

ORACLE SQL- AN INTRODUCTION

-Compiled by Sharad Ballepu

TABLE OF CONTENTS

1.	DATAMODELS.....	2
2.	ORACLE ARCHITECTURE.....	3
3.	SQL.....	7
3.1	SQL Data types.....	7
3.2	Data Definition Language.....	8
3.3	Data Manipulation Language.....	10
3.4	Constraints.....	12
3.5	Functions.....	14
3.6	Joins.....	23
3.7	Writing Subqueries.....	27
3.8	Database Objects.....	29
4.	APPENDEX - A	35

* All SQL statements marked in [blue](#)

* All SQL keywords marked in [CAPS blue](#)

* The 9i and 10g features marked in [red](#)

* Suggestions, comments and queries can be posted to sharad_ballepu@yahoo.co.in

1. DATA MODELS:

What is a database:

Collection of information in a structural and systematic manner.

Why to use a database:

In order to store and retrieve information in a fast and efficient manner.

Understand your database model:

There are basically two types of database models.

➤ OLTP :

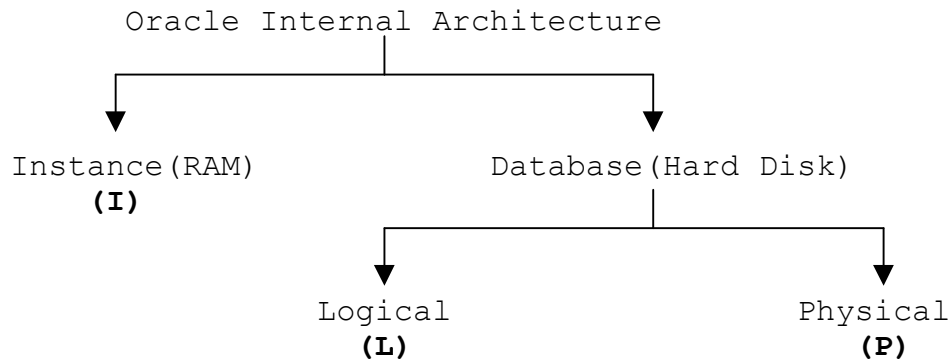
- ◆ On-Line Transaction Processing.
- ◆ Used for high volume transactions.
- ◆ Normalized data with more relations.
- ◆ Many joins, few indexes.
- ◆ Generally used by the clerical group.

➤ OLAP:

- ◆ On-Line Analytical Programming.
- ◆ Used for historical analysis.
- ◆ De-normalized data with less relations (star-schema).
- ◆ Few joins, many indexes.
- ◆ Generally used by the management group.

Based on the above pointers, figure out which type of database model would you like to implement which best suits your requirement.

2. ORACLE ARCHITECTURE



The Oracle internal architecture can typically be divided into two: the oracle instance and the database.

Let's discuss the elements that come under each of the above categories.

1) Datafile(P):

- Stores the physical information of the data contained in the database.
- A table/tablespace's data can be written in one data file or spread across multiple datafiles to increase the performance(when the table is huge and accessed very regularly).
- A datafile's size can be resized at any time
`ALTER DATABASE <datafile-path-name> RESIZE <new-size>`
- Different data file types:
 - **SYSTEM**: Contains all meta-data information. Info about the dictionary tables(tables which contain information about the user tables) is present here.
 - **SYSAUX(10g)**: Contains data about the 3rd party tools. These were earlier part of SYSTEM.
 - **UNDO**: All the uncommitted data is stored in this datafile. When the user gives the ROLLBACK command, the data is picked from here and the state is rolled back.
 - **TEMP**: This is a datafile where all the temporary computation takes place (e.g. sorting of data in a select order by clause)
 - **USERS**: USERS is the default data file created for the user tables(10g)

2) Log Files(P):

- All the transactions in the database are recorded in the log files.
- Both committed as well as uncommitted transactions are logged, the uncommitted ones have a check.
- Employs the concept of log groups (atleast 2 log groups should be defined). After the first group is over, starts copying into the other group (each group can have copies) which is called a log switch.
- This is used for recovery purpose. If all data files get corrupted, then the database can be brought to the desired state through recovery from the log files. There are tools to do this but it is a very time consuming process.

3) Control files(P):

- Contains information about the various data files and log files (file names and path).
- Can create upto 8 copies of control files for security.
- When the database is started it reads the control file to locate all the data and log files, and the database is mounted.

4) Parameter file(P):

- Called as the pfile or parameter file.
- Contains various set-up parameters to the database (e.g. db names, control file path, archive destination, buffer sizes etc).
- Saved as init.ora and resides in the <ORACLE_HOME>/database folder.
- When the database is started, it first tries to read this file and fetch all the information.

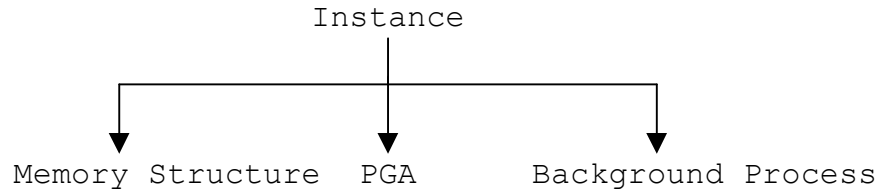
5) Server Parameter file(P) (spfile):

- Called as spfile. It is a binary file that resides on the server.
- Changes can only be done through the sql commands and not manually.
- Changes do not require a server shutdown/startup as in pfile.
- If spfile is present, reads from this file first, else goes to the pfile.

6) Tablespaces(L):

- The database comprises of the logical elements called the tablespace.
- It is a logical representation of a datafile.

In Oracle, data is stored in logical structures called data blocks, a set of data blocks form an extent and a set of extents form a segment.



7) Memory Structure (I):

The memory structure contains various buffers for various purposes.

- Data block buffer: Reads and write data from/to the data files.
 - Select list: Retrieves data from the data files and places them in the select list.
 - Dirty list: Whenever any insert/update/delete operation is performed, the data moves from the select list to the dirty list only when it is committed.
 - Undo list: The uncommitted data is stored in the undo list.
 - If there are any indexes, then we will have another list for it as well.
- Log buffer: All the transactions are written to this log buffer, which will be later moved to the log file.
- Shared SQL pool buffer:
 - Library cache: Stores the execution plan (parse, execute, fetch) and procedures, functions etc.
 - Dictionary cache: stores the meta-data.

The collection of all these 3 buffers is nothing but the SGA (System Global Area).

Whenever the user fires a sql statement, the server checks if the data is present in the data block, if yes then sends the data to the client.

If the execution plan is stored in the 'library cache' uses it to fetch the data else prepares the execution plan, fetches the data, stores the execution plan to the library cache, stores the data in the select list and then sends the data to the client. Depending upon the buffer size the plans and data are flushed out using the LRU algorithm.

Oracle uses many optimizers to fetch the data and such information is stored in the execution plan that will calculate the best path to fetch the data. The data from the buffer is moved to the data/log files in some given time interval.

8) PGA(I):

- Program Global Area.
- This is part of the shared sql pool buffer.
- The PL/SQL variables, cursors etc. are stored in this area.
- The temporary sorting of data is also performed here. If there is no buffer space available it uses the TEMP datafile for sorting.

9) Background Processes (I):

- (a) SMON: System Monitor, recovers the instance.
- (b) PMON: Process Monitor, used for eliminating deadlock.
- (c) DBWriter: Writes from dirty list to data file.
- (d) Log Writer: Writes from log buffer to log file.
- (e) Archiver: Write transactions from log file to archive destination.
- (f) Server process: Gets data from files to buffers.
- (g) User process: Sets environment at the client side.

3. STRUCTURED QUERY LANGUAGE (SQL)

SQL is a language that provides an interface to relational database system so that the user can retrieve information from the database, perform some manipulations and store data on the database.

3.1 Datatypes:

Some of the datatypes in Oracle are listed below.

Data type	Description	Default length
NUMBER	Stores integer as well as decimal values	38
INTEGER	Can store only integer values.	38
CHAR	Character string. Fixed length	2000
VARCHAR2	Character string, variable.	4000
DATE	To store date. Default format is dd-mon-yy	9
TIMESTAMP	Date with timestamp	9 (for millisecs)
TIMESTAMP WITH TIME ZONE(9i)	Date with timestamp and time zone	13
TIMESTAMP WITH LOCAL TIME ZONE(9i)	Date with local time zone	11
INTERVAL YEAR TO MONTH (9I)	Time in years and months	5
INTERVAL DATE TO SECOND (9I)	Time in days, hours, minutes and seconds	11
RAW	Binary data	2000
LONG RAW	Same as raw, size difference	2GB
BLOB	Binary Large Object	Infinity (from 10g)
BFILE	Binary file. Only reference is stores, where the image is stored in BLOB	Infinity (from 10g)
LONG	Character data.	2GB
CLOB	Character large object	Infinity (from 10g)
BINARY FLOAT (10g)	Equivalent to Java float.	
BINARY DOUBLE(10g)	Equivalent to Java double.	

3.2 Data Definition Language (DDL):

Data Definition Language is used to create, modify and drop database objects. An object in a database is nothing but a table, tablespace, procedure, function, trigger etc...We will discuss about objects in the later sections.

DDL primarily consists of the following commands

- CREATE
- ALTER
- TRUNCATE
- RENAME
- DROP
- FLASHBACK(10g)
- PURGE(10g)
- COMMENT

In the following examples we will only see the syntax for TABLES.

a) CREATE: Creates a table

```
CREATE TABLE <table-name> (<col1> <data-type1>, <col2>
<data-type2>...);
```

If the table contains many records, it can be partitioned depending upon some range.

```
CREATE TABLE <table-name> AS SELECT <col-names> FROM
<src-table>
```

Creates a table based on some other tables. The column names can be renamed in the new table as well.

b) ALTER: Alter the table structure.

```
ALTER TABLE <table-name> ADD(<col-name> <data-type>);
```

Adds the new column to the table.

```
ALTER TABLE <table-name> RENAME COLUMN <old-col-name> TO
<new-col-name>;
```

Renames a column in the table.

```
ALTER TABLE <table-name> DROP COLUMN(<col-name>);
```

Drops a column from the table.

```
ALTER TABLE <table-name> DROP (<col1>, <col2>...);
```

Drops multiple columns from the table

```
ALTER TABLE <table-name> SET UNUSED COLUMN <col-name>;
```

Sets a table as unused temporarily. This is an intermediate state where the column is present in the table but nobody can use the column. It is like logically dropping the column from the table. This can be used for various reasons like checking whether there is no effect with the column drop or to save time and drop later.

```
ALTER TABLE <table-name> SET UNUSED (<col1>, <col2>, ...);
```

Set a set of columns unused together.

```
ALTER TABLE <table-name> DROP UNUSED COLUMNS;
```

Drops the unused columns forever.

c) TRUNCATE: Deletes all records from the table.

```
TRUNCATE TABLE <table-name>
```

- Doesn't push data into the UNDO datafile, hence cannot rollback.
- Cannot use a where clause with TRUNCATE, should delete all columns.
- The table structure is as it is, only the records are deleted.

d) RENAME: Renames a table.

```
RENAME <old-table-name> TO <new-table-name>
```

e) DROP: Drops a table from the database.

```
DROP TABLE <table-name>
```

Till Oracle 9I, any table that was dropped would have been completely deleted and can be brought back only through the recovery process. But from 10g, when a table is dropped it goes into the recycle bin from where it can be fetched later.

```
DROP TABLE <table-name> CASCADE CONSTRAINTS
```

Deletes all references (foreign key) and then drops table.

```
DROP TABLE <table-name> PURGE (10g)
```

Drops the table forever.

f) FLASHBACK (10g): Retrieves the dropped table from the Recycle Bin.

```
FLASHBACK <recyclebin-name> TO BEFORE DROP RENAME TO  
<new-table-name>
```

We should know which version of the table we want to retrieve from the Recycle Bin as the same table can be dropped number of times. The only way to retrieve from the Recycle Bin is through the table name in the Recycle Bin.

g) **PURGE(10g)**: Deletes the table from the Recycle Bin.

```
PURGE {{ TABLE <table-name> | INDEX <index-name>}
      |{ RECYCLEBIN | DBA_RECYCLEBIN }| TABLESPACE
      <tablespace>[ USER <username> ]} ;
```

With the **TABLE** or **INDEX** keyword, you can choose a single table or index name

With the **USER** keyword, you can purge every copies of the object for the specified user

with the **RECYCLEBIN** keyword, you purge the current user's recycle bin content

With the **DBA_RECYCLEBIN** keyword, you can purge the whole content of the recycle bin (must have the SYSDBA system privilege)

with the **TABLESPACE** keyword, you could purge all the objects residing in the specified tablespace from the recycle bin

As of now, only the tables are stored in the Recycle Bin upon being dropped.

h) **COMMENT**: Comment on a table.

```
COMMENT ON TABLE <table-name> IS <comments>
Comment on a table.
```

```
COMMENT ON COLUMN <table-name>.<column-name> IS <comment>
Comment on a specific column of a table
```

3.3 Data Manipulation Language (DML):

Data Manipulation Language, as the name suggests, is used to manipulate data contained in the objects (e.g. tables).

a) **INSERT**: Inserts data into a table.

```
INSERT INTO <table-name> VALUES (<val1>,<val2>, ...);
Inserts all the column values .
```

```
INSERT INTO <table-name> (<col1>, <col2>...) VALUES  
(<val1>, <val2>,...);
```

Inserts only selected columns.

```
INSERT INTO <table-name> VALUES (&<col1>, &<col2>,...);
```

Insert using script. This syntax will allow the user to enter the value at runtime. When we need to run this command many number of times, we can go for this syntax.

```
INSERT INTO <table-name> SELECT <col-names> FROM <source-  
table>
```

Inserts from another table.

```
INSERT ALL  
  INTO <DESTINATION_TABLES1> VALUES (<col1>,<col2>)  
  INTO <DESTINATION_TABLES2> VALUES (<col1>,<col2>)  
  SELECT col1, col2 FROM <SOURCE_TABLE>;
```

Inserts into multiple destinations.

- b) **UPDATE**: Updates the column data in a table. Whenever update command is executed, it locks the table on those columns which are specified in the where clause (if no where clause is given then puts a lock on the entire table).

```
UPDATE <table-name> SET <col1> = <new-val1>, <col2> =  
<new-val2> [WHERE <condition>]
```

Updates the table by setting the column values to the new values specified in the SET clause.

- c) **DELETE**: Deletes the rows from the table. Like update, a delete will also hold a lock on the table on the given columns.

```
DELETE [FROM] <table-name> [WHERE <condition>]
```

- d) **MERGE(9i)**: Merges the data between two tables. This feature was introduced in 9i version but enhanced in 10g. We can achieve both INSERT and UPDATE in a single statement depending on some condition.

```
MERGE INTO <dest-table> USING <source-table> ON <join-  
condition>  
WHEN MATCHED THEN  
<perform some DML operation>  
WHEN NOT MATCHED THEN  
<perform some DML operation>
```

3.4 Constraints:

Constraints are used to impose rules on the data that is being stored into the tables. It manages the behavior of data in the table. There are table level constraints as well as column level constraints.

Column level constraints:

- ◆ NOT NULL
- ◆ UNIQUE
- ◆ PRIMARY KEY
- ◆ CHECK
- ◆ FOREIGN KEY

Constraints are defined at the time of table creation itself. They can also be added / modified after the table has been created, provided the existing data in the table is not violating any of the constraints. The general syntax for defining a constraint at column level is :

```
CREATE TABLE <table-name> (<col1> <data-type1>
[CONSTRAINT <const-name>] <const-type>);
```

Add a constraint at the column level.

```
ALTER TABLE <table-name> MODIFY (<col-name> <data-type>
[CONSTRAINT <const-name>] <const-type>);
```

Add/modify a constraint after the table is created.

```
ALTER TABLE <table-name> DISABLE/ENABLE CONSTRAINT
<const-name>
```

Enable/disable a constraint. This can be done only if the constraint has a name.

```
DROP CONSTRAINT <const-name>
```

Drops the CONSTRAINT

NOT NULL: The column can never hold a NULL value. While inserting data in the table using selected columns, be sure to enter data directly or indirectly (through triggers) into the NOT NULL columns.

UNIQUE: No two rows can have the same data i.e. there are no duplicate entries for this column.

PRIMARY KEY: The column is UNIQUE and NOT NULL at the same time. Any row can be clearly identified in the table using the primary key. There can be only one primary key in the whole table.

CHECK: Validates the columns at the time of inserting into the table. If the condition is not met, raises an error.

```
CREATE TABLE CHK_TAB (SAL NUMBER(10,4) CHECK SAL > 0)
```

The above example checks that the salary is always a positive number.

FOREIGN KEY: This constraint has a syntax a little different from the other constraints.

```
CREATE TABLE <tab-name>(<col1> <data-type1>, <col2> <data-  
type2> REFERENCES <master-table>(<mastertab-pkey>)  
[ON DELETE CASCADE / ON DELETE SET NULL])
```

A foreign key always references a column in the master table. The column in the master table should basically be a primary key. If the user tries to enter some data in the foreign key column of the child table and that particular data is not present in the master table, then an error is thrown. So, we first have to enter value in the master table and then enter the child table data.

The option ON DELETE CASCADE deletes all the rows(which has the master table deleted record) in the child table, if the master table record is deleted. The option ON DELETE SET NULL will set the column values to NULL to all the rows in the child table, if the master table record is deleted.

Table level Constraints:

When we talk about table level constraints, we can apply constraints on 2 or more columns as well and when we talk about the column level constraint 2 multiple constraints can be applied to a single column. Unless a constraint that spans over multiple columns is required, it's always better to go with the column level constraint. The keyword CONSTRAINT is mandatory for table level constraints.

The only constraints that can be used at the table are

- ◆ CHECK
- ◆ UNIQUE
- ◆ PRIMARY KEY
- ◆ FOREIGN KEY

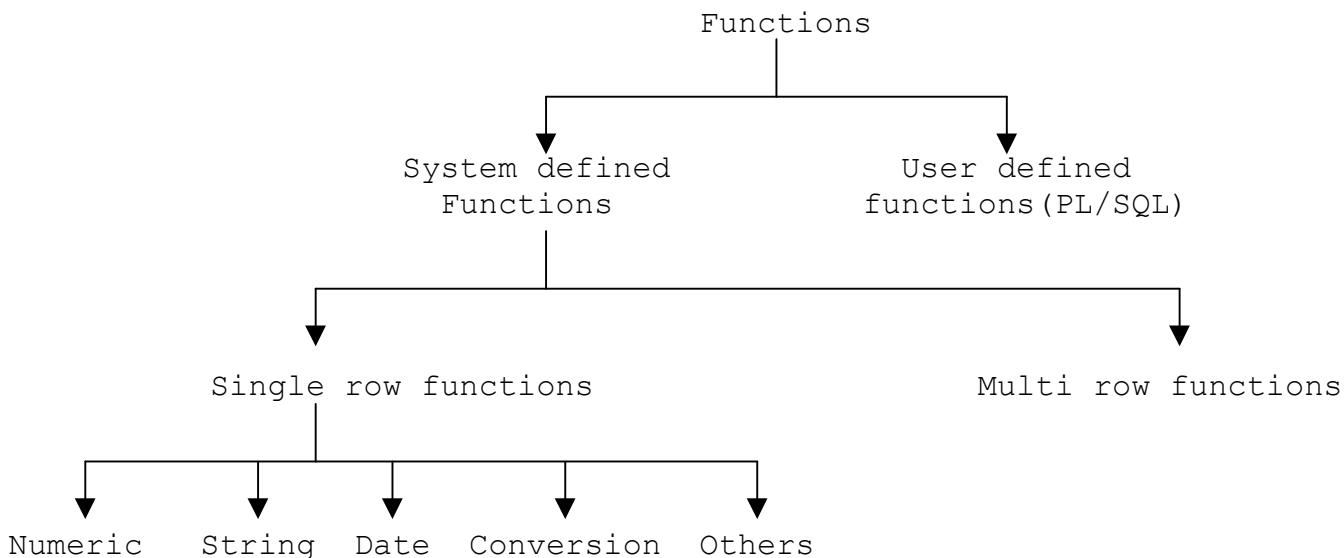
The general syntax of the table-level constraint is

```
CREATE TABLE <table-name> ( <col1> <data-type1>, <col2>
<data-type2> CONSTRAINT <const-name> <const-type>
(<columns>));
```

Again, for the foreign key the syntax is different

```
CREATE TABLE <table-name> (<col1> <data-type1>, <col2>
<data-type2>, CONSTRAINT <const-name> FOREIGN KEY (<col-
name>) REFERENCES <source-table>(<col-name>));
```

3.5 Functions:



Functions play a vital role in SQL. With the use of functions we can get the desired output with minimal code.

Functions can be broadly divided into two categories – System defined functions and user defined functions. The user can use the system-defined functions directly to get the desired functionality (defined by the system). Further, the system defined functions fall into the single-row function (which can be used on a single row only) and multi-row function (which is used to perform calculations on multiple rows for a single column).

We will discuss about some of the important functions in each of the categories. Oracle by default provides a dummy table called dual on which we can test all these functions, these functions can be applied on the user tables columns as well.

I. SINGLE-ROW FUNCTIONS:

- a) Numeric Functions: These functions may be applied on the columns with the numeric datatype (number, integer etc..).

ABS	CEIL	FLOOR	MOD	SQRT
POWER	ROUND	TRUNC	GREATEST	LEAST

ABS: Returns the absolute value of a number, i.e. the positive value of a number.

```
SELECT ABS(-10) FROM DUAL; // RESULT: 10
```

CEIL: Returns an integer value that is greater or equal to this number.

```
SELECT CEIL(10.4) FROM DUAL; // RESULT: 11
SELECT CEIL(-10.4) FROM DUAL; // RESULT: -9
```

FLOOR: Returns an integer value that is lesser or equal to this number.

```
SELECT FLOOR(10.4) FROM DUAL; // RESULT: 10
SELECT FLOOR(-10.4) FROM DUAL; // RESULT: -11
```

MOD: Takes 2 arguments. Returns the remainder after dividing the 1st by the 2nd. If 2nd argument is zero, returns the first argument.

```
SELECT MOD(10,3) FROM DUAL; // RESULT: 1
SELECT MOD(-10,3) FROM DUAL; // RESULT: -1
SELECT MOD(10,0) FROM DUAL; // RESULT: 10
```

SQRT: Returns the square root of the number.

```
SELECT SQRT(9) FROM DUAL; // RESULT: 3
```

POWER: Takes 2 arguments. Returns the value 1st argument raised to the power of the 2nd argument.

```
SELECT POWER(5,2) FROM DUAL; // RESULT: 25
```

ROUND: Returns the value rounded to the nearest integer.

```
SELECT ROUND(5.3) FROM DUAL; // RESULT: 5
SELECT ROUND(5.7) FROM DUAL; // RESULT: 6
SELECT ROUND(-5.3) FROM DUAL; // RESULT: -6
SELECT ROUND(-5.7) FROM DUAL; // RESULT: -5
```

ROUND can also take a 2nd argument, which specifies the number of decimals to be rounded.

```
SELECT ROUND(5.347,1) FROM DUAL; // RESULT: 5.3
SELECT ROUND(5.347,2) FROM DUAL; // RESULT: 5.35
```

TRUNC: Returns the truncated value. It takes 2 arguments. The first one being the number to truncate and the second one is the number of decimal places.

```
SELECT TRUNC(10.453) FROM DUAL; // RESULT: 10
SELECT TRUNC(10.453,1) FROM DUAL; // RESULT: 10.4
SELECT TRUNC(121.453,-1) FROM DUAL; // RESULT: 120
```

GREATEST: Returns the greatest value from the list of values.

```
SELECT GREATEST(10,5,13) FROM DUAL; // RESULT: 13
SELECT GREATEST(14,'5',13) FROM DUAL; // RESULT: 14
```

LEAST: Returns the least value from the list of values.

```
SELECT LEAST(10,5,34) FROM DUAL; // RESULT: 5
```

b) String Functions: Functions applied on Strings.

INITCAP	UPPER	LOWER	CONCAT	SUBSTR
INSTR	CHR	ASCII	LENGTH	LTRIM
RTRIM	TRIM	RPAD	LPAD	TRANSLATE
REPLACE	REVERSE			

INITCAP: Returns a String that sets the first character of each word to caps.

```
SELECT INITCAP('hello WORLD') FROM DUAL;
// RESULT: Hello World
```

UPPER: Returns the String with all characters in upper case.

```
SELECT UPPER('hello world') FROM DUAL;
// RESULT: HELLO WORLD
```

LOWER: Returns the String with all characters in the lower case.

```
SELECT LOWER('HELLO WORLD') FROM DUAL;
// RESULT: hello world.
```

CONCAT: Takes 2 arguments. Concatenates the first with the second string.

```
SELECT CONCAT('Hello','World') FROM DUAL;  
// RESULT: Hello World
```

SUBSTR: Returns a sub-string of a given string. Takes 3 arguments. Arg1 is the string, arg2 is the start position from where the sub-string is taken and arg3 the length of the sub-string. If length is not given then it takes all the characters till the end of the string.

```
SELECT SUBSTR('Hello World', 1, 4) FROM DUAL;  
// RESULT: Hell  
SELECT SUBSTR('Hello World', 0, 4) FROM DUAL;  
// RESULT: Hell (even if start position is 0, considers as 1)  
SELECT SUBSTR('Hello World', 7) FROM DUAL;  
// RESULT: World  
SELECT SUBSTR('Hello World', -5, 5) FROM DUAL;  
// RESULT: World (if start position is negative, starts from the end of the  
string)
```

INSTR: Returns the position of a sub-string in a String.

```
INSTR(str1, str2, [start-pos], [number]);
```

Str1 – string to be searched for

Str2 – the sub-string

Start-pos – the position from where it searches for the sub-string

Number – which occurrence of the sub-string

```
SELECT INSTR('Hello World','l',0) FROM DUAL;
```

```
//RESULT: 3
```

```
SELECT INSTR('Hello World','l',0, 2) FROM DUAL;
```

```
// RESULT: 4
```

INSTR has been enhanced from 10g to have regex for search pattern.

```
REGEXP_INSTR(<str>, <pattern-search-str>, <start-pos>,  
<occurrence>, <curr or next>, <case>)
```

Curr or next: given the current position or the next position.

Case: case sensitive or insensitive

CHR: Returns the character value for the ascii number

```
SELECT CHR(65) FROM DUAL; // RESULT: 'A'
```

ASCII: The opposite of CHR function. Returns the ASCII value for the character.

```
SELECT ASCII('A') FROM DUAL; // RESULT: 65
```

LENGTH: Returns the length of the String.

```
SELECT LENGTH('Hello World') FROM DUAL //RESULT:11
```

LTRIM: Takes two arguments. The first is the String and the second argument is the character that needs to be trimmed. By default the second argument is taken as a blank character. Trims all the characters(specified) from the left hand side of the String.

```
SELECT LTRIM(' Hello') FROM DUAL; // RESULT: Hello
SELECT LTRIM('ioioHello','io') FROM DUAL;// RESULT: Hello
```

RTRIM: Same as LTRIM, trims from the right-hand side of the String.

```
SELECT RTRIM('Hello ') FROM DUAL // RESULT: Hello
```

TRIM: Can trim from left or right or both.

```
TRIM([leading|trailing|both (trim-char) FROM] str)
By default takes both.
```

RPAD: Somewhat opposite to TRIM functions. Pads the String with the given character on the right hand side.

```
RPAD(str, length[, pad-str])
```

Str – original string

Length – total length of string after padding

Pad-str – the string to pad.

```
SELECT RPAD('Hello', 7, '$') FROM DUAL;// RESULT: Hello$$
```

LPAD: Same as RPAD, pads from the left.

TRANSLATE: Does a character to character translation.

```
SELECT TRANSLATE('Hello World', 'el','kh') FROM DUAL;
// RESULT: Hkhho Worhd
```

replaces all 'e' with 'k' and all 'l' with 'h'

REPLACE: Replaces the sequence with the new sequence

```
REPLACE(str, str-to-replace [, replacement-str]);
```

Str – original String

Str-to-replace - str to be replaced in str1.

Replacement-str - str to replace

If no replacement string is given, replaces by blank and trims.

```

SELECT REPLACE('Hello World', 'Hell', 'Cell') FROM DUAL;
// RESULT: Cello World.
SELECT REPLACE('Hello World', 'Hell') FROM DUAL;
// RESULT: o World

```

REVERSE: Reverse a String.

```

SELECT REVERSE('Hello World') FROM DUAL
// RESULT: dlroW olleH

```

c) Date Functions: Functions applied on the datatype date.

SYSDATE	ADD_MONTHS	MONTHS_BETWEEN
NEXT_DAY	CURRENT_DATE	CURRENT_TIMESTAMP
SYSTIMESTAMP	SESSIONTIMEZONE	DBTIMEZONE
LAST_DAY	LOCALTIMESTAMP	

SYSDATE: Returns the current system date and time on the local database. If the server is on some other system, gives the date and time of that system.

```

SELECT SYSDATE FROM DUAL; //RESULT: Current date in the default format.

```

ADD_MONTHS: Takes 2 arguments. The first argument is a date and the second is the number of months to be added to the date. This function returns a date after adding the given date with the number of months.

```

SELECT ADD_MONTHS('01-JAN-06', 4) FROM DUAL;
//RESULT: '01-MAY-06'

```

MONTHS_BETWEEN: Takes two date parameters and gives the difference of months between the two dates.

```

SELECT MONTHS_BETWEEN(TO_DATE('01-JAN-06', 'DD-MON-YY'),
TO_DATE('01-MAR-06', 'DD-MON-YY')) FROM DUAL; // RESULT: 3

```

```

SELECT MONTHS_BETWEEN(TO_DATE('01-MAR-06', 'DD-MON-YY'),
TO_DATE('01-JAN-06', 'DD-MON-YY')) FROM DUAL; // RESULT: 3

```

NEXT_DAY: Takes 2 arguments and returns the nearest future date that falls on the given weekday.

```

NEXT_DAY(<date>, <weekday>);

```

weekday =
SUNDAY/MONDAY/TUESDAY/WEDNESDAY/THURSDAY/FRIDAY/SATURDAY.
SELECT NEXT_DAY('01-JAN-06','WEDNESDAY') FROM DUAL; //
RESULT: '04-JAN-2006'

LAST_DAY: Returns the last day of the month for the given date.
SELECT LAST_DAY(TO_DATE('01-JAN-06','DD-MON-YY')) FROM
DUAL; // RESULT: '31-JAN-06'

CURRENT_DATE(9i): Returns the current date of the Client.
SELECT CURRENT_DATE FROM DUAL;
// RESULT: current client date.

CURRENT_TIMESTAMP(9i): Gives the current date with timestamp with
timezone of the client.
SELECT CURRENT_TIMESTAMP FROM DUAL; // RESULT: current
date with timestamp with timezone of the client.

LOCALTIMESTAMP(9i): Similar to the CURRENT_TIMESTAMP function,
but doesn't return the Time Zone along with Timestamp.
SELECT LOCALTIMESTAMP FROM DUAL;
// RESULT: current date with timestamp of the client.

SYSTIMESTAMP(9i): Returns server timestamp with timezone.
SELECT SYSTIMESTAMP FROM DUAL;

SESSIONTIMEZONE(9i): Return the session's timezone offset.
SELECT SESSIONTIMEZONE FROM DUAL // RESULT: +05:30

DBTIMEZONE(9i): Returns database timezone offset.
SELECT DBTIMEZONE FROM DUAL // RESULT: -08:00

Apart from the above standard DATE functions, there are some special date
functions too.

EXTRACT: Extract a value from a date or interval
EXTRACT (
{ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
| { TIMEZONE_HOUR | TIMEZONE_MINUTE }

```
| { TIMEZONE_REGION | TIMEZONE_ABBR }
FROM { date_value | interval_value } )
```

```
SELECT EXTRACT(YEAR FROM SYSDATE) FROM DUAL // RESULT:
2006
```

```
SELECT EXTRACT(MON FROM SYSDATE) FROM DUAL // RESULT:
NOV
```

ROUND: Rounds the given DATE to a specific unit of measure.

```
ROUND(<date>,<format>)
```

Depending on the <format> chosen the date is rounded. E.g. if the format is 'YEAR' rounds to the year, if the format is MONTH then rounds to the month etc.

```
SELECT ROUND(TO_DATE('02-JAN-2006','DD-MON-YYYY'), YEAR)
//RESULT: '01-JAN-2006'
```

```
SELECT ROUND(TO_DATE('02-SEP-2006','DD-MON-YYYY'), YEAR)
//RESULT: '01-JAN-2007'
```

TRUNC: Same as the ROUND(<date>,<format>) function, but truncates the date.

d) Conversion Functions: Used to convert from one data-type to the other.

TO_CHAR	TO_DATE	TO_NUMBER
TO_YMINTERVAL	TO_DSINTERVAL	

TO_CHAR: Converts a number or a date to a String.

```
TO_CHAR(<value>, [<format>], [nls_language])
```

Value – any date or number

Format – format you want the result to be converted into. E.g. YEAR, YY, MON, 99.99 etc..(when formatting for number 9 represents a value replaced by any number 0-9)

```
SELECT TO_CHAR(897.788,'999.9') FROM DUAL
// RESULT:      897.7
```

```
SELECT TO_CHAR(sysdate, 'Month DD, YYYY') FROM DUAL
// RESULT: November 25, 2006.
```

TO_DATE: Opposite of TO_CHAR. Takes the same number of arguments, the first argument is a sting. Converts the String value to a date.

TO_NUMBER: Converts a String to a number.

- e) Special Functions: There are some special functions that can be used on most of the datatypes.

DECODE: Decodes a value and substitutes with the new value given.

```
DECODE (<col-name>,  
        <col-value1>, <replace-val>,  
        <col-value2>, <replace-val>,  
        <default-val>);
```

SIGN: Returns 1 if the result of the expression is positive, -1 if the result is negative and 0 if equal.

NVL: Replaces the NULL value with the new value.

```
NVL (<null-col>, <replace-val>)
```

NVL2 (9i): Advanced feature of NVL.

```
NVL2 (<null-col>, <replace-if-not-null>, <replace-if-null>)
```

Replaces with the second argument value if the value of first argument is not null, if null replaces with the third argument value.

COALESCE(9i): Returns the first non-null expression in the list

```
COALESCE (<exp-1>, <exp-2>, <exp-3>...)
```

NULLIF(9i):

```
NULLIF (<arg-1>, <arg-2>)
```

If arg1 = arg2 => Returns NULL

If arg1 <> arg2 => Returns arg1

II. MULTI-ROW FUNCTIONS:

Multi-row functions are functions that can be applied on not one column of the row, but on all the rows of a column at the same time. These can be termed as the table level functions.

SUM: Adds the values of all the rows. For columns with datatype number.

```
SELECT SUM (<col-name>) FROM DUAL;
```

AVG: Finds the average of all the rows. Again this function is for the number columns. If a particular column value is NULL then doesn't consider that column for calculating the average.

```
SELECT AVG(<col-name>) FROM DUAL;
```

MAX: Returns the maximum value among a set of values.

```
SELECT MAX(<col-name>) FROM DUAL;
```

MIN: Returns the minimum value from the set of values.

```
SELECT MIN(<col-name>) FROM DUAL;
```

DISTINCT: Returns all distinct rows from the result. i.e. doesn't return any duplicate rows.

```
SELECT DISTINCT(<cols>) FROM DUAL;
```

COUNT: Returns the count of all the non-null rows.

```
SELECT COUNT(<col>) FROM DUAL;
```

STDDEV: Returns the standard deviation value.

```
SELECT STDDEV(<col-name>) FROM DUAL;
```

VARIANCE: Returns the variance value.

```
SELECT VARIANCE(<col-name>) FROM DUAL;
```

3.6 Joins

When there are relationships between tables, we would like to make use of those relationships and generate some query results based on the relationships. Joins helps us link two tables and extract the required information from the 2 tables on which the join has been performed.

There are two kinds of join syntax in Oracle

- Oracle 8i joins
- Oracle 9i joins

Let's discuss these in detail.

Oracle 8i joins:

- ◆ Simple Joins
 - Equi join
 - Non-equi join
- ◆ Outer Join
- ◆ Self Join

Simple Join: A Simple Join is one in which a column from one table is linked to a column in another table to fetch the required details from both the tables. Only the columns that match the given join condition are returned with the simple joins.

An equi join is one in which the '=' operator is used i.e. the condition that is checked for the columns in the two tables is the equality condition.

```
SELECT <cols> FROM <table-1>, <table-2>  
WHERE <table-1>.<col> = <table-2>.<col>
```

We can also use table aliases. An example of table aliases is given below.

```
SELECT <cols> FROM <table-1> [AS] tab1, <table-2> [AS] tab2  
WHERE tab1.<col> = tab2.<col>
```

In a non-equi join, the join condition is not an equality condition. It can be < (less than), > (greater than), <> (not equal to) etc..

```
SELECT <cols> FROM <table-1>, <table-2>  
WHERE <table-1>.<col> <> <table-2>.<col>
```

Outer Join: In an outer join, not only the matching columns (join condition) are retrieved but also the non-matching columns from one of the tables are retrieved. The outer join in Oracle 8i comes in two flavors – left outer join and the right outer join. For the outer join, all we need to do is to place a (+) after the column name (on the left or the right side of the equality condition). If we place the (+) on the right hand side column, it is a left-outer-join and if we place it on the left hand side column it is a right-outer-join. On the side the (+) is placed, that table returns all the columns irrespective of the match.

Assume the Emp table has all Employee details and the dept table contains the dept details. Now we want to display all employees with their department name and also those employees who don't have any dept associated. The right outer join would look like the following.

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

Now if we want to display all employee and corresponding dept details and also those departments which doesn't have any employees in it. The left-outer-join for this would be

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

Self Join: A self-join is joining within the same table. Say for example we have emp table with empno, ename and mgr, where the mgr is again an empno in the emp table only and we want to display the employee names with their managers, we have to go with a self-join. When we use a self-join, tables aliases should be used.

```
SELECT e1.ename, e2.ename FROM emp e1, emp e2 WHERE
e1.mgr=e2.empno;
```

Oracle 9i joins:

- ◆ Cross Joins
- ◆ Natural Joins
- ◆ Inner Join with USING Clause
- ◆ Inner Join with ON Clause
- ◆ Left Outer Join
- ◆ Right Outer Join
- ◆ Full OuterJoin

With 9i, Oracle has started supporting the ANSI SQL-99 syntax for JOINS. Prior to 9i, the JOINS in Oracle were very specific to the Oracle database and needed a lot of changes if someone had to do a changeover. But with 9i, Oracle started supporting the ANSI SQL-99 syntax to make it standard across the industry. From a performance aspect, there is not much difference between the two.

Some of the advantages of this new feature is:

- We can bring any SQL code and run them on Oracle easily. That would mean that we can easily port from other databases.
- Separation of JOINS from the WHERE clause conditions.

- With the new syntax the developer can get a clear picture of how the different tables are joined to each other.

The syntax for all the new 9i joins are as follows:

CROSS JOIN:

The cross join represents the Cartesian product of two or more tables selected without join conditions.

```
SELECT <cols> FROM <table-1> CROSS JOIN <table-2>
```

NATURAL JOIN:

The natural join is based on table columns with the same name and datatype. This join automatically integrates into the join condition all columns with the same name and datatype.

```
SELECT <cols> FROM <table-1> NATURAL JOIN <table-2>
```

This is the most dangerous JOINS we have, as the columns will automatically become part of the JOIN condition if they are same. We should be very careful with this JOIN and try to avoid using it.

JOIN with USING Clause:

While all matching columns of two tables are used for the join with a natural join, a specific column may be indicated explicitly for the join condition via the USING clause. Here, too, the columns must have the same name and datatype in both tables.

```
SELECT <cols> FROM <table-1> [INNER] JOIN <table-2>  
USING(<join-columns>) [WHERE <cond>]
```

JOIN with ON Clause:

Join predicates can also be defined with ON. This is necessary, for example, if the columns for the join condition do not have the same name in the two tables.

```
SELECT <cols> FROM <table-1> [INNER] JOIN <table-2> ON  
(<join-cond>) [AND <cond>] [WHERE <cond>];
```

LEFT OUTER JOIN: Similar to 8i LEFT OUTER JOIN.

```
SELECT <cols> FROM <table-1> LEFT [OUTER] JOIN <table-2> ON  
(<join-cond>) [AND <cond>] [WHERE <cond>]
```

RIGHT OUTER JOIN: Similar to 8i RIGHT OUTER JOIN.

```
SELECT <cols> FROM <table-1> RIGHT [OUTER] JOIN <table-2>  
ON (<join-cond>) [AND <cond>] [WHERE <cond>]
```

FULL OUTER JOIN: LEFT OUTER JOIN + RIGHT OUTER JOIN.

```
SELECT <cols> FROM <table-1> FULL OUTER JOIN <table-2> ON  
(<join-cond>) [AND <cond>] [WHERE <cond>]
```

3.7 Writing sub-queries

We have seen how to write simple queries and use different operators and joins with it. Queries become even more complex with the introduction of sub-queries.

Executing a query containing a sub-query, the sub-query is fired first and then the outer query is fired. Let us look at a few simple sub-query usage.

1. Calculating the maximum salary of all employees from the emp table.

```
SELECT sal FROM emp WHERE sal = (SELECT MAX(sal)  
FROM EMP);
```

The most common operators used with sub-queries are

ANY - least value from the sub-query

ALL - highest value from the sub-query

IN - value of outer query in a list of values from sub-query

EXISTS – Give more priority over IN as it comes out once a match is found.

When in SQL, it is always better to use operators than functions given a chance as the performance increases by using operators. So the same above query can also be written as

```
SELECT sal FROM emp WHERE sal >= ALL (SELECT sal  
FROM emp)
```

2. Select all employees from the emp table whose salary is more than the salary of any employee working for dept 20.

```
SELECT empno, ename FROM emp WHERE sal > ANY (SELECT  
sal FROM emp WHERE deptno=20)
```

Correlated Queries:

There is another version of the sub-queries called correlated queries. A correlated query is again a sub-query, but we can perform joins with a correlated sub-query.

A few things to keep in mind about the correlated sub-query

- First the outer query is fired and then the inner query
- Unlike the previous examples, they can have joins with the outer query
- Always use a table alias for the outer query for joining tables.

Example shows a correlated query that returns the last two rows of the emp table

```
SELECT * FROM emp a WHERE 2 > (SELECT COUNT(*) FROM emp b WHERE a.rowid < b.rowid)
```

Scalar sub-query(9i):

A query can be used in place of a column definition.

Example:

```
SELECT DNAME, COUNT(EMP.DEPTNO) FROM EMP, DEPT WHERE EMP.DEPTNO(+) = DEPT.DEPTNO GROUP BY DNAME
```

NOW instead of above we can use

```
SELECT DNAME, (SELECT COUNT(*) FROM EMP WHERE EMP.DEPTNO = DEPT.DEPTNO) FROM DEPT
```

With the scalar sub-query the system gives the outer join implicitly

In the similar manner we can use scalar sub-queries for column definitions in the WHERE and ORDER clauses too.

Inline View:

With Inline views, the output of one query will be given as the table for another. With the use of inline views we can simplify complex queries and reduce the number of joins.

With the use of Inline views, we can lessen the number of queries written. Lets take an example where the user wants to see the employee details and the average salary of the department in which the employee works, in a single resultset. This is possible using an inline view for the average salary calculation.

```
SELECT A.DEPTNO, A.SAL,A.ENAME, B.AVGSAL FROM EMP A,  
(SELECT DETPNO , AVG(SAL) AVGSAL FROM EMP GROUP BY  
DEPTNO) B WHERE A.DEPTNO=B.DEPTNO AND A.SAL>B.AVGSAL
```

3.8 Database Objects

Database objects can be categorized into two.

- Code objects
- Data objects

Code objects are objects for which there is no physical data stored separately, but they act on the data object's data and can modify them as well.

Data objects are objects for which physical data is stored and can be modified through the objects.

1) Table:

- ❖ A table is the most common object used in the database. Used to store the data. (like employee name, employee number etc)
- ❖ It is a data object
- ❖ Can be created altered, modified contents and dropped.
- ❖ We have seen a lot of operations on tables in the previous sections.

2) View:

- ❖ A code object.
- ❖ Contains a query that is pre-compiled.
- ❖ Can achieve security by hiding some columns from the user.
- ❖ Can be associated with one or more tables, but not with complex relations.
- ❖ When the view is created it is parsed and compiled, after creating a view if the underlying table structure changes (any alter command executed on the table) then the view becomes invalid and needs to be compiled again.
- ❖ If the column in the table is dropped, the view becomes permanently invalid.

```
CREATE [OR REPLACE] VIEW <view-name>[<cols>] AS  
<select-stmt> [WITH CHECK OPTION]
```

Making the view valid can be done by any one of the following

- Run the view.
- Make is ready for execution
`ALTER VIEW <view-name> COMPILE`

Dropping the view.

```
DROP VIEW <view-name>
```

3) Synonym:

- ❖ Synonyms are code objects
- ❖ They are the alternative names given to the database data objects like tables etc..
- ❖ It also has a security feature, as the user doesn't know which table/object he/she is accessing.
- ❖ Can use short names for long object names

```
CREATE [PUBLIC] SYNONYM <syn-name> FOR <obj-name>
```

By default they are private and can only be used by the owner of the object. If public access is given then all can access the synonym (provided they have appropriate privileges to the object)

Dropping a synonym.

```
DROP [PUBLIC] SYNONYM <syn-name>
```

4) Sequence:

- ❖ A data object
- ❖ Used for auto generation of numbers

<seq-name>.NEXTVAL – gives the next value in the sequence. i.e. runs the sequence.

<seq-name>.CURRVAL – gives the current value in the sequence.

Creating a Sequence

```
CREATE SEQUENCE <seq-name>  
START WITH <start-val>  
INCREMENT BY <inc-value>  
MINVALUE <min-val>  
MAXVALUE <max-val>  
CYCLE/NO CYCLE  
CACHE <cache-val>
```

start-val : The value the sequence should start with

inc-val: The value with which the sequence should increase

min-val: The minimum value the sequence can hold.
max-val: The maximum value the sequence can hold
CYCLE/NO CYCLE: When CYCLE mode is selected then the sequence can again start from first if the max value is exceeded. First the sequence starts with the <start-val> and is incremented by the inc-val each time, when it reaches the max-val for the next cycle starts with the min-val (and not the start-val). If NO CYCLE is selected then ends on reaching the maxvalue.

<cache-val>: number of values it can store in the cache.

ALTER statement can be used to change the attribute values

5) Index:

Indexes are one powerful feature that allows you to retrieve data very fast. It is a data object.

When an index is created on a table, the system internally creates another index table internally which contains pseudo columns containing the address of each row (sorted on the index column). The address is 18 characters and stored in the following format

```
1 - 6 : object ID
7 - 9 : file ID
10 - 15 : block ID
16 - 18 : sequence
```

- a) **Simple Index:** A simple index is one that is created on the table columns that may have duplicate values.

```
CREATE INDEX <index-name> ON <table-name> (<col-name>)
```

The above create statement creates an index table with the addresses of rows of the table sorted on the column on which index is applied. Now when the user tries to fetch data from the underlying table and uses the index column, the index table is parsed and the address for the matched row is fetched and information is extracted from it. This will eliminate the need to perform a full table scan. We still have to make a good design decision to make it work effectively by giving the right index columns. When there are more of inserts and updates than retrieval, having indexes would not be much effective as every insert/update on the table would also need an update on the index table.

- b) **Function based Index:** Indexes can also be created on functions. The index columns should not be used in the function.

```
CREATE INDEX <index-name> ON <table-name> (<function>)
```

- c) **Unique Index:** Indexes created on table columns that have unique columns. When we apply a UNIQUE or PRIMARY KEY constraint on any column, implicitly the UNIQUE index is created for that column. Also, when a unique index is created on a column, the UNIQUE constraint is imposed on that column implicitly.

```
CREATE UNIQUE INDEX <index-name> ON <table-name> (<col-name>)
```

- d) **Bitmap Index:** Used when we have low cardinality on a column. e.g. a gender column which has only two values male/female. They are highly compressed structures, which allows faster access.

```
CREATE BITMAP INDEX <index-name> ON <table-name> (<column-name>)
```

- e) **Reverse Key Index:** Reverses the value and stores in the index table. For example, empno of 43 is stored as 34.

```
CREATE INDEX <index-name> ON <table-name> (<column-name>) REVERSE
```

6) **Cluster:**

- ❖ Data objects.
- ❖ Stores related tables data.
- ❖ The cluster column must be indexed.

The cluster implementation is as follows

Table 1: EMP

EMPNO	ENAME	DNO
1	X	10
2	Y	10
3	Z	20

Table 2: DEPT

DNO	DNAME	LOC
10	A	I
20	B	J

Cluster on DNO:

10	A	I
	X - 1 Y - 2	
20	B	J
	Z - 3	

As it can be figured out from the above example, a cluster stores related tables' data together. In the above example a cluster is created on the department number and the cluster table holds all information from the related tables based on the department number.

Steps for creating a cluster.

- Create a cluster on a table column
`CREATE CLUSTER <cluster-name>(<col-name>
<datatype>) [TABLESPACE <tablespace-name>]`
- Create Index on the cluster
`CREATE INDEX <index-name> ON CLUSTER <cluster-
name>`
- Create the table giving the cluster name
`CREATE TABLE <tab-name>(<cols>) CLUSTER
<cluster-name>(<col-name>)`

7) DBLink:

- ❖ Combination of data-object and code-object.
- ❖ Connection between two databases.
- ❖ Allows to access objects on another database.

Creating a database link

```
CREATE DATABASE LINK <link-name> CONNECT TO  
<user-name> IDENTIFIED BY <password> USING  
<dbname>
```

Now from another database the objects from this database can be accessed.

```
SELECT <cols> FROM <username>.<table-name>@<link-  
name>
```

Using a snapshot allows us to get the copy of the table from the remote database to the local database.

```
CREATE SNAPSHOT <snapshot-name> ON <table-name>
```

4. APPENDIX - A

Data Dictionary tables:

Some of the dictionary tables used in Oracle.

user_<object-type> : objects created in the current user schema

all_<object-type> : objects created by all users

dba_<object-type> : objects owned by system

user_unused_col_tab : unused columns in a table

user_tab_comments : comments on table/column

user_constraints : all constraints on tables

user_cons_columns : constraint columns

user_objects : status of the user objects

all_objects : status of all objects

dba_objects : status of dba objects