

DBMS SQL

DBMS SQL

Introduction

What is SQL?

- SQL:
 - SQL stands for Structured Query Language.
 - SQL is used to communicate with a database.
 - Statements are used to perform tasks such as update data on a database, or retrieve data from a database.
 - Almost all modern Relational Database Management Systems like MS SQL Server, Microsoft Access, MSDE, Oracle, DB2, Sybase, MySQL, Postgres, and Informix use SQL as standard database language.
 - However, although all those RDBMS use SQL, they use different SQL dialects.
 - ❖ **For example:** MS SQL Server specific version of the SQL is called T-SQL, Oracle version of SQL is called PL/SQL which also supports Procedural Language features, MS Access version of SQL is called JET SQL, etc.

- Oracle has further modified SQL to support ORDBMS features.

- Benefits of SQL are:

 - ❖ It is a Non-Procedural Language.

 - ❖ It is a language for all users.

 - ❖ It is a unified language.

- Rules for SQL statements:

 - SQL keywords are not case sensitive. However, normally all commands (SELECT, UPDATE, etc) are upper-cased.

 - “Variable” and “parameter” names are displayed as lower-case.

 - New-line characters are ignored in SQL.

 - Many DBMS systems terminate SQL statements with a semi-colon character.

 - “Character strings” and “date values” are enclosed in single quotation marks while using them in WHERE clause or otherwise.

- A database is a collection of structures with appropriate authorizations and accesses that are defined.
- The structures in the database like tables, views, indexes, sequences etc. are called as objects in the database.
- All objects that belong to the same user are said to be the “schema” for the particular user.
- Information about existing objects can be retrieved from `dba_/user_/all_objects`.

SQL Commands

- “SQL commands” are “instructions” used to communicate with the database to perform “specific tasks” that work with data.
 - A database is a collection of structures with appropriately defined authorizations and accesses. The tables, indexes are structures in the database and are called as “objects” in the database.
 - The names of tables, indexes, and those of columns are called “identifiers”.
- SQL commands can be used not only for searching the database, but also to perform various other functions.
 - **For example:** You can create tables, add data to tables, or modify data, drop the table, set permissions for users, etc.

SQL Statement Groups

- Given below are the standard SQL statement groups:

Groups	Statements	Description
DQL	SELECT	DATA QUERY LANGUAGE - It is used to get data from the database and impose ordering upon it.
DML	DELETE INSERT UPDATE MERGE	DATA MANIPULATION LANGUAGE - It is used to change database data.
DDL	DROP TRUNCATE CREATE ALTER	DATA DEFINITION LANGUAGE - It is used to manipulate database structures and definitions.
TCL	COMMIT ROLLBACK SAVEPOINT	TCL statements are used to manage the transactions.
DCL (Rights)	REVOKE GRANT	They are used to remove and provide access rights to database objects.

SQL Statement Groups

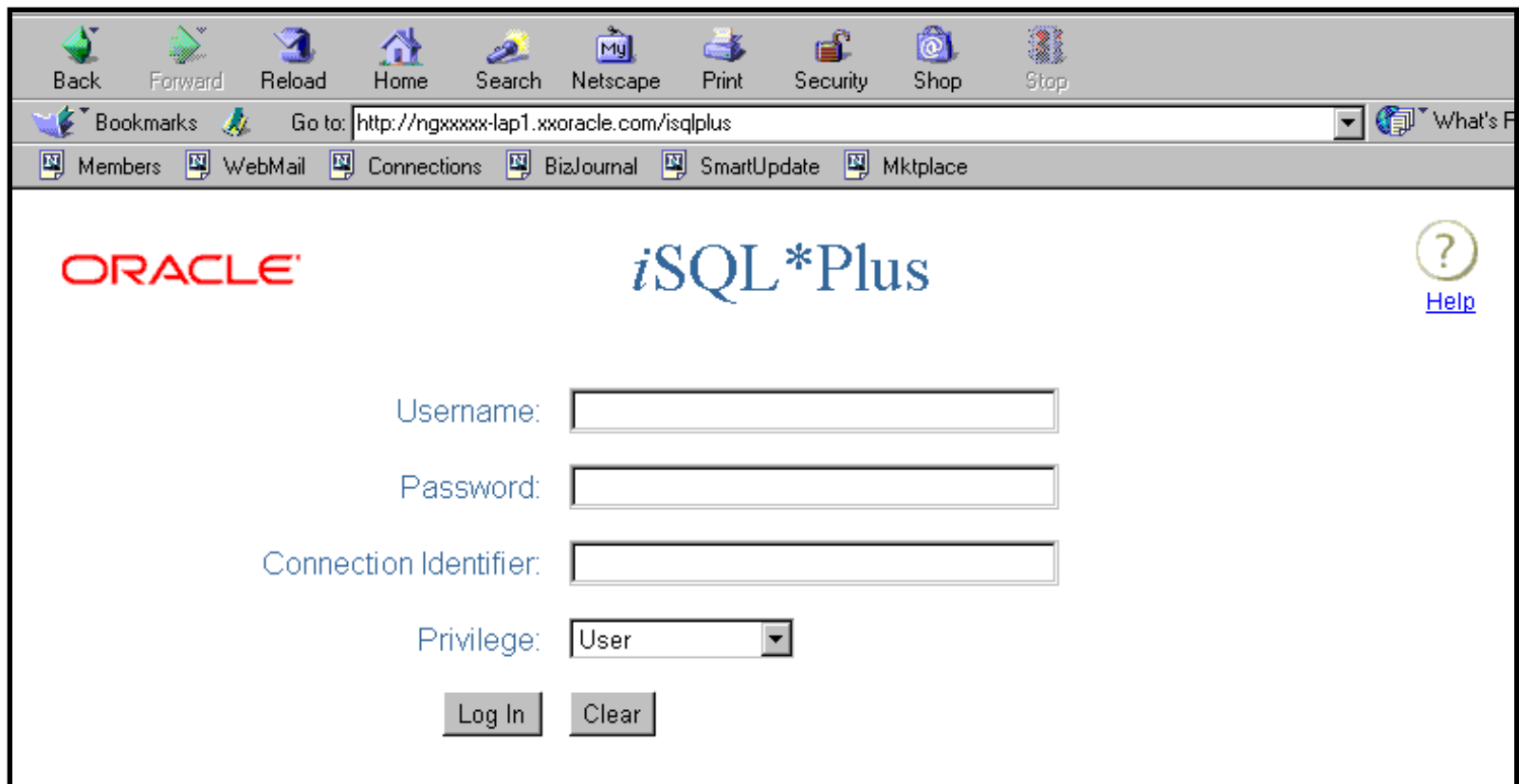
- SQL commands are grouped into four major categories depending on their functionality:
 - **Data Definition Language (DDL):** These SQL commands are used for creating, modifying, and dropping the structure of database objects. These SQL statements define the structure of a database, including rows, columns, tables, indexes, and database specifics such as file locations, etc. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
 - A CREATE statement in SQL creates an object inside a Relational Database Management System (RDBMS) such as table, index, constraints, etc. The types of objects that can be created depend on which RDBMS is being used. However, most support creation of Tables, Indexes, Users, and Databases.

SQL Statement Groups

- An ALTER statement in SQL changes the properties of an object inside a Relational Database Management System (RDBMS).
- **Data Manipulation Language (DML):** These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.
- **Transaction Control Language (TCL):** These SQL commands are used for managing changes that affect the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
- **Data Control Language (DCL):** These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.
 - GRANT is used to allow specified users to perform specified tasks.
 - REVOKE is used to cancel previously granted or denied permissions.

Logging to Server

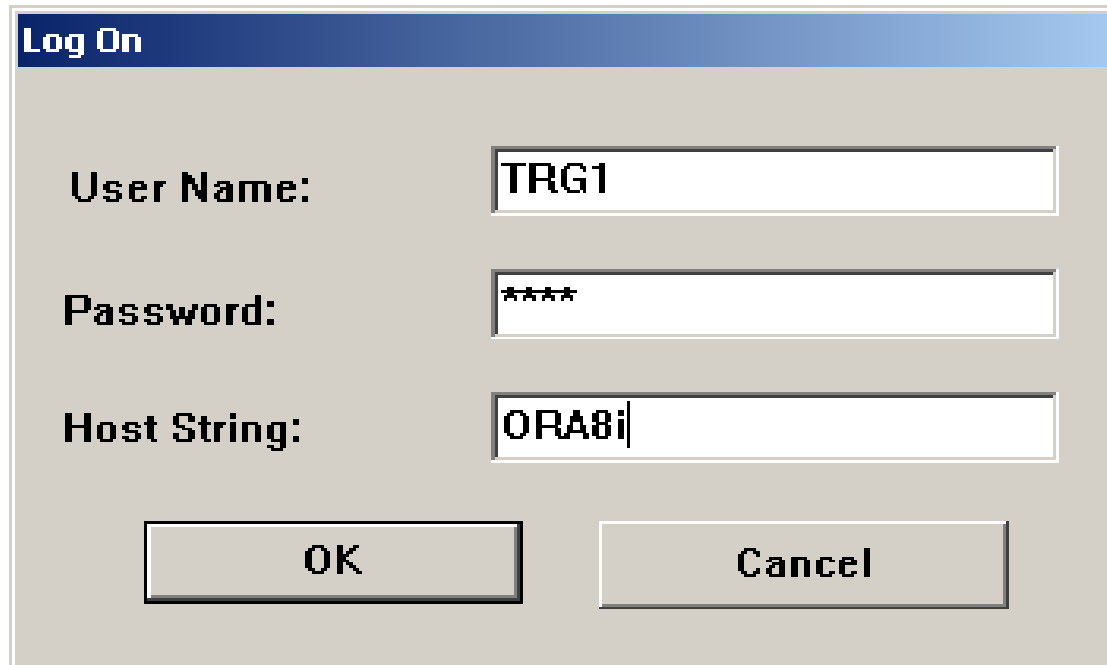
- To log into the iSQL*Plus environment:
 - In the Windows browser, type the URL in the address field. The user will be directed to iSQL*Plus environment screen.



The screenshot shows a Netscape browser window with the address bar set to `http://ngxxxx-lap1.xxoracle.com/isqlplus`. The browser's toolbar includes buttons for Back, Forward, Reload, Home, Search, Netscape, Print, Security, Shop, and Stop. Below the toolbar, there are links for Members, WebMail, Connections, BizJournal, SmartUpdate, and Mktplace. The main content area displays the Oracle logo and the text "iSQL*Plus". To the right of the text is a circular help icon with a question mark and a "Help" link. The login form consists of four input fields: "Username:", "Password:", "Connection Identifier:", and "Privilege:". The "Privilege:" field is a dropdown menu currently showing "User". At the bottom of the form are two buttons: "Log In" and "Clear".

Logging to Server

- To connect to the Oracle server:
 1. Select Start, go to Programs, and select Oracle-OraHome8I.
 2. Go to Application Development, and select SQL Plus. You will get the following logon screen:



The image shows a 'Log On' dialog box with a blue header bar. It contains three input fields: 'User Name:' with the text 'TRG1', 'Password:' with five asterisks '*****', and 'Host String:' with the text 'ORA8i'. At the bottom, there are two buttons: 'OK' and 'Cancel'.

Log On	
User Name:	TRG1
Password:	*****
Host String:	ORA8i
<div>OK Cancel</div>	

Executing SQL Commands

- The semicolon ';' at the end of the command is the command terminator. The indentation is optional.
 - There are many ways to terminate a command, namely:
 1. A semicolon at the end of a line
 2. A semicolon on a line by itself
 3. A slash '/' on a line by itself
 4. A blank line
 - Whenever an SQL command is issued, the command is stored in memory, in an area called "SQL buffer".
 - ❖ If a semicolon or slash is used as command terminator, then the command is immediately executed, and then stored in "SQL buffer".
 - ❖ When a blank line is used as command terminator, then the command is not immediately executed. It is merely stored in the buffer.

SQL Commands

To see the SQL command in the buffer

➤ SQL> LIST

To save the buffer contents into a file

➤ SQL> SAVE file_name

- If the user does not supply an extension, then ORACLE adds .sql as the default extension.

➤ SAVE file1

To load the contents of a file into memory after flushing the buffer

➤ SQL> CLEAR BUFFER

➤ SQL> GET file_name

To load the contents of a file into memory, and execute the commands in the file

➤ SQL> @filename OR SQL> START file_name

To execute the commands in the buffer

➤ SQL> RUN OR SQL> /

DBMS SQL

Data Query Language

SELECT Statement

- The SELECT command is used to retrieve rows from a single table or multiple Tables or Views.
 - A query may retrieve information from specified columns or from all of the columns in the Table.
 - It helps to select the required data from the table.

```
SELECT [ALL | DISTINCT] { * | col_name,...}
FROM table_name alias,...
    [ WHERE expr1 ]
    [ CONNECT BY expr2 [ START WITH expr3 ] ]
    [ GROUP BY expr4 ] [ HAVING expr5 ]
    [ UNION | INTERSECT | MINUS SELECT ... ]
    [ ORDER BY expr | ASC | DESC ];
```

SELECT Statement

- The tabular result is stored in a result table (called the result-set). The statement begins with the SELECT keyword. The basic SELECT statement has three clauses:
 - SELECT (clause specifies the table columns that are retrieved.)
 - FROM (clause specifies the tables accessed.)
 - WHERE (clause specifies which table rows are used. The WHERE clause is optional; if missing, all table rows are used.)
- **Note:**
 - Each clause is evaluated on the result set of a previous clause. The final result of the query will be always a “result table”.
- Only FROM clause is essential. The clauses WHERE, GROUP BY, HAVING, ORDER BY, UNION are optional.

SELECT Statement

- Example:

```
SELECT *  
FROM EMP
```

❖ Fetches all the rows and columns from EMP Table.

- Example:

```
SELECT EMPNO,ENAME  
FROM EMP
```

❖ Fetches all the rows and specific columns from EMP Table.

- Example:

```
SELECT EMPNO "Employee No." ,ENAME "Names"  
FROM EMP
```

❖ Fetches all the rows and specific columns from EMP Table and display the column headings as **Employee No** and **Names**.

SELECT Statement

- The WHERE clause is used to specify the criteria for selection.
- The WHERE clause is used to perform “selective retrieval” of rows. It follows the FROM clause, and specifies the search condition.
- The result of the WHERE clause is the row or rows retrieved from the Tables, which meet the search condition.
- **Comparison Predicates:**
 - The Comparison Predicates specify the comparison of two values.
 - It is of the form:

< Expression> < operator > < Expression>

< Expression> <operator> <subquery>

Operators

- Mathematical Operators:

- Examples: +, -, *, /

- Comparison Operators:

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

- Logical Operators:

- Examples: AND, OR, NOT

SELECT Clause

- Example:
SELECT *
FROM EMP
WHERE SAL > 1500
❖ Fetches all the rows and columns from EMP Table where salary is greater than 1500.
- Example:
SELECT EMPNO,ENAME
FROM EMP
WHERE JOB='CLERK'
❖ Fetches all the rows and specific columns from EMP Table where the designation is **CLERK**.
- Example:
SELECT EMPNO "Employee No.",ENAME "Names"
FROM EMP
WHERE HIREDATE > '22-FEB-81'
❖ Fetches all employee details who joined after 22nd FEB 1981.

Other Comparison operators	Description
[NOT] BETWEEN x AND y	Allows user to express a range. For example: Searching for numbers BETWEEN 5 and 10. The optional NOT would be used when searching for numbers that are NOT BETWEEN 5 AND 10.
[NOT] IN(x,y,...)	Is similar to the OR logical operator. Can search for records which meet at least one condition contained within the parentheses. For example: Pubid IN (1, 4, 5), only books with a publisher id of 1, 4, or 5 will be returned. The optional NOT keyword instructs Oracle to return books not published by Publisher 1, 4, or 5.
[NOT] LIKE	Can be used when searching for patterns if you are not certain how something is spelt. For example: title LIKE 'TH%'. Using the optional NOT indicates that records that do contain the specified pattern should not be included in the results.

Other Comparison Operators

Other Comparison operators	Description
IS[NOT]NULL	<p>Allows user to search for records which do not have an entry in the specified field.</p> <p>For example: Shipdate IS NULL.</p> <p>If you include the optional NOT, it would find the records that do not have an entry in the field.</p> <p>For example: Shipdate IS NOT NULL.</p>

- Logical operators are used to combine conditions.
 - Logical operators are NOT, AND, OR.
 - NOT reverses meaning.
 - AND both conditions must be true.
 - OR at least one condition must be true.
 - **Note** : The “**parenthesis**” specifies the order in which the operators should be evaluated. Without parenthesis, the AND operator has a stronger binding than the OR operator.

Operator Precedence

- Operator precedence is decided in the following order:

Levels	Operators
1	* (Multiply), / (Division), % (Modulo)
2	+ (Positive), - (Negative), + (Add), (+ Concatenate), - (Subtract), & (Bitwise AND)
3	=, >, <, >=, <=, <>, !=, !>, !< (Comparison operators)
4	NOT
5	OR
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	= (Assignment)

SELECT CLAUSE

- Example:

```
SELECT *  
FROM EMP  
WHERE SAL >= 1500 AND SAL <= 2500
```

- ❖ Fetches all the rows and columns from EMP Table where salary is greater than equal to 1500 and less than or equal to 2500.

```
SELECT *  
FROM EMP  
WHERE SAL BETWEEN 1500 AND 2500
```

- ❖ To inverse the search use **NOT** operator

```
SELECT *  
FROM EMP  
WHERE SAL NOT BETWEEN 1500 AND 2500
```


SELECT CLAUSE

- Example:

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE JOB='CLERK' OR JOB='MANAGER'
```

- ❖ Fetches all the rows and specific columns from EMP Table where the designation is **CLERK** or **MANAGER**.

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE JOB IN('CLERK','MANAGER')
```

- ❖ To inverse the search use the operator **NOT**.

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE JOB NOT IN('CLERK','MANAGER')
```

SELECT CLAUSE

- Example:

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE NAME LIKE 'A%'
```

- ❖ Fetches all the rows where the employee name starts with A.

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE NAME LIKE '_A%'
```

- ❖ Fetches all the rows where the employee name's second character is A.

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE NAME NOT LIKE '%A'
```

- ❖ Fetches all the rows where the employee name ends with A.

```
SELECT EMPNO,ENAME  
FROM EMP  
WHERE NAME NOT LIKE 'A%'
```

- ❖ To inverse the search use the operator NOT.

- The SQL DISTINCT clause is used to eliminate duplicate rows.
 - **For example:** List all jobs in the EMP table. However, job should be displayed only once, even if duplicate values exist for the job.
- The ORDER BY clause presents data in a sorted order.
 - It uses an “ascending order” by default.
 - You can use the DESC keyword to change the default sort order.
 - It can process a maximum of 255 columns.
 - In an ascending order, the values will be listed in the following sequence:
 - ❖ Numeric values, Character values, NULL values
 - In a descending order, the sequence is reversed.

SELECT CLAUSE

- Example:

```
SELECT DISTINCT JOB  
FROM EMP
```

- ❖ Fetches all the unique designations specified in EMP table.

```
SELECT EMPNO,ENAME  
FROM EMP  
ORDER BY ENAME
```

- ❖ Fetches all the rows in ascending order of name.

```
SELECT EMPNO,ENAME  
FROM EMP  
ORDER BY ENAME DESC
```

- ❖ Fetches all the rows in descending order of names.

```
SELECT EMPNO,ENAME  
FROM EMP  
ORDER BY JOB DESC,ENAME
```

- ❖ Fetches all the rows in descending order of jobs and ascending order of names.

Tips and Tricks

- It is necessary to always include a WHERE clause in your SELECT statement to narrow the number of rows returned.
 - If you do not use a WHERE clause, then Oracle will perform a table scan of your table, and return all the rows.
 - By returning data you do not need, you cause the SQL engine to perform I/O it does not need to perform, thus wasting SQL engine resources.
 - In addition, the above scenario increases network traffic, which can also lead to reduced performance.
 - And if the table is very large, a table scan will lock the table during the time-consuming scan, preventing other users from accessing it, and will hurt concurrency.

Tips and Tricks

- Carefully evaluate whether the SELECT query requires the DISTINCT clause or not.
 - The DISTINCT clause should only be used in SELECT statements.
 - ❖ This is mandatory if you know that “duplicate” returned rows are a possibility, and that having duplicate rows in the result set would cause problems with your application.
 - The DISTINCT clause creates a lot of extra work for SQL Server.
 - ❖ The extra load reduces the “physical resources” that other SQL statements have at their disposal.
 - Hence, use the DISTINCT clause only if it is necessary.

Tips and Tricks

- In a WHERE clause, the various “operators” that are used, directly affect the query performance.
 - Some operators tend to produce speedy results than other operators. Of course, you may not have choice of using an operator in your WHERE clauses, but sometimes you do have a choice.
 - Using simpler operands, and exact numbers, provides the best overall performance.
 - If a WHERE clause includes multiple expressions, there is generally no performance benefit gained by ordering the various expressions in any particular order.
 - This is because the Query Optimizer does this for you, saving you the effort. There are a few exceptions to this, which are discussed further in the lesson.

Tips and Tricks

- Given below are the key operators used in the WHERE clause, ordered by their performance. The operators at the top produce faster results, than those listed at the bottom.

❖ =

❖ >, >=, <, <=

❖ LIKE

❖ <>

- Use “=” as much as possible, and “<>” as least as possible.

- **Don't include code that does not do anything**

- This may sound obvious. However, this scenario is seen in some off-the-shelf applications.
- For example, you may see code which is given below:

- `SELECT column_name FROM table_name WHERE 1 = 0`

Tips and Tricks

- When this query is run, no rows will be returned. It is just wasting SQL Server resources.
- By default, some developers routinely include code, which is similar to the one given above, in their WHERE clauses when they make string comparisons.
- For example:
 - ❖ `SELECT column_name FROM table_name WHERE LOWER(column_name) = 'name'`
- **Any use of text functions in a WHERE clause decreases performance.**
- If your database has been configured to be case-sensitive, then using text functions in the WHERE clause does decrease performance. However, in such a case, use the technique described below, along

Tips and Tricks

- with appropriate indexes on the column in question:
 - ❖ `SELECT column_name FROM table_name WHERE column_name = 'NAME'`
or `column_name = 'name'`
- This code will run much faster than the first example.
- In your queries, do not return column data that is not required.
 - For example:
 - ❖ You should not use `SELECT *` to return all the columns from a table if all the data from each column is not required.
 - ❖ In addition, using `SELECT *` prevents the use of covered indexes, further potentially decreasing the query performance.

Tips and Tricks

- If you use LIKE in your WHERE clause, try to use one or more leading character in the clause, if at all possible.
 - **For example:** Use LIKE 'm%' not LIKE '%m'
 - If you use a leading character in your LIKE clause, then the Query Optimizer has the ability to potentially use an Index to perform the query. Thus speeding performance and reducing the load on SQL engine.
 - However, if the leading character in a LIKE clause is a “wildcard”, then the Query Optimizer will not be able to use an Index. Here a table scan must be run, thus reducing performance and taking more time.
 - The more leading characters you use in the LIKE clause, it is more likely that the Query Optimizer will find and use a suitable Index

Tips and Tricks

- Certain operators in the WHERE clause prevents the query optimizer from using an Index to perform a search.
 - For example: “IS NULL”, “<>”, “!=”, “!>”, “!<”, “NOT”, “NOT EXISTS”, “NOT IN”, “NOT LIKE”, and “LIKE ‘%500’”
- Suppose you have a choice of using the IN or the BETWEEN clauses. In such a case use the BETWEEN clause, as it is much more efficient.
 - Assuming there is a useful Index on customer_number, the Query Optimizer can locate a range of numbers much faster by using BETWEEN clause.
 - This is much faster than it can find a series of numbers by using the IN clause (which is really just another form of the OR clause).

Tips and Tricks

- Using Efficient Non-index WHERE clause sequencing:
 - Oracle evaluates un-indexed equations, linked by the AND verb in a bottom-up fashion.
 - ❖ This means that the first clause (last in the AND list) is evaluated, and if it is found TRUE, then the second clause is tested.
 - Always try to position the most expensive clause first in the WHERE clause sequencing.
 - Oracle evaluates un-indexed equations, linked by the OR verb in a top-down fashion.
 - ❖ This means that the first clause (first in the OR list) is evaluated, and if it is found FALSE, then the second clause is tested.
 - Always try to position the most expensive OR clause last in the WHERE clause sequencing.

Tips and Tricks

- Do not use ORDER BY in your SELECT statements unless you really need to use it.
 - The ORDER BY clause adds a lot of extra overhead.
 - ❖ **For example:** Sometimes it may be more efficient to sort the data at the client than at the server. In other cases, the client does not even need sorted data to achieve its goal. The key here is to remember that you should not automatically sort data, unless you know it is necessary.
 - Whenever SQL Server has to perform a sorting operation, additional resources have to be used to perform this task. Sorting often occurs when any of the following Transact-SQL statements are executed:
 - ❖ ORDER BY
 - ❖ GROUP BY
 - ❖ SELECT DISTINCT
 - ❖ UNION

Tips and Tricks

- ❖ CREATE INDEX (generally not as critical as happens much less often)
- In many cases, these commands cannot be avoided. On the other hand, there are few ways in which sorting overhead can be reduced, like:
 - Keep the number of rows to be sorted to a minimum. Do this by only returning those rows that absolutely need to be sorted.
 - Keep the number of columns to be sorted to the minimum. In other words, do not sort more columns than required.
 - Keep the width (physical size) of the columns to be sorted to a minimum.
 - Sort column with number datatypes instead of character datatypes.

DBMS SQL

Aggregate (GROUP) Functions

Functions

- Functions are similar to operators in that they manipulate data items and return a result.
- Functions differ from operators in the format in which they appear with their arguments. This format allows them to operate on zero, one, two, or more arguments:
 - `function(argument, argument, ...)`

Types of Functions

- There are two types of functions: SQL functions and User-defined functions.
 1. **SQL functions:** SQL functions are built into most DBMS and are available for use in various appropriate SQL statements. SQL functions are further categorized as:
 - a) **Single-Row functions:** Single-row functions return a single result row for every row of a queried Table or View.
 - ✓ For example: ABS, ROUND etc.
 - b) **Object Reference functions:** Object functions manipulate REFs, which are references to objects of specified object types.
 - ✓ For example: REF, VALUE, etc.
 - c) **Aggregate functions:** Aggregate functions return a single row based on groups of rows, rather than on single rows.
 - For example: MAX, MIN, COUNT, etc.

Types of Functions

2. User-defined functions: You can write user-defined functions in PL/SQL or Java to provide functionality that is not available in SQL or SQL functions. User-defined functions can appear in a SQL statement wherever SQL functions can appear, that is, wherever an expression can occur.

- **For example:** User-defined functions can be used in the following:
- a) The select list of a SELECT statement
 - b) The condition of a WHERE clause
 - c) CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
 - d) The VALUES clause of an INSERT statement
 - e) The SET clause of an UPDATE statement

Explanation

- SQL provides a set of “built-in” functions for producing a single value for an entire group. These functions are called as “Set functions” or “Aggregate (Group) functions”.
- These functions can work on a “normal result table” or a “grouped result table”.
 - If the result is not grouped, then the aggregate will be taken for the whole result table.
- The Group functions return one value for the entire groups of rows.

GROUP Functions

- Given below is a list of Group functions supported by SQL:

Function	Value returned
SUM (expr)	Sum value of expr, ignoring NULL values.
AVG (expr)	Average value of expr, ignoring NULL values.
COUNT (expr)	Number of rows where expr evaluates to something other than NULL. COUNT(*) counts all selected rows, including duplicates and rows with NULLs.
MIN (expr)	Minimum value of expr.
MAX (expr)	Maximum value of expr.

- The Aggregate functions ignore NULL values in the column.
 - To include NULL values, NVL function can be used with Aggregate functions.

HAVING CLAUSE

- HAVING clause is used to filter data based on the Group functions.
- The HAVING search condition applies to “each group”. It can be:
 - formed using various predicates like between, in, like, null, comparison, etc
 - combined with Boolean operators like AND, OR, NOT
- Since the search condition is for a “grouped table”. The predicates should be:
 - on a column by which grouping is done.
 - on a set function (Aggregate function) on other columns.

- When WHERE, GROUP BY, and HAVING clauses are used together in a SELECT statement:
 1. The WHERE clause is processed first in order.
 2. Subsequently, the rows that are returned after the WHERE clause is executed are grouped based on the GROUP BY clause.
 3. Finally, any conditions, on the Group functions in the HAVING clause, are applied to the grouped rows before the final output is displayed.

ROLL UP Operation

- ROLLUP operation is used to obtain sub-totals in a query.
- ROLLUP operation is an extension to the GROUP BY clause in a query that produces sub-totals in addition to the regular grouped rows.

```
SELECT DNAME, JOB , SUM(SAL), AVG(SAL) FROM EMP A,  
(SELECT DNAME, DEPTNO FROM DEPT ) B  
WHERE A.DEPTNO = B.DEPTNO  
GROUP BY ROLLUP(DNAME, JOB) ;
```


- The CUBE function, introduced with Oracle8i:
 - presents all possible combinations of subtotals, including the sum total (asymmetrical aggregation)
 - allows very simple cross-classified table reports that are required in drill-down functionalities of client tools

```
SELECT DNAME, JOB , SUM(SAL), AVG(SAL) FROM EMP A,  
(SELECT DNAME, DEPTNO FROM DEPT ) B  
WHERE A.DEPTNO = B.DEPTNO  
GROUP BY CUBE (DNAME, JOB)
```

- With multi-dimensional analyses, the CUBE function generates all the subtotals that can be formed for a cube with the specified dimensions

DNAME	JOB	SUM(SAL)	AVG(SAL)
ACCOUNTING	CLERK	1300	1300
ACCOUNTING	MANAGER	3450	1725
ACCOUNTING	PRESIDENT	5000	5000
ACCOUNTING		9750	2437.5
OPERATIONS	MANAGER	6000	1500
OPERATIONS		6000	1500
RESEARCH	ANALYST	6000	3000
RESEARCH	CLERK	1900	950
RESEARCH	MANAGER	2975	2975
RESEARCH		10875	2175
SALES	CLERK	950	950
SALES	MANAGE	2850	2850
SALES	SALESMAN	5600	1400
SALES		9400	1566.66
ANALYST		6000	3000
CLERK		4150	1037.5
MANAGER		15275	1909.37
PRESIDENT		5000	5000
SALESMAN		5600	1400
		36025	1896.05

Tips and Tricks

- Suppose your SELECT statement contains a HAVING clause. Then write your query such that the WHERE clause does most of the work (removing undesired rows) instead of the HAVING clause doing the work of removing undesired rows.
 - **For example:** In a SELECT statement with WHERE, GROUP BY, and HAVING clauses, the query executes in the following sequence.
 - ❖ First, the WHERE clause is used to select the appropriate rows that need to be grouped.
 - ❖ Next, the GROUP BY clause divides the rows into sets of grouped rows, and then aggregates their values.
 - ❖ And last, the HAVING clause then eliminates undesired aggregated groups.

Tips and Tricks

- If the WHERE clause is used to eliminate as many of the undesired rows as possible, then the GROUP BY and the HAVING clauses will have to do less work. Thus boosting the overall performance of the query.
- Use the GROUP BY clause only with an Aggregate function, and not otherwise.
 - Since in other cases, you can accomplish the same end result by using the DISTINCT option instead, and it is faster.

DBMS SQL

Single-Row Functions

Single-Row Functions

- Single-row functions return a single result row for every row of a queried Table or View.
 - Single-row functions can appear in SELECT lists, WHERE clauses, START WITH and CONNECT BY clauses, and HAVING clauses.
 - Different categories of single valued functions are:
 - ❖ Numerical functions
 - ✓ ABS, CEIL, TRUNC, ROUND, POWER, etc.
 - ❖ Character functions
 - ✓ CONCAT, LPAD, RPAD, TRIM, SUBSTR, etc.
 - ❖ Date and Time functions
 - ✓ SYSDATE, ADD_MONTHS, LAST_DAY, etc.
 - ❖ Conversion functions
 - ✓ CAST, ASCIISTR, ROWIDTOCHAR, etc.

Number Functions

❖ Collection functions

❖ Miscellaneous Single-row functions

✓ BFILENAME, DECODE, NVL, NULLIF, USERENV, etc.

- Number functions accept “numeric data” as argument, and returns “numeric values”.

CEIL (arg)	Returns the smallest integer greater than or equal to “arg”.
FLOOR (arg)	Returns the largest integer less than or equal to “arg”.
ROUND (arg,n)	Returns “arg” rounded to “n” decimal places. If “n” is omitted, then “arg” is rounded as an integer.
POWER (arg, n)	Returns the argument “arg” raised to the n th power.
SQRT (arg)	Returns the square root of “arg”.
SIGN (arg)	Returns -1, 0, or +1 according to “arg” which is negative, zero, or positive respectively.
ABS (arg)	Returns the absolute value of “arg”.
MOD (arg1, arg2)	Returns the remainder obtained by dividing “arg1” by “arg2”.

TRUNC and ROUND

- **TRUNC - Example:**

- `trunc(125.815)` would return 125
- `trunc(125.815, 0)` would return 125
- `trunc(125.815, 1)` would return 125.8
- `trunc(125.815, 2)` would return 125.81
- `trunc(125.815, 3)` would return 125.815
- `trunc(-125.815, 2)` would return -125.81
- `trunc(125.815, -1)` would return 120
- `trunc(125.815, -2)` would return 100
- `trunc(125.815, -3)` would return 0

- **ROUND - Example**

- `round(125.315)` would return 125
- `round(125.315, 0)` would return 125
- `round(125.315, 1)` would return 125.3
- `round(125.315, 2)` would return 125.32
- `round(125.315, 3)` would return 125.315
- `round(-125.315, 2)` would return -125.32

- Example 1: **SELECT ABS(-15) "Absolute" FROM DUAL;**
- DUAL table is a table owned by SYS.
 - SYS owns the “data dictionary”, and DUAL is part of the data dictionary.
 - DUAL is a small work-table, which consists of only one row and one column, and contains the value “x” in that column. Besides arithmetic calculations, it also supports “date retrieval” and it’s “formatting”.
 - Often a simple calculation needs to be done. A SELECT must have a table name in it’s FROM clause, else it fails. To facilitate such calculations via a SELECT, the DUAL dummy table is provided.
 - The structure of the DUAL table can be viewed by using the SQL statement: **DESC DUAL**

Character Functions

- Character functions accept “character data” as argument, and returns “character” or “number” values.

LOWER (arg)	Converts alphabetic character values to lowercase.
UPPER (arg)	Converts alphabetic character values to uppercase.
INITCAP (arg)	Capitalizes first letter of each word in the argument string.
CONCAT (arg1, arg2)	Concatenates the character strings “arg1” and “arg2”.
SUBSTR (arg, pos, n)	Extracts a substring from “arg”, “n” characters long, and starting at position “pos”.
LTRIM (arg)	Removes any leading blanks in the string “arg”.
RTRIM (arg)	Removes any trailing blanks in the string “arg”.
LENGTH (arg)	Returns the number of characters in the string “arg”.
REPLACE (arg, str1, str2)	Returns the string “arg” with all occurrences of the string “str1” replaced by “str2”.
LPAD (arg, n, ch)	Pads the string “arg” on the left with the character “ch”, to a total width of “n” characters.
RPAD (arg, n, ch)	Pads the string “arg” on the right with the character “ch”, to a total width of “n” characters.

Date Functions

- Given below are a few Date functions:
 - **ADD_MONTHS**(DATE1,int1) : returns a DATE, int1 times months added.
 - **MONTHS_BETWEEN** (DATE1,DATE2):returns number of months between the two DATES.
 - **LASTDAY**(d):returns the date of the last day of the month . You might use this function to determine how many days are left in the current month.
 - **NEXT_DAY**(d, char): returns the date of the first weekday named by char that is later than the date d.
 - **SYSDATE** :returns the current DATE and TIME.
 - **TRUNC** (DATE1) :truncates the time part from the DATE.

Date Formats

- Given below are formats of Date functions:

Format	Meaning
YYYY	Four digit year
YY	Last two digits of the year
MM	Month (01-12 where JAN = 01 ...)
MONTH	Name of the month stored as a length of nine characters.
MON	Name of the month in three letter format.
DD	Day of the month (01-31).
D	Day of the week.
DAY	Name of the day stored as a length of nine characters.
DY	Name of the day in three letter format.
FM	This prefix can be added to suppress blank padding.
HH	Hour of the day.
HH24	Hour in the 24 hour format.
MI	Minutes of the hour.
SS	Seconds of the minute.
TH	The suffix used with the day.

Conversion Functions

- Conversion functions facilitate the conversion of values from one datatype to another.

TO_CHAR (arg)	Converts a number or date “arg” to a specific character format.
TO_DATE (arg)	Converts a date value stored as string to date datatype
TO_NUMBER (arg)	Converts a number stored as a character to number data type.

- Example:

```
SELECT empno, hiredate FROM Emp WHERE hiredate = TO_DATE
('September 08,1981','Month DD, YYYY');
```

NVL Function

- NVL() function:
 - Many times there are records holding NULL values in a table. When an output of such a table is displayed, it is difficult to understand the reason of NULL or BLANK values shown in the output.
 - The only way to overcome this problem is to replace NULL values with some other meaningful value while computing the records. This can be done using the NVL function.
 - **Example:** In the following example, the query returns a list of employee names and commissions, substituting "Not Applicable" whenever the employee receives no commission

```
SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable')  
"COMMISSION" FROM Emp;
```

Decode Function

- Decode () function:
 - Decode is a value-substitution mechanism that returns plain-English equivalents for a coded field.
 - It is a special function that is used to evaluate expression. It works like the IF-THEN-ELSE conditional loop.
 - One of the advantages of DECODE function is it's speed.
 - It is much faster to query using the DECODE keyword, than to perform a join to a LOOKUP table, especially when using large tables.
 - Syntax:


```
DECODE (<exp or coln>, <val1>,<o/p1>,<val2>,<o/p2>,...,<default o/p>)
```
 - Example:


```
SELECT empno, ename, deptno,
       DECODE (deptno,10,'Ten',20,'Twenty','Other')
FROM emp WHERE job like 'CLERK'
```

Tips and Tricks

- If possible, avoid the SUBSTRING function in the WHERE clauses.
 - Depending on how it is constructed, using the SUBSTRING function can force a table scan instead of allowing the Optimizer to use an Index (assuming there is one). Instead, use the LIKE condition, for better performance.
 - If the substring you are searching for does not include the first character of the column you are searching for, then a table scan is performed.

DBMS SQL

Data Definition Language

Basic Datatypes

- Given below are the basic Data Types:

Datatype	Description
CHAR(n)	To store fixed length string. Maximum length = 2000 bytes For example: NAME CHAR(15)
VARCHAR2(n)	To store variable length string. Maximum length = 4000 bytes For example: DESCRIPTION VARCHAR2(100)
LONG(n)	To store variable length string . Maximum length = 2 GIGA bytes For example: SYNOPSIS LONG(5000)
NUMBER(p,s)	To store numeric data . Range is 1E-129 to 9.99E125 Max Number of significant digits = 38 For example: SALARY NUMBER(9,2)
DATE	To store DATE. Range from January 1, 4712 BC to December 31, 9999 AD. Both DATE and TIME are stored. Requires 7 bytes. For example: HIREDATE DATE
RAW(n)	To store data in binary format such as signature, photograph. Maximum size = 255 bytes
LONG RAW(n)	Same as RAW. Maximum size = 2 Gigabytes

Create TABLE

- Tables are objects, which store the user data.
- Use the CREATE TABLE statement to create a table, which is the basic structure to hold data.

```
CREATE TABLE DEPT ( Deptno number(2),Dname varchar2(14),Loc  
varchar2(13) );
```

- Let us see the types of Data Integrity:

- Nulls

- ❖ NOT NULL constraints for the rules associated with nulls in a column.

- “Null” is a rule defined on a single column that allows or disallows, inserts or updates on rows containing a “null” value (the absence of a value) in that column.

- Unique Column Values

- ❖ UNIQUE key constraints for the rule associated with unique column values. A “unique value” rule defined on a column (or set of columns) allows inserting or updating a row only if it contains a “unique value” in that column (or set of columns).

➤ Primary Key Values

- ❖ PRIMARY KEY constraints for the rule associated with primary identification values. A “primary key” value rule defined on a key (a column or set of columns) specifies that “each row in the table can be uniquely identified by the values in the key”.

➤ Referential Integrity

- ❖ FOREIGN KEY constraints for the rules associated with referential integrity. A “Referential Integrity” rule defined on a key (a column or set of columns) in one table guarantees that “the values in that key, match the values in a key in a related table (the referenced value)”. Oracle currently supports the use of FOREIGN KEY integrity constraints to define the referential integrity actions, including:

Data Integrity

- update and delete No Action
 - delete CASCADE
 - delete SET NULL
-
- CHECK constraints for complex integrity rules

- The user will not be allowed to enter null value.
 - Example: **CREATE TABLE EMP(empno number(4) not null, ename varchar2(10) not null,);**
 - A NULL value is different from a blank or a zero. It is used for a quantity that is “unknown”.
 - A NULL value can be inserted into a column of any data type that allows or disallows, inserts or updates on rows containing a “null” value (the absence of a value) in that column
 - A NULL value will evaluate to NULL in any expression
 - ❖ For example: NULL added to 10 is NULL.
 - If the column has a NULL value, Oracle ignores any UNIQUE, FOREIGN KEY, and CHECK constraints that may be attached to the column.

- If no value is given, then instead of using a “Not Null” constraint, it is sometimes useful to specify a default value for an attribute.

➤ For example:

```
CREATE TABLE EMP(empno number(4),ename varchar2(10),job  
  varchar2(9) default('CLERK'),mgr number(4),hiredate date,sal  
  number(7,2), comm number(7,2),  
  deptno number(2));
```


UNIQUE Constraint

- The UNIQUE constraint does not allow duplicate values in a column.
 - If the UNIQUE constraint encompasses two or more columns, then two equal combinations are not allowed.
 - However, if a column is not explicitly defined as NOT NULL, then NULLS can be inserted multiple number of times.
 - For example:

```
CREATE TABLE DEPT (Deptno number(2),Dname varchar2(14)
    unique,Loc varchar2(13) );
```
- A UNIQUE constraint can be extended over multiple columns.
- A “unique value” rule defined on a column (or set of columns) allows inserting or updating a row only if it contains a “unique value” in that column (or set of columns).

PRIMARY KEY CONSTRAINT

- A PRIMARY KEY combines a UNIQUE constraint and a NOT NULL constraint.
- Additionally, a table can have at the most one PRIMARY KEY.
- After creating a PRIMARY KEY, it can be referenced by a FOREIGN KEY.
- A “primary key” value rule defined on a key (a column or set of columns) specifies that “each row in the table can be uniquely identified by the values in the key”.

➤ For example:

```
CREATE TABLE EMP (empno number(4) constraint pk_emp primary  
key, .....);
```

CHECK Constraint

- CHECK constraint allows users to restrict possible attribute values for a column to admissible ones.
 - For example: `CREATE TABLE EMP (empno number(4) primary key, ename varchar2(40) constraint check_name check(ename = upper(ename)),);`
- A CHECK constraint **cannot** include a SUBQUERY.
- The condition in CHECK constraint must be a Boolean expression evaluated by using the values in the row being inserted or updated.
- A CHECK constraint cannot contain sub-queries, sequence, the SYSDATE, UID, USER, or USERENV SQL functions

FOREIGN KEY Constraint

- The FOREIGN KEY constraint specifies a “column” or a “list of columns” as a foreign key of the referencing table.
- The referencing table is called the “child-table”, and the referenced table is called “parent-table”.
 - **FOREIGN KEY:** Defines the column in the child table at the table constraint level.
 - **REFERENCES:** Identifies the table and column in the parent table.
 - **ON DELETE CASCADE:** Deletes the dependent rows in the child table when a row in the parent table is deleted.
 - **ON DELETE SET NULL:** Converts dependent FOREIGN KEY values to NULL.
 - For example: **CREATE TABLE EMP (empno number(4) constraint pk_emp primary key,, deptno number(3) constraint fk_deptno references dept(deptno));**

Viewing the Constraints

- You can query the USER_CONSTRAINTS table to view all constraint definitions and names.
- You can view the columns associated with the constraint names in the USER_CONS_COLUMNS view.

Example

- The CREATE TABLE example with some constraints as:

```
CREATE TABLE emp (empno NUMBER(4) CONSTRAINT P_KEY PRIMARY
KEY,  ename VARCHAR2(10) CONSTRAINT ENAME_NOT_NULL NOT
NULL,  deptno NUMBER(2),  job CHAR(9) CONSTRAINT JOB_ALL_UPPER
CHECK(job = UPPER(job)),  hiredate DATE DEFAULT SYSDATE );
```

- Creating new table based on an existing table:

```
CREATE TABLE NewEmp(Emp#,Emp_Name,Hire_Date) AS ( SELECT
empno,ename,hiredate FROM emp WHERE hiredate >'01-jan-82' );
```

- Constraints on an old table will not be applicable for the new table except NOT NULL constraint.

ALTER Table

- The ALTER TABLE statement will, by default, check all data in the table for compliance. This can take a long time for large tables.
- The ALTER TABLE statement will fail if there is invalid data, i.e. data in the column that does not adhere to the constraint.
- Syntax:

```
ALTER TABLE table_name[ADD (col_name col_datatype
col_constraint ,...)]|[ADD (table_constraint)]|[DROP CONSTRAINT
constraint_name]|[MODIFY existing_col_name new_col_datatype
new_constraint new_default] [DROP COLUMN existing_col_name]
[SET UNUSED COLUMN existing_col)name];
```

- A column cannot be removed from an existing table using ALTER TABLE.

ALTER Table

- The uses of modifying columns are:
 - ❖ Can increase the width of a character column, any time.
 - ❖ Can increase the number of digits in a number, any time.
 - ❖ Can increase or decrease the number of decimal places in a number column, any time. Any reduction on precision and scale can be on empty columns only.
 - ❖ Can add only NOT NULL constraint by using Column constraints. All other constraints have to be specified as Table constraints.
- The “Add” keyword is used to add a column or constraint to an existing table.

```
ALTER TABLE emp ADD (sal NUMBER(7,2) CONSTRAINT SAL_GRT_0
CHECK(sal >0), mgr NUMBER(4), comm NUMBER(9,2));
```


ALTER Table

- The uses of adding a column to the table by using Add clause are:
 - ❖ You can add any column without a NOT NULL specification.
 - ❖ You can add a NOT NULL column in three steps:
 1. Add a column without NOT NULL specification.
 2. Fill every row in that column with data.
 3. Modify the column to be NOT NULL.
- For adding Referential Integrity on “mgr” column, refer the following example:

```
ALTER TABLE emp ADD CONSTRAINT FK FOREIGN KEY (mgr)  
REFERENCES emp (empno);
```

ALTER Table

- **MODIFY clause:**

- The “Modify” keyword allows making modification to the existing columns of a table.

ALTER TABLE Emp MODIFY (Sal NUMBER (8,2));

- **ENABLE | DISABLE Clause:**

- The ENABLE | DISABLE clause allows constraints to be enabled or disabled according to the user choice without removing them from a table.

ALTER TABLE Emp DISABLE CONSTRAINT SYS_C000934;

ALTER Table

- The use of modifying the columns with the Enable | Disable clause are:
 - ❖ Can increase “column width” of a character any time.
 - ❖ Can increase the “number of digits” in a number at any time.
 - ❖ Can increase or decrease the “number of decimal places” in a number column at any time. Any reduction on “precision” and “scale” can only be on empty columns.
 - ❖ Can only add the NOT NULL constraint by using “column constraints”. Rest all other constraints have to be specified as “table constraints”.

ALTER Table

- During large transactions involving multiple dependencies, it is often difficult to efficiently process data due to the restrictions imposed by the constraints. An example of this will be the update of a PRIMARY KEY (PK), which is referenced by FOREIGN KEYS (FK).
- The primary key columns cannot be updated as this will orphan the dependent tables, and the dependent tables cannot be updated prior to the parent table as this will also make them orphans.
- Traditionally, this problem was solved by disabling the FOREIGN KEY constraints, or deleting the original records and recreating them. Since neither of these solutions is particularly satisfactory, Oracle 8i includes support for DEFERRED constraints. A DEFERRED constraint is only checked at the point when the transaction is committed.
- By default, constraints are created as NON DEFERRABLE. However, this can be overridden by using the DEFERRABLE keyword.

ALTER Table

- The DROP clause is used to remove constraints from a table.

- For Dropping the FOREIGN KEY constraint on “mgr”, refer the following example:

ALTER TABLE Emp DROP constraint Emp_FK

or

ALTER TABLE Emp DROP constraint SYS_C000934

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

ALTER Table

➤ To remove the PRIMARY KEY constraint on the DEPT table and drop the associated FOREIGN KEY constraint on the EMP.DEPTNO column.

- Given below are the ways for “Dropping” a column:

Marking the columns as unused and then later dropping them.

```
ALTER TABLE emp SET UNUSED COLUMN comm;
```

```
ALTER TABLE emp SET UNUSED (sal, hiredate);
```

The following command can be used later to permanently drop the columns.

```
ALTER TABLE emp DROP UNUSED COLUMNS;
```

- Columns once marked as unused cannot be recovered.
- Marking the columns as unused does not release the space occupied by them back to the database.
- The advantage of the marking column as “unused” and then dropping them is that marking the columns is much faster process than dropping the columns.

ALTER Table

You can refer to the data dictionary table USER_UNUSED_COL_TABS to get information regarding the tables with columns marked as unused.

2. Directly dropping the columns.

```
ALTER TABLE drop COLUMN sal;
```

- By default, when a Primary Key or unique constraint is disabled, its associated index is dropped, as well. This behavior can be changed by using the KEEP INDEX clause as shown in the following query:

```
ALTER TABLE drop primary key KEEP INDEX;
```

ALTER Table

- The DROP TABLE command is used to remove the definition of a table from the database.

DROP TABLE Emp;

DROP TABLE Dept CASCADE CONSTRAINTS;

- A table that is dropped cannot be recovered. When a table is dropped, dependent objects such as indexes are automatically dropped. Synonyms and views created on the table remain, but give an error if they are referenced.
- You cannot delete a table that is being referenced by another table.
- To view the names of tables owned by the user
SELECT table_name FROM user_tables

ALTER Table

- To view distinct object types owned by the user,
`SELECT DISTINCT object_type FROM user_objects ;`

Index

- Index is a database object that functions as a “performance-tuning” method for allowing faster retrieval of records.
- Index creates an entry for each value that appears in the indexed columns.
- The absence or presence of an Index does not require change in wording of any SQL statement.
- There are different types of Indexes, which can be created by the user.
- A “Table” can have any number of “Indexes”.
- An Index can be on a single column or on multiple columns.
 - In Oracle 8i, up to 32 columns can be included in one Index.
- Updation of all Indexes is handled by RDBMS.
- UNIQUE ensures that all rows in a Table are unique.
- An Index is in an ascending order, by default.

Index

- The RDBMS decides when to use the Index while accessing the data.
- Removal of an Index does not affect the Table on which it is based.
- When you create a PRIMARY or a UNIQUE constraint on the “Table”, a UNIQUE index is automatically created.

➤ The name of the constraint is used for the Index name.

```
CREATE [UNIQUE] INDEX index_name
ON table_name(col_name1 [ASC | DESC], col_name2,.....)
```

- Example

```
CREATE INDEX emp_sal_index ON emp(sal);
```

- Unique Index

```
CREATE UNIQUE INDEX emp_ename_unindex ON emp( ename );
```

- More Indexes on a Table does not mean faster queries.
 - Each DML operation that is committed on a Table with Indexes imply that the Indexes must be updated.
 - The more number of Indexes are associated with a Table, the more effort the Oracle server must make to update all the Indexes after a DML operation.
- You should create Indexes, only if:
 - the column contains a wide range of values
 - the column contains a large number of NULL values
 - one or more columns are frequently used together in a WHERE clause or join condition
 - the table is large and most queries are expected to retrieve less than 2-4% of the rows
- If you want to enforce uniqueness, you should define a UNIQUE

- constraint in the Table definition. Then a “unique index” will be automatically created.
- It is usually not worth creating an Index, if:
 - The Table is small.
 - The columns are not often used as a condition in the query.
 - Most queries are expected to retrieve more than 2 to 4 percent of the rows in the table.
 - The Table is frequently updated.
 - The indexed columns are referenced as part of an expression.

Types of Index

- Indexes can be either created “automatically” or “manually”.
 - **Automatically:** A unique Index is automatically created when you define a PRIMARY KEY or UNIQUE constraint in a table definition. The name of the Index is same as the name given to the constraint.
 - **Manually:** A non-unique index can be created on columns by users in order to speed up access to the rows.
 - ❖ **For example:** You can create a FOREIGN KEY column index for a “join” in a query in order to improve retrieval speed.

Bitmap Index

- In a Bitmap Index, a “bitmap” is used for each “key value” instead of a list of “rowids”.
 - A regular index does not store columns that contain NULL.
 - Oracle 8i, allows you to create Bitmap indexes in which you can store columns containing NULL.

```
CREATE BITMAP INDEX emp_idx ON emp(comm);
```

- The Bitmap Index is useful for some types of SQL statements like:

```
SELECT COUNT(*) FROM emp;
```

```
SELECT COUNT(*) FROM emp WHERE comm IS NULL;
```

```
CREATE BITMAP INDEX emp_idx ON emp(comm);
```

- Bitmap Indexes are used to tune queries that use non-selective columns in their limiting conditions.

Bitmap Index

- A Bitmap Index can have a maximum of 30 columns.
- In Bitmap structures, a “two-dimensional array” is created with one column for every row in the table, which is indexed.
 - Each column represents a distinct value within the bitmapped index. This two-dimensional array represents “each value within the index” multiplied by the “number of rows” in the table.
 - In the “row retrieval” stage, the Oracle decompresses the bitmap into the RAM data buffers, such that it can be rapidly scanned for matching values.
 - These matching values are delivered to Oracle in the form of a Row-ID list.
 - These Row-ID values may directly access the required information.
- The real benefit of bitmapped indexing occurs when “one

Bitmap Index

- table” includes “multiple bitmapped indexes”.
 - Each individual column may have low cardinality.
- The creation of multiple bitmapped indexes provides a very powerful method for rapidly answering difficult SQL queries.
 -

Function-Based index

- A Function-based Index computes the value of the “function” or “expression” and stores it in the Index.
- To create a Function-based Index:
 - the user should have an “Query Rewrite” privilege, or
 - the system parameter QUERY_REWRITE_ENABLE should be set to TRUE
- Function-based Indexes allow creation of indexes in PL/SQL and Java, on their:
 - Expressions
 - Internal functions
 - User-written functions
- Function-based indexes ensure that the Oracle designer is able to use an index for it’s query.
- Prior to Oracle8, the usage of a “built-in” function was not

Function-Based index

- able to match the performance of an “Index”. Consequently, Oracle performed the dreaded full-table scan.
- Examples of SQL with Function-based queries include the following:
 - Select * from customer where substr(cust_name,1,4) = 'BURL';
 - Select * from customer where to_char(order_date,'MM') = '01';
 - Select * from customer where upper(cust_name) = 'JONES';
 - Select * from customer where initcap(first_name) = 'Mike';
- In Oracle, the dbms always interrogates the WHERE clause of the SQL statement to check the existence of a matching Index.
 - By using Function-based Indexes, the Oracle designer can create a matching index, which exactly matches the predicates within the SQL WHERE clause.

Function-Based index

- This ensures that the query is retrieved with a minimal amount of disk I/O at the fastest possible speed.
- Example:
`CREATE INDEX uppercase_idx ON emp (UPPER(empname));`
- The USER_INDEXES data dictionary view contains the name of the index and its uniqueness.
- The USER_IND_COLUMNS view contains the index name, the table name, and the column name.

Synonym

- A “Synonym” is an “alias” that is used for any table, view, materialized view, sequence, procedure, function, or package.
 - Since a Synonym is simply an alias, it does not require storage except for storage of it’s definition in the data dictionary.
 - Synonyms are often used for “security” and “convenience”.
 - Synonyms can be created as either “public” or “private”.
 - Synonyms are useful in hiding ownership details of an object.
- A Synonym simplifies access to objects. A Synonym is simply another name for an object.
- With Synonyms, you can:
 - Ease referring to a table owned by another user.
 - Shorten lengthy object names.

Synonym

- Syntax

CREATE [PUBLIC] SYNONYM another_name FOR existing_name

➤ where:

- ❖ Existing_name is the name of a table, view, or sequence.

- ❖ PUBLIC is used to grant permission to all users for accessing the object by using the new name. (This is done only by a DBA.)

- Example:

CREATE SYNONYM e FOR Emp;

- Dropping a Synonym

DROP SYNONYM e;

Sequence

- A “Sequence” is an object, which can be used to generate sequential numbers.
- A Sequence is used to fill up columns, which are declared as UNIQUE or PRIMARY KEY.
- A Sequence uses “NEXTVAL” to retrieve the next value in the sequence order.
- A Sequence:
 - automatically generates unique numbers.
 - is a “sharable object”.
 - is typically used to create a PRIMARY KEY value.
 - replaces application code.
 - speeds up the efficiency of accessing sequence values when cached in memory.

Sequence

- Syntax:

```
CREATE SEQUENCE seq_name
[INCREMENT BY n1] [START WITH n2]
[MAXVALUE n3] [MINVALUE n4] [CYCLE | NOCYCLE]
[CACHE | NOCACHE];
```

➤ Where

- ❖ START WITH indicates the first NUMBER in the series.
- ❖ INCREMENT BY is the difference between consecutive numbers. If n1 is negative, then the numbers that are generated are in a descending order.
- ❖ MAXVALUE and MINVALUE indicate the extreme values.
- ❖ CYCLE indicates that once the extreme is reached, it starts the cycle again with n2.
- ❖ NOCYCLE means that once the extreme is reached, it stops generating numbers.

- ❖ CACHE caches the specified number of sequence values in the memory. This speeds access, but all cached numbers are lost when the database is shut down. The default value is 20

- Example:

```
CREATE SEQUENCE s1  
  INCREMENT BY 1  
  START WITH 1  
  MAXVALUE 10000  
  NOCYCLE ;
```

- s1 will generate numbers 1,2,3.....,10000, and then stop
- Confirming a Sequence

```
SELECT sequence_name, min_value, max_value, increment_by,  
       last_number  
FROM user_sequences;
```

Referencing Sequence

- After you create a Sequence, it generates sequential numbers that can be used in your tables. You can reference the Sequence values by using the NEXTVAL and CURRVAL pseudocolumns.
- **NEXTVAL and CURRVAL Pseudocolumns:**
 - The **NEXTVAL** pseudocolumn is used to extract successive sequence numbers from a specified Sequence. You must qualify NEXTVAL with the sequence name. When you reference sequence.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.
 - The **CURRVAL** pseudocolumn is used to refer a Sequence number that the current user has just generated.
 - NEXTVAL must be used to generate a sequence number in the session of the current user, before CURRVAL can be referenced.

Referencing Sequence

- You must qualify CURRVAL with the sequence name.
- When “sequence.CURRVAL” is referenced, the last value returned to that user’s process is displayed.
- To insert a record using sequence:
`INSERT into emp(empno,ename,deptno) VALUES(S1.NEXTVAL,'XYZ',10)`
- To view the current value for Sequence
`SELECT S1.CURRVAL FROM dual;`

Uses of Sequence

- Usage of a Sequence:
 - Caching the Sequence values in memory to give faster access to those Sequence values
 - ❖ The first time you refer the Sequence, the cache is populated.
 - ❖ Each request for the next Sequence value is retrieved from the cached sequence.
 - ❖ After the last sequence value is used, the next request for the Sequence pulls another cache of sequences into the memory.
 - ❖ Gaps in Sequence values can occur when:
 - ✓ a rollback occurs(Although “sequence generators” issue “sequential numbers” without gaps, this action occurs independent of a **commit** or **rollback**.
Therefore, if you roll back a statement containing a Sequence, the number is lost.)
 - ✓ the system crashes(Another event that can cause gaps in the Sequence is a

Uses of Sequence

- ✓ **system crash.** If the Sequence caches the values in memory, then those values are lost if there is a system crash.)
- ✓ a Sequence is used in another table (Since Sequences are not directly tied to tables, the same Sequence can be used for multiple tables. If you do so, each table can contain gaps in the Sequential numbers.)
- Viewing the next available value by querying the USER_SEQUENCES table, when the sequence is created with NOCACHE
 - ❖ If the Sequence is created with NOCACHE, it is possible to view the next available Sequence value without incrementing it by querying the USER_SEQUENCES table.

Dropping a Sequence

- A Sequence can be removed from the data dictionary by using the DROP SEQUENCE statement.
 - Once removed, the Sequence can no longer be referenced.
 - To remove the Sequence, you must be the owner of the Sequence or possess the DROP ANY SEQUENCE privilege.
- Example:
DROP SEQUENCE s1;

Altering a Sequence

- Syntax:
 ALTER SEQUENCE s1
 INCREMENT BY n1
 MAXVALUE n3 CYCLE;
- To make the Sequence start from a new number, drop the Sequence and create it again.
- ALTER SEQUENCE has no effect on numbers that are already generated.
- **Guidelines for modifying Sequences:**
 - You must be the owner or have the ALTER privilege for the Sequence.
 - Only future sequence numbers are affected.
 - The Sequence must be dropped, and re-created to restart the Sequence at a different number.
 - Some validation should be performed.

- A View can be thought of as a “stored query” or a “virtual table”, i.e. a logical table based on one or more tables.
 - A View can be used as if it is a table.
 - A View does not contain data.
- Views are used:
 - To restrict data access.
 - To make complex queries easy.
 - To provide data independence.
 - To present different views of the same data.
- Syntax:

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


View

- Characteristics of a View

- View is a logical table that is based on one or more Tables.
- View can be used as if it is a Table. View does not contain data.
- Whenever a View is accessed, the query is evaluated. Thus a View is dynamic.
- Any changes made in the View affects the Tables on which the View is based.
- View helps to hide the following from the user:
 - ❖ Ownership details of a Table, and
 - ❖ Complexity of the query used to retrieve the data
- "With check option" implies you cannot insert a row in the underlying Table, which cannot be selected by using the View.
- If you use a ORDER BY clause in the View, then it automatically becomes a "Read-only View".

- Example:

```
CREATE VIEW new_view AS
```

```
SELECT * FROM emp WHERE hiredate >'01-jan-82';
```

- Updating / Inserting rows in the base table is possible by using Views, however, with some restrictions. This is possible only if the View is based on a single table.
- The restrictions are :
 - Updation / Insertion not possible if View is based on two tables. However, this can be done in ORACLE 8.
 - Insertion is not allowed if the underlying table has any NOT NULL columns, which are not included in the View.
 - Insertion / Updation is not allowed if any column of the View referenced in UPDATE / INSERT contains “functions” or “calculations”.
 - Insertion / Updation / Deletion is not allowed if View contains GROUP BY or DISTINCT clauses in the query.

View

- Modifying a View
- Example:

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

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

- Creating a Read-only View:

- If a View is based on a single table, the user may manipulate records in the View, and it's underlying base table.
- The WITH READ ONLY clause of the CREATE VIEW command can be used to prevent users from manipulating records in a View.

```
CREATE VIEW new_view
```

```
AS
```

```
SELECT * FROM emp
```

```
WHERE hiredate >'01-jan-82' With read only
```

- Creating a View with WITH CHECK OPTION:
 - WITH CHECK OPTION Specifies that only the rows accessible to the view can be inserted or updated.

```
CREATE VIEW emp_view  
AS SELECT      * FROM  employees  
WHERE  deptno =10 WITH CHECK OPTION constraint cn;
```
 - Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.
- You can perform “DML operations” on simple Views.
- You cannot remove a row if the View contains the following:
 - Group functions
 - A GROUP BY clause
 - The DISTINCT keyword
 - The pseudocolumn ROWNUM keyword

- Dropping a View

- **DROP VIEW *emp_view*;**

- Inline View

- “Inline view” is not a schema object like a regular View. However, it is a temporary query with an alias.

- You can use ORDER BY clause in the Inline View. This is very useful when you want to find the top n values in a table.

```
SELECT DNAME, ENAME, SAL FROM EMP A,  
(SELECT DNAME, DEPTNO FROM DEPT ) B  
WHERE A.DEPTNO = B.DEPTNO
```

- You can use Order By clause, as well, in the Inline View.

```
select rownum, ename, job, sal from (select ename, job, sal from emp  
order by sal desc) where rownum < 5;
```

Deleting Objects

- Syntax:
`DROP Obj_Type obj_name;`
- Example:
`DROP TABLE emp;`
- A table, which is dropped, cannot be recovered.
- Dependent objects such as Indexes are automatically dropped.
- Synonyms and Views remain intact. However, they give an error when referenced.
- If a Synonym, then the Table is not affected in any way. Only the duplicate name is removed.

DBMS SQL

Data Manipulation Language

- Data Manipulation Language (DML) is used to perform the following routines on database information:
 - Retrieve
 - Insert
 - Modify
- DML changes data in an object. If you insert a row into a table, that is DML.
- All DML statements change data, and must be committed before the change becomes permanent.

INSERT Command

- **INSERT command:**
 - INSERT is a DML command. It is used to add rows to a table.
 - In the simplest form of the command, the values for different columns in the row to be inserted have to be specified.
 - Alternatively, the rows can be generated from some other tables by using a SQL query language command.
- **Requisites for using INSERT command:**
 - If values are specified for all columns in the order specified at creation, then column_names could be omitted.
 - Values should match “data type” of the respective columns.
 - Number of values should match the number of column names mentioned.
 - All columns declared as NOT NULL should be supplied with a value.

INSERT Command

- Character strings should be enclosed in quotes.
- Date values should be enclosed in quotes.
- Values will insert one row at a time.
- Query will insert all the rows returned by the query.
- The table_name can be a “table” or a “view”. If table_name is a “view”, then the following restrictions apply:
 1. The “view” cannot have a GROUP BY, CONNECT BY, START WITH, DISTINCT, UNION, INTERSECT, or MINUS clause or a join.
 2. If the “view” has WITH CHECK OPTION clause, then a row, which will not be returned by the view, cannot be inserted.

Syntax:

```
INSERT INTO table_name[(col_name1,col_name2,...)] {VALUES  
(value1,value2,...) | query};
```

INSERT Command

- Inserting rows in a table from another table using Subquery:

```
INSERT INTO new_emp_table
```

```
SELECT * FROM emp WHERE emp.hiredate > '01-jan-82';
```

- Inserting by using “substitution variables”:

```
INSERT INTO Dept VALUES (&deptno, '&dname', '&loc');
```

- The problem with the INSERT statement is that it adds only “one row” to the table.
- However, by using “substitution variables” the speed of data input can be increased.
- Whenever a “substitution variable” is placed in a “value” field, the user will be prompted to enter the “actual value” when the command is executed.

DELETE Command

- The DELETE command is used to delete one or more rows from a table.
- Syntax: **DELETE [FROM] {table_name | alias }
[WHERE cond.];**
 - The table_name can be a “table” or a “view”.
 - The DELETE command is used to delete one or more rows from a table.
 - The DELETE statement removes all rows identified by the WHERE clause.
 - ❖ This is another DML, which means we can rollback the deleted data, and that to make our changes permanent.
 - If WHERE clause is omitted, all rows from the table are removed. Else all rows which satisfy the condition are removed.
 - FROM clause can be omitted without affecting the statement.

UPDATE Command

- Use the UPDATE command to change single rows, groups of rows, or all rows in a table.
 - In all data modification statements, you can change the data in only “one table at a time”.
 - Syntax:


```
UPDATE table_name SET col_name = value |
                    col_name = SELECT_statement_returning_single_value |
                    (col_name,...) = SELECT_statement
                    [WHERE condition];
```
- The UPDATE command provides automatic navigation to the data.
 - **Note:** If the WHERE clause is omitted, all rows in the table will be updated by a value that is currently specified for the field. Else only those rows which satisfy the condition will be updated.

UPDATE Command

- Example:

UPDATE Dept

SET dname= 'Information Technology' , loc= 'California'

WHERE deptno=10;

UPDATE emp

SET SAL = (SELECT SAL FROM emp

WHERE empno = 7369)

WHERE ename = 'SMITH';

MERGE statement

- The MERGE statement, provides the ability to conditionally update or insert data into a database table.
- The MERGE statement, performs an UPDATE if the row exists, and an INSERT if it is a new row:
 - Increases performance and ease of use
 - Is useful in data warehousing applications
 - Avoids separate updates
- Syntax:

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


MERGE Statement

- **INTO** : This is how we specify the target for the MERGE. The target must be either a table or an updateable view (an in-line view cannot be used here);
- **USING** : the USING clause represents the source dataset for the MERGE. This can be a single table (as in our example) or an in-line view;
- **ON ()** : the ON clause is where we supply the join between the source dataset and target table. Note that the join conditions must be in parentheses;
- **WHEN MATCHED**: this clause is where we instruct Oracle on what to do when we already have a matching record in the target table (i.e. there is a join between the source and target datasets). We obviously want an UPDATE in this case. One of the restrictions of this clause is that we cannot update any of the columns used in the ON clause

MERGE Statement

- (though of course we don't need to as they already match). Any attempt to include a join column will raise an unintuitive invalid identifier exception; and
- **WHEN NOT MATCHED** : this clause is where we INSERT records for which there is no current match.

create table e as

```
select empno, ename, deptno from emp where 1=2;
```

```
merge into e using emp on (emp.deptno=e.deptno)
```

```
when matched then
```

```
update set empno=emp.empno, ename=emp.ename
```

```
when not matched then
```

```
insert(empno,ename) values (emp.empno,emp.ename);
```

DBMS SQL

Transaction Control Language

Transaction

- A “transaction” is a logical unit of work that contains one or more SQL statements. “Transaction” is an atomic unit.
 - The effects of all the SQL statements in a transaction can be either:
 - ❖ all committed (applied to the database), or
 - ❖ all rolled back (undone from the database)
 - A “transaction” begins with the first executable SQL statement.
 - A “transaction” ends when any of the following occurs:
 - ❖ A user issues a COMMIT or ROLLBACK statement without a SAVEPOINT clause.
 - ❖ A user runs a DDL statement such as CREATE, DROP, RENAME, or ALTER.
 - ✓ If the current transaction contains any DML statements, Oracle first commits the transaction, and then runs and commits the DDL statement as a new, single statement transaction.

- ❖ A user disconnects from Oracle. The current transaction is committed.
- ❖ A user process terminates abnormally. The current transaction is rolled back.
- ❖ After one transaction ends, the next executable SQL statement automatically starts the subsequent transaction.

- **Statement Execution and Transaction Control:**

- Executing successfully means that a single statement was:
 - ❖ Parsed
 - ❖ Found to be a valid SQL construction
 - ❖ Run without error as an atomic unit.
- For example: All rows of a multi-row update are changed.

Committing a Transaction

- A “SQL statement” that runs successfully is different from a committed transaction.
- However, until the “transaction” that contains the “statement” is **committed**, the “transaction” can be rolled back. As a result, all the changes in the statement can be undone.
- Committing a transaction means making “permanent” all the changes performed by the SQL statements within the transaction. This can be done either explicitly or implicitly.

- COMMIT statement makes “permanent” all the changes that are performed in the current transaction.

Rolling Back a Transaction

- Rolling back a transaction means “undoing changes” to data that have been performed by SQL statements within an “uncommitted transaction”.
 - Oracle uses “undo tablespaces” (or rollback segments) to store old values.
 - Oracle also uses the “redo log” that contains a record of changes.
- A SQL statement that fails causes a loss only of any work it would have performed by itself.
 - ❖ It does not cause the loss of any work that preceded it in the current transaction.
 - ❖ If the statement is a DDL statement, then the implicit commit that immediately preceded it is not undone.

➤ Types of Roll back:

- ❖ Statement-level rollback (due to statement or deadlock execution error)
- ❖ Rollback to a savepoint
- ❖ Rollback of a transaction due to user request
- ❖ Rollback of a transaction due to abnormal process termination
- ❖ Rollback of all outstanding transactions when an instance terminates abnormally
- ❖ Rollback of incomplete transactions during recovery

- In rolling back an entire transaction, without referencing any savepoints, there is an occurrence of the following sequence:
1. Oracle undoes all changes made by all the SQL statements in the transaction by using the corresponding undo tablespace.
 2. Oracle releases all the locks of data for the transaction.
 3. The transaction ends.

- **Statement-Level Rollback**

- If at any time during execution, a SQL statement causes an error, then all effects of the statement are rolled back.
 - ❖ The effect of the rollback is as if that statement had never been run.

This operation is a statement-level rollback.
- Errors discovered during SQL statement execution cause statement-level rollbacks.
 - ❖ For example: Attempting to insert a duplicate value in a primary key.
- Single SQL statements involved in a deadlock (competition for the same data) can also cause a statement-level rollback.
- Errors discovered during SQL statement parsing, such as a syntax error, have not yet been run, so they do not cause a statement-level rollback.

Savepoints

- In a transaction, you can declare intermediate markers called “savepoints” within the context of a transaction.
 - By using “savepoints”, you can arbitrarily mark your work at any point within a long transaction.
 - In this manner, you can keep an option that is available later to roll back the work performed, however:
 - ❖ before the current point in the transaction, and
 - ❖ after a declared savepoint within the transaction
 - Savepoints are useful in application programs, as well. If a procedure contains several functions, then you can create a savepoint at the beginning of each function.
 - Then, if a function fails, it is easy:
 - to return the data to it’s state before the function began, and
 - to re-run the function with revised parameters or perform a RECOVERY action.

- After a rollback to a savepoint, Oracle releases the data locks obtained by rolled back statements. As a result:
 - Other transactions that were waiting for the previously locked resources can proceed.
 - Other transactions that want to update previously locked rows can do
- When a transaction is rolled back to a savepoint, there is an occurrence of the following sequence:
 1. Oracle rolls back only the statements run after the savepoint.
 2. Oracle preserves the specified savepoint, but all savepoints that were established after the specified savepoint are lost.
 3. Oracle releases all table and row locks acquired since that savepoint, however, it retains all data locks acquired previous to the savepoint.
- The transaction remains active and can be continued.
 - **For example:** You can use savepoints throughout a long complex series of updates. So if you make an error, you do not need to resubmit every statement.

- Example

```
INSERT INTO dept VALUES (70, 'PERSONNEL', 'CALCUTTA') ;  
SAVEPOINT A ;  
INSERT INTO dept VALUES (80, 'MARKETING', 'BOMBAY') ;  
SAVEPOINT B ;  
ROLLBACK TO A ;
```

- **Advantages of COMMIT and ROLLBACK statements:**

- With COMMIT and ROLLBACK statements, you can:
 - ❖ ensure data consistency
 - ❖ preview data changes before making changes permanent
 - ❖ group logically related operations

- **State of the Data after COMMIT:**

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

- **State of the Data after ROLLBACK:**
 - DBMS discards all pending changes by using the ROLLBACK statement:
 - Data changes are undone.
 - Previous state of the data is restored.
 - Locks on the affected rows are released
- **Example of rolling back changes to a Marker:**
 - Create a marker in a current transaction by using the SAVEPOINT statement.
 - Roll back to that marker by using the ROLLBACK TO SAVEPOINT statement.

DBMS SQL

Set Operations

Set Operations

- There are situations when we need to “combine the results” from two or more SELECT statements. SQL enables us to handle these requirements by using “Set operations”.
- The result of each SELECT statement can be treated as a set. SQL set operations can be applied on these sets to arrive at a final result.
- SQL supports the following four Set operations:
 - **UNION ALL**
 - ❖ Combines the results of two SELECT statements into one result set.
 - **UNION**
 - ❖ Same as UNION ALL. Eliminates duplicate rows from that result set.

Set Operations

➤ MINUS

- ❖ Takes the result set of one SELECT statement, and removes those rows that are also returned by a second SELECT statement.

➤ INTERSECT

- ❖ Returns only those rows that are returned by each of two SELECT statements.

- All set operators have equal precedence.

- If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if there are no parentheses explicitly specifying another order.

Set Operations

- Each of these operations combines the results of two SELECT statements into a single result.
 - **Note:** While using SET operators, the column names from the first query appear in the result set.

Compound queries:

- SQL statements containing the Set operators are referred to as “compound queries”. Each SELECT statement in a compound query is referred to as a “component query”.
- Two SELECT statements can be combined into a compound query by a set operation only if they satisfy the following two conditions:
 - ❖ The “result sets” of both the queries must have the “same number of columns”.
 - ❖ The “datatype” of each column in the “second result set” must match the “datatype” of its corresponding column in the “first result set”.

Set Operations

If component queries select character data, then the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, then the returned values have datatype CHAR.
 - If either or both of the queries select values of datatype VARCHAR2, then the returned values have datatype VARCHAR2.
-
- **Tip:** The datatypes do not need to be the same, if those in the second result set can be automatically converted by Oracle (using implicit casting) to types that are compatible with those in the first result set.

- By using the UNION clause, multiple queries can be put together, and their output can be combined.
- The UNION operator returns the records retrieved by either of the queries.
 - By default, the UNION operator eliminates duplicate records.
 - If however we want to retain duplicates, we use UNION ALL instead of UNION.

```
SELECT empno, ename FROM Emp  
UNION  
SELECT empno, ename FROM Emp2
```

- The UNION clause forces all rows returned by each portion of the union to be sorted, merged and filtered for duplicates before the first row is returned to the “calling module”.
- A UNION ALL simply returns all rows including duplicates. It does not perform SORT, MERGE and FILTER.

INTERSECT and MINUS Operator

- The INTERSECT operator returns those rows, which are retrieved by both the queries.

```
SELECT * FROM Emp  
INTERSECT  
SELECT * FROM Emp2
```

- The MINUS operator returns all rows retrieved by the first query but not by the second query.

```
❖ SELECT * FROM Emp  
❖ MINUS  
❖ SELECT * FROM Emp2
```

DBMS SQL

Joins and SubQueries

- JOINS make it possible to select data from more than one table by means of a single statement.
 - The joining of tables is done in SQL by specifying the tables to be joined in the FROM clause of the SELECT statement.
 - When you join two tables a Cartesian product is formed.
 - The conditions for selecting rows from the product are determined by the predicates in the WHERE clause.
 - All the subsequent WHERE, GROUP BY, HAVING, ORDER BY clauses work on this product.
 - If the same table is used more than once in a FROM clause then “aliases” are used to remove conflicts and ambiguities. They are also called as “co-relation names” or “range variables”. Tables are joined on columns, which have the same “data type” and “data width” in the

tables.

- The JOIN operator specifies how to relate tables in the query.
 - ❖ When you join two tables a Cartesian product is formed, by default.
- Different types of joins are:
 - ❖ Inner / Equi-Join
 - ❖ Non-equijoin
 - ❖ Outer Join
 - ❖ Self Join

Types of Joins

- Given below is a list of JOINS:

Oracle Proprietary Joins	SQL: 1999 Compliant Joins
Cartesian Product	Cross Joins
Equijoin	Inner Joins (Natural Joins)
Outer-join	Left, Right, Full outer joins
Non-equijoin	Join on
Self-join	Join on

Inner Join / Equijoin

- In an Equijoin, the WHERE statement generally compares two columns from two tables with the equivalence operator “=”.
- This JOIN returns all rows from both tables, where there is a match.
- Example:
 - To determine an employee's department name, you compare the value in the DEPARTMENT_ID column in the EMP table with the DEPARTMENT_ID values in the DEPT table.
 - The relationship between the EMP and DEPT tables is an “Equijoin”, that is values in the DEPARTMENT_ID column on both tables must be equal.
 - Frequently, these type of JOIN involves PRIMARY and FOREIGN key complements.

Inner Join / Equijoin

- The Equijoin is also used to provide summary information. Suppose we want to know the details of all departments along with number of employees working in it.
 - Since the number of employees can be found out only through the emp table we have to take a join.
 - Also since we want number of employees per dept, we have to group the product of the join.

```
SELECT dept.deptno, dept.dname, dept.loc, count(*) "NO of
      employees" FROM dept,emp WHERE emp.deptno=dept.deptno
GROUP BY dept.deptno,dept.dname,dept.loc;
```

- NULL is an unknown and not a value, NULL values never satisfy Equi-join condition.

Inner Join / Equijoin

- To join together “n” tables, you need a minimum of “n-1” JOIN conditions.
 - **For example:** To join three tables, a minimum of two joins is required.
 - To display the last name, the department name, and the city for each employee, you have to join the EMP, DEPT, and LOC tables.

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

Non - equijoins

- When the comparison operator used in joining columns is other than equality, the join is called a Non Equi-join (Theta Join).
- Suppose we want to find the different pairs of employees doing the same job but belonging to different departments, the following query is used:

```
SELECT first.ename, first.deptno, second.ename,
       second.deptno, first.job
FROM emp first , emp second
WHERE first.empno != Second.empno
AND first.deptno != second.deptno
AND first.job=second.job
ORDER BY 1;
```

Outer Join

- If a row does not satisfy a JOIN condition, then the row will not appear in the query result.
- The missing row(s) can be returned if an OUTER JOIN operator is used in the JOIN condition.
- The operator is PLUS sign enclosed in parentheses (+), and is placed on the side of the join(table), which is deficient in information.
- Outer Joins are similar to Inner Joins. However, they give a bit more flexibility when selecting data from related tables. This type of join can be used in situations where it is desired to select “all rows from the table on the left (or right or both)”, regardless of whether the other table has values in common, and (usually) enter NULL where data is missing.

Outer Join

- Outer Join is an exclusive “union” of sets (whereas normal joins are intersection). OUTER JOINS can be simulated using UNIONS.
 - In a JOIN of two tables an Outer Join may be for the first table or the second table. If the Outer Join is taken on, say the DEPT table, then each row of DEPT table will be selected at least once whether or not a JOIN condition is satisfied.
- An Outer Join does not require each record in the two joint tables to have a matching record in the other table. The joint table retains each record – even if there is no other matching record.
- Outer Joins subdivide further into “left outer joins”, “right outer joins”, and “full outer joins”, depending on which table(s) one retains the rows from (left, right, or both).

Explanation

- **Left Outer Join:** A Left Outer Join returns all the values from the left table, plus matched values from the right table (or NULL in case of no matching join predicate).
- **Right Outer Join:** A Right Outer Join returns all the values from the right table and matched values from the left table (or NULL in case of no matching join predicate).
- **Full Outer Join:** A Full Outer Join combines the results of both Left and Right Outer Joins. The joint table will contain all records from both tables, and fill in NULLs for missing matches on either side.

- **Syntax**

- **WHERE table1 <OUTER JOIN INDICATOR> = table 2**

- In Oracle it is (+), and it is positioned after the second table.
- Table1.column = table2.column (+) means OUTER join is taken on table1.

Example

- Given below is an example of Outer Join:
 - List the department names, whose employees are there in the employee table, and
 - List the name of the department, whose employees are not present in the employee table

SELECT distinct dname FROM Emp, Dept

WHERE Department.deptno = Employee.deptno(+);

- Consider the query for getting all the department details along with the number of employees in it. In an Equijoin, the details of deptno 40 were not shown because there are no employees assigned to it in the EMP table.
- This can be avoided by taking an Outer Join. Outer Join is an exclusive union of sets (whereas normal joins are intersection).

Example

- Outer Joins can be simulated using UNIONS.
- In a join of two tables an outer join may be for the first table or the second table. If the outer join is taken on say the dept table then each row of dept table will be selected at least once whether or not a join condition is satisfied.

```
SELECT dept.deptno, dept.dname, dept.loc, count(empno) "No. of  
employees" FROM dept, emp  
WHERE emp.deptno(+)=dept.deptno  
GROUP BY dept.deptno,dept.dname,dept.loc;
```

Self join

- In Self Join, two rows from the “same table” combine to form a “resultant row”.
 - It is possible to join a table to itself, as if they were two separate tables, by using table labels (aliases).
 - This allows joining of rows in the same table.
- To join a table to itself, “two copies” of the same table have to be opened in the memory. Hence in the FROM clause, the table name needs to be mentioned twice.
- Since the table names are the same, the second table will overwrite the first table. In effect, this will result in only one table being in memory.
 - This is because a table name is translated into a specific memory location.
- To avoid this, each table is opened using an “alias”.
 - These two table aliases will cause two identical tables to be opened in different memory locations.
 - This will result in two identical tables to be physically present in the computer memory.

Self Join

- Example

```
SELECT a.ename name , a.sal salary , b.ename mgr, b.sal " Mgr  
Salary"  
FROM emp a , emp b  
WHERE A.mgr = B.empno;
```

Subqueries

- A sub-query is a form of an SQL statement that appears inside another SQL statement.
 - It is also called as a “nested query”.
- The statement, which contains the sub-query, is called the “parent statement” and it uses the rows returned by the sub-query.
- Subqueries can be used for the following purpose :
 - To insert records in a target table.
 - To create tables and insert records in the table created.
 - To update records in the target table.
 - To create views.
 - To provide values for conditions in the clauses, like WHERE, HAVING, IN, etc., which are used with SELECT, UPDATE and DELETE statements.

Examples

- Example 1: To display name and salary of employees who have the same job as the employee 7896.
- Example 2: To display the details of the employees working in BOSTON.

```
SELECT ename, sal FROM emp
```

```
WHERE job = (Select job from emp  
              where empno = 7896);
```

```
SELECT ename FROM emp
```

```
WHERE deptno =(SELECT deptno FROM dept WHERE  
loc='BOSTON');
```


Types of Subqueries

- Types of Sub queries

- There are two types of sub-queries: Co-related and Non Co-related Sub-queries.

- ❖ If the Sub-query uses any data from the FROM clause of the outer query, then the sub-query is evaluated for each row of the outer query. The Sub-query thus has to be repeated for each row of the outer query and is called as “co-related sub-query”.
 - ❖ When no data is needed from the outer query, the sub-query is evaluated only once. The Sub-query is executed, and the result set is obtained. Such a sub-query is called as “non co-related sub-query”.

Comparison Operators

- Let us discuss sub-queries that use comparison operators.
- Some comparison operators are:

Operator	Description
IN	Equals to any member of
NOT IN	Not equal to any member of
*ANY	compare value to every value returned by sub-query using operator *
*ALL	compare value to all values returned by sub-query using operator *

Subqueries

- When the WHERE clause needs a set of values which can be only obtained from another query, the Sub-query is used. In the WHERE clause it can become a part of the following predicates
 - COMPARISON Predicate
 - IN Predicate
 - ANY or ALL Predicate
 - EXISTS Predicate.
- It can be also used as a part of the condition in the HAVING clause.

```
SELECT ename , job FROM emp
WHERE deptno = 20 AND job IN
(SELECT job FROM emp WHERE deptno =
(SELECT deptno FROM dept WHERE dname = 'SALES')) )
```

Examples

- To display details of employees, who have salary greater than the lowest salary in dept 30.

```
SELECT * FROM emp
      WHERE sal > (SELECT min(sal) FROM emp
                  WHERE deptno = 30);
```

- To display details of employees who are not clerks but whose salary is less than some clerks.

```
SELECT * FROM emp
      WHERE job <> 'CLERK' AND sal < ANY (SELECT sal FROM emp
      WHERE job = 'CLERK');
```

Examples

- To display employees who have salary greater than the highest sal in dept 30.

```
SELECT * FROM emp WHERE sal > ALL  
  (SELECT sal FROM emp WHERE deptno = 30);
```

- The EXISTS / NOT EXISTS operator enables to test whether a value retrieved by the Outer query exists in the result-set of the values retrieved by the Inner query.
 - The EXISTS / NOT EXISTS operator is usually used with a co-related sub-query.
 - ❖ If the query returns at least one row, the operator returns TRUE.
 - ❖ If the value does not exist, it returns FALSE.
- The NOT EXISTS operator enables to test whether a value retrieved by the Outer query is not a part of the result-set of the values retrieved by the Inner query.

Examples

- Example 1: To display details of employees who have some other employees reporting to them.

```
SELECT * FROM emp X
      WHERE EXISTS (SELECT * FROM emp Y
                    WHERE Y.mgr = X.empno) ;
```

- Example 2: To display details of departments which have employees working in it.

```
SELECT * FROM dept
      WHERE EXISTS ( SELECT * FROM emp
                    WHERE emp.deptno = dept.deptno) ;
```

- Example 3: To display details of departments which have no employees working in it:

```
SELECT * From dept
      WHERE NOT EXISTS (SELECT * FROM emp
                       WHERE emp.deptno = dept.deptno) ;
```

- The Exists Predicate is of the form
 - <Query> WHERE EXISTS < sub-query>
- Query will be evaluated if EXISTS takes a value TRUE.
 - It takes the Value TRUE if the <sub-query> result table is non null and FALSE if it is Null.
 - It is mostly used with Co-Related Subqueries.
 - EXISTS does not check for a particular value. It checks whether subquery returns rows or not.

```
SELECT * FROM dept a
2 WHERE EXISTS (SELECT * FROM EMP b
WHERE b.deptno = a.deptno);
```