Notes on

Oracle 10g

&

SQL

Collected by:

Zulfiqar Ali

Contents:

<<<< ---------- >>>>

SQL COMMANDS BRIEFING

1. Command Writing Style:

SQL is not case sensitive. You can write commands in the case whatever you like.

a) SeLect feaTURE, section, PAGE FROM NEWSPAPER;
b) Select Feature, Section, Page from NEWSPAPER;

SQL behaves both above commands in the same manner. Just ***Literals*** matter case. So, 'F' is not equal to 'f'. 'Page' not equal to 'page'. But F = f. ***Literals*** must be enclosed in the single quotes.

2. Datatypes and Tables (Create, Drop and Alter):

a) Datatypes:

When a table is created and the columns in it are defined, they must each have a datatype specified. Oracle has the following datatypes. These are covered within other topics.

NUMBER, CHAR, VARCHAR2, DATE, LONG, RAW, LONG RAW, BLOB, CLOB and BFILE

b) Create, Drop and Alter Tables:

E.g.

```
Create table TROUBLE (
City            Varchar2 (13) NOT NULL,
SampleDate      Date NOT NULL,
Noon            Number (3, 1),
Midnight        Number (3, 1),
Precipitation   Number);
```

Fundamentals about creating a Table:

Names of table and column must start with an alphabet, may include all characters, length may be 1-to-30 characters, must be unique within the table and can't be an Oracle keyword.
Case doesn't matter in creating tables. DATE datatype has no option. Specify max length of Character datatypes. Numbers may be either high precision specified (up to 38 digits) or specified-precision. *Case will matter if column and table names are enclosed in double quotes.*

Character Width and Number Precision:

Specifying the max length for varchar2 and CHAR and precision for Number columns has a deep insight. If **insert** has a value for a column whose width is less than value, it gives error.

E.g. Error at line 1: ORA-01401: inserted value too large for a column.

The maximum width for CHAR (fixed-length) columns is 2,000 characters. VARCHAR2 (varying-length character) columns can have up to 4,000 characters.
Number column with incorrect precision either rejects the **Insert** or drop data's precision.

E.g. Insert into TROUBLE values ('Pleasant Lake', TO_DATE ('22-JUN-1999', 'DD-MON-YYYY'), 101.44, 86.2,1.63);

Error at line1: ORA-01438: value larger than specified in precision allowed for this column

Noon column is specified as NUMBER (3, 1). So, the column can have 2 numbers before and 1 after the decimal point. Can't override this rule. This error caused by 101, not the **.**44. The **.**44 is rounded but 101 caused error. If NUMBER (4, 1) is in **create table**, no error here.

E.g. Insert into TROUBLE values ('Pleasant Lake', TO_DATE ('22-JUN-1999', 'DD-MON-YYYY'), 101.44, 86.2,1.63);

Oracle responds: 1 row created

E.g. **For precision of NUMBER (4, 1)**

123.4 = 123.4 123.44 = 123.4
123.44 = 123.4 1234.5 = **insert** fails

For precision of NUMBER (4)

123.4 = 123 123.445 = 123
1234.5 = 1234 12345 = **insert** fails

For precision of NUMBER (4, -1)

123.4 = 120 123.445 = 120
125 = 130 1234.5 = 1230 12345 = **insert** fails

For precision of NUMBER

123.4 = 123.4 123.445 = 123.445
1234.5 = 1234.5 12345.67890123 = 12345.67890123

Constraints in Create Table:

The **create table** statement lets you enforce some single column or group of columns constraints: candidate keys, primary keys, foreign keys and check conditions. The more constraints you add to a table definition, the less work you have to do in applications to maintain the data. On the other hand, the more constraints there are in a table, the longer it takes to update the data. Two ways to add constraints: as part of column definition (a *column* constraint) or at the end of the **create table** (a *table* constraint). Clauses that constraint several columns must be table constraints.

(i) The Candidate Key:

A *candidate key* is a combination of one or more columns, the values of which uniquely identify each row of a table. Creating UNIQUE constraint for TROUBLE table:

E.g.
```
Create table TROUBLE (
City          Varchar2 (13) NOT NULL,
SampleDate    Date NOT NULL,
Noon          Number (4, 1),
Midnight      Number (4, 1),
Precipitation Number
Constraint TROUBLE_UQ UNIQUE (City, SampleDate));
```

The key of this table is the combination of City and SampleDate. Notice that the both columns are also declared to be NOT NULL.

(ii) The Primary Key:

A *Primary key* of a table is one of the candidate keys that you give some special characteristics. You can have only one primary key, and a primary key column cannot contain NULLs:

E.g.

```
Create table TROUBLE (
City            Varchar2 (13),
SampleDate      Date,
Noon            Number (4, 1),
Midnight        Number (4, 1),
Precipitation   Number
Constraint TROUBLE_PK PRIMARY KEY (City, SampleDate));
```

The **create table** statement has the same effect as the previous one, except that you can have several UNIQUE constraints but only one PRIMARY KEY constraint. For single-column primary or candidate keys, you can define the key on the column with a column constraint instead of a table constraint:

E.g.

```
Create table WORKER (
Name            Varchar2 (25) PRIMARY KEY,
Age             Number,
Loading         Varchar2 (15));
```

(iii)The Foreign Key:

A *Foreign key* (*referential integrity constraint*) is a combination of columns with values based on the primary key from another table.

E.g.

```
Create table WORKER (
Name            Varchar2 (25),
Age             Number,
Loading         Varchar2 (15)
Constraint WORKER_PK PRIMARY KEY (Name)
Foreign key (Lodging) REFERENCES LODGING (Lodging));
```

You can refer to a primary or a unique key, even in the same table. You can't refer to a table in a remote database in the **references** clause. You can use the table form (which is used here to create a PRIMARY KEY on the TROUBLE table) instead of the column form to specify foreign keys with multiple columns. While deleting the real row, you must first have to delete the dependent rows first or make the dependent column NULL first. Otherwise, you'll get an error for **delete**. The clause **on delete cascade** added to the **references** clauses tells Oracle to delete the dependent row when you delete the corresponding row in the parent table. This action automatically maintains referential integrity.

(iv)The Check Constraint:

Many columns must have values that are within a certain range or that satisfy certain conditions. With a *CHECH constraint,* you can specify an expression that must always be true for every row in the in the table.

E.g.

```
Create table WORKER (
Name            Varchar2 (25),
Age             Number CHECK (Age BETWEEN 18 AND 65),
Loading         Varchar2 (15)
Constraint WORKER_PK PRIMARY KEY (Name)
Foreign key (Lodging) REFERENCES LODGING (Lodging));
```

A column-level CHECK constraint can't refer to values in other rows; it can't use the pseudo-columns. SysDate, UID, User, UserEnv, CurrVal, NextVal, Level or RowNum. You can use the table constraint form (as opposed to the column constraint form) to refer to multiple columns in a CHECK constraint.

Naming Constraints:

The **constrain** clause of the **create table** command names the constraint (like, WORKER_PK). You may use this constraint name later when enabling or disabling constraints.

Creating a Table from a Table:

Oracle lets you create a new table on the fly, based on a **select** statement on an existing table:

E.g.
```
Create table RAIN as
Select * City, Precipitation from TROUBLE;
```

c) Dropping Tables:

Dropping tables is very simple. You use the words **drop table** and the table name.

E.g.
```
Drop table TROUBLE;
```

Oracle responds: Table dropped

To empty a table, instead of dropping table **truncate** command is used. This can't be rolled back.

E.g.
```
Truncate table TROUBLE;
```

Oracle responds: Table truncated

d) Altering Tables:

Three ways to **alter table**s: add a column in an existing table, change a column's definition or drop a column. Adding a column is straight forward and similar to creating a table, but having NOT NULL will generate error message.

E.g.
```
Alter table TROUBLE add (
        Condition Varchar2 (9) NOT NULL,
        Wind Number (3));
```

Error at line1: ORA-01758: table must be empty to add mandatory (NOT NULL) column

If Condition is not specified as NOT NULL **alter table** will succeed. Adding a NOT NULL column will work with only empty table, because it will have as much rows empty as the table have already that spoils the NOT NULL check. So, in this situation, add a column, fill all rows with data by using **update** and then **alter table** and modify column as NOT NULL. This is the process:

E.g.
```
Alter table TROUBLE add (
        Condition Varchar2 (9)
        Wind Number (3);
```

Oracle responds: Table altered

E.g.
```
Update TROUBLE set Condition = 'Sunny';
```

Condition column in all rows are updated.

E.g.
```
Alter table TROUBLE modify (
        Condition Varchar2 (9) NOT NULL
        City Varchar2 (17);
```

Oracle responds: Table altered *(City column is made wide 13 to 17 characters)*

To make a NOT NULL column nullable use the **alter table** command with the NULL clause:

E.g. Alter table TROUBLE modify (Condition NULL);

The Rules for Adding or Modifying a Column:

- You may add a column at any time if **NOT NULL** isn't specified.
- You may add a **NOT NULL** column in three steps:
  1. Add the column without **NOT NULL** specified.
  2. Fill every row in that column with data.
  3. Modify the column to be **NOT NULL**.

There are the rules for modifying a column:

- You can increase a character column's width at any time.
- You can increase the number of digits in a NUMBER column at any time.
- You can increase or decrease the number of decimal places in a NUMBER column at any time.

In addition, if a column is **NULL** for every row of the table, you can make any of these changes:

- You can change column's datatype. (If column is **NULL** or totally empty).
- You can decrease a character column's width.
- You can decrease the number of digits in a NUMBER column.

e) Dropping a Column:

Although, not impossible, dropping a column is very complicated than adding or modifying one, because it is very time-consuming. You can drop a column immediately or mark it as "unused" to be dropped at a later time when database is used less heavily. This option is only in **Oracle 8i**.

E.g. Alter table TROUBLE drop column Wind;

E.g. Alter table TROUBLE set unused column Wind;

E.g. Alter table TROUBLE drop unused columns;

E.g. Alter table TROUBLE drop (Condition, Wind); // for multiple columns

Note: To see all tables with columns marked as unused query USER_UNUSED_COL_TABS, DBA_UNUSED_COL_TABS, and ALL_UNUSED_COL_TABS. Once you have marked a column as "unused" you cannot access that column. When dropping multiple columns keyword **column** is not used, rather column names are enclosed in parentheses.
If the dropped columns are part of primary keys or unique constraints, you will need to also use the **cascade constraints** clause as part of your **alter table** command. If you drop a column that belongs to a primary key, Oracle will drop both the column and the primary key index.

Note: Oracle provides a built-in table DUAL, with one-row and one-column.

3. Relationship Concept & Temporary Tables:

Relationships Concept (Combining Tables):

If information from two tables is required in one Query, refer the column names from both tables in **Select** Clause and both tables in *From Clause* and if any column name is same in both tables **must** write table + period (**.**) + Column name. For example LOCATION.City, WEATHER.City. The resulting table is called a *Projection,* or *Result Table.* This logic is also called **Joining Tables.**

E.g. select WEATHER.City, Condition, Temperature, Latitude,

```
                    NorthSouth, Longitude, EastWest
From WEATHER, LOCATION
Where WEATHER.City = LOCATION.City;
```

This query selects mentioned columns from both tables but rows will be only that where *City* is same in both tables. All other columns and records will be ignored.

Using Temporary Tables:

As of Oracle8i, you can create a table that exists solely for your session, or whose data persists for the duration of your transaction. You can use temporary tables to support specialized rollups or specific application processing requirements.
To create a temporary table, use the **create global temporary table** command. When you create a temporary table, you can specify whether it should last for the duration of your session (via **on commit preserve rows** clause) or whether its rows should be deleted when the transaction completes (via the **on commit delete rows** clause). Unlike a permanent table, a temporary table does not allocate space when it is created. Space will be dynamically allocated for the table as rows are inserted.

E.g.

```
create global temporary table YEAR_ROLLUP (
        Year        Number (4),
        Month       Varchar2 (9),
        Amount      Number)
On commit preserve rows:
```

4. What is a View? :

A view is a way of hiding the logic that created the joined table just displayed. In spite of some restrictions it is treated as a real table.

Creating View:

E.g.

```
Create view INVASION as
Select WEATHER.City, Condition, Temperature,
Latitude, NorthSouth, Longitude, EastWest
From WEATHER, LOCATION
Where WEATHER.City = LOCATION.City;
```

Now no needs to specify the table for City column just make the simple query. There will be some Oracle Functions you won't be able to use in a view that you can use on a plain table, but they are few, and mostly involves modifying rows and indexing tables.

Note: View doesn't contain any data. Tables contain data as of Oracle8i, you can create "Materialized Views" that contain data, but they are truly tables, not views.

E.g.

```
select City, Condition, Temperature, Latitude, NorthSouth,
Longitude, EastWest
From INVASION;
```

Replacing View:

E.g.

```
Create or replace view INVASION as
Select WEATHER.City, Condition, Temperature, Latitude, NorthSouth, Longitude,
EastWest
From WEATHER, LOCATION
Where WEATHER.City = LOCATION.City and country = 'GREECE';
```

Rules for insert, update and delete:

If a view is based on a single underlying table, you can **insert, update** or **delete** rows in the view. This will actually insert, update or delete rows in the underlying table. The rules for these are:

- You can't **insert** if the underlying table has any **NOT NULL** columns that don't appear in the view.
- You can't **insert** or **update** if any of the view's columns referenced in the insert update or delete contains function or calculations.
- You can't **insert**, **update** or **delete** if the view contains **group by**, **distinct** or a reference to the pseudo-column RowNum.
- You can **insert** into a view based on multiple tables if Oracle can determine the proper rows or insert. In a multi-table view, Oracle determines which of the tables are *key-preserved*. If a view contains enough columns from a table to identify the primary key for that table, then the key is preserved and Oracle may be able to insert rows into the table via the view.

Stability of a View:

View does not have any data in itself. It gets data from the underlying table just at the querying moment. But materialized views are exception. These are similar to tables. If the view is created using **select *,** all **alter** commands on table are valid in the view, either you add a column or drop a column from the table.

Order by in Views:

You can't use an **order by** in a **create view** statement. Occasionally, a **group by** which can be used work same like **order by**. The following both queries will result same data:

E.g.
```
Select City, Precipitation from COMFORT
Order by Precipitation;
```

E.g.
```
Create view DISCOMFORT as
Select City, Precipitation from COMFORT
Group by City, Precipitation;
```

E.g.
```
Select * from DISCOMFORT;
```

The following query will work better for this purpose:

E.g.
```
Create view DISCOMFORT as
Select City, Precipitation from COMFORT
Group by City, Precipitation, RowNum;
```

Creating Read-Only View:

You can use **with read only** clause of the **create view** command to prevent users from manipulating records via the view, for example to prevent form **insert, update** or **delete** via the views. If the view is based on a join of multiple tables, the user's ability to update the view is limited; a view's base tables can't be updated unless only one table in involved in the **update** and the updated tables full primary key is included in the view's column.

E.g.
```
Create or replace view DISCOMFORT as
Select * from COMFORT
With read only;
```

Reality of Views:

Views are not snapshots if the data as certain point in the past. They are dynamic, and always reflect the data in the underlying tables. The data in a table is changed, any view created with that table changes as well.

5. Describe and Select Table:

Describe TableName
E.g. describe NEWSPAPER

It shows all column names, data types and Null/Not Null information.

E.g. *select column1, column2, and column3…*
From table
Where (condition);

E.g. select feature, section, page
From NEWSPAPER
Where section = 'F';

6. Logical Operators:

Operators are used in **where clause.**

a) Logical Test against Single-Values: -
Equal, Greater than, Less than, Not Equal
Page = 6, Page > 6, Page >= 6, Page < 6, Page <= 6,
Page != 6, Page ^= 6, Page <> 6 *(all ways are Page not equal to 6)*

LIKE

Feature LIKE 'Mo%'	Feature begins with the letter Mo
Feature LIKE '__I%'	Feature has an I in the third Position
Feature LIKE '%O%O%'	Feature has two O's in it

*(**LIKE** performs Pattern matching. An underscore (_) represents one space or character whatever that is, A percent sign (%) represents any number of spaces or characters)*

IS NULL, IS NOT NULL

Page IS NULL	Page is unknown
Page IS NOT NULL	Page is known

*(**NULL** tests to see if data exists in a column for a row. If the column is completely, it is said to be **NULL**. The word **IS** must be used with **NULL** and **NOT NULL:** equal, greater than or less than signs do not work with this)*

b) Logical Test against a list of Values: -

Logical Tests with numbers:

Page IN (1, 2, 3)	Page is in the list (1, 2, 3)
Page NOT IN (1, 2, 3)	Page is not in the list (1, 2, 3)
Page BETWEEN 6 AND 10	Page is equal to 6, 10 or anything in between
Page NOT BETWEEN 6 AND 10	Page is below 6 or above 10

Logical Tests with letters (or characters):

Section IN ('A','C', 'F')	Section is in the list ('A','C', 'F')
Section NOT IN ('A','C', 'F')	Section is not in the list ('A','C', 'F')
Section BETWEEN 'B' AND 'D'	Section is equal to 'B', 'D' or anything in between (alphabetically)

Section NOT BETWEEN 'B' AND 'D' Section is below 'B' or above 'D' (alphabetically)

AND-OR Logic:

AND requires true results from the both sides. *OR* requires one result true.

7. Functions:

A function is a predefined operation. Functions in Oracle work in one or two ways:

(i) Create new object from the old ones. (ii) Just produce results from the existing objects.

8. Managing Strings and String Functions:

A string is a mixture of letters, punctuation marks, numbers and spaces. A string is stored in two datatypes: VARCHAR2, CHAR (pronounced as "care").

CHAR for fixed length string fields and VARCHAR2 for all other character string fields. Some functions have a pair of parenthesis for the value to work on and set of option for it. You can type a column name in them or a string, column name without single quotes '' and string must have quotes. All the string functions as listed below:

(i)	| |, CONCAT	(ii)	LOWER, UPPER, INITCAP
(iii)	LENGTH	(iv)	LPAD, RPAD
(v)	LTRIM, RTIM	(vi)	SOUNDEX
(vii)	SUBSTR	(viii)	INSTR

(i) | |, CONCAT

These are concatenation functions. These glue columns or strings together with no spaces in between.

E.g. Select City | | Country from LOCATION;

Oracle responds:

CITY | | COUNTRY
ATHENSGREECE
CHICAGOUNITED STATES

E.g. Select CONCAT (City, Country) from LOCATION;

Oracle responds:

CONCAT (CITY, COUNTRY)
ATHENSGREECE
CHICAGOUNITED STATES

If you want to place a *space between two words* type the below query:

E.g. select City | | ', ' | | Country from LOCATION;

Oracle responds:

CITY | | ', ' | | COUNTRY
ATHENS, GREECE
CHICAGO, UNITED STATES

(ii) LOWER, UPPER, INITCAP

These functions are used for changing the case of any column or string.

E.g. Select City, UPPER (City), LOWER (City), INITCAP (City),
LOWER ('City') from LOCATION;

Oracle responds:

City	LOWER (City)	UPPER (City)	INITCAP (City)	LOWER ('City')
ATHENS	athens	ATHENS	Athens	city

CHICAGO chicago CHICAGO Chicago city

(iii) LENGTH

This is used for counting the letters in a string or a column.

E.g. Select City, LENGTH (City) from LOCATION;

Oracle responds:

CITY	LENGTH (City)
ATHENS	6
Chicago	7

(iv) LPAD, RPAD

These functions can wide any column as long as you set and align column left or right side filling the rest of width with the one of spaces, commas, periods, numbers, pound sign (#), or exclamation sign (!). *Syntax:*

RPAD (*string*, *length* [, '*set*'])
LPAD (*string*, *length* [, '*set*'])

String is a CHAR or VARCHAR2 column; length is its width and set is the set of characters that do padding. *The set must be in single quotes.* Default set is space.

E.g. Select RPAD (City, 10, '.'), Country from LOCATION;

Oracle responds:

RPAD (CITY, 15, '.')	COUNTRY
ATHENS ……	GREECE
CHICAGO ……..	UNITED STATES

E.g. Select LPAD (City, 10, '.'), Country from LOCATION;

Oracle responds:

LPAD (CITY, 15, '.')	COUNTRY
……ATHENS	GREECE
……..CHICAGO	UNITED STATES

E.g. Select RPAD (City, 10), LPAD (City, 10), Country from LOCATION;

Oracle responds:

RPAD (CITY, 15, '.')	LPAD (CITY, 15, '.')	COUNTRY
ATHENS	ATHENS	GREECE
CHICAGO	CHICAGO	UNITED STATES

This example uses the default set for padding, spaces.

(v) LTRIM, RTRIM

These functions are for removing (trimming) unwanted characters from the left or right ends of the string or column. *Syntax:*

RTRIM (*string* [, '*set*'])
LTRIM (*string* [, '*set*'])

String is a CHAR or VARCHAR2 column and set is the collection of characters you want to trim off. If no set is specified, the functions trim off spaces. More than one character can be entered in set to trim off.

E.g. Select RTRIM (Title, '."') From MAGAZINE;

This command will remove " and periods (.) from the right end of the given column.

E.g. Select LTRIM (Title, '"') from MAGAZINE;

This command will remove " from the left of the given column.

Combining Two Functions:

E.g. Select LTRIM (RTRIM (Title, '.'"),'"') from MAGAZINE;

This command trims first (.) and (") from the right end of the column and then removes (") from the left of the column. Now the column presents the actual Title. The better way for combining functions, without confusing, is to use one line for one function. Trim functions are designed to remove just letters. So, *don't try to trim a complete word*. If it is tried it will match every spelling with all rows and trim the matching letters.

E.g. Select name, RPAD (LTRIM (RTRIM (Title, '.'"),'"'), 20, '-^') from MAGAZINE;

This command will do also to right pad the Title column.

(vi) SOUNDEX:

It finds words that sound like other words but spellings are different. But the *both words must begin with the same letter*. It is used in a *where clause*. Syntax:

SOUNDEX (*string*)

E.g. Select City, Temperature, Condition from WEATHER
Where SOUNDEX (City) = SOUNDEX ('Sidney');

Oracle responds:

CITY	TEMPERATURE	CONDITION
Sydney	29	Snow

E.g. Select City, Temperature, Condition from WEATHER
Where SOUNDEX (City) = SOUNDEX ('menncestr');

Oracle responds:

CITY	TEMPERATURE	CONDITION
Manchester	66	Sunny

E.g. Select a.LastName, a.FirstName, a.Phone
From ADDRESS a, ADDRESS b
Where a.LastName != b.LastName and
SOUNDEX (a.LastName) = SOUNDEX (b.LastName);

Oracle responds:

LASTNAME	FIRSTNAME	PHONE
SZEP	FELICIA	214-522-8383
SEP	FELICIA	214-522-8383

This query selects all duplicate entries from ADDRESS table with slight difference in spellings.

(vii) SUBSTR:

This function is used to clip out a piece of a string. Syntax:

SUBSTR (*string, start* [*, count*])

This function clips out a subsection from *start* number of character to the *count* number. If count is not mentioned it will go to the end of the *string*.

E.g. Select SUBSTR (Name, 6, 4) from MAGAZINE;

Oracle responds:

SUBS
FREE
OLOG

E.g. Select LastName, FirstName, SUBSTR (Phone, 5) from ADDRESS
Where Phone like '415-%';

Oracle responds:

LASTNAME	FIRSTNAME	SUBSTR (P
ADAMS	JACK	453-7530

ZACK JACK 620-6842

This query clips out Phone no. and area code from International no. It starts clipping from 5th point to the end. If *start* value is negative numbers, clipping process starts from last character to the left side and *start* position considers as *count* value. *Count* value must be positive otherwise it returns NULL result.

E.g. Select SUBSTR (Phone, -4) from ADDRESS
Where Phone like '415-%';

Oracle responds:

SUBS
7512
6252

(viii) INSTR:

It searches a set of characters from a string, it doesn't clip anything off, and rather it just tells the place where the set is found. It is somewhat similar LIKE function and can be used in anywhere in the query without *from clause.* Syntax:

INSTR (*string, set* [, start [, *occurrence*]])

String is from what to search *set* of characters. *Start* is the position of character in the *string* for starting search. If character occurs more than once in the *string,* tell Oracle to what *occurrence* you want to search.

E.g. Select Author, INSTR (Author, '*O*') from MAGAZINE;

Oracle responds:

AUTHOR	INSTR (AUTHOR, '*O*')
Chesterton, G.K.	2
Crookes, William	3

E.g. Select Author, INSTR (Author, '*O*', 1, 2) from MAGAZINE;

Oracle responds:

AUTHOR	INSTR (AUTHOR, '*O*', 1, 2)
Chesterton, G.K.	0
Crookes, William	4

E.g. Select Author, INSTR (Author, 'WILLIAM') from MAGAZINE;

Oracle responds:

AUTHOR	INSTR (AUTHOR, 'WILLIAM')
Chesterton, G.K.	0
Crookes, William	10

E.g. Select Author, INSTR (Author, ',') from MAGAZINE;

Oracle responds:

AUTHOR	INSTR (AUTHOR, ',')
Chesterton, G.K.	11
Crookes, William	8

If reformatting of Author column as removing comma and the last word to first is needed:

E.g. (i) Select Author, SUBSTR (Author, 1, INSTR (Author, ',') -1) from MAGAZINE;

Oracle responds:

AUTHOR	SUBSTR (Author, 1, INSTR (AUT
Chesterton, G.K.	Chesterton
Crookes, William	Crookes

E.g. (ii) Select Author, SUBSTR (Author, INSTR (Author, ',') +2) from MAGAZINE;

Oracle responds:

AUTHOR	SUBSTR (Author, INSTR (AUT
Chesterton, G.K.	G.K.
Crookes, William	William

E.g. (iii) column ByFirstName heading "By First Name"

```
Select Author, SUBSTR (Author, INSTR (Author, ',') +2)
        || ' ' ||
        SUBSTR (Author, 1, INSTR (Author, ',') -1) as ByFirstName
From MAGAZINE;
```

Oracle responds:

AUTHOR	By First Name
Chesterton, G.K.	G.K. Chesterton
Crookes, William	William Crookes
Whitehead, Alfred	Alfred Whitehead
Ruth, George Herman	George Harman Ruth

Order By and Where with String Functions:

E.g.
```
Select City from WEATHER
Where Length (City) < 7;
```

Oracle responds:

CITY
LIMA
ATHENS
PARIS

E.g.
```
Select City from WEATHER
Order by Length (City);
```

Oracle responds:

CITY
LIMA
PARIS
ATHENS

Using INSTR as LIKE function:

E.g.
```
Select Author from MAGAZINE
Where INSTR (Author, 'O', 1, 2) > 0;
```

Oracle responds:

AUTHOR
BONHOEFFER, DIETRICH
CROOKES, WILLIAM

This query selects Authors that have a second *O* in their names. It works same as:
```
Where Author LIKE '%O%O%'
```

9. Playing with Number and Number Functions:

Number functions have three classes:
(1) Single-Value Functions (2) Group-Value Functions
(3) List-Value Functions

a) Single-Value Functions:

These functions work on a single value.

(i) + - * / (ii) NVL (iii) ROUND, TRUNC (iv) CEIL, FLOOR

(v) ABS, SIGN (vi) POWER, LOG, SQRT (vii) MOD

(i) + - * /:

E.g.
```
Select Above + Below As Plus, Above - Below As Minus,
       Above * Below As Times, Above + Below As Divided
From MATH
Where Name = 'HIGH DECIMAL';
```

Oracle responds:

PLUS	MINUS	TIMES	DIVIDED
-11.111	144.443	-5185.081482	.857143

(ii) NVL:

Syntax: NVL (*value*, *substitute*)

Substitute can be a literal number, another column or a computation. This function can be used with CHAR, VARCHAR2, DATE and other datatypes. But the value and substitute must be the same datatype.

Note: Any arithmetic operation that includes a NULL value has a NULL value as a result.

E.g.
```
Select Above + Empty As Plus, Above - Empty As Minus,
From MATH
Where Name = 'HIGH DECIMAL';
```

Oracle responds:

PLUS	MINUS

E.g.
```
Select Client, NVL (Weight, 43) AS NVL from SHIPPING;
```

Oracle responds:

CLIENT	NVL
Johnson Tool	59
DAGG Software	27
Tully Andover	43

(iii) ROUND, TRUNC:

ROUND rounds numbers to a given of digits of precision. **TRUNC** truncates or chops off, digits of precision from a number. Syntax:

ROUND (*value*, *precision*); TRUNC (*value*, *precision*);

E.g.
```
Select Above, ROUND (Above, 2), TRUNC (Above, 2) from MATH;
```

Oracle responds:

ABOVE	ROUND (Above, 2)	TRUNC (Above, 2)
66.666	66.67	66.66
33.33	33.33	33.33

E.g.
```
Select Above, ROUND (Above, 0), TRUNC (Above, 0) from MATH;
```

Oracle responds:

ABOVE	ROUND (Above, 0)	TRUNC (Above, 0)
66.666	67	66
33.33	33	33
55.5	56	55

E.g.
```
Select Above, ROUND (Above, -1), TRUNC (Above, -1) from MATH;
```

Oracle responds:

ABOVE	ROUND (Above, -1)	TRUNC (Above, -1)
66.666	70	60
33.33	30	30

55.5 60 50

(iv) CEIL, FLOOR:

CEIL produces the smallest integer (or whole number) that is greater than or equal to a specific value. Pay special attention to its effect on negative numbers. Syntax:

CEIL (*value*)

E.g. CEIL (2) = 2 CEIL (1.3) = 2 CEIL (-2) = -2 CEIL (-2.3) = -2

FLOOR is intuitive opposite of CEIL. Syntax:

FLOOR (*value*)

E.g. FLOOR (2) = 2 FLOOR (1.3) = 1 FLOOR (-2) = -2 FLOOR (-2.3) = -3

Note: Here is some difference in ROUND, TRUNC, CEIL, FLOOR functions:

E.g.
ROUND (55.5) = 56 ROUND (-55.5) = -56
TRUNC (55.5) = 55 TRUNC (-55.5) = - 55
CEIL (55.5) = 56 ROUND (-55.5) = -55
FLOOR (55.5) = 55 FLOOR (-55.5) = -56

(v) ABS, SIGN:

Absolute value is the measure of the magnitude of something. It is always a positive number. For example, the magnitude of change in temperature or stock index. Syntax:

ABS (*value*)

E.g. ABS (146) = 146 ABS (-30) = 30

SIGN is the flip side of value. Whereas ABS tells the magnitude of a value but not the its sign, SIGN tells you the sign of a value but not the its magnitude. Syntax:

SIGN (*value*)

E.g.
SIGN (146) = 1 Compare to: ABS (146) = 146
SIGN (-30) = -1 Compare to: ABS (-30) = 30
SIGN (0) =0

(vi) POWER, SQRT, LOG:

Power is ability to raise a value to a given positive exponent. Syntax:

POWER (*value*, *exponent*)

E.g. POWER (3, 2) = 9 POWER (3, 3) = 27 POWER (64, .5) = .8

SQRT (Square Root) gives the result equivalent to POWER (64, .5). Oracle gives an error while *value* is a negative number. Syntax:

SQRT (*value*)

E.g. SQRT (64) = 8 SQRT (9) = 3 SQRT (66.666) = 8.16492

LOG is rarely used in business calculations. It is used in scientific / technical fields. Syntax:

LOG (*value*)

E.g. LOG (EXP (1), 3) = 1.098612 LOG (10, 100) = 2

(vii) MOD:

MOD divides a value by a divisor and tells you the remainder. Both value and divisor can be any real number. If divisor is zero or a negative the value of MOD is zero. Syntax:

MOD (*value*, *divisor*)

E.g. MOD (23, 6) = 5 MOD (22, 23) = 22 MOD (-30.23, 7) = -2.23
MOD (4.1, .3) = .2 MOD (*value*, 1) = 0 (this confirms value if it is an integer)

b) Group-Value Functions:

These statistical functions tell something about a group of values as a whole.

(i) MIN (ii) MAX (iii) COUNT (iv) SUM (v) AVG
(vi) STEDDEV, VARIANCE

E.g.
```
Select MIN (Noon), MAX (Noon), COUNT (Noon), SUM (Noon), AVG (Noon)
From COMFORT
Where City = 'SAN FRANCISCO';
```

Oracle responds:

MIN (Noon)	MAX (Noon)	COUNT (Noon)	SUM (Noon)	AVG (Noon)
51.1	62.5	3	166.2	55.4

Combining Group-Value Functions and Single-Value Functions:

E.g.
```
Select AVG (Noon-Midnight) from COMFORT
Where City = 'KEENE';
```

Oracle responds:

AVG (NOON-MIDNNIGHT)
17.68

E.g.
```
Select AVG (ABS (Noon-Midnight)) from COMFORT
Where City = 'KEENE';
```

Oracle responds:

AVG (ABS (NOON-MIDNIGHT))
20.68

E.g.
```
Select MAX (Noon) – MIN (Noon)) from COMFORT
Where City = 'SAN FRANCISCO';
```

Oracle responds:

MAX (Noon) – MIN (Noon)
11.4

E.g.
```
Select City, AVG (Noon), MAX (Noon), MIN (Noon),
       MAX (Noon) – MIN (Noon) As Swing from COMFORT
Group by City;
```

Oracle responds:

City	AVG (Noon)	MAX (Noon)	MIN (Noon)	SWING
Keene	54.4	99.8	-7.2	107
San Francisco	55.4	62.5	51.1	11.4

(vi) STEDDEV, VARIANCE:

Standard deviation and variance have their own statistical meanings. Syntax:

STEDDEV (*value*) VARIANCE (*value*)

E.g.
```
Select AVG (Noon), MAX (Noon), MIN (Noon), STEDDEV (Noon)
       VARIANCE (Noon) from COMFORT
Where City = 'KEENE';
```

Oracle responds:

AVG (Noon)	MAX (Noon)	MIN (Noon)	STEDDEV (Noon)	VARIANCE (Noon)
54.4	99.8	-7.2	48.33	2336

DISTINCT in Group-Functions:

All group-functions have a DISTINCT versus ALL option. All is default, and works on all rows in the group. But DISTINCT forces functions to work on just unique rows. Syntax:

FUNCTION ([DISTINCT | ALL] *value*)

E.g. Select COUNT (DISTINCT City), COUNT (City), COUNT (*)
From COMFORT;

Oracle responds:

COUNT (DISTINCT City)	COUNT (City)	COUNT (*)
2	8	8

E.g. Select AVG (DISTINCT Age) As Average, SUM (DISTINCT Age) As Total
From BIRTHDAY;

Oracle responds:

AVERAGE	TOTAL
47	94

*If **COUNT** has asterisk value, it can COUNT all rows whether they are **NULL**. Even if all fields are NULL.*

c) List Functions:

List functions work on a group of columns, either actual or calculated values, within a single row. These are List Functions and their Syntax:

(i) GREATEST, LEAST

GREATEST (*value1, value2, value3……*)
LEAST (*value1, value2, value3……*)

E.g. Select City, SampleDate, GREATEST (Midnight, Noon) As High,
LEAST (Midnight, Noon) As Low
From COMFORT;

Oracle responds:

CITY	SAMPLEDATE	HIGH	LOW
San Francisco	21-MAR-99	62.5	42.3
San Francisco	23-SEP-99		
Keene	21-MAR-99	39.9	-1.2

These both functions can be used with many values, and the values can be columns, literal numbers, calculations, character columns or combinations of other columns.

E.g. GREATEST ('Bob', 'George', 'Andrew', 'Isaiah') = Isaiah
LEAST ('Bob', 'George', 'Andrew', 'Isaiah') = Andrew

Finding Rows with MAX or MIN:

E.g. Select City, SampleDate, MAX (Noon) from COMFORT;

This query generates an error because City and SampleDate columns have more than rows but MAX function will result only once. So, it can't be used here. The correct query is this:

E.g. Select City, SampleDate, Noon from COMFORT
Where Noon = (select MAX (Noon) from COMFORT);

Oracle responds:

CITY	SAMPLEDATE	NOON
KEENE	23-SEP-99	99.8

E.g. Select City, SampleDate, Midnight from COMFORT

Where Midnight = (select MIN (Midnight) from COMFORT);

Oracle responds:

CITY	SAMPLEDATE	MIDNIGHT
KEENE	21-MAR-99	-1.2
KEENE	22-DEC-99	-1.2

E.g. Select City, SampleDate, Noon from COMFORT
Where Noon = (select MAX (Noon) from COMFORT)
or Noon = (select MIN (Noon) from COMFORT);

Oracle responds:

CITY	SAMPLEDATE	NOON
KEENE	23-SEP-99	99.8
KEENE	22-DEC-99	-7.2

Precedence and Parentheses:

Be careful while using more than one arithmetic and logical operators in a single calculation, because first executing operator can change completely the answer.

E.g. Select 2/2/4 from DUAL;

Oracle responds:

2/2/4
.25

E.g. Select 2 / (2 / 4) from DUAL;

Oracle responds:

2/2/4
4

E.g. Select * from NEWSPAPER
Where Section = 'B' AND Page = 1 OR Page = 2;

Oracle responds:

FEATURE	S	PAGE
Weather	C	2
Modern Life	B	1
Bridge	B	2

E.g. Select * from NEWSPAPER
Where Page = 1 OR Page = 2 AND Section = 'B';

Oracle responds:

FEATURE	S	PAGE
National News	A	1
Sports	D	1
Business	E	1
Modern Life	B	1
Bridge	B	2

E.g. Select * from NEWSPAPER
Where Section = 'B' AND (Page = 1 OR Page = 2);

Oracle responds:

FEATURE	S	PAGE
Modern Life	B	1
Bridge	B	2

Summary of Number Functions:

Single-value functions work on values in a row-by-row fashion. List functions compare columns and choose just one, again in row-by-row fashion. Single-value functions almost always change the value of the column they are applied to. This doesn't mean that they modified the database,

from which the value was drawn, but they do make a calculation with that value, and the result is different than the original value. List functions don't change values in this way; rather they simply choose (or report) the GREATEST or LEAST of a series of values in a row. Both single-value and list functions will not produce a result if they encounter a value that is NULL and these can be used anywhere an expression can be used, such as in the **select** and **where** clauses. Group-value functions tell something about a whole group of numbers, all of the rows in a set. These ignore NULL values, and this fact must be in mind when reporting about groups of values, otherwise there is considerable risk of misunderstanding the data. Finally get the habit of using parentheses instead of simple expressions otherwise you will have unwanted results of your queries.

10. Working with Dates and Date Functions:

Oracle has DATE datatype for storing dates. This can store month, day, year, hour, minutes and second. The followings are date function:

(i)	SYSDATE	(ii)	ADD_MONTH	(iii)	GREATEST, LEAST
(iv)	NEXT_DAY	(v)	LAST_DAY	(vi)	MONTHS_BETWEEN
(vii)	ROUND, TRUNC	(viii)	NEW_TIME	(ix)	TO_CHAR, TO_DATE

(i) SYSDATE:

It is a function that gets the *current* date and time from Operating System.

E.g. Select SysDate from DUAL;

Oracle responds:

SYSDATE
13-May-04

Using Arithmetic Functions with dates (The Difference between two Dates):

E.g. Select Holiday, ActualDate, CelebratedDate from HOLIDAY
Where CelebratedDate – ActualDate != 0;

Oracle responds:

HOLIDAY	ACTUALDATE	CELEBRATEDATE
Lincolns Birthday	12-FEB-00	21-FEB-00
Memorial Day	30-MAY-00	29-MAY-00
Columbus Day	08-OCT-00	09-OCT-00

(ii) Adding Months, Subtracting Months:

Adds *count* months to *date*. Syntax:

ADD_MONTHS (*date, count*)

E.g. Select ADD_MONTH (CelebratedDate, 6) As FeastDay from HOLIDAY
Where Holiday like 'FAST%';

Oracle responds:

FEASTDAY	
22-AUG-00	*(Actual day was 22-FEB-00)*

E.g. Select ADD_MONTH (CelebratedDate, -6) -1 As LastDay from HOLIDAY
Where Holiday like 'COLUMBUS DAY';

Oracle responds:

LASTDAY	
08-APR-00	*(Actual day was 08-OCT-00)*

(iii) GREATES, LEAST:

LEAST finds the earliest and GREATEST finds the latest date. Syntax:

GREATEST (*date1, date2, date3…)* LEAST (*date1, date2, date3…)*

E.g. Select LEAST (ActualDate, CelebratedDate) As First, ActualDate, CelebratedDate
From HOLIDAY
Where ActualDate – CelebratedDate != 0;

Oracle responds:

FIRST	ACTUALDATE	CELEBRATEDDATE
15-JAN-00	15-JAN-00	17-JAN-00
12-FEB-00	12-FEB-00	21-FEB-00
21-FEB-00	22-FEB-00	21-FEB-00

E.g. Select LEAST ('20-JAN-00', '20-DEC-00') As First from DUAL;

Oracle responds:

FIRST
20-DEC-00

Note: This result is wrong. Oracle will not consider *literal strings* in single quotes as dates; rather these are treated as if they are strings. TO_DATE function converts strings to date.

E.g. Select GREATEST (TO_DATE ('20-JAN-00'), TO_DATE ('20-DEC-00')) As First
from DUAL;

Oracle responds:

FIRST
20-DEC-00

E.g. Select Holiday, CelebratedDate from HOLIDAY
Where CelebratedDate = LEAST (TO_DATE ('17-JAN-00'),
TO_DATE ('04-SEP-00'));

Oracle responds:

HOLIDAY	CELEBRAT
Martin Luther King, JR	17-JAN-00

(iv) NEXT_DAY:

It computes the date of the next named day of the week after given date. Syntax:

NEXT_DAY (*date, 'day'*)

E.g. Select NEXT_DAY ('15-May-04', 'SUNDAY') As REST_DAY,
NEXT_DAY ('16-May-04', 'SUNDAY') As REST
From DUAL;

Oracle responds:

REST_DAY	REST
16-MAY-04	23-MAY-04

Note: This function works as greater than function. It finds the *day* after the given *date*. If the given *date* is the required day, the result will be wrong. For this situation type this:

E.g. Select NEXT_DAY (TO_DATE ('16-May-04') -1, 'SUNDAY') from DUAL;

Oracle responds:

NEXT_DAY (
16-MAY-04

(v) LAST_DAY:

It produces the date of the last day of the month. Syntax:

LAST_DAY (*date*)

E.g. Select LAST_DAY (TO_DATE ('15-May-04')) As EndMonth from DUAL;

Oracle responds:

ENDMONTH

31-MAY-04

(vi) MONTHS_BETWEEN:

It gives differences between two dates in months (*date1 – date2*). Syntax:

MONTHS_BETWEEN (*date1, date2*)

E.g. Select MONTHS_BETWEEN (SysDate, TO_DATE ('15-DEC-1979')) As Months From DUAL;

Oracle responds:

MONTHS
292.9975

By dividing months with 12 will give years. But if it results in decimal use FLOOR ().

Combining Date Functions:

E.g. Select SysDate Today, LAST_DAY (ADD_MONTHS (SysDate, 6)) +1 Review From DUAL;

Oracle responds:

TODAY	REVIEW
16-MAR-00	01-OCT-00

ADD_MONTHS takes the SysDate and adds six months to it. LAST_DAY takes this result and figures the last day of that month. Then add 1 to the date to get the first day of the next month. Note the use of parentheses to assure the proper order of the calculation:

E.g. Select (LAST_DAY (ADD_MONTHS (SysDate, 6)) +1) -SysDate Wait From DUAL;

Oracle responds:

WAIT
199

(vii) ROUND, TRUNC:

ROUND (*date, 'format'*) TRUNC (*date, 'format'*)

E.g. Select TO_DATE ('17-MAR-00') – SysDate As Decimals from DUAL;

Oracle responds:

DECIMALS
.516

Note: The reason for the fractional number of days is that Oracle keeps hours, minutes and seconds with its dates, and SysDate is always current, up to second.

E.g. select TO_DATE ('17-MAR-00') – ROUND (SysDate) As Decimals from DUAL;

Oracle responds:

DECIMALS
1

E.g. Select TRUNC (((SYSDATE) – TO_DATE ('15-JUN-1979')) /365) from DUAL;

Oracle responds:

TRUNC (((
24 (Actual value 24.934247)

Some Rules about ROUND and TRUNC:

- A date entered as a literal, such as '17-MAR-00', is given a default time of 12 A.M.
- A date entered through SQLPLUS, unless a time is specially assigned to it, is set to 12 A.M. at the beginning of that day.
- **SYSDATE** includes both the date and time, unless you intentionally round it off. The **ROUND** function on any date sets it to 12 A.M. of that day if the time is before exactly noon,

and to 12 A.M. the next day if it is after noon. The **TRUNC** function acts similarly, except that it sets the time to 12 A.M. for any time up to and including one second before midnight.

(viii) NEW_TIME (Switching Time Zones):

This function gives the time and date of other zones. Syntax:

NEW_TIME (*date*, *'this'*, *'other'*)

E.g. Select BirthDate, NEW_TIME (BirthDate, 'EST', 'HST') from BIRTHDAY
Where FirstName = 'VICTORIA';

Oracle responds:

BIRTHDATE	NEW_TIME (
20-MAY-49	19-MAY-49

These both are two different dates. Using TO_CHAR can solve the problem:

E.g. Select TO_CHAR (BirthDate, ' fmMonth Ddspth, YYYY "at" HH:MI A.M.')
As Birth from BIRTHDAY
Where FirstName = 'VICTORIA';

Oracle responds:

BIRTH	NEW_TIME (
May 20th, 1949 at 3:27 A.M.	May 19th, 1949 at 10:27 P.M.

(ix) TO_CHAR, TO_DATE:

These both are somewhat similar powerful formatting capabilities. TO_DATE converts a character string or a number into an Oracle date, whereas TO_CHAR converts an Oracle date into a character string. Syntax:

TO_CHAR (*date* [, '*format'* [, '*NLSparameters'*]])
TO_CHAR (*string* [, '*format'* [, '*NLSparameters'*]])

Common TO_DATE and TO_CHAR Formats:

These date formats are used with both TO_CHAR and TO_DATE:

MM	Number of months: 12	RM	Roman numeral month: XII
MON	Three-letter abbreviation of month: AUG	Month	Month fully spelled out: AUGUST
DDD	Number of days in year, since Jan 1:354	DD	Number of days in month: 23
D	Number of days in week: 6	DY	Three-letter abbreviation of day: FRI
DAY	Day fully spelled out: FRIDAY	YYYY	Full four-digit year: 1946
Y,YYY	Year, with comma	YYY	Last three digits of year: 946
YY	Last two digits of year: 46	Y	Last one digit of year: 6
YEAR	Year Spelled out: Nineteen-Forty-Six	WW	Number of weeks in year: 46
W	Number of weeks in month	HH	Hours of Day, 1 to 12: 11
HH24	Hours of Day, 1 to 24: 18	MI	Minutes of hour: 50
SS	Seconds of minutes: 43	A.M.	Display time of day: A.M
P.M.	Display time of Night: P.M	AM, PM	Without comma

TO_CHAR Special Formats:

Case matter is taken from the original keyword, not from the suffix or prefix function:

FM	Eliminates extra spaces from days or months: fmMonth, fmDay	TH	Suffix TH with numbers or Dates: DDTH = 24TH DdTH, Ddth = 24Th
SP	To spell out a number: DDSP = THREE, DdSP = Three, ddSP = three	SPTH, THSP	To spell out a number and add ordinal suffix. DdSPTH = Third, Fourth, second

TO_CHAR Function:

It changes a column's formatting. You can change dash (-) with any punctuation or change the format of a column.

E.g.
```
Select SysDate, TO_CHAR (SysDate, 'MM/DD/YY') As Formatted,
                TO_CHAR (SysDate, 'MON~DD~YY') As Formatting,
                TO_CHAR (SysDate, 'MMDDYY') As NOSAPCE
From DUAL;
```

Oracle responds:

SYSDATE	FORMATTED	FORMATTING	NOSPACE
20-MAY-04	05/20/04	MAY~20~04	052004

Common Month, Day and Year formats:

(i) Month = August (ii) month = august
(iii) Mon = aug (iv) mon = aug

(v) DDth or DDTH = 11TH (vi) Ddth or DdTH = 11Th
(vii) ddth or ddTH = 11th

(viii) yyyy or Yyyy or yyyY or yYYy or yYYY = 2004

The words between double quotation marks are inserted as it is.

E.g.
```
Select SysDate, TO_CHAR (SysDate, 'Month, DDth "in," YyyY') As Formatted
From DUAL;
```

Oracle responds:

SYSDATE	FORMATTED
20-MAY-04	May, 12TH in, 2004

Eliminating Spaces from Date Formats:

Oracle inserts aligns justifiably month names and dates. Use the following to solve this:

(i) Month, ddth = August , 20th (ii) fmMonth, ddth = August, 20th
(iii) Day, ddth = Monday , 20th (iv) fmDay, ddth = Monday, 20th

E.g.
```
Select SysDate, TO_CHAR (SysDate, 'fmMonth, DDth, YyyY') Formatted,
TO_CHAR (SysDate, '"Today, " fmMonth DDth, YYYY, "I met her"') Formats,
From DUAL;
```

Oracle responds:

SYSDATE	FORMATTED	FORMATS
20-MAY-04	May, 12TH, 2004	Today, May 20th 2004 I met her

Oracle can spell out the date.

E.g.
```
Select SysDate, TO_CHAR (SysDate, '"Today, " Ddspth "of" fmMonth YYYY, "at"
HH: MI P.M. "I met her "') Formatted from DUAL;
```

Oracle responds:

SYSDATE	FORMATTED
20-MAY-04	Today, Twentieth of May 2004 at 9:55 P.M. I met her

Here **sp** control is to spell out the date, HH:MI for time and P.M. for day or night.

The Most Common TO_CHAR Error:

Month and minutes can keyword may be misused at each other place.

E.g.
```
Select TO_CHAR (SysDate, 'HH:MI:SS') Now,
       TO_CHAR (SysDate, 'HH:MM:SS') NOWWRONG from DUAL;
```

Oracle responds:

NOW	NOWWRONG

10:28:15 10:05:15

Warning: in the NOWWRONG column month (05) is selected in the minutes place.

TO_DATE Calculations:

(i) **TO_DATE** follows the same formatting conventions as **TO_CHAR,** with some restrictions. The purpose of **TO_DATE** is to turn a literal string (May 20, 1947) into an Oracle date format.

Syntax: TO_DATE (*string* [, *'format'*)]

To put 22-FEB-00 into Oracle date format, use this:

E.g. Select TO_DATE ('22-FEB-00', 'DD-MON-YY') from DUAL;

Oracle responds: **TO_DATE ('**
22-FEB-00

(ii) The date is in the default Oracle date format. When a literal string has a date in this format, the *format* in the TO_DATE can be left out. But if you want to specify century, use the *format* option in TO_DATE. This query will work same as the above:

E.g. Select TO_DATE ('22-FEB-00') from DUAL;

Oracle responds: **TO_DATE ('**
22-FEB-00

E.g. Select TO_DATE ('02/22/00') from DUAL;

This query will produce error "Not a valid Month". So, use this:

E.g. Select TO_DATE ('02/22/00', 'MM/DD/YY') from DUAL;

Oracle responds: **TO_DATE ('**
22-FEB-00

(iii) If you want to know the day of the week of February 22, TO_CHAR function will not work, even with the literal string in the proper format, because TO_CHAR requires a date:

E.g. Select TO_CHAR ('22-FEB-00', 'Day') from DUAL;

Oracle responds: ORA-01722: invalid number

(iv) It will work if you first convert the string to a date. Do this by combining the two functions TO_CHAR and TO_DATE:

E.g. Select TO_CHAR (TO_DATE ('22-FEB-00'), 'Day') from DUAL;

Oracle responds: **TO_CHAR (TO_DATE (**
Tuesday

(v) TO_DATE can also accept numbers, without single quotation marks, instead of strings, as long as they are formatted consistently. Look here:

E.g. Select TO_DATE (11051946, 'MM DD YYYY') from DUAL;

Oracle responds: **TO_DATE (1**
05-NOV-46

(v) The punctuation in the format is ignored, but the number must follow the order of the format controls. The number itself must not have punctuation. What a complex format:

E.g. Select TO_DATE ('Baby Girl on the Twentieth of May, 1949, at 3:27 A.M.',
'"Baby Girl on the " Ddspth "of" fmMonth, YYYY, "at" HH:MI P.M.')

As formatted from BIRTHDAY
Where FirstName = 'VICTORIA';

Oracle responds: ERROR: a non-numeric character found where a numeric was expected.

E.g.
```
Select TO_DATE ('August 20, 1949, 3:27 A.M> ', Month Dm, YYYY, HH:MI P.M.')
        As Formatted from HOLIDAY
Where FirstName = 'VICTORIA';
```

Oracle responds:

FORMATTED
20-AUG-49

(vii) Some restrictions on ***format*** that govern TO_DATE:

a) No literal strings are allowed, such as "Baby Girl on the"
b) Days cannot be spelled out. They must be numbers.
c) Punctuation is permitted.
d) fm is not necessary. If used, it is good.
e) If Month is used, the month in the string must be spelled out. If Mon is used, the month must be a three-letter abbreviation. Spelling case is ignored.

Where Clauses with Dates and Operators:

Dates can be used in where clauses with arithmetic operators and other logical operators, with some warning and restrictions.

E.g.
```
Select Holiday, CelebratedDate from HOLIDAY
Where CelebratedDate BETWEEN TO_DATE ('01-JAN-2000', 'DD-MON-YYYY')
        And TO_DATE ('22-FEB-2000', 'DD-MON-YYYY');
```

Oracle responds:

HOLIDAY	CELEBRATEDDATE
New Year Day	01-JAN-00
Martin Luther King, JR.	17-JAN-00
Lincoln Birthday	21-FEB-00
Washington Birthday	21-FEB-00
Fast Day, New Hampshire	22-FEB-00

E.g.
```
Select Holiday, CelebratedDate from HOLIDAY
Where CelebratedDate IN TO_DATE ('01-JAN-00', '22-FEB-00');
```

Oracle responds:

HOLIDAY	CELEBRATEDDATE
New Year Day	01-JAN-00
Fast Day, New Hampshire	22-FEB-00

If you want to specify century by yourself:

E.g.
```
Select Holiday, CelebratedDate from HOLIDAY
Where CelebratedDate IN (TO_DATE ('01-JAN-2000', 'DD-MON-YYYY'),
                         TO_DATE ('22-FEB-2000', 'DD-MON-YYYY'));
```

Oracle responds:

HOLIDAY	CELEBRATEDDATE
New Year Day	01-JAN-00
Fast Day, New Hampshire	22-FEB-00

Note: LEAST and GREATEST will not work in where clause. Because they assume the literal strings are *strings*, not *dates*.

Dealing with Multiple Centuries:

If your applications use only two-digit values for years, you may encounter problems related to the year 2000. If you only specify the last two digits of the year value, then Oracle, by default, use the current century as the century value when it inserts a record.

E.g.
```
Insert into BIRTHDAY (FirstName, LastName, BirthDate)
    Values ('ALICIA', 'ANN', '21-NOV-39');
```

E.g.
```
Select TO_CHAR (BirthDate, 'DD-MON-YYYY') as Bday
From BIRTHDAY
Where FirstName = 'ALICIA' and LastName = 'ANN';
```

Oracle responds:

BDAY
21-NOV-2039

The date of birth is far even from the current date. So, always write four-digit for year.

11. Conversion and Transformation Functions:

This topic looks at functions that convert or transform, one datatype into another. List of functions:

String Functions for CHAR & VARCHAR2 Datatypes:

(i) | | (Concatenation) (ii) ASCII (iii) CHR (iv) CONCAT (v) CONVERT
(vi) INITCAP (vii) INSTR (viii) INSTRB (ix) LENGTH (x) LENGTHB
(xi) LOWER (xii) LPAD (xiii) LTRIM (xiv) NLS_INITCAP (xv) NLS_LOWER
(xvi) NLS_UPPER (xvii) NLSSORT (xviii) REPLACE (xix) RPAD (xx) RTRIM
(xxi) SOUNDEX (xxii) SUBSTR (xxiii) SUBSTRB (xxiv) TRANSLATE
(xxv) UID (xxvi) UPPER (xxvii) USER (xxviii) USERENV

Arithmetic Functions for NUMBER datatypes:

(i) + - * / (ii) ABS (iii) ACOS (iv) ASIN (v) ATAB (vi) ATAN2
(vii) CEIL (viii) COS (ix) COSH (x) EXP (xi) FLOOR (xii) LN
(xiii) LOG (xiv) MOD (xv) POWER (xvi) ROUND (xvii) SIGN
(xviii) SIN (xix) SINH (xx) SQRT (xxi) TAN (xxii) TANH (xxiii) TRUNC

Date Functions For DATE Datatype:

(i) ADD_MONTHS (ii) LAST_DAY (iii) MONTHS_BETWEEN
(iv) NEW_TIME (v) NEXT_DAY (vi) ROUND (vii) TRUNC

Group Functions:

(i) AVG (ii) COUNT (iii) CUBE (iv) GLB (v) LUB (vi) MAX
(vii) MIN (viii) ROLLUP (ix) STEDDEV (x) SUM

Conversion Functions:

(i) CHARTOROWID (ii) CONVERT (iii) HEXTORAW (iv) RAWTOHEX
(v) RAWTOCHAR (vi) TO_CHAR (vii) TO_DATE (viii) TO_LOB
(ix) TO_MULTI_BYTE (x) TO_NUMBER

Miscellaneous Functions:

(i) DECODE (ii) DUMP (iii) GREATEST (iv) GREATEST_LB
(v) LEAST (vi) LEAST_UB (vii) NVL (viii) VSIZE

Elementary Conversion Functions:

There are three elementary Oracle function whose purpose is to convert one datatype into another.

- **TO_CHAR** transforms a DATE or NUMBER into a character string.

- **TO_DATE** transforms a NUMBER, CHAR, or VARCHAR2 into DATE.
- **TO_NUMBER** transforms a CHAR or VARCHAR2 into a NUMBER.

E.g. Select SUBSTR (TO_CHAR (948033515), 1, 5) | | '-' | |
SUBSTR (TO_CHAR (948033515), 6) As Zip from DUAL;

Oracle responds:

ZIP
94803-3515

E.g. Select SUBSTR (Zip, 1, 5) | | '-' | | SUBSTR (Zip, 6) As Zip from ADDRESS
Where FirstName = 'MARY';

Oracle responds:

ZIP
94941-4302
60126-2460

E.g. Select Zip, RTRIM (Zip, 20) from ADDRESS
Where FirstName = 'MARY';

Oracle responds:

ZIP	RTRIM (ZIP, 20)	
949414302	9494143	// it has removed 2s and 0s from right
601262460	60126246	

Automatic Conversion of Datatypes:

Oracle is automatically converting these numbers, both the Zip and the 20, into strings, as if they both had **TO_CHAR** functions in front of them. In fact, with a fewer exceptions, Oracle will automatically transform any datatype into any other datatype, based on the function that is going to affect it.

Guidelines for Automatic Conversion of Datatypes:

These guidelines describe the automatic conversion of data from one type to another, based on the function that will use the data:

- Any NUMBER or DATE can be converted into a character string. As a consequence, *any* string function can be used on a NUMBER or DATE column. Literal NUMBERs do not have to be enclosed in single quotation marks when used in a string function; literal DATEs do.
- A CHAR or VARCHAR2 will be converted to a NUMBER if it contains only numbers, a decimal point, or a minus sign on the left. There must be no embedded spaces or other characters.
- A CHAR or VARCHAR2 will be converted to a DATE if it is in the format DD-MON-YY, such as 07-AUG-95. This is true for all functions except **GREATEST** and **LEAST**, which will treat it as a string, and is true for **BETWEEN** only if the column to the left after the word **BETWEEN** is a DATE. Otherwise, **TO_DATE** must be used, with a proper format.
- A DATE will not be converted to a NUMBER. A NUMBER will not be converted to a DATE.

NOTE: A simple rule of thumb might be that it is best to use functions where the risk is low, such as string manipulation on numbers, rather than arithmetic functions on string.

E.g. Select INITCAP (LOWER (SysDate)), SUBSTR (SysDate, 4, 3) from DUAL;

Oracle responds:

INITCAP (LO	SUBSTR (SYSDA
01-Nov-99	NOV

E.g. Select LPAD (SysDate, 20, '9') As DATES, LPAD (9, 20, 0) As DIGITS from DUAL;

Oracle responds:

DATES	DIGITS
9999999999901-NOV-99	00000000000000000009

These examples show how string functions treat both NUMBERs and DATEs as if they were character strings. In the next examples, strings are treated as NUMBER and DATE.

E.g. Select FLOOR ('-323.78') As DECIMALS from DUAL;

Oracle responds:

DECIMALS
-324

E.g. Select MONTHS_BETWEEN ('16-MAY-99', '01-NOV-99') MONTHS from DUAL;

Oracle responds:

MONTHS
-5.516129

E.g. Select SysDate, SysDate + 1, SysDate -1 from DUAL;

Oracle responds:

SYSDATE	SYSDATE + 1	SYSDATE - 1
01-NOV-99	02-NOV-99	31-OCT-99

E.g. Select '01-NOV-99' + 1 from DUAL;

Oracle responds: ERROR at line 1: ORA-01722: invalid number

E.g. Select SysDate / 2 from DUAL;

Oracle responds: ERROR: ORA-00932: inconsistent data type

E.g. Select NEXT_DAY (110199, 'FRIDAY') from DUAL;

Oracle responds: ERROR at line 1: ORA-00932: inconsistent datatypes

Transformation Functions:

(i) TRANSLATE (ii) DECODE

(i) TRANSLATE:

It is a simple function that does an orderly character-by-character substitution in a string. Syntax:

TRANSLATE (*string*, *if*, *then*)

TRANSLATE looks at each character in *string*, and then checks *if* to see whether that character is there. If it is, it notes the position in *if* where it found the character, and looks at the same position in *then*. **TRANSLATE** substitutes whichever character it finds there for the character in *string*. Normally, the function is written on a single line.

E.g. Select TRANSLATE (7671234, 234567890, 'BCDEFGHIJ') Translation from DUAL;

Oracle responds:

TRANSLATION
GFG1BCD

When **TRANSLATE** sees a 7 in the *string*, it looks for a 7 in the *if*, and translates it to the character in the same position in the *then* (an uppercase G). If the character isn't in the *if*, it isn't translated (observe what **TRANSLATE** did with the 1). **TRANSLATE** is technically a string function, but it will do automatic data conversion and work with a mix of strings and numbers. It is like encoding.

E.g. Select TRANSLATE (HAL, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'BCDEFGHIJKLMNOPQRSTUVWXYZA') As Who
From DUAL;

Oracle responds:

WHO
IBM

(ii) DECODE:

If **TRANSLATE** is a character-by-character substitution, **DECODE** cab be considered a value-by-value substitution. For every value it sees in a field, **DECODE** checks for a match in a series of *if, then* tests. A complete chapter is devoted for **DECODE** later. Syntax:

DECODE (*value if1*, *then1*, *if2*, *then2*, *if3*, *then3*, …… *else*)

E.g. Select Feature, Section, DECODE (Page, '1', 'Front Page', 'Turn to' | | page) Decoding
From NEWSPAPER;

Oracle responds:

FEATURE	S	DEODING	
National News	A	Front Page	
Sports	D	Front Page	
Editorials	A	Turn to 12	// All Features detail

12. Grouping Things Together:

The use of group by and having:

Beyond group-functions there are also two group clauses: **having, group by.** These are same as **where** and **order by** clauses, except that they work on groups, not on individual rows.

E.g. Select Item, SUM (Amount) Total, COUNT (Item) from LEDGER
Where Action = 'PAID'
Group by Item;

Oracle responds:

ITEM	TOTAL	COUNT (ITEM)	
Cut Logs	.25	1	
Discus	1.50	1	
Work	27.98	26	
Sawing	3.50	4	// (all matching results)

Note: Notice the mix of a column name, Item, and two group functions, **SUM,** and **Count,** in the **select** clause. This mix is possible only because Item is referenced in the **group by** clause. Otherwise it will be the following and generate the error:

E.g. Select Item, SUM (Amount) Total, COUNT (Item)

Oracle responds: Error at line 1: ORA-00937: not a single-group group function

The **having** clause works very much like a **where** clause except that its logic is only related to the results of group functions, as opposed to columns or expressions for individual rows.

E.g. Select Item, SUM (Amount) Total from LEDGER
Where Action = 'PAID'
Group by Item
Having SUM (Amount) > 3;

Oracle responds:

ITEM	TOTAL	
Plowing	18.10	
Sawing	3.50	
Work	27.98	// (all matching results)

E.g. Select AVG (Amount) Average from LEDGER
Where Action = 'PAID';

Oracle responds:

AVERAGE
1.29

E.g.
```
Select Item, SUM (Amount) Total, AVG (Amount) Average from LEDGER
Where Action = 'PAID'
Group by Item
Having AVG (Amount) > (Select AVG (Amount) from LEDGER
                       Where Action = 'PAID');
```

Oracle responds:

ITEM	TOTAL	AVERAGE	
Digging of Grave	3.00	3.00	
Discus	1.50	1.50	
Painting	1.75	1.75	
Plowing	18.10	18.10	// (all matching results)

E.g.
```
Select Person, SUM (Amount) Total from LEDGER
Where Action = 'PAID'
Group by Person;
```

Oracle responds:

PERSON	TOTAL	
ADAH Talbot	1.00	
Andrew Dye	4.50	// (results all paid persons)

E.g. group by SUM (Amount) // this can't be used

Note: The purpose of **group by** is not to produce a desired sequence, but rather to collect "like" things together.

Adding an order by:

To manage the sorting in the ascending order, **order by** clause is used. For the descending order, **order by (column) DESC** is used.

E.g.
```
Select Person, SUM (Amount) Total from LEDGER
Where Action = 'PAID'
Group by Person
Having SUM (Amount) > 1.00
Order by Total;
```

Oracle responds:

PERSON	TOTAL	
Andrew Dye	4.50	
Dick Jones	6.30	// (all matching results)

E.g.
```
Select Person, SUM (Amount) Total from LEDGER
Where Action = 'PAID'
Group by Person
Having SUM (Amount) > 1.00
Order by Total DESC;
```

Oracle responds:

PERSON	TOTAL	
Dick Jones	6.30	
Andrew Dye	4.50	// (all matching results)

Note: You can use the column alias as part of the **order by** clause, you can't use it as part of the **having** clause. Attempting to use **having Total > 1.00** as a clause in this query will result in an "invalid column name" error.

Order of Execution for different clauses:

a) Choose rows based on the **Where** clause.

b) Group those rows based on the **group by** clause.
c) Calculate the results of the group functions for each group.
d) Choose and eliminate groups based on the **having** clause.
e) Order the groups based on the results of the group functions in the **order by** clause. The **order by** clause must use either a group function or a column specified in the **group by** clause.

Logic in the having Clause:

In the **having** clause, the choice of the group function, and the column on which it operates, might bear no relation to the column or group functions in the **select** clause:

E.g.

```
Select Person, SUM (Amount) Total from LEDGER
Where Action = 'PAID'
Group by Person
Having COUNT (Item) > 1
Order by Total DESC;
```

Oracle responds:

PERSON	TOTAL	
Dick Jones	6.30	
Andrew Dye	4.50	// (all matching results)

The **having** clause selected only those persons (the **group by** collected all the rows into groups by Person) who had more than one Item. The **order by** helps in finding duplicates. First, select columns that you want to be unique, followed by a **COUNT (*)** column. **Group by** the columns you want to be unique, and use the **having** clause to return only those groups having **COUNT (*) >1.** The only records returned will be duplicates. T

E.g.

```
Select Person, COUNT (*) from LEDGER
Group by Person
Having COUNT (*) > 1
Order by Person;
```

Join Columns:

Joining two tables, views or view and table together requires that they have a relationship defined by a common column. If one of the tables or views has just a single row, SQL joins the single row to every row in the other table or view, and no reference to the joining tables needs to be made in the **where** clause of the query. Any attempt to join two tables that has more than one row without specifying the joined columns in the **where** clause will produce what's known as *Cartesian product*, usually a giant result where every row in one table is joined every row in other table. A small 80-row table joined to a small 100-row table in this way would produce 8,000 rows in your display, and few of them would be at all meaningful.

13. Changing Data: Insert, Update and Delete:

This chapter shows how to change the data in a table, for example, how to insert new rows, update the values of columns in rows and delete rows entirely.

a) Insert:

The SQL command **insert** lets you place a row of information directly into a table (or indirectly, through a view).

E.g.

```
Insert into COMFORT
        Values ('Walpole', TO_DATE ('21-MAR-1999', 'DD-MON-YYYY'),
                56.7, 43.8, 0);
```

Oracle responds: 1 row created

Note: **Values** must precede the list of data to be inserted, as string and date in single quotation marks, numbers didn't, commas to separate columns and in order of table columns. To insert a date not in default format, use TO_DATE function, with that format.

E.g.
```
Insert into COMFORT
    Values ('Walpole', TO_DATE ('06/22/1999', 'MM/DD/YYYY'),
        56.7, 43.8, 0);
```

Oracle responds: 1 row created

Inserting a Time:

Inserting dates without time values will produce a default time of midnight. So, if you wish to insert a date with a time use TO_DATE function.

E.g.
```
Insert into COMFORT
Values ('Walpole', TO_DATE ('06/22/1999 1:35', 'MM/DD/YYYY HH24:MI'),
        56.7, 43.8, 0);
```

Oracle responds: 1 row created

Inserting columns out of the order and NULL values:

While inserting columns out of the order of table, list the order of data before **values** keyword. It doesn't affect the actual order of the data. You can also insert NULL values.

E.g.
```
Insert into COMFORT (SampleDate, Precipitation, City, Noon, Midnight)
    Values (TO_DATE ('23-SEP-1999', 'DD-MON-YYYY'),
        NULL, 'Walpole', 86.3, 72.1);
```

Oracle responds: 1 row created

Insert with Select:

You can also insert information that has been selected from other table. **Where** clause sets the criteria for select rows, and **insert** command inserts all retrieved rows, one or more. You can use number; string or date function in the select command; and the resulting values would be inserted into table.

E.g.
```
Insert into COMFORT (SampleDate, Precipitation, City, Noon, Midnight)
    Select TO_DATE ('22-DEC-1999', 'DD-MON-YYYY'), Precipitation,
    'Walpole', Noon, Midnight from COMFORT
    Where City = 'KEENE'
    And SampleDate = TO_DATE ('22-DEC-1999', 'DD-MON-YYYY');
```

Oracle responds: 1 row created

Append in Insert:

Append command inserts information by selecting from other table, after the last row of table. It works speedier than sole **insert** command with select option.

E.g.
```
Insert /*+ Append*/ into Worker (Name)
    Select Name from PROSPECT;
```

b) Delete:

Removing a row or rows from a table requires **delete** command. The **where** clause is essential to removing only the rows you intend. A **where** clause in a **delete**, just as in as **update** or a **select** that is part of an **insert**, may include SubQueries, union, intersects etc. **Delete** without a **where** clause will empty the table completely.

E.g. Delete from COMFORT where City = 'Walpole';

Oracle responds: 1 row deleted

c) Update:

Update requires setting specific values for each column you wish to change, and specifying which row or rows you wish to affect by using a carefully constructed **where** clause. **Where** clause works same here as in **select, insert,** and **delete**.

E.g.
```
Update COMFORT set Precipitation = .5, Midnight = 73.1
Where City = 'Walpole' and
        SampleDate = TO_DATE ('22-DEC-1999', 'DD-MON-YYYY');
```

Oracle responds: 1 row updated

E.g.
```
Update COMFORT set Midnight = Midnight + 1, Noon = Noon + 1
        Where City = 'Walpole';
```

Oracle responds: 1 row updated

Update with Embedded select:

It is possible to set values in an **update** by embedding a **select** statement right in the middle of it. This **select** has its own **where** clause, and the **update** has its own **where** clause.

E.g.
```
Update COMFORT set Midnight =
                        (Select Temperature from WEATHER
                        Where City = 'Manchester')
        Where City = 'Walpole' and
                SampleDate = TO_DATE ('22-DEC-19999', 'DD-MON-YYYY');
```

Oracle responds: 1 row updated

Note: When using SubQueries with **updates**, you must be certain that the SubQuery will return no more than one record for each of the records to be updated: otherwise the **update** will fail. You also can use an embedded **select** to **update** multiple columns at once the columns must be in parentheses and separated by a comma.

E.g.
```
Update COMFORT set (Noon, Midnight) =
                        (Select Humidity, Temperature from WEATHER
                        Where City = 'Manchester')
        Where City = 'Walpole' and
                SampleDate = TO_DATE ('22-DEC-19999', 'DD-MON-YYYY');
```

Oracle responds: 1 row updated

Update with NULL:

You also can **update** a table and set a column equal to **NULL**. This is the sole instance of using the sign with **NULL**, instead of the word "is".

E.g.
```
Update COMFORT set Noon = NULL
        Where City = 'Walpole' and
```

SampleDate = TO_DATE ('22-DEC-19999', 'DD-MON-YYYY');

Oracle responds: 1 row updated

d) Rollback and Commit:

Rollback will reverse all work that is not yet **committ**ed. Use **commit** command to save the work. **Set autocommit on** to save work automatically. **Show autocommit** displays its current setting (on / off). You can specify a number after that value **autocommit** will run. With the shared databases, if changes are not **commit** just you can view your changes, others can't.

E.g. Commit;

Oracle responds: commit complete

E.g. Rollback;

Oracle responds: rollback complete

Implicit commit:

Quit, exit or any **DDL** commands run **commit** implicitly.

Auto Rollback:

If computer shut down due to power failure (or other problems) all the work that is not **commit** automatically **rollback**.

Note: The primary issues with **insert, update** and **delete** are careful construction of **where** clause to affect (or **insert**) only the rows you really wish, and the normal use of SQL functions within these **inserts, updates** and **deletes**. It is extremely important that you exercise caution about committing work before you are certain it is correct. These three commands extend the power of Oracle well beyond simple query, and allow direct manipulation of data.

14. Advanced Use of Functions and Variables:

Functions in Order By:

Functions can be used in an order by to change the sorting sequence.

E.g.
```
select Author from MAGAZINE
Order by SUBSTR (Author, INSTR (Author, ',') + 2);
```

Oracle responds:

AUTHOR
WHITEHEAD, ALFRED
BONHOEFFER, DIETRICH
CHESTERTON, G. K.
RUTH, GEORGE HERMAN
CROOKES, WILLIAM

These Author names are **ordered by** first name, after the comma.

Bar Charts and graphs:

You also can produce simple bar charts and graphs in SQLPLUS, using a mix of **LPAD** and a numeric calculation. First look at the column formatting commands that will be used:

E.g.
```
Column Name format a16
Column Age format 999
Column Graph Heading 'Age|        1       2       3       4       5       6       7-
|…. 0…. 0…. 0…. 0…. 0…. 0…. 0' justify c
```

Column Graph format a35

The dash at the end of the third line tells the **column** command is wrapped onto the next line.

E.g.
```
Select Name, Age, LPAD ('o', ROUND (Age / 2, 0), 'o') As Graph from WORKER
Where Age is NOT NULL
Order by Age;
```

This query creates a horizontal bar chart (with lower *o*) of age by person. Padding also with it. The following query creates a graph (with lower *x*) of age by person. Padding with a space.

E.g.
```
Select Name, Age, LPAD ('x', ROUND (Age / 2, 0), ' ') As Graph from WORKER
Where Age is NOT NULL
Order by Age;
```

Another way to Graph age is by its distribution rather than by person. First, a view is created that puts each age into its decade. So, 15, 16 and 18 become 10; 20 through 29 become 20; 30 through 39 become 30; and so on. The following is the process:

E.g.
```
Create view AGERANGE as
Select TRUNC (Age, -1) As Decade from WORKER;
```

E.g.
```
Column Graph Heading 'Count |          1        1|…. 5…. 0…. 5'- justify c
Column Graph a15
Column People Heading 'Head | Count' format 9999
```

E.g.
```
Select Decade, Count (Decade) As People, LPAD ('o', COUNT (Decade), 'o') As Graph
From AGERANGE
Order by Decade;
```

Because **COUNT** ignores **NULL**s, it couldn't include the number of workers for whom the age is unknown. Use **COUNT (*)** if you want to make a graph for each row in the Decade column.

E.g.
```
Select Decade, Count (*) As People, LPAD ('o', COUNT (*), 'o') As Graph
From AGERANGE
Order by Decade;
```

Using TRANSLATE:

TRANSLATE converts characters in a string into different characters, based on a substitution plan you give it, from *if* to *then*. Syntax:

TRANSLATE (*string*, *if*, *then*)

E.g.
```
Select TRANSLATE ('NOW VOWELS ARE UNDER ATTACK', 'TAEIOU', 'Taeiou')
from DUAL;
```

Oracle responds:

TRANSLATE (NOW VOWELS ARE
NoW VoWeLS aRe uNDeR aTTaCK

Eliminating Characters:

Extending this logic, what happens if the *if* string is TEIOU and the *then* string is only *T*? Checking for the letter *E* (as in the word VOWELS) finds it in position 3 of 'TAEIOU'. There is no position 3 in the *then* string (which is just the letter *T*), so the value in position 3 is nothing. So, *E* is replaced by nothing.

E.g. Select TRANSLATE ('NOW VOWELS ARE UNDER ATTACK', 'TAEIOU', 'T') from DUAL;

Oracle responds:

TRANSLATE (NOW VOW
NW VWLS R NDR TTCK

This feature of **TRANSLATE**, the ability to eliminate characters from a string, can be very useful in cleaning data. Recall the MAGAZINE titles in the string function section:

E.g. Select LTRIM (RTRIM (Titles, '."'), '"') from MAGAZINE;

E.g. Select TRANSLATE (Titles, 'T."', 'T') As Titles from MAGAZINE;

The both queries produce the same results, Titles column without any full stop (.) and comma (,).

Cleaning Up Dollar Sings and Commas:

When cleaning up numeric values from commas, decimal points and dollar signs, you can use **TO_CHAR** or **TRANSLATE** functions.

E.g. Select AmountChar, TRANSLATE (AmountChar, '1, $', '1') Numeric from COMMA;

Oracle responds:

AMOUNTCHAR	NUMERIC	
$123.25	123.25	
$123,456.25	123456.25	
$1,234.25	1234.25	// all records are followed

Why are there always at least one letter or number translated, a 1 here and a *T* in the previous example? Because without at least one real character in the *then* string, **TRANSLATE** produces nothing. Al least one character must be in both the *if* and *then* strings.

E.g. Select AmountChar, TRANSLATE (AmountChar, ', $', '') Numeric from COMMA;

Oracle responds:

AMOUNTCHAR	NUMERIC	
$123.25		
$123,456.25		
$1,234.25		// all records are followed

Put Commas into a Number:

The way to put commas into a number is to use the **TO_CHAR** function.

E.g. Select TO_CHAR (123456789, '999,999,999') CommaTest from DUAL;

Oracle responds:

COMMATEST
123,456,789

Complex Cut And Paste:

Suppose that you have a Name column that includes both First and Last Names. If you want to separate this these names to save in two columns FirstName and LastName. You would do:

E.g.
```
Select SUBSTR (Name, 1, INSTR (Name, ' ')) FirstName,
       SUBSTR (Name, INSTR (Name, ' ') + 1) LastName,
From NAME;
```

Oracle responds:

FIRSTNAME	LASTNAME
HORATIO	NELSON
	VALDO
MARIE	DE MEDICIS
FLAVIUS	JOSEPHUS

EDYTHE P. M. GAMMIERE

If only one value is found for both FirstName and LastName this function doesn't work properly. Look one empty row in FirstName and a mixed name, P.M. GAMMIERE, in LastName.

E.g. Select SUBSTR (Name, 1,
DECODE (INSTR (Name, ' '), 0, 99,INSTR (Name, ' '))) FirstName from NAME;

Oracle responds:

FIRSTNAME
HORATIO
MARIE
FLAVIUS
EDYTHE

E.g. Select SUBSTR (Name, GREATEST (INSTR (Name, ' '), INSTR (Name, ' ', 1, 2),
INSTR (Name, ' ', 1, 3)) + 1) LastName from NAME;

Oracle responds:

LASTNAME
NELSON
VALDO
MEDICIS
GAMMIERE

GREATEST could also have been used similarly in place of **DECODE** in the previous example.

E.g. Select SUBSTR (Name, INSTR (Name, ' ', -1) + 1) LastName from NAME;

Oracle responds:

LASTNAME
NELSON
VALDO
MEDICIS
GAMMIERE

Explanation: The **–1** in the **INSTR** tells it to start its search in the final position and go *backward*, or right to left, in the Name column. When it finds the space, **INSTR** returns its position, counting from the left as usual. The –1 simply makes **INSTR** start searching from the end rather from the beginning. A –2 would make it start searching from the second position from the end, and so on. The **+1** in the **SUBSTR** command has same purpose as in the previous example: once the space is found, **SUBSTR** has to move one position to the right to begin clipping out the Name. If no space is found, the **INSTR** return 0, and **SUBSTR** therefore starts with position 1. That's why VALDO made the list. How do you get rid of VOLDO? Add an ending position to the **SUBSTR** instead of its default (which goes automatically all the way to the end). The ending position is found by using this:

E.g. DECODE (INSTR (Name, ' ', 0), LENGTH (Name))

Counting String Occurrences Within Lager Strings:

The combination of the **LENGTH** and **REPLACE** functions determines how many times a string (such as ABC) occurs within a lager string (such as ABCDEFABC). The **REPLACE** function replaces a character or characters a string zero or more characters.

E.g. Select REPLACE ('ADAH', 'A', 'BLAH') Replacing from DUAL;

Oracle responds:

REPLACING
BLAHDBLAHH // replacing characters with **REPLACE**

E.g. Select REPLACE ('GEORGE', 'GE', NULL) Replacing from DUAL;

Oracle responds:

REPLACING

OR // eliminating characters with **REPLACE**

E.g. Select LENGTH (REPLACE ('GEORGE', 'GE', NULL)) Counting from DUAL;

Oracle responds: **REPLACING**
2 // counting characters occurrences

E.g. Select LENGTH ('GEORGE') - LENGTH (REPLACE ('GEORGE', 'GE', NULL))
/ LENGTH ('GE') Counting from DUAL;

Oracle responds: **REPLACING**
5 // counting characters occurrences

Additional facts about Variables:

The command **accept** forces SQLPLUS to **define** the variable as equal to the entered value. And it can do this with a text message, with control over the datatype entered, and even with the response blanked out from viewing (such as for passwords). You can pass argument to a start file when it is started by embedding numbered variables in the **select** statements (rather than variables with names).

E.g. Select SysDate from DUAL Where '&1' > '&2';

Oracle responds: Enter value for 1:
Enter value for 2:

E.g. Select * from LEDGER Where Action BETWEEN '&1' AND '&2';

E.g. start ledger.sql 01-JAN-01 31-DEC-01

Enter the values for variables and then the answers will be returned. As with other datatypes, character and DATE datatypes must be enclosed in single quotation marks in the SQL statement. One limitation of this is that each argument following the word **start** must be a single word without spaces. Variable substitutions are not restricted to the SQL statement. The start file may use variables for such things as SQLPLUS command. Variables can be concatenated simply by pushing them together. You can concatenate a variable with a constant using a period:

E.g.
```
Select SUM (Amount) from LEDGER
Where ActionDate BETWEEN '01-&&Month.-01' AND
      LAST_DAY (TO_DATE ('01-&&Month.-01'));
```

This **select** statement will query once for a month and then build the two dates using a period as a concatenation operator.

Note: no period is necessary before the variable—only after it. The ampersand (s) tells SQLPLUS that a variable is beginning.

Related set Commands:

Set define changes the variable defining symbol from ampersand (&) to any symbol. **Set escape** defines a character you can place just in front of the ampersand (or other defined symbol) so that SQLPLUS will treat the symbol as a literal, not as denoting a variable. **Set concat** changes the concatenation symbol from a period to another. **Set scan** turns **on** or **off** the variable defining feature in SQLPLUS.

15. DECODE: Amazing Power in a Single Word:

It is one of the most powerful functions in Oracle. Syntax:

DECODE (*value*, *if1*, *then1*, *if2*, *then2*, *if3*, *then3*, . . . *else*)

Value represents any column in a table (regardless of datatype) or any result of a computation. *Value* is tested for each row. If *value* equals *if1* then the result of the DECODE is *then1*, and so on. If *value* equals none of the *if*s, the result of the DECODE is *else*. Each of the *if*s, *then*s and the *else* also can be a column or the result of a function or computation.

E.g.

```
Select ClientName, TRUNC ((AsOf - InvoiceDate) / 30) As Days,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 0, Amount, NULL) As This,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 1, Amount, NULL) As Thirty,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 2, Amount, NULL) As Sixty,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 3, Amount, NULL) As Ninety
From INVOICE, ASOF
Order by InvoiceDate;
```

A variable can also be used in a start file, with the **TO_DATE** function.

E.g.

```
Select ClientName, TRUNC ((AsOf - InvoiceDate) / 30) As Days,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 0, Amount, NULL) As This,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 1, Amount, NULL) As Thirty,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 2, Amount, NULL) As Sixty,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 3, Amount, NULL) As Ninety
From INVOICE, ASOF
Order by InvoiceDate;
```

For intervals other than exact multiplies of 30, this will not be a whole number, so it's truncated, thereby assuring a whole number as a result. Any date less than 30 before December 15 will produce a 0. A date 30 to 59 days before will produce a 1. A date 60 to 89 days before will produce a 2. A date 90 to 119 days before will produce a 3. The number 0, 1, 2, or 3 is the *value* in the **DECODE** statement.
Looking at the last **DECODE**. Following the first comma is a 3. This is the *if* test. If the *value* is 3, *then* the whole **DECODE** is statement will be the Amount in this row. If the value is anything other than 3 (meaning less than 90 days or more than 119), the **DECODE** will be equal to **NULL**. Compare the invoice dates with the Amounts in the NINETY column and judge the logic.

Collecting Clients Together:

As the next step in your analysis, you may want to collect all the clients together, along with the amounts they owe by period. A simple order by is added to the last SQL statement accomplishes this.

E.g.

```
Select ClientName, TRUNC ((AsOf - InvoiceDate) / 30) As Days,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 0, Amount, NULL) As This,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 1, Amount, NULL) As Thirty,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 2, Amount, NULL) As Sixty,
        DECODE (TRUNC ((AsOf – InvoiceDate) /30), 3, Amount, NULL) As Ninety
From INVOICE, ASOF
Order by ClientName, InvoiceDate;
```

Using MOD in DECODE:

The modulus function, **MOD**, can be used in conjunction with **DECODE** and RowNum to produce useful effects. RowNum is not a part of the row's location in the table or database, and has nothing to do with RowID. It is a number tacked onto the row and pulled from the database. Since **DECODE** acts by comparing a value to another value, you can use **MOD** within **DECODE** to facilitate comparisons. For any integer value, **MOD** (*value*, 1) will always be 0.

Order By and RowNum:

E.g.
```
Column Line Format 9999
Select RowNum, DECODE (MOD (RowNum, 5), 0, RowNum, NULL), Line,
        InvoiceDate, Amount from INVOICE;
```

The **order by** rearranges the order of the row numbers, which destroys the usefulness of the **DECODE**. If it is important to put the rows in a certain order, such as by InvoiceDate, and also to use the features of the **DECODE**, **MOD** and RowNum combination, this can be accomplish by creating a view where a **group by** does the ordering, as shown here:

E.g.
```
Create or replace view DATEANDAMOUNT as
        Select InvoiceDate, Amount from Invoice
                Group by InvoiceDate, Amount;
```

E.g.
```
Select RowNum, DECODE (MOD (RowNum, 5), 0, RowNum, NULL), Line,
        InvoiceDate, Amount from INVOICE
        Order by InvoiceDate;
```

E.g.
```
Select RowNum, DECODE (MOD (RowNum, 5), 0, RowNum, NULL), Line,
        InvoiceDate, Amount from DATEANDAMOUNT;
```

Columns and Computations in then and else:

Thus far, the *value* portion of **DECODE** has been the only real location of column names or computations. These can easily occur in the *then* and *else* portions of **DECODE**.

E.g.
```
Create or replace view AVERAGEPAY as
        Select AVG (DailyRate) AveragePay from PAY;
```

E.g.
```
Column DailyRate format 9999999.99
Select Name, DailyRate from PAY;
```

E.g.
```
Column NewRate format 9999999.99
Select Name, DailyRate
        DECODE (DailyRate, AveragePay, DailyRate, DailyRate * 1.15) As NewRate
        From PAY, AVERAGEPAY;          // increase DailyRate by 15 percent
```

Greater Than, Less Than, and Equal To in DECODE:

Clever use of functions and computations in the place of *value* can allow **DECODE** to have the effective ability to act based on a value being grater than, less than, or equal to another value.

E.g.
```
Select Name, DailyRate
        DECODE (SIGN ((DailyRate / AveragePay) –1), 1, DailyRate * 1.05,
                -1, DailyRate * 1.15, DailyRate) As NewRate
        From PAY, AVERAGEPAY;
```

Summary:

DECODE is a powerful function that uses *if*, *then*, *else* logic, and that can be combined with other Oracle functions to produce spreads of values (such as aging of invoices), effectively flip tables onto their sides, control display and page movement, make calculations and force row by row changes based on the *value* in a column (or computation) being tested.

16. SubQueries:

If some information is required that depends on some varying value, Subquery is the best solution for this. Otherwise, you have to write queries twice. A Subquery is a query that is written in the *where clause* of the main query. The subquery runs first and gives results to the

main query and now main query works with this result same as if it were written in the *where clause* of the main query.

E.g.
```
Select * from NEWSPAPER
Where section = (select section from NEWSPAPER
                    Where Feature = 'Doctor Is In');
```

This query first searches the NEWSPAPRE table's Section column where Feature is 'Doctor Is In' and then selects all columns of NEWSPAPER where section is that subquery has resulted.

Single-Values from a subquery:

a)
```
Select * from NEWSPAPER
Where section = (select section from NEWSPAPER
                    Where Page = 1);
```

This query will generate an Error:

* ERROR**:** ORA-1427**:** Single row subquery returns more than one rows.

Equal sign is single-value test but now it results all the rows where Page is equal to 1. If subquery returns single row, all logical operators can work properly.

b)
```
Select * from NEWSPAPER
Where Section < (select Section from NEWSPAPER
                    Where Feature = 'Doctor Is In');
```

List of Values from a subquery:

a)
```
Select City, Country from LOCATION
Where City IN (select City from WEATHER
                Where Condition = 'CLOUDY');
```

This query will select all rows from WEATHER table where Condition is equals 'CLOUDY' and then select City and Country columns from LOCATION where City is in the list that subquery resulted.

Note: Subquery must be enclosed in Parenthesis. One row-producing subquery can use single- or many-value operators but subquery producing many rows uses only many-value operators. ***BETWEEN*** can't be used with a subquery.

E.g.
```
Select * from WEATHER
Where TEMPERATURE BETWEEN 60
                    AND (select TEMPERATURE
                    From WEATHER
                    Where City = 'PARIS');
```

Using Subqueries Within the from Clause:

When you write a query that joins a table to a view, the view must already exist. If the view would only be used for that one query, though, you may be able to embed within the query the **select** statement that you would normally use to create view.

E.g.
```
Create or replace view TOTAL as
Select SUM (Amount) As TOTAL
        From LEDGER
Where ActionDate BETWEEN TO_DATE ('01-MAR-1901', 'DD-MM-YYYY')
                AND TO_DATE ('31-MAR-1901', 'DD-MM-YYYY')
        And Action IN ('BOUGHT', 'PAID');
```

The TOTAL view can be joined to LEDGER to show the percentage that each Amount value contributed to the sum of all the Amount values.

E.g.
```
Select L.Person, L.Amount, 100 * L.Amount / T.TOTAL
    From LEDGER L, TOTAL T
Where L.ActionDate BETWEEN TO_DATE ('01-MAR-1901', 'DD-MM-YYYY')
        AND TO_DATE ('31-MAR-1901', 'DD-MM-YYYY')
    And L.Action IN ('BOUGHT', 'PAID');
```

You can use place the view's syntax directly into the **from** clause of the query: you don't need to create the TOTAL view. The following listing shows the combined query. In the combined query, the TOTAL view's SQL is entered as a subquery in the query's **from** clause. The Total column from the subquery is used in the column list of the main query. The combined query is functionally identical to the query that used the TOTAL view.

E.g.
```
Select L1.Person, L1.Amount, 100 * L1.Amount / Total
    From LEDGER L1
    (Select SUM (Amount) TOTAL
    From LEDGER
    Where ActionDate BETWEEN TO_DATE ('01-MAR-1901', 'DD-MM-YYYY')
        AND TO_DATE ('31-MAR-1901', 'DD-MM-YYYY')
    And L.Action IN ('BOUGHT', 'PAID'))
Where ActionDate BETWEEN TO_DATE ('01-MAR-1901', 'DD-MM-YYYY')
    AND TO_DATE ('31-MAR-1901', 'DD-MM-YYYY')
    And L.Action IN ('BOUGHT', 'PAID'));
```

The benefit of the integrated approach is that you no longer need to create and maintain the TOTAL view. The query that required the view now contains a *subquery* that replaces the view definition.

Note: Subqueries can be complex. If you need to join columns from the subquery with columns from another table, simply give each column a unique alias, and reference the aliases in joins in the query's **where** clause.

17. Security for Users and Databases:

a) Creating User:

E.g.
```
Create user ZULAFQAR identified by ALI;
```

New user ZULAFQAR is created and password is ALI. *SQL is not case-sensitive.*

b) Changing Password:

E.g. (i)
```
Alter user ZULAFQAR identified by ZULFI;
```

Now password is ZULFI.

E.g. (ii)
```
Password                    //(Press Enter Key)
Changing password for ZULAFQAR.
Old password:
New password:
Retype new password:
```

Both command are used same for changing password.

c) What is ROLE?

ROLE is an object that has privileges (rights) for users. ROLE can have more than one privilege. We can create our roles or use already existing ones. Three built-in ROLES are: CONNECT, RESOURCE, DBA.

d) Granting Privileges to Users (DBA to User):

E.g.
```
Grant CONNECT, RESOURCE, DBA to ZULAFQAR
[With admin option];
```

With this method ROLES/ Privileges can be granted to users. If the user can grant this privilege to other users use ***with admin option.***

e) Revoking Privileges from Users:

E.g.
```
Revoke CONNECT, RESOURCE, DBA from ZULAFQAR
[With admin option];
```
E.g.
```
revoke all on QUALITY from ZULAFQAR;
```

A DBA user can revoke all users even other DBA user from any privileges.

NOTE: Revoking everything from a given user does not eliminate that user from Oracle, nor does it destroy any tables that user had created: it simply prohibits that user's access to them. Other users with access to the tables will still have exactly the access they've always had.

f) Deleting Users:

E.g.
```
drop user ZULAFQAR [cascade];
```

The *cascade* option drops the user along with all the objects owned by the user, including referential integrity constraints. Without *cascade* option a user that have any object can't be deleted.

g) What users can Grant:

E.g.
```
Grant object privilege [(column [, column])]
On object to {user | role}
[With grant option];
```

Privilege list is the following:

- On the user's tables, views and snapshots (materialized views):

  (i) INSERT (ii) UPDATE (all or specific column)
  (iii) DELETE (iv) SELECT

- On tables you can also grant:

  (i) ALTER (tables—all or specific columns—or sequence)
  (ii) REFERENCES (iii) INDEX (columns in a table)
  (iv) ALL (of the items previously listed)

- On procedures, functions, packages, abstract datatypes, libraries, indextypes and operators:

  (i) EXECUTE

- On sequences:

  (i) SELECT (ii) ALTER

- On directories (for BFILE LOB datatypes):

  READ

E.g.
```
Grant select on QUALITY to ZULAFQAR;
```

h) Moving to another user with connect:

While managing users you can move between users:

E.g. (i) Connect ZULAFQAR/ ZULFI;
E.g. (ii) Connect
Enter user-name:
Enter Password:
E.g. (iii) Connect ZULAFQAR
Enter Password:

i) Querying other users' tables:

While querying other users' tables you must type username.table.

E.g. (i) Select * from Stock.QUALITY;
E.g. (ii) Create view Quality as select * from Stock.QUALITY;

j) Granting / Revoking Privileges on tables (DBA / User to user):

A user or DBA can grant privileges on personal tables, views etc:

E.g. (i) Grant select, update on QUALITY to ZULAFQAR;
E.g. (ii) revoke all on QUALITY from ZULAFQAR;

k) Creating a Role:

Instead of granting privileges to users, one by one, on tables, simply create a ROLE and grant all desired privileges to it and then grant it to which you want. To create a role you must have CREATE ROLE system privilege. Syntax:

Create role role_name
[Not identified | identified [by password | externally]] ;

E.g. create role CLERK;
E.g. create role MANAGER;

(i) **Granting Privileges to a Role:**

E.g. Grant select on QUALITY to CLERK;
E.g. Grant CREATE SESSION to CLERK;
E.g. Grant CREATE SESSION, CREATE DATABASE LINK to MANAGER;

(ii) **Granting a Role to another Role:**

E.g. Grant CLERK to MANAGER;
E.g. Grant CLERK to MANAGER with admin option;

With admin option give grantee the power to alter, drop or grant the role to other roles, users.

(iii) **Granting a Role to Users:**

E.g. Grant CLERK to ZULAFQAR;
E.g. Grant CLERK to ZULAFQAR with admin option;

NOTE: Privileges that are granted to users via roles cannot be used as the basis for views, procedures, functions, packages or foreign keys When creating these types of datatypes, you must rely on direct grants of the necessary privileges.

(iv) **Adding a password to a Role:**

E.g. Alter role MANAGER identified by Zulfi;
E.g. Alter role MANAGER identified externally;

(v) **Removing a password from a Role:**

E.g. Alter role MANAGER not identified;

(vi) **Enabling and Disabling Roles:**

To set a role default for a user, type ***alter user*** command. Syntax:

```
Alter user username
Default role {[role1, role2]
[All | all except role1, role2] [None]};
```

E.g.
```
Alter use ZULAFQAR
Default role CLERK;
```

To enable a nonedefault role use ***set role*** command.

E.g. set role CLERK;
E.g. set role all;
E.g. Set role all except CLERK;
E.g. Set role MANAGER identified by Zulfi; // for role with password

To disable a role in your session:

E.g. set role none;

(vii) **Revoking Privileges from a Role:**

E.g. Revoke SELECT on QUALITY from CLERK;

Now all users with CLERK role are unable to query QUALITY table.

(viii) **Dropping a Role:**

E.g. drop role MANAGER;

(ix) **Granting update to specific Columns:**

E.g. Grant update (Noon, Midnight) on COMFORT to ZULAFQAR;

(x) **Revoking Privileges:**

E.g.
```
Revoke object privilege [object privilege…]
On object
From {user | role} [, {user | Role}]
[Cascade constraints];
```

E.g. revoke all;

l) Security by User:

A table has the information of salary of workers the manager wants to give access to each user to select the table but can view only his own salary row.

E.g.
```
Create view SALARY as
Select * from WORKER
Where SUBSTR (Name, 1, INSTR (Name, ' ') -1) = User;
```
E.g. Create public synonym for ZULAFQAR.SALARY;

m) Granting Access to the Public:

E.g. Grant select on SALARY to public;

n) Granting Limited Resources:

E.g.
```
Alter user ZULAFQAR
Quota 100M on USERS;
```

18. Random Functions:

a) Selecting random number

E.g. **Select** dbms_random.random **from** dual;

b) Selecting random row from a table:

E.g.
```
SELECT column FROM
(SELECT column FROM table
ORDER BY dbms_random.value)
WHERE rownum = 1;
```

c) Selecting a few random records of a table

E.g. Select * from <table> sample(0.2); -- 0.2 is %

d) Selecting a few random records of a table

E.g.
```
SELECT * FROM table_name
WHERE primary_key IN
(SELECT primary_key FROM
(SELECT primary_key, SYS.DBMS_RANDOM.RANDOM
 FROM table_name ORDER BY 2)
WHERE rownum <= 10);
```

19. Snapshots:

a) Create actual table's log in the same user

E.g. Create Snapshot Log on Emp

b) Create Snapshot in the target user & database

E.g. Create Snapshot Emp as Select * from Emp@databaselink

c) Drop Snapshot in the target user & database

E.g. Drop materialized view Emp

d) Drop Snapshot log from the actual user

E.g. Drop snapshot log on Scott.Emp

SQL*PLUS COMMANDS BRIEFING

1. Set commands:

Set *keyword* [Option]

E.g. set feedback off
E.g. set autocommit off
E.g. set autocommit 5

This command is used for setting something option SQL*PLUS.

2. Show commands:

Show *keyword*

E.g. show feedback

This command is used for viewing what option is set in SQL*PLUS.

a) Set feedback off /on
b) Set feedback 25
c) Show feedback

(Feedback is the information that SQL returns to a user when he runs a command.
E.g. *14 rows selected. Table created.* These commands set it on/off or *on* if rows are more are more than 25. Show feedback gives the information what option is set.)

d) Set numwidth 5
e) Show numwidth

3. Listing Query rows:

E.g. list

It will number all rows in the last query and also place * at current row. You can make any row the current row.

E.g. list 1

It will make row 1 the current row.

4. Changing SQL Prompt:

E.g. Set SQLPROMPT 'Zulfi> '

5. Modifying a Query:

After using *List* command you can modify any row in the last query.

E.g. change /select/select

It replaces spelling of the word after first '/' with second '/'. Then write / on SQL and the last query will run with new spelling. You can also use **$** instead of /.

6. Deleting rows:

After listing rows you can easily delete any number of rows from the query. *The word "delete" should not be used.* It will replace the entire query with word "delete". Always use "**del"** word**.**

E.g. del // deletes the current line

E.g. del 3 7 // deletes lines 3 to 7

E.g. del 2 LAST // deletes lines 2 to last line

7. Append command:

Append places its text right up at the end of the current line, with no spaces in between. To put a space in, type *two spaces* between the word *append* and *text*.

E.g. list 1
1 * select Feature, Section, Page
E.g. append "Where It Is"

8. Input command:

You may also **input** a whole new line after the current line.

E.g. list
Select * from NEWSPAPER
E.g. input where Section = 'A'

9. Clear Buffer:

To remove all query from the buffer use the following command:

E.g. clear buffer

10. Save Command:

To save any SQL command for using later, just typing its name type:

E.g. Save Zulfi.sql

To save changes in an existing file:

E.g. Save Zulfi.sql repl

11. Start Command:

To start any saved SQL file having command type:

E.g. Start Zulfi.sql

12. Spool/Spool off Command:

To save the start file's results into a file type the following in the end of each SQL query.

E.g. Spool / spool off

13. Store Command:

To save the current SQL*PLUS setting into a file type:

E.g. Store set my_setting.sql create

For an existing file use **replace** instead of **create**. You can also **append** new settings to an existing file.

Uninstall Oracle:

The simplest way to remove Oracle is to run the Oracle installer:

Start > Programs > Oracle Installation Products > Universal Installer

1. On the first screen click on "Deinstall Products..."
2. Expand the tree view (just so that the second level is visible) and make sure you select everything that is selectable.
3. Click on "Remove..."
4. On the confirmation screen click "Yes"
5. When it has finished click "Close" and then "Exit" to quit the installer

Whilst the Oracle installer removes many components there are a number of things that it leaves behind. In order to completely remove all traces of Oracle the following additional steps will need to be taken:

i. Stop any Oracle services that have been left running. (Start > Settings > Control Panel > Services.)
Services which I have found left behind are 'OracleOraHome90TNSListener' and 'OracleServiceORACLE'. However there may be others depending on your installation. Look for any services with names starting with 'Oracle'.

ii. Run regedit (Start > Run > Enter "regedit", click "Ok"), find and delete the following keys:

```
HKEY_LOCAL_MACHINE
  \SOFTWARE
    \ORACLE

HKEY_LOCAL_MACHINE
  \SYSTEM
    \CurrentControlSet
      \Services
        \EventLog
          \Application
            \Oracle.oracle
```

Note: I have had it reported that some people also have registry entries saved under HKEY_CURRENT_USER\SOFTWARE\ORACLE, this registry entry may be created by some Oracle utilities. If it exists then delete it.

iii. Delete the Oracle home directory:
"C:\Oracle"
This will also remove your database files (unless you located them elsewhere, in which case you will need to delete them separately).

iv. Delete the Oracle program Files directory:
"C:\Program Files\Oracle"

v. Delete the Oracle programs profile directory:
"C:\Documents and Settings\All Users\Start Menu\Programs\Oracle - OraHome90"
if you did not first run the Oracle installer to remove Oracle then there may be other Oracle profile group directories to remove.

vi. Some of the Oracle services may be left behind by the uninstall. Open 'services' on the control panel, make a note of which Oracle services remain and see the notes 'How to remove a service' to remove them.

vii. If you didn't first run the Oracle Installer to remove Oracle then you may have some references to Oracle left in the path. To remove these: Start > Settings > Control Panel > System > Advanced > Environment Variables, look at both the use and system variable 'PATH' and edit them to remove any references to Oracle.

Acknowledgements: My appreciation to Alistair Jones for help on this procedure and for encouragement to write it.