friendship is like sound health; the value of it is seldom known until it is lost.”*



Larry Ellison has been married and divorced several times. He was married to Adda Quinn from 1967 to 1974. He was married to Nancy Wheeler Jenkins between 1977 and 1978. From 1983 to 1986, he was married to Barbara Boothe: two children were born of this marriage, a son and daughter named David and Megan. On 18 December 2003, Ellison married Melanie Craft, a romance novelist, at his Woodside estate. His friend Steve Jobs (CEO of Apple, Inc) was the official wedding photographer and Representative Tom Lantos officiated. Ellison and Melanie Craft-Ellison divorced in September 2010.

## Chapter 3 (DDL commands)

"People never grow up; they just learn how to act in public."

Start

### 3.1 Creating and dropping tables

Things to note:

- For readability, upper case all the Oracle Reserved words.
- Oracle is not case sensitive. The data being inserted is case sensitive.
- Separate every column definition with a comma except for the last one
- Use a semicolon to end the SQL statement.
- Do not put any blank lines between your code otherwise you will get an error message.
- A user cannot have two tables with the same name.
- Identifiers cannot be more than 30 characters long.
- A table can have up to 1000 columns

#### Basic syntax

```
CREATE TABLE tablename
(
    Columnname    TYPE,
    Columnname    TYPE,
    Columnname    TYPE
);
```

#### Some common types:

**VARCHAR2(n)** Variable-length character data where **n** represents the column's maximum length.  
The maximum size is 4000 characters.

**CHAR(n)** Fixed-length character columns where **n** represents the column's length. The default is 1 and the maximum size is 2000.

**NUMBER(p,s)** Numeric column where **p** indicates precision(total number of digits to the left and right of the decimal position- max 38 digits) and **s** indicates scale (number of positions to the right of the decimal).

**DATE** Stores date and time between January 1, 4712 BC and December 31, 9999 AD.  
Oracle's default date format is DD-MON-YY.

**NOTE:** There are other datatypes such BINARY\_FLOAT, BINARY\_DOUBLE, INTEGER, LONG, CLOB, RAW, LONG RAW, BLOB, BFILE, TIMESTAMP and INTERVAL.

### ***Example 3.1a (Create table)***

Create a new, empty table called patient

```
CREATE TABLE patient
(
    patient_id      NUMBER,
    fname           VARCHAR2(20),
    lname           VARCHAR2(30),
    gender          CHAR,
    DOB             DATE,
    annual_salary   NUMBER
);
```

```
table PATIENT created.
```

### ***Example 3.1b (DESC command)***

DESC provides information about the columns in a table. The columns appear on the left hand side and their datatypes on the right hand side. The Heading NULL will be discussed later. A semicolon is not required for the DESC command.

```
DESC patient;
```

```
table PATIENT created.
DESC patient
Name      Null Type
-----
PATIENT_ID      NUMBER
FNAME          VARCHAR2(20)
LNAME          VARCHAR2(30)
GENDER          CHAR(1)
DOB             DATE
ANNUAL_SALARY   NUMBER
```

### ***Example 3.1c (View contents of table)***

Looking at the contents of the table. Currently there are no rows in the patient table.

```
SELECT * FROM patient;
```

PATIENT_ID	FNAME	LNAME	GENDER	DOB	ANNUAL_SALARY
------------	-------	-------	--------	-----	---------------

### ***Example 3.1d (Drop table command)***

Use the DROP TABLE statement to move a table to the recycle bin. All the data inside the table is erased.

```
Drop TABLE patient;
```

```
table PATIENT dropped.
```

### **Example 3.1e (Flashback)**

A dropped table which is moved to the recycle bin can be recovered using FLASHBACK TABLE command.

```
--Table does not exist since it was dropped earlier.
DESC patient;

--Restore the table
FLASHBACK TABLE patient TO BEFORE DROP;

--Confirm that the table has been restored
DESC patient;

```

DESC patient ERROR: ----- ERROR: object PATIENT does not exist
table PATIENT succeeded. DESC patient Name Null Type ----- PATIENT_ID NUMBER FNAME VARCHAR2(20) LNAME VARCHAR2(30) GENDER CHAR(1) DOB DATE ANNUAL_SALARY NUMBER

### **Example 3.1f (Recyclebin)**

Purging from the recyclebin

```
CREATE TABLE tst
(
  col CHAR
);

INSERT INTO tst VALUES ('a');          --Insert a row into the table
SELECT * FROM tst;                  --Examine contents
DROP TABLE tst;                    --Remove table

FLASHBACK TABLE tst TO BEFORE DROP; --Recover table

SELECT * FROM tst;                --Confirm recovery
DROP TABLE tst;
PURGE RECYCLEBIN;                --Recovery is not possible after this line
FLASHBACK TABLE tst TO BEFORE DROP; --Error: Nothing to recover
```

```

table TST created.
1 rows inserted.
COL
---
a

table TST dropped.
table TST succeeded.
COL
---
a

table TST dropped.
purge recyclebin

Error starting at line : 18 in command -
flashback table tst to before drop
Error report -
SQL Error: ORA-38305: object not in RECYCLE BIN
38305. 00000 - "object not in RECYCLE BIN"
*Cause: Trying to Flashback Drop an object which is not in RecycleBin.
*Action: Only the objects in RecycleBin can be Flashback Dropped.

```

### **Example 3.1g (Not NULL and default constraints)**

Create a table with a NOT NULL and DEFAULT constraint. If no value is entered for a column, the value is considered NULL, indicating an absence of data.

```

CREATE TABLE patient
(
    patient_id      NUMBER NOT NULL,
    fname           VARCHAR2(20),
    lname           VARCHAR2(30),
    gender          CHAR DEFAULT 'm',
    DOB             DATE,
    annual_salary   NUMBER
);

Desc patient;
table PATIENT created.
Desc patient
Name      Null      Type
-----
PATIENT_ID  NOT NULL NUMBER
FNAME           VARCHAR2(20)
LNAME           VARCHAR2(30)
GENDER          CHAR(1)
DOB             DATE
ANNUAL_SALARY   NUMBER

```

### **Example 3.1h (Insert statement)**

Insert two rows of data into the table. Notice the data that is being inserted into the table is case-sensitive but the syntax is not.

```
INSERT INTO patient VALUES (11, 'John', 'Smith', 'm', '01-FEB-1970',
55000);
INSERT INTO patient VALUES (12, 'Jill', 'Doe', 'f', '20-FEB-1970', 95000);
1 rows inserted.
1 rows inserted.
```

### **Example 3.1i (View contents)**

Look at the contents of the table

--The asterisk shows all the columns for each of the two rows.

```
SELECT * FROM patient;
```

--This statement only shows the three columns for each of the two rows.

```
SELECT fname, lname, DOB FROM patient;
```

PATIENT_ID	FNAME	LNAME	GENDER	DOB	ANNUAL_SALARY
11	John	Smith	m	01-FEB-70	55000
12	Jill	Doe	f	20-FEB-70	95000
FNAME	LNAME	DOB			
John	Smith	01-FEB-70			
Jill	Doe	20-FEB-70			

### **Example 3.1j (Data dictionary tables: user\_tables)**

System tables: Oracle uses a Data Dictionary to store details of all the Tables, Columns etc..Here is an example of a row that is automatically inserted into one of the system tables (user\_tables). Other system tables that we will be exploring will be dictionary, user\_constraints, user\_cons\_columns, user\_indexes, user\_ind\_columns.

--Only some of the columns are displayed.

```
DESC user_tables;
```

```
DESC user_tables
Name           Null    Type
-----
TABLE_NAME      NOT NULL VARCHAR2(30)
TABLESPACE_NAME          VARCHAR2(30)
CLUSTER_NAME        VARCHAR2(30)
IOT_NAME          VARCHAR2(30)
STATUS            VARCHAR2(8)
PCT_FREE          NUMBER
PCT_USED          NUMBER
INI_TRANS         NUMBER
MAX_TRANS         NUMBER
INITIAL_EXTENT   NUMBER
NEXT_EXTENT       NUMBER
MIN_EXTENTS      NUMBER
MAX_EXTENTS      NUMBER
PCT_INCREASE     NUMBER
FREELISTS         NUMBER
FREELIST_GROUPS  NUMBER
LOGGING           VARCHAR2(3)
```

--All the tables that have been created by the user that is logged in are displayed.

```
SELECT table_name FROM user_tables;
```

```
TABLE_NAME
-----
DESKTOP
PATIENT
```

### ***Example 3.1k (Delete)***

The table will still exist but its contents will be deleted. The data can still be recovered if an implicit or explicit commit has not been implemented. Delete can be applied to all or specific rows in a table.

--Notice that there is no asterisk(\*) in the delete statement.

```
DELETE FROM patient;
```

```
2 rows deleted.
```

### ***Example 3.1l (Truncate)***

The table will still exist but its contents will be deleted. The data cannot be recovered. It is a lot faster than delete. Unlike delete, it is applied to all the rows in the table.

```
TRUNCATE TABLE patient;
```

```
table PATIENT truncated.
```

### **Example 3.1m (Order of columns)**

Can identify the columns in any order.

```
CREATE TABLE patient
(
    fname          VARCHAR2(20),
    lname          VARCHAR2(30),
    DOB            DATE,
    patient_id    NUMBER,
    gender         CHAR,
    annual_salary NUMBER
);
```

### **Example 3.1n (Problems??)**

What is wrong the following?

```
--What is wrong?
Delete * FROM patient;

Error starting at line : 1 in command -
Delete * FROM patient
Error at Command Line : 1 Column : 8
Error report -
SQL Error: ORA-00903: invalid table name
00903. 00000 - "invalid table name"
*Cause:
*Action:
```

```
CREATE TABLE
{
    Patient_id      NUMBER,
    Fname           VARCHAR2
                    VARCHAR2(20),
    Lname           VARCHAR2(20),
    Gender          CHAR,
    DOB             DATE (10),
    Annual_salary   NUMBER,
};
```

```
Error starting at line : 1 in command -
CREATE  TABLE
{
    Patient_id  NUMBER,
    Fname      VARCHAR2
    Lname      VARCHAR2(20),

    Gender     CHAR,
    Dob        DATE (10),
    Annual_salary  NUMBER,
}
Error at Command Line : 2 Column : 1
Error report -
SQL Error: ORA-00903: invalid table name
00903. 00000 -  "invalid table name"
*Cause:
*Action:
```

### **Example 3.1o (Renaming a table)**

--Renames the table patient to sickPerson.  
RENAME patient TO sickperson;

### **✓ CHECK 3A**

1. Create a table called Person comprised of SSN, Iname, fname, and salary columns.
2. Insert a record into the table and view its data.
3. Confirm the entry in the system table (USER\_TABLES).
4. Delete the record.
5. Truncate the table.
6. Drop the table.
7. What is the difference between delete and truncate?
8. What is the difference between CHAR and VARCHAR?

```
CREATE TABLE Person(
    SSN  NUMBER,
    Iname VARCHAR(20),
    fname VARCHAR(25),
    salary NUMBER
);
DROP TABLE Person;
INSERT INTO Person VALUES(11, 'dee', 'Jam', 50000);
DESC user_tables;
DELETE FROM Person;
TRUNCATE TABLE Person;
DROP TABLE Person;
```

*"A quiet conscience sleeps in thunder"*

## 3.2 Adding columns using ALTER command

At times, you need to make structural changes to a table. For example, you might need to add a column, delete a column, or simply change a column's size. Each of these changes is made with the ALTER TABLE command.

ALTER TABLE tablename ADD | MODIFY | DROP | columnname (definition);

Using an ADD clause with the ALTER TABLE command allows a user to add a new column to a table. The same rules for creating a column in a new table apply to adding a column to an

existing table. The new column must be defined by a column name and datatype (and width, if applicable). A default value can also be assigned. The difference is that the new column is added at the end of the existing table— it will be the last column.

### ***Example 3.2a (Additional column)***

Add an additional column of type char

--This statement adds marital\_status as a column to the table patient. The datatype of this new --column is CHAR.

```
ALTER TABLE patient ADD marital_status CHAR;
```

table PATIENT altered.

```
ALTER TABLE person ADD marital_status CHAR
```

Name	Null	Type
PATIENT_ID	NOT NULL	NUMBER
FNAME		VARCHAR2(20)
LNAME		VARCHAR2(20)
GENDER		CHAR(1)
DOB		DATE
ANNUAL_SALARY		NUMBER
MARITAL_STATUS		CHAR(1)

### ***Example 3.2b (Adding multiple columns)***

Adding multiple columns at the same type

--For multiple columns, the syntax looks like the create table statement.

```
ALTER TABLE patient ADD
(
    Height NUMBER,
    Weight NUMBER
);
```

```
ALTER TABLE Person ADD(
    ...
);
```

Name	Null	Type
PATIENT_ID	NOT NULL	NUMBER
FNAME		VARCHAR2(20)
LNAME		VARCHAR2(20)
GENDER		CHAR(1)
DOB		DATE
ANNUAL_SALARY		NUMBER
MARITAL_STATUS		CHAR(1)
HEIGHT		NUMBER
WEIGHT		NUMBER

### 3.3 Modifying using ALTER command

To change an existing column's definition, you can use a MODIFY clause with the ALTER TABLE command. The changes that can be made to a column include the following:

- Changing the column size ( increase or decrease)
- Changing the datatype ( such as VARCHAR2 to CHAR) •
- Changing or adding the default value of a column ( such as DEFAULT SYSDATE)

You should be aware of three rules when modifying existing columns: •

- A column must be as wide as the data fields it already contains.
- If a NUMBER column already contains data, you can't decrease the column's precision or scale.
- Changing the default value of a column doesn't change the values of data already in the table

#### *Example 3.3a (Modify)*

As long as the table is empty, the type can be modified and the constraint can be changed without any problems.

```
DROP TABLE patient;

CREATE TABLE Patient
(
    Patient_id  NUMBER NOT NULL,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20),
    Gender      CHAR DEFAULT 'm',
    DOB         DATE,
    Annual_salary NUMBER
);

--We are free to change the datatypes by shortening, lengthening or even modifying
--from textual to numeric, as long as there is no data in the table. Once there is data, then there are
--some restrictions.
ALTER TABLE patient MODIFY fname VARCHAR2(19);
ALTER TABLE patient MODIFY fname VARCHAR2(29);
ALTER TABLE patient MODIFY fname NUMBER;
ALTER TABLE patient MODIFY fname VARCHAR2(20);
ALTER TABLE patient MODIFY patient_id CHAR;
ALTER TABLE patient MODIFY patient_id NUMBER NULL;
ALTER TABLE patient MODIFY patient_id NOT NULL;
```

```
table PATIENT dropped.
table PATIENT created.
table PATIENT altered.
```

### ***Example 3.3b (Restrictions on modify)***

The modify command is more restrictive in terms of what can and cannot be done if the table is not empty

```
DROP TABLE patient;

CREATE TABLE Patient
(
    Patient_id      NUMBER NOT NULL,
    Fname           VARCHAR2(20),
    Lname           VARCHAR2(20),
    Gender          CHAR DEFAULT 'm',
    DOB             DATE,
    Annual_salary   NUMBER
);

INSERT INTO patient VALUES (11, 'John', 'Smith', 'm', '01-FEB-1970', 55000);
INSERT INTO patient VALUES (12, 'Jill', 'Doe', 'f', '20-FEB-1970', 95000);

--Only the results from this point forward are displayed below.
ALTER TABLE patient MODIFY fname VARCHAR2(19);

--INVALID: The length is too short because it truncates existing data.
ALTER TABLE patient MODIFY fname VARCHAR2(3);

ALTER TABLE patient MODIFY fname VARCHAR2(29);           --valid

--INVALID: Datatype does not match existing data.
ALTER TABLE patient MODIFY fname NUMBER;

ALTER TABLE patient MODIFY fname VARCHAR2(20);           --valid

--INVALID: Data type does not match.
ALTER TABLE patient MODIFY patient_id CHAR;

ALTER TABLE patient MODIFY patient_id NUMBER NULL;      --valid
ALTER TABLE patient MODIFY patient_id NOT NULL;        --valid
```

```

table PATIENT altered.
Error starting at line 16 in command:
ALTER TABLE patient MODIFY fname VARCHAR2(3)
Error report:
SQL Error: ORA-01441: cannot decrease column length because some value is too big
01441. 00000 - "cannot decrease column length because some value is too big"
*Cause:
*Action:
table PATIENT altered.

Error starting at line 18 in command:
ALTER TABLE patient MODIFY fname NUMBER
Error report:
SQL Error: ORA-01439: column to be modified must be empty to change datatype
01439. 00000 - "column to be modified must be empty to change datatype"
*Cause:
*Action:
table PATIENT altered.

Error starting at line 20 in command:
ALTER TABLE patient MODIFY patient_id CHAR
Error report:
SQL Error: ORA-01439: column to be modified must be empty to change datatype
01439. 00000 - "column to be modified must be empty to change datatype"
*Cause:
*Action:
table PATIENT altered.
table PATIENT altered.

```

## 3.4 Dropping columns using ALTER command

### *Example 3.4a (Dropping column)*

Getting rid of a column (Cannot be rolled back)

<b>--Add the column</b> <pre>ALTER TABLE patient ADD height NUMBER; DESC patient;</pre>	<b>--Get rid of the column and all the data in it</b> <pre>ALTER TABLE Person DROP (height); DESC patient;</pre>
--	---

```

table PATIENT altered.
DESC patient
Name      Null      Type
-----
PATIENT_ID    NOT NULL NUMBER
FNAME          VARCHAR2(20)
LNAME          VARCHAR2(20)
GENDER         CHAR(1)
DOB            DATE
ANNUAL_SALARY  NUMBER
HEIGHT         NUMBER

table PATIENT altered.
DESC patient
Name      Null      Type
-----
PATIENT_ID    NOT NULL NUMBER
FNAME          VARCHAR2(20)
LNAME          VARCHAR2(20)
GENDER         CHAR(1)
DOB            DATE
ANNUAL_SALARY  NUMBER

```

### **Example 3.4b (Dropping columns)**

Getting rid of multiple columns.

```
--Use a comma to separate the columns that are to be dropped. All the data along with the columns
--will be discarded.
ALTER TABLE patient  DROP (annual_salary, marital_status);

DESC patient;
```

```

table PATIENT altered.
DESC patient
Name      Null      Type
-----
PATIENT_ID    NOT NULL NUMBER
GENDER         CHAR(1)
DOB            DATE
ANNUAL_SALARY  NUMBER

```

```
ALTER TABLE person DROP (lname, fname);
```

### **Example 3.4c (Set unused)**

Setting columns to unused and then dropping them (Cannot be rolled back).

setting columns to unused cannot be rolled back.

--Cannot recover the data

```
ALTER TABLE patient SET UNUSED (DOB, gender);
DESC patient;
```

```
ALTER TABLE Person SET UNUSED (fname, lname)
cannot be recovered once dropped.
ALTER TABLE Person DROP (fname, lname);
```

--Cannot recover the data

```
ALTER TABLE patient DROP UNUSED COLUMNS;
```

```

table PATIENT altered.
DESC patient
Name      Null      Type
-----
PATIENT_ID    NOT NULL NUMBER
ANNUAL_SALARY           NUMBER

table PATIENT altered.

```

## 3.5 Renaming column using ALTER command

```
ALTER TABLE Perspn RENAME COLUMN fname TO first_name;
```

### **Example 3.5a (Renaming a column)**

Renames a column from fname to first\_name. Only one column can be renamed at a time.

```
ALTER TABLE patient RENAME COLUMN fname TO first_name;
```

```

CREATE TABLE Person(
SSN NUMBER,
Iname VARCHAR(20),
fname VARCHAR(25),
salary NUMBER
);
DROP TABLE Person;
ALTER TABLE Person ADD DOB CHAR(10);
SELECT * FROM Person;
ALTER TABLE Person MODIFY DOB VARCHAR(10);
ALTER TABLE Person DROP(fname);
ALTER TABLE Person SET UNUSED COLUMN DOB;
ALTER TABLE Person DROP UNUSED COLUMNS;
ALTER TABLE Person RENAME COLUMN Iname TO fname;

```

### ✓ **CHECK 3B**

1. Add the column DOB (type char(10)) to the Person table using the ALTER syntax.
2. Modify the datatype to date.
3. Drop the column fname.
4. Set the DOB column to unused and then drop all unused columns.
5. Rename the Iname column to fname using the ALTER syntax.

*"I can't change the direction of the wind, but I can adjust my sails to always reach my destination."*

## 3.6 Constraints

Constraints are rules used to enforce business rules, practices, and policies to ensure the accuracy and integrity of data. You can add constraints during table creation as part of the CREATE TABLE command, or you can do so after the table is created by using the ALTER TABLE command.

When creating a table, you can create a constraint in two ways: at the column level or the table level. Creating a constraint at the column level means the constraint's definition is included as part of the column definition, similar to assigning a default value to a column. Creating a constraint at the table level means the constraint's definition is separate from the column definition. The main difference in the syntax of a column- level constraint and a table- level constraint is that you provide column names for the table- level constraint at the end of the constraint definition inside parentheses, instead of at the beginning of the constraint definition.

You can create any type of constraint at the column level— unless the constraint is being defined for more than one column (for example, a composite primary key). If the constraint applies to more than one column, you must create the constraint at the table level. Also, a NULL and DEFAULT constraint can only be created at the column level.

PRIMARY KEY	Determines which column(s) uniquely identifies each record. The primary key cannot be NULL and the data values must be unique
FOREIGN KEY	In one to many relationships, the constraint is added to the “many” table. The constraint ensures that if a value is entered in a specified column, it must already exist in the “One” table, or the record isn’t added
UNIQUE	Ensures that all data values stored in specified column are unique. The UNIQUE constraint differs from the PRIMARY KEY constraint in that it allows NULL values.
CHECK	Ensures that a specified condition is true before the data value is added to a table. For example, an order’s ship date can’t be earlier than its order date.
NOT NULL	Ensures that a specified column can’t contain a NULL value. The NOT NULL constraint can be created only with the column level approach to the table creation.

When creating a constraint, you can name the constraint or omit the constraint name and allow Oracle to generate the name. If the Oracle names the constraint, it follows the format SYS\_Cn, where n is an assigned numeric value to make the name unique. Providing a descriptive name for a constraint is the preferred practice so that you can identify it easily in the future. For example, constraint violation errors reference the constraint name, so an easy- to- understand name indicating the table, column, and type of constraint is quite helpful. Industry convention is to use the format tablename\_ columnname\_ constrainttype for the constraint name— for example, patient\_patient#\_ pk. Constraint types are designated by abbreviations, as such:

Constraint	Abbreviation
PRIMARY KEY	_pk
FOREIGN KEY	_fk
UNIQUE	_uk
CHECK	_ck
NOT NULL	_nn

When creating a constraint using the ALTER statement, the add and the modify options can be used. A constraint name cannot be assigned with the modify option when creating a check constraint.

- A constraint is a rule applied to data being added to a table. It represents business rules, policies, or procedures. Data violating the constraint isn’t added to the table.
- A constraint can be included during table creation as part of the CREATE TABLE command or added to an existing table with the ALTER TABLE command.

- A constraint based on composite columns (more than one column) must be created by using the table- level approach.
- **A NOT NULL constraint** can be created only with the column- level approach.
- A **PRIMARY KEY** constraint doesn't allow duplicate or NULL values in the designated column.
- **Only one PRIMARY KEY** constraint is allowed in a table.
- A FOREIGN KEY constraint requires that the column entry match a referenced column entry in the table or be NULL.
- A UNIQUE constraint is similar to a PRIMARY KEY constraint, except it allows storing NULL values in the specified column.
- A CHECK constraint ensures that data meets a given condition before it's added to the table. The condition can't reference the **SYSDATE** function or values stored in other rows.
- **A NOT NULL** constraint is a special type of CHECK constraint. If you're adding to an existing column, the ALTER TABLE ... MODIFY command must be used.
- A column can be assigned multiple constraints.
- The data dictionary views **USER\_CONSTRAINTS** and **USER\_CONS\_COLUMNS** enable you to verify existing constraints.
- A constraint can be disabled or enabled with the ALTER TABLE command and the DISABLE and ENABLE keywords.
- A constraint can't be modified. To change a constraint, you must first drop it with the DROP command and then re- create it.

When a constraint exists for a column, each entry made to that column is evaluated to determine whether the value is allowed in that column (that is, it doesn't violate the constraint). If you're adding a large block of data to a table, this validation process can severely slow down the Oracle server's processing speed. If you're certain the data you're adding adheres to the constraints, you can disable the constraints while adding that particular block of data to the table.

## 3.7 Primary key constraint

### *Example 3.7a (Primary key at the column level)*

The constraint makes certain the columns identified as the table's primary key are unique and do not contain **NULL** values.

```

DROP TABLE patient;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);

INSERT INTO patient VALUES (11,'John', 'Smith');

--The primary key is violated because patient_id (11) is repeated.
INSERT INTO patient VALUES (11, 'Jill', 'Doe');

```

```

table PATIENT dropped.
table PATIENT created.
1 rows inserted.

Error starting at line 12 in command:
INSERT INTO patient VALUES (11, 'Jill', 'Doe')
Error report:
SQL Error: ORA-00001: unique constraint (IRAJ.SYS_C0013286) violated
00001. 00000 - "unique constraint (%s.%s) violated"
*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.
        For Trusted Oracle configured in DBMS MAC mode, you may see
        this message if a duplicate entry exists at a different level.
*Action: Either remove the unique restriction or do not insert the key.

```

### ***Example 3.7b (User\_constraints table)***

Examining the system table user\_constraints

```

--Constraint_type (p) stands for primary key. The constraint name is generated by the system.
--In the SQL statement, PATIENT must be typed as upper-case because that is how it is stored in
--ORACLE system table.
SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';

```

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT	SYS_C0013286	P

### **Example 3.7c (Giving name to constraints)**

Providing a descriptive name for a constraint is good practice so that you can identify it easily in the future. For example, constraint violation errors reference the constraint name, so an easy-to-understand name, indicating the table, column, and type of constraint is quite helpful.

```
DROP TABLE patient;

--Notice the use of the CONSTRAINT keyword when giving a name to a constraint.
CREATE TABLE Patient
(
    Patient_id NUMBER CONSTRAINT patient_patient_id_pk PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);

INSERT INTO patient VALUES (11,'John', 'Smith');
INSERT INTO patient VALUES (11,'Jill', 'Doe');

--Results are displayed below from this point forward.
--Notice the name of the constraint is no longer generated by the system.
SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';
```

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT	PATIENT_PATIENT_ID_PK P	

### **Example 3.7d (Primary key at the table level)**

Creating a primary key constraint at the table level.

```
DROP TABLE patient;

--Use the CONSTRAINT keyword when giving a name to a constraint.
--The constraint is created at the table level after all column definitions.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    CONSTRAINT patient_patient_id_pk PRIMARY KEY (patient_id)
);

DROP TABLE patient;

--In this case, Oracle will generate a constraint name.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    PRIMARY KEY (patient_id)
);
```

### **Example 3.7e (Composite primary key)**

This example creates a composite primary key, with the assumption that the combination of fname and lname are unique. Composite key cannot be created at the column level.

```
DROP TABLE patient;

--Invalid example: Cannot have two primary keys.
--The syntax for composite primary key requires a table level syntax.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20) PRIMARY KEY,
    Lname      VARCHAR2(20) PRIMARY KEY
);

--Here is the syntax for a composite primary key with a constraint name.
--Notice the constraint keyword is used to give a name to the constraint.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    CONSTRAINT patient_fname_lname_pk PRIMARY KEY (Fname, Lname)
);

Error starting at line 4 in command:
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20) PRIMARY KEY,
    Lname      VARCHAR2(20) PRIMARY KEY
)
Error at Command Line:8 Column:21
Error report:
SQL Error: ORA-02260: table can have only one primary key
02260. 00000 -  "table can have only one primary key"
*Cause:    Self-evident.
*Action:   Remove the extra primary key.
table PATIENT created.
```



### **Example 3.7f (Using alter table command)**

Creating a primary key using the alter command

```
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);
```

--Can create a primary key using alter table statement.

```
ALTER TABLE patient ADD PRIMARY KEY (patient_id);
```

--Add a constraint name.

```
ALTER TABLE patient ADD CONSTRAINT patient_patient_id_pk PRIMARY KEY  
(patient_id);
```



--Instead of adding, modify can be used as well.

```
ALTER TABLE patient MODIFY patient_id PRIMARY KEY;
```

```
ALTER TABLE patient MODIFY patient_id CHAR PRIMARY KEY;
```

```
ALTER TABLE patient MODIFY patient_id CONSTRAINT patient_patient_id_pk  
PRIMARY KEY;
```

### ***Example 3.7g (Using alter table for composite primary key)***

Creating a composite primary key using the alter command

```
DROP TABLE patient;
```

```
CREATE TABLE Patient  
(  
    Patient_id NUMBER,  
    Fname VARCHAR2(20),  
    Lname VARCHAR2(20)  
) ;
```

--Creating a composite primary key without a constraint name.

```
ALTER TABLE patient ADD PRIMARY KEY (fname, lname);
```

```
table PATIENT dropped.  
table PATIENT created.  
table PATIENT altered.
```

### ***✓ CHECK 3C***

- ✓ 1. What is the difference between a table and a column level constraint?
- ✓ 2. What happens if you don't give a name to a constraint?
- ✓ 3. What keyword do you use to give constraints a name? **CONSTRAINT**
- ✓ 4. What constraint do you have to create at the table level? **composite**
- ✓ 5. What constraint do you have to create at the column level? **column level constraint**

*"Don't walk in front of me, I may not follow.*

*Don't walk behind me, I may not lead.*

*Just walk beside me and be my friend. "*

## 3.8 Unique constraint

### **Example 3.8a (Unique key at the column level)**

Creating a unique key constraint at the column level



```

DROP TABLE patient;

--Use the UNIQUE keyword to create a candidate or unique key.
--A table can have many unique keys. Also, unlike a primary key which cannot-be NULL, a unique key
--column can have NULLs.
CREATE TABLE Patient
(
    Patient_id NUMBER UNIQUE,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);

INSERT INTO patient VALUES (11, 'John', 'Smith');

--INVALID: The patient_id (11) is a duplicate.
INSERT INTO patient VALUES (11, 'Jill', 'Doe');

--NULL is allowed
INSERT INTO patient VALUES (NULL, 'Jill', 'Doe');

--Since NULL means "void of data", it is not considered a duplicate and is therefore valid.
INSERT INTO patient VALUES (NULL, 'Jill', 'Doe');
INSERT INTO patient VALUES (NULL, 'Jill', 'Doe');

```

1 rows inserted.

Error starting at line 16 in command:

```
INSERT INTO patient VALUES (11, 'Jill', 'Doe')
```

Error report:

```
SQL Error: ORA-00001: unique constraint (IRAJ.SYS_C0013306) violated
00001. 00000 - "unique constraint (%s.%s) violated"
```

\*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.  
For Trusted Oracle configured in DBMS MAC mode, you may see  
this message if a duplicate entry exists at a different level.

\*Action: Either remove the unique restriction or do not insert the key.

1 rows inserted.

1 rows inserted.

1 rows inserted.

### **Example 3.8b (User\_constraints)**

Examining the system table user\_constraints

--The constraint name was generated by the Oracle since the constraint keyword was not used. The --constraint type for unique key is (U).

```
SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';
```

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT	SYS_C0013306	U

### **Example 3.8c (Giving a name to constraints)**

Providing a descriptive name for a constraint is a good practice so that you can identify it easily in the future. For example, constraint violation errors reference the constraint name, so an easy-to-understand name, indicating the table, column, and type of constraint is quite helpful.

```
DROP TABLE patient;
```

--The unique key is given a name using the constraint keyword.

```
CREATE TABLE Patient
(
    Patient_id NUMBER CONSTRAINT patient_patient_id_uk UNIQUE,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);
```

```
INSERT INTO patient VALUES (11,'John', 'Smith');
```

--Notice that the error message identifies the name of the constraint that is being violated.

```
INSERT INTO patient VALUES (11,'Jill', 'Doe');
```

--Notice the constraint type is (U). Also the table name must be in upper-case otherwise --it will not be found in the ORACLE system table.

```
SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';
```

table PATIENT dropped.

table PATIENT created.

1 rows inserted.

Error starting at line 14 in command:

```
INSERT INTO patient VALUES (11,'Jill', 'Doe')
```

Error report:

SQL Error: ORA-00001: unique constraint (IRAJ.PATIENT\_PATIENT\_ID\_UK) violated  
00001. 00000 - "unique constraint (%s.%s) violated"

\*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.  
For Trusted Oracle configured in DBMS MAC mode, you may see  
this message if a duplicate entry exists at a different level.

\*Action: Either remove the unique restriction or do not insert the key.

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
1 PATIENT	PATIENT_PATIENT_ID_UK	U

### **Example 3.8d (Unique key at the table level)**

Creating a unique key constraint at the table level

```
DROP TABLE patient;

--Unique key constraint can also be created at the table level just like a primary key.
--The constraint keyword is used to give a name to the unique key.

CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    CONSTRAINT patient_patient_id_uk UNIQUE (patient_id)
);

DROP TABLE patient;

--The system generates a name for the unique key.

CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    UNIQUE(patient_id)
);
```

?

### **Example 3.8e (Composite unique key)**

This example creates a composite unique key with the assumption that the combination of fname and lname must be unique. Composite key cannot be created at the column level. Unlike the primary key constraint, we can have many unique keys. Both of the following are correct; however, in one case each individual column must be unique from row to row whereas in the other, the combination must be unique from row to row

```
DROP TABLE patient;

--Unlike a primary key, a table can have multiple unique keys. This is not a composite, unique key. Fname and
--lname are unique by themselves. The system generates a name.

CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20) UNIQUE,
    Lname      VARCHAR2(20) UNIQUE
);
INSERT INTO patient VALUES (11,'John','Doe');

--The unique key is violated because of the duplicate lname (Doe)
INSERT INTO patient values(22,'jill','Doe');

DROP TABLE patient;
```

✓

```
--Composite unique key has the same basic syntax as a composite primary key.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    CONSTRAINT patient_fname_lname_uk UNIQUE (Fname, Lname)
);

INSERT INTO patient VALUES (11, 'John', 'Doe');

--Unique key is not violated because the combination, lname and fname, must together be unique.
INSERT INTO patient values(22,'jill','Doe');
```

```
table PATIENT dropped.
table PATIENT created.
1 rows inserted.

Error starting at line 13 in command:
INSERT INTO patient values(22,'jill','Doe')
-
Error report:
SQL Error: ORA-00001: unique constraint (IRAJ.SYS_C0013321) violated
00001. 00000 - "unique constraint (%s.%s) violated"
*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.
        For Trusted Oracle configured in DBMS MAC mode, you may see
        this message if a duplicate entry exists at a different level.
*Action: Either remove the unique restriction or do not insert the key.
table PATIENT dropped.
table PATIENT created.
1 rows inserted.
1 rows inserted.
```

### ***Example 3.8f (Creating unique with alter table command)***

```
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);
ALTER TABLE patient ADD CONSTRAINT patient_patient_id_uk UNIQUE (patient_id);

--Can use MODIFY for a unique constraint without the constraint keyword.
ALTER TABLE patient MODIFY patient_id UNIQUE;

--Use the constraint keyword to give a name to the constraint.
ALTER TABLE patient MODIFY patient_id CONSTRAINT patient_patient_id_uk
UNIQUE;
```

### **Example 3.8g (Using alter to create composite unique key)**

This example creates a composite unique key using the alter command.

```
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);

--Creates a composite unique key after the table has been created.
ALTER TABLE patient ADD UNIQUE (fname, lname);
```

## 3.9 Check constraint

### **Example 3.9a (Column v. table level check constraint)**

Column level versus table level without a constraint name

```
DROP TABLE patient;

--Use the Check keyword to create a check constraint. The name for this constraint will be
--generated by the system. This constraint is created at the column level because the comma
--appears after the column name, datatype and the actual check constraint.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Height     NUMBER CHECK (height>10),
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);

DROP TABLE patient;

--This check constraint is being created at the table level. Notice all the columns are defined fully and
--the check constraint appears at the very end. The check constraint does not have a name.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Height     NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    CHECK (height>10)
);

--Violates the check constraint because of the height. Must be greater than 10.
INSERT INTO patient VALUES (111, 9, 'John', 'Doe');

--This is okay
INSERT INTO patient VALUES (111, 19, 'John', 'Doe');
```

```

table PATIENT dropped.
table PATIENT created.
table PATIENT dropped.
table PATIENT created.
Error starting at line 29 in command:
INSERT INTO patient VALUES  (111,9,'John','Doe')
Error report:
SQL Error: ORA-02290: check constraint (IRAJ.SYS_C0013327) violated
02290. 00000 - "check constraint (%s.%s) violated"
*Cause: The values being inserted do not satisfy the named check

*Action: do not insert values that violate the constraint.
1 rows inserted.

```

### **Example 3.9b (User\_constraints)**

Examining the check constraint in the system table

--Constraint type is (c), which stands for check. Note, table name must be in UPPER CASE.

```

SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';

```

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT	SYS_C0013327	C

### **Example 3.9c (Constraint name for table versus column level)**

Column level versus table level with a constraint name

```

DROP TABLE patient;

--The check constraint, created at the column level, is given a name using the CONSTRAINT keyword.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Height      NUMBER CONSTRAINT patient_height_ck CHECK height>10,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);
DROP TABLE patient;

--This check constraint is created at the table level.
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20),
    CONSTRAINT patient_height_ck  CHECK (height>10)
);

```

```
--Check constraint is violated.
INSERT INTO patient VALUES  (111,9,'John','Doe');
INSERT INTO patient VALUES  (111,19,'John','Doe');

--Examine the constraint name.
SELECT table_name, constraint_name, constraint_type FROM
user constraints WHERE table_name='PATIENT';

table PATIENT dropped.
table PATIENT created.
table PATIENT dropped.
table PATIENT created.

Error starting at line 23 in command:
INSERT INTO patient VALUES  (111,9,'John','Doe')
Error report:
SQL Error: ORA-02290: check constraint (IRAJ.PATIENT_HEIGHT_CK) violated
02290. 00000 - "check constraint (%.%s) violated"
*Cause: The values being inserted do not satisfy the named check

*Action: do not insert values that violate the constraint.
1 rows inserted.



| TABLE_NAME | CONSTRAINT_NAME   | CONSTRAINT_TYPE |
|------------|-------------------|-----------------|
| PATIENT    | PATIENT_HEIGHT_CK | C               |


```

### ***Example 3.9d (Using alter table to create check constraint)***

Creating a check constraint using the alter statement with and without a constraint name

```
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

ALTER TABLE patient MODIFY height CONSTRAINT patient_height_ck
CHECK(height>10);

--ALTER TABLE patient MODIFY height CHECK(height>10);
```

--Can use ADD to add a check constraint with or without a name.

--ALTER TABLE patient ADD CHECK(height>10);

--ALTER TABLE patient ADD CONSTRAINT patient\_height\_ck CHECK(height>10);

table PATIENT created.

table PATIENT altered.

### 3.10 Not NULL constraint

#### *Example 3.10a (Not NULL constraint)*

```
DROP TABLE patient;

--Cannot create NULL constraint at the table level. In this example, the constraint doesn't have a name.
--This means that data has to be provided and it can be a duplicate, unlike primary and unique keys.
--There is no such thing as a composite NOT NULL.

CREATE TABLE Patient
(
    Patient_id NUMBER NOT NULL,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

--Notice the the constraint type for a not NULL constraint is (C).
SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';
```



--This gives an error message because of the NULL keyword. Patient id cannot be NULL  
 INSERT INTO patient VALUES (NULL,10,'John','Doe');

```
DROP TABLE patient;
```

--Like any other constraint, you can give this constraint a name.

```
CREATE TABLE Patient
(
    Patient_id NUMBER CONSTRAINT patient_patient_id_nn NOT NULL,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);
```

```
SELECT table_name, constraint_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT';
```

```
table PATIENT dropped.
table PATIENT created.
```

TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT	SYS_C0013335	C
 Error starting at line 20 in command: INSERT INTO patient VALUES (NULL,10,'John','Doe') Error at Command Line:20 Column:36 Error report: SQL Error: ORA-00911: invalid character 00911. 00000 - "invalid character" *Cause: identifiers may not start with any ASCII character other than letters and numbers. \$#_ are also allowed after the first character. Identifiers enclosed by doublequotes may contain any character other than a doublequote. Alternative quotes (q'#...#') cannot use spaces, tabs, or carriage returns as delimiters. For all other contexts, consult the SQL Language Reference Manual. *Action: table PATIENT dropped. table PATIENT created.		
TABLE_NAME	CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT	PATIENT_PATIENT_ID_NN	C

### Example 3.10b (Using modify command)

Notice that only modify can be used with the ALTER command when dealing with a not NULL constraint, which may or may not include the datatype. Can only use add if the column does not exist in the table.

```
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

--Invalid: cannot add a not NULL constraint.
ALTER TABLE patient ADD patient_id NOT NULL;

--Invalid: cannot add a not NULL constraint.
ALTER TABLE patient ADD patient_id NUMBER NOT NULL;

--Must use modify
ALTER TABLE patient MODIFY patient_id NOT NULL; ✓

--ALTER TABLE patient MODIFY patient_id NUMBER NOT NULL;
--Can also give the constraint a name
--ALTER TABLE patient MODIFY patient_id NUMBER CONSTRAINT patient_pat_id_nn NOT NULL;
```

```

table PATIENT dropped.
table PATIENT created.

Error starting at line 11 in command:
ALTER TABLE patient ADD patient_id NOT NULL
Error report:
SQL Error: ORA-02263: need to specify the datatype for this column
02263. 00000 - "need to specify the datatype for this column"
*Cause: The required datatype for the column is missing.
*Action: Specify the required datatype.

Error starting at line 13 in command:
ALTER TABLE patient ADD patient_id NUMBER NOT NULL
Error report:
SQL Error: ORA-01430: column being added already exists in table
01430. 00000 - "column being added already exists in table"
*Cause:
*Action:
table PATIENT altered.

```

### ***Example 3.10c (Between clause)***

More examples of check constraints. The Between clause is inclusive, which in this example means that both 10 and 100 are included.

```

DROP TABLE patient;

--An example of a different type of check constraint using the IN and BETWEEN clause.
CREATE TABLE Patient
(
    Patient_id      NUMBER,
    Height          NUMBER CONSTRAINT patient_height_ck CHECK
                    (height BETWEEN 5 and 10),
    gender          CHAR CHECK(gender in ('m','f')),
    Weight          NUMBER CHECK(weight BETWEEN 10 AND 100),
    Fname           VARCHAR2(20),
    Lname           VARCHAR2(20)
);
table PATIENT dropped.
table PATIENT created.

```

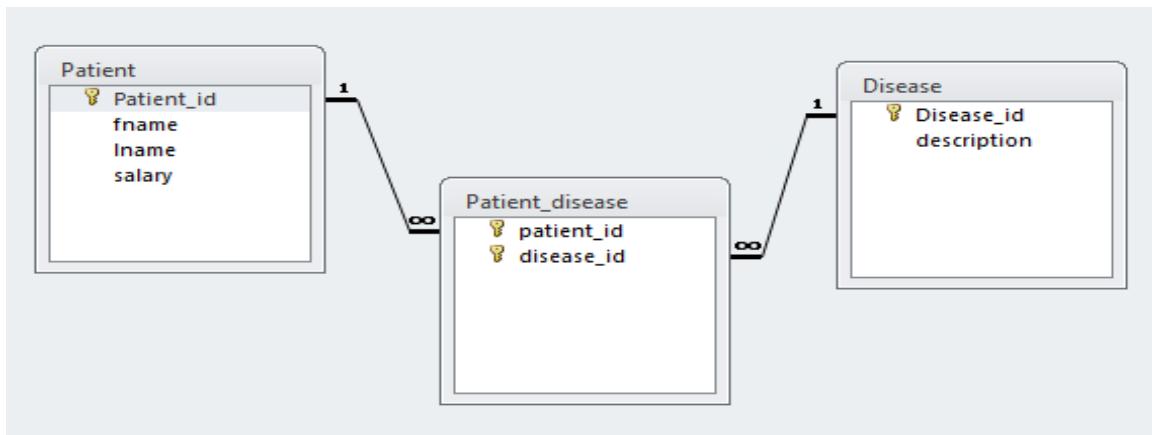


### **CHECK 3D**

1. Drop the PERSON table and re-create it with the primary key defined at the table level and the unique key at the column level (Person\_id (PK), SSN (UK), Lname, salary, DOB). Give your primary key constraint a name.
2. After the table has been created, create a NOT NULL constraint on DOB.
3. Insert data to verify that your constraints are correct.
4. Create a check constraint such that salary is always > 10000. Insert data to verify.

*"Perservance is not a long race; it is many short races one after another!"*

### 3.11 Foreign key constraint



#### *Example 3.11a (References keyword)*

The REFERENCES keyword refers to referential integrity, which means the user is referring to something that exists in another table. For example, the value entered in patient\_id column of the patient\_disease table references a value in the patient\_id column of the patient table. The REFERENCES keyword is used to identify the table and column that must already contain the data being entered. The column referenced must be a primary key column. The child table (patient\_disease) refers to the parent tables (patient, disease). The datatype of the columns that are establishing the relationship between the two tables must be the same.

```

DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid   NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);
  
```



--To create a foreign key, go to the many table and use the references keyword to point to the --column on the one table. The column on the one side should be a primary key. Use constraint --keyword to give the foreign key a name.

```
CREATE TABLE patient_disease
(
    Patient_id      NUMBER REFERENCES patient (patient_id),
    Disease_id      NUMBER CONSTRAINT dis_id_fk REFERENCES disease,
    PRIMARY KEY (patient_id, disease_id)
);
```

--The Foreign key constraint resides in the PATIENT\_DISEASE table and constraint type is (R). The --constraint name on the many side connects to the r\_constraint\_name on the one side, which is a --primary key.

```
SELECT table_name, constraint_name, r_constraint_name, constraint_type
FROM user_constraints WHERE table_name='PATIENT_DISEASE';
```

```
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
>Action:
table PATIENT dropped.
```

```
Error starting at line : 3 in command -
DROP TABLE disease
Error report -
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
>Action:
table PATIENT created.
table DISEASE created.
table PATIENT_DISEASE created.
```

TABLE_NAME	CONSTRAINT_NAME	R_CONSTRAINT_NAME	CONSTRAINT_TYPE
PATIENT_DISEASE	SYS_C0014314		P
PATIENT_DISEASE	SYS_C0014315	SYS_C0014312	R
PATIENT_DISEASE	DIS_ID_FK	SYS_C0014313	R

### ***Example 3.11b (Foreign keys and insert statement)***

Notice that the parent tables have to be populated first before any child records are inserted. The data in the child table must match data from the parent tables.

```
INSERT INTO patient VALUES (111, 90, 'John', 'Doe');
INSERT INTO disease VALUES (22, 'Malaria');
```

--(111) connects to the patient table whereas (22) connects to disease table.

```
INSERT INTO patient_disease VALUES (111, 22);
```

```
--INVALID: Connection to patient table (333) is problematic
INSERT INTO patient_disease VALUES (333,22);

--INVALID: Connection to disease table (33) is problematic
INSERT INTO patient_disease VALUES (111,33);

--INVALID: Primary key violation
INSERT INTO patient_disease VALUES (111,22);
1 rows inserted.
1 rows inserted.
1 rows inserted.

Error starting at line 6 in command:
INSERT INTO patient_disease VALUES (333,22)
Error report:
SQL Error: ORA-02291: integrity constraint (IRAJ.SYS_C0013365) violated - parent key not
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause: A foreign key value has no matching primary key value.
*Action: Delete the foreign key or add a matching primary key.

Error starting at line 8 in command:
INSERT INTO patient_disease VALUES (111,33)
Error report:
SQL Error: ORA-02291: integrity constraint (IRAJ.DIS_ID_FK) violated - parent key not fo
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause: A foreign key value has no matching primary key value.
*Action: Delete the foreign key or add a matching primary key.

Error starting at line 10 in command:
INSERT INTO patient_disease VALUES (111,22)
Error report:
SQL Error: ORA-00001: unique constraint (IRAJ.SYS_C0013364) violated
00001. 00000 - "unique constraint (%s.%s) violated"
```



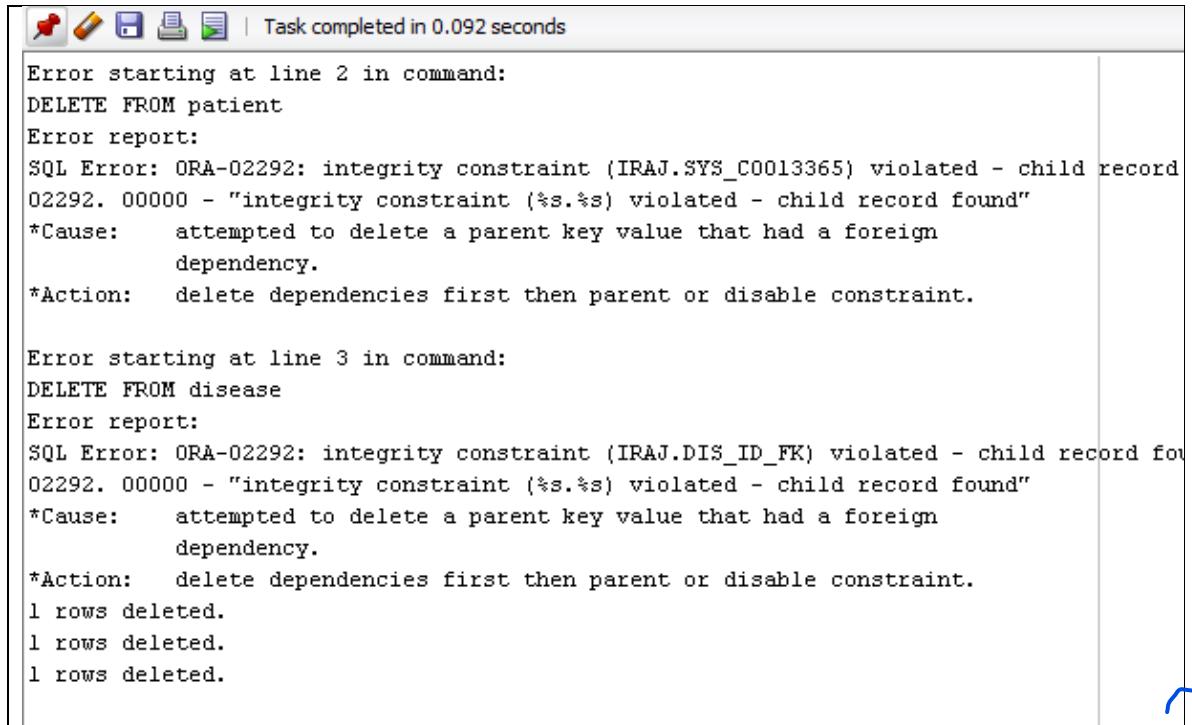
### ***Example 3.11c (Foreign keys and delete statement)***

Note that the child records must be deleted prior to deleting the parent records.

```
--Both these deletes are invalid because there is a corresponding record in the patient_disease table
--that is connected to both of these tables.
DELETE FROM patient;
DELETE FROM disease;

--Note the order of deletes. Must delete from patient_disease to get rid -of the association.
DELETE FROM patient_disease;

--The order of parent tables is not important.
DELETE FROM patient;
DELETE FROM disease;
```



```

Task completed in 0.092 seconds

Error starting at line 2 in command:
DELETE FROM patient
Error report:
SQL Error: ORA-02292: integrity constraint (IRAJ.SYS_C0013365) violated - child record
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"
*Cause:    attempted to delete a parent key value that had a foreign
            dependency.
*Action:   delete dependencies first then parent or disable constraint.

Error starting at line 3 in command:
DELETE FROM disease
Error report:
SQL Error: ORA-02292: integrity constraint (IRAJ.DIS_ID_FK) violated - child record fo
02292. 00000 - "integrity constraint (%s.%s) violated - child record found"
*Cause:    attempted to delete a parent key value that had a foreign
            dependency.
*Action:   delete dependencies first then parent or disable constraint.
1 rows deleted.
1 rows deleted.
1 rows deleted.

```

### ***Example 3.11d (Truncate v. delete)***

(TRUNCATE vs DELETE) Note: you cannot use TRUNCATE TABLE on a table referenced by a FOREIGN KEY constraint; instead, use DELETE statement without a WHERE clause. You have to get rid of the foreign key constraint(s) if you want to truncate the parent table.

```

DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid    NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);

CREATE TABLE patient_disease
(
    Patient_id    NUMBER REFERENCES patient (patient_id),
    Disease_id    NUMBER CONSTRAINT dis_id_fk REFERENCES disease,
    PRIMARY KEY (patient_id, disease_id)
);

```

```

INSERT INTO patient VALUES (111, 90, 'John', 'Doe');
INSERT INTO disease VALUES (22, 'Malaria');
INSERT INTO patient_disease VALUES (111, 22);

--INVALID. Must truncate child table first.
TRUNCATE TABLE patient;

--VALID. Can only truncate child table
TRUNCATE TABLE patient_disease;

--INVALID. Unlike the delete command, parent tables cannot be truncated unless the foreign keys
--are dropped or disabled. Use delete instead.
TRUNCATE TABLE patient;
TRUNCATE TABLE disease;
DELETE FROM patient;
DELETE FROM disease;

```

```

table PATIENT_DISEASE dropped.
table PATIENT dropped.
table DISEASE dropped.
table PATIENT created.
table DISEASE created.
table PATIENT_DISEASE created.
1 rows inserted.
1 rows inserted.
1 rows inserted.
Error starting at line : 30 in command -
TRUNCATE TABLE patient
Error report -
SQL Error: ORA-02266: unique/primary keys in table referenced by enabled foreign keys
02266. 00000 - "unique/primary keys in table referenced by enabled foreign keys"
*Cause: An attempt was made to truncate a table with unique or
    primary keys referenced by foreign keys enabled in another table.
    Other operations not allowed are dropping/truncating a partition of a
    partitioned table or an ALTER TABLE EXCHANGE PARTITION.
*Action: Before performing the above operations the table, disable the
    foreign key constraints in other tables. You can see what
    constraints are referencing a table by issuing the following
    command:
    SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";
table PATIENT_DISEASE truncated.
Error starting at line : 37 in command -
TRUNCATE TABLE patient
Error report -
SQL Error: ORA-02266: unique/primary keys in table referenced by enabled foreign keys
02266. 00000 - "unique/primary keys in table referenced by enabled foreign keys"
*Cause: An attempt was made to truncate a table with unique or
    primary keys referenced by foreign keys enabled in another table.

```

Other operations not allowed are dropping/truncating a partition of a partitioned table or an ALTER TABLE EXCHANGE PARTITION.

\*Action: Before performing the above operations the table, disable the foreign key constraints in other tables. You can see what constraints are referencing a table by issuing the following command:

```
SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";
```

Error starting at line : 38 in command -  
TRUNCATE TABLE disease  
Error report -  
SQL Error: ORA-02266: unique/primary keys in table referenced by enabled foreign keys  
02266. 00000 - "unique/primary keys in table referenced by enabled foreign keys"  
\*Cause: An attempt was made to truncate a table with unique or primary keys referenced by foreign keys enabled in another table.  
Other operations not allowed are dropping/truncating a partition of a partitioned table or an ALTER TABLE EXCHANGE PARTITION.

\*Action: Before performing the above operations the table, disable the foreign key constraints in other tables. You can see what constraints are referencing a table by issuing the following command:

```
SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";
```

1 rows deleted.  
1 rows deleted.

### ***Example 3.11e (Delete cascade)***

Delete cascade option allows for a record in the parent table to be deleted even if it is being referenced in the child table. Using this option will remove all the corresponding records in the child table in order to maintain referential integrity.

```
DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid   NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);
```



```

CREATE TABLE patient_disease
(
    Patient_id      NUMBER REFERENCES patient ON DELETE CASCADE,
    Disease_id      NUMBER REFERENCES disease ON DELETE CASCADE,
    PRIMARY KEY (patient_id, disease_id)
);

--Notice that first the parent tables are populated and then the child table. Also notice how the
--foreign keys match up with the primary keys.
INSERT INTO patient VALUES (111,90,'John','Doe');
INSERT INTO disease VALUES (22,'Malaria');
INSERT INTO patient_disease VALUES (111,22);

/* This would usually fail because there are child records that are connected to this table and so the
children record have to be deleted first. However, with cascade delete, when a record is deleted
from the parent table, all the related records in the child table are automatically deleted as well. */
DELETE FROM disease;
SELECT * FROM disease;
SELECT * FROM patient_disease;
SELECT * FROM patient;

```

---

table PATIENT\_DISEASE created.

1 rows inserted.  
1 rows inserted.  
1 rows inserted.  
1 rows deleted.  
no rows selected

no rows selected

PATIENT_ID	HEIGHT	FNAME	LNAME
111	90	John	Doe

### *Example 3.11f (Foreign keys and dropping tables)*

When getting rid of the tables, the child table has to be deleted prior to the parents.

--The child table must be dropped first and then the parent tables can be dropped in any order.

--This will cause an error.

```

DROP TABLE patient;
DROP TABLE disease;
DROP TABLE patient_disease;

```

```
Error starting at line 1 in command:
DROP TABLE patient
Error report:
SQL Error: ORA-02449: unique/primary keys in table referenced by foreign keys
02449. 00000 - "unique/primary keys in table referenced by foreign keys"
*Cause: An attempt was made to drop a table with unique or
        primary keys referenced by foreign keys in another table.
*Action: Before performing the above operations the table, drop the
        foreign key constraints in other tables. You can see what
        constraints are referencing a table by issuing the following
        command:
        SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";

Error starting at line 2 in command:
DROP TABLE disease
Error report:
SQL Error: ORA-02449: unique/primary keys in table referenced by foreign keys
02449. 00000 - "unique/primary keys in table referenced by foreign keys"
*Cause: An attempt was made to drop a table with unique or
        primary keys referenced by foreign keys in another table.
*Action: Before performing the above operations the table, drop the
        foreign key constraints in other tables. You can see what
        constraints are referencing a table by issuing the following
        command:
        SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";
table PATIENT_DISEASE dropped.
```

### ***Example 3.11g (Foreign key at the table level)***

Notice the syntax FOREIGN KEY is used to refer to the column in the child table and the references syntax is used to refer to the column in the parent table.

```
DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid   NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);
```



--Foreign key is being created at the table level. This requires the additional -(FOREIGN KEY) keyword.  
 --Since all the columns have been defined in the child table already, we need to identify the  
 --column that will be the foreign key in the child table and then we use the REFERENCES keyword  
 --as in the previous syntax. Once again the CONSTRAINT keyword is needed to give it a name.

```
CREATE TABLE patient_disease
(
    Patient_id      NUMBER,
    Disease_id      NUMBER,
    PRIMARY KEY (patient_id, disease_id),
    FOREIGN KEY (patient_id) REFERENCES patient ON DELETE CASCADE,
    CONSTRAINT patient_disease_disease_id_fk FOREIGN KEY (disease_id)
    REFERENCES disease
);
```

### ***Example 3.11h (Using alter table to create foreign keys)***

Notice the syntax FOREIGN KEY is used to refer to the column in the child table and the references syntax is used to refer to the column in the parent table.

```
DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;

CREATE TABLE Patient
(
    Patient_id  NUMBER PRIMARY KEY,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid   NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);

CREATE TABLE patient_disease
(
    Patient_id      NUMBER,
    Disease_id      NUMBER,
    PRIMARY KEY      (patient_id, disease_id)
);

--Alter table is used to identify a foreign key
ALTER TABLE patient_disease ADD FOREIGN KEY(patient_id) REFERENCES
patient;

ALTER TABLE patient_disease ADD CONSTRAINT
patient_disease_disease_id_fk FOREIGN KEY(disease_id) REFERENCES
disease;
```



```

Error starting at line : 1 in command -
DROP TABLE patient_disease
Error report -
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
table PATIENT dropped.
Error starting at line : 3 in command -
DROP TABLE disease
Error report -
SQL Error: ORA-00942: table or view does not exist
00942. 00000 - "table or view does not exist"
*Cause:
*Action:
table PATIENT created.
table DISEASE created.
table PATIENT_DISEASE created.
table PATIENT_DISEASE altered.
table PATIENT_DISEASE altered.

```

### ✓ *CHECK 3E*

- ?
1. Given the tables Person and Personality, create the foreign key relationship:
    - a. At the table level (With a constraint name)
    - b. At the column level (Without a constraint name)
    - c. After the tables have been created
  2. Insert to verify.
  3. Drop foreign key constraint and re-create with DELETE CASCADE option.
  4. Insert and delete to verify constraint.
  5. Truncate both tables.

*"My father always used to say that when you die, if you've got five real friends, then you've had a great life. "*

### 3.12 Disabling/Enabling/Dropping constraints

To DISABLE a constraint, you issue an ALTER TABLE command and change the constraint's status to DISABLE. Later, you can reissue the ALTER TABLE command and change the constraint's status back to ENABLE. After the constraint is enabled, data added or modified is again checked by the constraint.

#### **Example 3.12a (Disable/Enable constraints)**

```
DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;
CREATE TABLE Patient
(
    Patient_id NUMBER CONSTRAINT patient_patient_id_pk PRIMARY KEY,
    Height      NUMBER CONSTRAINT patient_height_ck CHECK (height>4),
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20),
    CONSTRAINT patient_fname_lname_uk UNIQUE(fname, lname)
);

--By disabling the constraint, records can be inserted which would have created a problem with the
--constraint. A disabled constraint can later be enabled. The primary key can be enabled or
--disabled using just the PRIMARY KEY syntax with or without the columns.
ALTER TABLE patient DISABLE PRIMARY KEY;
```

--The status column conveys if the constraint is enabled or disabled  

```
SELECT constraint_name, constraint_type, status FROM user_constraints
WHERE table_name='PATIENT';
```

table PATIENT created.  
 table PATIENT altered.

CONSTRAINT_NAME	CONSTRAINT_TYPE	STATUS
PATIENT_HEIGHT_CK	C	ENABLED
PATIENT_PATIENT_ID_PK	P	DISABLED
PATIENT_FNAME_LNAME_UK	U	ENABLED

```
ALTER TABLE patient ENABLE PRIMARY KEY;
ALTER TABLE patient DISABLE CONSTRAINT patient_patient_id_pk;
ALTER TABLE patient DISABLE CONSTRAINT patient_fname_lname_uk;
ALTER TABLE patient ENABLE CONSTRAINT patient_fname_lname_uk;
ALTER TABLE patient DISABLE UNIQUE (fname, lname);
```

table PATIENT altered.  
 table PATIENT altered.  
 table PATIENT altered.  
 table PATIENT altered.  
 table PATIENT altered.

If you create a constraint, you can delete it from the table with the **DROP ( constraintname )** command. In addition, if you need to change or modify a constraint, your only option is to delete the constraint and then create a new one. You use the ALTER TABLE command to drop an existing constraint from a table. If this ALTER TABLE command is executed successfully, the constraint no longer exists, and any value is accepted as input to the column.

- The DROP clause varies depending on the type of constraint being deleted. If the DROP clause references the PRIMARY KEY constraint for the table, using the keywords PRIMARY KEY is enough because only one such clause is allowed for each table in the database. •
- To delete a UNIQUE constraint, only the column name affected by the constraint is required because a column is referenced by only one UNIQUE constraint.
- Any other type of constraint must be referenced by the constraint's actual name—regardless of whether the constraint name is assigned by a user or the Oracle server.

### **Example 3.12b (Drop constraints)**

```
DROP TABLE patient;

CREATE TABLE Patient
(
    Patient_id  NUMBER CONSTRAINT patient_patient_id_pk PRIMARY KEY,
    Height      NUMBER CONSTRAINT patient_height_ck CHECK (height>4),
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20),
    CONSTRAINT  patient_fname_lname_uk UNIQUE(fname, lname)
);
```

--Constraints can also be dropped. Once dropped, there is no way to get them back. They have to be --recreated, which is different from disabled constraints.

```
ALTER TABLE patient DROP CONSTRAINT patient_height_ck;
ALTER TABLE patient DROP PRIMARY KEY;
ALTER TABLE patient DROP UNIQUE (fname, lname);
```

```
table PATIENT dropped.
table PATIENT created.
table PATIENT altered.
table PATIENT altered.
table PATIENT altered.
```

### **Example 3.12c (Primary key with a cascade option)**

If this ALTER TABLE command is executed successfully, the constraint no longer exists, and any value is accepted as input to the column. If needed, the associated FOREIGN KEY can be deleted along with the PRIMARY KEY deletion by using the CASCADE option.

```

DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid   NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);

CREATE TABLE patient_disease
(
    Patient_id   NUMBER REFERENCES patient (patient_id),
    Disease_id   NUMBER REFERENCES disease,
    PRIMARY KEY (patient_id, disease_id)
);

--Notice there are two foreign keys
SELECT constraint_name, table_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT_DISEASE';

--This is going to cause a problem because a foreign key is attached to it.
ALTER TABLE patient DROP PRIMARY KEY;

--This will automatically get rid of the foreign key constraint that is attached to this primary key.
ALTER TABLE patient DROP PRIMARY KEY CASCADE;

--Notice, there is only one foreign key.
SELECT constraint_name, table_name, constraint_type FROM
user_constraints WHERE table_name='PATIENT_DISEASE';

```

```

table PATIENT_DISEASE dropped.
table PATIENT dropped.
table DISEASE dropped.
table PATIENT created.
table DISEASE created.
table PATIENT_DISEASE created.

CONSTRAINT_NAME          TABLE_NAME           CONSTRAINT_TYPE
-----
SYS_C0013403              PATIENT_DISEASE        P
SYS_C0013404              PATIENT_DISEASE        R
SYS_C0013405              PATIENT_DISEASE        R

Error starting at line 24 in command:
ALTER TABLE patient DROP PRIMARY KEY
Error report:
SQL Error: ORA-02273: this unique/primary key is referenced by some foreign keys
02273. 00000 - "this unique/primary key is referenced by some foreign keys"
*Cause: Self-evident.
>Action: Remove all references to the key before the key is to be dropped.
table PATIENT altered.

CONSTRAINT_NAME          TABLE_NAME           CONSTRAINT_TYPE
-----
SYS_C0013403              PATIENT_DISEASE        P
SYS_C0013405              PATIENT_DISEASE        R

```

### ✓ CHECK 3F

1. Drop the primary key constraint in the Person table.
2. Disable the unique key in the Person table.
3. Insert a record to verify.

*"The real measure of our wealth is how much we'd be worth if we lost all our money"*

## 3.13 Indexes

A database index is much like the index at the end of a book. As rows of data are inserted, they're physically added to the table in no particular order. As rows are deleted, the space can be reused by new rows. Therefore, if a search condition such as " WHERE zip = 90404" is included in a SELECT statement, a full table scan is performed. In a full table scan, each row of the table is read, and the zip value is checked to determine whether it satisfies the condition. This issue can be addressed by applying indexes to table columns. The B- tree ( balanced- tree) index is the most common index used in Oracle.

### **Example 3.13a (Indexes)**

An index can be created implicitly or explicitly. Oracle creates an index automatically when a PRIMARY KEY or UNIQUE constraint is created for a column.

```

DROP TABLE patient;
--Indexes will be created automatically with primary keys and unique keys.
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Address     VARCHAR2(30),
    Height      NUMBER,
    Fname       VARCHAR2(20),
    Lname       VARCHAR2(20),
    UNIQUE (fname, lname)
);

--In addition, indexes can be created manually.
CREATE INDEX patient_address_idx ON patient (address);

--The primary key and unique key information will be in the user_constraints table.
SELECT constraint_name, constraint_type FROM user_constraints WHERE
table_name='PATIENT';

--The index that was created using create index command along with the primary key and unique
--key indexes will be in this table. The name of the indexes for the primary key and unique will be
--the same as the constraint names that appear in the user_constraints table
SELECT index_name FROM user_indexes WHERE table_name='PATIENT';



|                                    |                 |
|------------------------------------|-----------------|
| table PATIENT dropped.             |                 |
| table PATIENT created.             |                 |
| index PATIENT_ADDRESS_IDX created. |                 |
| CONSTRAINT_NAME                    | CONSTRAINT_TYPE |
| SYS_C0013345                       | P               |
| SYS_C0013346                       | U               |
| INDEX_NAME                         |                 |
| SYS_C0013345                       |                 |
| SYS_C0013346                       |                 |
| PATIENT_ADDRESS_IDX                |                 |


```

### ***Example 3.13b (Alter index command)***

The only modification you can perform on an existing index is a name change. If you need to change the name of an index, use the ALTER INDEX command.

--The name of the index is altered.

```
ALTER INDEX patient_address_idx RENAME TO pat_add_idx;
```

### ***Example 3.13c (Dropping an index)***

DO NOT USE the ALTER command for dropping indexes. It does not work.

--To drop an index, use DROP INDEX. Many make the common mistake of using the ALTER TABLE --command but that would be wrong. Also, when a table is dropped, all the associated constraints --and indexes are dropped as well.

```
DROP INDEX patient_address_idx;
```



### **CHECK 3G**

- 1) When are indexes created automatically?
- 2) Create an index on the Person table based on lname and salary.
- 3) Drop the index.

*“If we could read the secret history of our enemies, we should find in each person's life sorrow and suffering enough to disarm all hostility. ”*

## Summary example

```

DROP TABLE patient_disease;
DROP TABLE disease;
DROP TABLE patient;
DROP TABLE sickperson;

--Notice that primary key is two words.
--Use the constraint keyword to name a constraint.
--Use the word unique, not unique key to create a unique constraint.
--In this example, all constraints with the exception of unique are at the column level
--NOT NULL, NULL and DEFAULT must be defined at the column level
--Composite keys must be defined at the table level.
--Can have many unique keys but only one primary key.

CREATE TABLE patient
(
    patient_id      NUMBER PRIMARY KEY,
    height          NUMBER CHECK (height BETWEEN 10 AND 20),
    fname           VARCHAR2(20) CONSTRAINT patient_fname_nn NOT NULL,
    lname           VARCHAR2(20),
    salary          NUMBER      DEFAULT 10000,
    CONSTRAINT patient_uk UNIQUE(lname) ,
    UNIQUE(height,salary)
);

CREATE TABLE disease
(
    diseaseid       NUMBER PRIMARY KEY,
    disease_desc    VARCHAR2(20)
);

--The parent tables can be created in any order.
--Foreign keys can be created at the table level or column level. When creating foreign keys at
--table level, use FOREIGN KEY syntax.
--Composite keys have to be created at the table level.
--ON DELETE CASCADE allows the deletion of parent records which will automatically
--trigger the deletion of the associated child records.

CREATE TABLE patient_disease
(
    patient_id      NUMBER REFERENCES patient ON DELETE CASCADE,
    disease_id      NUMBER,
    CONSTRAINT patient_disease_dis_id_fk FOREIGN KEY(disease_id)
        REFERENCES disease (diseaseid),
        PRIMARY KEY (patient_id, disease_id)
);

--With the foreign key in place, parent records must be inserted before the child records;
--otherwise there will be an integrity error.

INSERT INTO disease VALUES (22,'malaria');
INSERT INTO patient VALUES (111,15,'john','james');
INSERT INTO patient_disease VALUES (111,22);
SELECT * FROM patient;

```

--Additional columns can be added.

```
ALTER TABLE patient ADD DOB DATE UNIQUE;
```

--Columns can be renamed.

```
ALTER TABLE patient RENAME COLUMN DOB TO dateofbirth;
```

--A constraint that is disabled can be also enabled (ENABLE). In this case, because it is attached --to a foreign key, it cannot be disabled. The foreign key will have to be removed first.

```
ALTER TABLE patient DISABLE PRIMARY KEY;
```

--Drops the column height and all its data.

```
ALTER TABLE patient DROP (height);
```

--Drops the constraint.

```
ALTER TABLE patient DROP CONSTRAINT patient_fname_nn;
```

--Constraint information for the patient table.

```
SELECT table_name, constraint_name, constraint_type FROM user_constraints WHERE table_name='PATIENT';
```

--This can only be done with the cascade delete option. Without this option the children --records will have to be deleted first.

```
DELETE FROM patient;
```

--All references to the patient table are changed to sickperson .

```
RENAME patient TO sickperson;
```

```
DESC sickperson;
```

--Deletes all the records from the table sickperson (can get it back through rollback).

```
DELETE FROM sickperson;
```

--Deletes all th records from the table sickperson but you cannot get it back.

```
TRUNCATE TABLE sickperson;
```

--Create an index on fname. An index is automatically created with unique and primary keys.

```
CREATE INDEX myname ON sickperson (fname);
```

--Drop the index.

```
DROP INDEX myname;
```

--Some basic constraint information for tables stored in the system table user\_constraints.

```
SELECT table_name, constraint_name, constraint_type, search_condition FROM user_constraints WHERE table_name='NAME OF YOUR TABLE IN UPPER CASE';
```

--The table, the constraints and all the records in the table are eliminated.

--If there is a parent table, then the child table must first be dropped.

```
DROP TABLE myname;
```



*At nine months, he contracted pneumonia, and his unmarried 19-year-old mother gave him to her aunt and uncle in Chicago to raise. Lawrence was raised in a two-bedroom apartment on the city's South Side. Until he was twelve years old he did not know that he was adopted. His adoptive father had lost his real estate business in the Great Depression and made a modest living as an auditor for the public housing authority. As a boy, Larry Ellison showed an independent, rebellious streak and often clashed with his adoptive father.*

## Chapter 4 (Data Manipulation (inserts))

```
INSERT INTO tablename [(columnname, ...)]  
VALUES (datavalue, ...);
```

*"The reason Las Vegas is so crowded is that no one has the plane fare to leave."*

The keywords INSERT INTO are followed by the name of the table into which rows will be entered. The table name is followed by a list of the columns containing the data. The VALUES clause identifies the data values to be inserted in the table. You list the data values in parentheses after the VALUES keyword.

If the data entered in the VALUES clause contains a value for every column and is in the same order as columns in the table. Column names can be omitted in the INSERT INTO clause.

If you enter data for only some columns, or if columns are listed in a different order than they're listed in the table, the column names must be provided in the INSERT INTO clause in the same order as they're given in the VALUES clause. You must list the column names inside parentheses after the table name in the INSERT INTO clause.

If more than one column is listed, column names must be separated by commas. If more than one data value is entered, the values must be separated by commas.

You must use single quotes to enclose non-numeric data inserted in a column.

You can take one of the following approaches to indicate that a column contains a NULL value (indicating an absence of data):

- List all columns except the one that will not have any value in the INSERT INTO clause, and provide data for the listed columns in the VALUES clause.
- In the VALUES clause, substitute two single quotes in the position that should contain the account manager's assigned region. Oracle interprets the two single quotes to mean that a NULL value should be stored in the column. Be sure you don't add a blank space between these single quotes, however. Doing so adds a blank space value rather than a NULL value.
- In the VALUES clause, include the keyword NULL in the position where the region should be listed. As long as the keyword NULL isn't enclosed in single quotes, Oracle leaves the column blank. However, if the keyword is mistakenly entered as 'NULL', Oracle tries to store the word "NULL" in the column.

## 4.1 Inserting text

### ***Example 4.1a (Inserting textual data)***

To determine the number of characters in a string, you can use the LENGTH function.

```

DROP TABLE char_test;

--VARCHAR does not pad with spaces whereas CHAR is fixed and is padded
--with spaces if enough characters are not provided. Keep in mind that space is data in that
--"HI " is different from " HI".
CREATE TABLE char_test
(
    cola VARCHAR(3),
    colb CHAR,
    colc CHAR(2)
);

DESC char_test;

--Have to have a value for every column in the table. The order of the values must match
--the order in which the columns appeared in the create table statement. Can use the
-- DESC TABLE _name statement to find out what the proper order is.
INSERT INTO char_test VALUES ('aaa', 'b', 'cc');
SELECT * FROM char_test;

-- Can identify the columns, which means that the order can be different.
INSERT INTO char_test (cola,colb,colc) VALUES
('aaa', 'b', 'cc');
SELECT * FROM char_test;

-- Can identify the columns in different order
INSERT INTO char_test (colb,colc,cola) VALUES
('b', 'cc', 'aaa');

-- Can identify fewer columns as long as a constraint is not violated
INSERT INTO char_test (colb,colc) VALUES ('b', 'cc');

SELECT * FROM char_test;

```

```

table CHAR_TEST dropped.
table CHAR_TEST created.
DESC char_test
Name Null Type
-----
COLA      VARCHAR2(3)
COLB      CHAR(1)
COLC      CHAR(2)

1 rows inserted.
COLA COLB COLC
-----
aaa  b    cc

1 rows inserted.
COLA COLB COLC
-----
aaa  b    cc
aaa  b    cc

1 rows inserted.
1 rows inserted.
COLA COLB COLC
-----
aaa  b    cc
aaa  b    cc
aaa  b    cc
      b   cc

```

```

-- Invalid: cola can be equal or less than three characters but not more.
INSERT INTO char_test (cola) VALUES ('aaaa');

-- cola VARCHAR(3) can be less and will not pad it with spaces
INSERT INTO char_test (cola) VALUES ('aa');
SELECT cola, LENGTH(cola), colb, colc FROM char_test;

--Can be NULL. Use the word NULL or single quotes with no spaces in between
INSERT INTO char_test VALUES (NULL, '', 'c');
SELECT * FROM char_test;

-- colc char(2) blank padded
INSERT INTO char_test (colc) VALUES ('c');

--To confirm the space, we can use the length function and feed into it colc which
--will tell us how many characters make up the data.
SELECT colc, LENGTH(colc) FROM char_test;

```

```

INSERT INTO char_test (cola) VALUES ('aaaa')
Error report -
SQL Error: ORA-12899: value too large for column "IRAJ"."CHAR_TEST"."COLA" (actual: 4, maximum: 3)
12899. 00000 - "value too large for column $s (actual: $s, maximum: $s)"
*Cause: An attempt was made to insert or update a column with a value
which is too wide for the width of the destination column.
The name of the column is given, along with the actual width
of the value, and the maximum allowed width of the column.
Note that widths are reported in characters if character length
semantics are in effect for the column, otherwise widths are
reported in bytes.
*Action: Examine the SQL statement for correctness. Check source
and destination column data types.
Either make the destination column wider, or use a subset
of the source column (i.e. use substring).|  

1 rows inserted.  

COLA LENGTH(COLA) COLB COLC  

---- -----  

aaa      3 b    cc  

aaa      3 b    cc  

aaa      3 b    cc  

          b    cc  

aa       2  
  

1 rows inserted.  

COLA COLB COLC  

---- -----  

aaa  b    cc  

aaa  b    cc  

aaa  b    cc  

          b    cc  

aa  

          c  
  

DELETE FROM char_test;  
  

-- cola can be equal or less than three characters but not more. Invalid
INSERT INTO char_test (cola) VALUES ('aaaa');  
  

-- cola VARCHAR(3) can be less and will not pad it with spaces
INSERT INTO char_test (cola) VALUES ('aa');
SELECT cola, LENGTH(cola), colb, colc FROM char_test;  
  

--can be NULL. Use the word NULL or single quotes with no spaces in between
INSERT INTO char_test VALUES (NULL, '', 'c');
SELECT * FROM char_test;  
  

-- colc char(2) blank padded
INSERT INTO char_test (colc) VALUES ('c');  
  

--To confirm the space, we can use the length function and feed into it.
--colc which will tell us how many characters make up the data.
SELECT colc, LENGTH(colc) FROM char_test;

```

```

Error starting at line : 3 in command -
INSERT INTO char_test (cola) VALUES ('aaaa')
Error report -
SQL Error: ORA-12899: value too large for column "IRAJ"."CHAR_TEST"."COLA" (actual: 4, maximum: 3)
12899. 00000 -  "value too large for column $s (actual: $s, maximum: $s)"
*Cause: An attempt was made to insert or update a column with a value
which is too wide for the width of the destination column.
The name of the column is given, along with the actual width
of the value, and the maximum allowed width of the column.
Note that widths are reported in characters if character length
semantics are in effect for the column, otherwise widths are
reported in bytes.
*Action: Examine the SQL statement for correctness. Check source
and destination column data types.
Either make the destination column wider, or use a subset
of the source column (i.e. use substring).

1 rows inserted.
COLA LENGTH(COLA) COLB COLC
-----
aa          2

1 rows inserted.
COLA COLB COLC
-----
aa
      c

1 rows inserted.
COLC LENGTH(COLC)
-----
c          2
c          2

```

## 4.2 Inserting numbers

### *Example 4.2a (Inserting numerical data)*

Single should not be used for numeric values in the INSERT INTO statement, such as the salary and commission values. Also, no formatting characters, such as a comma or dollar sign, should be included in values because they raise an error. A NUMBER column stores only digits. You add formatting characters when querying numeric values.

```

DROP TABLE number_test;
--The NUMBER data type can store real or whole numbers
--NUMBER(5) will store whole numbers that don't exceed five digits. It will round real numbers
--NUMBER(5,3): Can store a real number. Two digits can be a whole number and three digits
--can be after the decimal point. If there are more digits after the decimal point then it
--rounds. If the whole number part is more than two digits, then it will give you an error
CREATE TABLE number_test
(
    col0 NUMBER,
    col1 NUMBER(5),
    col2 NUMBER(5,3)
);

-- With number you can store integers or floats.
INSERT INTO number_test VALUES (12.2345, 2345, 23.253);
SELECT * FROM number_test;

-- Can put in NULLs using the NULL keyword or single quotes with no spaces
INSERT INTO number_test VALUES (NULL, '', NULL);
SELECT * FROM number_test;

-- Can be less.
INSERT INTO number_test VALUES (123,23,2);
SELECT * FROM number_test;

INSERT INTO number_test (col0) VALUES (1234556);
INSERT INTO number_test (col0) VALUES (12.34566);
SELECT * FROM number_test;

-- col1 number(5). More than five digits is invalid.
INSERT INTO number_test (col1) VALUES (123456);
--Will round to whole number
INSERT INTO number_test (col1) VALUES (12.3456);
SELECT * FROM number_test;
-- Rounds up. Only looks at the first number after the decimal point for rounding whole numbers.
INSERT INTO number_test (col1) VALUES (12.5);
INSERT INTO number_test (col1) VALUES (12.45);
SELECT * FROM number_test;
-- col2 number(5,3). Will round the significant digits to 3 digits.
INSERT INTO number_test (col2) VALUES (12.2545);
SELECT * FROM number_test;

-- col2 number(5,3) is invalid. Can only have two digits to the left of decimal point.
INSERT INTO number_test (col2) VALUES (1234.2);

-- col2 number(5,3) invalid. Can only have two digits to the left of decimal point.
INSERT INTO number_test (col2) VALUES (123);

-- col2 number(5,3) will round the significant digits to 23.235
INSERT INTO number_test (col2) VALUES (23.23456);
SELECT * FROM number_test;

-- INVALID: Goes beyond the two digits for the whole number part.
INSERT INTO number_test (col2) VALUES (99.9999);

```

```
SQL> CREATE TABLE number_test
2 (
3   col0 NUMBER,
4   col1 NUMBER(5),
5   col2 NUMBER(5,3)
6 );
Table created.

SQL>
SQL> -- With number you can store integers or floats.
SQL> INSERT INTO number_test VALUES (12.2345, 2345, 23.253);
1 row created.

SQL> SELECT * FROM number_test;
      COL0        COL1        COL2
-----  -----
  12.2345      2345     23.253

SQL>
SQL> -- Can put in nulls using the null keyword or single quotes with no spaces
SQL> INSERT INTO number_test VALUES (null,'',null);
1 row created.

SQL> SELECT * FROM number_test;
      COL0        COL1        COL2
-----  -----
  12.2345      2345     23.253

SQL>
SQL> -- Can be less.
SQL> INSERT INTO number_test VALUES (123,23,2);
1 row created.
```

```
SQL> SELECT * FROM number_test;
      COL0      COL1      COL2
----- -----
  12.2345    2345    23.253
      123       23        2

SQL>
SQL> INSERT INTO number_test (col0) VALUES (1234556);
1 row created.

SQL> INSERT INTO number_test (col0) VALUES (12.34566);
1 row created.

SQL> SELECT * FROM number_test;
      COL0      COL1      COL2
----- -----
  12.2345    2345    23.253
      123       23        2
      1234556
  12.34566

SQL>
SQL> -- col1 number(5). More than five is invalid
SQL> INSERT INTO number_test (col1) VALUES (123456);
INSERT INTO number_test (col1) VALUES (123456)
*
ERROR at line 1:
ORA-01438: value larger than specified precision allowed for this column

SQL>
SQL> -- will round to whole number
SQL> INSERT INTO number_test (col1) VALUES (12.3456);
1 row created.
```

```
SQL> SELECT * FROM number_test;
      COL0      COL1      COL2
-----  -----
    12.2345    2345    23.253
        123      23       2
    1234556
    12.34566
                  12

6 rows selected.

SQL>
SQL> -- Rounds up, only looks at the first number after the decimal point
SQL> -- for rounding whole numbers
SQL> INSERT INTO number_test (col1) VALUES (12.5);

1 row created.

SQL> INSERT INTO number_test (col1) VALUES (12.45);

1 row created.

SQL> SELECT * FROM number_test;
      COL0      COL1      COL2
-----  -----
    12.2345    2345    23.253
        123      23       2
    1234556
    12.34566
          12
          13
          12

8 rows selected.

SQL>
SQL> -- col2 number(5,3). Will round the significant digits to 3 digits
SQL> INSERT INTO number_test (col2) VALUES (12.2545);

1 row created.
```

```

SQL> SELECT * FROM number_test;
      COL0      COL1      COL2
-----  -----
    12.2345    2345    23.253
        123      23       2
    1234556
    12.34566
          12
          13
          12
                           12.255

9 rows selected.

SQL>
SQL> -- col2 number(5,3) invalid. can only have two digits to the left of
SQL> -- decimal point. This would be invalid
SQL> INSERT INTO number_test (col2) VALUES (1234.2);
INSERT INTO number_test (col2) VALUES (1234.2)
*
ERROR at line 1:
ORA-01438: value larger than specified precision allowed for this column

SQL>
SQL> -- col2 number(5,3) invalid. can only have two digits to the left of
SQL> -- decimal point
SQL> INSERT INTO number_test (col2) VALUES (123);
INSERT INTO number_test (col2) VALUES (123)
*
ERROR at line 1:
ORA-01438: value larger than specified precision allowed for this column

SQL>
SQL> -- col2 number(5,3) will round the significant digits to 23.235
SQL> INSERT INTO number_test (col2) VALUES (23.23456);

1 row created.

SQL> SELECT * FROM number_test;
      COL0      COL1      COL2
-----  -----
    12.2345    2345    23.253
        123      23       2
    1234556
    12.34566
          12
          13
          12
                           12.255
                           23.235

10 rows selected.

SQL>
SQL> -- INVALID: goes beyond the two digits for the whole number
SQL> INSERT INTO number_test (col2) VALUES (99.9999);
INSERT INTO number_test (col2) VALUES (99.9999)
*
ERROR at line 1:
ORA-01438: value larger than specified precision allowed for this column

```

## 4.3 Inserting Dates

The date value uses the default date format for Oracle: two- digit day, three- character-month, and two-digit year, separated by hyphens. (You can also provide a four- digit year value.)

The syntax of the TO\_DATE function is TO\_DATE(' d', ' f'), where d represents the date entered by the user, and f is the format-tint instruction for the date.

The TO\_CHAR function is often used to convert dates and numbers to a formatted character string. It's the opposite of the TO\_DATE function for handling date data discussed previously. The TO\_DATE function allows you to enter a date in any type of format and use the format argument to read the value as a date. The TO\_CHAR function, on the other hand, is used to have Oracle display dates in a particular format. The syntax of the TO\_CHAR function is TO\_CHAR( n, ' f'), where n is the date or number to format, and f is the format-tint instruction to use.

Date	Description	Example
Month	Name of the month spelled out	APRIL
MON	Three-letter abbreviation for the name of the month	APR
MM	Two-digit numeric value for the month	04
RM	Roman numeral representing the month	IV
D	Numeric value for the day of the week	4 (indicates Wednesday)
DD	Numeric value for the day of the month	28
DDD	Numeric value for day of the year	365 (indicates December 31)
DAY	Name of day of the week	WEDNESDAY
DY	Three-letter abbreviation for day of the week	WED
YYYY	Four digit numeric value for the year	2009
YY or YY or Y	Numeric value for the last three, two, or single digit of the year	009, 09, or 9
YEAR	Spelled-out version of the year	TWO THOUSAND NINE
BC or AD	Value indicating B.C. or A.D.	2009 A.D.

Time	Description	Example
SS	Seconds	0-59
SSSS	Seconds past midnight	0-86399
MI	Minutes	0-59
HH or HH12	Hours	12
HH24	Hours (for military time)	0-23
A.M. or P.M.	Value indicating morning or evening hours	A.M. or P.M

### **Example 4.3a (Inserting dates)**

```

CREATE TABLE date_test
( col DATE );

--Default date format "dd mon yy" or four digit year can be separated with spaces or dashes
--What is inserted is both date and time
INSERT INTO date_test VALUES ('01 feb 11');
INSERT INTO date_test VALUES ('01-feb-11');

--This will be inserted as year 2045.
INSERT INTO date_test VALUES ('01-feb-45');

INSERT INTO date_test VALUES ('01-feb-1945');

--Insert the current date and time.
INSERT INTO date_test VALUES (SYSDATE);

--The time is not visible but it is in there.
SELECT * FROM date_test;

SQL> CREATE TABLE date_test
2 (
3   col DATE
4 );
Table created.

SQL>
SQL> --default date format mm dd yy or four digit year
SQL> --can be separated with spaces or dash
SQL> --What is inserted is both date and time
SQL> INSERT INTO date_test VALUES ('01 feb 11');

1 row created.

SQL> INSERT INTO date_test VALUES ('01-feb-11');

1 row created.

SQL> --This will be inserted as year 2045
SQL> INSERT INTO date_test VALUES ('01-feb-45');

1 row created.

SQL> INSERT INTO date_test VALUES ('01-feb-1945');

1 row created.

SQL> --Insert the current date and time
SQL> INSERT INTO date_test VALUES (SYSDATE);

1 row created.

SQL> --The time is not visible but it is in there
SQL> SELECT * FROM date_test;

COL
-----
01-FEB-11
01-FEB-11
01-FEB-45
01-FEB-45
12-DEC-11

```

--Use the to\_char function to display in some other format. It will display each date in the (COL) column according to the codes .

```
SELECT TO_CHAR (col,'YYYY/MM/DD') FROM date_test;
SELECT TO_CHAR (col,'YY MM DD') FROM date_test;
```

--Note that (mi) is for minutes.

```
SELECT TO_CHAR (col,'YYYY/MM/DD hh24:mi:ss') FROM date_test;
```

```
SQL> --Use the to_char function to display in some other format
SQL> --It will display each date in the (COL) column according to the
SQL> --codes
SQL> SELECT TO_CHAR (col,'YYYY/MM/DD') FROM date_test;
```

```
TO_CHAR(CO
```

```
-----
2011/02/01
2011/02/01
2045/02/01
1945/02/01
2011/12/12
```

```
SQL> SELECT TO_CHAR (col,'YY MM DY') FROM date_test;
```

```
TO_CHAR(C
```

```
-----
11 02 TUE
11 02 TUE
45 02 WED
45 02 THU
11 12 MON
```

--Note that (mi) is for minutes.

```
SQL> SELECT TO_CHAR (col,'YYYY/MM/DD hh24:mi:ss') FROM date_test;
```

```
TO_CHAR(COL,'YYYY/M
```

```
-----
2011/02/01 00:00:00
2011/02/01 00:00:00
2045/02/01 00:00:00
1945/02/01 00:00:00
2011/12/12 10:36:56
```

--inserting a non-default format use to\_date

```
INSERT INTO date_test VALUES (TO_DATE('1999/02/04','YYYY/MM/DD'));
```

```
INSERT INTO date_test VALUES (TO_DATE('99/02/04','YY/DD/MM'));
```

```
INSERT INTO date_test VALUES (TO_DATE('99/02/04 23:23:20','YY/DD/MM
HH24:MI:SS'));
```

```
SQL> --inserting a non-default format use to_date
SQL> INSERT INTO date_test VALUES (TO_DATE('1999/02/04','YYYY/MM/DD'));
```

1 row created.

```
SQL> INSERT INTO date_test VALUES (TO_DATE('99/02/04','YY/DD/MM'));
```

1 row created.

```
SQL> INSERT INTO date_test VALUES (TO_DATE('99/02/04 23:23:20','YY/DD/MM HH24
:MI:SS'));
```

1 row created.

```

SELECT TO_CHAR (col,'YYYY/MM/DD hh24:mi:ss') FROM date_test;
SQL> SELECT TO_CHAR (col,'YYYY/MM/DD hh24:mi:ss') FROM date_test;
TO_CHAR(COL,'YYYY/M
-----
2011/02/01 00:00:00
2011/02/01 00:00:00
2045/02/01 00:00:00
1945/02/01 00:00:00
2011/12/12 10:36:56
1999/02/04 00:00:00
2099/04/02 00:00:00
2099/04/02 23:23:20
8 rows selected.

```

### Example 4.3b (Change date format)

- Change the way date is dealt with for the session only.

```

--Change the way date is dealt with for the session only. These changes occur to the
--NLS_SESSION_PARAMETERS table. This becomes the new default format that needs to be followed
--when inserting dates. Dates will also be displayed in this format.
ALTER SESSION SET nls_date_format = 'MON-DD-YY HH24:MI:SS';
SELECT sysdate FROM dual;

--INVALID: This does not fit the new format
INSERT INTO date_test VALUES ('01 feb 99');

--This does fit the new format.
INSERT INTO date_test VALUES ('FEB-01-99');

SQL> -- Change the way date is dealt with for the session only
SQL> -- These changes occur to the NLS_SESSION_PARAMETERS table
SQL> --This becomes the new default format that needs to be followed when
SQL> --inserting dates. Dates will also be displayed in this format
SQL> ALTER SESSION SET nls_date_format = 'MON-DD-YY HH24:MI:SS';

Session altered.

SQL> SELECT sysdate FROM dual;
SYSDATE
-----
DEC-12-11 10:40:18

SQL>
SQL> -- This does not fit the new format
SQL> INSERT INTO date_test VALUES ('01 feb 99');
INSERT INTO date_test VALUES ('01 feb 99')
*
ERROR at line 1:
ORA-01843: not a valid month

SQL>
SQL> -- This does fit the new format
SQL> INSERT INTO date_test VALUES ('FEB-01-99');

1 row created.

```

```
--Change the way date is dealt with for the session
ALTER SESSION SET nls_date_format = 'dy mm-YY';
SELECT sysdate FROM dual;

--INVALID: What day of the month are we inserting?
INSERT INTO date_test VALUES ('mon:02-99');

SQL> -- Change the way date is dealt with for the session
SQL> ALTER SESSION SET nls_date_format = 'dy mm-YY';
Session altered.

SQL> SELECT sysdate FROM dual;
SYSDATE
-----
mon 12-11

SQL>
SQL> -- Problem because what day of the month are we inserting because it
SQL> --doesn't know which day of the month we are inserting
SQL> INSERT INTO date_test VALUES ('mon:02-99');
SQL> INSERT INTO date_test VALUES ('mon:02-99')
      *
ERROR at line 1:
ORA-01835: day of week conflicts with Julian date
```

## ✓ CHECK 4A

1. Insert a record into the Person table. The date should be in the format mm/yyyy/dd  
hh24:mi:ss
2. Display the records in the Person table. The date should be displayed in the format  
yyyy/dd/mm

*“The advantage of a bad memory is that one enjoys several times the same good things for the first time.”*

## 4.4 Sequences

A sequence generates sequential integers that can serve as primary keys for tables. Sequences aren't assigned to a specific column or table. They are independent objects and, therefore, different users can use the same sequence to generate values that are inserted into several different tables. In other words, the same sequence could be used to generate order numbers and customer numbers. This use results in gaps in the sequence values appearing in each table

The INCREMENT BY clause specifies the interval between two sequential values. For checks and invoices, this interval is usually 1.

The START WITH clause establishes the starting value for the sequence.

The MINVALUE and MAXVALUE clauses establish a minimum or maximum value for the sequence. If the sequence is incremented with a positive value, using the MINVALUE clause doesn't make sense.

The CYCLE and NOCYCLE options determine whether Oracle should begin reusing values from the sequence after reaching the minimum or maximum value.

### ***Example 4.4a (Create sequence)***

--Creating a sequence

```
DROP TABLE patient_disease;
DROP TABLE patient;
DROP TABLE disease;
DROP SEQUENCE patient_patient_id_seq;
DROP SEQUENCE disease_disease_id_seq;

CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Height     NUMBER,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20)
);

CREATE TABLE Disease
(
    diseaseid NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);

--The sequence starts with 1 and then increments by 1.
--Note that the sequence is not associated with any table. It will be used in the insert
--statement and can actually be used with many tables
CREATE SEQUENCE patient_patient_id_seq START WITH 1;

--A sequence is created that starts with 10 and is incremented by 4
CREATE SEQUENCE disease_disease_id_seq INCREMENTBY 4 START WITH 10;
```

```

SQL> CREATE TABLE Patient
  2 (
  3   Patient_id      NUMBER PRIMARY KEY,
  4   Height          NUMBER,
  5   Fname           VARCHAR2(20),
  6   Lname           VARCHAR2(20)
  7 );
Table created.

SQL>
SQL> CREATE TABLE Disease
  2 (
  3   diseaseid       NUMBER PRIMARY KEY,
  4   disease_desc    VARCHAR2(20)
  5 );
Table created.

SQL>
SQL> --A sequence is created that starts with 1 and then increments by 1
SQL> --Note that the sequence is not associated with any table. It will be used
in the insert
SQL> --statement and can actually be used with many tables
SQL> CREATE SEQUENCE patient_patient_id_seq START WITH 1;
Sequence created.

SQL> --A sequence is created that starts with 10 and is incremented by 4
SQL> CREATE SEQUENCE disease_disease_id_seq INCREMENT BY 4 START WITH 10;
Sequence created.

```

### ***Example 4.4b (Get access to next sequence)***

You can access sequence values by using the two pseudo-columns NEXTVAL and CURRVAL. The pseudo-column NEXTVAL is used to generate the sequence value. After a value is generated, it's stored in the CURRVAL (CURRENT VALUE) pseudo-column so that you can reference it again.

```

--Inserts the next number in the sequence.
INSERT INTO patient VALUES
(patient_patient_id_seq.nextval,10,'john','james');

/* Can use the DUAL table to find out what the next value is. When nextval is used, it
--automatically increments the sequence. There will be more discussion on the DUAL table. */
SELECT patient_patient_id_seq.nextval FROM DUAL;

INSERT INTO patient VALUES
(patient_patient_id_seq.nextval,40,'jimm','jones');

SELECT * FROM patient;

```

```

SQL> --inserts the next number in the sequence
SQL> INSERT INTO patient VALUES (patient_patient_id_seq.nextval,10,'john','james
');
1 row created.

SQL> --can use the DUAL table to find out what the next value is. When nextval is used, it
SQL> --automatically increments the sequence. There will be more discussion on the DUAL table
SQL> SELECT patient_patient_id_seq.nextval FROM DUAL;
  NEXTVAL
  -----
    3

SQL> INSERT INTO patient VALUES (patient_patient_id_seq.nextval,40,'jimm','jones
');
1 row created.

SQL> SELECT * FROM patient;
PATIENT_ID      HEIGHT FNAME          LNAME
-----  -----
  2              10   john            james
  4              40   jimm            jones

```

### ***Example 4.4c (User\_sequences)***

--Examining the user\_sequences table

--This is a system table that contains information about all the sequences that the user has --created.

```

DESC USER_SEQUENCES;
SELECT sequence_name FROM user_sequences;
SQL> DESC USER_SEQUENCES;
Name                           Null?    Type
-----                         -----
SEQUENCE_NAME                  NOT NULL VARCHAR2(30)
MIN_VALUE                      NUMBER
MAX_VALUE                      NUMBER
INCREMENT_BY                   NOT NULL NUMBER
CYCLE_FLAG                     VARCHAR2(1)
ORDER_FLAG                      VARCHAR2(1)
CACHE_SIZE                      NOT NULL NUMBER
LAST_NUMBER                     NOT NULL NUMBER

SQL> SELECT sequence_name FROM user_sequences;
SEQUENCE_NAME
-----
DISEASE_DISEASE_ID_SEQ
PATIENT_PATIENT_ID_SEQ

```

### ***Example 4.4d (Alter sequences)***

You can change settings for a sequence by using the ALTER SEQUENCE command. However, any changes are applied only to values generated after the modifications are made. The only restrictions that apply to changing the sequence settings are as follows:

- The START WITH clause can't be changed because the sequence would have to be dropped and re-created to make this change. The changes can't make previously issued sequence values invalid.

--Can use the alter command to modify the increment value

```
ALTER SEQUENCE patient_patient_id_seq INCREMENT BY 2;
```

--Confirm the change. There will be more discussion on the DUAL table

```
SELECT patient_patient_id_seq.currval FROM DUAL;
```

```
SELECT patient_patient_id_seq.nextval FROM DUAL;
```

```
SQL> --Can use the alter command to modify the increment value
SQL> ALTER SEQUENCE patient_patient_id_seq INCREMENT BY 2;
```

```
Sequence altered.
```

--Confirm the change. There will be more discussion on the DUAL table

```
SQL> SELECT patient_patient_id_seq.currval FROM DUAL;
```

```
CURRVAL
```

```
-----
```

```
4
```

```
SQL> SELECT patient_patient_id_seq.nextval FROM DUAL;
```

```
NEXTVAL
```

```
-----
```

```
6
```

```
INSERT INTO disease VALUES (disease_disease_id_seq.nextval,'yellow fever');
```

```
SELECT * FROM disease;
```

```
SELECT disease_disease_id_seq.currval FROM DUAL;
```

```
SQL> INSERT INTO disease VALUES (disease_disease_id_seq.nextval,'yellow fever');
```

```
1 row created.
```

```
SQL> SELECT * FROM disease;
```

```
DISEASEID DISEASE_DESC
```

```
-----
```

```
14 malaria
```

```
22 yellow fever
```

```
SQL> SELECT disease_disease_id_seq.currval FROM DUAL;
```

```
CURRVAL
```

```
-----
```

```
22
```

--This identifies the current value without incrementing the sequence.

```
SELECT patient_patient_id_seq.currval FROM DUAL;
```

```
INSERT INTO disease VALUES (disease_disease_id_seq.nextval,'malaria');
```

```
SELECT disease_disease_id_seq.nextval FROM DUAL;
```

```
SQL> --This identifies the current value without incrementing
SQL> SELECT patient_patient_id_seq.currval FROM DUAL;
  CURRVAL
  -----
        4

SQL>
SQL> INSERT INTO disease VALUES (disease_disease_id_seq.nextval,'malaria');
1 row created.

SQL> SELECT disease_disease_id_seq.nextval FROM DUAL;
  NEXTVAL
  -----
        18
```

### ***Example 4.4e (Drop sequences)***

```
DROP SEQUENCE patient_patient_id_seq;
```

### ***✓ CHECK 4B***

1. Create a sequence for the primary key of the Person table.
2. Insert a record into the Person table using the newly created sequence.

*"A computer lets you make more mistakes faster than any invention in human history - with the possible exceptions of handguns and tequila. "*

## Summary Examples

```

DROP table test;
CREATE TABLE test
(
    colA VARCHAR(3),
    colB CHAR(2),
    colC NUMBER(3,2),
    colD NUMBER(3),
    colE DATE,
    colF char
);

--Inserts have to be done in the same sequence as the columns appear in the create table statement if the
--columns are not specified.
--VARCHAR inserts the data without padding the data with zeroes.
--CHAR pads the data with spaces to reach the max size.
--NUMBER(3,2): one digit for the whole number and two digits for the precision.
--The precision will be rounded if it is more.
--The default date format is dd-mon-yy or yyyy.
--The keyword NULL or single quotes with no spaces can be used to represent NULL (void of data).
INSERT INTO TEST VALUES ('ab','a',1.234,123,'01-feb-1996',NULL);

/* Can identify the columns that you want without concern for the order in the create table statement. Keep in
mind that when inserting, constraints cannot be violated, which means that if there is a primary key constraint,
it requires data and thus, has to be included in the column list.
When inserting other than default date format, the to_date function has to be used. Both the date and time are
inserted into a date column*/
INSERT INTO TEST (colA, colE) VALUES ('ab',to_date('01/01/1990','mm/dd/yyyy'));

/* When displaying the date in some other format use to_char. Keep in mind, for inserting, use to_date, for
displaying use to_char. mi is for minutes and ss is for seconds. They are not displayed by default.*/
SELECT to_char(colE,'mm day yyyy mi:ss') FROM TEST;

--Create a sequence named some_seq. The system tabl, user_sequences, contains all info on sequences.
CREATE SEQUENCE some_seq INCREMENT BY 2 START WITH 2;

--The value for colD is the current value of the sequence. All other columns will contain a NULL
INSERT INTO TEST (colD) VALUES(some_seq.currval);

--Insert the nextval of sequence into the table.
INSERT INTO TEST (colD) VALUES (some_seq.nextval);

--Display the current value of sequence.
SELECT some_seq.currval FROM dual;

--System table, user_sequences, contains sequence info.
SELECT sequence_name FROM user_sequences;

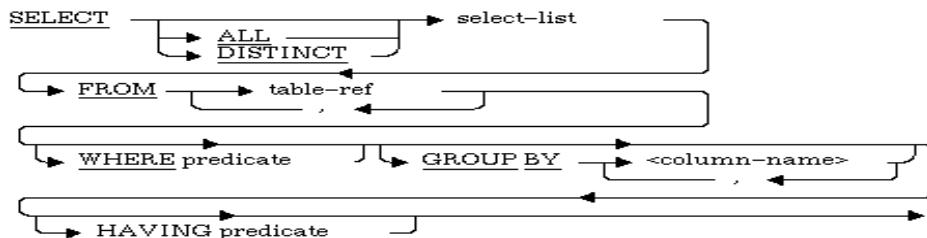
--Drop the sequence
DROP SEQUENCE some_seq;

```



*He enrolled at the University of Chicago, but dropped out after the first semester. His adoptive father was convinced that Larry would never make anything of himself, but the seemingly aimless young man had already learned the rudiments of computer programming in Chicago. He took this skill with him to Berkeley, California, arriving with just enough money for fast food and a few tanks of gas. For the next eight years, Ellison bounced from job to job, working as a technician for Fireman's Fund and Wells Fargo bank. As a programmer at Amdahl Corporation, he participated in building the first IBM-compatible mainframe system.*

## Chapter 5 (SELECT Statements)



*"Few things are worse than that moment during an argument when you realize you're wrong."*

## 5.1 What is a SELECT statement

Most of the SQL operations performed on a database in a typical organization are SELECT statements, which enable users to retrieve data from tables. The SELECT statement asks the database a question, which is why it's also known as a query.

```
SELECT [DISTINCT | UNIQUE] (*, columnname [ AS alias], ...)
  FROM tablename
  [WHERE condition]
  [GROUP BY group_by_expression]
  [HAVING group_condition]
  [ORDER BY columnname];
```

The only clauses required for the SELECT statement are SELECT and FROM. Square brackets indicate optional portions of the statement. SQL statements can be entered over several lines or on one line. Most SQL statements are entered with each clause on a separate line to improve readability and make editing easier.

The examples in this section are based on the following data:

```
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR,
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state      VARCHAR2(20)
);

INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000,
'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',50000,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-1985',60000,'Las
Vegas','NV');
INSERT INTO patient values (115,'dove','Grime','f','04-JUN-
1960',20000,'Sacramento','CA');
```

## 5.2 What is a function

A function is a predefined block of code that accepts one or more arguments— values listed inside parentheses— and then returns a single value as output. The nature of an argument depends on the syntax of the function being executed. Single- row functions return one row of results for each record processed. By contrast, multiple- row functions return only one result per group or category of rows processed, such as counting the number of books published by each publisher.

Any single- row function can be nested inside other single- row functions. When nesting functions, you should remember the following important rules:

- All arguments required for each function must be provided.
- Every opening parenthesis must have a corresponding closing parenthesis.
- The nested, or inner, function is evaluated first. The inner function's result is then passed to the outer function, and the outer function is executed.

## 5.3 Simple Select clause

The asterisk (\*) is a symbol that instructs Oracle to include all columns in the table. This symbol can be used only in the SELECT clause of a SELECT statement. If you need to view or display all columns in a table, typing an asterisk is much simpler than typing the name of each column. If the table contains a large number of fields, the results might look cluttered, or maybe, the table contains sensitive data you don't want other users to see. In these situations, you can instruct Oracle to return only specific columns in the results. Choosing specific columns in a SELECT statement is called projection. You can select one column— or as many as all columns— contained in the table.

Projection

```
SELECT * FROM patient;

--separate each of the columns with a comma.
SELECT fname, lname FROM patient;
```

PATIENT_ID	FNAME	LNAME	GENDER	DOB	SALARY	CITY	STATE
111	john	Doe	m	11-FEB-78	25000	Davis	CA
112	john	Smith	m	01-MAR-81	40000	Davis	CA
113	jill	Crane	m	12-APR-99	50000	Reno	NV
114	billy	Bob	f	05-MAY-85	60000	Las Vegas	NV
115	dove	Grime	f	04-JUN-60	20000	Sacramento	CA

FNAME	LNAME
john	Doe
john	Smith
jill	Crane
billy	Bob
dove	Grime



## 5.4 Alias

Sometimes a column name is a vague indicator of the data that's displayed. To better describe the data displayed in the output, you can substitute a column alias for the column name in query results. You need to keep some guidelines in mind when using a column alias. If the column alias contains spaces or special symbols, or if you don't want it to appear in all uppercase letters, you must enclose it in quotation marks (""). By default, column headings shown in query results are capitalized. Using quotation marks overrides this default setting. If the column alias consists of only one word without special symbols, it doesn't need to be enclosed in quotation marks

--Can modify the headings with your choice of alias. This does not change the underlying --columnname in the table. It only displays a different heading for this one SQL statement.

```
SELECT fname, lname AS lastname FROM patient;
```

--Can also come up with an alias without using the AS keyword.

```
SELECT fname, lname lastname FROM patient;
```

--If an alias is comprised of multiple words then enclose it in double quotes.

```
SELECT fname, lname "Last Name" FROM patient;
```

```
SQL> SELECT fname, lname AS lastname FROM patient;
```

FNAME	LASTNAME
john	Doe
john	Smith
jill	Crane
billy	Bob
dove	Grime

```
SQL> SELECT fname, lname lastname FROM patient;
```

FNAME	LASTNAME
john	Doe
john	Smith
jill	Crane
billy	Bob
dove	Grime

SELECT Lname AS "Last\_Name" FROM Chapter5;  
SELECT Fname AS First\_Name, Lname AS Last\_Name FROM Chapter5;

```
SQL> SELECT fname, lname "Last Name" FROM patient;
```

FNAME	Last Name
john	Doe
john	Smith
jill	Crane
billy	Bob
dove	Grime

## 5.5 Concatenation

Use the concatenation operator (||) to concatenate, or combine, data from columns with string literals.

--Notice the comma is removed and everything is displayed as a single columns

```
SELECT fname || lname || gender FROM patient;
```

-- spaces do not matter

```
--SELECT fname||lname||gender FROM patient;
```

--can add spaces to make the info more readable. Also can add an alias

```
SELECT fname || ' ' || lname || ':' ' || gender AS Info FROM patient;
```

```

SQL> SELECT fname || lname || gender FROM patient;
FNAME||LNAME||GENDER
-----
johnDoe
johnSmith
jillCrane
billyBob
doveGrime

SQL> -- spaces do not matter
SQL> --SELECT fname||lname||gender FROM patient;
SQL> SELECT fname || ' ' || lname || ':' || gender AS Info FROM patient;

INFO
-----
john Doe: m
john Smith: m
jill Crane: m
billy Bob: f
dove Grime: f

```

The CONCAT function can also be used to concatenate data from two columns. The main difference between the concatenation operator and the CONCAT function is that you can combine a long list of columns and string literals with the concatenation operator. By contrast, you can combine only two items (columns or string literals) with the **CONCAT function**.

The concatenation operator is usually preferred because it's not limited to two items. If you need to combine more than two items with the CONCAT function, you must nest a CONCAT function inside another CONCAT function.

--Instead of the pipe symbols, the concat function can also be used. It only takes two arguments.

```

SELECT CONCAT (fname, lname) FROM patient;

SELECT CONCAT (fname, lname || gender) FROM patient;

```

--Can embed one concat inside of another.

```

SELECT CONCAT (fname, CONCAT( lname, gender||lname)) as heading FROM
patient;

```

```

SQL> --Instead of the pipe symbols, the concat function can also be used. It takes two arguments
SQL> SELECT CONCAT (fname, lname) FROM patient;
CONCAT(FNAME,LNAME)
-----
johnDoe
johnSmith
jillCrane
billyBob
doveGrime

SQL> SELECT CONCAT (fname, lname || gender) FROM patient;
CONCAT(FNAME,LNAME||GENDER)
-----
johnDoeM
johnSmithM
jillCranem
billyBobf
doveGrimef

SQL> --Can embed one concat inside of another
SQL> SELECT CONCAT (fname,CONCAT( lname, gender||lname)) as heading FROM patient;
HEADING
-----
johnDoeM Doe
johnSmithM Smith
jillCranem Crane
billyBobf Bob
doveGrimef Grime

```

## 5.6 LTRIM/RTRIM/TRIM

You can use the LTRIM function to remove a specific string of characters from the left side of data values. The syntax of the LTRIM function is LTRIM(c, s), where c represents the field to modify, and s represents the string to remove from the left side of data.

Oracle also supports the RTRIM function to remove specific characters from the right side of data values. The syntax of the RTRIM function is RTRIM(c, s). The c represents the field to modify, and s represents the string to remove from the right side of data. The TRIM function has several variations that can potentially replace both RTRIM and LTRIM.

```

--This record is being inserted so that we can remove the hyphens using the trim function.
INSERT INTO patient VALUES (777, '---john----', ' Doe', 'f', '05-DEC-1990', 90000, 'Davis', 'CA');

--Removes the dashes from the left side and right side: Keep in mind this is for display purposes only.
-- It doesn't change the actual data in the table.
SELECT LTRIM(fname, '-'), RTRIM(fname, '-') FROM patient;

--Can also use the trim function with LEADING or TRAILING options. Without those options it will
--look at both the left and the right sides.
SELECT TRIM(LEADING '-' FROM fname), TRIM (TRAILING '-' FROM fname),
TRIM('-' FROM fname) FROM patient;

```

```

trim function
SQL> INSERT INTO patient VALUES (777,'---john----', ' Doe','f','05-DEC-1990
00, 'Davis','CA');

1 row created.

SQL> --removes the trims from the left side and right side: Keep in mind th
for display. It doesn't actually change
SQL> --the date in the table
SQL> SELECT LTRIM(fname,'-'), RTRIM(fname,'-') FROM patient;

LTRIM(FNAME,'-')      RTRIM(FNAME,'-')
-----
john                  john
john                  john
jill                  jill
billy                 billy
dove                 dove
john----              ---john

6 rows selected.

SQL> SELECT TRIM(LEADING '-' FROM fname),TRIM (TRAILING'-' FROM fname), TRI
      FROM fname) FROM patient;

TRIM(LEADING'-'FROMF TRIM(TRAILING'-'FROM TRIM('-'FROMFNAME)
-----
john                  john                  john
john                  john                  john
jill                  jill                  jill
billy                 billy                 billy
dove                 dove                 dove
john----              ---john              john

6 rows selected.

```

## 5.7 DISTINCT or UNIQUE

Eliminate duplication values.

The DISTINCT/UNIQUE keyword eliminates duplicate values in the results.

```

SELECT city, state  FROM patient;

--Gets rid of the duplicate cities.
SELECT DISTINCT city FROM patient;

--Gets rid of the duplicate city, state combination like a composite key.
SELECT DISTINCT city, state FROM patient;

--INVALID. Must apply DISTINCT to the entire row.
SELECT city, DISTINCT state FROM patient;

--Can also use the UNIQUE keyword instead of distinct.
SELECT UNIQUE city, state FROM patient;

```

```

SQL> SELECT city, state FROM patient;
CITY           STATE
-----
Davis          CA
Davis          CA
Reno           NV
Las Vegas     NV
Sacramento    CA
Davis          CA

6 rows selected.

SQL> --Gets rid of the duplicate cities
SQL> SELECT DISTINCT city FROM patient;
CITY
-----
Las Vegas
Davis
Sacramento
Reno

SQL> --Gets rid of the duplicate city, state combination like a composite key
SQL> SELECT DISTINCT city, state FROM patient;
CITY           STATE
-----
Sacramento    CA
Las Vegas     NV
Reno           NV
Davis          CA

SQL> --INVALID. Must apply DISTINCT to the entire row
SQL> SELECT city, DISTINCT state FROM patient;
SELECT city, DISTINCT state FROM patient
*
ERROR at line 1:
ORA-00936: missing expression

SQL> --can also use the UNIQUE keyword instead of distinct
SQL> SELECT UNIQUE city, state FROM patient;
CITY           STATE
-----
Sacramento    CA
Las Vegas     NV
Reno           NV
Davis          CA
SQL>

```

## ✓ CHECK 5A

1. Create the person table and insert some records before starting with the questions.
2. Display the contents of the Person table. Use the alias Last name.
3. Concatenate the firstname, lastname and the salary separated by a space, using the concat function.
4. Display all the unique lastnames from the Person table.

*"Great minds discuss ideas, Average minds discuss events, Small minds discuss people"*

## 5.8 DUAL table

Any of the single- row functions covered in this chapter can be used with the DUAL table

### *Example 5.8a (Dual table)*

--The dual table contains only one column called dummy.

```
DESC DUAL;
```

--There is only a single row of information in this table which is (X).

```
SELECT * FROM DUAL;
```

```
DESC DUAL
Name Null Type
-----
DUMMY      VARCHAR2(1)

DUMMY
-----
X
```

### *Example 5.8b (Using the Dual table)*

--The reason why we want to use the dual table is because there is only one record. In this instance, notice it displays the literal text (hello) for every record

```
SELECT 'hello', fname FROM patient;
SELECT 'hello' FROM patient;
```

--Displays hello only one time

```
SELECT 'hello' FROM dual;
```

```
SQL> --The reason why we want to use the dual table is because there is only one record. In this
SQL> --instance, notice it displays the literal text (hello) for every record
SQL> SELECT 'hello', fname FROM patient;
```

```
'HELL FNAME
-----
hello john
hello john
hello jill
hello billy
hello dove
hello ---john----
```

6 rows selected.

```
SQL> SELECT 'hello' FROM patient;
```

```
'HELL
-----
hello
hello
hello
hello
hello
hello
```

6 rows selected.

```
SQL> --Displays hello only one time
SQL> SELECT 'hello' FROM dual;
```

```
'HELL
-----
hello
```

```
--When displaying the square root, we only want to see the result one time. We don't want it
--to be repeated for each row of a table. It is redundant which is why we would use the dual table.
SELECT SQRT(4), fname FROM patient;

SELECT SQRT(4) FROM patient;

SELECT SQRT(4) FROM dual;

SQL> --When displaying the square root, we only want to see the result one
   We don't want it
SQL> --to be repeated for each row of a table. It is redundant
SQL> SELECT SQRT(4), fname FROM patient;
   SQRT(4) FNAME
   -----
      2 john
      2 john
      2 jill
      2 billy
      2 dove
      2 ---john---
6 rows selected.

SQL> SELECT SQRT(4) FROM patient;
   SQRT(4)
   -----
      2
      2
      2
      2
      2
      2
6 rows selected.

SQL> SELECT SQRT(4) FROM dual;
   SQRT(4)
   -----
      2
```

## 5.9 INITCAP

The initcap function sets the first character in each word to uppercase and the rest to lowercase.  
The syntax for the initcap function is: `initcap(string1)` where string1 is the string argument whose first character in each word will be converted to uppercase and all remaining characters converted to lowercase.

```
--Does not change, just displays. First letter is upper cased.
SELECT INITCAP('tech on the net') FROM dual;

--First letter is uppercased.
SELECT INITCAP('GEORGE SOROS') FROM dual;
SELECT INITCAP(fname) FROM patient;
```

```

SQL> --Does not change, just displays, First letter is upper cased
SQL> SELECT INITCAP('tech on the net') FROM dual;
INITCAP('TECHON
-----
Tech On The Net

SQL> --First letter is uppercased
SQL> SELECT INITCAP('GEORGE SOROS') FROM dual;
INITCAP('GEO
-----
George Soros

SQL> SELECT INITCAP(fname) FROM patient;
INITCAP(FNAME)
-----
John
John
Jill
Billy
Dove
---John-----
6 rows selected.

```

## 5.10 SUBSTR, INSTR and REPLACE

The INSTR (instrng) function searches a string for a specified set of characters or a sub-string, and then returns a numeric value representing the first character position in which the substring is found. If the substring doesn't exist in the string value, a 0 ( zero) is returned. Two arguments must be provided to the INSTR function: the string value to search and the characters or substring (enclosed in single quotes) to locate. Two optional arguments are also available: start position, indicating on which character of the string value the search should begin, and occurrence, which is the instance of the search value to locate ( that is, first occurrence, second occurrence, and so on). By default, the search begins at the beginning of the string value, and the position of the first occurrence is located.

### *Example 5.10a (Instr)*

```

--Return first occurrence of 'e'.
SELECT INSTR ('Tech on the net', 'e') FROM DUAL;

--The first occurrence of 'e'.
SELECT INSTR ('Tech on the net', 'e', 1, 1) FROM DUAL;

SQL> --Return first occurrence of "e"
SQL> SELECT INSTR ('Tech on the net', 'e') FROM DUAL;
INSTR('TECHONTHENET','E')
-----
2

SQL>
SQL> --The first occurrence of 'e'
SQL> SELECT INSTR ('Tech on the net', 'e', 1, 1) FROM DUAL;
INSTR('TECHONTHENET','E',1,1)
-----
2

```

```
--The second occurrence of 'e'.
SELECT INSTR ('Tech on the net', 'e', 1, 2) FROM DUAL;

--The third occurrence of 'e'.
SELECT INSTR ('Tech on the net', 'e', 1, 3) FROM DUAL;

--Looks for the first occurrence of the letter (l).
SELECT INSTR(fname,'l') , fname FROM PATIENT;
SQL> --The second occurrence of 'e'
SQL> SELECT INSTR ('Tech on the net', 'e', 1, 2) FROM DUAL;
INSTR('TECHONTHENET','E',1,2)
-----
11

SQL>
SQL> --The third occurrence of 'e'
SQL> SELECT INSTR ('Tech on the net', 'e', 1, 3) FROM DUAL;
INSTR('TECHONTHENET','E',1,3)
-----
14

SQL>
SQL> SELECT INSTR(fname,'l') , fname FROM PATIENT;
INSTR(FNAME,'L') FNAME
-----
0 john
0 john
3 jill
3 billy
0 dove
0 ---john-----
6 rows selected.
```

### **Example 5.10b (Substr)**

You can use the SUBSTR function to return a substring (a portion of a string). The syntax of this function is SUBSTR( c, p, l), where c represents the character string, p represents the beginning character position for the extraction, and l represents the length of the string to return in the query results.

```
--Extracts two characters, starting from the 6th position.
SELECT SUBSTR ('This is a test', 6, 2) FROM DUAL;

-- If the second parameter is omitted, substr will return the entire string.
SELECT SUBSTR ('This is a test', 6) FROM DUAL;

SQL> --extracts two characters starting from the 6th position
SQL> SELECT SUBSTR ('This is a test', 6, 2) FROM DUAL;
SU
--
is

SQL> -- If the second parameter is omitted, substr will return
SQL> SELECT SUBSTR ('This is a test', 6) FROM DUAL;
SUBSTR('T
-----'
is a test
```

```
--Extracts four characters, starting from the 1st position.
SELECT SUBSTR ('TechOnTheNet', 1, 4) FROM DUAL;

--If start_position is a negative number, then it starts from the end of the string and counts backwards.
SELECT SUBSTR ('TechOnTheNet', -3, 3) FROM DUAL;
SELECT SUBSTR ('TechOnTheNet', -6, 3) FROM DUAL;
SELECT SUBSTR ('TechOnTheNet', -8, 2) FROM DUAL;
```

```
SQL> --extracts four characters starting from the 1st position
SQL> SELECT SUBSTR ('TechOnTheNet', 1, 4) FROM DUAL;
SUBS
---
Tech

SQL>
SQL> --If start_position is a negative number, then substr sta
SQL> -- the end of the string and counts backwards.
SQL> SELECT SUBSTR ('TechOnTheNet', -3, 3) FROM DUAL;
SUB
---
Net

SQL> SELECT SUBSTR ('TechOnTheNet', -6, 3) FROM DUAL;
SUB
---
The

SQL> SELECT SUBSTR ('TechOnTheNet', -8, 2) FROM DUAL;
SU
--
On
```

```
SELECT SUBSTR (fname,2) , fname FROM patient;
SQL> SELECT SUBSTR (fname,2) , fname FROM patient;
SUBSTR(FNAME,2)      FNAME
-----
ohn                  john
ohn                  john
ill                  jill
illy                 billy
ove                 dove
--john----          ---john---

6 rows selected.
```

### *Example 5.10c (Replace)*

The REPLACE function is similar to the “search and replace” function used in many programs. It looks for the occurrence of a specified string of characters and, if found, substitutes it with another set of characters. The syntax of the REPLACE function is REPLACE( c, s, r), where c represents the field to search, s represents the string of characters to find, and r represents the string of characters to substitute for s.

```
-- Gets rid of everything in the second argument. It is looking for the exact pattern.
SELECT REPLACE ('123123tech', '123') FROM DUAL;

-- Doesn't matter where the 123 is.
SELECT REPLACE('123tech123', '123') FROM DUAL;

--Replaces all every 2 with a 3.
SELECT REPLACE('222tech', '2', '3') FROM DUAL;

-- Gets rid of all the zeros.
SELECT REPLACE('0000123', '0') FROM DUAL;

-- Replaces the zeros with blank spaces.
SELECT REPLACE('0000123', '0', ' ') FROM DUAL;
```

```
SQL> -- Gets rid of everything in the second argument. It is looking for the exact pattern. For example
SQL> --if the second argument is 122, it does not get rid of a
SQL> --ch a pattern does not --- exist
SQL> SELECT REPLACE ('123123tech', '123') FROM DUAL;
REPL
-----
tech

SQL> -- Doesn't matter where the 123 is
SQL> SELECT REPLACE('123tech123', '123') FROM DUAL;

REPL
-----
tech

SQL> --replaces all every 2 with a 3
SQL> SELECT REPLACE('222tech', '2', '3') FROM DUAL;

REPLACE
-----
333tech

SQL> -- gets rid of all the zeros
SQL> SELECT REPLACE('0000123', '0') FROM DUAL;

REP
---
123

SQL> -- Replaces the zeros with blank spaces
SQL> SELECT REPLACE('0000123', '0', ' ') FROM DUAL;

REPLACE('00
-----
123
```

SELECT REPLACE(fname,'bill','kill') changed, fname FROM patient;																
SQL> SELECT REPLACE(fname,'bill','kill') changed, fname FROM patient;																
<table border="1"> <thead> <tr> <th>CHANGED</th> <th>FNAME</th> </tr> </thead> <tbody> <tr> <td>john</td> <td>john</td> </tr> <tr> <td>john</td> <td>john</td> </tr> <tr> <td>jill</td> <td>jill</td> </tr> <tr> <td>killy</td> <td>billy</td> </tr> <tr> <td>dove</td> <td>dove</td> </tr> <tr> <td colspan="2">---john--- ---john----</td> </tr> <tr> <td colspan="2">-</td> </tr> </tbody> </table>	CHANGED	FNAME	john	john	john	john	jill	jill	killy	billy	dove	dove	---john--- ---john----		-	
CHANGED	FNAME															
john	john															
john	john															
jill	jill															
killy	billy															
dove	dove															
---john--- ---john----																
-																
6 rows selected.																

### ***Example 5.10d (combination of functions)***

```

DELETE FROM patient;
INSERT INTO patient VALUES (999, 'Bond, James', '', 'f', '05-DEC-1990', 90000, 'Davis', 'CA');

SELECT INSTR(fname, ',')-1, INSTR(fname,',')+1 FROM patient;
--Column contains lastname and first name separated by a comma. Want only the last name (Bond).
SELECT SUBSTR(fname,1, INSTR(fname,',')-1) FROM patient;

-- Want to extract the first name. This includes leading spaces.
SELECT SUBSTR(fname,INSTR(fname,',')+1) FROM patient;

--Get rid of the spaces.
SELECT LTRIM(SUBSTR(fname,INSTR(fname,',')+1)) FROM patient;

SQL> DELETE FROM patient;
6 rows deleted.

SQL> INSERT INTO patient VALUES (999, 'Bond, James', '', 'f', '05-DEC-1990', 90000, 'Davis', 'CA');
1 row created.

SQL>
SQL> SELECT INSTR(fname,',')-1, INSTR(fname,',')+1 FROM patient;
INSTR(FNAME,',')-1 INSTR(FNAME,',')+1
----- -----
        4               6

SQL> --column contains lasname and first name separated by a comma
SQL> -- Want only the last name (Bond)
SQL> SELECT SUBSTR(fname,1, INSTR(fname,',')-1) FROM patient;
SUBSTR(FNAME,1,INSTR(
-----
Bond

SQL>
SQL> -- Want to extract the first name. This includes leading spaces
SQL> SELECT SUBSTR(fname,INSTR(fname,',')+1) FROM patient;
SUBSTR(FNAME,INSTR(F
-----
James

SQL>
SQL> --Get rid of the spaces
SQL> SELECT LTRIM(SUBSTR(fname,INSTR(fname,',')+1)) FROM patient;
LTRIM(SUBSTR(FNAME,I
-----
James

```

### ***✓ CHECK 5B***

1. Display the first three letters of the last name. Make sure the first letter is in uppercase.
2. Identify the location of the letter 'e' in all the firstnames.

*“The trouble with our times is that the future is not what it used to be.”*

## 5.11 LPAD, RPAD

The LPAD function can be used to pad, or fill in, the area to the left of a character string with a specific character—or even a blank space. The syntax of the LPAD function is LPAD( c, l, s), where **c** represents the character string to pad, **l** represents the length of the character string after padding, and **s** represents the symbol or character (enclosed in single quotes) to use as padding, RPAD is similar except that it pads from the right hand side.

```
--Pad the left hand side with dots.
SELECT LPAD(fname, 20, '.') FROM patient;

--Pad the right hand side with dots.
SELECT RPAD(fname,20,'.') FROM patient;
SQL> --pad the left hand side with dots
SQL> SELECT LPAD(fname, 20, '.') FROM patient;
LPAD(FNAME,20,'.')
-----
.....Bond, James

SQL> --pad the right hand side with dots
SQL> SELECT RPAD(fname,20,'.') FROM patient;
RPAD(FNAME,20,'.')
-----
Bond, James.....
```

## TRUNC, ROUND, FLOOR, CEIL

The syntax of the ROUND function is ROUND (d, u), where **d** represents the date data, or field, to round, and **u** represents the unit to use for rounding. A date can be rounded by the unit of month or year.

### *Example 5.12a (Round)*

```
SELECT ROUND(12.5) FROM DUAL;
SELECT ROUND(12.1) FROM DUAL;
SELECT ROUND(12.56,1) FROM DUAL;
SELECT ROUND(12.54,1) FROM DUAL;
SQL> SELECT ROUND(12.5) FROM DUAL;
ROUND(12.5)
-----
13
SQL> SELECT ROUND(12.1) FROM DUAL;
ROUND(12.1)
-----
12
SQL> SELECT ROUND(12.56,1) FROM DUAL;
ROUND(12.56,1)
-----
12.6
SQL> SELECT ROUND(12.54,1) FROM DUAL;
ROUND(12.54,1)
-----
12.5
```

### ***Example 5.12b (Trunc)***

At times you need to truncate, rather than round, numeric data. You can use the TRUNC ( truncate) function to truncate a numeric value to a specific position. Any numbers after that position are simply removed (truncated). The syntax of the TRUNC function is TRUNC( n, p), where n represents the numeric data or field to truncate, and p represents the position of the digit where data should be truncated.

```

SELECT TRUNC(12.549,2) FROM DUAL;
SELECT TRUNC(12.5) FROM DUAL;
SELECT TRUNC(12.1) FROM DUAL;
SELECT TRUNC(12.56,1) FROM DUAL;
SELECT TRUNC(12.54,1) FROM DUAL;
SELECT TRUNC(12.549,2) FROM DUAL;

SQL> SELECT TRUNC(12.549,2) FROM DUAL;
TRUNC(12.549,2)
-----
      12.54

SQL> SELECT TRUNC(12.5) FROM DUAL;
TRUNC(12.5)
-----
      12

SQL> SELECT TRUNC(12.1) FROM DUAL;
TRUNC(12.1)
-----
      12

SQL> SELECT TRUNC(12.56,1) FROM DUAL;
TRUNC(12.56,1)
-----
      12.5

SQL> SELECT TRUNC(12.54,1) FROM DUAL;
TRUNC(12.54,1)
-----
      12.5

SQL> SELECT TRUNC(12.549,2) FROM DUAL;
TRUNC(12.549,2)
-----
      12.54
  
```

### ***Example 5.12c (Ceil)***

Returns the ceiling value (next highest integer above a number).

The syntax for the ceil function is: ceil( number )

```

SELECT CEIL(12.5) FROM DUAL;

SELECT CEIL(12.1) FROM DUAL;

SELECT CEIL(13.9) FROM DUAL;
  
```

```

SQL> SELECT CEIL(12.5) FROM DUAL;
CEIL(12.5)
-----
13

SQL> SELECT CEIL(12.1) FROM DUAL;
CEIL(12.1)
-----
13

SQL> SELECT CEIL(13.9) FROM DUAL;
CEIL(13.9)
-----
14

```

### ***Example 5.12d (Floor)***

The FLOOR function rounds the specified number down.  
The syntax for the floor function is: floor( number )

```

SELECT FLOOR(12.5) FROM DUAL;

SELECT FLOOR(12.1) FROM DUAL;

SELECT FLOOR(13.9) FROM DUAL;

SQL> SELECT FLOOR(12.5) FROM DUAL;
FLOOR(12.5)
-----
12

SQL> SELECT FLOOR(12.1) FROM DUAL;
FLOOR(12.1)
-----
12

SQL> SELECT FLOOR(13.9) FROM DUAL;
FLOOR(13.9)
-----
13

```

## **5.12 Arithmetic**

Simple arithmetic operations, such as multiplication (\*), division (/), addition (+), and subtraction (-), can be used in the SELECT clause of a query. Keep in mind that Oracle adheres to the standard order of operations: 1. Moving from left to right in the arithmetic equation, any required multiplication and division operations are solved first. 2. Addition and subtraction operations are solved after multiplication and division, again moving from left to right in the equation. To override this order of operations, you can use parentheses to enclose the portion of the equation that should be calculated first.

### **Example 5.13a (Basic math)**

--Does not change the underlying table.

```
SELECT salary, salary +200, salary/2, salary * 2, salary-200 FROM patient;
```

```
SQL> SELECT salary, salary +200, salary/2, salary * 2, salary-200 FROM patient;
```

SALARY	SALARY+200	SALARY/2	SALARY*2	SALARY-200
90000	90200	45000	180000	89800

```
SQL>
```

### **Example 5.13b (Mod)**

The MOD (modulus) function returns only the remainder of a division operation.

```
SELECT salary, MOD(salary, 2) FROM patient;
```

```
SQL> SELECT salary, MOD(salary, 2) FROM patient;
```

SALARY	MOD(SALARY,2)
90000	0

### **Example 5.13c (ABS)**

The ABS (absolute) function returns the absolute, or positive, value of the numeric values supplied as the argument.

```
SELECT ABS(0) FROM DUAL;
SELECT ABS(10) FROM DUAL;
SELECT ABS(-10) FROM DUAL;
```

```
SQL> SELECT ABS(0) FROM DUAL;
```

ABS(0)
0

```
SQL> SELECT ABS(10) FROM DUAL;
```

ABS(10)
10

```
SQL> SELECT ABS(-10) FROM DUAL;
```

ABS(-10)
10

### ***Example 5.13d (Power)***

The POWER function raises the number in first argument to the power indicated as the second argument. The syntax of the POWER function is POWER( x, y), where x represents the number you're raising, and y represents the power to which you're raising it.

```
SELECT POWER(5,3) FROM DUAL;
SELECT POWER(5,0) FROM DUAL;
SQL> SELECT POWER(5,3) FROM DUAL;
POWER(5,3)
-----
125
SQL> SELECT POWER(5,0) FROM DUAL;
POWER(5,0)
-----
1
```

### ***Example 5.13e (SQRT)***

The SQRT function returns the square root of a number

```
SELECT SQRT(10) FROM DUAL;
SQL> SELECT SQRT(10) FROM DUAL;
SQRT(10)
-----
3.16227766
```

## **5.13 GREATEST, LEAST**

The greatest function returns the greatest value in a list of expressions. The syntax for the greatest function is: greatest( expr1, expr2, ... expr\_n ) where expr1, expr2, .expr\_n are expressions that are evaluated by the greatest function. If the datatypes of the expressions are different, all expressions will be converted to whatever datatype expr1 is.

The least function returns the smallest value in a list of expressions. The syntax for the least function is: Least (expr1, expr2, ... expr\_n ) where expr1, expr2, .expr\_n are expressions that are evaluated by the least function. If the datatypes of the expressions are different, all expressions will be converted to whatever datatype expr1 is.

```
SELECT GREATEST(10,60,90,3) FROM DUAL;
SELECT LEAST(10,60,90,3) FROM DUAL;
SQL> SELECT GREATEST(10,60,90,3) FROM DUAL;
GREATEST(10,60,90,3)
-----
90
SQL> SELECT LEAST(10,60,90,3) FROM DUAL;
LEAST(10,60,90,3)
-----
3
```

## 5.14 Date functions

Oracle has a number of functions that apply to a date

Sysdate	Returns the current date/time
ADD_MONTHS	Function to add a number of months to a date. For example: add_months(SYSDATE,3) returns 3 months after sysdate. This could be rounded to below if the resulting month has fewer days than the month this function is applied to.
+,- (plus-minus)	In Oracle you can add or subtract a number of days from a date. Example: sysdate+5 means systemdate/time plus 5 days
GREATEST	With the greatest function you can select the date/time that is the highest in a range of date/times. Example: greatest (sysdate+4,sysdate,sysdate-5) = sysdate+4.
LEAST	With the least function you can select the earliest date/time in a range of date/times. Example: least (sysdate+4,sysdate,sysdate-5) = sysdate-5.
LAST_DAY	Returns the last_day of a month based on the month the passed date is in. Example: last_day(sysdate) returns the last day of this month.
MONTHS_BETWEEN	Returns the number of months between two dates. The number is not rounded. Example: months_between(sysdate, to_date('01-01-2007','dd-mm-yyyy')) returns the number of months since jan 1, 2007.
NEXT_DAY	Date of next specified date following a date NEXT_DAY(, ) Options are SUN, MON, TUE, WED, THU, FRI, and SAT SELECT NEXT_DAY(SYSDATE, 'FRI') FROM dual;
ROUND	Returns date rounded to the unit specified by the format model. If you omit the format, the date is rounded to the nearest day ROUND(, ) SELECT ROUND(TO_DATE('27-OCT-00'),'YEAR') NEW_YEAR FROM dual;
TRUNC	Convert a date to the date without time (0:00h) Example: TRUNC(sysdate) returns today without time.
TO_CHAR(date,format_mask)	Converts a date to a string using a format mask.

### ***Example 5.15a (Months\_between)***

The syntax is MONTHS\_BETWEEN( d1, d2), where d1 and d2 are the two dates in question, and d2 is subtracted from d1. To eliminate the decimal portion of the output and return only the number of whole months between the two dates, you can nest the MONTHS\_BETWEEN function inside the TRUNC function,

```

SELECT DOB, MONTHS_BETWEEN(SYSDATE, DOB) FROM patient;

SELECT DOB, TRUNC(MONTHS_BETWEEN(SYSDATE, DOB)) FROM patient;

--Can also find out the years by dividing the results of months_between into 12.
SELECT DOB, TRUNC(MONTHS_BETWEEN(SYSDATE, DOB)/12) FROM patient;

SELECT DOB, ROUND(MONTHS_BETWEEN(SYSDATE, DOB)/12) FROM patient;

SQL> SELECT dob, MONTHS_BETWEEN(SYSDATE, dob) FROM patient;
DOB      MONTHS_BETWEEN(SYSDATE,DOB)
----- -----
05-DEC-90          252.24253

SQL> SELECT dob, TRUNC(MONTHS_BETWEEN(SYSDATE, dob)) FROM patient;
DOB      TRUNC(MONTHS_BETWEEN(SYSDATE,DOB))
----- -----
05-DEC-90          252

SQL> --can also find out the years by dividing the results of months_between into 12
SQL> SELECT dob, TRUNC(MONTHS_BETWEEN(SYSDATE, dob)/12) FROM patient;
DOB      TRUNC(MONTHS_BETWEEN(SYSDATE,DOB)/12)
----- -----
05-DEC-90          21

SQL> SELECT dob, ROUND(MONTHS_BETWEEN(SYSDATE, dob)/12) FROM patient;
DOB      ROUND(MONTHS_BETWEEN(SYSDATE,DOB)/12)
----- -----
05-DEC-90          21

```

### ***Example 5.15b (Add\_months)***

The syntax of the ADD\_MONTHS function is ADD\_MONTHS( d, m), where d represents the beginning date for the calculation, and m represents the number of months to add to the date.

```

SELECT DOB, ADD_MONTHS(DOB,4) FROM patient;
SELECT DOB, ADD_MONTHS(DOB,-14) FROM patient;

SQL> SELECT dob, ADD_MONTHS(dob,4) FROM patient;
DOB      ADD_MONTH
----- -----
05-DEC-90 05-APR-91

SQL> SELECT dob, ADD_MONTHS(dob,-14) FROM patient;
DOB      ADD_MONTH
----- -----
05-DEC-90 05-OCT-89

```

### **Example 5.15c (Next\_day)**

The syntax of the NEXT\_DAY function is NEXT\_DAY( d, DAY), where d represents the starting date, and DAY represents the day of the week to identify. The LAST\_DAY function is similar to the NEXT\_DAY function, except it always determines the last day of the month for a given date.

```
--The second argument represents the day as follows: --Sunday=1, Monday=2, Tuesday=3, ...
SELECT DOB,NEXT_DAY(DOB, 4) , TO_CHAR(DOB,'DY') , TO_CHAR(NEXT_DAY(DOB, 4),
,'DY') FROM patient;
SELECT DOB, LAST_DAY(DOB) , DOB FROM patient;

SQL> SELECT dob, NEXT_DAY(dob,4) , TO_CHAR(dob,'DY') , TO_CHAR(NEXT_DAY(dob,4) ,
'DY')FROM patient;
DOB      NEXT_DAY( TO_ TO_
-----
05-DEC-90 12-DEC-90 WED WED

SQL> SELECT dob, LAST_DAY(dob) , dob FROM patient;
DOB      LAST_DAY( DOB
-----
05-DEC-90 31-DEC-90 05-DEC-90
```

### **Example 5.15d (Round)**

```
SELECT sysdate FROM dual;
SELECT ROUND(sysdate,'YEAR') FROM dual;
SELECT ROUND(sysdate,'MONTH') FROM dual;
SELECT ROUND(TO_DATE('27-oct-2011'),'YEAR') FROM dual;
SELECT ROUND(TO_DATE('27-JAN-2011'),'YEAR') FROM dual;
SELECT ROUND(TO_DATE('27-OCT-2011'),'MONTH') FROM dual;
SELECT ROUND(TO_DATE('01-OCT-2011'),'MONTH') FROM dual;

SQL> SELECT sysdate FROM dual;
SYSDATE
-----
12-DEC-11

SQL> SELECT ROUND(sysdate,'YEAR') FROM dual;
ROUND(SYS
-----
01-JAN-12

SQL> SELECT ROUND(sysdate,'MONTH') FROM dual;
ROUND(SYS
-----
01-DEC-11

SQL> SELECT ROUND(TO_DATE('27-oct-2011'),'YEAR') FROM dual;
ROUND(TO_
-----
01-JAN-12

SQL> SELECT ROUND(TO_DATE('27-JAN-2011'),'YEAR') FROM dual;
ROUND(TO_
-----
01-JAN-11

SQL> SELECT ROUND(TO_DATE('27-OCT-2011'),'MONTH') FROM dual;
ROUND(TO_
-----
01-NOV-11

SQL> SELECT ROUND(TO_DATE('01-OCT-2011'),'MONTH') FROM dual;
ROUND(TO_
-----
01-OCT-11
```

### ***Example 5.15e (Greatest, Least)***

```

SELECT sysdate, sysdate+5, sysdate-4 FROM dual;
SELECT GREATEST(sysdate, sysdate+5, sysdate-4) FROM dual;
SELECT LEAST(sysdate, sysdate+5, sysdate-4) FROM dual;
SQL> SELECT sysdate, sysdate+5, sysdate-4 FROM dual;
SYSDATE      SYSDATE+5 SYSDATE-4
-----  -----
12-DEC-11 17-DEC-11 08-DEC-11
SQL> SELECT GREATEST(sysdate, sysdate+5, sysdate-4) FROM dual;
GREATEST(
-----
17-DEC-11
SQL> SELECT LEAST(sysdate, sysdate+5, sysdate-4) FROM dual;
LEAST(SYS
-----
08-DEC-11

```

### **✓ CHECK 5C**

1. Using a select statement, display the age (make everyone six months older) in years.

*"It is not enough to succeed. Others must fail."*

## **5.15 NVL and NVL2 Functions**

You can use the NVL function to address problems caused when performing arithmetic operations with fields that might contain NULL values. When a NULL value is used in a calculation, the result is always a NULL value. The NVL function is used to substitute a value for the existing NULL so that the calculation can be completed. The syntax of the NVL function is NVL(x, y), where y represents the value to substitute if x is NULL. In many cases, the substitute for a NULL value in a calculation is zero (0).

### ***Example 5.16a (NVL)***

```

DELETE FROM patient;
INSERT INTO patient values (211,'john','Doe','m','11-FEB-1978',NULL,
'Davis','CA');
INSERT INTO patient values (219,'Billy','smith','f','19-FEB-1998',2000,
'Sacramento','CA');

SELECT fname, salary FROM patient;

--Every salary that is NULL is replaced with zero.
SELECT fname, NVL (salary,0)  FROM patient;

--Every salary that is NULL is replaced by the text (poor). Notice that to_char is used
--because both arguments have to be the same type (text).
SELECT fname, NVL(TO_CHAR(salary),'poor') "SALary"  FROM patient;

```

```

SQL> DELETE FROM patient;
1 row deleted.

SQL> INSERT INTO patient values (211,'john','Doe','m','11-FEB-1978',NULL, 'Davis
 ','CA');

1 row created.

SQL> INSERT INTO patient values (219,'Billy','smith','f','19-FEB-1998',2000, 'Sa
 cramento','CA');

1 row created.

SQL> SELECT fname, salary FROM patient;
FNAME           SALARY
-----
john
Billy          2000

SQL> --Every salary that is null is replaced with zero
SQL> SELECT fname, NVL(salary,0) FROM patient;
FNAME           NVL(SALARY,0)
-----
john
Billy          0
                2000

SQL> --Every salary that is null is replaced by the text (poor). Notice that to_
char is used
SQL> --because both arguments have to be the same type (text)
SQL> SELECT fname, NVL(TO_CHAR(salary),'poor') "SALary" FROM patient;
FNAME           SALArY
-----
john
Billy          poor
                2000

```

### ***Example 5.16a (NVL2)***

The NVL function is a variation of the NVL function with different options based on whether a NULL value exists. The syntax of the NVL2 function is NVL2( x, y, z), where y represents what should be substituted if x isn't NULL, and z represents what should be substituted if x is NULL. This variation gives you a little more flexibility when working with NULL values.

--NVL2 provides a value if it is not NULL and also provides a value if it is NULL. In this case,  
--poor if there is data and rich if it is NULL.

```

SELECT fname, salary, NVL2(TO_CHAR(salary), 'poor', 'rich')   FROM patient;
SQL> SELECT fname, salary, NVL2(TO_CHAR(salary), 'poor', 'rich')   FROM patient;
FNAME           SALARY NVL2
-----
john
Billy          rich
                2000 poor

```

## 5.16 DECODE

The DECODE function takes a specified value and compares it to values in a list. If a match is found, the specified result is returned. If no match is found, a default result is returned. If no default result is defined, a NULL is returned. The syntax of the DECODE function is DECODE( V, L1, R1, L2, R2, ..., D), where V is the value you're searching for, L1 represents the first value in the list, R1 represents the result to return if L1 and V are equivalent, and so on, and D is the default result to return if no match is found.

```

INSERT INTO patient values (311,'Jakey','Doey','U','11-FEB-1978',NULL,
'Davis','CA');

--Notice there are an odd number of arguments.
--If gender='m' then
--    display Male
--else if gender='f' then
--    display Female
SELECT gender, DECODE (gender,'m','MALE','f','FEMALE') FROM patient;

--Notice there are an even number of arguments. The last argument pertains to the else condition.
--If gender='m' then
--    display Male
--else if gender='f' then
--    display Female
-- else
--    display unknown
SELECT gender, DECODE (gender,'m','MALE','f','FEMALE','UNKNOWN') FROM patient;

SQL> INSERT INTO patient values (311,'Jakey','Doey','U','11-FEB-1978',NULL, 'Dav
is','CA');

1 row created.

SQL> --Notice there are an odd number of arguments
SQL> --If gender='m' then
SQL> --    display Male
SQL> --else if gender='f' then
SQL> --    display Female
SQL> SELECT gender, DECODE (gender,'m','MALE','f','FEMALE') FROM patient;

G DECODE
-----
m MALE
f FEMALE
U

SQL>
SQL> --Notice there are an even number of arguments. The last argument pertains
to the else condition
SQL> --If gender='m' then
SQL> --    display Male
SQL> --else if gender='f' then
SQL> --    display Female
SQL> -- else
SQL> --    display unknown
SQL> SELECT gender, DECODE (gender,'m','MALE','f','FEMALE','UNKNOWN') FROM pati
ent;

G DECODE(
-----
m MALE
f FEMALE
U UNKNOWN

```

## 5.17 SIGN

The syntax for the sign function is: sign (number)

number is the number to test for its sign.

If number < 0, then sign returns -1.

If number = 0, then sign returns 0.

If number > 0, then sign returns 1.

```

DELETE FROM patient;
INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000,
'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',500000,'Reno','NV');

SELECT salary, DECODE(SIGN(salary-40000),0,'Good money',-1,'Need
more','Donate') FROM patient;

SELECT salary, DECODE(SIGN(salary-40000),0,'Good money',-1,'Need
more',1,'Donate') FROM patient;

SQL> DELETE FROM patient;
3 rows deleted.

SQL> INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000, 'Davi
s','CA');

1 row created.

SQL> INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000, 'Da
vis','CA');

1 row created.

SQL> INSERT INTO patient values (113,'jill','Crane','m','12-APR-1999',500000,'Re
no','NV');

1 row created.

SQL> SELECT salary, DECODE(SIGN(salary-40000),0,'Good money',-1,'Need more','Don
ate') FROM patient;
      SALARY DECODE(SIG
-----
 25000 Need more
 40000 Good money
500000 Donate

SQL> SELECT salary, DECODE(SIGN(salary-40000),0,'Good money',-1,'Need more',1,'D
onate') FROM patient;
      SALARY DECODE(SIG
-----
 25000 Need more
 40000 Good money
500000 Donate

```

## 5.18 CASE

CASE expression syntax is similar to an IF-THEN-ELSE statement. Oracle checks each condition starting from the first condition (left to right). When a particular condition is satisfied (WHEN part) the expression returns the tagged value (THEN part). If none of the conditions are matched, the value mentioned in the ELSE part is returned. The ELSE part of the expression is not mandatory. CASE expression will return NULL if nothing is satisfied.

```
case when <condition> then <value>
      when<condition> then <value>
      ...
      else<value>
end
```

```
SELECT fname, salary,
       CASE WHEN salary<40000 THEN 'Need More'
            WHEN salary=40000 THEN 'Okay'
            ELSE 'Donate'
       END
  FROM patient;
```

```
SQL> SELECT fname, salary, CASE WHEN salary<40000 THEN 'Need More'
2                               WHEN salary=40000 THEN ,
3                               ELSE 'Donate'
4
5  FROM patient;
```

FNAME	SALARY CASEWHENS
john	25000 Need More
john	40000 Okay
jill	500000 Donate

## 5.19 TO\_NUMBER

The TO\_NUMBER function converts a value to a numeric data type, if possible. For example, the string value 2009 stored in a date or character string could be converted to a numeric data type to use in calculations. If the string being converted contains non-numeric characters, the function returns an error. It just so happens that in the following cases, it does not matter if a conversion is made because ORACLE is smart enough to make the conversion by itself.

```
SELECT '2009' * 2 FROM DUAL;
SELECT TO_CHAR(2009) * 2 FROM DUAL;
SELECT TO_NUMBER('2009') * 2 FROM DUAL;

SQL> SELECT '2009' * 2 FROM DUAL;
      '2009'*2
-----
      4018

SQL> SELECT TO_CHAR(2009) * 2 FROM DUAL;
TO_CHAR(2009)*2
-----
      4018

SQL> SELECT TO_NUMBER('2009') * 2 FROM DUAL;
TO_NUMBER('2009')*2
-----
      4018
```

## ✓ CHECK 5D

- 4) Display the lastname and salary for each person. If the salary is blank, display poor otherwise display the salary.
  - Use the NVL, NVL2, DECODE
- 5) What is the purpose of the dual table?

*"A hungry man is an angry man."*

## Summary Examples

```

/*Concatentatoin can be done using the pipe symbols or the concat function. Also, an alias can be
assigned with or without the AS keyword. Aliases that are comprised of multiple words must be enclosed
in double quotes.*/
SELECT fname || lname fullname, concat(fname, lname) AS "Some heading" FROM
patient;

--Can trim specified characters (space if none specified) from either the left or the right hand side.
SELECT ltrim(fname, '-'), rtrim(fname), ltrim(rtrim(fname)), trim(fname),
lpad(fname, 10, '-') FROM patient;

/* Months_between can subtract two dates. Sysdate is the current date and time. When adding a
number to a date, it adds days. Can also add months using add_months. Next day gives the next given
day, starting with sunday as being 1, monday=2 ... Can also use the round and trunc functions with these
date functions.*/
SELECT trunc(months_between(SYSDATE, DOB)), next_day(DOB, 3), DOB+15,
add_months(DOB, 10) FROM patient;

--Suppresses duplicates. Distinct has to be applied to all the columns between the select and from.
SELECT DISTINCT lname, fname FROM patient;

--Different math functions and operators --ceil rounds up, floor rounds down, trunc gets rid of precision.
SELECT mod(salary, 2), abs(salary), sqrt(salary), salary*10, salary-10,
salary/10, round(salary), floor(salary), ceil(salary), trunc(salary) FROM
patient;

--Dual table contains only one row and is used to test functions or retrieve just single row results.
--greatest gives the largest value in the set whereas least does the opposite.
SELECT greatest(10, 20, 30), least(20, 30, 40) FROM dual;

--NVL:if the column contains a NULL then it is replaced with the second argument.
--NVL2: if the column contains data then display the second argument else the third argument.
SELECT nvl(salary, 0), nvl2(salary, 10, 20), nvl(fname, to_char(salary)) FROM
patient;

--The sign function returns 1, -1, 0.
--The decode is like an if/else if. In this case if the first argument is 0 then good, if 1 then bad, if -1 then
--ugly
SELECT decode(sign(salary, 1000), 0, 'good', 1, 'bad', -1, 'ugly') FROM patient;

--Case is an easier way of dealing with conditions.
SELECT CASE WHEN salary>1000 THEN 'more'
            WHEN salary<1000 THEN 'bad'
            ELSE 'who cares'
        END
From patient

/* Want to extract the first name. This includes leading spaces.
SELECT    SUBSTR(name, INSTR(name, ',')+1) FROM patient;

```



*In 1998, Ellison and Sayonara won the Sydney to Hobart race, overcoming near-hurricane winds that sank five other boats, drowning six participants. Ellison is a principal supporter of the BMW Oracle Racing team, which has been a significant force in America's Cup competition. His yacht, Rising Sun, over 450 feet long, is one of the largest privately owned vessels in the world.*

## **Chapter 6 (Restricting)**

*"If we made it illegal, do you think more people would vote?"*

Element	Description
WHERE clause	Used to specify conditions that must be true for a record to be included in query results.
Mathematical comparison operators (=, <, >, <=, >=, <>, !=, ^=	How a record relates to a specific value
BETWEEN, AND, IN, OR, NOT, LIKE, IS NULL	Searching for values that include patterns, ranges or NULL values
AND, OR, NOT	Used to join multiple conditions or reverse the meaning of a search condition through NOT

```

SELECT [DISTINCT | UNIQUE] (*, columnname [ AS alias], ... )
      FROM tablename
      [WHERE condition]
      [GROUP BY group_by_expression]
      [HAVING group_condition]
      [ORDER BY columnname];

```

A condition identifies what must exist or a requirement that must be met for a record to be included in the results. Oracle searches through each record to determine whether the condition is TRUE. If a record meets the condition, it's returned in the query results

## 6.1 Numeric versus Text

When you use a string literal as part of a search condition, the value must be enclosed in single quotation marks and, as a result, is interpreted exactly as listed. By contrast, if the field referenced in a condition consists only of numbers, singlequotation marks aren't required.

```

DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR,
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state      VARCHAR2(20)
);
INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000,
'Davis','CA');

```

```
INSERT INTO patient values (113,'jill','Crane','m','12-APR-1999',NULL,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-1985',60000,'Las Vegas','NV');
INSERT INTO patient VALUES (115,'dove','Grime','f','04-JUN-1960',20000,'Sacramento','CA');
```

--All folks whose salaries is greater than 30000.

```
SELECT fname, lname, salary FROM patient WHERE salary >30000;
```

--All folks whose salaries is less than or equal to 30000.

```
SELECT fname, lname, salary FROM patient WHERE salary <=30000;
```

--All folks whose salaries is not equal 30000. For not equal, use != or <>.

```
SELECT fname, lname, salary FROM patient WHERE salary <>30000;
```

```
SQL> --All folks whose salaries is greater than 30000
SQL> SELECT fname, lname, salary FROM patient WHERE salary >30000;
```

FNAME	LNAME	SALARY
john	Smith	40000
billy	Bob	60000

SQL> --All folks whose salaries is less than or equal to 30000

```
SQL> SELECT fname, lname, salary FROM patient WHERE salary <=30000;
```

FNAME	LNAME	SALARY
john	Doe	25000
dove	Grime	20000

SQL> --All folks whose salaries is not equal 30000. For not equal can use != or <>

```
SQL> SELECT fname, lname, salary FROM patient WHERE salary <>30000;
```

FNAME	LNAME	SALARY
john	Doe	25000
john	Smith	40000
billy	Bob	60000
dove	Grime	20000

--When filtering for textual data, the contents are case sensitive.

```
SELECT fname, lname, salary , city FROM patient WHERE city='Davis' ;
```

```
SQL> --when filtering for textual data, the contents are case sensitive
SQL> SELECT fname, lname, salary , city FROM patient WHERE city='Davis' ;
```

FNAME	LNAME	SALARY	CITY
john	Doe	25000	Davis
john	Smith	40000	Davis

```
SELECT fname, lname, salary , city FROM patient WHERE city> 'Davis' ;
SELECT fname, lname, salary, city FROM patient WHERE city<>'Davis';
SELECT fname, lname, salary, city FROM patient WHERE city!= 'Davis' ;
```

```
SQL> SELECT fname, lname, salary , city FROM patient WHERE city>'Davis' ;
```

FNAME	LNAME	SALARY	CITY
jill	Crane		Reno
billy	Bob	60000	Las Vegas
dove	Grime	20000	Sacramento

```
SQL> SELECT fname, lname, salary, city FROM patient WHERE city<>'Davis';
```

FNAME	LNAME	SALARY	CITY
jill	Crane		Reno
billy	Bob	60000	Las Vegas
dove	Grime	20000	Sacramento

```
SQL> SELECT fname, lname, salary, city FROM patient WHERE city!= 'Davis' ;
```

FNAME	LNAME	SALARY	CITY
jill	Crane		Reno
billy	Bob	60000	Las Vegas
dove	Grime	20000	Sacramento

## 6.2 Dates

Sometimes you need to use a date as a search condition. Oracle displays dates in the default format DD-MON-YY, with MON being the standard three-letter abbreviation for the month. Because the date field contains letters and hyphens, it's not considered a numeric value when Oracle performs searches. Therefore, a date value must be enclosed in single quotation marks.

--When filtering for dates, can use the normal operators.

```
SELECT fname, lname,DOB, salary FROM patient WHERE DOB>'11-FEB-1978';
```

--If the date format is not default, then to\_date has to be used.

```
SELECT fname, lname,DOB, salary FROM patient WHERE
DOB>TO_DATE('02/11/1978','mm/dd/yyyy');
```

SQL> --When filtering for dates, can use the normal operators;

```
SQL> SELECT fname, lname,dob, salary FROM patient WHERE dob>'11-FEB-1978';
```

FNAME	LNAME	DOB	SALARY
john	Smith	01-MAR-81	40000
jill	Crane	12-APR-99	
billy	Bob	05-MAY-85	60000

SQL> --If the date format is not default, then to\_date has to be used,

```
SQL> SELECT fname, lname,dob, salary FROM patient WHERE dob>TO_DATE('02/11/1978
','mm/dd/yyyy');
```

FNAME	LNAME	DOB	SALARY
john	Smith	01-MAR-81	40000
jill	Crane	12-APR-99	
billy	Bob	05-MAY-85	60000

## 6.3 LOWER, UPPER

The upper function converts all letters in the specified string to uppercase. If there are characters in the string that are not letters, they are unaffected by this function. The syntax for the upper function is: upper( string1 )string1 is the string to convert to uppercase.

The lower function converts all letters in the specified string to lowercase. If there are characters in the string that are not letters, they are unaffected by this function. The syntax for the lower function is: lower(string1) string1 is the string to convert to lowercase.

```
--These examples are to indicate that the data in the table is case sensitive.
SELECT fname, lname,city FROM patient WHERE city='davis';
SELECT fname, lname,city FROM patient WHERE city='Davis';

--To bypass case-sensitivity, use the upper or lower function. When using
--upper, make sure that it is being compared with something that is in upper case and the
--opposite when checking for lower case.
SELECT fname, lname,city FROM patient WHERE UPPER(city)='DAVIS';
SELECT fname, lname,city FROM patient WHERE LOWER(city)='davis';

SQL> --These examples are to indicate that the data in the table is case sensitive.
SQL> SELECT fname, lname,city FROM patient WHERE city='davis';
no rows selected
SQL> SELECT fname, lname,city FROM patient WHERE city='Davis';
FNAME          LNAME          CITY
-----          -----
john           Doe            Davis
john           Smith          Davis

SQL> --To bypass case-sensitivity, use the upper or lower function. When using
SQL> --upper, make sure that it is being compared with something that is in upper
case and the
SQL> --opposite when checking for lower case
SQL> SELECT fname, lname,city FROM patient WHERE UPPER(city)='DAVIS';
FNAME          LNAME          CITY
-----          -----
john           Doe            Davis
john           Smith          Davis

SQL> SELECT fname, lname,city FROM patient WHERE LOWER(city)='davis';
FNAME          LNAME          CITY
-----          -----
john           Doe            Davis
john           Smith          Davis
```

## 6.4 AND, BETWEEN

### *Example 6.4a (The and clause)*

At times, you need to search for records based on two or more conditions. In these situations, you can use logical operators to combine search conditions. The logical operators AND and OR are commonly used for this purpose. ( The NOT operator is also a logical operator in Oracle , but it's used to reverse the meaning of search conditions rather than combine them.) Keep in mind that

when a query executes, records can be filtered with WHERE clause conditions. If the condition is TRUE when compared with a record, the record is included in the results. When the AND operator is used in the WHERE clause, both conditions combined by the AND operator must be evaluated as TRUE, or the record isn't included in the results.

--Can use the and operator to check for two conditions. Checks for a range in this case.

```
SELECT fname, lname, DOB, salary FROM patient WHERE salary>30000 and salary <80000;
```

```
SELECT fname, lname, DOB, salary FROM patient WHERE DOB>'11-FEB-1978' and DOB<'15-MAR-1990';
```

```
SELECT fname, lname, DOB, salary, city FROM patient WHERE salary>30000 and LOWER(city)='davis';
```

```
SELECT fname, lname, DOB, salary, city FROM patient WHERE LOWER(city)='davis' and LOWER(city)='reno';
```

--What is wrong?? Notice that the ranges overlap which makes it logically invalid.

```
SELECT fname, lname, DOB, salary FROM patient WHERE salary>30000 and salary >80000;
```

SQL> --can use the and operator to check for two conditions. Checks for a range in this case

```
SQL> SELECT fname, lname, dob, salary FROM patient WHERE salary>30000 and salary <80000;
```

FNAME	LNAME	DOB	SALARY
john	Smith	01-MAR-81	40000
billy	Bob	05-MAY-85	60000

```
SQL> SELECT fname, lname, dob, salary FROM patient WHERE dob>'11-FEB-1978' and dob<'15-MAR-1990';
```

FNAME	LNAME	DOB	SALARY
john	Smith	01-MAR-81	40000
billy	Bob	05-MAY-85	60000

SQL>

```
SQL> SELECT fname, lname, dob, salary, city FROM patient WHERE salary>30000 and LOWER(city)='davis';
```

FNAME	LNAME	DOB	SALARY
CITY			
john	Smith	01-MAR-81	40000
Davis			

```
SQL> SELECT fname, lname, dob, salary, city FROM patient WHERE LOWER(city)='davis' and LOWER(city)='reno';
```

no rows selected

SQL> --What is wrong?? Notice that the ranges overlap which makes it logically invalid

```
SQL> SELECT fname, lname, dob, salary FROM patient WHERE salary>30000 and salary >80000;
```

no rows selected

### **Example 6.4b (Between)**

The two values defining the range for SQL BETWEEN clause can be dates, numbers or just text. (Inclusive of the end points)

```
--When looking for a range using the and clause, if it is inclusive of the end points then it
-- is easier to use the between clause instead.
SELECT fname, lname, DOB, salary FROM patient WHERE salary>=30000 AND
salary <=80000;

SELECT fname, lname, DOB, salary FROM patient WHERE salary BETWEEN 30000
AND 80000;

SQL> --When looking for a range using the and clause, if it is inclusive of the
end points then it is easier
SQL> --to use the between clause.
SQL> SELECT fname, lname, dob, salary FROM patient WHERE salary>=30000 AND salary <=80000;
FNAME          LNAME        DOB          SALARY
-----          -----        -----          -----
john           Smith       01-MAR-81    40000
billy          Bob         05-MAY-85    60000

SQL> SELECT fname, lname, dob, salary FROM patient WHERE salary BETWEEN 30000 AND 80000;
FNAME          LNAME        DOB          SALARY
-----          -----        -----          -----
john           Smith       01-MAR-81    40000
billy          Bob         05-MAY-85    60000
```

## 6.5 OR, IN

### **Example 6.5a (Or, In)**

The IN operator allows you to specify multiple values in a WHERE clause.

--If the same column is being checked against multiple values, then the IN clause is an easier route.

```
SELECT fname, lname, city FROM patient WHERE city='Davis' OR
city='Sacramento' OR city='Chico';
```

--Same as above

```
SELECT fname, lname, city FROM patient WHERE city IN
('Davis', 'Sacramento', 'Chico');
```

SQL> --if the same column is being checked against multiple values, then the IN clause is an

SQL> --easier route

```
SQL> SELECT fname, lname, city FROM patient WHERE city='Davis' OR city='Sacramento' OR city='Chico';
```

FNAME	LNAME	CITY
john	Doe	Davis
john	Smith	Davis
dove	Grime	Sacramento

SQL> --Same as above

```
SQL> SELECT fname, lname, city FROM patient WHERE city IN ('Davis', 'Sacramento',
'Chico');
```

FNAME	LNAME	CITY
john	Doe	Davis
john	Smith	Davis
dove	Grime	Sacramento

```
SELECT fname, lname, salary FROM patient WHERE salary=10000 OR
salary=20000 OR salary=30000;
```

```
SELECT fname, lname, salary FROM patient WHERE salary
IN(10000,20000,30000);
```

--Negates the IN clause using the NOT operator which means it displays the record as  
--long as the salary is not one of the three items on the list.

```
SELECT fname, lname, salary FROM patient WHERE salary NOT
IN(10000,20000,30000);
```

FNAME	LNAME	CITY
john	Doe	Davis
john	Smith	Davis
dove	Grime	Sacramento

```
SQL> SELECT fname, lname, salary FROM patient WHERE salary=10000 OR salary=20000
OR salary=30000;
```

FNAME	LNAME	SALARY
dove	Grime	20000

```
SQL> SELECT fname, lname, salary FROM patient WHERE salary IN(10000,20000,30000)
;
```

FNAME	LNAME	SALARY
dove	Grime	20000

SQL> --Negates the IN clause using the NOT operator which means it displays the  
record as long

SQL> --as the salary is not one of the three items on the list

```
SQL> SELECT fname, lname, salary FROM patient WHERE salary NOT IN(10000,20000,30000);
```

FNAME	LNAME	SALARY
john	Doe	25000
john	Smith	40000
billy	Bob	60000

### ***Example 6.5b (Order of precedence)***

Next, take a look at the order of logical operators. Because the WHERE clause can contain multiple types of operators, you need to understand the order in which they're resolved:

- Arithmetic operations are solved first.
- Comparison operators are solved next.
- Logical operators have a lower precedence and are evaluated last in the order NOT, AND, and OR.

--The AND clause is going to be done first and then the OR clause. This means it first  
--looks at the salary range and comes back with a result. If the salary is in the range or  
--the city is Davis then the record is selected.

```
SELECT lname, DOB, salary, city FROM patient WHERE salary>=30000 AND
salary <=80000 OR city='Davis';
```

```

SQL> --the AND clause is going to be done first and then the OR clause. This means it first looks at the
SQL> --salary range and comes back with a result. If the salary is in the range or the city is Davis
SQL> --then the record is selected
SQL> SELECT lname, dob, salary, city FROM patient WHERE salary >=30000 AND salary <=80000 OR city='Davis';

LNAME          DOB          SALARY CITY
-----
Doe            11-FEB-78    25000 Davis
Smith          01-MAR-81    40000 Davis
Bob             05-MAY-85    60000 Las Vegas

```

## 6.6 ANY, ALL operator

Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.

### *Example 6.6a (Any )*

```

-- Since the values overlap, the following is logically not correct.
SELECT fname, lname, salary FROM patient WHERE salary > 30000 OR
salary>40000;

--This accomplishes the same thing as above.
SELECT fname, lname, salary FROM patient WHERE salary >ANY(30000,40000);
SQL> SELECT fname, lname, salary FROM patient WHERE salary > 30000 OR salary>40000;
FNAME          LNAME          SALARY
-----
john           Smith          40000
silly          Bob            60000

SQL> SELECT fname, lname, salary FROM patient WHERE salary >ANY(30000,40000);
FNAME          LNAME          SALARY
-----
john           Smith          40000
silly          Bob            60000

```

### *Example 6.6b (All )*

Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to TRUE if the query returns no rows.

```

--The salary has to be greater than both of those values. Once again is logically not correct.
SELECT fname, lname, salary FROM patient WHERE salary > 30000 AND
salary>40000;

--Same as above
SELECT fname, lname, salary FROM patient WHERE salary >ALL(30000,40000);

```

```

SQL> --The salary has to be greater than both of those values once again is logically not correct
SQL> SELECT fname, lname, salary FROM patient WHERE salary > 30000 AND salary > 40000;
FNAME          LNAME          SALARY
-----          -----
billy           Bob            60000

SQL> --Same as above
SQL> SELECT fname, lname, salary FROM patient WHERE salary > ALL(30000, 40000);
FNAME          LNAME          SALARY
-----          -----
billy           Bob            60000

```

## 6.7 Like Clause

The LIKE operator is unique, in that it's used with wildcard characters to search for patterns. Wildcard characters are used to represent one or more alphanumeric characters. The wildcard characters available for pattern searches in Oracle are the percent sign (%) and the underscore symbol (\_). The percent sign represents any number of characters (zero, one, or more), and the underscore symbol represents exactly one character.

```

DELETE FROM patient;
INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (113,'john','Doe','m','11-FEB-1978',25000,
'Mavis','CA');

--All the records where the city starts with the letter (D). % means any number of characters.
SELECT fname, lname, DOB, city FROM patient WHERE city LIKE 'D%';

--Must use the LIKE operator. If the = is used then it is looking for an exact match.
SELECT fname, lname, DOB, city FROM patient WHERE city = 'D%';

--All the records that contain a (D) somewhere in the city.
SELECT fname, lname, DOB, city FROM patient WHERE city LIKE '%D%';

SQL> --All the records where the city starts with the letter (D). % means any number of characters
SQL> SELECT fname, lname, dob, city FROM patient WHERE city LIKE 'D%';
FNAME          LNAME          DOB          CITY
-----          -----
john           Doe           11-FEB-78 Davis

SQL> --Must use the LIKE operator. If the = is used then it is looking for an exact match
SQL> SELECT fname, lname, dob, city FROM patient WHERE city = 'D%';
no rows selected

SQL> --all the records that contain a (D) somewhere in the city
SQL> SELECT fname, lname, dob, city FROM patient WHERE city LIKE '%D%';
FNAME          LNAME          DOB          CITY
-----          -----
john           Doe           11-FEB-78 Davis

```

--It must end with (S) regardless of whether it is upper or lower case (S).

```
SELECT fname, lname, DOB, city FROM patient WHERE UPPER(city) LIKE '%S';
```

--The second letter can be anything. (-) means a single character.

```
SELECT fname, lname, DOB, city FROM patient WHERE city LIKE 'D_vis';
```

--The exact opposite of the above when the NOT operator is used.

```
SELECT fname, lname, DOB, city FROM patient WHERE city NOT LIKE 'D vis';
```

SQL> --It must end with (S) regardless of whether it is upper or lower case (S)  
SQL> SELECT fname, lname, dob, city FROM patient WHERE UPPER(city) LIKE '%S';

FNAME	LNAME	DOB	CITY
john	Doe	11-FEB-78	Davis
john	Doe	11-FEB-78	Mavis

SQL> --The second letter can be anything. (-) means a single character  
SQL> SELECT fname, lname, dob, city FROM patient WHERE city LIKE 'D\_vis';

FNAME	LNAME	DOB	CITY
john	Doe	11-FEB-78	Davis

SQL> --The exact opposite of the above when the NOT operator is used  
SQL> SELECT fname, lname, dob, city FROM patient WHERE city NOT LIKE 'D\_vis';

FNAME	LNAME	DOB	CITY
john	Doe	11-FEB-78	Mavis

## 6.8 NULL

When performing arithmetic operations or search conditions, NULL values can cause unexpected results. A NULL value means no value has been stored in that field. Don't confuse a NULL value with a blank space. A NULL is the absence of data in a field; a field containing a blank space does contain a value—a blank space. When searching for NULL values, you can't use the equal sign because there's no value to use for comparison in the search condition. When checking for a NULL value, you're actually checking the status of the column: Does data exist or not? If you need to identify records that have a NULL value, you must use the IS NULL comparison operator.

```
INSERT INTO patient values (978, 'john', 'Doe', 'm', '11-FEB-1978', 25000,  
NULL, 'CA');
```

```
SELECT fname, lname, city FROM patient;
```

--Not correct. When checking for NULL , the (IS) keyword must be used.

```
SELECT fname, lname, city FROM patient WHERE city= NULL;
```

```
SQL> SELECT fname, lname, city FROM patient;
```

FNAME	LNAME	CITY
john	Doe	Davis
john	Doe	Mavis
john	Doe	

SQL> --Not correct. When checking for null , the (IS) keyword must be used  
SQL> SELECT fname, lname, city FROM patient WHERE city= NULL;

```
no rows selected
```

--Not correct: NULL cannot be enclosed in single quotes. In this case, it is looking for the words NULL as

--data in the table.

```
SELECT fname, lname, city FROM patient WHERE city='NULL';
SELECT fname, lname, city FROM patient WHERE city IS NULL;
```

--All the records that have something in their city column.

```
SELECT fname, lname, city FROM patient WHERE city IS NOT NULL;
```

--NULL is not displayed because it does not match the filtering criterion. It only looks at columns that have data and NULL means void of data. All these operators check data against data. To check against non-data, the (IS) operator must be used.

```
SELECT fname, lname, city FROM patient WHERE city <> 'Davis';
```

```
SQL> --Not correct: Null cannot be enclosed in single quotes. In this case it is looking for the words
SQL> --NULL as data in the table
SQL> SELECT fname, lname, city FROM patient WHERE city='NULL';
```

no rows selected

```
SQL> SELECT fname, lname, city FROM patient WHERE city IS NULL;
```

FNAME	LNAME	CITY
john	Doe	

```
SQL> --All the records that have something in their city column
SQL> SELECT fname, lname, city FROM patient WHERE city IS NOT NULL;
```

FNAME	LNAME	CITY
john	Doe	Davis
john	Doe	Mavis

SQL> --Null is not displayed because it does not match the filtering criterion. It only looks at columns

```
SQL> --that have data and null means void of data. All these operators check data against data. To
SQL> --check against non-data, the (IS) operator must be used
SQL> SELECT fname, lname, city FROM patient WHERE city <> 'Davis';
```

FNAME	LNAME	CITY
john	Doe	Mavis

```
SELECT fname, lname, city FROM patient WHERE city NOT IN ('Davis');
```

--NULL is displayed.

```
SELECT fname, lname, city FROM patient WHERE city <> 'Davis' OR city IS NULL;
```

```
SQL> SELECT fname, lname, city FROM patient WHERE city NOT IN ('Davis');
```

FNAME	LNAME	CITY
john	Doe	Mavis

SQL> --Null is displayed

```
SQL> SELECT fname, lname, city FROM patient WHERE city <> 'Davis' OR city IS NULL;
```

FNAME	LNAME	CITY
john	Doe	Mavis
john	Doe	

## ✓ CHECK 6A

1. Display all the records from the Person table whose age is between 30 and 35.
  - o Use AND and BETWEEN
2. Display all the individuals whose lastname is Fickle or Grapes, regardless of case.
  - o Use OR and IN
3. Display all those whose lastname starts with B and ends with L, regardless of case.
4. Display all those whose lastname contains the letter B in the third position.
5. Display poor if the salary is NULL, otherwise display the actual salary plus an additional \$2000.
  - o Use case
6. Display all the people whose salaries are not NULL.
7. Display all the people whose salary is NULL and the lastname contains a C regardless of case.

*“Be great in little things.”*

## 6.9 Creating tables with select statements

### *Example 6.9a (Creating table using select)*

```
--Can feed the results to another table instead of displaying it to the screen. The new
--table (patient_temp) will be comprised of fname, lname, salary. This new table will
--contain the data from the patient table.
--Keep in mind that the column names must be valid identifiers, which means that things
--such as calculations or function names which have parentheses in them will not be
--accepted. For this reason, an alias is used.
SELECT fname,city from patient;
```

FNAME	CITY
john	Davis
Jill	Mavis
Jack	

```
CREATE TABLE patient_temp AS SELECT fname, lname, salary, city FROM
patient WHERE city='Mavis';
```

```
SELECT * FROM patient_temp;
```

```
SQL> CREATE TABLE patient_temp AS SELECT fname, lname, salary, city FROM patient
WHERE city='Mavis';
Table created.
```

```
SQL> SELECT * FROM patient_temp;
```

FNAME	LNAME	SALARY	CITY
Jill	Doe	25000	Mavis

### **Example 6.9b (Creating tables and aliases)**

--Creating a table and using an alias name

```
DROP TABLE patient_temp;

--INVALID: have to use an alias.
CREATE TABLE patient_temp AS SELECT fname, lname, salary*2 FROM patient
WHERE salary >30000;

CREATE TABLE patient_temp AS SELECT fname, lname, salary*2 new_salary
FROM patient WHERE salary >30000;

--Notice the column name new_salary.
DESC patient_temp;

SQL> DROP TABLE patient_temp;
Table dropped.

SQL>
SQL> --INVALID: have to use an alias
SQL> CREATE TABLE patient_temp AS SELECT fname, lname, salary*2 FROM patient WHERE salary >30000;
CREATE TABLE patient_temp AS SELECT fname, lname, salary*2 FROM patient WHERE salary >30000
                                         *
ERROR at line 1:
ORA-00998: must name this expression with a column alias

SQL>
SQL> CREATE TABLE patient_temp AS SELECT fname, lname, salary*2 new_salary FROM patient WHERE salary >30000;
Table created.

SQL>
SQL> --Notice the column name new_salary
SQL> DESC patient_temp;
      Name          Null?    Type
-----  -----
FNAME           VARCHAR2(20)
LNAME           VARCHAR2(20)
NEW_SALARY      NUMBER
```

### **Example 6.9c (Creating table with constraints)**

In creating a copy, only the not NULL constraint gets copied over. Also the name of the constraint will be different for the new table

```
DROP TABLE test;
DROP TABLE test2;
COLUMN search_condition FORMAT A10
CREATE TABLE test
(
  col1 NUMBER PRIMARY KEY,
  col2 NUMBER UNIQUE,
  col3 NUMBER CHECK (col3>20),
  col4 NUMBER NOT NULL
);
```

```
--Only thing that gets transferred over when creating a table in this way is the NOT NULL constraint
CREATE TABLE test2 AS SELECT * FROM test;

--Confirms what was created. Not NULL constraint is a check constraint. Details about the check
--constraints are in the column search_condition.
SELECT constraint_type, table_name, search_condition FROM user_constraints
WHERE table_name IN ('TEST','TEST2');

DESC test;

DESC test2;

SQL> --The only thing that gets transferred over when creating a table in this w
ay is the NOT NULL
SQL> --constraint
SQL> CREATE TABLE test2 AS SELECT * FROM test;
Table created.

SQL> --Confirms what was created. Not null constraint is a check constraint. Det
ails about
SQL> --the check constraints are in the column search_condition
SQL> SELECT constraint_type, table_name, search_condition FROM user_constraints
WHERE table_name IN ('TEST','TEST2');

C TABLE_NAME          SEARCH_CON
----- -----
C TEST                "COL4" IS
                      NOT NULL
P TEST                col3>20
U TEST
C TEST2               "COL4" IS
                      NOT NULL

SQL> DESC test;
      Name           Null?    Type
----- -----
COL1            NOT NULL NUMBER
COL2            NOT NULL NUMBER
COL3            NOT NULL NUMBER
COL4            NOT NULL NUMBER

SQL> DESC test2;
      Name           Null?    Type
----- -----
COL1            NUMBER
COL2            NUMBER
COL3            NUMBER
COL4            NOT NULL NUMBER
```

## 6.10 Updating tables using the update statement

The UPDATE statement allows you to update a single record or multiple records in a table.

The syntax for the UPDATE statement is:

UPDATE table SET column = expression WHERE predicates;

### ***Example 6.10a (Updating)***

```

SELECT fname, lname, salary FROM patient where patient_id=111;

--Updates the patient table by setting the fname, lname and salary for patient (11).
UPDATE patient SET fname='Bill', lname='Bob', salary=10000 WHERE
patient_id=111;

--Confirm results.
SELECT fname, lname, salary FROM patient where patient_id=111;

SQL> SELECT fname, lname, salary FROM patient where patient_id=111;
FNAME          LNAME          SALARY
-----          -----          -----
john           Doe            50000
SQL> --Updates the patient table by setting the fname, lname and salary for patient (11)
SQL> UPDATE patient SET fname='Bill', lname='Bob', salary=10000 WHERE patient_id
=111;
1 row updated.

SQL> --Confirm results
SQL> SELECT fname, lname, salary FROM patient where patient_id=111;
FNAME          LNAME          SALARY
-----          -----          -----
Bill           Bob            10000

```

### ***✓ CHECK 6B***

1. Create a brand new table called Person2 which contains only those folks who are making less than 10000 and whose lastname ends with the letter 'j', regardless of case.
  
2. Update the salaries to 8000 for all the people in person2 who are making more than 20000 or are older than 40 years old.

*“Education is not the filling of a pail, but the lighting of a fire.”*

## Summary examples

--Use to\_lower or to\_upper to bypass the case of the data stored in the table.

--USE the IN clause to check for multiple values instead of a bunch of OR clauses.

--AND clauses are executed before OR clauses.

```
SELECT fname, lname, TO_DATE(DOB, 'mmddyy') FROM patient WHERE
LOWER(fname)='john' AND UPPER(lname) IN ('DOE','MO');
```

--NVL function is used to deal with NULL values. Both arguments have to be the same type. Since  
--salary is a number, it is first converted to a character to match with the second argument.

--The LIKE operator is used for wildcard characters. % means any number of characters whereas \_  
--represents only a single character in that specific position. This would mean that we don't care  
--about the first character, the second character must be an (h) and beyond that we don't care. Also  
--by using to\_lower we are bypassing the case.

```
SELECT NVL(TO_CHAR(salary), 'poor') FROM patient WHERE LOWER(lname) LIKE '_h%';
```

--NVL2 has three arguments: If it is not NULL use the second argument. If it is NULL use the third argument.

--When checking for NULL, the IS or IS NOT operator has to be used. Can use MONTHS\_BETWEEN

--to find the span of months between two dates.

```
SELECT NVL2(TO_CHAR(salary), 'RICH', 'Poor') FROM patient WHERE lname IS NOT NULL
AND MONTHS_BETWEEN(SYSDATE, DOB)>150;
```

--Can direct the output to a new table in this case somepatients. Since the column names for the  
--new table come from the information between the SELECT and FROM, aliases have to be used  
--when special symbols are being used.

```
CREATE TABLE somepatients AS SELECT fname||lname AS fullname, salary*2
double_salary FROM patient WHERE salary BETWEEN 10000 AND 2000;
```

--Can use the update statement to update certain records.

```
UPDATE patient SET salary=20000 WHERE salary IS NULL;
```



*After many years of pursuing a victory in the America's Cup yacht race, Ellison triumphed at last in 2010.*

## Chapter 7 (SORTING)

Employee Id	Name	Department Id
100	MILLS	20
140	KATE	10
120	JOHN	30
110	KING	10
150	RYAN	20
130	NEO	10

Employee Id	Name	Department Id
100	NEO	10
110	KATE	10
120	KING	10
130	MILLS	20
140	RYAN	20
150	JOHN	30



*"Classic is a book which people praise, but do not read."*

### Example 7a (Order by)

The ORDER BY clause, used to display query results in a sorted order, is listed at the end of the SELECT statement. The columns used to sort the results are listed in the ORDER BY clause. In the query results, the second column (Name) is listed in ascending alphabetical order. Note these important points:

```
DELETE FROM patient;
INSERT INTO patient values (111,'john','Wei','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-1985',60000,'Las
Vegas','NV');
INSERT INTO patient values (115,'dove','Grime','f','04-JUN-
1960',20000,'Sacramento','CA');
INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000,
'Davis','CA');
INSERT INTO patient values (978,'john','Doe','m','11-FEB-
1978',25000,NULL,'CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',50000,'Reno','NV');
```

--Default sort is in ascending order.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city;
```

--ASC is implied by default.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city ASC;
```

--Can sort in descending order.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city DESC;
```

SQL> --Default sort is in ascending order

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

SQL> --ASC is implied by default

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city ASC;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

SQL> --Can sort in descending order

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city DESC
;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
978	john	Doe	Sacramento
115	dove	Grime	Reno
113	jill	Crane	Las Vegas
114	billy	Bob	Davis
112	john	Smith	Davis
111	john	Wei	

6 rows selected.

--Can sort by the position of the column between the select and from clause.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY 4;
```

--Can use the alias name for sorting.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY CITYNAME;
```

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY 4;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

--can use the alias name for sorting

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY CITYNAME;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

### **Example 7b (NULLs)**

When sorting in ascending order, values are listed in this order:

1. Blank and special characters
2. Numeric values
3. Character values ( uppercase first)
4. NULL values

Unless you specify “ DESC” for descending, the ORDER BY clause sorts in ascending order by default. If a column alias is given to a field in the SELECT clause, you can reference the field in the ORDER BY clause with the column alias— although doing so isn’t required. You can also use the ORDER BY clause with the optional NULLS FIRST or NULLS LAST keywords to change the order for listing NULL values. By default, NULL values are listed last when results are sorted in ascending order and first when they’re sorted in descending order.

--NULLs appear last by default. You can change the default to make them appear first.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city
NULLS FIRST;
```

--NULLs appear last which is implied.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city
NULLS LAST;
```

SQL> --Nulls by default appear last. You can change the default to make them appear first  
SQL> SELECT patient\_id, fname, lname, city CITYNAME FROM patient ORDER BY city NULLS FIRST;

PATIENT_ID	FNAME	LNAME	CITYNAME
978	john	Doe	
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento

6 rows selected.

SQL> --Nulls appear last which is implied

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY city NULLS LAST;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

SQL> --Notice that "Nothing" appears at the end

```
SQL> SELECT patient_id, fname, lname, nvl(city, 'Nothing') CITYNAME FROM patient ORDER BY city;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	Nothing

6 rows selected.

### ***Example 7c (Secondary sorts)***

In the previous examples, only one column was specified in the ORDER BY clause, which is called a primary sort. In some cases, you might want to include a secondary sort, which specifies a second field to sort by if an exact match occurs between two or more rows in the primary sort.

--Orders by fname. When it comes across fnames that are the same, then it further sorts

--by lname. Both sorts are in ascending order.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY fname,
lname;
```

--Can make fname in descending and lname in ascending order which is default.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY fname
DESC, lname;
```

SQL> --Orders by fname. When it comes across fnames that are the same, then it further sorts by

--lname. Both sorts are in ascending order

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY fname, lname;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
114	billy	Bob	Las Vegas
115	dove	Grime	Sacramento
113	jill	Crane	Reno
978	john	Doe	
112	john	Smith	Davis
111	john	Wei	Davis

6 rows selected.

SQL> --Can make fname in descending and lname in ascending order which is default

```
SQL> SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY fname DESC, lname;
```

PATIENT_ID	FNAME	LNAME	CITYNAME
978	john	Doe	
112	john	Smith	Davis
111	john	Wei	Davis
113	jill	Crane	Reno
115	dove	Grime	Sacramento
114	billy	Bob	Las Vegas

6 rows selected.

### **Example 7d (Position)**

Oracle also provides an abbreviated method for referencing the sort column if the name is used in the SELECT clause. In the previous example, State and City are used in both the SELECT and ORDER BY clauses. Instead of listing these column names again in the ORDER BY clause, you can reference them by their positions in the SELECT clause's column list. You can also use the column alias.

--Can use the order by on the actual column, the position of where it appears between

--the select and from or by the alias.

```
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY 4;
SELECT patient_id, fname, lname, city CITYNAME FROM patient ORDER BY CITYNAME;
```

SQL> --Can use the order by on the actual column, the position of where it appears between the  
 SQL> --select and from or by the alias  
 SQL> SELECT patient\_id,fname,lname,CITYNAME FROM patient ORDER BY 4;

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

SQL> SELECT patient\_id,fname,lname,CITYNAME FROM patient ORDER BY CITYNAME;

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
113	jill	Crane	Reno
115	dove	Grime	Sacramento
978	john	Doe	

6 rows selected.

--Notice that "Nothing" appears at the end.

SELECT patient\_id,fname,lname,nvl(city,'Nothing') CITYNAME FROM patient ORDER BY city;

SQL> --Notice that "Nothing" is sorted in the result set  
 SQL> SELECT patient\_id,fname,lname,nvl(city,'Nothing') CITYNAME FROM patient ORDER BY CITYNAME;

PATIENT_ID	FNAME	LNAME	CITYNAME
111	john	Wei	Davis
112	john	Smith	Davis
114	billy	Bob	Las Vegas
978	john	Doe	Nothing
113	jill	Crane	Reno
115	dove	Grime	Sacramento

6 rows selected.

### ***Example 7e (column versus alias)***

Difference in ordering by the column versus by the alias

--In this dataset there is one record that has a NULL for salary.

--Order by alias after zero has been substituted.

SELECT NVL(salary,0) pay FROM patient ORDER BY pay;

--Order by position after zero has been substituted.

SELECT NVL(salary,0) FROM patient ORDER BY 1;

```

SQL> --In this dataset there is one record that has a null for salary
SQL> --Order by alias after zero has been substituted
SQL> SELECT NVL(salary,0) pay FROM patient ORDER BY pay;

    PAY
-----
-20000
      0
  25000
  25000
  40000
  60000

6 rows selected.

SQL> --Order by position after zero has been substituted
SQL> SELECT NVL(salary,0) FROM patient ORDER BY 1;

NVL(SALARY,0)
-----
-20000
      0
  25000
  25000
  40000
  60000

6 rows selected.

```

--Order after the zero has been substituted.

```
SELECT NVL(salary,0) FROM patient ORDER BY NVL(salary,0);
```

--Order before the zero has been substituted, which means the NULLs appear at the end by default.

```
SELECT NVL(salary,0) FROM patient ORDER BY salary;
```

SQL> --Order after the zero has been substituted

```
SQL> SELECT NVL(salary,0) FROM patient ORDER BY NVL(salary,0);
```

NVL(SALARY,0)

```
-----
-20000
      0
  25000
  25000
  40000
  60000
```

6 rows selected.

SQL> --Order before the zero has been substituted which means the nulls appear at the end by default

```
SQL> SELECT NVL(salary,0) FROM patient ORDER BY salary;
```

NVL(SALARY,0)

```
-----
-20000
  25000
  25000
  40000
  60000
      0
```

6 rows selected.

## ✓ CHECK 7A

1. Display the fname and lname (use alias lastname) of the people whose salaries are NULL. Do a sort within a sort using lastname and firstname.
  - o Use the column names (Ascending order)
  - o Use the position of the columns (Descending order)
  - o Use the alias (ascending) and column name (descending)

*"What man does not understand, he fears; and what he fears, he tends to destroy."*

## Summary Examples

--Can have sorts within sorts.  
 --Can use a number to identify the position of the column between SELECT and FROM.  
 --Can use the computed column, as in salary \*2.  
 --Can use the alias.  
 --Default is ASCENDING but can be changed to DESCENDING.  
 --Also NULLs appear last but the order can be changed.

```
SELECT fname, salary *2, DOB date_of_birth FROM patient
ORDER BY 1 DESC, salary*2 NULLSFIRST, date_of_birth ASC;
```

--Order by position after zero has been substituted. If the number one is replaced with  
 --salary then the sorting order will be based on the actual contents in the table and all  
 --NULLs will be displayed last.

```
SELECT NVL(salary,0) FROM patient ORDER BY 1;
```



*In April 2009, Oracle announced its intent to buy Sun Microsystems after a tug of war with IBM and Hewlett-Packard.[10] The European Union approved the acquisition by Oracle of Sun Microsystems on January 21, 2010 and agreed that "Oracle's acquisition of Sun has the potential to revitalize important assets and create new and innovative products*

## Chapter 8 (GROUP BY)

---

Albuquerque (6)	\$147.26	ordered on	8/30/1996 by Rattlesnake Canyon Grocery
	\$150.15	ordered on	9/27/1996 by Rattlesnake Canyon Grocery
	\$142.08	ordered on	11/5/1996 by Rattlesnake Canyon Grocery
	\$708.95	ordered on	3/19/1997 by Rattlesnake Canyon Grocery
	\$174.05	ordered on	1/26/1998 by Rattlesnake Canyon Grocery
	\$280.61	ordered on	2/16/1998 by Rattlesnake Canyon Grocery
Anchorage (4)	\$257.62	ordered on	9/13/1996 by Old World Delicatessen
	\$135.63	ordered on	10/16/1997 by Old World Delicatessen
	\$170.97	ordered on	1/27/1998 by Old World Delicatessen
	\$144.38	ordered on	3/20/1998 by Old World Delicatessen
Boise (20)	\$214.27	ordered on	10/8/1996 by Save-a-lot Markets
	\$126.56	ordered on	12/25/1996 by Save-a-lot Markets
	\$140.26	ordered on	2/20/1997 by Save-a-lot Markets
	\$367.63	ordered on	4/18/1997 by Save-a-lot Markets
	\$252.49	ordered on	6/2/1997 by Save-a-lot Markets
	\$200.24	ordered on	7/22/1997 by Save-a-lot Markets
	\$544.08	ordered on	7/28/1997 by Save-a-lot Markets
	\$107.46	ordered on	8/11/1997 by Save-a-lot Markets
	\$352.69	ordered on	9/4/1997 by Save-a-lot Markets

---

*"Conference: The confusion of one man multiplied by the number present."*

Group functions, also called multiple- row functions, return one result per group of rows processed. Multiple- row functions covered in this chapter include SUM, AVG, COUNT, MIN, and MAX. This chapter also explains using the GROUP BY clause to identify groups of records to process and the HAVING clause to restrict groups returned in the query results.

When using the GROUP BY clause, remember the following:

- If a group function is used in the SELECT clause, any single ( non-aggregate) columns listed in the SELECT clause must also be listed in the GROUP BY clause.
- Columns used to group data in the GROUP BY clause don't have to be listed in the SELECT clause. They're included in the SELECT clause only to have these groups identified in the output. • Column aliases can't be used in the GROUP BY clause.
- Results returned from a SELECT statement that includes a GROUP BY clause are displayed in ascending order of the columns listed in the GROUP BY clause.

To have a different sort sequence, use the ORDER BY clause. When a SELECT statement includes all three clauses, the order in which they're evaluated is as follows:

- The WHERE clause • The GROUP BY clause
- The HAVING clause In essence, the WHERE clause filters the data before grouping, and the HAVING clause filters the groups after the grouping occurs.

## 8.1 Group by examples

```
DELETE FROM patient;
```

```
INSERT INTO patient values (111,'john','Wei','m','11-FEB-1978',25000, 'Davis','CA');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-1985',60000, 'Davis','NV');
INSERT INTO patient values (115,'dove','Grime','f','04-JUN-1960',20000, 'Sacramento','CA');
INSERT INTO patient values (199,'john','Dali','m','11-FEB-1978',25000, 'Davis','CA');
INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000, 'Davis','CA');
INSERT INTO patient values (978,'john','Doe','m','11-FEB-1978',25000,NULL,'CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-1999',50000,'Reno','NV');
```

	<u>RESULT</u>
111 john Wei m 11-FEB-1978 25000 Davis CA	
114 billy Bob f 05-MAY-1985 60000 Davis NV	
115 dove Grime f 04-JUN-1960 20000 Sacramento CA	
199 john Dali m 11-FEB-1978 25000 Davis CA	
112 john Smith m 01-MAR-1981 40000 Davis CA	245000
978 john Doe m 11-FEB-1978 25000 NULL CA	
113 jill Crane m 12-APR-1999 50000 Reno NV	
SELECT SUM(salary) FROM patient;	

<u>Group by city</u>	<u>RESULT</u>
111 john Wei m 11-FEB-1978 25000 Davis CA	
114 billy Bob f 05-MAY-1985 60000 Davis NV	
199 john Dali m 11-FEB-1978 25000 Davis CA	
112 john Smith m 01-MAR-1981 40000 Davis CA	Davis 150000
115 dove Grime f 04-JUN-1960 20000 Sacramento CA	Sacramento 20000
978 john Doe m 11-FEB-1978 25000 NULL CA	NULL 25000
113 jill Crane m 12-APR-1999 50000 Reno NV	Reno 50000
SELECT city, SUM(salary) FROM patient GROUP BY city;	
<u>Group by state</u>	<u>RESULT</u>
111 john Wei m 11-FEB-1978 25000 Davis CA	
199 john Dali m 11-FEB-1978 25000 Davis CA	
112 john Smith m 01-MAR-1981 40000 Davis CA	
978 john Doe m 11-FEB-1978 25000 NULL CA	CA 135000
115 dove Grime f 04-JUN-1960 20000 Sacramento CA	NV 110000
114 billy Bob f 05-MAY-1985 60000 Davis NV	
113 jill Crane m 12-APR-1999 50000 Reno NV	
SELECT state, SUM(salary) FROM patient GROUP BY state;	
<u>Group by gender</u>	<u>RESULT</u>
111 john Wei m 11-FEB-1978 25000 Davis CA	
199 john Dali m 11-FEB-1978 25000 Davis CA	
112 john Smith m 01-MAR-1981 40000 Davis CA	
978 john Doe m 11-FEB-1978 25000 NULL CA	m 165000
113 jill Crane m 12-APR-1999 50000 Reno NV	f 80000
114 billy Bob f 05-MAY-1985 60000 Davis NV	
115 dove Grime f 04-JUN-1960 20000 Sacramento CA	
SELECT gender, SUM(salary) FROM patient GROUP BY gender;	
<u>Group by city and gender</u>	<u>RESULT</u>
111 john Wei m 11-FEB-1978 25000 Davis CA	
199 john Dali m 11-FEB-1978 25000 Davis CA	
112 john Smith m 01-MAR-1981 40000 Davis CA	
114 billy Bob f 05-MAY-1985 60000 Davis NV	Davis m 90000
115 dove Grime f 04-JUN-1960 20000 Sacramento CA	Davis f 60000
978 john Doe m 11-FEB-1978 25000 NULL CA	Sacramento f 20000
113 jill Crane m 12-APR-1999 50000 Reno NV	NULL m 25000
	Reno m 50000
SELECT city, gender, SUM(salary) FROM patient GROUP BY city, gender;	

<b>Group by state and gender</b>							<b>RESULT</b>
111	john	Wei	m	11-FEB-1978	25000	Davis	CA
199	john	Dali	m	11-FEB-1978	25000	Davis	CA
112	john	Smith	m	01-MAR-1981	40000	Davis	CA
978	john	Doe	m	11-FEB-1978	25000	NULL	CA
115	dove	Grime	f	04-JUN-1960	20000	Sacramento	CA
114	billy	Bob	f	05-MAY-1985	60000	Davis	NV
113	jill	Crane	m	12-APR-1999	50000	Reno	NV
SELECT state, gender, SUM(salary) FROM patient GROUP BY state, gender;							
<b>Group by fname and city</b>							<b>RESULT</b>
111	john	Wei	m	11-FEB-1978	25000	Davis	CA
199	john	Dali	m	11-FEB-1978	25000	Davis	CA
112	john	Smith	m	01-MAR-1981	40000	Davis	CA
978	john	Doe	m	11-FEB-1978	25000	NULL	CA
113	jill	Crane	m	12-APR-1999	50000	Reno	NV
114	billy	Bob	f	05-MAY-1985	60000	Davis	NV
115	dove	Grime	f	04-JUN-1960	20000	Sacramento	CA
SELECT fname, city, SUM(salary) FROM patient GROUP BY fname, city;							

## 8.2 SUM

The SUM function is used to calculate the total amount stored in a numeric field for a group of records.

The syntax of the SUM function is `SUM([ DISTINCT | ALL ] n)`, where n is a column containing numeric data.

```
SELECT salary from patient;
```

--The result is a single number that is the summation of all the salaries.

```
SELECT SUM (salary) FROM patient;
```

--All is implied as in the above statement.

```
SELECT SUM (ALL salary) FROM patient;
```

--The result is a single number that is the summation of all the distinct salaries, which

--means it suppresses the duplicates before doing a summation.

```
SELECT SUM (DISTINCT salary) FROM patient;
```

```

SQL> SELECT salary from patient;
      SALARY
-----
      25000
      60000
      20000
      40000
      25000

6 rows selected.

SQL> --The result is a single number that is the summation of all the salaries
SQL> SELECT SUM (salary) FROM patient;

      SUM(SALARY)
-----
      170000

SQL> --All is implied as in the above statement
SQL> SELECT SUM (ALL salary) FROM patient;

      SUM(ALLSALARY)
-----
      170000

SQL> --The result is a single number that is the summation of all the distinct salaries which means
SQL> --it suppresses the duplicates before doing a summation
SQL> SELECT SUM (DISTINCT salary) FROM patient;

      SUM(DISTINCTSALARY)
-----
      145000

```

--Error: how do we align the single summation number with all the cities?

```
SELECT city, SUM(salary) FROM patient;
```

```

SQL> --This is problematic because how do we align the single summation number with all the cities
SQL> SELECT city, SUM(salary) FROM patient;
SELECT city, SUM(salary) FROM patient
*
ERROR at line 1:
ORA-00937: not a single-group group function

```

--Creates a grouping for each city, which means that it suppresses the duplicates in each group and --and then comes up with a summation for each group order by is the last clause.

```
SELECT city, SUM(salary) FROM patient GROUP BY city ORDER BY 1;
```

--Creates a group for each of the states and gives a summation for each state.

```
SELECT state, SUM(salary) FROM patient GROUP BY state;
```

--Creates combination of fname, city catagories and provides a summation for each group.

```
SELECT fname,city, SUM(salary) FROM patient GROUP BY fname,city;
```

```

SQL> --Creates a grouping for each city which means
SQL> --that it suppresses the duplicates in each group and
SQL> --and then comes up with a summation for each group
SQL> --order by is the last clause.
SQL> SELECT city, SUM(salary) FROM patient GROUP BY city ORDER BY 1;

CITY          SUM(SALARY)
-----
Davis           65000
Las Vegas       60000
Reno            20000
Sacramento     25000

SQL> --Creates a group for each of the states and gives a summation for each state
SQL> SELECT state, SUM(salary) FROM patient GROUP BY state;

STATE          SUM(SALARY)
-----
CA              110000
NV              60000

SQL> --Creates combination of fname, city catagories and provides a summation for each group
SQL> SELECT fname,city, SUM(salary) FROM patient GROUP BY fname,city;

FNAME          CITY          SUM(SALARY)
-----
billy          Las Vegas      60000
dove           Sacramento    20000
john           Davis         65000
john           Reno          25000
jill           Reno          25000

```

--First the where clause filters. Then it does a grouping with the data that is left over. It groups the different cities and then for each city group, it comes up with a summation.

```

SELECT city, SUM(salary) FROM patient WHERE UPPER(city)<>'RENO' GROUP BY city ORDER BY 1;

```

```

SQL> SELECT city, SUM(salary) FROM patient WHERE UPPER(city) <> 'RENO' GROUP BY city ORDER BY 1;

CITY          SUM(SALARY)
-----
Davis           65000
Las Vegas       60000
Sacramento     20000

```

--First the where clause filters. Then it does a grouping with the data that is left over. It groups the different cities and then for each city group, it comes up with a summation.

```

SELECT city, SUM(salary) FROM patient WHERE UPPER(city) != 'RENO' or city is NULL GROUP BY city ORDER BY 1;

```

SQL> --First the where clause filters. Then it does a grouping with the data that is left over. It groups

SQL> --the different cities and for each city group it comes up with a summation

```

SQL> SELECT city, SUM(salary) FROM patient WHERE UPPER(city) != 'RENO' or city is NULL GROUP BY city ORDER BY 1;

```

```

CITY          SUM(SALARY)
-----
Davis           65000
Las Vegas       60000
Sacramento     20000

```

## 8.3 DISTINCT

The optional DISTINCT keyword instructs Oracle to include only unique numeric values in its calculation. The ALL keyword instructs Oracle to include multiple occurrences of numeric values when totaling a field. If the DISTINCT or ALL keywords aren't included when using the SUM function, Oracle assumes the ALL keyword by default and uses all the numeric values in the field when the query is executed.

--ALL is implied and is not needed.

```
SELECT SUM (ALL salary) FROM patient;
```

--Suppresses the duplicates and gives a summation.

```
SELECT SUM (DISTINCT salary) FROM patient;
```

--Suppresses duplicates and gives a summation for each city category.

```
SELECT city, SUM(DISTINCT salary) FROM patient GROUP BY city ;
```

--Does not suppress the duplicates for salary and gives a summation for each city category.

```
SELECT city, SUM(ALL salary) FROM patient GROUP BY city;
```

```
SQL> --ALL is implied and is not needed
SQL> SELECT SUM (ALL salary) FROM patient;
```

```
SUM(ALLSALARY)
```

```
-----
```

```
170000
```

```
SQL> --Suppresses the duplicates and gives a summation
SQL> SELECT SUM (DISTINCT salary) FROM patient;
```

```
SUM(DISTINCTSALARY)
```

```
-----
```

```
145000
```

```
SQL> --suppresses duplicates and gives a summation for each city category
SQL> SELECT city, SUM(DISTINCT salary) FROM patient GROUP BY city ;
```

CITY	SUM(DISTINCTSALARY)
Las Vegas	25000
Davis	60000
Sacramento	65000
Reno	20000

```
SQL> --Does not suppress the duplicates for salary and gives a summation for each city category
```

```
SQL> SELECT city, SUM(ALL salary) FROM patient GROUP BY city;
```

CITY	SUM(ALLSALARY)
Las Vegas	25000
Davis	60000
Sacramento	65000
Reno	20000

--Just like the above; however, there is an additional filtering with the having clause. It  
-- includes only groups that have a sum greater than 25000.

```
SELECT city, SUM(ALL salary) FROM patient GROUP BY city HAVING sum(salary)>25000;
```

```
SQL> SELECT city, SUM(ALL salary) FROM patient GROUP BY city HAVING sum(salary)>25000;
```

CITY	SUM(ALLSALARY)
Las Vegas	60000
Davis	65000

```
SQL>
```

## 8.4 AVG

The AVG function calculates the average of numeric values in a specified column. The syn-tax of the AVG function is AVG([ DISTINCT | ALL ] n), where n is a column containing numeric data.

```
SELECT salary FROM patient;
SQL> SELECT salary FROM patient;
      SALARY
-----
    25000
    60000
    20000
    40000
    25000

6 rows selected.
```

--Gives a single average for all the salaries.

```
SELECT AVG (salary) FROM patient;
```

--Same as above

```
SELECT AVG (ALL salary) FROM patient;
```

--Suppresses duplicates and then gives an average.

```
SELECT AVG (DISTINCT salary) FROM patient;
```

--Invalid: Does not know how to display the one single average salary with all the cities.

```
SELECT city, AVG (salary) FROM patient;
```

--Displays the average salary for each city category.

```
SELECT city, AVG (salary) FROM patient GROUP BY city ORDER BY 1;
```

```

SQL> --Gives a single average for all the salaries
SQL> SELECT AVG (salary) FROM patient;
AVG(SALARY)
-----
34000

SQL> --Same as above
SQL> SELECT AVG (ALL salary) FROM patient;
AVG(ALLSALARY)
-----
34000

SQL> --Suppresses duplicates and then gives an average
SQL> SELECT AVG (DISTINCT salary) FROM patient;
AVG(DISTINCTSALARY)
-----
36250

SQL> --Invalid: Does not know how to display the one single average salary with
all the cities
SQL> SELECT city, AVG (salary) FROM patient;
SELECT city, AVG (salary) FROM patient
*
ERROR at line 1:
ORA-00937: not a single-group group function

SQL> --Displays the average salary for each city category
SQL> SELECT city, AVG (salary) FROM patient GROUP BY city ORDER BY 1;
CITY          AVG(SALARY)
-----
Davis          32500
Las Vegas      60000
Reno           20000
Sacramento     25000

```

--First it filters the data based on the where clause. Then it takes the left over records and  
--does a grouping for each of the cities and provides an average for each grouping.

```
SELECT city, AVG (salary) FROM patient WHERE UPPER(city) <> 'RENO' GROUP BY
city ORDER BY 1;
```

--First it filters the data based on the where clause. Then it takes the left over records and  
--does a grouping for each of the cities and provides an average for each grouping. If there is a  
--city group that does not have a salary, which means that it is NULL, then it will replace it with a  
--zero. There is additional filtering using the having clause after all the grouping is done.

```
SELECT city, AVG (nvl(salary,0)) FROM patient WHERE UPPER(city) <> 'RENO'
GROUP BY city HAVING AVG(salary)>20000 ORDER BY 1;
```

```

SQL> --First it filters the data based on the where clause. Then it takes the left over records and does a
SQL> --grouping for each of the cities and provides an average for each grouping
SQL> SELECT city, AVG (salary) FROM patient WHERE UPPER(city) <> 'RENO' GROUP BY city ORDER BY 1;
CITY          AVG(SALARY)
-----
Davis           32500
Las Vegas       60000
Sacramento      20000

SQL> --First it filters the data based on the where clause. Then it takes the left over records and does a
SQL> --grouping for each of the cities and provides an average for each grouping
. If there is a city group that
SQL> --does not have a salary which means that it is null, then it will replace
it with a zero
SQL> SELECT city, AVG (nvl(salary,0)) FROM patient WHERE UPPER(city) <> 'RENO'
GROUP BY city HAVING AVG(salary)>20000 ORDER BY 1;
CITY          AVG(NVL(SALARY,0))
-----
Davis           32500
Las Vegas       60000
SQL>

```

## 8.5 COUNT

Depending on the argument used, the COUNT function can count the records having non- NULL values in a specified field or count the total records meeting a specific condition, including those containing NULL values. The syntax of the COUNT function is COUNT(\* [ DISTINCT | ALL ] c), where c represents a numeric or non-numeric column.

```

SELECT fname, lname, city FROM patient;

--Counts the number of rows.
SELECT COUNT (*) FROM patient;

--Counts the number rows based on the contents of the city. If the city for a given row contains
--a NULL, then it will not be counted.
SELECT COUNT (city) FROM patient;

--Same as above
SELECT COUNT (ALL city) FROM patient;

--Invalid: Does not know how to display a single number with the six different cities.
SELECT city, COUNT (*) FROM patient;

```

```

SQL> SELECT fname, lname, city FROM patient;


| FNAME | LNAME | CITY       |
|-------|-------|------------|
| john  | Wei   | Davis      |
| billy | Bob   | Las Vegas  |
| dove  | Grime | Sacramento |
| john  | Smith | Davis      |
| john  | Doe   |            |
| jill  | Crane | Reno       |


6 rows selected.

SQL> --Counts the number of rows
SQL> SELECT COUNT (*) FROM patient;
COUNT(*)
-----
6

SQL> --Counts the number rows based on the contents of the city. If the city for
a given row contains
SQL> --a null then it will not be counted
SQL> SELECT COUNT (city) FROM patient;

COUNT(CITY)
-----
5

SQL> --Same as above
SQL> SELECT COUNT (ALL city) FROM patient;
COUNT(ALLCITY)
-----
5

SQL> --Invalid: Does not know how to display a single number with the six different cities
SQL> SELECT city, COUNT (*) FROM patient;
SELECT city, COUNT (*) FROM patient
*
ERROR at line 1:
ORA-00937: not a single-group group function

```

--Create a group for each of the different cities and do a count for each category. NULL cities are excluded from the count.

```
SELECT city, COUNT (city) FROM patient GROUP BY city;
```

--Same as above but the NULLs are not excluded.

```
SELECT city, COUNT (*) FROM patient GROUP BY city;
```

--After it has come up with the count per grouping, there is an additional filtering, which includes only those records where the count is greater than 1.

```
SELECT city, COUNT (*) FROM patient GROUP BY city HAVING COUNT(*) >1;
```

```

SQL> --Create a group for each of the different cities and do a count for each c
category. Null cities are excluded
SQL> --from the count
SQL> SELECT city, COUNT(city) FROM patient GROUP BY city;
CITY          COUNT(CITY)
-----
Las Vegas      0
Davis          1
Sacramento    2
Reno           1

SQL> --Same as above but the nulls are not excluded
SQL> SELECT city, COUNT(*) FROM patient GROUP BY city;
CITY          COUNT(*)
-----
Las Vegas      1
Davis          1
Sacramento    2
Reno           1

SQL> --After it has come up with the count per grouping, there is an additional
filtering which only includes
SQL> --those where the count is greater than 1.
SQL> SELECT city, COUNT(*) FROM patient GROUP BY city HAVING COUNT(*) > 1;
CITY          COUNT(*)
-----
Davis          2

```

## 8.6 MAX

The MAX function returns the largest value stored in the specified column. The syntax of the MAX function is MAX([ DISTINCT| ALL] c), where c can represent any numeric, character, or date column.

```

SELECT salary FROM patient;

--The highest salary is displayed.
SELECT MAX (salary) FROM patient;
--same as above
SELECT MAX (ALL salary) FROM patient;
SQL> SELECT salary FROM patient;
SALARY
-----
25000
60000
20000
40000
25000

6 rows selected.

SQL> --The highest salary
SQL> SELECT MAX (salary) FROM patient;
MAX(SALARY)
-----
60000

SQL> --same as above
SQL> SELECT MAX (ALL salary) FROM patient;
MAX(ALLSALARY)
-----
60000

```

--Invalid: the highest salary is a single number and cannot be associated with all cities.

```
SELECT city, MAX (salary) FROM patient;
```

--Displays highest salary for each city.

```
SELECT city, MAX (salary) FROM patient GROUP BY city;
```

SQL> --Invalid: the highest salary is a single number and cannot be associated with all cities

```
SQL> SELECT city, MAX (salary) FROM patient;
```

```
SELECT city, MAX (salary) FROM patient
```

\*

ERROR at line 1:

ORA-00937: not a single-group group function

SQL> -- highest salary for each city

```
SQL> SELECT city, MAX (salary) FROM patient GROUP BY city;
```

CITY	MAX(SALARY)
Las Vegas	25000
Davis	60000
Sacramento	40000
Reno	20000

--Given the fname, city combination, display the the number of records and the highest salary

--for each of those combination categories.

```
SELECT fname, city, COUNT(*), AVG (salary), MAX(salary) FROM patient GROUP BY
fname, city;
```

--Same as above except that after the final result, do some additional filtering based on the count.

```
SELECT fname, city, COUNT(*), AVG (salary), MAX(salary) FROM patient GROUP BY
fname, city HAVING COUNT(*) > 2;
```

SQL> --Given the fname, city combination, display the the number of records, and the highest salary

SQL> --for each of those combination categories

```
SQL> SELECT fname, city, COUNT(*), AVG (salary), MAX(salary) FROM patient GROUP
BY fname, city;
```

FNAME	CITY	COUNT(*)	AVG(SALARY)	MAX(SALARY)
billy	Las Vegas	1	60000	60000
dove	Sacramento	1	20000	20000
john	Davis	2	32500	40000
john		1	25000	25000
jill	Reno	1		

SQL> --Same as above except that after the final result do some additional filtering based on the count

```
SQL> SELECT fname, city, COUNT(*), AVG (salary), MAX(salary) FROM patient GROUP
BY fname, city HAVING COUNT(*) > 2;
```

no rows selected

## 8.7 MIN

In contrast to the MAX function, the MIN function returns the smallest value in a specified column. As with the MAX function, the MIN function works with any numeric, character, or date column. The syntax of the MIN function is MIN([ DISTINCT | ALL ] c), where c represents any character, numeric, or date column. The MIN function uses the same logic as the MAX function for numeric and character data, except it returns the smallest value rather than the largest value.

```

SELECT salary FROM patient;

--The lowest salary is displayed.
SELECT MIN (salary) FROM patient;

--Invalid: cannot display a single number with six cities.
SELECT city, MIN(salary) FROM patient;

--Display the lowest salary for each city category and display the number of records in each group.
SELECT city, MIN (salary), COUNT(*) FROM patient GROUP BY city;
SQL> SELECT salary FROM patient;
      SALARY
-----
25000
60000
20000
40000
25000

6 rows selected.

SQL> --The lowest salary
SQL> SELECT MIN (salary) FROM patient;
      MIN(SALARY)
-----
20000

SQL> --Invalid: cannot display a single number with six cities
SQL> SELECT city, MIN(salary) FROM patient;
      SELECT city, MIN(salary) FROM patient
      *
ERROR at line 1:
ORA-00937: not a single-group group function

SQL> --the lowest salary for each city category and display the number of records in each group
SQL> SELECT city, MIN (salary), COUNT(*) FROM patient GROUP BY city;
      CITY          MIN(SALARY)    COUNT(*)
-----
Las Vegas            25000           1
Davis                60000           1
Sacramento          25000           2
Reno                 20000           1

```

--Filters with the where clause. Then given the remaining records, it groups by city and finds the lowest salary for each city category. Given the result set, it only includes the ones where there are more than two records for each group. The results are sorted by city.

```
SELECT city, MIN (salary) FROM patient WHERE city IS NOT NULL GROUP BY city HAVING count(*) >1 ORDER BY 1 DESC;
```

SQL> --Filters with the where clause, then given the remaining records, groups by city and finds the lowest salary for each city category. Given the result set, it only includes the ones where there are more than two records for each group. Order by city

```
SQL> SELECT city, MIN (salary) FROM patient WHERE city IS NOT NULL GROUP BY city HAVING count(*) > 1 ORDER BY 1 DESC;
```

CITY	MIN(SALARY)
Davis	25000

## 8.8 Dates and group functions

--Displays oldest person, youngest person, the number of records (excludes all those that have a NULL in --DOB), and number of records (Suppresses duplicate DOB).

```
SELECT min(DOB), max (DOB), count(DOB), count (DISTINCT DOB) FROM patient;
```

--Invalid: Cannot apply AVG to date formats. Use months\_between to convert it into a number

--and then do an average.

```
SELECT AVG(DOB) FROM patient;
```

--Invalid: can do a sum on date formats.

```
SELECT SUM(DOB) FROM patient;
```

SQL> --Oldest person, youngest person, the number of records (excludes all those that have a null in dob),

SQL> --number of records (Suppresses duplicate dob)

```
SQL> SELECT min(dob), max (dob), count(dob), count (DISTINCT dob) FROM patient;
```

MIN(DOB)	MAX(DOB)	COUNT(DOB)	COUNT(DISTINCT DOB)
04-JUN-60	12-APR-99	6	5

SQL> --Invalid: Cannot apply avg to date formats. Use months\_between to convert it into a number and then

--do an average

```
SQL> SELECT avg(dob) FROM patient;
```

```
SELECT avg(dob) FROM patient
```

\*

ERROR at line 1:

ORA-00932: inconsistent datatypes: expected NUMBER got DATE

SQL> --Invalid: can do a sum on date formats

```
SQL> SELECT SUM(dob) FROM patient;
```

```
SELECT SUM(dob) FROM patient
```

\*

ERROR at line 1:

ORA-00932: inconsistent datatypes: expected NUMBER got DATE

## ✓ CHECK 8A

1. Display the count of all people who make less than 10000 for each of the different personality types.
2. Display the average age and maximum salary for each personality type. Display both the average age and personality types.
3. What is wrong with the following:
  - o `SELECT * FROM patient WHERE salary> AVG(salary);`
  - o `SELECT fname AS firstname, SUM (salary) summation FROM patient WHERE firstname='john'`  
`HAVING summation>10000`

*"Be kind, Remember everyone you meet is fighting a hard battle. "*

## Summary Examples

--Display oldest person, youngest person, average salary (Does not include NULLs) and sum of all salaries.  
`SELECT min(DOB), max (DOB), count(DOB), , AVG(salary), sum(salary) FROM patient;`

--COUNT(salary): number of rows where salary is not NULL.

--COUNT(\*): number of rows including NULLs.

--COUNT(NVL(salary, 0)): Number of rows including NULLs because the NULLs are replaced with zeroes.

--Note: NVL does not change the actual data in the underlying table.

`SELECT COUNT(salary), COUNT(*), COUNT(NVL(salary, 0)) FROM patient;`

--First it does the filtering with the where clause based on salary and gender.

--It takes the results and creates grouping based on lname and gender and comes up with the number of rows for --each group. It then accepts all counts greater than 1 and finally orders the entire result set based on gender.

```
SELECT lname, gender, count(*) FROM patient WHERE
    salary> (SELECT AVG(salary) FROM patient) AND gender
    IS NOT NULL
    GROUP BY lname, gender
    HAVING count(*)>1
    ORDER BY 2;
```



*Ellison styled his estimated \$70 million Woodside, California, estate after feudal Japanese architecture, complete with a man-made 2.3-acre lake and an extensive seismic retrofit. In 2004 and 2005, Ellison purchased more than 12 properties in Malibu, California, worth more than \$180 million. The \$65 million Ellison spent on five contiguous lots on Malibu's Carbon Beach was the most costly residential transaction in United States history until Ron Perelman sold his Palm Beach, Florida compound for \$70 million later that same year. His entertainment system cost \$1 million, and includes a rock concert-sized video projector at one end of a drained swimming pool, using the gaping hole as a giant subwoofer.*

## Chapter 9(SUBQUERIES)

*"A friend is one who knows us, but loves us anyway."*

**Example 9a**

```

DROP TABLE patient_disease;
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR,
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state      VARCHAR2(20)
);

INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (112,'john','Smith','m','01-MAR-1981',40000,
'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',NULL,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-1985',60000,'Las
Vegas','NV');
INSERT INTO patient values (115,'dove','Grime','f','04-JUN-
1960',20000,'Sacramento','CA');

DROP TABLE disease;
CREATE TABLE disease
(
    disease_id NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);
INSERT INTO disease VALUES (11,'Cancer');
INSERT INTO disease VALUES (22,'Malaria');
INSERT INTO disease VALUES (33,'Flu');

CREATE TABLE patient_disease
(
    Patient_id      NUMBER REFERENCES patient,
    disease_id      NUMBER REFERENCES disease,
    PRIMARY KEY (patient_id, disease_id)
);
INSERT INTO patient_disease VALUES (111,11);
INSERT INTO patient_disease VALUES (111,22);
INSERT INTO patient_disease VALUES (113,11);

```

<u>Patient</u>				<u>Disease</u>			
<u>Patient_id</u> <u>Fname</u> <u>Lname</u> ...				<u>Disease_id</u> <u>Disease_desc</u>			
111        john     Doe     ...				11              Cancer			
114        billy    Bob     ...				22              Malaria			
112        john    Smith    ...				33              Flu			
113        jill    Crane    ...							
<u>Patient_Disease</u>							
<u>Patient_id</u> <u>Disease_id</u>							
111            11							
111            22							
113            11							

A subquery is a SELECT statement used in another SQL command. Any type of action you can perform with a SELECT statement (such as filtering rows, filtering columns, and calculating aggregate amounts) can be performed when creating a table with a subquery. This first query is the subquery. The subquery's results are passed as input to the outer query (also called the parent query). The outer query incorporates this value into its calculations to determine the final output.

You can nest subqueries inside the FROM, WHERE, or HAVING clauses of other subqueries. In Oracle, subqueries in a WHERE clause can be nested to a depth of 255 subqueries, and there's no depth limit when subqueries are nested in a FROM clause. When nesting sub-queries, you might want to use the following strategy:

Determine exactly what you're trying to find—in other words, the goal of the query.

Write the innermost subquery first.

Next, look at the value you can pass to the outer query. If it isn't the value the outer query needs (for example, it references the wrong column), analyze how you need to convert the data to get the correct rows. If necessary, use another subquery between the outer query and the nested subquery.

Keep the following rules in mind when working with any type of subquery:

- A subquery must be a complete query in itself—in other words, it must have at least a SELECT and a FROM clause.

A subquery, except one in the FROM clause, can't have an ORDER BY clause. If you need to display output in a specific order, include an ORDER BY clause as the outer query's last clause.

A subquery must be enclosed in parentheses to separate it from the outer query.

If you place a subquery in the outer query's WHERE or HAVING clause, you can do so only on the right side of the comparison operator.

### **Example 9b (using separate queries)**

--In this example, we want to know the names of all the people who are infected with Malaria.  
--Using separate queries, we first have to find out what the disease\_id is for malaria which  
--is in the disease table. We then use the disease\_id to find out which patient\_id(s) have malaria.  
--For this we need to go to the patient\_disease table. Finally, we take the list of patient\_ids and  
--feed it into the patient table to get their names. Based on what we know so far, we would write  
--three separate queries but in the next example, we will connect them into a single query. This  
--would be done through the subquery mechanism.

```
SELECT disease_id FROM disease WHERE disease_desc='Malaria';
SELECT patient_id FROM patient_disease WHERE disease_id=22;
SELECT fname, lname FROM patient WHERE patient_id=111;
```

```
SQL> SELECT disease_id FROM disease WHERE disease_desc='Malaria';
DISEASE_ID
-----
22
SQL> SELECT patient_id FROM patient_disease WHERE disease_id=22;
PATIENT_ID
-----
111
SQL> SELECT fname, lname FROM patient WHERE patient_id=111;
FNAME          LNAME
-----          -----
john           Doe
```

### **Example 9c (Using subqueries-one single row)**

A single- row subquery can return only one row of results consisting of only one column to the outer query. A single- row subquery can also be nested in the outer query's SELECT clause.

--Display all the patients who have malaria. Returns a single row.  
--Start with inner most query and work your way to the outer query. Notice the number of  
--parentheses. The indentation is used for readability.

```
SELECT fname, lname FROM patient WHERE patient_id=(  
    SELECT patient_id FROM patient_disease WHERE disease_id=(  
        SELECT disease_id FROM disease WHERE  
disease_desc='Malaria'));
```

--Invalid. Notice the asterisk does not match up with disease\_id.

--The disease\_id must match with disease\_id and not the asterisk.

```
SELECT fname, lname FROM patient WHERE patient_id=(  
    SELECT patient_id FROM patient_disease WHERE disease_id=(  
        SELECT * FROM disease WHERE disease_desc='Malaria'));
```

--Invalid. Notice the asterisk does not match up with patient\_id.

```
SELECT fname, lname FROM patient WHERE patient_id=(  
    SELECT * FROM patient_disease WHERE disease_id=(  
        SELECT disease_id FROM disease WHERE  
disease_desc='Malaria'));
```

```

SQL> --Display all the patients who have malaria. Returns a single row
SQL> --Start with inner most query and work your way to the outer query. Notice
the number of
SQL> --parantheses. The indentation is used for readability
SQL> SELECT fname, lname FROM patient WHERE patient_id=(
  2   SELECT patient_id FROM patient_disease WHERE disease_id=(
  3     SELECT disease_id FROM disease WHERE disease_desc='Malaria'));

FNAME          LNAME
-----          -----
john           Doe

SQL>
SQL> --Invalid. Notice the asterisk does not match up with disease_id
SQL> --the disease_id must match with disease_id and not the asterisk.
SQL> SELECT fname, lname FROM patient WHERE patient_id=(
  2   SELECT patient_id FROM patient_disease WHERE disease_id=(
  3     SELECT * FROM disease WHERE disease_desc='Malaria'));
          SELECT * FROM disease WHERE disease_desc='Malaria')
*
ERROR at line 3:
ORA-00913: too many values

SQL>
SQL> --Invalid. Notice the asterisk does not match up with patient_id
SQL> SELECT fname, lname FROM patient WHERE patient_id=(
  2   SELECT * FROM patient_disease WHERE disease_id=(
  3     SELECT disease_id FROM disease WHERE disease_desc='Malaria'));

          SELECT * FROM patient_disease WHERE disease_id=
*
ERROR at line 2:
ORA-00913: too many values

```

### ***Example 9d (Multiple rows)***

Multiple- row subqueries are nested queries that can return more than one row of results to the parent query. The main rule to keep in mind when working with multiple- row subqueries is that you must use multiple- row operators. If a single- row operator is used with a subquery that returns more than one row of results, Oracle returns an error message. Valid multiple- row operators include IN, ALL, and ANY must be used. Of the three, the IN Operator is used most often.

```

--Invalid. Display all the patients who have Cancer. Returns multiple rows.
--Start from the inner-most query. The result would then bubble up to the outer queries. The
--problem with this query is that there are multiple patients who suffer from cancer. This
--would mean that the second subquery would return multiple rows; however, the (=) from the outer
--most query can only handle one single piece of information.
SELECT fname, lname FROM patient WHERE patient_id=(
  SELECT patient_id FROM patient_disease WHERE disease_id=(
    SELECT disease_id FROM disease WHERE disease_desc='Cancer'));

--To correct the above problem, we change from (=) to (in). The in operator can handle multiple values.
SELECT fname, lname FROM patient WHERE patient_id IN(
  SELECT patient_id FROM patient_disease WHERE disease_id=(
    SELECT disease_id FROM disease WHERE disease_desc='Cancer')));

```

```

SQL> --Invalid. Display all the patients who have Cancer. Returns multiple rows
SQL> --Start from the innermost query. The result would then bubble up to the outer queries. The
SQL> --problem with this query is that there are multiple patients who suffer from cancer. This would
SQL> --mean that the second subquery would return multiple rows; however, the () from the
SQL> --outer-most query can only handle one single piece of information
SQL> SELECT fname, lname FROM patient WHERE patient_id=(
  2   SELECT patient_id FROM patient_disease WHERE disease_id=(
  3     SELECT disease_id FROM disease WHERE disease_desc='Cancer'));
      SELECT patient_id FROM patient_disease WHERE disease_id=*
ERROR at line 2:
ORA-01427: single-row subquery returns more than one row

SQL>
SQL> --To correct the above problem, we change from (=) to (in). The in operator can handle multiple
SQL> --values
SQL> SELECT fname, lname FROM patient WHERE patient_id IN(
  2   SELECT patient_id FROM patient_disease WHERE disease_id=(
  3     SELECT disease_id FROM disease WHERE disease_desc='Cancer'));

FNAME          LNAME
-----
john           Doe
jill           Crane

```

### *Example 9e (Single and multiple rows)*

--Display all the diseases that "jill crane" has.

```

SELECT disease_desc FROM disease WHERE disease_id=(
  SELECT disease_id FROM patient_disease WHERE patient_id=(
    SELECT patient_id FROM patient WHERE fname='jill' and lname='Crane' ));

```

--Invalid. Display all the diseases that "John Doe" has. Must use in clause.

```

SELECT disease_desc FROM disease WHERE disease_id = (
  SELECT disease_id FROM patient_disease WHERE patient_id=(
    SELECT patient_id FROM patient WHERE fname='john' and lname='Doe' ));

```

```

SELECT disease_desc FROM disease WHERE disease_id IN (
  SELECT disease_id FROM patient_disease WHERE patient_id=(
    SELECT patient_id FROM patient WHERE fname='john' and lname='Doe' ));

```

--Notice the concatenation operator in the subquery.

--The concatenation operator takes the two pieces of data and connect them together  
--to make it appear as one single piece.

```

SELECT disease_desc FROM disease WHERE disease_id IN (
  SELECT disease_id FROM patient_disease WHERE patient_id=(
    SELECT patient_id FROM patient WHERE (fname || lname)=('johnDoe')));

```

```

SQL> --Display all the diseases that "jill crane" has
SQL> SELECT disease_desc FROM disease WHERE disease_id=
  2      SELECT disease_id FROM patient_disease WHERE patient_id=
  3          SELECT patient_id FROM patient WHERE fname='jill' and lname='Crane');
DISEASE_DESC
-----
Cancer

SQL> --Invalid. Display all the diseases that "John Doe" has. Must use In clause
SQL> SELECT disease_desc FROM disease WHERE disease_id = (
  2      SELECT disease_id FROM patient_disease WHERE patient_id=
  3          SELECT patient_id FROM patient WHERE fname='john' and lname='Doe',
  );
      SELECT disease_id FROM patient_disease WHERE patient_id=*
*
ERROR at line 2:
ORA-01427: single-row subquery returns more than one row

SQL>
SQL> SELECT disease_desc FROM disease WHERE disease_id IN (
  2      SELECT disease_id FROM patient_disease WHERE patient_id=
  3          SELECT patient_id FROM patient WHERE fname='john' and lname='Doe',
  );
DISEASE_DESC
-----
Malaria
Cancer

SQL>
SQL> --Notice the concatenation operator in the subquery
SQL> --The concatenation operator takes the two pieces of data and connect them
together
SQL> --to make it appear as one single piece.
SQL> SELECT disease_desc FROM disease WHERE disease_id IN (
  2      SELECT disease_id FROM patient_disease WHERE patient_id=
  3          SELECT patient_id FROM patient WHERE (fname || lname)='johnDoe',
  );
DISEASE_DESC
-----
Malaria

```

### ***Example 9f (Multiple column subquery)***

Multiple- column subquery returns more than one column to the outer query. The syntax of the outer WHERE clause is WHERE ( columnname, columnname, ...) IN subquery.

Keep these rules in mind:

- Because the WHERE clause contains more than one column name, the column list must be enclosed in parentheses. Column names listed in the WHERE clause must be in the same order as they're listed in the subquery's SELECT clause.

```

DROP TABLE special_names;
CREATE TABLE special_names (fname VARCHAR2(20), lname VARCHAR2(30));
INSERT INTO special_names VALUES ('john', 'Doe');
INSERT INTO special_names VALUES ('jill', 'Crane');

--Notice that fname and lname are enclosed in parantheses in the outer query. It works
--like the concatenation operator in that fname and lname become a single piece of data
--which are compared against fname and lname in the inner query. The (IN) operator is used
--because it is possible that the inner query may yield multiple rows.
SELECT patient_id FROM patient WHERE (fname, lname) IN (
    SELECT fname, lname FROM special_names);

```

```

SQL> SELECT patient_id FROM patient WHERE (fname, lname) IN (
2          SELECT fname, lname FROM special_names);

PATIENT_ID
-----
111
113

```

### **Example 9g (Group functions and subqueries)**

```

SELECT AVG(salary) FROM patient;

--Invalid. Must use a subquery. For every row that is processed from the patient table
--we have to compare its salary against the AVG(salary). We cannot combine a group
--function with row level processing which is why this gives us an error.
SELECT fname, lname, salary FROM patient WHERE salary > AVG(salary);

```

```

SQL> SELECT AVG(salary) FROM patient;

AVG(SALARY)
-----
36250

SQL>
SQL> --Invalid. Must use a subquery. For every row that is processed from the pa
tient table
SQL> --we have to compare its salary against the AVG(salary). We cannot combine
a group
SQL> --function with row level processing which is why this gives us an error
SQL> SELECT fname, lname, salary FROM patient WHERE salary > AVG(salary);
SELECT fname, lname, salary FROM patient WHERE salary > AVG(salary)
*
ERROR at line 1:
ORA-00934: group function is not allowed here

```

--In this case, the inner query will be executed which comes up with a single number.  
--That single number will be fed to the outer query which can be used to compare  
--against every row in the patient table.

```

SELECT fname, lname, salary FROM patient WHERE salary >
(SELECT AVG(salary) FROM patient);

```

--Invalid: AVG cannot be used on DATE datatypes

```

SELECT fname, lname, DOB FROM patient where DOB>
(SELECT AVG(DOB) FROM patient);

```

```

SQL> SELECT fname, lname, salary FROM patient WHERE salary >
2      (SELECT AVG(salary) FROM patient);
FNAME          LNAME          SALARY
-----          -----          -----
john            Smith          40000
billy           Bob            60000

SQL> --Invalid: AVG cannot be used on DATE datatypes
SQL> SELECT fname, lname, dob FROM patient where dob>
2      (SELECT AVG(dob) FROM patient);
      (SELECT AVG(dob) FROM patient)
*
ERROR at line 2:
ORA-00932: inconsistent datatypes: expected NUMBER got DATE

```

--To make AVG work, dates have to be converted to numbers which can be done by using --MONTHS\_BETWEEN. Notice that a subquery has to be used to deal with the AVG first.

```

SELECT fname, lname, DOB FROM patient where MONTHS_BETWEEN(sysdate,DOB)>
      (SELECT AVG(MONTHS_BETWEEN(sysdate,DOB)) FROM patient);

```

```

SQL> --To make AVG work, dates have to be converted to numbers which can be done
      by
SQL> --using MONTHS_BETWEEN. Notice the a subquery has to be used to deal with
      the AVG
SQL> --first
SQL> SELECT fname, lname, dob FROM patient where MONTHS_BETWEEN(sysdate,dob)>
      2      (SELECT AVG(MONTHS_BETWEEN(sysdate,dob)) FROM patient);

```

FNAME	LNAME	DOB
john	Doe	11-FEB-78
dove	Grime	04-JUN-60

### **Example 9h (Create table and subqueries)**

You can also perform CREATE TABLE AS by using subqueries.

--Instead of displaying the information on to the screen, it can be fed into a brand new table.

```

CREATE TABLE NEW_TABLE1 AS SELECT patient_id FROM patient WHERE
(fname, lname) IN (SELECT fname, lname FROM special_names);

```

```

SELECT * FROM NEW_TABLE1;

```

```

SQL> CREATE TABLE NEW_TABLE1 AS SELECT patient_id FROM patient WHERE (fname, lname) IN (
2           SELECT fname, lname FROM special_names);

```

Table created.

```

SQL>

```

```

SQL> SELECT * FROM NEW_TABLE1;

```

PATIENT\_ID

111
113

```
--Invalid. Must use an alias because the new table will be using the information between
--the select and from to come up with the column names for the new tables. Since
--salary*2 is not a valid column name, an alias has to be used.
CREATE TABLE NEW_TABLE2 AS SELECT patient_id, salary * 2 FROM patient
WHERE (fname, lname) IN (SELECT fname, lname FROM special_names);

--This query corrects the problem from the previous example.
CREATE TABLE NEW_TABLE2 AS SELECT patient_id, salary * 2 Increase FROM patient
WHERE (fname, lname) IN (    SELECT fname, lname FROM
special_names);

SELECT * FROM NEW_TABLE2;

SQL> --Invalid. Must use an Alias because the new table will be using the inform
ation between
SQL> --the select and from to come up with the column names for the new tables.
Since
SQL> --salary*2 is not a valid column name, an alias has to be used
SQL> CREATE TABLE NEW_TABLE2 AS SELECT patient_id, salary * 2 FROM patient WHERE
 (fname, lname) IN (    SELECT fname, lname FROM special_names);
CREATE TABLE NEW_TABLE2 AS SELECT patient_id, salary * 2 FROM patient WHERE (fna
me, lname) IN (    SELECT fname, lname FROM special_names)
*
ERROR at line 1:
ORA-00998: must name this expression with a column alias

SQL>
SQL> --This query corrects the problem from the previous example
SQL> CREATE TABLE NEW_TABLE2 AS SELECT patient_id, salary * 2 Increase FROM pati
ent WHERE (fname, lname) IN (    SELECT fname, lname FROM special_names);

Table created.

SQL>
SQL> SELECT * FROM NEW_TABLE2;
PATIENT_ID  INCREASE
-----
 111        50000
 113
```

### **Example 9i (Update and delete using subqueries)**

You can also perform UPDATE and DELETE statements

```

SELECT patient_id, salary FROM patient;

--This example updates the salary for all those patients who have cancer.
UPDATE patient SET salary=salary*2 WHERE patient_id IN (
    SELECT patient_id FROM patient_disease WHERE disease_id=(
        SELECT disease_id FROM disease WHERE disease_desc='Cancer')) ;

SELECT patient_id, salary FROM patient;

--This example deletes all the records from the patient_disease table for all those who have cancer.
DELETE FROM patient_disease WHERE disease_id IN (
    SELECT disease_id FROM disease WHERE disease_desc='Cancer');

SELECT * FROM patient_disease;

SQL> SELECT patient_id, salary FROM patient;
PATIENT_ID      SALARY
-----  -----
      111      25000
      112      40000
      113
      114      60000
      115      20000

SQL> --This example updates the salary for all those patients who have cancer
SQL> UPDATE patient SET salary=salary*2 WHERE patient_id IN (
2       SELECT patient_id FROM patient_disease WHERE disease_id=(
3           SELECT disease_id FROM disease WHERE disease_desc='Cancer')) ;

2 rows updated.

SQL> SELECT patient_id, salary FROM patient;
PATIENT_ID      SALARY
-----  -----
      111      50000
      112      40000
      113
      114      60000
      115      20000

SQL> --This example delete all the records from the patient_disease table for all those who have cancer
SQL> DELETE FROM patient_disease WHERE disease_id IN (
2       SELECT disease_id FROM disease WHERE disease_desc='Cancer');

2 rows deleted.

SQL> SELECT * FROM patient_disease;
PATIENT_ID DISEASE_ID
-----  -----
      111      22

```

```
SQL> CREATE TABLE Candidate2 AS SELECT fname, lname, DECODE (partyid,(SELECT partyid FROM Party WHERE UPPER (partydesc)='REPUBLICAN'), (salary-salary*.10), salary) new_salary FROM Candidate;
```

✓ Table created.

## CHECK 9A

1. Display the salary of all those who are good (regardless of case).
2. Display only the personality description for those people who have a personality.
3. Display the name of all those who are making more than the average salary.
4. Delete all those who are making less than the average salary.
5. What is wrong with the following?

```
SELECT * FROM patient WHERE patient_id =
    (SELECT * FROM patient_disease WHERE disease_id =
        (SELECT disease_id FROM disease WHERE disease_desc='MaLaRia' ORDER BY fname));
```

```
SELECT * FROM patient WHERE (fname, lname) IN (SELECT fname FROM patient2);
```

*"The true price of anything is the amount of life you exchange for it"*

## Summary Examples

--This example updates the salary for all those patients who have cancer.

--Notice the IN clause will have to be used to deal with multiple rows.

--Also, note that the where condition for the outer query filters on both the patient\_id and the salary.

```
SELECT patient_id, salary FROM patient WHERE patient_id IN (
    SELECT patient_id FROM patient_disease WHERE disease_id=(
        SELECT disease_id FROM disease WHERE to_lower(disease_desc)='cancer' ))
AND salary IS NOT NULL ;
```

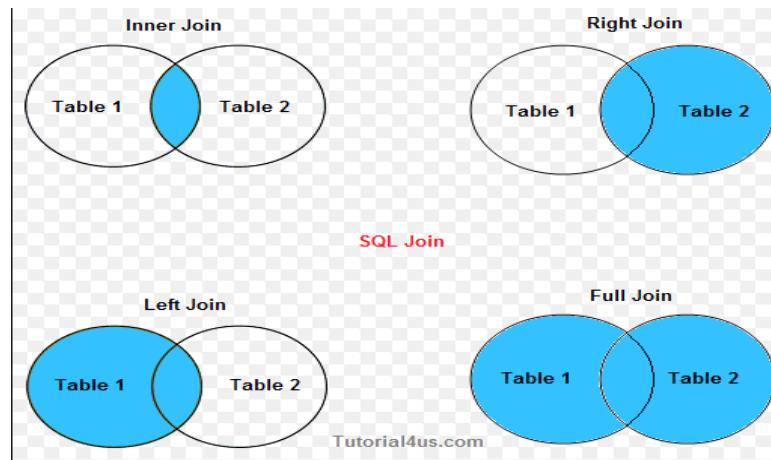
--Notice that the AVG function is enclosed in its own query. Salary>AVG(salary) is wrong.

```
SELECT patient_id, salary FROM patient WHERE salary >
    (SELECT AVG(salary) FROM Patient)
```

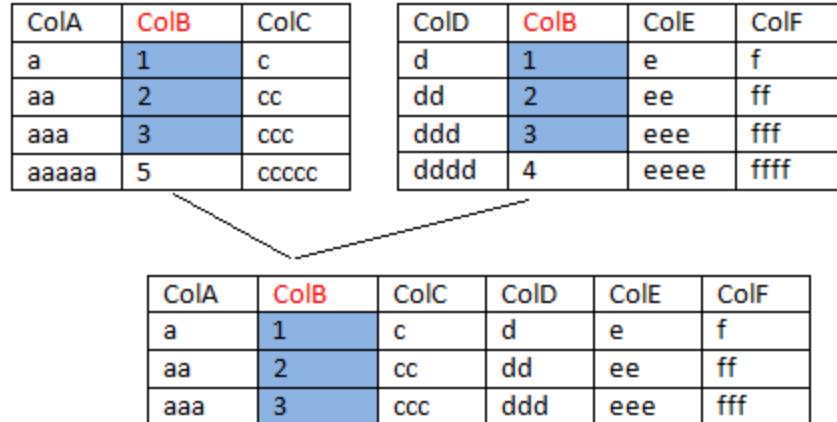


*Ellison wrote: "Many years ago, I put virtually all of my assets into a trust with the intent of giving away at least 95 percent of my wealth to charitable causes. I have already given hundreds of millions of dollars to medical research and education, and I will give billions more over time. Until now, I have done this giving quietly—because I have long believed that charitable giving is a personal and private matter.*

## Chapter 10 (Joins)



*"Life is not the amount of breaths you take, it's the moments that take your breath away..."*



Cartesian product or CROSS JOIN	Replicates each row from the first table with every row from the second table
Equality join also known as equijoin, inner join or a simple join	Creates a join by using a commonly named and defined column
Non-equality join	Joins tables when there are no equivalent rows in the tables to be joined
Self-join	Joins a table to itself.
Outer join	Includes records of a table when there is no matching record in the other table.
Set operators UNION, UNION ALL, INTERSECT and MINUS	Combines results from multiple select statements

WHERE	In the traditional approach, the WHERE clause indicates which columns should be joined
NATURAL JOIN	The keywords are used in the FROM clause to join tables containing a common column with the same name and definition
JOIN ... USING	The JOIN keyword is used in the FROM clause; combined with the USING clause, it identifies the common column used to join the tables.
JOIN ... ON	The JOIN keyword is used in the FROM clause. The ON clause identifies the columns used to join the tables
OUTER JOIN can be used with LEFT, RIGHT, FULL	Indicates that at least one of the tables doesn't have a matching row in the other table

```

DROP TABLE patient_disease;
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR,
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state      VARCHAR2(20)
);

INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',NULL,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-1985',60000,'Las
Vegas','NV');

DROP TABLE disease;
CREATE TABLE disease
(
    disease_id NUMBER PRIMARY KEY,
    disease_desc VARCHAR2(20)
);
INSERT INTO disease VALUES (11,'Cancer');
INSERT INTO disease VALUES (22,'Malaria');
INSERT INTO disease VALUES (33,'Flu');

CREATE TABLE patient_disease
(
    Patient_id          NUMBER REFERENCES patient,
    disease_id          NUMBER REFERENCES disease,
    PRIMARY KEY (patient_id, disease_id)
);
INSERT INTO patient_disease VALUES (111,11);
INSERT INTO patient_disease VALUES (111,22);
INSERT INTO patient_disease VALUES (113,11);

```

<b>Patient</b>				<b>Disease</b>	
<u>Patient_id</u>	<u>Fname</u>	<u>Lname...</u>		<u>Disease_id</u>	<u>Disease_desc</u>
111	john	Doe	...		Cancer
114	billy	Bob	...		Malaria
113	jill	Crane	...		Flu
<b>Patient Disease</b>					
<u>Patient_id</u>	<u>Disease_id</u>				
111	11				
111	22				
113	11				

```

DROP TABLE heroes;
DROP TABLE skills;
CREATE TABLE heroes
(
    name VARCHAR(10),
    skill_code NUMBER
);
INSERT INTO heroes VALUES ('superman',1);
INSERT INTO heroes VALUES ('aqua-man',2);
INSERT INTO heroes VALUES ('flash',NULL);

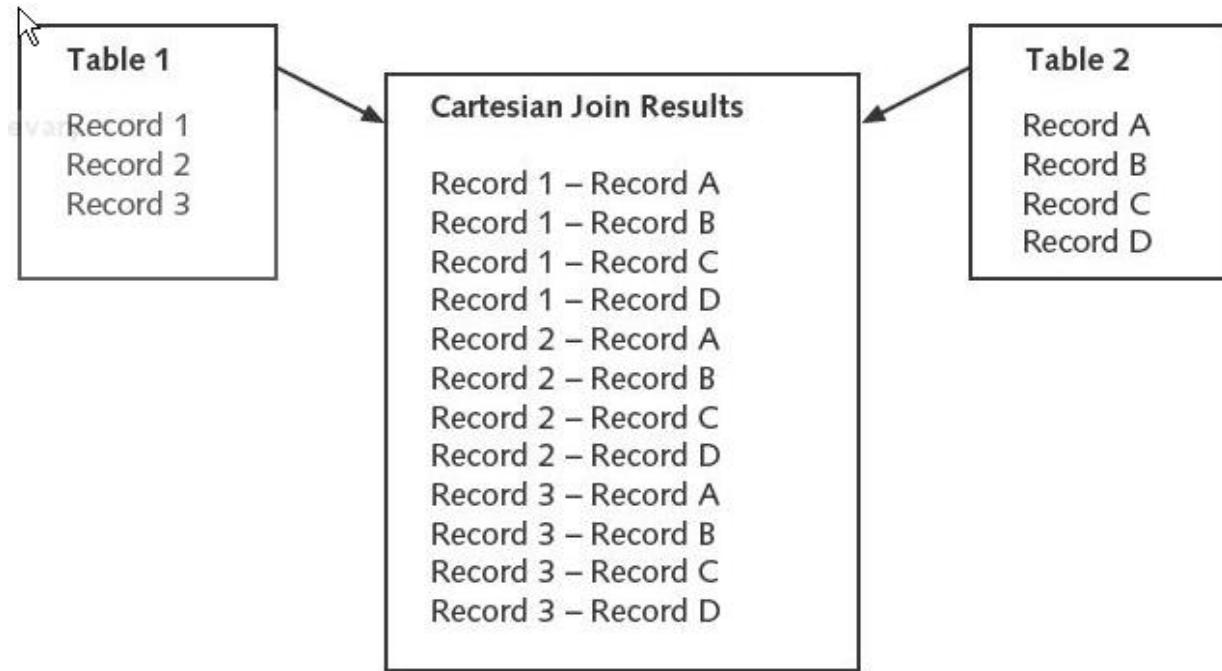
CREATE TABLE skills
(
    skill_code NUMBER,
    skill_name VARCHAR(10)
);
INSERT INTO skills VALUES (1,'fly');
INSERT INTO skills VALUES (2,'swim');
INSERT INTO skills VALUES (3,'freeze');

```

<b>Heroes</b>		<b>Skills</b>	
<u>Name</u>	<u>skill_code</u>	<u>Skill_code</u>	<u>skill_name</u>
superman	1	1	fly
Aquaman	2	2	swim
Flash	NULL	3	freeze

## 10.1 Cartesian/Cross Join

In a Cartesian join, also called a Cartesian product or cross join, each record in the first table is matched with each record in the second table. This type of join is useful when you're performing certain statistical procedures for data analysis. Therefore, if you have three records in the first table and four in the second table, the first record from the first table is matched with each of the four records in the second table. Then the second record of the first table is matched with each of the four records from the second table, and so on.



The CROSS keyword, combined with the JOIN keyword, can be used in the FROM clause to explicitly instruct Oracle to create a Cartesian (cross) join. The CROSS JOIN keywords instruct the database system to create cross-products, using all records of the tables listed in the query.

**Example 10.1a (Cartesian product, cross join)**

```

SELECT * FROM skills;
SELECT * FROM heroes;

--Cartesian product gives every combination.
SELECT * FROM heroes, skills;

--Alternative way to do a cartesian product is to use a cross join.
SELECT * FROM heroes CROSS JOIN skills;

```

```

SQL> SELECT * FROM skills;
SKILL_CODE SKILL_NAME
-----
1 fly
2 swim
3 freeze

SQL> SELECT * FROM heroes;
NAME      SKILL_CODE
-----
superman      1
aqua-man      2
flash

SQL> SELECT * FROM heroes, skills;
NAME      SKILL_CODE SKILL_CODE SKILL_NAME
-----
superman      1          1 fly
superman      1          2 swim
superman      1          3 freeze
aqua-man      2          1 fly
aqua-man      2          2 swim
aqua-man      2          3 freeze
flash         3          1 fly
flash         3          2 swim
flash         3          3 freeze

9 rows selected.

SQL> SELECT * FROM heroes CROSS JOIN skills;
NAME      SKILL_CODE SKILL_CODE SKILL_NAME
-----
superman      1          1 fly
superman      1          2 swim
superman      1          3 freeze
aqua-man      2          1 fly
aqua-man      2          2 swim
aqua-man      2          3 freeze
flash         3          1 fly
flash         3          2 swim
flash         3          3 freeze

9 rows selected.

```

### **Example 10.1b (Patient example)**

```

SELECT patient_id, fname, lname, salary FROM patient;
SELECT * FROM disease;
SELECT * FROM patient_disease;

--Cartesian product
SELECT fname, lname, disease_desc FROM patient, disease,
patient_disease;

--Another way of doing a cartesian product.
SELECT fname, lname, disease_desc FROM patient CROSS JOIN disease CROSS
JOIN patient_disease;

```

SQL> SELECT patient_id, fname, lname, salary FROM patient;
PATIENT_ID FNAME LNAME SALARY
-----
111 john Doe 25000
113 jill Crane
114 billy Bob 60000
SQL> SELECT * FROM disease;
DISEASE_ID DISEASE_DESC
-----
11 Cancer
22 Malaria
33 Flu
SQL> SELECT * FROM patient_disease;
PATIENT_ID DISEASE_ID
-----
111 11
111 22
113 11

```
SQL> --Cartesian product
SQL> SELECT fname, lname, disease_desc FROM patient, disease, patient_disease;
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Cancer
john	Doe	Cancer
john	Doe	Malaria
john	Doe	Malaria
john	Doe	Malaria
john	Doe	Flu
john	Doe	Flu
john	Doe	Flu
jill	Crane	Cancer
jill	Crane	Cancer
jill	Crane	Cancer
jill	Crane	Malaria
jill	Crane	Malaria
jill	Crane	Malaria
jill	Crane	Flu
jill	Crane	Flu
jill	Crane	Flu
billy	Bob	Cancer
billy	Bob	Cancer
billy	Bob	Cancer
billy	Bob	Malaria
billy	Bob	Malaria
billy	Bob	Malaria
billy	Bob	Flu
billy	Bob	Flu
billy	Bob	Flu

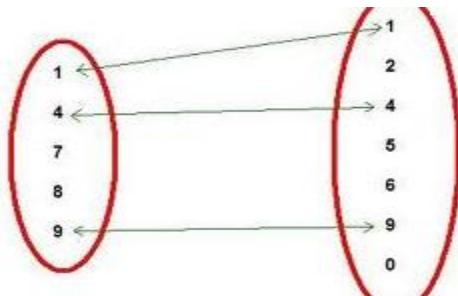
27 rows selected.

```
SQL> --Another way of doing a Cartesian product
SQL> SELECT fname, lname, disease_desc FROM patient CROSS JOIN disease CROSS JOIN patient_disease;
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Cancer
john	Doe	Cancer
john	Doe	Malaria
john	Doe	Malaria
john	Doe	Malaria
john	Doe	Flu
john	Doe	Flu
john	Doe	Flu
jill	Crane	Cancer
jill	Crane	Cancer
jill	Crane	Cancer
jill	Crane	Malaria
jill	Crane	Malaria
jill	Crane	Malaria
jill	Crane	Flu
jill	Crane	Flu
jill	Crane	Flu
billy	Bob	Cancer
billy	Bob	Cancer
billy	Bob	Cancer
billy	Bob	Malaria
billy	Bob	Malaria
billy	Bob	Malaria
billy	Bob	Flu
billy	Bob	Flu
billy	Bob	Flu

27 rows selected.

## 10.2 Inner Join



The most common type of join is based on two ( or more) tables having equivalent data stored in a common column. These joins are called equality joins but are also referred to as equijoins, inner joins, or simple joins. The traditional way to include join conditions and avoid an unintended Cartesian result is to use the WHERE clause. The WHERE clause can perform two different activities: joining tables and providing conditions to limit or filter the rows that are affected. A column qualifier indicates the table containing the column being referenced. With the equality, non- equality, and self- joins, a row is returned only if a corresponding record in each table is queried. These types of joins can be categorized as inner joins because records are listed in the results only if a match is found in each table.

### **Example 10.2a (Simple join)**

--We want to retrieve all the heroes and their corresponding skills. We have to connect the **common columns**. For this we need an inner join (Also referred to as equi-join).

--Since skill\_code appears in both tables, we have to prefix the columns with the table name  
--to avoid ambiguity.

```
SELECT * FROM heroes,skills WHERE heroes.skill_code=skills.skill_code;
```

--Since the name of the tables can be long, we can use aliases to refer to the tables. Once  
--aliases are assigned, we cannot use the table names.

```
SELECT * FROM heroes h,skills s WHERE h.skill_code=s.skill_code;
```

--Invalid: cannot use table names once aliases have been assigned.

```
SELECT * FROM heroes h,skills s WHERE heroes.skill_code=skills.skill_code;
```

--h.\* refers to all the columns in the heroes table. S.\* refers to all the columns in the skills table.

```
SELECT h.* , s.* FROM heroes h,skills s WHERE h.skill_code=s.skill_code;
```

--Display only the needed columns.

```
SELECT name, skill_name FROM heroes h,skills s WHERE  
h.skill_code=s.skill_code;
```



```

SQL> --Want to retrieve all the heroes and their appropriate skills. We have to
connect the common
SQL> --columns. For this we need an inner join (Also referred to as equi-join)
SQL> --Since skill_code appears in both tables, we have to prefix the columns w
ith the table name
SQL> --to avoid ambiguity
SQL> SELECT * FROM heroes,skills WHERE heroes.skill_code=skills.skill_code;

NAME      SKILL_CODE SKILL_CODE SKILL_NAME
-----
superman      1          1 fly
aqua-man      2          2 swim

SQL> --Since the name of the tables can be long, we can use aliases to refer to
the tables. Once
SQL> --aliases are assigned, then we cannot use the table names.
SQL> SELECT * FROM heroes h,skills s WHERE h.skill_code=s.skill_code;

NAME      SKILL_CODE SKILL_CODE SKILL_NAME
-----
superman      1          1 fly
aqua-man      2          2 swim

SQL> --invalid. Cannot use table names once aliases have been assigned
SQL> SELECT * FROM heroes h,skills s WHERE heroes.skill_code=skills.skill_code;
SELECT * FROM heroes h,skills s WHERE heroes.skill_code=skills.skill_code
*
ERROR at line 1:
ORA-00904: "SKILLS"."SKILL_CODE": invalid identifier

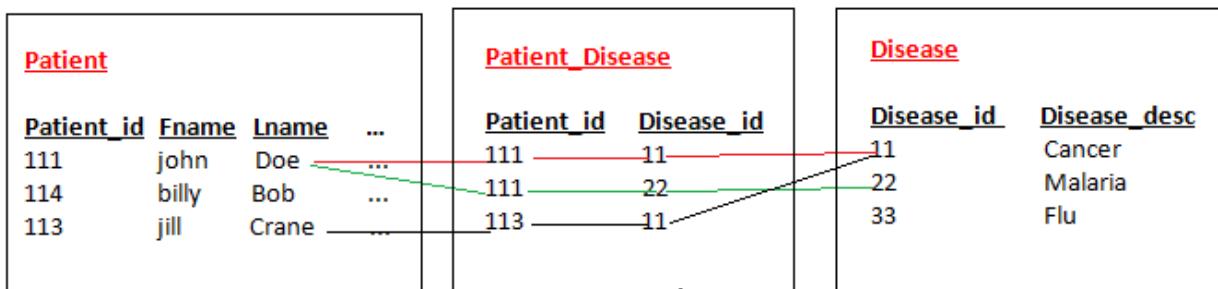
SQL> --h.* refers to all the columns in the heroes table. S.* refers to all the
columns in the skills table
SQL> SELECT h.*, s.* FROM heroes h,skills s WHERE h.skill_code=s.skill_code;

NAME      SKILL_CODE SKILL_CODE SKILL_NAME
-----
superman      1          1 fly
aqua-man      2          2 swim

SQL> --Display only the needed columns
SQL> SELECT name, skill_name FROM heroes h,skills s WHERE h.skill_code=s.skill_c
ode;

NAME      SKILL_NAME
-----
superman    fly
aqua-man    swim

```



### **Example 10.2b (Patient example)**

--We want to display the names and the diseases of the different people. Since names appear --in one table and descriptions in another, we have to do a join. The join is done by connecting --the common columns. All three tables have to be connected. If a table is included in the FROM --clause but is not connected in the WHERE clause, then the results will look like a Cartesian --product, which is more than likely not what we want.

--The columns can be connected in any order as long as all three tables have the connection.

```
SELECT fname, lname, disease_desc FROM patient p, disease d, patient_disease pd
WHERE p.patient_id=pd. patient_id AND pd.disease_id=d. disease_id;
```

--This is not what we want. Notice the disease table is not joined. The result is that patient and --patient\_disease will be inner joined. Their result will then be cross joined with --the disease table, which would logically be erroneous.

```
SELECT fname, lname, disease_desc FROM patient p, disease d, patient_disease pd
WHERE p. patient_id=pd. patient_id;
```

SQL> --We want to display the names and the diseases of the different people. Since names appear  
SQL> --in one table and descriptions in another, we have to do a join. The join is done by connecting  
SQL> --the common columns. All three tables have to be connected. If a table is included in the FROM  
SQL> --clause but is not connected in the WHERE clause, then the results will look like a Cartesian  
SQL> --product which is more than likely not what we want  
SQL> --The columns can be connected in any order as long as all three tables have the connection  
SQL> SELECT fname, lname, disease\_desc FROM patient p, disease d, patient\_disease pd
2 WHERE p.patient\_id=pd. patient\_id AND pd.disease\_id=d. disease\_id;

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer

SQL>  
SQL> -- Not what we want. Notice the disease table is not joined. The result is that patient and  
SQL> --patient\_disease will be inner joined. Their result will then be Cartesian produced with  
SQL> --the disease table which would be erroneous.  
SQL> SELECT fname, lname, disease\_desc FROM patient p, disease d, patient\_disease pd
2 WHERE p. patient\_id =pd. patient\_id;

FNAME	LNAME	DISEASE_DESC
john	Doe	Flu
john	Doe	Flu
john	Doe	Malaria
john	Doe	Malaria
john	Doe	Cancer
john	Doe	Cancer
jill	Crane	Flu
jill	Crane	Malaria
jill	Crane	Cancer

9 rows selected.

You can use three approaches to create an equality join that uses the JOIN keyword: NATURAL JOIN, JOIN ... USING, and JOIN ... ON:

- The NATURAL JOIN keywords create a join automatically between two tables, based on columns with matching names.
- The USING clause allows you to create joins based on a column that has the same name and definition in both tables.
- When the tables to be joined in a USING clause don't have a commonly named and defined field, you must add the ON clause to the JOIN keyword to specify how the tables are related.

There are two main differences between using the USING and ON clauses with the JOIN keyword:

- The USING clause can be used only if the tables being joined have a common column with the same name. This rule isn't a requirement for the ON clause.
- A condition is specified in the ON clause; this isn't allowed in the USING clause. The USING clause can contain only the name of the common column.

### ***Example 10.2c (Natural join)***

```
--Inner joins can be done using a variety of syntax. Instead of using the WHERE clause as in
--last example, the key words NATURAL JOIN can be used. It will automatically find the commonly
--named columns and connect them together.
```

```
--Also, table aliases are not allowed. Notice the skill_code appears in both tables but with this
--new syntax, we don't need to prefix the column with the table name. Natural join can figure
--things out by itself.
```

```
SELECT name, skill_code FROM heroes NATURAL JOIN skills;
```

--The order of NATURAL JOIN does not matter.

```
SELECT * FROM skills NATURAL JOIN heroes;
```

--Can also use the plain JOIN syntax and the USING clause to identify the common column.

```
SELECT name, skill_name FROM heroes JOIN skills USING (skill_code);
```

--Can use the JOIN clause and the ON keyword. This syntax begins to look like the first join that
--we did using a WHERE clause.

```
SELECT name, skill_name FROM heroes JOIN skills s ON h.skill_code=s.skill_code;
```

```

SQL> --Inner joins can be done using a variety of syntax. Instead of using the WHERE clause as in
SQL> --last example the key words NATURAL JOIN can be used. It will automatically find the commonly
SQL> --named columns and connect them together.
SQL> --Also table aliases are not allowed. Notice the skill_code appears in both tables but with this
SQL> --new syntax, we don't need to prefix the column with the table name. Natural join can figure
SQL> --things out by itself
SQL> SELECT name, skill_code FROM heroes NATURAL JOIN skills;

NAME      SKILL_CODE
-----
superman      1
aqua-man      2

SQL> --The order of NATURAL JOIN does not matter
SQL> SELECT * FROM skills NATURAL JOIN heroes;

SKILL_CODE SKILL_NAME NAME
-----
1 fly        superman
2 swim       aqua-man

SQL> --Can also use the plain JOIN syntax and the USING clause to identify the common column
SQL> SELECT name, skill_name FROM heroes JOIN skills USING (skill_code);

NAME      SKILL_NAME
-----
superman    fly
aqua-man    swim

SQL> --Can use the JOIN clause and the ON keyword. This syntax begins to look like the first join that
SQL> --we did using a WHERE clause
SQL> SELECT name, skill_name FROM heroes h JOIN skills s ON h.skill_code=s.skill_code;

NAME      SKILL_NAME
-----
superman    fly
aqua-man    swim

```

### ***Example 10.2d (Patient Example)***

```

--Can do a natural join against multiple tables.
--Order does not matter when using natural join.
SELECT fname, lname, disease_desc FROM patient NATURAL JOIN disease NATURAL
JOIN patient_disease;

-- ORDER DOES NOT MATTER (NATURAL JOIN)
SELECT fname, lname, disease_desc FROM patient NATURAL JOIN patient_disease
NATURAL JOIN disease ;

```

```

SQL> --Can do a natural join against multiple tables
SQL> -- ORDER DOES NOT MATTER (NATURAL JOIN)
SQL> SELECT fname, lname, disease_desc FROM patient NATURAL JOIN disease NATURAL JOIN patient_disease;

FNAME          LNAME          DISEASE_DESC
-----          -----          -----
john           Doe            Cancer
john           Doe            Malaria
jill           Crane          Cancer

SQL> -- ORDER DOES NOT MATTER (NATURAL JOIN)
SQL> SELECT fname, lname, disease_desc FROM patient NATURAL JOIN patient_disease NATURAL JOIN disease ;

FNAME          LNAME          DISEASE_DESC
-----          -----          -----
john           Doe            Cancer
john           Doe            Malaria
jill           Crane          Cancer

```

### *Example 10.2e (Natural join with multiple columns)*

```

Drop TABLE a;
DROP TABLE b;
CREATE TABLE A
(
    COLA NUMBER,
    COLB NUMBER,
    COLC NUMBER
);

INSERT INTO a VALUES (1,1,1);
INSERT INTO a VALUES (2,2,2);
INSERT INTO a VALUES (3,3,3);
INSERT INTO a VALUES (4,4,4);

CREATE TABLE B (COLd NUMBER, COLB NUMBER, COLA NUMBER);
INSERT INTO b VALUES (6,1,1);
INSERT INTO b VALUES (9,3,2);
INSERT INTO b VALUES (7,3,3);
INSERT INTO b VALUES (5,5,5);

--Matches on all the columns that have the same name.
--If it doesn't find any matches between the column names, then it works like a cross join.
SELECT * FROM a NATURAL JOIN b;

```

```

SQL> --Matches on all the columns that have the same name
SQL> --If it doesn't find any matches between the column names, then it works like a cross join
SQL> SELECT * FROM a NATURAL JOIN b;

```

COLA	COLB	COLC	COLD
1 3	1 3	1 3	6 7

### **Example 10.2f (Patient Example with GROUP BY)**

--In this example, we are trying to find the number of diseases each person has as long as they  
--they have more than one disease. Since the name comes from the patient table but the actual disease  
--association comes from patient\_disease, we have to do an inner join.  
--ERROR: Alias names cannot be used in the GROUP BY or the HAVING clause.

```
SELECT fname firstname, count(*) DiseaseCount
  FROM patient p, patient_disease pd
 WHERE p.patient_id=pd. patient_id
   GROUP BY firstName HAVING DiseaseCount >1;
```

--This is a correction to the above statement.

```
SELECT fname, count(*) DiseaseCount FROM patient p, disease d,
patient_disease pd WHERE p.patient_id=pd. patient_id and
pd.disease_id=d. disease_id GROUP BY fname HAVING count(*) >1;
```

SQL> -- ERROR: Alias names cannot be used in the GROUP BY or the HAVING clause

```
SQL> SELECT fname firstname, count(*) DiseaseCount
  2  FROM patient p, patient_disease pd
  3  WHERE p.patient_id=pd. patient_id
  4  GROUP BY firstName HAVING DiseaseCount >1;
  GROUP BY firstName HAVING DiseaseCount >1
  *
```

ERROR at line 4:

ORA-00904: "DISEASECOUNT": invalid identifier

SQL>

SQL> --This is a correction to the above statement

```
SQL> SELECT fname, count(*) DiseaseCount FROM patient p, disease d, patient_disease pd
  2 WHERE p.patient_id=pd. patient_id and pd.disease_id=d. disease_id GROUP BY fname HAVING count(*) >1;
```

FNAME	DISEASECOUNT
john	2

A **non-equality join** is used when the related columns can't be joined with an equal sign— meaning there are no equivalent rows in the tables to be joined.

```
DROP TABLE students;
DROP TABLE grade_range;

CREATE TABLE students
(
name VARCHAR(10),
score NUMBER
);
INSERT INTO students VALUES ('jack',80);
INSERT INTO students VALUES ('scott',73);
```

```
CREATE TABLE grade_range
(
```

```

beg_score NUMBER,
end_score NUMBER,
grade char
);

INSERT INTO grade_range VALUES (90,100,'A');
INSERT INTO grade_range VALUES (80,89,'B');
INSERT INTO grade_range VALUES (70,79,'C');
INSERT INTO grade_range VALUES (60,69,'D');

```

<u>students</u>	<u>Grade range</u>
<u>Name score</u>	<u>Beg score end_score grade</u>
Jack 80	90 100 A
Scott 73	80 89 B
	70 79 C
	60 69 D

### Example 10.2g (Non-equi join)

SELECT \* FROM students;  
SELECT \* FROM grade\_range;

--A non-equi join is like a inner join in that it joins records from multiple tables but the columns  
--may not have the same name. In other words, there may not be a foreign key relationship.

SELECT name,score,grade FROM students,grade\_range WHERE  
score BETWEEN beg\_score AND end\_score;

SQL> SELECT \* FROM students;

NAME	SCORE
jack	80
scott	73

SQL> SELECT \* FROM grade\_range;

BEG_SCORE	END_SCORE	G
90	100	A
80	89	B
70	79	C
60	69	D

SQL> --A non-equi join is like a inner join in that it joins records from multiple tables but the columns  
SQL> --may not have the same name. In other words, there may not be a foreign key relationship

SQL> SELECT name,score,grade FROM students,grade\_range WHERE  
2 score BETWEEN beg\_score AND end\_score;

NAME	SCORE	G
jack	80	B
scott	73	C

### **Example 10.2h (Numbering each line)**

```
CREATE TABLE student
(
Name VARCHAR2(10),
Class VARCHAR2(10)
);

INSERT INTO student VALUES ('abdul','philosophy');
INSERT INTO student VALUES ('bob','philosophy');
INSERT INTO student VALUES ('doe','philosophy');
INSERT INTO student VALUES ('jack','Religion');
INSERT INTO student VALUES ('kennedy','Religion');
INSERT INTO student VALUES ('jim','Science');
INSERT INTO student VALUES ('jones','Science');
INSERT INTO student VALUES ('harry','Science');
INSERT INTO student VALUES ('potter','Science');
```

*/\*Below is the desired result set that we are looking for. We want to get a count sequence for each of the different categories. Notice this is not like a group by in that we are not trying to suppress any information. We want to number our records. \*/*

<b>1</b>	<b>abdul</b>	<b>philosophy</b>
<b>2</b>	<b>bob</b>	<b>philosophy</b>
<b>3</b>	<b>doe</b>	<b>philosophy</b>
1	jack	Religion
2	kennedy	Religion
<b>1</b>	<b>jim</b>	<b>Science</b>
<b>2</b>	<b>jones</b>	<b>Science</b>
<b>3</b>	<b>harry</b>	<b>Science</b>
<b>4</b>	<b>potter</b>	<b>Science</b>

*/\* Rownum is a pseudo-column that is available to us. It is a number that Oracle assigns to each record in the order in which the records were either physically or virtually inserted into the table. Create a table (temp1) with new rownumber. \*/*

```
CREATE TABLE temp1 AS
SELECT rownum AS line_number, name, class
FROM student;
```

<b>1</b>	<b>abdul</b>	<b>philosophy</b>
<b>2</b>	<b>bob</b>	<b>philosophy</b>
<b>3</b>	<b>doe</b>	<b>philosophy</b>
4	jack	Religion
5	kennedy	Religion
<b>6</b>	<b>jim</b>	<b>Science</b>
<b>7</b>	<b>jones</b>	<b>Science</b>
<b>8</b>	<b>harry</b>	<b>Science</b>
<b>9</b>	<b>potter</b>	<b>Science</b>

--Create a table (temp2) that identifies the beginning point.

```
CREATE TABLE temp2 AS
SELECT min(rownum) AS beg_line_number, class
FROM student GROUP BY class
```

<b>beg_line_number</b>	<b>class</b>
6	Science
4	Religion
1	Philosophy

<b>Temp1</b>			<b>Temp2</b>
6	jim	Science	6 Science
7	jones	Science	
8	harry	Science	
9	potter	Science	
4	jack	Religion	4 Religion
5	kennedy	Religion	
1	abdul	philosophy	1 philosophy
2	bob	philosophy	
3	dole	philosophy	

```
SELECT (temp1.line_number - temp2.beg_line_number)+1 , name, temp1.class
FROM temp1, temp2
WHERE temp1.class=temp2.class
ORDER BY 3,1;
```

<b>Num</b>	<b>Name</b>	<b>Class</b>
1	abdul	philosophy
2	bob	philosophy
3	dole	philosophy
1	jack	Religion
2	kennedy	Religion
1	jim	Science
2	jones	Science
3	harry	Science
4	potter	Science

```

select * from people p, personanlity pp where p.pid = pp.pid;
select * from people natural join persoanlity ;
select * from people cross join persoanlity;

```

200

## ✓ CHECK 10A

1. Display all the people and all the potential personality types that they can have. Display name and personality description
  - a. With and without CROSS JOIN
2. Display the name and personailty description of all those people who have a personality
  - a. Use both the old and new Oracle syntax

*"I am not young enough to know everything"*

```

select name, p_desc from person p, persoanlity pp where
p.pid = pp.pid;

```

## 10.3 Self Join

Sometimes data in one column of a table has a relationship with another column in the same table. This type of join is known as a self- join.

```

DROP TABLE employee;
CREATE TABLE employee
(
    ssn      VARCHAR2(11),
    name     VARCHAR2(11),
    manager  VARCHAR2(11),
    salary   NUMBER
);
INSERT INTO employee VALUES ('111','jack','222',10000);
INSERT INTO employee VALUES ('333','john','222',20000);
INSERT INTO employee VALUES ('444','jill','111',10000);
INSERT INTO employee VALUES ('222','joe','999',10000);

```

<u>e</u>				<u>M</u>			
<b>SSN</b>	<b>Name</b>	<b>Manager</b>	<b>Salary</b>	<b>SSN</b>	<b>Name</b>	<b>Manager</b>	<b>Salary</b>
111	jack	222	10000	111	jack	222	10000
333	john	222	20000	333	john	222	20000
444	jill	111	10000	444	jill	111	10000
222	joe	999	10000	222	joe	999	10000

```
SELECT * FROM employee;
```

*/\*In this example, we want to find the names of all the employees and their managers. Notice that both names reside in the same table. We can do a join against the same table and assign a different alias to each table making it appear as if we have two separate tables. We can then connect the foreign key (manager) to the primary key (ssn). Notice the use of the alias before each of the columns because they appear in both tables. Without the alias we would get an ambiguously defined column error. Also Joe is not included in the result because Manager (999) does not match with any ssns. \*/*

```
SELECT e.name EMPLOYEE, m.name Manager FROM employee e, employee m
WHERE e.manager=m.ssn;
```

--We want to find all the people who are making the same salary.

-- Problem: This is not what we want because it duplicates the entries.

```
SELECT e1.name, e1.salary FROM employee e1, employee e2
WHERE e1.salary=e2.salary AND e1.ssn!=e2.ssn ;
```

--This resolves the duplicate issue.

```
SELECT DISTINCT e1.name, e1.salary FROM employee e1, employee e2
WHERE e1.salary=e2.salary AND e1.ssn!=e2.ssn ;
```

SSN	NAME	MANAGER_SSN	SALARY
111	jack	222	10000
333	john	222	20000
444	jill	111	10000
222	joe	999	10000

SQL> --In this example we want to find the names of all the employees and their managers. Notice  
SQL> --that both names reside in the same table. We can do a join against the same table and assign  
SQL> --a different alias to each table making it appear as if we have two separate tables. We can then  
SQL> --connect the foreign key (manager) to the primary key(ssn)  
SQL> --Notice the use of the alias before each of the columns because they appear in both tables.  
SQL> --Without the alias we would get an ambiguously defined column error  
SQL> --Also Joe is not included in the result because Manager (999) does not match with any ssns  
SQL> SELECT e.name EMPLOYEE , m.name Manager FROM employee e, employee m  
2 WHERE e.manager\_ssn=m.ssn;

EMPLOYEE	MANAGER
jill	jack
john	joe
jack	joe

SQL>  
SQL>

SQL> --We want to find all the people who are making the same salary  
SQL> -- Problem: This is not what we want because it duplicates the entries  
SQL> SELECT e1.name, e1.salary FROM employee e1, employee e2  
2 WHERE e1.salary=e2.salary AND e1.ssn!=e2.ssn ;

NAME	SALARY
joe	10000
jill	10000
joe	10000
jack	10000
jill	10000
jack	10000

6 rows selected.

SQL>  
SQL> --This resolves the duplicate issue  
SQL> SELECT DISTINCT e1.name, e1.salary FROM employee e1, employee e2  
2 WHERE e1.salary=e2.salary AND e1.ssn!=e2.ssn ;

NAME	SALARY
jill	10000
joe	10000
jack	10000

## 10.4 Outer Join



ColA	ColB	ColC
a	1	c
aa	2	cc
aaa	3	ccc
aaaaa	5	ccccc

ColD	ColB	ColE	ColF
d	1	e	f
dd	2	ee	ff
ddd	3	eee	fff
dddd	4	eeee	ffff

ColA	ColB	ColC	ColD	ColE	ColF
a	1	c	d	e	f
aa	2	cc	dd	ee	ff
aaa	3	ccc	ddd	eee	fff
aaaaa	5	ccccc	NULL	NULL	NULL

ColA	ColB	ColC	ColD	ColE	ColF
a	1	c	d	e	f
aa	2	cc	dd	ee	ff
aaa	3	ccc	ddd	eee	fff
NULL	4	NULL	dddd	eeee	ffff

To tell Oracle to create NULL rows for records that don't have a matching row, use an outer join operator, which looks like this: (+). It's placed in the WHERE clause immediately after the column name from the table that's missing the matching row. It tells Oracle to create a NULL row in that table to join with the row in the other table. You need to be aware of some limitations when using the traditional approach to outer joins: The outer join operator can be used for only one table in the joining condition. In other words, you can't create NULL rows in both tables at the same time. A condition that includes the outer join operator can't use the IN or OR operator.

```

drop table a;
drop table b;
drop table c;

create table a( col1 char(3) );
create table b( col2 char(3) );
create table c( col3 char(3) );

insert into a values ('a');
insert into a values ('ab');
insert into a values ('ac');
insert into a values ('abc');

insert into b values ('b');
insert into b values ('ab');
insert into b values ('bc');
insert into b values ('abc');

```

```
insert into c values ('c');
insert into c values ('bc');
insert into c values ('ac');
insert into c values ('abc');
```

A	B	C
COL1	COL2	COL3
-----	-----	-----
a	b	c
ab	ab	bc
ac	bc	ac
abc	abc	abc

### Example 10.4a (Inner join)

-- Inner join: Connect all three tables otherwise we will get a Cartesian product.

```
select * from a,b,c where col1=col2 and col2=col3;
```

SQL> -- Inner join: Connect all three tables otherwise we will get a Cartesian product  
SQL> select \* from a,b,c where col1=col2 and col2=col3;

```
COL1 COL2 COL3
----- -----
abc abc abc
```

### Example 10.4b (One outer join condition)

--The (+) is used for outer join which means (You don't really care). When you don't care, it creates a virtual NULL record behind the scenes. An outer join is first and foremost an inner join. --Then if there is a record for which there is not a match, the (+) says, it is okay and will allow it to go through.

--In this case, if there is something in col1 for which there is no match in col2, it will automatically create a NULL record in col2. The problem is that then the NULL record will be compared to the data in col3. NULLs cannot be checked in this way and so in this case the (+) means nothing.

```
select * from a,b,c where col1=col2(+) and col2=col3;
```

SQL> --In col2, NULLs cannot be checked in this way and so in this case the (+) means nothing  
SQL> select \* from a,b,c where col1=col2(+) and col2=col3;

```
COL1 COL2 COL3
----- -----
abc abc abc
```

### **Example 10.4c (One outer join condition)**

--In this example, we find all the common records to a, b and c. In addition, we find all the --records that are common to (a) and (c).  
--It checks (ac) against matches in col2 but it cannot find any. So it sets it to NULL because of (+). It then --checks for (ac) in col3 and it finds a match.

```
select * from a,b,c where col1=col2(+) and col1=col3;
```

```
SQL> select * from a,b,c where col1=col2(+) and col1=col3;
COL1 COL2 COL3
-----
ac      ac
abc    abc  abc
```

### **Example 10.4d (One outer join condition)**

--Common to all and also to (a) and (b).  
--It looks for (ab) in col3 but doesn't find a match. So it sets it to NULL. It checks (ab) to col2 and --does find a match.

```
select * from a,b,c where col1=col3(+) and col1=col2;
```

```
SQL> select * from a,b,c where col1=col3(+) and col1=col2;
COL1 COL2 COL3
-----
abc    abc  abc
ab     ab
```

### **Example 10.4e (One outer join condition)**

--Common to all and also to b and c.  
--Notice that col1 is not included, which means that we will have a Cartesian product .  
--(abc) in col2 will be found in col3 and also (bc) will be found in col3. The other records that are --in col2, which are not in col3, will go through because of the (+). However, they would be ignored --because it is looking to match that data with col3 again. The results (ab, abc) will be cross joined -- with every record in table (a).

```
select * from a,b,c where col2=col3(+) and col2=col3;
```

```
SQL> select * from a,b,c where col2=col3(+) and col2=col3;
```

```
COL1 COL2 COL3
-----
a      bc   bc
ab    bc   bc
ac    bc   bc
abc   bc   bc
a      abc  abc
ab    abc  abc
ac    abc  abc
abc   abc  abc
```

8 rows selected.

### **Example 10.4f (Two outer join conditions)**

--All the intersecting points are displayed.  
--If there is something that is col1 but not in col2, the (+) says it is okay. It then creates a NULL record.  
--If there is something in col1 but not in col3, the (+) says it is okay. It also creates a NULL record.

```
select * from a,b,c where col1=col2(+) and col1=col3(+);
```

```
SQL> select * from a,b,c where col1=col2(+) and col1=col3(+);
COL1 COL2 COL3
---- ---- -
ac      ac
abc    abc  abc
ab     ab
a
```

### **Example 10.4g (Two outer join conditions)**

--Displays all the intersecting points.  
--If there is something that is in col1 but not in col2, the (+) says it is okay and it creates a NULL record.  
--If there is something in col2 but not in col3, the (+) says it is okay and also creates a NULL record.

```
select * from a,b,c where col1=col2(+) and col2=col3(+);
```

```
SQL> select * from a,b,c where col1=col2(+) and col2=col3(+);
COL1 COL2 COL3
---- ---- -
abc    abc  abc
a
ac
ab    ab
```

### **Example 10.4h (Plus sign on only one side)**

-- Invalid: Only one + sign can be used.

```
select * from a,b where col1(+) = col2(+);
```

```
SQL> select * from a,b where col1(+) = col2(+);
select * from a,b where col1(+) = col2(+)
*
ERROR at line 1:
ORA-01468: a predicate may reference only one outer-joined table
```

### **Example 10.4i (Inner join)**

```
SELECT * FROM patient;
SELECT * FROM disease;
SELECT * FROM patient_disease;
```

--This is an inner join, which gives us a listing of all the patient names and their disease descriptions.

```
SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p.patient_id = pd.patient_id and pd.disease_id = d.disease_id;
```

```

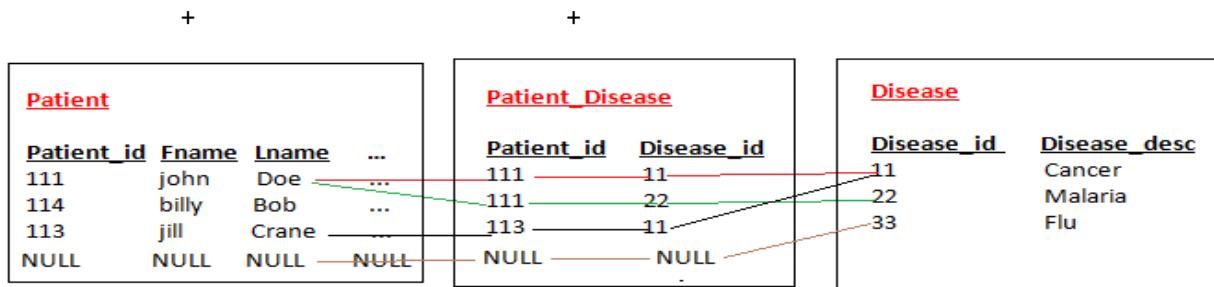
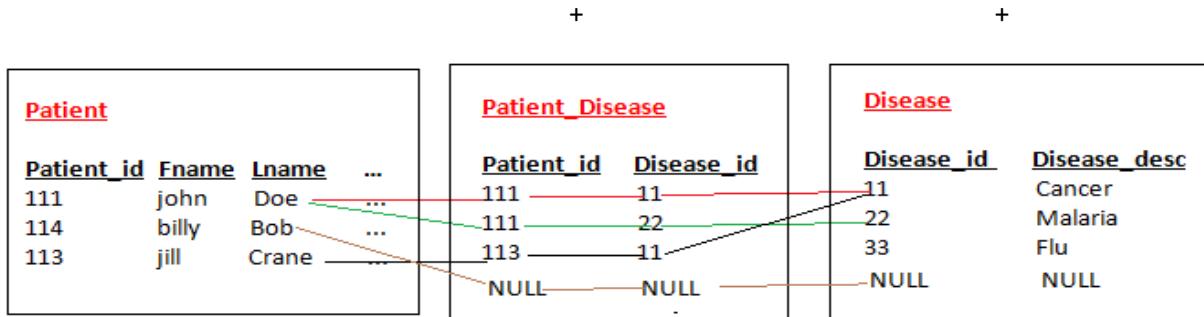
SQL> SELECT * FROM disease;
DISEASE_ID DISEASE_DESC
-----
11 Cancer
22 Malaria
33 Flu

SQL> SELECT * FROM patient_disease;
PATIENT_ID DISEASE_ID
-----
111      11
111      22
113      11

SQL> --This is an inner join which gives us a listing of all the patient names and their disease descriptions
SQL> SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
  2 WHERE p.patient_id=pd.patient_id AND pd.disease_id=d.disease_id;

FNAME          LNAME          DISEASE_DESC
-----          -----          -----
john           Doe            Cancer
john           Doe            Malaria
jill           Crane          Cancer

```



### *Example 10.4j (Including records that don't match with anything else)*

--Select all the people and their disease descriptions. Also, include in the result set the individuals  
--who are not sick. In this case (billy bob) is not sick.

```

SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p.patient_id = pd.patient_id (+) AND pd.disease_id = d.disease_id (+);

```

--Select all the people and their disease descriptions. Also, include in the result set the diseases  
--that are not associated with any individual. In this case (flu) is not associated with anyone.

```
SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p. patient_id (+)=pd. patient_id AND pd. disease_id (+)=d. disease_id;
```

```
SQL> SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
2 WHERE p. patient_id =pd. patient_id (+) AND pd. disease_id =d. disease_id (+);
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer
billy	Bob	

SQL>

SQL> --Pick up all the people and their disease descriptions. Also include in the result set the diseases

SQL> --that are not associated with any individual. In this case (flu) is not associated with anyone

```
SQL> SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
2 WHERE p. patient_id (+)=pd. patient_id AND pd. disease_id (+)=d. disease_id;
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer
		Flu

### ***Example 10.4k (All join conditions must be included)***

--This is not what we want. It is missing a join condition. Given the inner join between the patient and  
--patient disease, along with the individuals who are not associated with any diseases, which is  
--the outer join, we will Cartesian product the result set with the disease table.

```
SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p. patient_id= pd. patient_id(+);
```

--This is not what we want. In this case the (+) is associated with the wrong table. In outer joins we want  
--to include records that are not matched in some other table. In this case, all the records in the  
--patient\_disease are matched up in the patient table. The (+) is extreaneous.

```
SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p. patient_id(+) =pd. patient_id AND pd. disease_id =d. disease_id;
```

```
SQL> --the outer join, we will cartesian product that result set with the disease table
SQL> SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
2 WHERE p. patient_id =pd. patient_id (+);
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
john	Doe	Flu
john	Doe	Cancer
john	Doe	Malaria
john	Doe	Flu
jill	Crane	Cancer
jill	Crane	Malaria
jill	Crane	Flu
billy	Bob	Cancer
billy	Bob	Malaria
billy	Bob	Flu

12 rows selected.

```
SQL>
SQL> -- Not what we want. In this case the (+) is associated with the wrong table. In outer joins we want
SQL> --to include records that are not matched in some other table. In this case, all the records in the
SQL> --patient_disease are matched up in the patient table so the (+) is extreaneous
SQL> SELECT fname, lname, disease_desc FROM patient p, patient_disease pd, disease d
2 WHERE p. patient_id(+) =pd. patient_id AND pd. disease_id =d. disease_id;
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer

When creating a traditional outer join with the outer join operator, the join can be applied to only one table—not both. However, with the JOIN keyword, you can specify which table the join should be applied to by using a left, right, or full outer join. Left and right outer joins specify which table the outer join should be applied to, based on the table’s location in the join condition. For example, a left outer join instructs Oracle to keep any rows in the table listed on the left side of the join condition, even if no matches are found with the table listed on the right. A full outer join keeps all rows from both tables in the results, no matter which table is deficient when matching rows. (That is, it performs a combination of left and right outer joins.)

### Example 10.4L (Using join syntax)

--Can use the alternate syntax of LEFT or RIGHT OUTER JOIN to replace the old (+).  
--Notice that in this syntax, disease is on the left of the LEFT OUTER JOIN syntax whereas  
--patient\_disease is on the right. This would mean that aside from the inner join, we want to  
--include records in the disease table that are not the patient\_disease table.

```
SELECT patient_id, disease_desc from disease d LEFT OUTER JOIN patient_disease
pd ON d. disease_id =pd. disease_id;
```

--Notice that in this syntax, disease is on the right of the RIGHT OUTER JOIN syntax whereas  
--patient\_disease is on the left. This would mean that aside from the inner join, we want to  
--include records in the disease table that are not in the patient\_disease table.

```
SELECT patient_id, disease_desc from patient_disease pd RIGHT OUTER JOIN
disease d ON pd. disease_id =d. disease_id;
```

--Here is how we accomplish the same thing using the old (+) syntax.

```
SELECT patient_id, disease_desc FROM patient_disease pd, disease d WHERE pd.
disease_id (+)=d. disease_id;
```

SQL> --include records in the disease table that are not the patient\_disease table

```
SQL> SELECT patient_id, disease_desc from disease d LEFT OUTER JOIN patient_disease pd ON d. disease_id =pd.
disease_id;
```

PATIENT\_ID DISEASE\_DESC

```
-----
111 Cancer
111 Malaria
113 Cancer
Flu
```

SQL>

SQL> --Notice that in this syntax, disease is on the right of the RIGHT OUTER JOIN syntax whereas

SQL> --patient\_disease is on the left. This would mean that aside from the inner join, we want to

SQL> --include records in the disease table that are not the patient\_disease table

```
SQL> SELECT patient_id, disease_desc from patient_disease pd RIGHT OUTER JOIN disease d ON pd. disease_id =
d. disease_id;
```

PATIENT\_ID DISEASE\_DESC

```
-----
111 Cancer
111 Malaria
113 Cancer
Flu
```

SQL>

SQL> --Here is how we accomplish the same thing using the old (+) syntax

```
SQL> SELECT patient_id, disease_desc FROM patient_disease pd, disease d WHERE pd. disease_id (+)=d. disease
_id;
```

PATIENT\_ID DISEASE\_DESC

```
-----
111 Cancer
111 Malaria
113 Cancer
Flu
```

### Example 10.4m (Left and right outer join syntax)

--Notice that in this syntax, patient is on the left of the LEFT OUTER JOIN syntax whereas  
--patient\_disease is on the right. This would mean that aside from the inner join, we want to  
--include records in the patient table that are not the patient\_disease table.

```
SELECT fname, lname, disease_id
FROM patient p
LEFT OUTER JOIN patient_disease pd
ON p.patient_id = pd.patient_id;
```

--Notice that in this syntax, patient is on the right of the RIGHT OUTER JOIN syntax whereas  
--patient\_disease is on the left. This would mean that aside from the inner join, we want to  
--include records in the patient table that are not the patient\_disease table.

```
SELECT fname, lname, disease_id
FROM patient_disease pd
RIGHT OUTER JOIN patient p
ON p.patient_id = pd.patient_id;
```

--Here is how we accomplish the same thing using the old (+) syntax.

```
SELECT fname, lname, disease_id
FROM patient_disease pd, patient p
WHERE pd.patient_id (+)=p.patient_id;
```

SQL> --Include records in the patient table that are not the patient\_disease table

```
SQL> SELECT fname, lname, disease_id
      FROM patient p
      LEFT OUTER JOIN patient_disease pd
      ON p.patient_id = pd.patient_id;
```

FNAME	LNAME	DISEASE_ID
john	Doe	11
john	Doe	22
jill	Crane	11
billy	Bob	

SQL>

--Notice that in this syntax, patient is on the right of the RIGHT OUTER JOIN syntax whereas  
--patient\_disease is on the left. This would mean that aside from the inner join, we want to  
--include records in the patient table that are not the patient\_disease table

SQL>

```
SQL> SELECT fname, lname, disease_id
      FROM patient_disease pd
      RIGHT OUTER JOIN patient p
      ON p.patient_id = pd.patient_id;
```

FNAME	LNAME	DISEASE_ID
john	Doe	11
john	Doe	22
jill	Crane	11
billy	Bob	

SQL>

--Here is how we accomplish the same thing using the old (+) syntax

SQL>

```
SQL> SELECT fname, lname, disease_id
      FROM patient_disease pd, patient p
      WHERE pd.patient_id (+)=p.patient_id;
```

FNAME	LNAME	DISEASE_ID
john	Doe	11
john	Doe	22
jill	Crane	11
billy	Bob	

### Example 10.4n (Left and right outer join syntax)

--First we will do the inner join between the three tables.

--Second we will do the LEFT OUTER JOIN between the disease and patient\_disease which means  
--that we are including the diseases that are in the disease table that are not associated with anyone.  
--This extra record would now be sitting on the left hand side of the second LEFT OUTER JOIN which  
--says to include that in the result set as well.

```
SELECT fname, lname, disease_desc
FROM disease d
LEFT OUTER JOIN patient_disease pd
ON d.disease_id = pd.disease_id
LEFT OUTER JOIN patient p
ON pd.patient_id = p.patient_id;
```

--First we will do the inner join between the three tables.

--Second we will do the RIGHT OUTER JOIN between the disease and patient\_disease which means

--that we are including the diseases that are in the disease table that are not associated with anyone.

--This extra record would now be sitting on the left hand side of the LEFT OUTER JOIN which

--says to include that in the result set as well.

```
SELECT fname, lname, disease_desc from patient_disease pd RIGHT OUTER JOIN disease d ON pd.disease_id=d.disease_id LEFT OUTER JOIN patient p ON pd.patient_id=p.patient_id;
```

--The same thing can be done with the old syntax.

```
SELECT fname, lname, disease_desc FROM patient_disease pd, disease d , patient p WHERE pd.disease_id (+)=d.disease_id AND pd.patient_id =p.patient_id (+);
```

SQL> --says to include that in the result set as well

```
SQL> SELECT fname, lname, disease_desc from disease d LEFT OUTER JOIN patient_disease pd ON d.disease_id =pd.disease_id LEFT OUTER JOIN patient p ON pd.patient_id=p.patient_id;
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer
		Flu

SQL>

SQL> --First we will do the inner join between the three tables

SQL> --Second we will do the RIGHT OUTER JOIN between the disease and patient\_disease which means

SQL> --that we are including the diseases that are in the disease table that are not associated with anyone

SQL> --This extra record would now be sitting on the left hand side of the LEFT OUTER JOIN which

SQL> --says to include that in the result set as well

```
SQL> SELECT fname, lname, disease_desc from patient_disease pd RIGHT OUTER JOIN disease d ON pd.disease_id =d.disease_id LEFT OUTER JOIN patient p ON pd.patient_id =p.patient_id;
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer
		Flu

SQL>

SQL> --The same thing can be done with the old syntax

```
SQL> SELECT fname, lname, disease_desc FROM patient_disease pd, disease d , patient p WHERE pd.disease_id (+)=d.disease_id AND pd.patient_id =p.patient_id (+);
```

FNAME	LNAME	DISEASE_DESC
john	Doe	Cancer
john	Doe	Malaria
jill	Crane	Cancer
		Flu

### Example 10.4o (Full outer join conditions)

ColA	ColB	ColC
a	1	c
aa	2	cc
aaa	3	ccc
aaaa	5	cccc

ColD	ColB	ColE	ColF
d	1	e	f
dd	2	ee	ff
ddd	3	eee	fff
dddd	4	eeee	ffff

Inner Join

ColA	ColB	ColC	ColD	ColE	ColF
a	1	c	d	e	f
aa	2	cc	dd	ee	ff
aaa	3	ccc	ddd	eee	fff

Left Outer Join

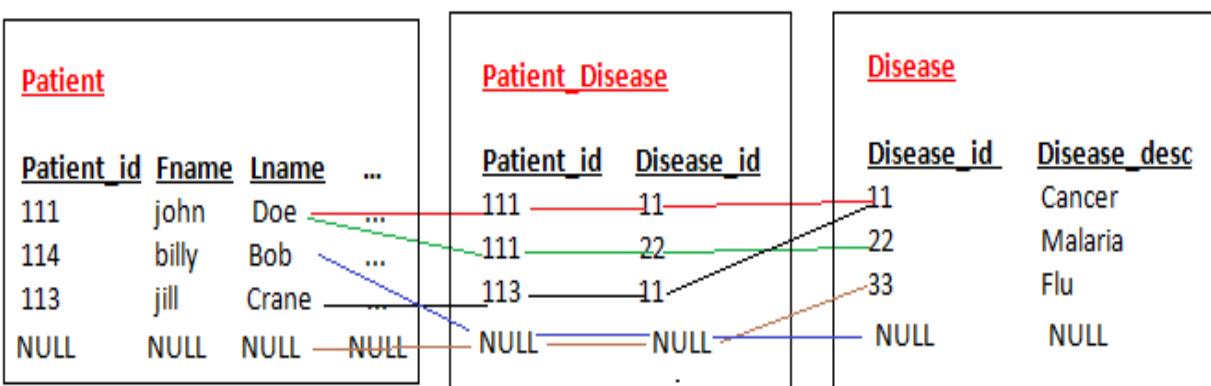
ColA	ColB	ColC	ColD	ColE	ColF
a	1	c	d	e	f
aa	2	cc	dd	ee	ff
aaa	3	ccc	ddd	eee	fff
aaaa	5	cccc	NULL	NULL	NULL

Right Outer Join

ColA	ColB	ColC	ColD	ColE	ColF
a	1	c	d	e	f
aa	2	cc	dd	ee	ff
aaa	3	ccc	ddd	eee	fff
NULL	4	NULL	dddd	eeee	ffff

Full Outer Join

ColA	ColB	ColC	ColD	ColE	ColF
a	1	c	d	e	f
aa	2	cc	dd	ee	ff
aaa	3	ccc	ddd	eee	fff
NULL	4	NULL	dddd	eeee	ffff
aaaa	5	cccc	NULL	NULL	NULL



--Full outer is equivalent to a LEFT and RIGHT OUTER JOIN at the same time. This means that  
--in addition to an inner join, include the stuff on the left hand side for which there is no match  
--no the right hand side. Also include the stuff on the right hand side for which there is no  
--match on the left hand side.

```
SELECT pd.disease_id, d.disease_desc FROM disease d FULL OUTER JOIN
patient_disease pd ON d.disease_id=pd.disease_id;
SQL> SELECT pd.disease_id, d.disease_desc FROM disease d FULL OUTER JOIN patient_disease pd ON d.disease_id =pd.disease_id;
```

DISEASE_ID	DISEASE_DESC
11	Cancer
22	Malaria
11	Cancer
	Flu

--This is the same as above because all the records that are in the patient\_disease table are matched  
--against the records in the disease table because it is a bridge table.

```
--SELECT pd.disease_id, d.disease_desc FROM disease d LEFT OUTER JOIN patient_disease pd ON
--d.disease_id =pd.disease_id;
```

--Same scenario as above except that it is using the patient table instead of the disease table.

```
SELECT pd.disease_id, p.fname FROM patient p FULL OUTER JOIN patient_disease
pd ON p.patient_id =pd.patient_id;
```

```
SQL> --Same scenario as above except that it is using the patient table instead of the disease table
SQL> SELECT pd.disease_id, p.fname FROM patient p FULL OUTER JOIN patient_disease pd ON p.patient_id =pd.patient_id;
```

DISEASE_ID	FNAME
11	john
22	john
11	jill
	billy

--This is the same as above. Don't need the FULL OUTER JOIN because all the records in the patient\_disease  
--table match up with the records in the patient table because it is a bridge table.

```
SELECT pd.disease_id, p.fname FROM patient p LEFT OUTER JOIN patient_disease
pd ON p.patient_id =pd.patient_id;
```

```
SQL> --table match up with the records in the patient table because it is a bridge table
SQL> SELECT pd.disease_id, p.fname FROM patient p LEFT OUTER JOIN patient_disease pd ON p.
2 patient_id =pd.patient_id;
```

DISEASE_ID	FNAME
11	john
22	john
11	jill
	billy

--Include all the data from the disease table in the FULL OUTER JOIN. Take that result set which --includes the common records and the orphan records in the disease table and include them with --all the records in the patient table, even the records in the patient table that don't match with --the patient disease table.

```
SELECT p.fname, d.disease_desc FROM disease d FULL OUTER JOIN
patient_disease pd ON d.disease_id =pd.disease_id FULL OUTER JOIN patient p
ON p.patient_id =pd.patient_id;
```

--This is the same as above. Because the patient\_diseases table is a bridge table, we can do a LEFT OUTER JOIN --to include all the stuff from the disease table. Next, we can do a full outer join to include those --results along with all the records from the patient table.

```
SELECT p.fname, d.disease_desc FROM disease d LEFT OUTER JOIN
patient_disease pd ON d.disease_id =pd.disease_id FULL OUTER JOIN patient p
ON p.patient_id =pd.patient_id;
```

```
SQL> SELECT p.fname, d.disease_desc FROM disease d FULL OUTER JOIN patient_disease pd ON d.disease_id =pd.disease_id FULL OUTER JOIN patient p ON p.patient_id =pd.patient_id;
```

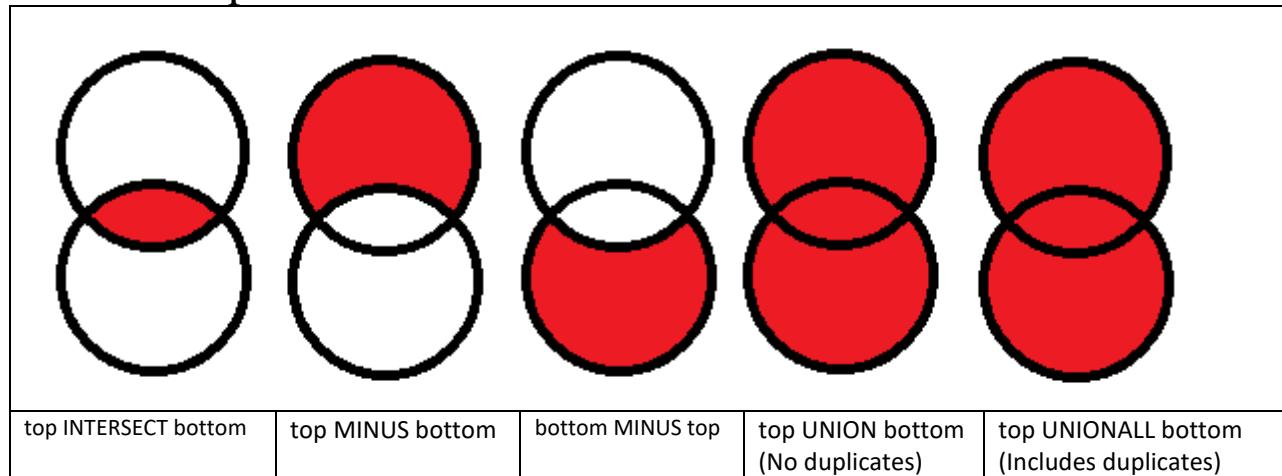
FNAME	DISEASE_DESC
john	Cancer
john	Malaria
jill	Cancer
billy	Flu

## ✓ CHECK 10B

1. Display name and description of all those who have a personality and also those who don't. For those who don't have a personality, display "Bland" for description.
  - a. Use LEFT, RIGHT and + operator syntax
2. Display the name and description of all those who have a personality, all those who don't have a personality and all personalities that are not associated with anyone. (Display fname, lname, personality description)

*"Some drink deeply from the river of knowledge. Others only gargle"*

## 10.5 Set Operators



Column headings come from the 1<sup>st</sup> SQL statement only

```

SELECT col1_text, col2_text, col3_numeric "Alias column name" FROM table_1
          (set operator)
SELECT c1_text , c2_text , c3_numeric FROM table_2
          (set operator)
SELECT co1_text, co2_text, co3_numeric FROM table_3 ORDER BY 2;
          
```

ORDER BY clause is always associated with the last SQL statement and refers to columns in the 1<sup>st</sup> sql statement

- The number of columns and the data type of columns must match for all SQL statements involved in the set operations .

Set operators are used to combine the results of two ( or more) SELECT statements. Valid set operators in Oracle are UNION, UNION ALL, INTERSECT, and MINUS. When used with two SELECT statements, the UNION set operator returns the results of both queries. However, if there are any duplicates, they are removed, and the duplicated record is listed only once. To include duplicates in the results, use the UNION ALL set operator. INTERSECT lists only records that are returned by both queries; the MINUS set operator removes the second query's results from the output if they are also found in the first query's results. INTERSECT and MINUS set operations produce unduplicated results.

UNION	Returns the results of both queries and removes duplicates
UNION ALL	Returns the results of both queries but includes duplicates
INTERSECT	Returns only the rows included in the results of both queries
MINUS	Subtracts the second query's results if they're also returned in the first query's result

Keep in mind some guidelines for multiple- column set operations: All columns are included to perform the set comparison. Each query must contain the same number of columns, which are compared positionally. Column names can be different in the queries.

```
CREATE TABLE hourly
(
    Lname          VARCHAR2(20),
    Hrly_wage      NUMBER
);

INSERT INTO hourly VALUES ('Smith',10.25);
INSERT INTO hourly VALUES ('Wesseon',30.50);
INSERT INTO hourly VALUES ('Smith',10.25);
INSERT INTO hourly VALUES ('Wesseon',30.50);

CREATE TABLE salaried
(
    Lname          VARCHAR2(20),
    salary         NUMBER
);

INSERT INTO salaried VALUES ('Smith',500);
INSERT INTO salaried VALUES ('Jones',600);
```

### Hourly

#### LnameHrly wage

Smith	10.25
Wesseon	30.50
Smith	10.25
Wesseon	30.50

### Salaried

#### LnameSalary

Smith	500
Jones	600

## **Example 10.5a (UNION)**

--Appends the result from the second query to the first query. Make sure the number of --columns in both queries are the same. Also their types must be the same. The column --headings come from the first query and ORDER BY clause appears at the end of the last query.  
--The order by clause refers to columns from the first query

```
SELECT lname FROM hourly
UNION
SELECT lname FROM salaried;
```

```
SQL> SELECT lname FROM hourly
2 UNION
3 SELECT lname FROM salaried;
```

```
LNAME
-----
Jones
Smith
Wesseon
```

--Note that both have two columns and they are both numeric. Also note the column heading

```
SELECT lname, hrly_wage * 40 pay FROM hourly
UNION
SELECT lname, salary FROM salaried;
```

SQL> --Note that both have two columns and they are both numeric. Also note the column heading

```
SQL> SELECT lname, hrly_wage * 40 pay FROM hourly
2 UNION
3 SELECT lname, salary FROM salaried;
```

LNAME	PAY
Jones	600
Smith	410
Smith	500
Weseson	1220

--Note that there are three columns for each of the queries (textual, numeric, textual). The last --column is a literal text that will be displayed for every record.

```
SELECT lname, hrly_wage * 40 pay, 'FROM HOURLY' FROM hourly
UNION
SELECT lname, salary, 'FROM SALRIED' FROM salaried ORDER BY 1;
```

SQL> --Note that there are three columns for each of the queries (textual, numeric, textual). The last --column is a literal text that will be displayed for every record.

```
SQL> SELECT lname, hrly_wage * 40 pay, 'FROM HOURLY' FROM hourly
2 UNION
3 SELECT lname, salary, 'FROM SALRIED' FROM salaried ORDER BY 1;
```

LNAME	PAY	'FROMHOURLY'
Jones	600	FROM SALRIED
Smith	410	FROM HOURLY
Smith	500	FROM SALRIED
Weseson	1220	FROM HOURLY

--Invalid: wrong number of columns

```
SELECT lname FROM hourly
UNION
SELECT lname, salary FROM salaried;
```

SQL>
SQL> --Invalid: wrong number of columns
SQL> SELECT lname FROM hourly
2 UNION
3 SELECT lname, salary FROM salaried;
SELECT lname FROM hourly
\*

ERROR at line 1:  
ORA-01789: query block has incorrect number of result columns

```
--Invalid: column types do not match
SELECT lname, hrly_wage * 40 FROM hourly
UNION
SELECT salary, lname FROM salaried;

SQL>
SQL> --Invalid: column types do not match
SQL> SELECT lname, hrly_wage * 40 FROM hourly
2 UNION
3 SELECT salary, lname FROM salaried;
SELECT lname, hrly_wage * 40 FROM hourly
*
ERROR at line 1:
ORA-01790: expression must have same datatype as corresponding expression
....
```

### Example 10.5b (UNION)

--Instead of doing DECODE or a CASE statement we can just append the results of  
--one query to another. Notice the number of columns and their types match.

```
SELECT lname, hrly_wage, 'poor' FROM hourly WHERE hrly_wage<=15
UNION
SELECT lname , hrly_wage, 'okay' FROM hourly WHERE hrly_wage>15;
```

--Notice that the number of columns and data types match using the TO\_CHAR and  
--TO\_NUMBER functions.

```
SELECT lname, '', hrly_wage poor FROM hourly WHERE hrly_wage<=15
UNION
SELECT lname , TO_CHAR(hrly_wage), TO_NUMBER('') FROM hourly WHERE
hrly wage>15;
SQL> --One query to another. Notice the number of columns and their types match
SQL> SELECT lname, hrly_wage, 'poor' FROM hourly WHERE hrly_wage<=15
2 UNION
3 SELECT lname , hrly_wage, 'okay' FROM hourly WHERE hrly_wage>15;
```

LNAME	HRLY_WAGE	'POOR'
Smith	10.25	poor
Wesseyon	30.5	okay

```
SQL>
SQL> --Notice that the number of columns and data types match using the TO_CHAR and
SQL> --TO_NUMBER functions
SQL> SELECT lname, '',
          hrly_wage poor FROM hourly WHERE hrly_wage<=
15
2 UNION
3 SELECT lname , TO_CHAR(hrly_wage), TO_NUMBER('') FROM hourly WHERE hrly_wage>15;
```

LNAME	''	POOR
Smith		10.25
Wesseyon	30.5	

***Example 10.5c (UNION ALL)***

--Whereas UNION suppresses duplicates, UNIONALL does not.

```
SELECT lname FROM hourly
```

```
UNION ALL
```

```
SELECT lname FROM salaried;
```

SQL> --Whereas UNION suppresses duplicates, UNIONALL does not

```
SQL> SELECT lname FROM hourly
```

```
2 UNION ALL
```

```
3 SELECT lname FROM salaried;
```

LNAME

-----

Smith

Wesseon

Smith

Wesseon

Smith

Jones

6 rows selected.

***Example 10.5d (INTERSECT)***

--Find the lnames that are common between the two tables.

--Duplicates are suppressed.

```
SELECT lname FROM hourly
```

```
INTERSECT
```

```
SELECT lname FROM salaried;
```

SQL> --Find the lnames that are common between the two tables

SQL> --Duplicates are suppressed

```
SQL> SELECT lname FROM hourly
```

```
2 INTERSECT
```

```
3 SELECT lname FROM salaried;
```

LNAME

-----

Smith

--Notice that the literal text hello appears in both queries, which means that it works just like the  
--above queries.

```
SELECT lname, 'hello' FROM hourly
```

```
INTERSECT
```

```
SELECT lname, 'hello' FROM salaried;
```

SQL> --Notice that the literal text hello appears in both queries

SQL> --which means that it works just like the above queries

```
SQL> SELECT lname, 'hello' FROM hourly
```

```
2 INTERSECT
```

```
3 SELECT lname, 'hello' FROM salaried;
```

LNAME

'HELL'

-----

Smith

hello

7.

--Same as above except the duplicates are not suppressed. Have to use DISTINCT to get the same results.

```
SELECT lname FROM hourly where lname IN
(SELECT lname FROM Salaried);
```

```
SQL> --same as above except the duplicates are not suppressed. Have
SQL> --to use DISTINCT to get the same results
SQL> SELECT lname FROM hourly where lname IN
2  (SELECT lname FROM Salaried);
```

LNAME

```
-----
Smith
Smith
```

### **Example 10.5e (MINUS)**

--All the records that are in hourly which are not in salaried are displayed.

--Must have the same number of columns and type.

```
SELECT lname FROM hourly
MINUS
SELECT lname FROM salaried;
```

```
SQL> SELECT lname FROM hourly
2 MINUS
3 SELECT lname FROM salaried;
```

LNAME

```
-----
Weseson
```

--All the records that are in salaried that are not in hourly are displayed.

```
SELECT lname FROM salaried
MINUS
SELECT lname FROM hourly;
```

```
SQL> --All the records that are in salaried that are not in hourly
SQL> SELECT lname FROM salaried
2 MINUS
3 SELECT lname FROM hourly;
```

LNAME

```
-----
Jones
```

--Notice that the literal text is the same in both which yields the same result as above.

```
SELECT lname, 'hello' FROM salaried
MINUS
SELECT lname, 'hello' FROM hourly;
```

```
SQL> --Notice that the literal text is the same in both which yields the same
SQL> --result as above
SQL> SELECT lname, 'hello' FROM salaried
2 MINUS
3 SELECT lname, 'hello' FROM hourly;
```

LNAME	'HELL'
Jones	hello

### **Example 10.5f (EXISTS Uncorrelated)**

The EXISTS function searches for the presence of a single row meeting the stated criteria as opposed to the IN statement which looks for all occurrences.

Rule of thumb:

- If the majority of the filtering criteria are in the subquery then the IN variation may be more efficient.
- If the majority of the filtering criteria are in the top query then the EXISTS variation may be more efficient.

EXISTS is usually more efficient than IN Because EXISTS use indexes of the table and hence scans the table faster as well as it returns the boolean value (T or F) If T is received for EXISTS clause than the rows will be returned otherwise not. Whereas IN works as simple query where it will scan all possible values in the table and then compares the condition given by you and then the result.

```
--The following displays all the patients who have diseases.
--The inner query is done first. The results are then passed on to the outer query.
--Notice that the patient_id in the outer query connects to the patient_id in the inner query.
--Note that patient_ids that are NULL in the outer query will not be considered because the IN clause only
--looks at data. NULL is void of data.

SELECT patient_id, lname FROM patient WHERE patient_id IN (SELECT patient_id FROM patient_disease);

SQL> SELECT patient_id, lname FROM patient WHERE patient_id IN (SELECT patient_id FROM patient_disease);
PATIENT_ID LNAME
-----
111 Doe
113 Crane
```

```
--All the patients, even the ones that don't have a disease are displayed.
--For every row that is being processed in the outer query, the inner query is executed. Notice
--that unlike the IN clause, there is nothing to connect the outer to the inner query. There is no
--column in the where clause. If EXISTS (SELECT * FROM patient_disease) which appears after
--the WHERE clause comes back with a TRUE result, then the row that is being processed by
--the outer query is accepted, otherwise it is rejected. When we get rows back from
--(SELECT * FROM patient_disease) then we have a true condition. If we get no rows back then it is false.

SELECT patient_id, lname FROM patient WHERE EXISTS (SELECT * FROM
patient_disease);

SQL> SELECT patient_id, lname FROM patient WHERE EXISTS (SELECT * FROM patient_disease);
PATIENT_ID LNAME
-----
111 Doe
113 Crane
114 Bob
```

--All the patients that don't have a disease are displayed.  
--The NOT IN will not come back with any results if the inner query has any nulls in it.  
--If there is a NULL, then it cannot check.  
--The NOT IN will only check against data. NULL is void of data. The IS operator will have to be used instead.

```
SELECT patient_id, lname FROM patient WHERE patient_id NOT IN (SELECT patient_id FROM patient_disease);
```

```
SQL> SELECT patient_id, lname FROM patient WHERE patient_id NOT IN (SELECT patient_id FROM patient_disease)
;
PATIENT_ID LNAME
-----
114 Bob
```

/\*The query below does not display any of the patients because there is always something in the patient\_disease table. For every row that is being processed in the outer query, the inner query is executed. Notice that unlike the IN clause, there is nothing to connect the outer to the inner query. There is no column in the where clause. If NOT EXISTS (SELECT \* FROM patient\_disease) which appears after the WHERE clause comes back with a TRUE result then the row that is being processed by the outer query is accepted, otherwise it is rejected. When we get rows back from (SELECT \* FROM patient\_disease), we have a true condition. we negate it using the NOT operator and it becomes false. Therefore the row from the outer query is rejected.

```
SELECT patient_id, lname FROM patient WHERE NOT EXISTS (SELECT * FROM patient_disease);
SQL> --ROW FROM THE OUTER QUERY IS ACCEPTED
SQL> SELECT patient_id, lname FROM patient WHERE NOT EXISTS (SELECT * FROM patient_disease);
no rows selected
```

### **Example 10.5g (EXISTS correlated)**

So far you have studied mostly uncorrelated subqueries: The subquery is executed first, its results are passed to the outer query, and then the outer query is executed. In a correlated subquery, Oracle uses a different procedure to execute a query. A correlated subquery references one or more columns in the outer query, and the EXISTS operator is used to test whether the relationship or link is present.

--All the patients who have a disease are displayed.  
--Unlike the previous example, this one connects the outer query to the inner query. For every  
--row that is processed in the outer query, the inner query also gets processed. For every row in the outer  
--query, the patient-id is compared against the patient\_id in the inner query.

```
SELECT patient_id, lname FROM patient p WHERE EXISTS (SELECT * FROM patient_disease pd WHERE p.patient_id=pd.patient_id );
SQL> --FROM THE OUTER QUERY FOR EVERY ROW IS COMPARED AGAINST THE PATIENT_ID IN THE INNER QUERY
SQL> SELECT patient_id, lname FROM patient p WHERE EXISTS (SELECT * FROM patient_disease pd WHERE p.patient_id=pd.patient_id );
PATIENT_ID LNAME
-----
111 Doe
113 Crane
```

--All the patients that don't have a disease are displayed.

--The opposite of above is done using the NOT EXISTS clause.

```
SELECT patient_id, lname FROM patient p WHERE NOT EXISTS (SELECT * FROM
patient_disease pd WHERE p.patient_id=pd.patient_id);
```

```
ONLY THE OPPOSITE OF ABOVE USING THE NOT EXISTING
SQL> SELECT patient_id, lname FROM patient p WHERE NOT EXISTS (SELECT * FROM patient_disease pd WHERE p.pa
tient_id=pd.patient_id);
```

PATIENT_ID	LNAME
114	Bob

## ✓ CHECK 10C

1. Display the fname and personality description of all with salaries greater than 10000. Union the results with all those who are making less than 5000.
2. Display the name of all those who don't have a personality. Display name and "No personality"
  - a. Use not exists
  - b. Use minus
  - c. Use not in

*"The only people with whom you should try to get even are those who have helped you"*

## Summary Examples

--Cartesian product (All combinations)

--No connection is made between the tables.

```
SELECT * FROM patient, patient_disease;
SELECT * FROM patient CROSS JOIN patient_disease;
```

--Inner join (Only commonalities)

--All tables are connected.

```
SELECT p.patient_id, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p.patient_id=pd.patient_id AND pd.disease_id=d.disease_id;
```

--No alias is needed with natural join.

```
SELECT patient_id, disease_desc FROM patient NATURAL JOIN patient_disease
NATURAL JOIN disease;
```

--Connect on the common column.

```
SELECT patient_id, disease_id FROM patient JOIN patient_disease USING
(patient_id) ;
```

--Replace the keyword WHERE with ON.

```
SELECT p.patient_id, disease_id FROM patient p JOIN patient_disease pd ON
p.patient_id=pd.patient_id;
```

--Example: For each name and description category, display the sum for the salary for only those

-- categories whose sum is greater than 10000.

```
SELECT lname, description, sum(salary) FROM patient p, patient_disease pd,
disease d WHERE p.patient_id=pd.patient_id AND pd.disease_id=d.disease_id GROUP
BY lname, description HAVING sum(salary)>10000;
```

--Outer join (Commonalities plus orphan records from one side)

--(+) means that if you cannot find a match then create an implicit NULL row which means we don't care

--if we find a match or not. (+) is associated with the table that does not have the matching record.

--(+) should be associated with the table only once and cannot appear on both sides of (=).

```
SELECT p.patient_id, disease_desc FROM patient p, patient_disease pd, disease d
WHERE p.patient_id=pd.patient_id(+) AND pd.disease_id=d.disease_id(+);
```

--Given the syntax LEFT OUTER JOIN, patient is on the left hand side. This means include all

--records, even the ones that don't match up. Take that result set and do another left outer join, which

--means include non-matching records.

```
SELECT p.patient_id, disease_desc FROM patient p LEFT OUTER JOIN patient_disease
pd ON p.patient_id=pd.patient_id LEFT OUTER JOIN disease d ON
pd.disease_id=d.disease_id;
```

--Can use RIGHT OUTER JOIN to accomplish the same thing.

```
SELECT p.patient_id, disease_desc FROM patient_disease pd RIGHT OUTER JOIN
patient p ON p.patient_id=pd.patient_id LEFT OUTER JOIN disease d ON
pd.disease_id=d.disease_id;
```

**--Full outer join (Commonalities and orphan records from both sides)**

--FULL OUTER JOIN includes the results that are non-matching from both the left and the right hand side.

```
SELECT p.patient_id, disease_desc FROM patient_disease pd RIGHT OUTER JOIN
patient p ON p.patient_id=pd.patient_id  FULL OUTER JOIN disease d ON
pd.disease_id=d.disease_id ;
```

**--Same as above**

```
SELECT p.patient_id, disease_desc FROM patient p LEFT OUTER JOIN patient_disease
pd ON p.patient_id=pd.patient_id  FULL OUTER JOIN disease d ON
pd.disease_id=d.disease_id;
```

--With unions, unionall, intersect, and minus, the number of columns and the type of columns must  
--match. Also, the column headings come from the first query. Order by must be associated with the  
--last query.

--UNION : Suppresses duplicates whereas UNIONALL does not suppress. They both appends the results of one  
--result set to another.

```
SELECT patient_id FROM patient
UNION
SELECT patient_id FROM sick
```

--INTERSECT: Finds the common patient\_ids between the two tables.

```
SELECT patient_id FROM patient
INTERSECT
SELECT patient_id FROM sick
```

--MINUS: The patient\_ids that are in patient but not in sick table are displayed.

```
SELECT patient_id FROM patient
MINUS
SELECT patient_id FROM sick
```

--Notice the parentheses. We want to do the minus first. Then the result is going to be intersected.

```
SELECT patient_id FROM patient
INTERSECT
(SELECT patient_id FROM sick
MINUS
SELECT patient_id FROM another)
```

**--IN/EXISTS**

--The inner query is done first and the results are passed back to the outer query.

--Connection is made with the patient\_id.

```
SELECT * FROM patient WHERE patient_id IN (SELECT patient_id FROM
patient_disease)
```

--For every row being processed in the outer query, the inner query is processed. Notice the  
--(\*) and also, there is no connecting column in the outer WHERE clause. If rows come back from  
--the inner query then it is true otherwise it is false. When false, the row from the outer query is rejected  
--otherwise it is accepted.

```
SELECT * FROM patient p WHERE EXISTS (SELECT * FROM patient_disease pd WHERE
p.patient_id=pd.patient_id)
```



*Ellison has followed professional tennis all his life, and took up the game in 2004. In 2010, he purchased a 50% share in one of the top four tournaments in the United States, the BNP Paribas Open.[29] This purchase saved the tournament from being sold and moved outside of the U.S.*

## Chapter 11 (Views)

*"I can resist everything except temptation."*

The diagram illustrates the relationship between three tables: Products, Orders, and Order Details. The Products table contains product details like ProductID, ProductName, SupplierID, and CategoryID. The Orders table contains order details like OrderID, CustomerID, EmployeeID, and OrderDate. The Order Details table links these two, containing OrderID, ProductID, UnitPrice, Quantity, and Discount. Arrows show the foreign key relationships: ProductID in Products points to ProductID in Order Details, OrderID in Orders points to OrderID in Order Details, and OrderID in Orders points back to ProductID in Products.

ProductID	ProductName	SupplierID	CategoryID
1	Chai	1	1
2	Chang	1	1
3	Aniseed Syrup	1	2
4	Chef Anton's Cajun Seasoning	2	2
5	Chef Anton's Gumbo Mix	2	2
6	Grandma's Boysenberry Jam	3	2
7	Uncle Bob's Organic Dried Pears	3	2
8	Northwoods Cranberry Sauce	3	2
9	Mishi Kobe Niku	4	6
10	SavorySausage	4	8
11	Queso Cabrales	5	4
12	Queso Manchego L.	5	4
13	Konbu	6	8
14	Tofu	6	7
15	Genen Shouyu	6	2
16	Pavlova	7	3
17	Alice Mutton	7	6

ProductName	Total
Alice Mutton	326.95
Aniseed Syrup	10.4
Boston Cream Pie	17910.6288923492
Chef Anton's Gumbo Mix	46825.4801330566
Carnarvon Tigers	29171.075
Chai	12780.1000595093
Chang	16355.9600446608
Chèvre bleu	12294.5399494171
Chef Anton's Cajun Seasoning	8567.899993891648
Chef Anton's Gumbo Mix	5347.200004577664
Chocolate	1368.7125241406
Côte de Blaye	14199.735229492
Escargots de Bourgogne	500.67504862013
Filo Mtl	3232.94999504089
Flötmyröst	19551.0250015259
Getost	1648.125
Genen Shouyu	1784.82499694824

OrderID	ProductID	UnitPrice	Quantity	Discount
10248	11	14	12	0
10248	42	9.8	10	0
10248	72	34.8	5	0
10249	14	18.6	9	0
10249	51	42.4	40	0
10250	41	7.7	10	0
10250	51	42.4	35	0.15
10250	65	16.8	15	0.15
10251	22	16.8	6	0.05
10251	57	15.6	15	0.05
10251	65	16.8	20	0
10252	20	64.8	40	0.05
10252	33	2	25	0.05
10252	60	27.2	40	0
10253	31	10	20	0
10253	39	14.4	42	0
10253	49	16	40	0
10254	24	3.6	15	0.15
10254	55	19.2	21	0.15

Views are database objects that store a SELECT statement and allow using a query's results as a table. Views have two purposes: Simplify issuing complex SQL queries and restrict users' access to sensitive data. Although views are database objects, they don't actually store data. A view stores a query and is used to access data in the underlying tables. Views can use all the features of a query, including specifying columns, restricting rows, and aggregating data.

**Simple view** is based on a subquery that references only one table and doesn't include group functions, expressions or GROUP BY clauses.

**Complex view** is based on a subquery that retrieves or derives data from one or more tables and can contain functions or grouped data.

**Inline view** is a subquery used in the FROM clause of a SELECT statement to create a temporary table that can be referenced by the outer query's SELECT and WHERE clauses.

```

DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR CHECK (Gender in ('m', 'f')),
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state      VARCHAR2(20)
);

```

```

INSERT INTO patient values (111,'john','Doe','m','11-FEB-
1978',25000, 'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',30000,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-
1985',60000,'Las Vegas','NV');

```

The following is an overview of the syntax elements

- **CREATE VIEW/ CREATE OR REPLACE VIEW:** You use the CREATE VIEW keywords to create a view, choosing a name that no other database object in the current schema is using. There's no way to modify or change an existing view, so if you need to change a view, you must use the CREATE OR REPLACE VIEW keywords. The OR REPLACE option notifies Oracle that a view with the same name might already exist; if it does, the view's previous version should be replaced with the one defined in the new command.
- **FORCE/ NOFORCE:** NOFORCE is the default mode for the CREATE VIEW command, which means all tables and columns must be valid, or the view isn't created. So if you attempt to create a view based on a table that doesn't exist or is currently unavailable (for example, offline), Oracle returns an error message, and the view isn't created. However, if you include the FORCE keyword in the CREATE clause, Oracle creates the view in spite of the absence of any referenced tables. This approach is commonly used when a new data-base is being developed and the data hasn't yet been loaded, or entered, into database objects.
- **View name:** You should give each view a name that isn't already assigned to another database object in the same schema.
- **Column names:** If you want to assign new names for columns the view displays, list them after the VIEW keyword inside parentheses. The number of names listed must match the number of columns returned by the SELECT statement. An alternative is using column aliases in the query. In this case, Oracle uses the aliases as column names in the view that's created.
- **AS clause:** The query listed after the AS keyword must be a complete SELECT statement (including both SELECT and FROM clauses) and can reference more than one table. The query can also include single-row and group functions, WHERE and GROUP BY clauses, nested subqueries, and so on. However, as with subqueries, the query can't include the ORDER BY clause. The query results are the content of the view that's created.
- **WITH READ ONLY option:** The WITH READ ONLY option prevents performing any DML operations on the view. This option is used often when it's important that users can only query data, not make any changes to it.

- WITH CHECK OPTION constraint: The WITH CHECK OPTION constraint ensures that any DML operations performed on the view (such as adding rows or changing data) don't prevent the view from accessing the row because it no longer meets the condition in the WHERE clause. If WITH CHECK OPTION is omitted when the view is created, any valid DML operation is allowed, even if the result is that rows being changed are no longer included in the view. However, if you're creating a view with the sole purpose of displaying data, the WITH READ ONLY option can be used instead to ensure that data can't be changed.

### **Example 11a (Simple view)**

```

SELECT patient_id, lname FROM patient;
--Creates a view called simple which runs the select statement. The view is like a
--shortcut. It does not store data but only the SQL statement.
CREATE VIEW simple AS SELECT patient_id, lname FROM patient WHERE
salary<40000;

DESC simple;

--Select from the view
SELECT * FROM simple;

SQL> SELECT patient_id, lname FROM patient;
PATIENT_ID LNAME
-----
111 Doe
113 Crane
114 Bob

SQL> --Creates a view called simple which runs the select statement. The view is like a
SQL> --shortcut. It does not store data but only the SQL statement. It will reflect all the
SQL> --underlying changes in the table
SQL> CREATE VIEW simple AS SELECT patient_id, lname FROM patient WHERE salary<40000;

View created.

SQL>
SQL> DESC simple;
Name          Null?    Type
-----          -----
PATIENT_ID      NOT NULL NUMBER
LNAME           VARCHAR2(20)

SQL> --Select from the view
SQL> SELECT * FROM simple;

PATIENT_ID LNAME
-----
111 Doe

```

### **Example 11b (System table)**

```
--USER_VIEWS table is a system table that contains all the information about the
--views that have been created.
DESC USER_VIEWS;

--The text column contains the actual SQL statement. Make sure the name of the
--view is in uppercase when checking in the system table.
SELECT view_name, text FROM user_views WHERE view_name='SIMPLE';

DROP VIEW simple;

SQL> --USER_VIEWS table is a system table that contains all the information about the
SQL> --views that have been created
SQL> DESC USER_VIEWS;
Name          Null?    Type
-----
VIEW_NAME        NOT NULL VARCHAR2(30)
TEXT_LENGTH      NUMBER
TEXT            LONG
TYPE_TEXT_LENGTH NUMBER
TYPE_TEXT       VARCHAR2(4000)
OID_TEXT_LENGTH NUMBER
OID_TEXT        VARCHAR2(4000)
VIEW_TYPE_OWNER VARCHAR2(30)
VIEW_TYPE       VARCHAR2(30)
SUPERVIEW_NAME  VARCHAR2(30)
EDITIONING_VIEW VARCHAR2(1)
READ_ONLY        VARCHAR2(1)

SQL> --The text column contains the actual SQL statement. Make sure the name of the
SQL> --view is upper case when checking in the system table
SQL> SELECT view_name, text FROM user_views WHERE view_name='SIMPLE';

VIEW_NAME
-----
TEXT
-----
SIMPLE
SELECT patient_id, lname FROM patient WHERE salary<40000

SQL> DROP VIEW simple;
View dropped
```

### **Example 11c (Inserting into a view)**

```

DROP VIEW simple;

--Create a more complicated view.
CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM
patient WHERE salary<40000;

--Can insert records into the view that would go into the underlying table. The
--data being inserted can be inside the filtering criterion of the view.
INSERT INTO simple VALUES (999,'Robin','f',30000);

--The data can also be inserted that is outside the filtering criterion.
INSERT INTO simple VALUES (101,'Julie','f',80000);

SELECT * FROM simple;
SELECT fname, lname, DOB, salary,city FROM patient;

SQL> DROP VIEW simple;
View dropped.

SQL> --Create a more complicated view
SQL> CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM patient WHERE salary<40000;
View created.

SQL> --Can insert records into the view that would go into the underlying table. The
SQL> --data being inserted can be inside the filtering criterion of the view
SQL> INSERT INTO simple VALUES (999,'Robin','f',30000);
INSERT INTO simple VALUES (999,'Robin','f',30000)
*
ERROR at line 1:
ORA-00001: unique constraint (IRAJ.SYS_C0013662) violated

SQL> --The data can also be inserted that is outside the filtering Criterion. Even though the
SQL> --data is inserted, it can only be verified in the table and not the view
SQL> INSERT INTO simple VALUES (101,'Julie','f',80000);
INSERT INTO simple VALUES (101,'Julie','f',80000)
*
ERROR at line 1:
ORA-00001: unique constraint (IRAJ.SYS_C0013662) violated

SQL> SELECT * FROM simple;
PATIENT_ID LNAME      G    SALARY
-----
  111 Doe          m   25000
  999 Robin        f   30000

SQL> SELECT fname, lname, dob, salary,city FROM patient;
FNAME      LNAME      DOB      SALARY CITY
-----
john       Doe       11-FEB-78   25000 Davis
jill       Crane     12-APR-99   30000 Reno
billy      Bob       05-MAY-85   60000 Las Vegas
                  Robin     30000
                  Julie     80000

```

### **Example 11d (More DML commands)**

```

DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR CHECK (Gender in ('m', 'f')),
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state          VARCHAR2(20)
);
INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',30000,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-
1985',60000,'Las Vegas','NV');
DROP VIEW simple;
CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM
patient WHERE salary<40000;

--Invalid: Inserts into views cannot violate constraints. This insert violates the primary key constraint.
INSERT INTO simple VALUES (111,'Robin','f',10000);

--Invalid: This insert violates the check constraint.
INSERT INTO simple VALUES (1000,'Robin','h',90000);

--We can also update views as long as we are inside the filtering criterion.
UPDATE simple SET lname='Jack' WHERE patient_id=111;
SQL> --Invalid: Inserts into views cannot violate constraints. Violates the primary key constraint
SQL> INSERT INTO simple VALUES (111,'Robin','f',10000);
INSERT INTO simple VALUES (111,'Robin','f',10000)
*
ERROR at line 1:
ORA-00001: unique constraint (IRAJ.SYS_C0013697) violated

SQL> --Invalid: Violates the check constraint
SQL> INSERT INTO simple VALUES (1000,'Robin','h',90000);
INSERT INTO simple VALUES (1000,'Robin','h',90000)
*
ERROR at line 1:
ORA-02290: check constraint (IRAJ.SYS_C0013696) violated

SQL>
SQL> --Can also update views that are inside the filtering criterion
SQL> UPDATE simple SET lname='Jack' WHERE patient_id=111;

1 row updated.

```

```
--Cannot update data that is outside the filtering criterion of the view.
UPDATE simple SET lname='Jack' WHERE patient_id=114;

-- Cannot delete data that is outside the filtering criterion of the view.
DELETE simple WHERE patient_id=114;

-- Can delete data that is inside the filtering criterion of the view.
DELETE simple WHERE patient_id=111;

--Delete everything from the view. There may still be data in the underlying table.
DELETE FROM simple;

SELECT * FROM simple;
SELECT * FROM patient;

SQL> --Cannot update data that is outside the filtering criterion of the view
SQL> UPDATE simple SET lname='Jack' WHERE patient_id=114;

0 rows updated.

SQL> -- Cannot delete data that is outside the filtering criterion of the view
SQL> DELETE simple WHERE patient_id=114;

0 rows deleted.

SQL> -- Can delete data that is inside the filtering criterion of the view
SQL> DELETE simple WHERE patient_id=111;

1 row deleted.

SQL> --Everything from the view; There may still be data in the table
SQL> DELETE FROM simple;

1 row deleted.

SQL> SELECT * FROM simple;
no rows selected

SQL> SELECT * FROM patient;

PATIENT_ID FNAME          LNAME          G DOB          SALARY CITY          STATE
-----  -----
114      billy           Bob            f 05-MAY-85    60000 Las Vegas        NV
```

### **Example 11e (Read only)**

```
DROP VIEW simple;

--The view is created with a READ ONLY option which means that nothing can be
--inserted inside the view.
CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM
patient WHERE salary<40000 WITH READ ONLY;

--Invalid: Can only select from the view. We cannot insert, update, or delete.
INSERT INTO simple VALUES (411,'Anthony','m',20000);
```

```

SQL> DROP VIEW simple;
View dropped.

SQL> --The view is created with a READ ONLY option which means that nothing can be
SQL> --inserted inside the view
SQL> CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM patient WHERE salary<40000 WITH RE
AD ONLY;

View created.

SQL> --Invalid: Can only select not insert update or delete
SQL> INSERT INTO simple VALUES (411,'Anthony','m',20000);
INSERT INTO simple VALUES (411,'Anthony','m',20000)
*
ERROR at line 1:
ORA-42399: cannot perform a DML operation on a read-only view

```

### ***Example 11f (Check option)***

```

DROP VIEW simple;
--With check option clause inserts can only be done if the record falls within the filtering
--criterion of the view.
CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM
patient WHERE salary<40000 WITH CHECK OPTION;

--Invalid: Insert falls outside the view.
INSERT INTO simple VALUES (411,'Anthony','m',90000);

--Valid: Insert falls within the view.
INSERT INTO simple VALUES (415,'Billy','m',20000);

SQL> --With the check option Inserts can only be done if the record falls within the filtering
SQL> --criterion of the view
SQL> CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM patient WHERE salary<40000 WITH CH
ECK OPTION;

View created.

SQL> --Invalid: Outside the view
SQL> INSERT INTO simple VALUES (411,'Anthony','m',90000);
INSERT INTO simple VALUES (411,'Anthony','m',90000)
*
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation

SQL> --Valid: within the view
SQL> INSERT INTO simple VALUES (415,'Billy','m',20000);

. row created.

```

### **Example 11g (Changing underlying table)**

```

DROP VIEW simple;

--Notice the view is created with the (*) option to include all the columns.
--When the underlying table is changed to include an additional column,
--the view does not reflect this change. To capture these changes, the view must be dropped and
--re-created.
CREATE VIEW simple AS SELECT * FROM patient WHERE salary<40000;
DESC simple

--A column is added but the (*) in the view does not catch it.
ALTER TABLE patient ADD status CHAR;
DESC patient;
DESC simple;

--Drop the view and re-create it.
DROP VIEW simple;
CREATE VIEW simple AS SELECT * FROM patient WHERE salary<40000;
DESC simple;

```

### **Example 11h (Dropping underlying table)**

```

DROP VIEW simple;
CREATE VIEW simple AS SELECT fname FROM patient WHERE salary<40000;
DESC simple
DROP TABLE patient;

--The view still exists. It does not get deleted. The view has to be dropped manually. Even
--if the underlying table is re-created again, this view will be invalid. It will have to be
--dropped and recreated.
DESC simple;

SQL> DROP VIEW simple;
DROP VIEW simple
*
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> CREATE VIEW simple AS SELECT fname FROM patient WHERE salary<40000;
View created.

SQL> DESC simple
      Name          Null?    Type
----- 
      FNAME           VARCHAR2(20)

SQL> DROP TABLE patient;
Table dropped.

SQL> --The view still exists. It does not get deleted. The view has to be dropped manually. Even
SQL> --if the underlying table is re-created again, this view will be invalid. It will have to be
SQL> --dropped and recreated
SQL> DESC simple;
ERROR:
ORA-24372: invalid object for describe

```

### **Example 11i (DML commands with more complex views)**

```

DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER PRIMARY KEY,
    Fname      VARCHAR2(20),
    Lname      VARCHAR2(20),
    Gender     CHAR CHECK (Gender in ('m','f')),
    DOB        DATE,
    salary     NUMBER ,
    city       VARCHAR2(20),
    state      VARCHAR2(20)
);
INSERT INTO patient values (111,'john','Doe','m','11-FEB-1978',25000,
'Davis','CA');
INSERT INTO patient values (113,'jill','Crane','m','12-APR-
1999',30000,'Reno','NV');
INSERT INTO patient values (114,'billy','Bob','f','05-MAY-
1985',60000,'Las Vegas','NV');

CREATE VIEW complex AS SELECT salary, COUNT(*) howmany FROM patient GROUP
BY salary;

--Invalid: Cannot figure out how to insert into the underlying table.
INSERT INTO complex VALUES (123,1);

SQL> CREATE VIEW complex AS SELECT salary, COUNT(*) howmany FROM patient GROUP BY salary;
View created.

SQL> --Invalid: Does not know how to insert into the underlying table
SQL> INSERT INTO complex VALUES (123,1);
INSERT INTO complex VALUES (123,1)
*
ERROR at line 1:
ORA-01733: virtual column not allowed here

```

### **Example 11j(TOP-N Analysis Inline view)**

You can use a subquery in the FROM clause of a SELECT statement to create a “temporary” table that could be referenced by the SELECT and WHERE clauses. It is considered temporary because a copy of the data the subquery returned wasn’t stored in the database. This temporary table is similar to what’s called an inline view in Oracle . The main difference between an inline view and the other views discussed is that an inline view exists only while the command is being executed. It’s not a permanent database object and can’t be referenced again by a subsequent query. This view is used most often to provide a temporary data source while a command is being executed. One of the most common uses for an inline view is performing a TOP- N analysis.

You use TOP- N analysis, in which the concepts of an inline view and the ROWNUM pseudo-column are merged to create a temporary list of records in a sorted order, and then the top N, or number, of records are retrieved. An inline view must be used for this analysis because the subquery must use an

ORDER BY clause to put records in the correct order before passing the results to the outer query—and ORDER BY clauses aren't allowed in the CREATE VIEW command.

```

DROP TABLE patient;
CREATE TABLE Patient
(
    Lname      VARCHAR2(20),
    salary     NUMBER
);

INSERT INTO patient values ('Doe',25000);
INSERT INTO patient values ('Crane',30000);
INSERT INTO patient values ('Bob',15000);
INSERT INTO patient values ('Sib',10000);
INSERT INTO patient values ('DOB',40000);

SELECT rownum, lname, salary FROM patient;

SELECT rownum, lname, salary FROM patient ORDER BY 3;
SQL> SELECT rownum, lname, salary FROM patient;
  ROWNUM LNAME          SALARY
----- -----
    1 Doe            25000
    2 Crane          30000
    3 Bob             15000
    4 Sib             10000
    5 Dob            40000

SQL> SELECT rownum, lname, salary FROM patient ORDER BY 3;
  ROWNUM LNAME          SALARY
----- -----
    4 Sib            10000
    3 Bob            15000
    1 Doe            25000
    2 Crane          30000
    5 Dob            40000

```

--We want the bottom three salaries but this doesn't do it.

```

SELECT lname, salary FROM patient WHERE rownum< 4;

DROP VIEW theView;
CREATE VIEW theView AS SELECT lname, salary FROM patient ORDER BY 2;
--Notice the rownum gets renumbered in the view.
SELECT rownum, lname, salary FROM theView;

```

SQL> --Notice the rownum gets renumbered in the view  
SQL> SELECT rownum, lname, salary FROM theView;

ROWNUM	LNAME	SALARY
1	Doe	25000
2	Crane	30000
3	Bob	60000

```
SELECT lname, salary FROM theView WHERE rownum<4;
SQL> SELECT lname, salary FROM theView WHERE rownum<4;
LNAME          SALARY
-----
Doe            25000
Crane          30000
Bob             60000
```

--The inner Query, which is referred to as an inline view, will have new rownums.

```
SELECT lname, salary FROM (SELECT lname, salary FROM patient ORDER BY 2) WHERE rownum<4;
```

SQL> --The inner Query will have new rownums which is referred to as an inline view  
SQL> SELECT lname, salary FROM (SELECT lname, salary FROM patient ORDER BY 2) WHERE rownum<4;

LNAME	SALARY
Doe	25000
Crane	30000
Bob	60000

--Top three salaries

```
SELECT lname, salary FROM (SELECT lname, salary FROM patient ORDER BY 2 DESC) WHERE rownum<4;
```

SQL> --Top three salaries  
SQL> SELECT lname, salary FROM (SELECT lname, salary FROM patient ORDER BY 2 DESC) WHERE rownum<4;

LNAME	SALARY
Bob	60000
Crane	30000
Doe	25000

## ✓ CHECK 11A

1. Create a view called Colorful that contains the list of all those who have a good personality. This view should contain the lastname, date of birth (Add two years) and personality description
2. Drop the view
3. Display the top two highest paid individuals

*Bessie Braddock: "Sir, you are drunk."*

*Churchill: "Madam, you are ugly. In the morning, I shall be sober."*

## Summary Examples

- Creates a view. The view does not contain any actual data. We can insert into the view like a normal table. The insert values can be outside the search criterion.
- Updates and deletes can only be done on records that are available to the view.

```
CREATE VIEW simple AS SELECT patient_id, lname FROM patient WHERE
salary<40000 ;
SELECT * FROM simple;
```

- We can only insert if it does not violate any of the constraints in the underlying table.

```
INSERT INTO simple VALUES(11,'John');
```

- Can only select from the the view. We cannot do any inserts,deletes, or updates.

```
CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM
patient WHERE salary<40000 WITH READ ONLY;
```

- All inserts into the view will have to fall within the seach criterion of the view.

```
CREATE VIEW simple AS SELECT patient_id, lname, gender, salary FROM
patient WHERE salary<40000 WITH CHECK OPTION;
```

- This is an inline view. Instead of using a table name after FROM, a select statement --is used. This select statement will have its rownum renumbered.

```
SELECT lname, salary FROM (SELECT lname, salary FROM patient ORDER
BY 2 DESC) WHERE rownum<4;
```



*Ellison is a licensed pilot and has owned several aircrafts*

## Chapter 12 (System Tables)

*"Marriages are made in heaven. But then again, so are thunder and lightning."*

Oracle uses a Data Dictionary to store details of all the Tables, Columns etc. You will normally be interested only in your own Tables, which are provided by the 'USER' views, which are for the User who is currently logged in. The System Administrator or DBA will usually be interested in the 'ALL' Views, which show data for all Users. System tables should not be altered directly by any user. For example, do not attempt to modify system tables with DELETE, UPDATE, or INSERT statements, or user-defined triggers.

A table containing the list of tables that make up the data dictionary. In Oracle this table is named **DICTIONARY**. A table containing the columns in each table of the data dictionary. In Oracle this table is named DICT\_COLUMNS

## Contents of the DICTIONARY table

<b>Table Name</b>	<b>Comments</b>
ALL_CATALOG	All tables, views, synonyms, sequences accessible to the user
ALL_COL_COMMENTS	Comments on columns of accessible tables and views
ALL_COL_PRIVS	Grants on columns for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
ALL_COL_PRIVS_MADE	Grants on columns for which the user is owner or grantor
ALL_COL_PRIVS_REC'D	Grants on columns for which the user, PUBLIC or enabled role is the grantee
ALL_CONSTRAINTS	Constraint definitions on accessible tables
ALL_CONS_COLUMNS	Information about accessible columns in constraint definitions
ALL_DB_LINKS	Database links accessible to the user
ALL_DEF_AUDIT_OPTS	Auditing options for newly created objects
ALL_DEPENDENCIES	Dependencies to and from objects accessible to the user
ALL_ERRORS	Current errors on stored objects that user is allowed to create
ALL_INDEXES	Descriptions of indexes on tables accessible to the user
ALL_IND_COLUMNS	COLUMNS comprising INDEXes on accessible TABLES
ALL_OBJECTS	Objects accessible to the user
ALL_REFRESH	All the refresh groups that the user can touch
ALL_REFRESH_CHILDREN	All the objects in refresh groups, where the user can touch the group
ALL_SEQUENCES	Description of SEQUENCES accessible to the user
ALL_SNAPSHOTS	Snapshots the user can look at
ALL_SOURCE	Current source on stored objects that user is allowed to create
ALL_SYNONYMS	All synonyms accessible to the user
ALL_TABLES	Description of tables accessible to the user
ALL_TAB_COLUMNS	Columns of all tables, views and clusters

dba —  
all —  
User —  
 ↗

ALL_TAB_COMMENTS	Comments on tables and views accessible to the user
ALL_TAB_PRIVS	Grants on objects for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
ALL_TAB_PRIVS_MADE	User's grants and grants on user's objects
ALL_TAB_PRIVS_REC'D	Grants on objects for which the user, PUBLIC or enabled role is the grantee
ALL_TRIGGERS	Triggers accessible to the current user
ALL_TRIGGER_COLS	Column usage in user's triggers or in triggers on user's tables
ALL_USERS	Information about all users of the database
ALL_VIEWS	Text of views accessible to the user
AUDIT_ACTIONS	Description table for audit trail action type codes.
COLUMN_PRIVILEGES	Maps action type numbers to action type names
DBA_2PC_NEIGHBORS	Grants on columns for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
DBA_2PC_PENDING	information about incoming and outgoing connections for pending transactions
DBA_AUDIT_EXISTS	info about distributed transactions awaiting recovery
DBA_AUDIT_OBJECT	Lists audit trail entries produced by AUDIT NOT EXISTS and AUDIT EXISTS
DBA_AUDIT_STATEMENT	Audit trail records for statements concerning objects, specifically: table, cluster, view, index, sequence, [public] database link, [public] synonym, procedure, trigger, rollback segment, tablespace, role, user
DBA_AUDIT_TRAIL	Audit trail records concerning grant, revoke, audit, noaudit and alter system
DBA_CATALOG	All audit trail entries
DBA_CLUSTERS	All database Tables, Views, Synonyms, Sequences
DBA_CLU_COLUMNS	Description of all clusters in the database
DBA_COL_COMMENTS	Mapping of table columns to cluster columns
DBA_COL_PRIVS	Comments on columns of all tables and views
DBA_CONSTRAINTS	All grants on columns in the database
DBA_CONS_COLUMNS	Constraint definitions on all tables
DBA_DATA_FILES	Information about accessible columns in constraint definitions
DBA_DB_LINKS	Information about database files
DBA_DEPENDENCIES	All database links in the database
DBA_ERRORS	Dependencies to and from objects
DBA_EXP_FILES	Current errors on all stored objects in the database
DBA_EXP_OBJECTS	Description of export files
DBA_EXP_VERSION	Objects that have been incrementally exported
DBA_EXTENTS	Version number of the last export session
DBA_FREE_SPACE	Extents comprising all segments in the database
	Free extents in all tablespaces

DBA_INDEXES	Description for all indexes in the database
DBA_IND_COLUMNS	COLUMNS comprising INDEXes on all TABLEs and CLUSTERs
DBA_JOBS	All jobs in the database
DBA_JOBS_RUNNING	All jobs in the database which are currently running, join v\$lock and job\$
DBA_OBJECTS	All objects in the database
DBA_OBJECT_SIZE	Sizes, in bytes, of various pl/sql objects
DBA_OBJ_AUDIT_OPTS	Auditing options for all tables and views
DBA_PRIV_AUDIT_OPTS	Describes current system privileges being audited across the system and by user
DBA_PROFILES	Display all profiles and their limits
DBA_RCHILD	All the children in any refresh group. This view is not a join.
DBA_REFRESH	All the refresh groups
DBA_REFRESH_CHILDREN	All the objects in refresh groups
DBA_RGROUP	All refresh groups. This view is not a join.
DBA_ROLES	All Roles which exist in the database
DBA_ROLE_PRIVS	Roles granted to users and roles
DBA_ROLLBACK_SEGS	Description of rollback segments
DBA_SEGMENTS	Storage allocated for all database segments
DBA_SEQUENCES	Description of all SEQUENCEs in the database
DBA_SNAPSHOTS	All snapshots in the database
DBA_SNAPSHOT_LOGS	All snapshot logs in the database
DBA_SOURCE	Source of all stored objects in the database
DBA_STMT_AUDIT_OPTS	Describes current system auditing options across the system and by user
DBA_SYNONYMS	All synonyms in the database
DBA_SYS_PRIVS	System privileges granted to users and roles
DBA_TABLES	Description of all tables in the database
DBA_TABLESPACES	Description of all tablespaces
DBA_TAB_COLUMNS	Columns of all tables, views and clusters
DBA_TAB_COMMENTS	Comments on all tables and views in the database
DBA_TAB_PRIVS	All grants on objects in the database
DBA_TRIGGERS	All triggers in the database
DBA_TRIGGER_COLS	Column usage in all triggers
DBA_TS_QUOTAS	Tablespace quotas for all users
DBA_USERS	Information about all users of the database
DBA_VIEWS	Text of all views in the database
DICTIONARY	Description of data dictionary tables and views
DICT_COLUMNS	Description of columns in data dictionary tables and views
GLOBAL_NAME	global database name
INDEX_HISTOGRAM	statistics on keys with repeat count
INDEX_STATS	statistics on the b-tree
RESOURCE_COST	Cost for each resource

ROLE_ROLE_PRIVS	Roles which are granted to roles
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
SESSION_PRIVS	Privileges which the user currently has set
SESSION_ROLES	Roles which the user currently has enabled
TABLE_PRIVILEGES	Grants on objects for which the user is the grantor, grantee, owner, or an enabled role or PUBLIC is the grantee
USER_AUDIT_OBJECT	Audit trail records for statements concerning objects, specifically: table, cluster, view, index, sequence, [public] database link, [public] synonym, procedure, trigger, rollback segment, tablespace, role, user
USER_AUDIT_STATEMENT	Audit trail records concerning grant, revoke, audit, noaudit and alter system
USER_AUDIT_TRAIL	Audit trail entries relevant to the user
USER_CATALOG	Tables, Views, Synonyms and Sequences owned by the user
USER_CLUSTERS	Descriptions of user's own clusters
USER_CLU_COLUMNS	Mapping of table columns to cluster columns
USER_COL_COMMENTS	Comments on columns of user's tables and views
USER_COL_PRIVS	Grants on columns for which the user is the owner, grantor or grantee
USER_COL_PRIVS_MADE	All grants on columns of objects owned by the user
USER_COL_PRIVS_REC'D	Grants on columns for which the user is the grantee
USER_CONSTRAINTS	Constraint definitions on user's own tables
USER_CONS_COLUMNS	Information about accessible columns in constraint definitions
USER_DB_LINKS	Database links owned by the user
USER_DEPENDENCIES	Dependencies to and from a users objects
USER_ERRORS	Current errors on stored objects owned by the user
USER_EXTENTS	Extents comprising segments owned by the user
USER_FREE_SPACE	Free extents in tablespaces accessible to the user
USER_INDEXES	Description of the user's own indexes
USER_IND_COLUMNS	COLUMNS comprising user's INDEXes or on user's TABLES
USER_JOBS	All jobs owned by this user
USER_OBJECTS	Objects owned by the user
USER_OBJECT_SIZE	Sizes, in bytes, of various pl/sql objects
USER_OBJ_AUDIT_OPTS	Auditing options for user's own tables and views
USER_REFRESH	All the refresh groups
USER_REFRESH_CHILDREN	All the objects in refresh groups, where the user owns the refresh group
USER_RESOURCE_LIMITS	Display resource limit of the user
USER_ROLE_PRIVS	Roles granted to current user
USER_SEGMENTS	Storage allocated for all database segments
USER_SEQUENCES	Description of the user's own SEQUENCEs

USER_SNAPSHOTS	Snapshots the user can look at
USER_SNAPSHOT_LOGS	All snapshot logs owned by the user
USER_SOURCE	Source of stored objects accessible to the user
USER_SYNONYMS	The user's private synonyms
USER_SYS_PRIVS	System privileges granted to current user
USER_TABLES	Description of the user's own tables
USER_TABLESPACES	Description of accessible tablespaces
USER_TAB_COLUMNS	Columns of user's tables, views and clusters
USER_TAB_COMMENTS	Comments on the tables and views owned by the user
USER_TAB_PRIVS	Grants on objects for which the user is the owner, grantor or grantee
USER_TAB_PRIVS_MADE	All grants on objects owned by the user
USER_TAB_PRIVS_REC'D	Grants on objects for which the user is the grantee
USER_TRIGGERS	Triggers owned by the user
USER_TRIGGER_COLS	Column usage in user's triggers
USER_TS_QUOTAS	Tablespace quotas for the user
USER_USERS	Information about the current user
USER_VIEWS	Text of views owned by the user

```

DROP TABLE patient_disease;
DROP TABLE patient;
CREATE TABLE Patient
(
    Patient_id NUMBER CONSTRAINT patient_patient_id_pk PRIMARY KEY,
    Fname VARCHAR2(20),
    Lname VARCHAR2(20),
    Gender CHAR CONSTRAINT patient_gender_ck CHECK
        (Gender IN ('m', 'f')),
    salary NUMBER CHECK (salary > 10000),
    UNIQUE (fname, lname)
);

CREATE TABLE patient_disease
(
    Patient_id NUMBER REFERENCES patient,
    disease_id NUMBER,
    PRIMARY KEY (patient_id, disease_id)
);

CREATE VIEW patient_view AS SELECT fname FROM patient;

```

User-views  
User-indices  
User-tables

### Example 12a(user\_tables)

--List all the tables created by the user.

```
SELECT table_name FROM user_tables;
```

```
SQL> SELECT table_name FROM user_tables;
TABLE_NAME
-----
PATIENT
PATIENT_DISEASE
```

### ***Example 12b(user\_constraints)***

Different constraint types:

- P= Primary key
- C= check constraint
- U= Unique key
- R= foreign key, references



--This table does not contain the columns associated with a constraint such as a primary-key, unique or a foreign key. To get the column information we have to go to USER\_CONS\_COLUMNS table. Connect the constraint\_name from the user\_constraints-table to the constraint\_name in the user\_cons\_columns table.

--r\_constraint\_name is used for foreign keys. For foreign keys there is the child table and then there is the parent table. The foreign key is a column in the child table whose name is identified in the constraint\_name column of the user\_constraints table.. Then this foreign key connects to a column in another table, which is a primary key. The information about the primary key would reside in the r\_constraint\_name column in the user\_constraints table.. This would mean that to get the exact information on foreign keys, we have to match up the constraint\_name and the r\_constraint\_name in the user\_constraints table against the constraint\_name in the user\_cons\_columns table.

```
SELECT table_name, constraint_name, r_constraint_name, constraint_type,
status FROM user_constraints WHERE table_name IN
('PATIENT','PATIENT_DISEASE');

SELECT table_name, constraint_name, column_name FROM user_cons_columns
WHERE table_name IN ('PATIENT','PATIENT_DISEASE');
```

SQL> -- constraint\_name is used for foreign keys  
 SQL> SELECT table\_name, constraint\_name, r\_constraint\_name, constraint\_type, status FROM user\_constraints WHERE table\_name IN ('PATIENT', 'PATIENT\_DISEASE');

TABLE_NAME	CONSTRAINT_NAME	R_CONSTRAINT_NAME	CONSTRAINT_TYP	STATUS	Search_condn
PATIENT	PATIENT_GENDER_CK		C	ENABLED	→ good in ...
PATIENT	SYS_C0013707		C	ENABLED	→ sys > low
PATIENT	PATIENT_PATIENT_ID_PK		P	ENABLED	
PATIENT	SYS_C0013709				
PATIENT_DISEASE	SYS_C0013710				
PATIENT_DISEASE	SYS_C0013711	PATIENT_PATIENT_ID_PK	U	ENABLED	

6 rows selected.

SQL> SELECT table\_name, constraint\_name, column\_name FROM user\_cons\_columns WHERE table\_name IN ('PATIENT', 'PATIENT\_DISEASE');

TABLE_NAME	CONSTRAINT_NAME	COLUMN_NAME
PATIENT	PATIENT_GENDER_CK	GENDER
PATIENT	SYS_C0013707	SALARY
PATIENT	PATIENT_PATIENT_ID_PK	PATIENT_ID
PATIENT	SYS_C0013709	FNAME
PATIENT	SYS_C0013709	LNAME
PATIENT_DISEASE	SYS_C0013710	PATIENT_ID
PATIENT_DISEASE	SYS_C0013710	DISEASE_ID
PATIENT_DISEASE	SYS_C0013711	PATIENT_ID

8 rows selected.

↓  
 column-name  
 UC.table\_name  
 Select , from  
 UC, UCC where  
 → UC.Constraint-name = UCC.constraint-name  
 and  
 constraint-type = 'P' and  
 UC.table-name = 'Patient' )

### Example 12c(user\_indexes)

--The system table, user\_indexes, contains information on the different indexes that have been created. An index created automatically with a primary key and a unique key. The index name will be the same as the constraint name for both unique and primary keys.

SELECT index\_name, table\_name FROM user\_indexes WHERE table\_name IN ('PATIENT', 'PATIENT\_DISEASE');

SQL> SELECT index\_name, table\_name FROM user\_indexes WHERE table\_name IN ('PATIENT', 'PATIENT\_DISEASE');

INDEX_NAME	TABLE_NAME
PATIENT_PATIENT_ID_PK	PATIENT
SYS_C0013709	PATIENT
SYS_C0013710	PATIENT_DISEASE

### **Example 12d(user\_ind\_columns)**

--Just like the relationship between user\_constraints and user\_cons\_columns, we have  
 --a similar relationship between user\_indexes and user\_ind\_columns. The column information  
 --for the indexes are in user\_ind\_columns table.  
 SELECT index\_name, column\_name, table\_name FROM user\_ind\_columns WHERE  
 table\_name='PATIENT';

```
SQL> SELECT index_name, column_name, table_name FROM user_ind_columns WHERE table_name='PATIENT';
```

INDEX_NAME	COLUMN_NAME	TABLE_NAME
PATIENT_PATIENT_ID_PK	PATIENT_ID	PATIENT
SYS_C0013709	FNAME	PATIENT
SYS_C0013709	LNAME	PATIENT

SQL >

### **Example 12e(user\_views)**

--View information is in the user\_views table.

```
SELECT view_name, text FROM user_views;
```

```
SQL> SELECT view_name, text FROM user_views;
```

VIEW_NAME	TEXT
COMPLEX	SELECT salary, COUNT(*) howmany FROM patient GROUP BY salary
PATIENT_VIEW	SELECT fname FROM patient
SIMPLE	SELECT fname FROM patient WHERE salary<40000
THEVIEW	SELECT lname, salary FROM patient ORDER BY 2

### **✓ CHECK 12A**

1. Identify the name of all the check constraints in the Person table.
2. Identify the name of the primary key and unique key in the Person table.
3. Identify both the constraint name and the r\_constraint names for the foreign key relationship.
4. Identify the columns that make up the primary key constraint.
5. Identify the name of all indexes for the Person table.

*"No matter how much cats fight, there always seem to be plenty of Kittens"*

## Summary Examples

--Get info on constraints: constraint\_name and r\_constraint\_name connect together for foreign key. Search  
--condition contains the condition for check constraints. Primary key and unique key information will be in the  
--constraint\_name column.

--References (R) , Primary key(P), Unique (U), Check (C)

```
SELECT table_name constraint_name, r_constraint_name, constraint_type,  
search_condition FROM user_constraints WHERE table_name='PATIENT';
```

--Get information on the columns that make up the primary key, unique or foreign key constraints.

```
SELECT table_name constraint_name FROM user_cons_columns WHERE  
table_name='PATIENT';
```

--Get information about the index names.

```
SELECT index_name, table_name FROM user_indexes;
```

--Get information on the columns that make up the indexes.

```
SELECT index_name, column_name FROM user_ind_columns;
```

--Get information on views.

```
SELECT view_name, text FROM user_views;
```





*Oracle provides guests an exclusive area to eat and relax with prime viewing of the air show demonstrations. A continental breakfast, full catered lunch, and afternoon snacks, with an array of beverage choices are usually a part of the guest experience*

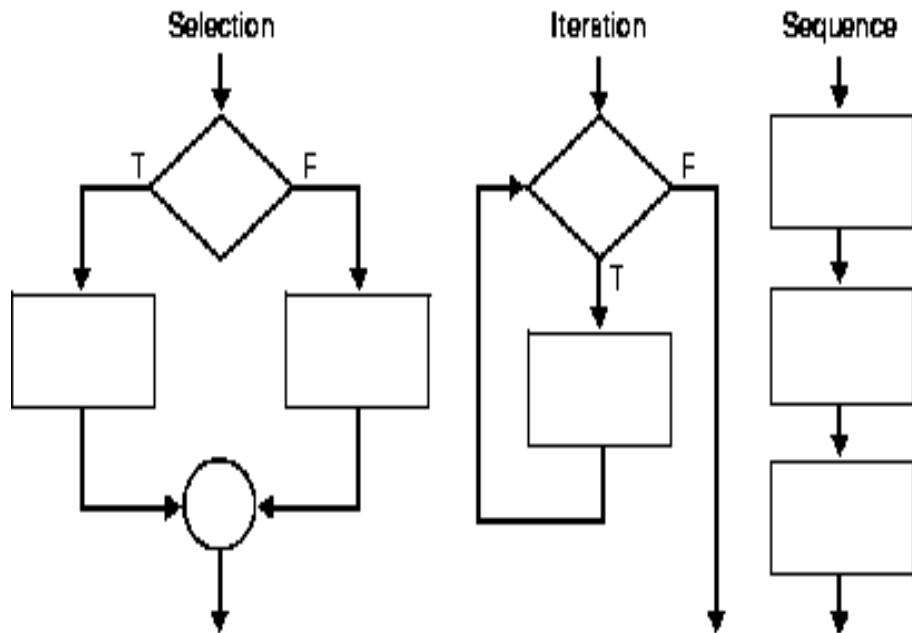
## **Chapter 13 (PL/SQL)**

*"Oh Lord, defend me from my friends; I can defend myself from my enemies"*

*"I can explain it to you but I can't understand it for you"*

## 13.1 What is a PL/SQL

PL/SQL (Procedure Language SQL) is Oracle's programming language and is comparable to FORTRAN, COBOL, C or java. PL/SQL allows the developer to define local variables, cursor, IF-THEN-ELSE constructs, and different type of loops. It also allows the user to construct procedures, functions that can be grouped into packages and libraries.

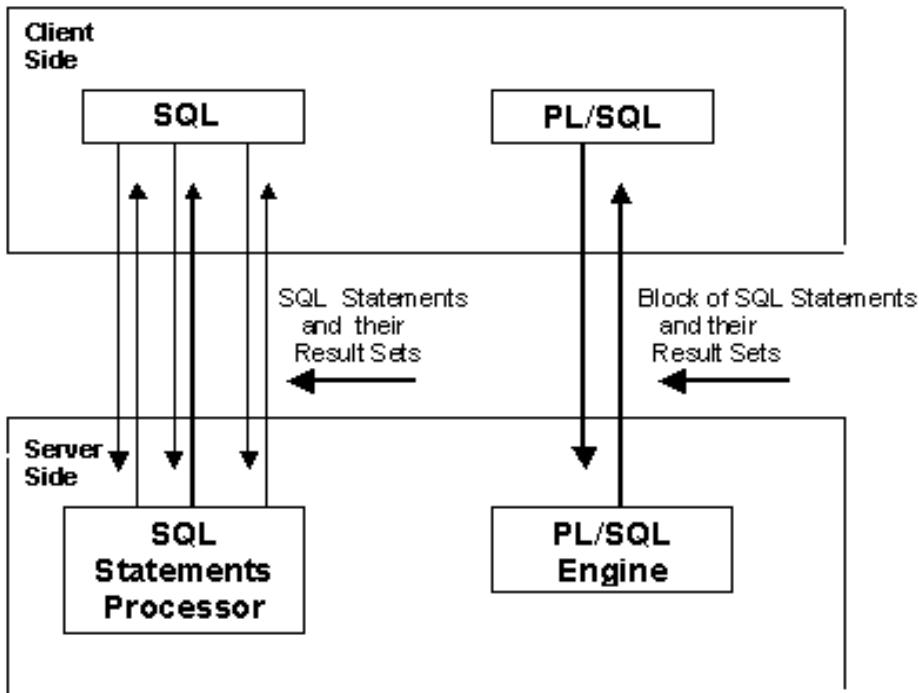


## 13.2 Advantages of PL/SQL

This illustration shows the different ways of running SQL from an application:

- Submitting individual SQL statements
- Including several SQL statements inside a PL/SQL block
- Calling a stored procedure where the SQL is already present in the database

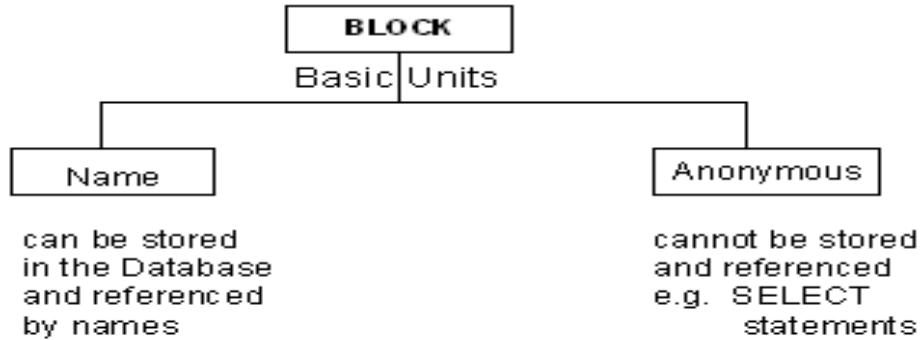
The diagram shows how each successive method involves less network traffic.



These are the advantages of PL/SQL.

- **Block Structures:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.
- **Procedural Language Capability:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).
- **Better Performance:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.
- **Error Handling:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

### 13.3 Anatomy of PL/SQL



- ◆ What is a PL/SQL Block?
  - Named block
    - ◆ Stored Procedure
    - ◆ Stored Function
    - ◆ Triggers
  - Anonymous block

### 13.4 What is a PL/SQL block made up of

Each PL/SQL program consists of SQL and PL/SQL statements which form a PL/SQL block.

A PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception (or Error) Handling section (optional).

#### Declaration Section:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

#### Execution Section:

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

#### Exception Section:

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

Every statement in the above three sections must end with a semicolon . PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

### ***Example 13.4a (Execution section)***

Simple **anonymous block**without optional **DECLARATION** or **EXCEPTION** sections:

-- To see the output, use the following command. By default, the serveroutput is off.  
SET SERVEROUTPUT ON

--Store the block in the SQLPlus buffer.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
.
```

--Run the contents of the buffer by typing Run or using the slash /.

RUN

-- Execute the block.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World! ');
END;
/
```

```

SQL> SET SERVEROUTPUT ON
SQL> --store the block in the SQLPlus buffer
SQL> BEGIN
  2   DBMS_OUTPUT.PUT_LINE('Hello World!');
  3   END;
  4 .
SQL>
SQL> run
  1 BEGIN
  2   DBMS_OUTPUT.PUT_LINE('Hello World!');
  3* END;
Hello World!
PL/SQL procedure successfully completed.

SQL> -- Execute the block
SQL> BEGIN
  2   DBMS_OUTPUT.PUT_LINE('Hello World! ');
  3   END;
  4 /
Hello World!
PL/SQL procedure successfully completed.

```

```

-- Syntax error: DBM should be DMBS.
BEGIN
  DBM_OUTPUT.PUT_LINE('Hello World! ');
END;
/
SQL> BEGIN
  2   DBM_OUTPUT.PUT_LINE('Hello World! ');
  3   END;
  4 /
  DBM_OUTPUT.PUT_LINE('Hello World! ');
*
ERROR at line 2:
ORA-06550: line 2, column 4:
PLS-00201: identifier 'DBM_OUTPUT.PUT_LINE' must be declared
ORA-06550: line 2, column 4:
PL/SQL: Statement ignored

```

## 13.5 Variable declaration

Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below.

Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

The General Syntax to declare a variable is:

**variable\_name datatype [NOT NULL := value];**

- variable\_name is the name of the variable.
- datatype is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable.

- value or DEFAULT value is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of an employee, you can use a variable.

DECLARE

```
salary number (6);
```

### Example 13.5a (variable declaration)

Here are some more examples:

```
DECLARE
```

```
    var1      NUMBER NOT NULL := 10109;
    var2      NUMBER(3,1);
    var3      NUMBER(5,2) := 31.8;
    var4      NUMBER(9,2) := var2 * 131;
    var5      CHAR(9);
    var6      VARCHAR2(12) := 'Hello';
    var7      DATE := SYSDATE;
    var8      BOOLEAN :=TRUE;
    var9      BOOLEAN :=false;
    var10     BOOLEAN;
```

```
BEGIN
```

```
    var5:='good';

    --By default all variables are assigned to NULL.
    DBMS_OUTPUT.PUT_LINE(var2 ||' ' ||var5 );
```

-- We cannot display the contents of a Boolean variable.

```
    --DBMS_OUTPUT.PUT_LINE(var10 );
```

```
End;
```

```
SQL> DECLARE
```

```
2   var1      NUMBER NOT NULL := 10109;
3   var2      NUMBER(3,1);
4   var3      NUMBER(5,2) := 31.8;
5   var4      NUMBER(9,2) := var2 * 131;
6   var5      CHAR(9);
7   var6      VARCHAR2(12) := 'Hello';
8   var7      DATE := SYSDATE;
9   var8      BOOLEAN :=TRUE;
10  var9      BOOLEAN :=false;
11  var10     BOOLEAN;
12 BEGIN
13   var5:='good';
14   --By default all variables are assigned to NULL
15   DBMS_OUTPUT.PUT_LINE(var2 ||' ' ||var5 );
16   -- cannot display the contents of a Boolean variable
17   -- DBMS_OUTPUT.PUT_LINE(var10 );
18 End;
19 /
```

```
PL/SQL procedure successfully completed.
```

### *Example 13.5b (variable declaration)*

```
--Declaring a variable
DECLARE
    greetings VARCHAR2(20);
    name VARCHAR2(20):='John';
BEGIN
    greetings := 'Hello';
    DBMS_OUTPUT.PUT_LINE(greetings);
    DBMS_OUTPUT.PUT_LINE(greetings || "|| name);
END;
/
```

```
SQL> --declaring a variable
SQL> DECLARE
2      greetings VARCHAR2(20);
3      name VARCHAR2(20):='John';
4      BEGIN
5          greetings := 'Hello';
6          dbms_output.put_line(greetings);
7          dbms_output.put_line(greetings || '|| name);
8      END;
9  /
Hello
HelloJohn
PL/SQL procedure successfully completed.
```

--Copying the name from a table to a variable

--Only a single record is expected. Multiple records would create an error.

```
DROP TABLE employee;
CREATE TABLE Employee (
    ssn  VARCHAR2(11),
    name VARCHAR2(11),
    salary NUMBER
);

INSERT INTO Employee VALUES ('777','jim',20000);
INSERT INTO Employee VALUES ('666','john',30000);
INSERT INTO Employee VALUES ('888','jack',40000);

DECLARE
    name_var  VARCHAR2(11);
BEGIN
    --We have the option of not enclosing the number in single quotes.
    SELECT name INTO name_var FROM Employee WHERE ssn = 777;
    DBMS_OUTPUT.PUT_LINE(name_var);
END;
/
```

--Only a single record is expected. Multiple records would create an error.

```

DECLARE
    name_var VARCHAR2(11);
BEGIN
    SELECT name INTO name_var FROM Employee ;
    DBMS_OUTPUT.PUT_LINE(name_var);
END;
/
SQL> CREATE TABLE Employee (
2      ssn  VARCHAR2(11),
3      name VARCHAR2(11),
4      salary NUMBER
5  );
Table created.

SQL>
SQL> INSERT INTO Employee VALUES ('777','jim',20000);
1 row created.

SQL> INSERT INTO Employee VALUES ('666','john',30000);
1 row created.

SQL> INSERT INTO Employee VALUES ('888','jack',40000);
1 row created.

SQL> DECLARE
2      name_var  VARCHAR2(11);
3  BEGIN
4      --Notice you have the option of not enclosing the number in single quotes
5      SELECT name INTO name_var FROM Employee WHERE ssn = 777;
6      DBMS_OUTPUT.PUT_LINE(name_var);
7  END;
8
jim
PL/SQL procedure successfully completed.

SQL>
SQL> --Only a single record is expected. Multiple records would create an error.
SQL> Declare
2      name_var  VARCHAR2(11);
3  BEGIN
4      SELECT name INTO name_var  FROM Employee ;
5      DBMS_OUTPUT.PUT_LINE(name_var);
6  END;
7
Declare
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4

```

### ***Example 13.5c (substitution variable)***

A substitution variable in an SQL command instructs Oracle to substitute a value in place of the variable at the time the command is actually executed. To include a substitution variable in an SQL command, simply enter an ampersand (&) followed by the name used for the variable.

```

DECLARE
    greetings  VARCHAR2(20):='&user';
BEGIN
    DBMS_OUTPUT.PUT_LINE(greetings);
END;
/

```

```

DECLARE
    greetings VARCHAR2(20);
BEGIN
    greetings:= '&user';
    DBMS_OUTPUT.PUT_LINE(greetings);
END;
/
SQL> DECLARE
  2    greetings VARCHAR2(20):='&user';
  3    BEGIN
  4      dbms_output.put_line(greetings);
  5    END;
  6  /
Enter value for user:
old 2:    greetings VARCHAR2(20):='&user';
new 2:    greetings VARCHAR2(20):='';

PL/SQL procedure successfully completed.

SQL> DECLARE
  2    greetings VARCHAR2(20);
  3    BEGIN
  4      greetings:= '&user';
  5      dbms_output.put_line(greetings);
  6    END;
  7  /
Enter value for user: jim
old 4:      greetings:= '&user';
new 4:      greetings:= 'jim';
jim

PL/SQL procedure successfully completed.

SQL>

```

## 13.6 Scope of variables

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- Local variables - These are declared in a inner block and cannot be referenced by outside Blocks.
- Global variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

For Example: In the below example we are creating two variables in the outer block and assigning their product to the third variable created in the inner block. The variable 'var\_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var\_num1' and 'var\_num2' can be accessed anywhere in the block.

### *Example 13.6a (variable declaration)*

```
--Outer, inner block
DECLARE
    var_num1 NUMBER;
    var_num2 NUMBER;
BEGIN
    var_num1 := 100;
    var_num2 := 200;
    DECLARE
        var_mult NUMBER;
    BEGIN
        var_mult := var_num1 * var_num2;
    END;
END;
/
```

```
SQL> --Outer, inner block
SQL> DECLARE
  2  var_num1 NUMBER;
  3  var_num2 NUMBER;
  4  BEGIN
  5      var_num1 := 100;
  6      var_num2 := 200;
  7      DECLARE
  8          var_mult NUMBER;
  9          BEGIN
 10              var_mult := var_num1 * var_num2;
 11          END;
 12      END;
 13  /
```

PL/SQL procedure successfully completed.

## 13.7 Constants

As the name implies a constant is a value used in a PL/SQL Block that remains unchanged throughout the program. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.

The General Syntax to declare a constant is:

**constant\_name CONSTANT datatype := VALUE;**

- **constant\_name** is the name of the constant i.e. similar to a variable name.
- The word **CONSTANT** is a reserved word and ensures that the value does not change.
- **VALUE** - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

## ✓ CHECK 13A

1. Write an anonymous block that displays the name and salary and personality description of the individual whose ssn is 111.
2. Write an anonymous block that takes a number from the user and adds it to the salary of the individual with ssn 111. Display the result of summation

*"Too many people overvalue what they are not and undervalue what they are."*

## 13.8 PL/SQL records

### What are records?

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

### Declaring a record

To declare a record, you must first define a composite datatype; then declare a record for that type. The General Syntax to define a composite datatype is:

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

- *record\_type\_name* - it is the name of the composite type you want to define.
- *first\_col\_name, second\_col\_name, etc.,* - it is the names the fields/columns within the record.
- *column\_datatype* defines the scalar datatype of the fields.

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

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

```
col_name table_name.column_name%type;
```

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

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

```
record_name table_name%ROWTYPE;
```

### *Example 13.8a (Type and ROWTYPE)*

--One field

```
DECLARE
    rec Employee.name%TYPE;
BEGIN
    SELECT name INTO rec FROM Employee WHERE ssn = 777;
    DBMS_OUTPUT.PUT_LINE(rec);
END;
/
```

--Multiple fields

```
DECLARE
    var_ssn   Employee.ssn%TYPE;
    var_name  Employee.name%TYPE;
BEGIN
    SELECT ssn,name INTO var_ssn,var_name FROM Employee WHERE ssn = 777;
    DBMS_OUTPUT.PUT_LINE(var_ssn || var_name);
END;
/
```

SQL> --one field

```
SQL> Declare
  2      rec Employee.name%TYPE;
  3  BEGIN
  4      SELECT name INTO rec FROM Employee WHERE ssn = 777;
  5      DBMS_OUTPUT.PUT_LINE(rec);
  6  END;
  7 /
```

jim

PL/SQL procedure successfully completed.

SQL>

--multiple fields

```
SQL> Declare
  2      var_ssn Employee.ssn%TYPE;
  3      var_name Employee.name%TYPE;
  4  BEGIN
  5      SELECT ssn,name INTO var_ssn, var_name FROM Employee WHERE ssn = 777;
  6      DBMS_OUTPUT.PUT_LINE(var_ssn || var_name);
  7  END;
  8 /
```

777jim

PL/SQL procedure successfully completed.

--Entire row

```
DECLARE
    rec Employee%ROWTYPE;
BEGIN
    SELECT * INTO rec FROM Employee WHERE ssn = 777;
    DBMS_OUTPUT.PUT_LINE(rec.ssn || rec.name);
END;
/
```

```

SQL> Declare
2    rec Employee%ROWTYPE;
3    BEGIN
4      SELECT * INTO rec FROM Employee WHERE ssn = 777;
5      DBMS_OUTPUT.PUT_LINE(rec.ssn || rec.name);
6    END;
7  /
777jim
PL/SQL procedure successfully completed.

```

--Declaring a type

```

Declare
  TYPE employee_type IS RECORD
  (
    ssn Employee(ssn%TYPE,
    Name Employee.name%TYPE
  );
  Emp_rec employee_type;
BEGIN
  SELECT ssn,name INTO emp_rec.ssn, emp_rec.name FROM employee WHERE ssn = 777;
  DBMS_OUTPUT.PUT_LINE(emp_rec.ssn || emp_rec.name);

```

--Can insert the results into the entire structure all at one time.

```
SELECT ssn, name INTO emp_rec FROM employee WHERE ssn = 666;
```

--ERROR: The number of columns and variables do not match.

```
--SELECT ssn, name, salary INTO emp_rec FROM employee WHERE ssn = 666;
DBMS_OUTPUT.PUT_LINE(emp_rec.ssn || emp_rec.name);
```

```
END;
/
```

```

SQL> --Declaring a type
SQL> Declare
2    TYPE employee_type IS RECORD
3    (
4      ssn Employee(ssn%TYPE,
5      Name Employee.name%TYPE
6    );
7    Emp_rec employee_type;
8  BEGIN
9    SELECT ssn,name INTO emp_rec.ssn, emp_rec.name FROM employee WHERE ssn
= 777;
10   DBMS_OUTPUT.PUT_LINE(emp_rec.ssn || emp_rec.name);
11   --can insert it into the entire structure all at one time
12   SELECT ssn, name INTO emp_rec FROM employee WHERE ssn = 666;
13   --Must make sure the number of values match: ERROR
14   --SELECT ssn, name,salary INTO emp_rec FROM employee WHERE ssn = 666;
15   DBMS_OUTPUT.PUT_LINE(emp_rec.ssn || emp_rec.name);
16 END;
17 /
777jim
666john

```

```
PL/SQL procedure successfully completed.
```

## 13.9 Conditional Statements

As the name implies, PL/SQL supports programming language features like conditional statements, iterative statements.

The programming constructs are similar to how you use in programming languages like Java and C++. In this section I will provide you syntax of how to use conditional statements in PL/SQL programming.

### **IF THEN ELSE STATEMENT**

```
1)
IF condition THEN
    statement 1;
ELSE
    statement 2;
END IF;

2)
IF condition 1 THEN
    statement 1;
    statement 2;
ELSIF condition2 THEN
    statement 3;
ELSE
    statement 4;
END IF;
```

#### **Example 13.9a (If statement)**

--Can use any of the functions such as lower or upper.

```
DECLARE
    var_name Employee.name%TYPE;
BEGIN
    SELECT name INTO var_name FROM Employee WHERE ssn ='777';
    IF (LOWER(var_name) = 'john') THEN
        DBMS_OUTPUT.PUT_LINE('hello');
    END IF;
END;
/
```

--Can use the in clause.

```
DECLARE
    var_name Employee.name%TYPE;
BEGIN
    SELECT name INTO var_name FROM Employee WHERE ssn ='777';
    IF (var_name IN ('jim','john','jack')) THEN
        DBMS_OUTPUT.PUT_LINE('hello');
    END IF;
END;
/
```

```

SQL> --can use any of the functions such as lower or upper
SQL> DECLARE
2   var_name Employee.name%TYPE;
3   BEGIN
4     SELECT name INTO var_name FROM Employee WHERE ssn ='777';
5     IF (LOWER(var_name) = 'john') THEN
6       DBMS_OUTPUT.PUT_LINE('hello');
7     END IF;
8   END;
9 /
PL/SQL procedure successfully completed.

SQL> --Can use the in clause
SQL> DECLARE
2   var_name Employee.name%TYPE;
3   BEGIN
4     SELECT name INTO var_name FROM Employee WHERE ssn ='777';
5     IF (var_name IN ('jim', 'john', 'jack')) THEN
6       DBMS_OUTPUT.PUT_LINE('hello');
7     END IF;
8   END;
9 /
hello
PL/SQL procedure successfully completed.

```

--Can use the like clause.

```

DECLARE
  var_name Employee.name%TYPE;
BEGIN
  SELECT name INTO var_name FROM Employee WHERE ssn ='777';
  IF (var_name LIKE 'j%') THEN
    DBMS_OUTPUT.PUT_LINE('hello');
  END IF;
END;
/

```

--Use other operators

```

DECLARE
  X NUMBER;
BEGIN
  X := 3;
  IF X > 2 THEN
    DBMS_OUTPUT.PUT_LINE('X IS GREATER THAN 2');
  ELSE
    DBMS_OUTPUT.PUT_LINE('X IS NOT GREATER THAN 2');
  END IF;
END;
/

```

```

SQL> --Can use the like clause
SQL> DECLARE
2      var_name Employee.name%TYPE;
3  BEGIN
4      SELECT name INTO var_name FROM Employee WHERE ssn ='777';
5      IF (var_name LIKE 'j%') THEN
6          DBMS_OUTPUT.PUT_LINE('hello');
7      END IF;
8  END;
9 /
hello
PL/SQL procedure successfully completed.

SQL>
SQL> --Use other operators
SQL> DECLARE
2      X NUMBER;
3  BEGIN
4      X := 3;
5      IF X > 2 THEN
6          DBMS_OUTPUT.PUT_LINE('X IS GREATER THAN 2');
7      ELSE
8          DBMS_OUTPUT.PUT_LINE('X IS NOT GREATER THAN 2');
9      END IF;
10 END;
11 /
X IS GREATER THAN 2
PL/SQL procedure successfully completed.

```

CASE selector

```

WHEN expression1 THEN sequence_of_statements1;
WHEN expression2 THEN sequence_of_statements2; ...
WHEN expressionN THEN sequence_of_statementsN;
ELSE sequence_of_statements N+1;
END CASE ;

```

### *Example 13.9b (case)*

```

DROP TABLE students;
CREATE TABLE students
(
    Name VARCHAR2(10),
    Grade CHAR
);

INSERT INTO students values ('jack','b');
INSERT INTO students values ('john','c');

```

--Using case

```

DECLARE
    grade students.grade%TYPE;
BEGIN
    SELECT grade INTO grade FROM students WHERE name='jack';
    CASE UPPER(grade)
        WHEN 'A' THEN
            DBMS_OUTPUT.PUT_LINE('Excellent');
        WHEN 'B' THEN
            DBMS_OUTPUT.PUT_LINE('Very Good');
        WHEN 'C' THEN
            DBMS_OUTPUT.PUT_LINE('Good');
        WHEN 'D' THEN
            DBMS_OUTPUT.PUT_LINE('Fair');
        WHEN 'F' THEN
            DBMS_OUTPUT.PUT_LINE('Poor');
        ELSE
            DBMS_OUTPUT.PUT_LINE('No such grade');
    END CASE;
END;
/

```

```

SQL> --Using case
SQL> DECLARE
2      grade    students.grade%TYPE;
3  BEGIN
4      SELECT grade INTO grade FROM students WHERE name='jack';
5      CASE UPPER(grade)
6          WHEN 'A' THEN
7              dbms_output.put_line('Excellent');
8          WHEN 'B' THEN
9              dbms_output.put_line('Very Good');
10         WHEN 'C' THEN
11             dbms_output.put_line('Good');
12         WHEN 'D' THEN
13             dbms_output.put_line('Fair');
14         WHEN 'F' THEN
15             dbms_output.put_line('Poor');
16         ELSE
17             dbms_output.put_line('No such grade');
18     END CASE;
19 END;
20 /
Very Good

```

PL/SQL procedure successfully completed.

--IF/ELSIF NOTICE it is elsif and not elseif.

```

DECLARE
    var_name Employee.name%TYPE;
    var_ssn Employee.ssn%TYPE;
BEGIN
    SELECT name,ssn INTO var_name, var_ssn FROM Employee WHERE ssn='777';
    DBMS_OUTPUT.PUT_LINE(var_name);
    IF (var_name = 'iraj') THEN
        DBMS_OUTPUT.PUT_LINE('hello');
    ELSIF (var_name='jack') THEN
        DBMS_OUTPUT.PUT_LINE('hi');
    ELSE
        DBMS_OUTPUT.PUT_LINE('goodbye');
    END IF;
END;
/

```

```

SQL> --IF/ELSIF      NOTICE it is elsif and not elseif
SQL> DECLARE
2      var_name Employee.name%TYPE;
3      var_ssn Employee.ssn%TYPE;
4 BEGIN
5      SELECT name,ssn INTO var_name, var_ssn FROM Employee WHERE ssn='777';
6      DBMS_OUTPUT.PUT_LINE(var_name);
7      IF (var_name = 'iraj') THEN
8          DBMS_OUTPUT.PUT_LINE('hello');
9      ELSIF (var_name='jack') THEN
10         DBMS_OUTPUT.PUT_LINE('hi');
11     ELSE
12         DBMS_OUTPUT.PUT_LINE('goodbye');
13     END IF;
14 END;
15 /
jim
goodbye
PL/SQL procedure successfully completed.

```

## ✓ CHECK 13B

1. Write an anonymous block that displays the name, salary and personality description of the individual whose ssn is 111. Use the type syntax. If the salary is less than the average salary, then display poor.

*"Some cause happiness wherever they go; others whenever they go"*

## 13.10 Loops

An iterative control Statements are used when we want to repeat the execution of one or more statements for specified number of times. These are similar to those in

There are three types of loops in PL/SQL:

- Simple Loop
- While Loop
- For Loop

### 1) Simple Loop

A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates. An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations. When the EXIT condition is satisfied the process exits from the loop.

The General Syntax to write a Simple Loop is:

```
LOOP
  statements;
  EXIT;
  {or EXIT WHEN condition;}
END LOOP;
```

### 2) While Loop

A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true. The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

The General Syntax to write a WHILE LOOP is:

```
WHILE <condition>
  LOOP statements;
END LOOP;
```

### 3) FOR Loop (Can only increment or decrement by 1. Loop variable cannot be changed inside the loop)

A FOR LOOP is used to execute a set of statements for a predetermined number of times. Iteration occurs between the start and end integer values given. The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer.

The General Syntax to write a FOR LOOP is:

```
FOR counter IN val1..val2
  LOOP statements;
END LOOP;
```

- val1 - Start integer value.
- val2 - End integer value.

**Example 13.10a (Simple loop)**

```
--Error: Infinite loop
/* DECLARE
   a NUMBER:=5;
BEGIN
   LOOP
      a:=a+1;
      DBMS_OUTPUT.PUT_LINE(a);
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('DONE...');

END;
/
*/
```

--Finite loop using if exit

```
DECLARE
   a NUMBER:=5;
BEGIN
   LOOP
      a:=a-1;
      DBMS_OUTPUT.PUT_LINE(a);
      IF a=1 THEN
         EXIT;
      END IF;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('DONE...');

END;
/
```

--Finite loop using exit when

```
DECLARE
   a NUMBER:=1;
BEGIN
   LOOP
      a:=a+1;
      DBMS_OUTPUT.PUT_LINE(a);
      EXIT WHEN
         a=5;
   END LOOP;
   DBMS_OUTPUT.PUT_LINE('DONE...');

END;
/
```

--Can use if exit or exit when

```

DECLARE
    a NUMBER:=0;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE(a );
        a:=a+1;
        --If a > 5 THEN
        --    EXIT;
        -- END IF;
        EXIT WHEN a>5;

    END LOOP;
END;
/

```

```

SQL> DECLARE
2      a NUMBER:=1;
3      BEGIN
4          LOOP
5              a:=a+1;
6              DBMS_OUTPUT.PUT_LINE(a);
7              EXIT WHEN
8              a=5;
9          END LOOP;
10     DBMS_OUTPUT.PUT_LINE('DONE.');
11     END;
12 /
2
3
4
5
DONE.

```

PL/SQL procedure successfully completed.

### Example 13.10b (While loop)

--While loop

```

DECLARE
    b NUMBER :=3;
BEGIN
    WHILE b < 5
    LOOP
        DBMS_OUTPUT.PUT_LINE(b);
        b:=b+1;
    END LOOP;
END;
/

```

```
--Using a Boolean flag to end the loop
DECLARE
    flag BOOLEAN:=TRUE;
    a NUMBER:=3;
BEGIN
    WHILE flag
    LOOP
        IF a < 1 THEN
            flag :=false;
        END IF;
        a:=a-1;
        DBMS_OUTPUT.PUT_LINE( a );
    END LOOP;
END;
/

```

```
SQL> DECLARE
  2      b NUMBER :=3;
  3      BEGIN
  4          WHILE b < 5
  5          LOOP
  6              DBMS_OUTPUT.PUT_LINE(b);
  7              b:=b+1;
  8          END LOOP;
  9      END;
 10  /
3
4

PL/SQL procedure successfully completed.
```

```
SQL>
SQL> --using a Boolean flag to end the loop
SQL> DECLARE
  2      flag BOOLEAN:=TRUE;
  3      a NUMBER:=3;
  4      BEGIN
  5          WHILE flag
  6          LOOP
  7              IF a < 1 THEN
  8                  flag :=false;
  9              END IF;
 10             a:=a-1;
 11             DBMS_OUTPUT.PUT_LINE( a );
 12         END LOOP;
 13     END;
 14  /
2
1
0
-1
```

--Premature exit using IF

```
DECLARE
    flag BOOLEAN:=TRUE;
    a NUMBER:=3;
BEGIN
    WHILE flag
    LOOP
        IF a < 1 THEN
            flag :=false;
        ELSIF a=2 THEN
            EXIT;
        END IF;
        a:=a-1;
        DBMS_OUTPUT.PUT_LINE( a );
    END LOOP;
END;
/
```

--Premature exit using WHEN

```
DECLARE
    flag BOOLEAN:=TRUE;
    a NUMBER:=3;
BEGIN
    WHILE flag
    LOOP
        IF a < 1 THEN
            flag :=false;
        END IF;
        EXIT WHEN
            a=2;
        a:=a-1;
        DBMS_OUTPUT.PUT_LINE( a );
    END LOOP;
END;
/
```

```

SQL> --Premature exit using IF
SQL> DECLARE
2      flag BOOLEAN:=TRUE;
3      a NUMBER:=3;
4      BEGIN
5          WHILE flag
6              LOOP
7                  IF a < 1 THEN
8                      flag :=false;
9                  ELSIF a=2 THEN
10                      EXIT;
11                  END IF;
12                  a:=a-1;
13                  DBMS_OUTPUT.PUT_LINE( a );
14              END LOOP;
15          END;
16      /
2

```

PL/SQL procedure successfully completed.

```

SQL> --Premature exit using WHEN
SQL> DECLARE
2      flag BOOLEAN:=TRUE;
3      a NUMBER:=3;
4      BEGIN
5          WHILE flag
6              LOOP
7                  IF a < 1 THEN
8                      flag :=false;
9                  END IF;
10                 EXIT WHEN
11                         a=2;
12                         a:=a-1;
13                         DBMS_OUTPUT.PUT_LINE( a );
14                     END LOOP;
15                 END;
16             /
2

```

PL/SQL procedure successfully completed.

### *Example 13.10c (For loop)*

--For loop. Don't have to declare the loop variable. The loop variable is not available  
--outside the loop.

```

BEGIN
    FOR a IN 1..5
        LOOP
            DBMS_OUTPUT.PUT_LINE( a );
        END LOOP;
    END;
/

```

--ERROR: cannot modify the loop variable.

```

BEGIN
    FOR a IN 1..5
        LOOP
            a:=a+1;
            DBMS_OUTPUT.PUT_LINE( a );
        END LOOP;
    END;
/

```

```

SQL> --outside the loop
SQL> BEGIN
  2   FOR a IN 1..5
  3     LOOP
  4       DBMS_OUTPUT.PUT_LINE( a );
  5     END LOOP;
  6   END;
  7 /
1
2
3
4
5

PL/SQL procedure successfully completed.

SQL> --Cannot modify the loop variable. Will get an error message
SQL> BEGIN
  2   FOR a IN 1..5
  3     LOOP
  4       a:=a+1;
  5       DBMS_OUTPUT.PUT_LINE( a );
  6     END LOOP;
  7   END;
  8 /
        a:=a+1;
      *
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00363: expression 'A' cannot be used as an assignment target
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored

```

--Variable "a" is available outside the loop.

```

DECLARE
  a NUMBER:=2;
BEGIN
  FOR a IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE( a );
  END LOOP;
  --The content that will be displayed is the integer 2.
  DBMS_OUTPUT.PUT_LINE(a);
END;

```

--Variable "a" is available outside the loop

```

SQL> DECLARE
  2   a NUMBER:=2;
  3 BEGIN
  4   FOR a IN 1..5
  5     LOOP
  6       DBMS_OUTPUT.PUT_LINE( a );
  7     END LOOP;
  8   --the content that will be displayed is the integer 2
  9   DBMS_OUTPUT.PUT_LINE( a );
 10  END;
 11 /
1
2
3
4
5
2

```

```
--Premature exit using if exit
BEGIN
    FOR a IN REVERSE 1..5
    LOOP
        IF (a=3) THEN
            EXIT;
        END IF;
        DBMS_OUTPUT.PUT_LINE( a );
    END LOOP;
END;
/
```

--Premature exit using exit when

```
BEGIN
    FOR a IN REVERSE 1..5
    LOOP
        EXIT WHEN
            a=3;
        DBMS_OUTPUT.PUT_LINE( a );
    END LOOP;
END;
/
```

```
SQL> --premature exit using if exit
SQL> BEGIN
2   FOR a IN REVERSE 1..5
3     LOOP
4       IF (a=3) THEN
5         EXIT;
6       END IF;
7       DBMS_OUTPUT.PUT_LINE( a );
8     END LOOP;
9   END;
10 /
5
4
```

PL/SQL procedure successfully completed.

```
SQL> --premature exit using exit when
SQL> BEGIN
2   FOR a IN REVERSE 1..5
3     LOOP
4       EXIT WHEN
5           a=3;
6     DBMS_OUTPUT.PUT_LINE( a );
7   END LOOP;
8   END;
9
10 /
5
4
```

PL/SQL procedure successfully completed.

### *Example 13.10d (Decrementing a loop)*

```

BEGIN
    FOR a IN REVERSE 1..5
    LOOP
        FOR b IN 1..5
        LOOP
            DBMS_OUTPUT.PUT_LINE( a || ' ' || b );
        END LOOP;
    END LOOP;
END;
/
SQL> BEGIN
2      FOR a IN REVERSE 1..5
3      LOOP
4          FOR b IN 1..5
5          LOOP
6              DBMS_OUTPUT.PUT_LINE( a || ' ' || b );
7          END LOOP;
8      END LOOP;
9
10
51
52
53
54
55
41
42
43
44
45
31
32
33
34
35
21
22
23
24
25
11
12
13
14
15

```

### *Example 13.10e (Labels)*

```

--Can give loops labels
BEGIN
    <<OUTER>>
    FOR a IN REVERSE 1..5
    LOOP
        <<INNER>>
        FOR a IN 1..5
        LOOP
            EXIT outer WHEN a=2;
            DBMS_OUTPUT.PUT_LINE(a );
        END LOOP INNER;
    END LOOP OUTER;
END;
/

```

```

SQL> --can give loops labels
SQL> BEGIN
2   <<OUTER>>
3     FOR a IN REVERSE 1..5
4       LOOP
5         <<INNER>>
6           FOR a IN 1..5
7             LOOP
8               EXIT inner WHEN a=2;
9               DBMS_OUTPUT.PUT_LINE(a);
10              END LOOP INNER;
11            END LOOP OUTER;
12          END;
13      /
1
1
1
1
1
PL/SQL procedure successfully completed.

```

### **Example 13.10f (GOTO- Do not use)**

--Can go up

```

BEGIN
  ...
  <<update_row>>
  BEGIN
    UPDATE emp SET ...
    ...
  END;
  ...
  GOTO update_row;
  ...
END;

```

-- Must precede an executable statement.

```

DECLARE
  done BOOLEAN;
BEGIN
  ...
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
  ...
  <<end_loop>> -- not allowed
  END LOOP; -- not an executable statement
END;

```

-- Fix the problem with the executable statement

```

FOR i IN 1..50 LOOP
  IF done THEN
    GOTO end_loop;
  END IF;
  ...
  <<end_loop>>
  NULL; -- an executable statement
END LOOP;

```

```
-- Can't branch into IF statement
BEGIN
  ...
  GOTO update_row; -- can't branch into IF statement
  ...
  IF valid THEN
    ...
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;
```

### ✓ CHECK 13C

1. Write an anonymous block that asks the user for two inputs. Display all the odd numbers in the range. Assume the first number is smaller than the second number. Skip the number if it is 13.
  - a. Simple, while and for loops

*"Misery is almost always the result of thinking"*

## 13.11 Cursors

A cursor is a temporary work area created in the system memory when an SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The cursor must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row. A cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor.

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

*CURSOR cursor\_name IS select\_statement;*

- *cursor\_name – A suitable name for the cursor.*
- *select\_statement – A select query which returns multiple rows.*

There are four steps in using an Cursor.

- DECLARE the cursor in the declaration section.

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

1) Declaring a Cursor in the Declaration Section:

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

2) Accessing the records in the cursor:

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

These are the three steps in accessing the cursor.

- 1) Open the cursor.
- 2) Fetch the records in the cursor one at a time.
- 3) Close the cursor.

General Syntax to open a cursor is:

```
OPEN cursor_name;
```

General Syntax to fetch records from a cursor is:

```
FETCH cursor_name INTO record_name;
```

OR

```
FETCH cursor_name INTO variable_list;
```

General Syntax to close a cursor is:

```
CLOSE cursor_name;
```

When does an error occur while accessing an cursor?

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

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

Attributes	Return values	Example
%FOUND	TRUE, if fetch statement returns at least one row.	Cursor_name%FOUND
	FALSE, if fetch statement doesn't return a row.	
%NOTFOUND	TRUE, , if fetch statement doesn't return a row.	Cursor_name%NOTFOUND
	FALSE, if fetch statement returns at least one row.	
%ROWCOUNT	The number of rows fetched by the fetch statement	Cursor_name%ROWCOUNT
	If no row is returned, the PL/SQL statement returns an error.	
%ISOPEN	TRUE, if the cursor is already open in the program	Cursor_name%ISNAME
	FALSE, if the cursor is not opened in the program.	

### **Example 13.11a (Cursor , reading a record)**

```

DROP TABLE student;
CREATE TABLE student (name VARCHAR2(20), state VARCHAR2(20), salary
NUMBER);
INSERT INTO student VALUES ('jim','CA',10000);
INSERT INTO student VALUES ('john','NY',20000);
INSERT INTO student VALUES ('johnny','NY',20000);
INSERT INTO student VALUES ('johnoo','NY',20000);

DECLARE
    vr_stuff student%ROWTYPE;
    CURSOR my_cursor IS
        SELECT * FROM student WHERE state ='NY';
BEGIN
    OPEN my_cursor;
    LOOP
        FETCH my_cursor INTO vr_stuff;
        EXIT WHEN
            my_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(vr_stuff.name);
    END LOOP;
    CLOSE my_cursor;
END;
/

```

```

SQL> DECLARE
2   vr_stuff student%ROWTYPE;
3   CURSOR my_cursor IS
4       SELECT * FROM student WHERE state ='NY';
5   BEGIN
6       OPEN my_cursor;
7       LOOP
8           FETCH my_cursor INTO vr_stuff;
9           EXIT WHEN
10              my_cursor%NOTFOUND;
11          DBMS_OUTPUT.PUT_LINE(vr_stuff.name);
12      END LOOP;
13      CLOSE my_cursor;
14  END;
15 /
john

```

PL/SQL procedure successfully completed.

### **Example 13.11b (Modifying the contents of the cursor)**

--Making changes to the table while in the cursor does not affect the cursor's result set.  
--In this case, it still looks at the old value Joseph. Once the cursor is created, changes to the  
--underlying table do not affect it.

```

DROP TABLE student;

CREATE TABLE student (name VARCHAR2(20), state VARCHAR2(20), salary
NUMBER);

INSERT INTO student VALUES ('jim','CA',10000);
INSERT INTO student VALUES ('john','NY',20000);
INSERT INTO student VALUES ('jill','NY',10000);

```

```

INSERT INTO student VALUES ('joseph','NY',50000);

DECLARE
    vr_stuff  student%ROWTYPE;
    CURSOR my_cursor IS
        SELECT * FROM student WHERE state ='NY';
BEGIN
    OPEN my_cursor;
    LOOP
        FETCH my_cursor INTO vr_stuff;
        EXIT WHEN
            my_cursor%NOTFOUND;
        UPDATE student SET name='jared' WHERE salary=50000;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE(vr_stuff.name);
    END LOOP;
    CLOSE my_cursor;
END;
/
SQL> DECLARE
  2    vr_stuff  student%ROWTYPE;
  3    CURSOR my_cursor IS
  4        SELECT * FROM student WHERE state ='NY';
  5 BEGIN
  6    OPEN my_cursor;
  7    LOOP
  8        FETCH my_cursor INTO vr_stuff;
  9        EXIT WHEN
 10            my_cursor%NOTFOUND;
 11        UPDATE student SET name='jared' WHERE salary=50000;
 12        COMMIT;
 13        DBMS_OUTPUT.PUT_LINE(vr_stuff.name);
 14    END LOOP;
 15    CLOSE my_cursor;
 16 END;
 17 /
john
jill
joseph

```

--Can also declare a variable of type cursor. Make sure that the cursor is defined first otherwise  
-- it will give you an error.

```

DECLARE
    CURSOR my_cursor IS
        SELECT *FROM student WHERE state ='NY';
    vr_stuff  my_cursor%ROWTYPE;

BEGIN
    OPEN my_cursor;
    LOOP
        FETCH my_cursor INTO vr_stuff;
        EXIT WHEN
            my_cursor%NOTFOUND;
        UPDATE student SET name='iraj' WHERE salary=50000;
        COMMIT;
        DBMS_OUTPUT.PUT_LINE(vr_stuff.name);
    END LOOP;

```

```

CLOSE my_cursor;
END;

SQL> DECLARE
2      CURSOR my_cursor IS
3          SELECT *FROM student WHERE state ='NY';
4          vr_stuff  my_cursor%ROWTYPE;
5
6      BEGIN
7          OPEN my_cursor;
8          LOOP
9              FETCH my_cursor INTO vr_stuff;
10             EXIT WHEN
11                 my_cursor%NOTFOUND;
12             UPDATE student SET name='iraj' WHERE salary=50000;
13             COMMIT;
14             DBMS_OUTPUT.PUT_LINE(vr_stuff.name);
15         END LOOP;
16         CLOSE my_cursor;
17     END;
18 /  

john
jill
jared

```

### *Example 13.11c (Reading into a variable)*

```

DECLARE
    v_name student.name%TYPE;
    CURSOR c_student IS
        SELECT name FROM student;
BEGIN
    OPEN c_student;
    LOOP
        FETCH c_student INTO v_name;
        EXIT WHEN
            c_student%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(v_name);
    END LOOP;
    CLOSE c_student;
EXCEPTION
    WHEN others THEN
        IF  c_student%ISOPEN THEN
            CLOSE c_student;
        END IF;
END;

```

```

SQL> DECLARE
2      v_name student.name%TYPE;
3          CURSOR c_student IS
4              SELECT name FROM student;
5      BEGIN
6          OPEN c_student;
7          LOOP
8              FETCH c_student INTO v_name;
9              EXIT WHEN
10                 c_student%NOTFOUND;
11             DBMS_OUTPUT.PUT_LINE(v_name);
12         END LOOP;
13         CLOSE c_student;
14     EXCEPTION
15         WHEN others THEN
16             IF c_student%ISOPEN THEN
17                 CLOSE c_student;
18             END IF;
19     END;
20 /
jim
john
jill
iraj
PL/SQL procedure successfully completed.

```

### ***Example 13.11d (Using a for loop)***

```

DECLARE
    v_state student%ROWTYPE;
    CURSOR c_state IS
        SELECT * FROM student;
BEGIN
    FOR v_state IN c_state
    LOOP
        DBMS_OUTPUT.PUT_LINE (v_state.state);
    END LOOP;
END;
SQL> DECLARE
2      v_state student%ROWTYPE;
3          CURSOR c_state IS
4              SELECT * FROM student;
5      BEGIN
6          FOR v_state IN c_state
7          LOOP
8              DBMS_OUTPUT.PUT_LINE (v_state.state);
9          END LOOP;
10     END;
11 /
CA
NY
NY
NY

```

### ***Example 13.11e (Skipping the declare section)***

```

BEGIN
    FOR r_student IN (SELECT * FROM student)
    LOOP
        DBMS_OUTPUT.PUT_LINE(r_student.state );
    END LOOP;
END;

```

```

SQL> BEGIN
2   FOR r_student IN (SELECT * FROM student)
3     LOOP
4       DBMS_OUTPUT.PUT_LINE(r_student.state );
5     END LOOP;
6   END;
7
CA
NY
NY
NY

```

### ✓ CHECK 13D

1. Write a cursor that displays the lname, salary and personality description for each individual.

When displaying the salary, add 50 to the salaries of all individuals who are ugly.

- a. Simple, while and for loop

```

DECLARE
  CURSOR per_cursor IS
    SELECT * FROM person, personality WHERE person.id = personality.id;
  ver_cursor per_cursor%ROWTYPE;
BEGIN
  OPEN per_cursor;
  LOOP
    FETCH per_cursor INTO ver_cursor;
    EXIT WHEN per_cursor%NOTFOUND;
    IF TO_LOWER(ver_cursor.description) = 'ugly' THEN
      DBMS_OUTPUT.PUT_LINE(ver_cursor.lname || ver_cursor.salary + 50 ||
ver_cursor.description);
    ELSE
      DBMS_OUTPUT.PUT_LINE(ver_cursor.lname || ver_cursor.salary ||
ver_cursor.description);
    END IF;
  END LOOP;
  CLOSE per_cursor;
END;
/

```

```

DECLARE
    CURSOR per_cursor IS
        SELECT * FROM person, personality WHERE person.id = personality.id;
    ver_cursor per_cursor%ROWTYPE;
BEGIN
    FOR ver_cursor IN per_cursor
        IF TO_LOWER(ver_cursor.description) = 'ugly' THEN
            DBMS_OUTPUT.PUT_LINE(ver_cursor.lname || ver_cursor.salary + 50 || ver_cursor.description);
        ELSE
            DBMS_OUTPUT.PUT_LINE(ver_cursor.lname || ver_cursor.salary || ver_cursor.description);
        END IF;
    END LOOP;
END;
/

```

*"Worry gives a small thing a big shadow"*

## 13.12 Stored procedures

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

We can pass parameters to procedures in three ways.

- 1) IN-parameters
- 2) OUT-parameters
- 3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

```

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

```

*Exception section*  
 END;

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

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

### ***Example 13.12a (creating a procedure)***

```
DROP TABLE employee;
DROP PROCEDURE hello_world;
CREATE TABLE Employee (
    ssn      VARCHAR2(11),
    name     VARCHAR2(11),
    salary   NUMBER
);
INSERT INTO Employee VALUES ('777','jim',20000);
INSERT INTO Employee VALUES ('666','john',30000);
INSERT INTO Employee VALUES ('888','jack',40000);

CREATE PROCEDURE hello_world
IS
    greetings VARCHAR2(20);
BEGIN
    greetings := 'Hello World!';
    DBMS_OUTPUT.PUT_LINE(greetings);
END hello_world;
/

--System tables
SELECT object_name FROM user_objects WHERE object_name='HELLO_WORLD';
```

```

SQL> CREATE PROCEDURE hello_world
  2  IS
  3      greetings VARCHAR2(20);
  4  BEGIN
  5      greetings := 'Hello World!';
  6      dbms_output.put_line(greetings);
  7  END hello_world;
  8 /


Procedure created.

SQL>
SQL> --System tables
SQL> SELECT object_name FROM user_objects WHERE object_name='HELLO_WORLD';

OBJECT_NAME
-----
HELLO_WORLD

select * FROM user_source;

--Dropping a procedure
DROP PROCEDURE hello_world;
SELECT object_name FROM user_objects WHERE object_name='HELLO_WORLD';

SQL> select * FROM user_source;
NAME          TYPE          LINE
TEXT
-----
HELLO_WORLD      PROCEDURE      1
PROCEDURE hello_world
HELLO_WORLD      PROCEDURE      2
IS
HELLO_WORLD      PROCEDURE      3
    greetings VARCHAR2(20);

NAME          TYPE          LINE
TEXT
-----
HELLO_WORLD      PROCEDURE      4
BEGIN
HELLO_WORLD      PROCEDURE      5
    greetings := 'Hello World!';
HELLO_WORLD      PROCEDURE      6
    dbms_output.put_line(greetings);

NAME          TYPE          LINE
TEXT
-----
HELLO_WORLD      PROCEDURE      7
END hello_world;

7 rows selected.

```

```

SQL> --Dropping a procedure
SQL> DROP PROCEDURE hello_world;
Procedure dropped.

SQL> SELECT object_name FROM user_objects WHERE object_name='HELLO_WORLD';
no rows selected

```

--Error messages: Entry is created in the system table even though there is an error.

```

CREATE PROCEDURE hello_world
IS
    greetings VARCHAR2(20);
BEGIN
    greeings := 'Hello World!';
    DBMS_OUTPUT.PUT_LINE(greetings);
END hello_world;
/

```

--Display the error messages

```
SHOW ERRORS
```

```

SQL> CREATE PROCEDURE hello_world
  2  IS
  3      greetings VARCHAR2(20);
  4  BEGIN
  5      greeings := 'Hello World!';
  6      dbms_output.put_line(greetings);
  7  END hello_world;
  8 /

```

Warning: Procedure created with compilation errors.

```
SQL>
```

--Display the error messages

```
SQL> Show errors
```

Errors for PROCEDURE HELLO\_WORLD:

LINE/COL ERROR

```
-----  
5/5     PL/SQL: Statement ignored  
5/5     PLS-00201: identifier 'GREEINGS' must be declared
```

-- Creates a procedure 'employer\_details' which gives the details of the employee table.

```

CREATE OR REPLACE PROCEDURE employer_details
IS
    CURSOR emp_cur IS SELECT ssn, name FROM employee;
    emp_rec emp_cur%ROWTYPE;
BEGIN
    FOR emp_rec IN emp_cur
    LOOP
        DBMS_OUTPUT.PUT_LINE(emp_rec.ssn || ' ' || emp_rec.name);
    END LOOP;
END;
/

```

```

SQL> -- Creates a procedure 'employer_details' which gives the det
ployee table.
SQL> CREATE OR REPLACE PROCEDURE employer_details
  2 IS
  3   CURSOR emp_cur IS SELECT ssn, name FROM employee;
  4   emp_rec emp_cur%rowtype;
  5 BEGIN
  6   FOR emp_rec IN emp_cur
  7   LOOP
  8     dbms_output.put_line(emp_rec.ssn || ' ' || emp_rec.name);
  9   END LOOP;
 10 END;
 11 /
Procedure created.

```

### How to execute a Stored Procedure?

There are two ways to execute a procedure:

- 1) From the SQL prompt.

EXECUTE [or EXEC] procedure\_name;

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

procedure\_name;

### *Example 13.12b (calling a procedure)*

```

BEGIN
  hello_world;
  employer_details;
END;
/
--Execute at the SQL prompt
EXECUTE hello_world;

SQL> BEGIN
  2   hello_world;
  3   employer_details;
  4 END;
 5 /
Hello World!
777 jim
666 john
888 jack
PL/SQL procedure successfully completed.

SQL> --Execute at the SQL prompt
SQL> Execute hello_world;
Hello World!
PL/SQL procedure successfully completed.

```

### **Example 13.12c (Procedure inside an anonymous block)**

```
DROP PROCEDURE hello_world;

-- Declare a procedure inside an anonymous block
DECLARE
    PROCEDURE hello_world IS
        greetings VARCHAR2(20);
    BEGIN
        greetings := 'Hello World!';
    DBMS_OUTPUT.PUT_LINE(greetings);
    END hello_world;
BEGIN
    hello_world;
END;
/

-- Procedure is created temporarily. Nothing is inserted into the system tables.
SELECT * FROM user_source WHERE name='HELLO_WORLD';
SQL> -- Declare a procedure inside an anonymous block
SQL> DECLARE
  2      PROCEDURE hello_world IS
  3          greetings VARCHAR2(20);
  4      BEGIN
  5          greetings := 'Hello World!';
  6          dbms_output.put_line(greetings);
  7          END hello_world;
  8      BEGIN
  9          hello_world;
 10      END;
 11 /
Hello World!
PL/SQL procedure successfully completed.

SQL>
SQL> -- Procedure is created temporarily and so it is not inserted into the
tem tables
SQL> SELECT * FROM user_source WHERE name='HELLO_WORLD';
no rows selected
```

## **13.13 Functions**

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

The General Syntax to create a function is:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS
    Declaration_section
```

```

BEGIN
  Execution_section
  Return return_variable;
EXCEPTION
  exception_section
  Return return_variable;
END;

```

**1) Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.

2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a function called "employer\_details\_func" similar to the one created in stored proc

```

1> CREATE OR REPLACE FUNCTION employer_details_func
2>   RETURN VARCHAR(20);
3> IS
4>   emp_name VARCHAR(20);
5> BEGIN
6>   SELECT name INTO emp_name
7>   FROM employee WHERE ssn = '777';
8>   RETURN emp_name;
9>
10> END;
11> /

```

In the example we are retrieving the 'first\_name' of employee with ssn 777 to variable 'emp\_name'.

The return type of the function is VARCHAR which is declared in line no 2.

The function returns the 'emp\_name' which is of type VARCHAR as the return value in line no 9.

#### How to execute a PL/SQL Function?

A function can be executed in the following ways.

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

```
employee_name := employer_details_func;
```

If 'employee\_name' is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

```
SELECT employer_details_func FROM dual;
```

3) In a PL/SQL Statements like,

```
DBMS_OUTPUT.PUT_LINE(employer_details_func);
```

This line displays the value returned by the function

### **Example 13.13a (Creating and executing functions)**

#### --Create a function

```
CREATE OR REPLACE FUNCTION say_hi_func
    RETURN VARCHAR2 IS
BEGIN
    RETURN 'hi';
END;
/
```

#### --Inserts entries into system tables

```
SELECT * FROM user_source WHERE name='SAY_HI_FUNC';
```

```
SQL> --Create a function
SQL> CREATE OR REPLACE FUNCTION say_hi_func
2    RETURN VARCHAR2 IS
3    BEGIN
4        RETURN 'hi';
5    END;
6    /
```

Function created.

```
SQL>
```

```
SQL> --Inserts entries into system tables
SQL> SELECT * FROM user_source WHERE name='SAY_HI_FUNC';
```

NAME	TYPE	LINE
TEXT		
SAY_HI_FUNC	FUNCTION	1
FUNCTION say_hi_func		
SAY_HI_FUNC	FUNCTION	2
RETURN VARCHAR2 IS		
SAY_HI_FUNC	FUNCTION	3
BEGIN		
NAME	TYPE	LINE
TEXT		
SAY_HI_FUNC	FUNCTION	4
RETURN 'hi';		
SAY_HI_FUNC	FUNCTION	5
END;		

#### --Create another function

```
CREATE OR REPLACE FUNCTION employer_details_func
    RETURN VARCHAR2 IS
    emp_name VARCHAR(20);
BEGIN
    SELECT name INTO emp_name
    FROM employee WHERE ssn = '777';
    RETURN emp_name;
END;
/
```

```
--Executing a function
DECLARE
    greetings  VARCHAR2(10);
BEGIN
    SELECT say_hi_func INTO greetings FROM dual;
    DBMS_OUTPUT.PUT_LINE(greetings);
    DBMS_OUTPUT.PUT_LINE(say_hi_func);
    greetings:=say_hi_func;
    DBMS_OUTPUT.PUT_LINE(greetings);
END;
/
SQL> --Create another function
SQL> CREATE OR REPLACE FUNCTION employer_details_func
2      RETURN VARCHAR2 IS
3      emp_name VARCHAR(20);
4      BEGIN
5          SELECT name INTO emp_name
6          FROM employee WHERE ssn = '777';
7          RETURN emp_name;
8      END;
9
Function created.

SQL>
SQL> --Executing a function
SQL> DECLARE
2      greetings  VARCHAR2(10);
3      BEGIN
4          SELECT say_hi_func INTO greetings FROM dual;
5          Dbms_output.put_line(greetings);
6          Dbms_output.put_line(say_hi_func);
7          greetings:=say_hi_func;
8          Dbms_output.put_line(greetings);
9      END;
10
hi
hi
hi
hi

PL/SQL procedure successfully completed.
```

```
--Another way of executing a function. Cannot do this with procedures
SELECT say_hi_func FROM dual;

--Dropping a function
DROP FUNCTION say_hi_func;

--Gets removed from the system table
SELECT * FROM user_source WHERE name='SAY_HI_FUNC';
```

```
SQL> SELECT say_hi_func FROM dual;
SAY_HI_FUNC
-----
hi

SQL> --Dropping a function
SQL> DROP FUNCTION say_hi_func;
Function dropped.

SQL> --Gets removed from the system table
SQL> SELECT * FROM user_source WHERE name='SAY_HI_FUNC';
no rows selected
```

### **Example 13.13b (Function inside an anonymous block)**

```

DROP FUNCTION say_hi_func;

DECLARE
    greetings  VARCHAR2(10);
--All declarations should be before the function definition.
FUNCTION say_hi_func RETURN VARCHAR2 AS
BEGIN
    RETURN 'hi';
END;
-- Cannot declare here
--Somevariable VARCHAR2(10);
BEGIN
    -- Cannot use this because it is not inserted into any system table
    --SELECT say_hi_func INTO greetings FROM dual;
    DBMS_OUTPUT.PUT_LINE(say_hi_func);
    greetings:=say_hi_func;
    DBMS_OUTPUT.PUT_LINE(greetings);
END;
/
--Noting inserted into the system table
SELECT * FROM user_source WHERE name='SAY_HI_FUNC';

```

```

SQL> DECLARE
  2  greetings  VARCHAR2(10);
  3      --All declarations should be before the function definition
  4  FUNCTION say_hi_func RETURN VARCHAR2 AS
  5  BEGIN
  6      RETURN 'hi';
  7  END;
  8      -- Cannot declare here
  9      --Somevariable VARCHAR2(10);
10  BEGIN
11      -- Cannot use this because it is not inserted into any system table
12      --SELECT say_hi_func INTO greetings FROM dual;
13      Dbms_output.put_line(say_hi_func);
14      greetings:=say_hi_func;
15      Dbms_output.put_line(greetings);
16  END;
17 /
hi
hi

PL/SQL procedure successfully completed.

SQL>
SQL> --Noting inserted into the system table
SQL> SELECT * FROM user_source WHERE name='SAY_HI_FUNC';

no rows selected

```

### **Example 13.13c (Bind variables)**

Bind variables are used in SQL and PL/SQL statements for holding data or result sets. They are commonly used in SQL statements to optimize statement performance. A statement with a bind variable may be re-executed multiple times without needing to be re-parsed. Their values can be set and referenced in PL/SQL blocks. They can be referenced in SQL statements e.g. SELECT. Except in the VARIABLE and PRINT commands, bind variable references should be prefixed with a colon.

Bind variables are created with the VARIABLE command. There is no way to undefine or delete a bind variable in a SQL\*Plus session. However, bind variables are not remembered when you exit SQL\*Plus.

```
--Using a bind variable in PL/SQL block
begin
  :bv := 8;
  DBMS_OUTPUT.PUT_LINE ('this is the bind variable' || :bv);
end;
/

CREATE OR REPLACE FUNCTION say_hi_func
  RETURN VARCHAR2 IS
BEGIN
  RETURN 'hi';
END;
/


SQL> --Using a bind variable in PL/SQL block
SQL> begin
2   :bv := 8;
3   dbms_output.put_line ('this is the bind variable' || :bv);
4 end;
5 /
this is the bind variable8
PL/SQL procedure successfully completed.
```

```
--Another way of declaring variables
VAR num_var  NUMBER
VAR char_var  VARCHAR2(40)

--Examining the datatype of all the variables
VAR

--Examining the datatype for the one variable
VAR num_var

--Capturing the return value of a function
EXECUTE :char_var :=say_hi_func()

--Display the contents of the bind variable
SELECT :char_var FROM dual;

--Displaying the contents of the bind variable
PRINT char_var
```

```

Function created.

SQL>
SQL> --Another way of declaring variables
SQL> var num_var number
SQL> var char_var varchar2(40)
SQL>
SQL> --Examining the datatype of all the variables
SQL> var
variable by
datatype NUMBER

variable num_var
datatype NUMBER

variable char_var
datatype VARCHAR2(40)
SQL>
SQL> --Examining the datatype for the one variable
SQL> var num_var
variable num_var
datatype NUMBER
SQL>
SQL> --Capturing the return value of a function
SQL> execute :char_var :=say_hi_func()

PL/SQL procedure successfully completed.

SQL>
SQL> --Display the contents of the bind variable
SQL> Select :char_var from dual;

:CHAR_VAR
-----
hi

SQL>
SQL> --Displaying the contents of the bind variable
SQL> print char_var

CHAR_VAR
-----
hi

```

```

DROP TABLE employee;

CREATE TABLE employee
(
    name      VARCHAR(10),
    ssn       VARCHAR(11),
    salary    NUMBER
);

INSERT INTO employee VALUES ('john','555',10000);
INSERT into employee VALUES ('jack','666',20000);
INSERT INTO employee VALUES ('jill','777',30000);

VAR salary NUMBER

```

```
--Using a bind variable in the select statement
Declare
    var_ssn Employee.ssn%TYPE;
    var_name Employee.name%TYPE;
BEGIN
    SELECT ssn,name, salary INTO var_ssn, var_name,:salary FROM
        Employee WHERE ssn = 777;
    DBMS_OUTPUT.PUT_LINE(var_ssn || var_name||:salary);
END;
/
```

```
SQL> VAR salary NUMBER
SQL>
SQL> --Using a bind variable in the select statement
SQL> Declare
2    var_ssn Employee.ssn%TYPE;
3    var_name Employee.name%TYPE;
4    BEGIN
5        SELECT ssn,name, salary INTO var_ssn, var_name,:salary  FROM Employee
WHERE ssn = 777;
6        DBMS_OUTPUT.PUT_LINE(var_ssn || var_name||:salary);
7    END;
8 /
777jim20000
PL/SQL procedure successfully completed.
```

## 13.14 Parameters in functions and procedures

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

### 1) IN parameter:

This is similar to passing parameters in programming languages. We can pass values to the stored procedure through these parameters or variables. This type of parameter is a read only parameter.

The General syntax to pass a IN parameter is

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

### 2) OUT Parameter:

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

The General syntax to create an OUT parameter is

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

The parameter should be explicitly declared as OUT parameter.

### **3) IN OUT Parameter:**

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

The General syntax to create an IN OUT parameter is

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

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

**Example1:**

Using IN and OUT parameter:

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

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

We can call the procedure 'emp\_name' in this way from a PL/SQL Block.

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

**Example 2:**

Using IN OUT parameter in procedures:

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

```

12>   ELSIF tmp_sal between 20000 and 30000 THEN
13>     salary_inout := tmp_sal * 1.3;
14>   ELSIF tmp_sal > 30000 THEN
15>     salary_inout := tmp_sal * 1.4;
16>   END IF;
17> END;
18> /

```

The below PL/SQL block shows how to execute the above 'emp\_salary\_increase' procedure.

```

1> DECLARE
2>   CURSOR updated_sal IS
3>     SELECT empID,salary
4>       FROM emp_tbl;
5>   pre_sal NUMBER;
6> BEGIN
7>   FOR emp_rec IN updated_sal LOOP
8>     pre_sal := emp_rec.salary;
9>     emp_salary_increase(emp_rec.empID, emp_rec.salary);
10>    DBMS_OUTPUT.PUT_LINE('The salary of ' || emp_rec.empID ||
11>                          ' increased from '|| pre_sal || ' to '|| emp_rec.salary);
12>   END LOOP;
13> END;
14> /

```

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

### ***Example 13.14a (Function with two parameter)***

--Function accepts two string parameters and returns a string.

```

CREATE OR REPLACE FUNCTION
  first_function (first VARCHAR2, last VARCHAR2) RETURN VARCHAR2 AS
BEGIN
  RETURN 'Good morning ' || first || last ;
END;
/

```

--Execute the function

```

SELECT first_function('john', 'smith') FROM dual;

```

--Execute the function

```

DECLARE
  fname VARCHAR2(10):='john';
  lname VARCHAR2(10):='smith';
  greetings VARCHAR2(30);
BEGIN
  SELECT first_function(fname, lname) INTO greetings FROM dual;
  DBMS_OUTPUT.PUT_LINE(greetings);
END;
/

```

```

SQL> CREATE OR REPLACE FUNCTION
2      first_function (first VARCHAR2, last VARCHAR2) RETURN VARCHAR2 AS
3  BEGIN
4    RETURN 'Good morning ' || first || last ;
5  END;
6 /


Function created.

SQL>
SQL> --Execute the function,
SQL> SELECT first_function('john', 'smith') FROM dual;
FIRST_FUNCTION('JOHN','SMITH')
-----
Good morning johnsmith

SQL>
SQL> --Execute the function
SQL> DECLARE
2    fname VARCHAR2(10):='john';
3    lname VARCHAR2(10):='smith';
4    greetings VARCHAR2(30);
5  BEGIN
6    SELECT first_function(fname, lname) INTO greetings  FROM dual;
7    DBMS_OUTPUT.PUT_LINE(greetings);
8  END;
9 /
Good morning johnsmith

PL/SQL procedure successfully completed.

```

### **Example 13.14b (procedure with two parameter)**

--Procedure accepts two string parameters and displays a string.

```

CREATE OR REPLACE PROCEDURE
    first_procedure (first VARCHAR2, last VARCHAR2) AS
BEGIN
    DBMS_OUTPUT.PUT_LINE( 'Good morning ' || first || last );
END;
/


--Execute the Procedure
EXECUTE first_procedure('john', 'smith')

--Execute the Procedure
DECLARE
    fname VARCHAR2(10):='john';
    lname VARCHAR2(10):='smith';
    greetings VARCHAR2(30);
BEGIN
    first_procedure(fname, lname);
END;
/

```

```

SQL> --Procedure accepts two string parameters and displays a string
SQL> CREATE OR REPLACE PROCEDURE
2      first_procedure (first VARCHAR2, last VARCHAR2) AS
3      BEGIN
4          DBMS_output.put_line( 'Good morning ' || first || last );
5      END;
6  /
Procedure created.

SQL>
SQL> --Execute the Procedure
SQL> EXECUTE first_procedure('john', 'smith')
Good morning johnsmith
PL/SQL procedure successfully completed.

SQL>
SQL> --Execute the Procedure
SQL> DECLARE
2      fname VARCHAR2(10):='john';
3      lname VARCHAR2(10):='smith';
4      greetings VARCHAR2(30);
5      BEGIN
6          first_procedure(fname, lname);
7      END;
8  /
Good morning johnsmith
PL/SQL procedure successfully completed.

```

### ***Example 13.14c (Function and Procedure in an anonymous block)***

```

--Declaring a function and procedure in an anonymous block and executing them.

DECLARE
    --Variables must be declared before the functions and procedures.
    first VARCHAR2(10):='John';
    last VARCHAR2(10):='smith';

    FUNCTION first_function (first VARCHAR2, last VARCHAR2) RETURN
    VARCHAR2 AS
    BEGIN
        RETURN 'Good morning ' || first || last ;
    END;

    PROCEDURE first_procedure (first VARCHAR2, last VARCHAR2) AS
    BEGIN
        DBMS_OUTPUT.PUT_LINE( first_function(first,last));
    END;
BEGIN
    first_procedure(first,last);
END;
/

```

```

SQL> --Declaring a function and procedure in an anonymous block and executing th
em
SQL> DECLARE
2      --variables must be declared before the functions and procedures
3      first VARCHAR2(10):='John';
4      last VARCHAR2(10):='smith';
5
6      FUNCTION first_function (first VARCHAR2, last VARCHAR2) RETURN VARCHAR2
AS
7      BEGIN
8          RETURN 'Good morning ' || first || last ;
9      END;
10
11     PROCEDURE first_procedure (first VARCHAR2, last VARCHAR2) AS
12     BEGIN
13         DBMS_output.put_line( first_function(first,last));
14     END;
15     BEGIN
16         first_procedure(first,last);
17     END;
18 /
Good morning Johnsmith
PL/SQL procedure successfully completed.

```

### ✓ CHECK 13E

1. Create a function that returns the average salary.
2. Create a procedure that takes in the ssn and displays the name and salary if the individual is making less than the average salary (call the above function).
3. Execute the procedure from both an anonymous block and also SQL\*Plus.

*"People may doubt what you say, but they will believe what you do"*

## 13.15 Exceptions

PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a messages which explains its cause is received. By Handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

PL/SQL Exception message consists of three parts.

- 1) Type of Exception
- 2) An Error Code
- 3) A message

## Structure of Exception Handling

```

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

```

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

When an exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'ex\_name1 ', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing fo the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

## Types of Exception

There are 3 types of Exceptions.

- a) Named System Exceptions
- b) Unnamed System Exceptions
- c) User-defined Exceptions

## Named System Exceptions

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

**For example:** NO\_DATA\_FOUND and ZERO\_DIVIDE are called Named System exceptions.

Named system exceptions are:

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

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

**For Example:** Suppose a NO\_DATA\_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

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

### Example 13.15a (Exception Value error)

```
--Cannot use goto from an exception to jump to execution
DECLARE
    X NUMBER;
BEGIN
    X := 'YYYY';
    DBMS_OUTPUT.PUT_LINE('IT WORKS');
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('VALUE_ERROR EXCEPTION HANDLER');
END;
/

DECLARE
    X NUMBER;
BEGIN
    X := 2001;
    DBMS_OUTPUT.PUT_LINE('IT WORKS');
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('VALUE_ERROR EXCEPTION
        HANDLER');
END;
/
```

```

SQL> DECLARE
2      X NUMBER;
3  BEGIN
4      X := 'YYYY';
5      DBMS_OUTPUT.PUT_LINE('IT WORKS');
6  EXCEPTION
7      WHEN VALUE_ERROR THEN
8          DBMS_OUTPUT.PUT_LINE('VALUE_ERROR EXCEPTION HANDLER');
9  END;
10 /
VALUE_ERROR EXCEPTION HANDLER

PL/SQL procedure successfully completed.

SQL> DECLARE
2      X NUMBER;
3  BEGIN
4      X := 2001;
5      DBMS_OUTPUT.PUT_LINE('IT WORKS');
6  EXCEPTION
7      WHEN VALUE_ERROR THEN
8          DBMS_OUTPUT.PUT_LINE('VALUE_ERROR EXCEPTION
9          HANDLER');
10 END;
11 /
IT WORKS

PL/SQL procedure successfully completed.

```

### ***Example 13.15b (Unnamed system Exception)***

Those system exception for which oracle does not provide a name is known as unnamed system exception. These exception do not occur frequently. You can use the WHEN OTHERS exception handler

--Try putting in a character and then experiment again with a number. When dealing with --substitution variables, they are dealt with at compile time. This means that it will ask for input --first, do the substitution and then go through the execution. If this was a procedure, it would --behave like a double ampersand, in that while the procedure is being created, it asks for input and --would use that same input everytime the procedure runs.

```

DECLARE
    v_number number;
BEGIN
    DBMS_OUTPUT.PUT_LINE('hello');
    v_number:=&ui;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(SQRT(v_number)));
EXCEPTION
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('put in a number');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Something else');
END;
/

```

```

SQL> DECLARE
  2    v_number number;
  3  BEGIN
  4    DBMS_OUTPUT.PUT_LINE('hello');
  5    v_number:=&ui;
  6    DBMS_OUTPUT.PUT_LINE(SQRT(v_number));
  7  EXCEPTION
  8    WHEN VALUE_ERROR THEN
  9      DBMS_OUTPUT.PUT_LINE('put in a number');
 10   WHEN OTHERS THEN
 11     DBMS_OUTPUT.PUT_LINE('Something else');
 12 END;
 13 /
Enter value for ui: 25
old   5:      v_number:=&ui;
new   5:      v_number:=25;
hello
5
PL/SQL procedure successfully completed.

```

-Try putting in a character and then experiment again with a number.

```

DECLARE
  v_number NUMBER;
BEGIN
  v_number:=&ui;
  DBMS_OUTPUT.PUT_LINE(SQRT(v_number));
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Something else');
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('put in a number');
END;
/

```

---

```

1  DECLARE
2    v_number number;
3  BEGIN
4    DBMS_OUTPUT.PUT_LINE('hello');
5    v_number:=&ui;
6    DBMS_OUTPUT.PUT_LINE(SQRT(v_number));
7  EXCEPTION
8    WHEN VALUE_ERROR THEN
9      DBMS_OUTPUT.PUT_LINE('put in a number');
10   WHEN OTHERS THEN
11     DBMS_OUTPUT.PUT_LINE('Something else');
12* END;
Enter value for ui: k
old   5:      v_number:=&ui;
new   5:      v_number:=k;
v_number:=k;
*
ERROR at line 5:
ORA-06550: line 5, column 17:
PLS-00201: identifier 'K' must be declared
ORA-06550: line 5, column 7:
PL/SQL: Statement ignored

```

### **Example 13.15c (Other Exceptions)**

```
--Experiment with misspelling the name, and also removing where clause.
DECLARE
    err_msg VARCHAR2(100);
    err_cde NUMBER;
    var_name employee.name%TYPE;
BEGIN
    SELECT name INTO var_name FROM employee WHERE name='jack';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('not found');
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE ('too many');
    WHEN OTHERS THEN
        err_msg := SUBSTR(SQLERRM,1,100);
        err_cde := SQLCODE;
        DBMS_OUTPUT.PUT_LINE (CONCAT(TO_CHAR(err_cde), err_msg));
        DBMS_OUTPUT.PUT_LINE(concat(SQLERRM,TO_CHAR(SQLCODE)));
END;
/
SQL> DECLARE
  2      err_msg VARCHAR2(100);
  3      err_cde NUMBER;
  4      var_name employee.name%TYPE;
  5  BEGIN
  6      SELECT name INTO var_name FROM employee WHERE name='jack';
  7  EXCEPTION
  8      WHEN NO_DATA_FOUND THEN
  9          DBMS_OUTPUT.PUT_LINE ('not found');
10      WHEN TOO_MANY_ROWS THEN
11          DBMS_OUTPUT.PUT_LINE ('too many');
12      WHEN OTHERS THEN
13          err_msg := SUBSTR(SQLERRM,1,100);
14          err_cde := SQLCODE;
15          DBMS_OUTPUT.PUT_LINE(CONCAT(TO_CHAR(err_cde), err_msg));
16          DBMS_OUTPUT.PUT_LINE(concat(SQLERRM,TO_CHAR(SQLCODE)));
17  END;
18 /
```

PL/SQL procedure successfully completed.

## **User-defined Exceptions**

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

Steps to be followed to use user-defined exceptions:

- They should be explicitly declared in the declaration section.
- They should be explicitly raised in the Execution Section.
- They should be handled by referencing the user-defined exception name in the exception section.

**Example 13.15d (Raising an exception )**

```

DECLARE
    fname  VARCHAR2(100);
    poor   EXCEPTION;
    middle EXCEPTION;
BEGIN
SELECT name INTO fname FROM employee WHERE salary = 20000;
    IF (fname='jim') THEN
        RAISE Poor;
    ELSE
        RAISE middle;
    END IF;
EXCEPTION
    WHEN poor THEN
        DBMS_OUTPUT.PUT_LINE('You need to start stealing');
    WHEN middle THEN
        DBMS_OUTPUT.PUT_LINE('You can survive');
END;
/

```

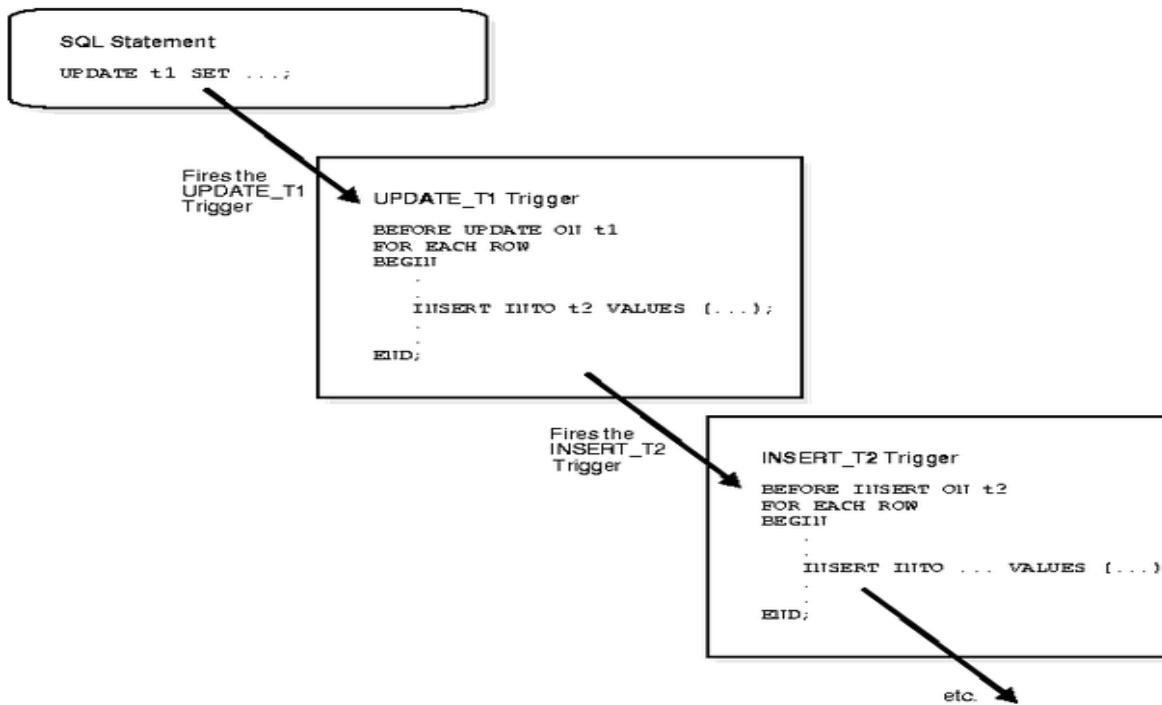
```

SQL> DECLARE
2      fname VARCHAR2(100);
3      poor EXCEPTION;
4      middle EXCEPTION;
5  BEGIN
6      SELECT name INTO fname FROM employee WHERE salary = 20000;
7      IF (fname='jim') THEN
8          RAISE Poor;
9      ELSE
10         RAISE middle;
11     END IF;
12   EXCEPTION
13     WHEN poor THEN
14         DBMS_OUTPUT.PUT_LINE('You need to start stealing');
15     WHEN middle THEN
16         DBMS_OUTPUT.PUT_LINE('You can survive');
17   END;
18 /
You need to start stealing
PL/SQL procedure successfully completed.

```



## Chapter 14 (Triggers)



*"He that falls in love with himself will have no rivals"*

Triggers are procedures that are written in PL/SQL. The main difference between a procedure and a trigger is that a procedure is executed by an explicit function call, while a trigger is executed implicitly when a condition is met- usually when a record is inserted, records are updated, or the file is closed.

- 1. You can create and replace triggers.**
- 2. The trigger body is made up of PL/SQL blocks.**
- 3. You have to own a table or to have alter or alter any table system privileges.**
- 4. You have to own a trigger or to have alter or alter any trigger system privileges.**
- 5. Row-level trigger executes once for each row in a transaction. It is the most common type of triggers (single row at once)**
- 6. Statement-level trigger executes once for each transaction. (Many rows at once)**
- 7. With Before and after triggers you will be able to reference the old and new values involved in the transaction.**
- 8. Update and delete usually reference old values. New values are the data values that the transaction creates (such as the columns in an inserted recorded)**
- 9. If you need to set a column value in an inserted row via your trigger, then you will need to use a BEFORE INSERT trigger in order to access the new value.**
- 10. Using an AFTER INSERT trigger would not allow you to set the inserted value since the row will already have been inserted.**
- 11. You may use an INSTEAD OF trigger to redirect table inserts into different table or to update multiple tables that are part of view.**
- 12. The code in the instead of trigger is executed in place of the insert, update, or delete commands you enter.**
- 13. you can combine triggertypes**
- 14. you can enable and disable triggers**
- 15. you can drop triggers.**

There are 14 types of triggers, and these are built from several basic triggers.  
There are row-level triggers, which execute once for each row in a transaction.  
There are statement-level triggers which execute once for each transaction

There are BEFORE triggers, which occur before an event takes place. There are AFTER triggers, which occur after an event takes place, and there are INSTEAD OF triggers, which occur instead of an event. Combining them, we get the following configurations:

BEFORE INSERT row  
 BEFORE INSERT statement  
 AFTER INSERT row  
 AFTER INSERT statement  
 BEFORE UPDATE row  
 BEFORE UPDATE statement  
 AFTER UPDATE row  
 AFTER UPDATE statement  
 BEFORE DELETE row  
 BEFORE DELETE statement  
 AFTER DELETE row  
 AFTER DELETE statement  
 INSTEAD OF row  
 INSTEAD OF statement

BEFORE and AFTER triggers are the only types of triggers that are available for tables. INSTEAD OF triggers can only be used for views. When a statement in a trigger body causes another trigger to be fired, the triggers are said to be *cascading*. Oracle Database allows up to 32 triggers to cascade at any one time.

### ***Example 14a (Simple Before Insert/Update)***

```
DROP TABLE employee;
CREATE TABLE employee(
    ssn    VARCHAR(10),
    lname  VARCHAR(10),
    fname  VARCHAR(10),
    DOB    DATE
);

CREATE OR REPLACE TRIGGER employee_uppercase
BEFORE INSERT OR UPDATE OF lname ON employee    FOR EACH ROW
BEGIN
    :NEW.lname :=UPPER(:NEW.lname);
    :NEW.fname :=UPPER(:NEW.fname);
END;
/

INSERT INTO employee VALUES('111','smith','bob','12-dec-56');
INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
SELECT * FROM employee;
```

```

SQL> CREATE OR REPLACE TRIGGER employee_uppercase
2 BEFORE INSERT OR UPDATE OF lname ON employee    FOR EACH ROW
3 BEGIN
4   :new.lname :=UPPER(:new.lname);
5   :new.fname :=UPPER(:new.fname);
6 END;
7 /
Trigger created.

SQL>
SQL> INSERT INTO employee VALUES('111','smith','bob','12-dec-56');
1 row created.

SQL> INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
1 row created.

SQL> SELECT * FROM employee;
SSN      LNAME      FNAME      DOB
-----  -----  -----  -----
111      SMITH      BOB       12-DEC-56
222      JONES      JACK      12-JUN-60

```

### *Example 14b (Inserting to a log file)*

```

--Creating a log table
DROP TABLE employee_log;
DROP TABLE employee;
CREATE TABLE employee(
  ssn      VARCHAR(10),  lname VARCHAR(10),  fname VARCHAR(10),  DOB      DATE
);

CREATE TABLE employee_log(
  ssn      VARCHAR(10),
  lname VARCHAR(10),
  fname VARCHAR(10),
  DOB      DATE,
  logdate  DATE
);

CREATE OR REPLACE TRIGGER employee_log
BEFORE INSERT OR UPDATE ON employee    FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO employee_log VALUES (:NEW.ssn,  :NEW.lname, :NEW.fname,
                                      :NEW.DOB, sysdate);
  ELSIF UPDATING THEN
    INSERT INTO employee_log    VALUES (:OLD.ssn,  :NEW.lname,
                                         :OLD.fname, :OLD.DOB,SYSDATE);
  END IF;
END;
/

INSERT INTO employee VALUES('111','smith','bob','12-dec-56');
INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
SELECT * FROM employee_log;

UPDATE employee SET lname='micky' , fname='mouse';

--Notice an insert was done for each record into the log table
SELECT * FROM employee_log;

```

```

SQL> CREATE OR REPLACE TRIGGER employee_log
2 BEFORE INSERT OR UPDATE ON employee FOR EACH ROW
3 BEGIN
4   IF INSERTING THEN
5     INSERT INTO employee_log VALUES (:new.ssn, :new.lname, :new.fname,
6                                     :new.dob, sysdate);
7   ELSIF UPDATING THEN
8     INSERT INTO employee_log VALUES (:old.ssn, :new.lname,
9                                     :old.fname, :old.dob,sysdate);
10  END IF;
11 /
12

Trigger created.

SQL> INSERT INTO employee VALUES('111','smith','bob','12-dec-56');
1 row created.

SQL> INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
1 row created.

SQL> SELECT * FROM employee_log;
SSN      LNAME      FNAME      DOB      LOGDATE
-----  -----  -----  -----  -----
111      smith      bob      12-DEC-56 31-MAY-15
222      jones      jack      12-JUN-60 31-MAY-15

SQL> UPDATE employee SET lname='micky' , fname='mouse';
2 rows updated.

SQL> --Notice an insert was done for each record into the log file
SQL> SELECT * FROM employee_log;
SSN      LNAME      FNAME      DOB      LOGDATE
-----  -----  -----  -----  -----
111      smith      bob      12-DEC-56 31-MAY-15
222      jones      jack      12-JUN-60 31-MAY-15
111      micky      bob      12-DEC-56 31-MAY-15
222      micky      jack      12-JUN-60 31-MAY-15

```

### *Example 14c (System table)*

#### --System table

```

DESC user_triggers;
SELECT trigger_name, trigger_body FROM user_triggers WHERE
trigger_name='EMPLOYEE_UPPERCASE';

```

#### --Dropping the table drops the trigger

```

SELECT trigger_name, trigger_body FROM user_triggers WHERE
trigger_name='EMPLOYEE_UPPERCASE';

```

#### --Can also drop the trigger

```

CREATE TABLE employee (
  ssn VARCHAR(10),
  lname VARCHAR(10),
  fname VARCHAR(10),
  DOB DATE
);

CREATE OR REPLACE TRIGGER employee_uppercase
BEFORE INSERT OR UPDATE OF lname ON employee FOR EACH ROW

```

```

BEGIN
  :NEW.lname :=UPPER(:NEW.lname);
  :NEW.fname :=UPPER(:NEW.fname);
END;
/

DROP TRIGGER employee_uppercase ;

SQL> CREATE OR REPLACE TRIGGER employee_uppercase
  2  BEFORE INSERT OR UPDATE OF lname ON employee FOR EACH ROW
  3  BEGIN
  4    :new.lname :=UPPER(:new.lname);
  5    :new.fname :=UPPER(:new.fname);
  6  END;
  7 /


Trigger created.

SQL> DROP TRIGGER employee_uppercase ;

Trigger dropped.

```

--A more complex example

```

CREATE OR REPLACE TRIGGER check_age
  BEFORE INSERT OR UPDATE    ON employee    FOR EACH ROW
DECLARE
  --Declare two variables.
  years_old NUMBER;
  error_msg CHAR(180);
BEGIN
  --The variable :new.DOB will be holding the new birth date
  --of the record to be inserted or updated. Subtract from
  --the system date and divide by 365 to get years.
  years_old :=((sysdate-:new.DOB)/365)
  --Now check to see if the new employee is under age.
  --If so, then show an error.
  IF(years_old<16) THEN
    error_msg :='Do not hire '||  :new.fname ||' '||:new.lname
    ||'.They are only '||TO_CHAR(years_old,'99.9')||'years old.';
    --Signal the user there is a problem with this data.
    --Abort the SQL statement for the current row
    RAISE_APPLICATION_ERROR(-20601,error_msg);
  END IF ;
END;
/
INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');

```

### **Example 14d (More complex example)**

```

SQL> --A more complex example
SQL> CREATE OR REPLACE TRIGGER check_age
2   BEFORE INSERT OR UPDATE    ON employee    FOR EACH ROW
3   DECLARE
4       --Declare two variables.
5       years_old NUMBER;
6       error_msg CHAR(180);
7   BEGIN
8       --The variable :new.dob will be holding the new birth date
9       --of the record to be inserted or updated. Subtract from
10      --the system date and divide by 365 to get years.  years_old :=((sy
sdate->new.dob)/365);
11      --Now check to see if the new employee is under age.
12      --If so, then show an error.
13      IF(years_old<16) THEN
14          error_msg :='Do not hire ''|| :new.fname ||' ','||:new.lname
15          ||'. They are only '||TO_CHAR(years_old,'99.9')||'years old.';
16          --Signal the user there is a problem with this data.
17          -- This also aborts the affects of the SQL statement for the
current row
18          RAISE_APPLICATION_ERROR(-20601,error_msg);
19      END IF ;
20  END;
21 /
```

Trigger created.

```

SQL>
SQL> INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
1 row created.
```

### **✓ CHECK 14A**

1. Create a trigger (Before update and insert) such that anyone whose salary is > 100000 gets a 50% bonus before the update or insert takes place.

*"You can complain because roses have thorns, or you can rejoice because thorns have roses"*

### **Example 14e (Disable/Enable)**

--Can also enable/disable triggers

```
ALTER TRIGGER check_age DISABLE;
INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
ALTER TRIGGER check_age ENABLE;
INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
ALTER TABLE employee DISABLE ALL triggers;
INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
ALTER TABLE employee ENABLE ALL triggers;
INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
```

SQL> --Can also enable/disable triggers;

SQL> ALTER TRIGGER check\_age DISABLE;

Trigger altered.

SQL> INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');

1 row created.

SQL> ALTER TRIGGER check\_age ENABLE;

Trigger altered.

SQL> INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');

1 row created.

SQL> ALTER TABLE employee DISABLE ALL triggers;

Table altered.

SQL> INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');

1 row created.

SQL> ALTER TABLE employee ENABLE ALL triggers;

Table altered.

SQL> INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');

1 row created.

### **Example 14f (Using the when syntax)**

```
DROP TRIGGER check_age;
```

```
CREATE OR REPLACE TRIGGER age_span
BEFORE UPDATE OF DOB ON employee FOR EACH ROW
WHEN (MONTHS_BETWEEN(new.DOB,old.DOB)>2)
BEGIN
```

--Adds 5 days to the new date for updates that meet condition

```
:new.DOB:=new.DOB+5;
DBMS_OUTPUT.PUT_LINE(:new.DOB);
```

```
END;
```

```
/
```

```
DELETE FROM employee;
```

```
INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
```

```
UPDATE employee SET DOB=ADD_MONTHS(DOB,3);
```

```
SELECT * FROM employee;
```

```

SQL> CREATE OR REPLACE TRIGGER age_span
2 BEFORE UPDATE OF dob ON employee FOR EACH ROW
3 WHEN (MONTHS_BETWEEN(new.dob,old.dob)>2)
4 BEGIN
5   --adds 5 days to the new date for updates that meet condition
6   :new.dob:=:new.dob+5;
7   DBMS_OUTPUT.PUT_LINE(:new.dob);
8 END;
9 /
Trigger created.

SQL>
SQL> DELETE FROM employee;
5 rows deleted.

SQL>
SQL> INSERT INTO employee VALUES('555','mickey','mouse','14-aug-97');
1 row created.

SQL>
SQL> UPDATE employee SET dob=ADD_MONTHS(dob,3);
19-NOV-97

1 row updated.

SQL> SELECT * FROM employee;
SSN      LNAME      FNAME      DOB
-----  -----  -----  -----
555      mickey     mouse     19-NOV-97

```

### *Example 14g (After syntax)*

```

DROP TABLE dept;
DROP TABLE emp;

CREATE TABLE dept
(
  deptno NUMBER ,
  total_sal NUMBER
);

CREATE TABLE emp
(
  ssn VARCHAR(10),
  deptno NUMBER,
  sal NUMBER
);

/* This trigger maintains the dept table. Upon inserts it will add revenue to the department. If the
department does not exist it will add the department. When deleting, it will deduct from the
department. When updating it will deduct and then add new value. At any point if the department does
not have any revenue, it is removed.
*/

```

```

CREATE OR REPLACE TRIGGER total_salary
AFTER DELETE OR INSERT OR UPDATE OF deptno, sal ON emp      FOR EACH ROW
DECLARE
    totSal NUMBER;
BEGIN
    --If deleting salary must be deducted from dept
    --If updating , the salary must be deducted from old department.
    --The new salary will be added later.
    IF updating then
        DBMS_OUTPUT.PUT_LINE(:old.deptno||:new.deptno);
    End if;

    IF DELETING OR UPDATING THEN
        UPDATE dept SET total_sal = total_sal - :old.sal WHERE deptno =
                                                :old.deptno;
        --if total_sal is zero, then eliminate the department.
        IF (SQL%ROWCOUNT >0) THEN
            SELECT total_sal INTO totSal FROM dept WHERE deptno =
                                                :old.deptno;
            IF (totSal = 0) THEN
                DELETE FROM dept WHERE deptno = :old.deptno;
            END IF;
        END IF;
    END IF;

    --If inserting or updating then the amounts need to be added.
    IF INSERTING OR UPDATING THEN
        UPDATE dept SET total_sal = total_sal + :new.sal WHERE deptno
                                                = :new.deptno;
        --if department does not exist, then create it and re-insert record.
        If (SQL%ROWCOUNT=0) THEN
            INSERT INTO dept VALUES (:new.deptno, 0);
            UPDATE dept SET total_sal = total_sal + :new.sal WHERE
                                                deptno = :new.deptno;
        END IF;
    END IF;
END;
/
INSERT INTO emp VALUES ('555','10','30000');
SELECT * FROM dept;

--Modifying existing department
UPDATE emp SET sal=50 WHERE ssn=555;
SELECT * FROM dept;

--Creating a new department
UPDATE emp SET sal=50,deptno=20 WHERE ssn =555;
SELECT * FROM dept;

--Deleting the employee
DELETE FROM emp WHERE ssn=555;
SELECT * FROM dept;

```

```

SQL> INSERT INTO emp VALUES ('555','10','30000');
1 row created.

SQL> SELECT * FROM dept;
  DEPTNO    TOTAL_SAL
----- -----
      10        30000

SQL>
SQL> --modifying existing department
SQL> UPDATE emp SET sal=50 WHERE ssn=555;
1010

1 row updated.

SQL> SELECT * FROM dept;
  DEPTNO    TOTAL_SAL
----- -----
      10          50

SQL>
SQL> --creating a new department
SQL> UPDATE emp SET sal=50,deptno=20 WHERE ssn =555;
1020

1 row updated.

SQL> SELECT * FROM dept;
  DEPTNO    TOTAL_SAL
----- -----
      20          50

SQL>
SQL> --Deleting the employee
SQL> DELETE FROM emp WHERE ssn=555;

1 row deleted.

SQL> SELECT * FROM dept;
no rows selected

```

### ***Example 14h (Statement level trigger)***

```

DROP TABLE employee_log;
DROP TABLE employee;
CREATE TABLE employee(
    ssn      VARCHAR(10), lname VARCHAR(10), fname VARCHAR(10), DOB      DATE );

CREATE TABLE employee_log(
    description      VARCHAR2(10),
    logdate         DATE
);

--Statement level trigger. Will trigger after the entire operation is done.
CREATE OR REPLACE TRIGGER employee_log
AFTER INSERT OR UPDATE  OR DELETE ON employee
BEGIN
    IF INSERTING THEN
        INSERT INTO employee_log VALUES ('Inserted', sysdate);
    ELSIF UPDATING THEN
        INSERT INTO employee_log VALUES ('Updated', sysdate);
    ELSIF DELETING THEN
        INSERT INTO employee_log VALUES ('Deleted', sysdate);
    END IF;
END;
/

```

```

INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
SELECT * FROM employee_log;

UPDATE employee SET lname='micky' , fname='mouse';

--Notice an insert was done into a log table for each record.
SELECT * FROM employee_log;

DELETE FROM employee;
SELECT * FROM employee_log;

SQL> INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
1 row created.
SQL> INSERT INTO employee VALUES('222','jones','jack','12-jun-60');
1 row created.
SQL> SELECT * FROM employee_log;
DESCRIPTIO LOGDATE
-----
Inserted 31-MAY-15
Inserted 31-MAY-15

SQL>
SQL> UPDATE employee SET lname='micky' , fname='mouse';
2 rows updated.

SQL>
SQL> --Notice an insert was done for each record into the log file
SQL> SELECT * FROM employee_log;
DESCRIPTIO LOGDATE
-----
Inserted 31-MAY-15
Inserted 31-MAY-15
Updated 31-MAY-15

SQL>
SQL> DELETE FROM employee;
2 rows deleted.

SQL> SELECT * FROM employee_log;
DESCRIPTIO LOGDATE
-----
Inserted 31-MAY-15
Inserted 31-MAY-15
Updated 31-MAY-15
Deleted 31-MAY-15

```

## ✓ CHECK 14B

1. Create a trigger (After insert) such that anyone who has a good personality gets a record inserted into another table (employee\_bonus) that contains only the firstname and lastname.

*"It is not what you are called, but what you answer to."*

# Appendix A (Commit, Rollback, Savepoint)

A COMMIT command issued implicitly or explicitly permanently saves the DML statements issued previously. An explicit COMMIT occurs when you enter a COMMIT statement. By default, an implicit COMMIT occurs when you exit client tools, such as SQL Developer. It also occurs if a DDL command, such as CREATE or ALTER TABLE, is issued. In other words, if a user adds several records to a table and then creates a new table, the records added before the DDL command is issued are committed automatically (implicitly). In Oracle , a transaction consists of a series of statements that have been issued and not committed. A transaction could consist of one SQL statement or 2000 SQL statements issued over an extended period. The duration of a transaction is defined by when a commit occurs implicitly or explicitly.

Unless a DML operation is committed, it can be undone by issuing the ROLLBACK command. For example, if you haven't exited Oracle since beginning to work through the examples in this chapter, executing a ROLLBACK statement reverses all the rows you entered or altered during your work in this chapter.

Along with COMMIT commands, developers sometimes use the SAVEPOINT command to create a type of bookmark in a transaction

COMMIT	Saves changed data in a table permanantly
ROLLBACK	Allows undoing uncommitted changes to data
SAVEPOINT	Enables setting markers in a transaction

```

DROP TABLE temp;
CREATE TABLE temp(num1 NUMBER);

INSERT INTO temp VALUES (10);
INSERT INTO temp VALUES (11);
SELECT * FROM temp;

SQL> DROP TABLE temp;
DROP TABLE temp
*
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> CREATE TABLE temp(num1 NUMBER);
Table created.

SQL>
SQL> INSERT INTO temp VALUES (10);
1 row created.

SQL> INSERT INTO temp VALUES (11);
1 row created.

SQL> SELECT * FROM temp;
      NUM1
-----
      10
      11
  
```

***Example 1 (Rollback)***

```
--ROLLBACK
SELECT * FROM temp;
ROLLBACK;
SELECT * FROM temp;
SQL> --ROLLBACK
SQL> SELECT * FROM temp;

    NUM1
-----
     10
     11

SQL> ROLLBACK;
Rollback complete.

SQL> SELECT * FROM temp;
no rows selected
```

***Example 2 (Savepoint)***

```
INSERT INTO temp VALUES (12);

--SAVEPOINT
SAVEPOINT level_1;
INSERT INTO temp VALUES (13);
ROLLBACK TO level_1;
SELECT * FROM temp;
SQL> INSERT INTO temp VALUES (12);
1 row created.
SQL> -- SAVEPOINT
SQL> SAVEPOINT level_1;
Savepoint created.
SQL> INSERT INTO temp VALUES (13);
1 row created.
SQL>
SQL> ROLLBACK TO level_1;
Rollback complete.

SQL> SELECT * FROM temp;

    NUM1
-----
     12
```

```
-- DDL statements
ROLLBACK;
INSERT INTO temp VALUES (15);

ALTER TABLE temp ADD num2 CHAR;

ROLLBACK;

SELECT * FROM temp;
```

```

SQL> -- ddl statements
SQL> ROLLBACK;
Rollback complete.
SQL> INSERT INTO temp VALUES (15);
1 row created.
SQL> ALTER TABLE temp ADD num2 CHAR;
Table altered.
SQL> ROLLBACK;
Rollback complete.
SQL> SELECT * FROM temp;
      NUM1 N
----- -
        15

```

### *Example 3 (Explicit Commit)*

```

-- COMMIT
INSERT INTO temp VALUES (16,'a');
COMMIT;
ROLLBACK;
SELECT * FROM temp;

SQL> -- COMMIT
SQL> INSERT INTO temp VALUES (16,'a');
1 row created.
SQL> COMMIT;
Commit complete.
SQL> ROLLBACK;
Rollback complete.
SQL> SELECT * FROM temp;
      NUM1 N
----- -
        15
        16 a

```

### *Example 4 (Autocommit)*

```

SHOW AUTOCOMMIT

--Set AUTOCOMMIT ON
SET AUTOCOMMIT ON

INSERT INTO temp VALUES (17,'c');

ROLLBACK;

SELECT * FROM temp;

```

```

SQL> SET AUTOCOMMIT ON
SQL> INSERT INTO temp VALUES (17, 'c');
1 row created.
Commit complete.
SQL> ROLLBACK;
Rollback complete.
SQL> SELECT * FROM temp;
  NUM1 N
-----
  15
  16 a
  17 c

```

### *Example 5 (Implicit commit)*

```

SET AUTOCOMMIT OFF;
SELECT * FROM temp;
INSERT INTO temp VALUES (17, 'b');
ALTER TABLE temp ADD col2 NUMBER;

SQL> SET AUTOCOMMIT OFF;
SQL> SELECT * FROM temp;
  NUM1 N
-----
  15
  16 a
  17 c
SQL> INSERT INTO temp VALUES (17, 'b');
1 row created.
SQL> ALTER TABLE temp ADD col2 NUMBER;
Table altered.

```

```

SELECT * FROM temp;
ROLLBACK;
SELECT * FROM temp;

SQL> SELECT * FROM temp;
  NUM1 N      COL2
-----
  15
  16 a
  17 b
SQL> ROLLBACK;
Rollback complete.
SQL> SELECT * FROM temp;
  NUM1 N      COL2
-----
  15
  16 a
  17 c
  17 b

```

“You have enemies? Good. That means you’ve stood up for something, sometime in your life. “

## Appendix B (Substitution)

*“If you have ten thousand regulations, you destroy all respect for the law. “*

Sometimes just adding a record to a table seems like a lot of effort, and modifying an existing record seems to take even more effort, especially if you need to add or modify 10 or 20 records. A substitution variable in an SQL command instructs Oracle to substitute a value in place of the variable at the time the command is actually executed. To include a substitution variable in an SQL command, simply enter an ampersand (&) followed by the name used for the variable. When Oracle executes the command, the user is first prompted to enter a value for the substitution variable named Region. The name of a substitution variable doesn't need to be the same as an existing column name; however, it should clearly indicate the data being requested from the user. In PL/SQL blocks, the substitution happens at compile time as opposed to runtime. This means that before a procedure or a function is created, it does the substitution first and then it attempts to compile the code.

```
drop table employee;
create table employee
(
    name      varchar(10),
    ssn       varchar(11),
    salary    number
);

insert into employee values ('john','555',10000);
insert into employee values ('jack','666',20000);
insert into employee values ('jill','777',30000);
```

### Example 1 (Define)

```
show define
set define ")"
show define
set define "&"

SQL> show define
define "&" (hex 26)
SQL> set define ")"
SQL> show define
define ")" (hex 29)
SQL> set define "&"
SQL>
```

### Example 2 (Substitution &)

```
-- Ask for InputSalary
SELECT salary FROM employee WHERE salary = &InputSalary;

SQL> SELECT salary FROM employee WHERE salary = &InputSalary;
Enter value for inputsalary: 20000
old  1: SELECT salary FROM employee WHERE salary = &InputSalary
new  1: SELECT salary FROM employee WHERE salary = 20000

SALARY
-----
20000
```

--ERROR: Must put in the quotes when entering data

```
SELECT salary FROM employee WHERE name = &name;
```

```
SQL> SELECT salary FROM employee WHERE name = &name;
Enter value for name: jill
old   1: SELECT salary FROM employee WHERE name = &name
new   1: SELECT salary FROM employee WHERE name = jill
SELECT salary FROM employee WHERE name = jill
*ERROR at line 1:
ORA-00904: "JILL": invalid identifier
```

-- If you don't want to put in the single quotes when being prompted then put it before hand

```
SELECT salary FROM employee WHERE name = '&name';
```

```
SQL> SELECT salary FROM employee WHERE name = '&name';
Enter value for name: jill
old   1: SELECT salary FROM employee WHERE name = '&name'
new   1: SELECT salary FROM employee WHERE name = 'jill'

SALARY
-----
30000
```

-- More elaborate substitution

```
SELECT name, &OTHER_FIELD FROM employee WHERE &CONDITION;
```

```
SQL> SELECT name, &OTHER_FIELD FROM employee WHERE &CONDITION;
Enter value for other_field: salary
Enter value for condition: name='jack'
old   1: SELECT name, &OTHER_FIELD FROM employee WHERE &CONDITION
new   1: SELECT name, salary FROM employee WHERE name='jack'

NAME      SALARY
-----  -----
jack        20000
```

### *Example 3 (Substitution &&)*

-- Will only ask the first time

```
SELECT name FROM employee WHERE salary = &&salary;
```

--Does not prompt for a salary the second time

```
SELECT name FROM employee WHERE salary = &&salary;
```

```

SQL> -- everytime you run this sql statement. Only prompts once
SQL> SELECT name FROM employee WHERE salary = &&salary;
Enter value for salary: 10000
old  1: SELECT name FROM employee WHERE salary = &&salary
new  1: SELECT name FROM employee WHERE salary = 10000

NAME
-----
john

SQL> --Does not prompt for a salary the second time
SQL> SELECT name FROM employee WHERE salary = &&salary;
old  1: SELECT name FROM employee WHERE salary = &&salary
new  1: SELECT name FROM employee WHERE salary = 10000

NAME
-----
john

```

```

SELECT name FROM employee WHERE name = '&&name';

--Does not prompt for salary the second time
SELECT name FROM employee WHERE name = '&&name';

SQL> undef name
SQL> SELECT name FROM employee WHERE name = '&&name';
Enter value for name: jack
old  1: SELECT name FROM employee WHERE name = '&&name'
new  1: SELECT name FROM employee WHERE name = 'jack'

NAME
-----
jack

SQL> --Does not prompt for a salary the second time
SQL> SELECT name FROM employee WHERE name = '&&name';
old  1: SELECT name FROM employee WHERE name = '&&name'
new  1: SELECT name FROM employee WHERE name = 'jack'

NAME
-----
jack

```

```

-- Give a listing of all the content
DEF

-- Include single quotes
DEF name='john'

DEF salary = 20000
--Undefine the variable
Undef salary

DEFINE NAME      = "'john'" (CHAR)
DEFINE SALARY    = "10000" (CHAR)
SQL> -- Include single quotes
SQL> DEF name='john'
SQL>
SQL> DEF salary = 20000
SQL>
SQL> -- undefine the variable
SQL> Undef salary

```

## Appendix C (Import/Export)

Exp and imp are utilities present when Oracle is installed. Their prime purpose is to move logical objects out of and into the database respectively. Oracle provides these very important tools, which we can use to import and export schemas, tables or whole database from remote database to our local database respectively. These commands will have to be issued in the DOS environment.

--Brings up all the help options  
 EXP HELP=Y

--File is the file that will be created. FULL means the entire schema, ROWS=N means that only the structure of date is exported. Default is data and structure.  
 --You can choose certain tables to export using the tables command.  
 --The name of the log file is log.log which will be overwritten everytime.  
 EXP username/password FILE=sample.dmp FULL=Y ROWS=N LOG=log.log  
 EXP username/password FILE=sample.dmp TABLES=(patient,disease) LOG=log.log

--Brings up all the help options  
 IMP HELP=Y

--File refers to the export file that was created. FULL means all the data inside the dump file.  
 --You can choose certain tables from the dump file using the tables command.  
 -- ROWS=N means that only the structure of date is exported. Default is data and structure.  
 --The name of the log file is log.log. It will be overwritten everytime.  
 IMP username/password FILE=sample.dmp FULL=Y ROWS=N LOG=log.log  
 IMP username/password FILE=sample.dmp TABLES=(patient,disease) ROWS=N LOG=log.log

*“A pessimist sees the difficulty in every opportunity; an optimist sees the opportunity in every difficulty. “*

## Appendix D (SQL-LOADER)

SQL\* Loader is a utility packaged with Oracle to load data from external files into tables. It gives you more flexibility for reading data in different formats and filtering and manipulating data during load operations.

There are three things that you need

- 1) The data file
- 2) The control file
- 3) Issuing the SQLLDR command in DOS

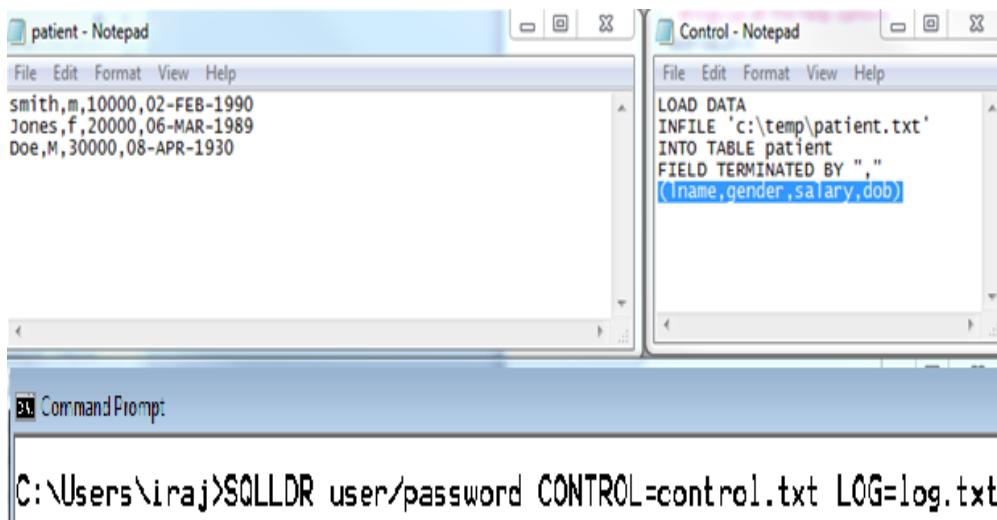
Create the control file using notepad. The contents of the control file would look something like this:

```
LOAD DATA INFILE patient.txt INTO TABLE patient FIELDS TERMINATED BY ","
(Iname,gender,salary,DOB)
```

The above loads the data that is in student\_data.txt into an Oracle Table called test. The fields in the text file are delimited with a comma and are going to be transferred in the order of fname, lname, ssn and DOB into the Oracle test table. (Table must be empty)

```
LOAD DATA INFILE patient.txt [TRUNCATE,INSERT,APPEND,REPLACE] INTO TABLE patientFIELDS
TERMINATED BY "," (Iname,gender,salary,DOB)
```

You can use one of the options in the brackets if you want to append or replace the existing data in the Oracle table (Do not include the brackets). Here is an example;



*“To improve is to change; to be perfect is to change often. “*

## INDEX

### a

add\_months, 119  
adding columns, 35  
alias, 101  
all, 136  
alter, 35  
alternate key, 18  
any, 136  
avg, 160

### b

between, 57  
between, 132  
bind variables, 294  
btitle, 12

### c

candidate, 18, 25  
cartesian, 185  
case, 125  
ceil, 113  
check, 52  
check option, 233  
commit, 320  
composite primary key, 46  
composite unique key, 50  
concatenation, 101  
connect, 8  
constants, 259  
constraints, 41  
count, 162  
create table, 29  
creating tables, 140  
cross join, 185  
cursors, 278

### d

DA, 5  
database, 2  
date functions, 118  
DBA, 5  
DBMS, 3  
decode, 123  
default, 31  
delete, 33, 61  
delete cascade, 63  
disable, 68  
disconnect, 8  
distinct, 104, 159

DML, 6  
drop constraints, 69  
drop table, 29  
dropping columns, 39  
dual, 106

### e

enable, 68  
entity, 3  
ER diagram, 20  
exceptions, 301  
exists, 220  
export, 327

### f

first normal form, 19  
flashback, 30  
floor, 113  
foreign, 25  
foreign key, 18, 58  
full outer join, 211  
functions, 289

### g

greatest, 117  
group by, 154

### h

having, 154  
history, 2

### i

if then else, 263  
import, 327  
in, 134  
indexes, 71  
initcap, 107  
inner join, 189  
insert, 32  
inserting dates, 87  
inserting numbers, 81  
inserting text, 78  
instr, 108  
intersect, 218  
is null, 139

### j

join, 189

**I**

least, 117  
 left outer join, 209  
 like, 137  
 linesize, 11  
 loops, 268  
 lower, 132  
 ltrim, 103

**m**

many to many, 23  
 max, 164  
 min, 166  
 minus, 219  
 mod, 116  
 modify, 37  
 months\_between, 119

**n**

natural join, 192  
 non-equality join, 195  
 normal forms, 17  
 not null, 31, 55  
 null, 138  
 nvl, 121  
 nvl2, 121

**o**

one to many, 22  
 order by, 146  
 outer join, 201

**p**

pagesize, 11  
 pl/sql, 13, 15, 250  
 primary key, 18, 25, 43, 45  
 procedures, 284

**r**

raising an exception, 307  
 RDBMS, 4  
 read only views, 232  
 record, 3, 260  
 recyclebin, 30  
 relational model, 2  
 rename table, 35  
 renaming column, 41  
 replace, 108, 113  
 right outer join, 209  
 rollback, 320  
 round, 113

row, 3  
 rowtype, 261  
 rtrim, 103

**s**

savepoint, 320  
 second normal form, 20  
 select, 99, 100  
 self join, 199  
 sequences, 92  
 set operators, 214  
 set unused, 40  
 sign, 124  
 spool, 9  
 SQL, 5  
 SQL\*Plus, 6  
 SQL-Loader, 328  
 subqueries, 169  
 substitution, 324  
 substitution variable, 257  
 substr, 108  
 sum, 156  
 system tables, 239

**t**

third normal form, 20  
 to\_char, 118  
 to\_number, 125  
 top-n analysis, 235  
 triggers, 308  
 trim, 103  
 trunc, 113  
 truncate, 33, 61  
 ttitle, 12  
 tuple, 3

**u**

union, 215  
 union all, 218  
 unique, 48  
 update, 142  
 upper, 249  
 user\_constraints, 44, 245  
 user\_ind\_columns, 247  
 user\_indexes, 246  
 user\_tables, 32, 245  
 user\_views, 247

**v**

variable declaration, 254  
 views, 225



