SQL

SQL is a standard language for accessing and manipulating databases. SQL stands for Structured Query Language. It lets you access and manipulate databases. It is an ANSI (American National Standards Institute) standard.

SQL can

* execute queries against a database
* retrieve data from a database
* insert records in a database
* update records in a database
* delete records from a database
* create new databases
* create new tables in a database
* create stored procedures in a database
* create views in a database
* set permissions on tables, procedures, and views

Although SQL is an ANSI (American National Standards Institute) standard, there are many different versions of the SQL language. However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner. Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard.

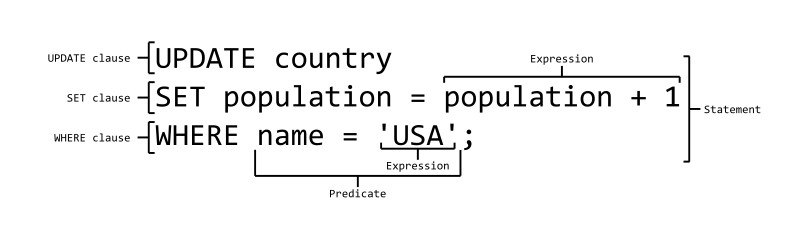
**Type of SQL**

* **DDL (Data Definition Language)**
  + Defines database structure
  + Create/Alter/Truncate/Drop
* **DML (Data Manipulation Language)**
  + Managing/Manipulating data
  + Select/Insert/Update/Delete/Merge
* **DCL (Data Control Language)**
  + Gives User access privileges
  + Grant/Revoke privileges
* **TCL (Transaction Control Language)**
  + Transactions (set of DMLs)
  + Commit/Rollback transactions

**Elements of SQL**

The SQL language is subdivided into several language elements, including:

* **Clauses**: These are constituent components of statements and queries.
* **Expressions**: These can produce either scalar values or tables consisting of columns and rows of data.
* **Predicates**: These specify conditions that can be evaluated to SQL three-valued logic (3VL), which are used to limit the effects of statements and queries, or to change program flow.
* **Queries**: These retrieve the data based on specific criteria. This is the most important element of SQL.
* **Statements**: These may have a persistent effect on schemata and data, or which may control transactions, program flow, connections, sessions, or diagnostics.
* **Insignificant whitespace**: is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

****

Oracle SQL Syntax

**DML**

**SELECT**

The SELECT statement is used to select data from a database. The result is stored in result-set.

SELECT column\_name(s)   
FROM table\_name;

This Query will return all data of column names mentioned from the table\_name.

**Note:** For all columns “ \* ” can be used.

**SELECT DISTINCT**

The Select Distinct is used to select Unique (non-duplicate) records.

SELECT DISTINCT column\_name(s)   
FROM table\_name;

This Query will return all the unique records of columns from table\_name

**SELECT with WHERE**

The “WHERE” clause is used to extract only those records that fulfill a specified criterion.

SELECT column\_name(s)   
FROM table\_name  
WHERE column\_name operator value;

**Note:** For Text Values always use single quotes, but not for numeric values. Another column name can be used instead of value.

Operators:

= Equal

!= Not-Equal

> Greater Than

< Less Than

>= Greater than Equal to

<= Less than Equal to

BETWEEN Between a range

LIKE Search for a pattern using wildcards\*

IN To specify multiple possible values for a column

AND If both conditions are true

OR If either condition is true

NOT If the condition is not true

ROWNUM Number of TOP rows to display

\**wildcards: SQL wildcards can be used when searching for data in a database. SQL wildcards can substitute for one or more characters when searching for data in a database. SQL wildcards must be used with the SQL LIKE operator.*

*% A substitute for zero or more characters*

*\_ A substitute for exactly one character*

*[Charlist] Any single character in charlist*

*[! Charlist] Any single character not in charlist*

**SELECT INTO**

The SELECT INTO statement selects data from one table and inserts it into a different table.

SELECT column\_name(s)  
INTO new\_table\_name [IN externaldatabase]  
FROM old\_tablename;

**Note:** It can be used with WHERE Clause and JOINs

**ORDER BY**

The ORDER BY keyword is used to sort the result-set by a specified column.

SELECT column\_name(s)   
FROM table\_name  
ORDER BY column\_name(s) ASC|DESC;

**Note:** Use DESC for descending order and ASC for ascending order. ASC is default.

**ALIAS**

You can give a table or a column another name by using an alias.

For Table Name

SELECT column\_name(s)  
FROM table\_name  
AS alias\_name;

For Column Name

SELECT column\_name AS alias\_name  
FROM table\_name;

**JOINS**

The JOIN keyword is used in an SQL statement to query data from two or more tables, based on a relationship between certain columns in these tables.

The types of JOINs

**JOIN**: Return rows when there is at least one match in both tables

SELECT column\_name(s)  
FROM table\_name1  
INNER JOIN table\_name2  
ON table\_name1.column\_name=table\_name2.column\_name;

**LEFT JOIN**: Return all rows from the left table, even if there are no matches in the right table

SELECT column\_name(s)  
FROM table\_name1  
LEFT JOIN table\_name2  
ON table\_name1.column\_name=table\_name2.column\_name;

**RIGHT JOIN**: Return all rows from the right table, even if there are no matches in the left table

SELECT column\_name(s)  
FROM table\_name1  
RIGHT JOIN table\_name2  
ON table\_name1.column\_name=table\_name2.column\_name;

**FULL JOIN**: Return rows when there is a match in one of the tables

SELECT column\_name(s)  
FROM table\_name1  
FULL JOIN table\_name2  
ON table\_name1.column\_name=table\_name2.column\_name;

**Cross JOIN**: Returns Cartesian product of rows.

SELECT column\_name(s)  
FROM table\_name1  
cross JOIN table\_name2;

Or

SELECT column\_name(s)  
FROM table\_name1,table\_name2;

**Natural JOIN**: Return rows when there is a common column name(s) in both tables and perform join on them.

SELECT column\_name(s)  
FROM table\_name1  
Natural JOIN table\_name2;

**UNION**

The UNION operator is used to combine the result-set of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

SELECT column\_name(s) FROM table\_name1  
UNION  
SELECT column\_name(s) FROM table\_name2;

**Note:** The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.

SELECT column\_name(s) FROM table\_name1  
UNION ALL  
SELECT column\_name(s) FROM table\_name2;

**PS:** The column names in the result-set of a UNION are always equal to the column names in the first SELECT statement in the UNION.

**GROUP BY**

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

SELECT column\_name, aggregate\_function (column\_name)  
FROM table\_name  
WHERE column\_name operator value  
GROUP BY column\_name;

**Note:** We can also use the GROUP BY statement on more than one column.

**HAVING**

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

SELECT column\_name, aggregate\_function (column\_name)  
FROM table\_name   
GROUP BY column\_name  
HAVING aggregate\_function(column\_name) operator value;

Aggregate Functions: Avg(), count(), first(), last(), max(), min(), sum(), ucase(), lcase(), mid(), len(), round(), stdev(), Variance() etc.

**INTERSECT**

The Intersect operator will combine the result and returns only those rows returned by both queries.

SELECT column\_name(s) FROM table\_name1  
INTERSECT  
SELECT column\_name(s) FROM table\_name2;

**MINUS**

The MINUS operator combines results and returns only rows returned by the first query but not by the second.

SELECT column\_name(s) FROM table\_name1  
MINUS  
SELECT column\_name(s) FROM table\_name2;

**SUB Query**

A query Nested within another query is known as SUB Query.

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name operator(ANOTHER-QUERY);

SELECT column\_name(s)  
FROM (ANOTHER-QUERY)  
WHERE CONDITION;

**Note:** We can also use sub query within a sub query.

**WITH**

Another way to write a sub-Query is to use WITH statement.

With  
ALIAS\_1 AS (ANOTHER-QUERY)   
SELECT column\_name FROM ALIAS\_1;

**Start With – Connect BY**

This statement is used in Hierarchical Database. But it can also be used in Relational database.

It is used where ever a recursion is required.

SELECT column\_name  
FROM table\_name  
START WITH initial condition  
CONNECT BY recursion condition;

**CASE**

This statement has the functionality of if then else

select table\_name,   
CASE [ expression ]   
WHEN condition\_1 THEN result\_1  
WHEN condition\_2 THEN result\_2  
...   
WHEN condition\_n THEN result\_n  
ELSE result  
END   
FROM table\_name;

**NOTE:** EXPRESSION and ELSE are optional.

**PS:** You can also use CASE to evaluate two or more fields by using AND or OR in condition.

**MERGE**

Use the MERGE statement to select rows from one or more sources for update or insertion into a table or view. You can specify conditions to determine whether to update or insert into the target table or view.

Basic syntax with ‘SELECT’

MERGE INTO (SELECT column\_name(s) FROM TABLE\_NAME WHERE condition)   
USING (SELECT column\_name(s) FROM new\_table\_name)   
ON condition   
WHEN MATCHED THEN UPDATE SET condition;

**CONDITIONS**

In SQL operators are evaluated before conditions.

**EXISTS/NOT EXISTS**

It checks if sub-query returns any row or not.

SELECT column\_name(s) FROM table\_name   
where EXISTS|NOT EXISTS (SUB-QUERY) ;

**INFINITE**

It checks if expression is Infinite or not

SELECT column\_name(s) FROM table\_name   
 WHERE expression IS INFINITE;

**Note:** NAN can also be used for same purpose

**ALL/ANY**

It checks if operator is true for ALL or ANY sub-query or list.

SELECT column\_name(s) FROM table\_name   
 WHERE column\_name operator ALL|ANY ([Sub-Query]| [List]);

**Note:** SOME can also be used instead of ANY

**LIKE**

It finds strings of pattern defined

SELECT column\_name(s) FROM table\_name   
WHERE column\_name LIKE [pattern];

Pattern can be defined using wild cards

**%** replaces one or more characters   
**\_** replaces exactly one character   
**[list]** if any one of the list is present   
**[! list]** if none of the list is present

**AND/OR/NOT**

AND or OR checks if either both conditions are true or any condition is true respectively.

SELECT column\_name(s) FROM table\_name   
 WHERE condition1 AND|OR condition2;

Not reverses the condition

SELECT column\_name(s) FROM table\_name   
 WHERE NOT condition;

**IN**

It checks if the expression is present in LIST|SUB-QUERY or not

SELECT column\_name(s) FROM table\_name   
 WHERE expression IN ([SUB-QUERY]| [LIST]);

**IS NULL/IS NOT NULL**

It checks if column\_name is NULL or NOT

SELECT column\_name(s) FROM table\_name   
 WHERE column\_name IS [NOT] NULL;

**BETWEEN/NOT BETWEEN**

It checks if the column\_name is in BETWEEN a list or not

SELECT column\_name(s) FROM table\_name   
 where column\_name [NOT] BETWEEN value1 AND value2;

**Single Comparison**

It compares two expressions or values

SELECT column\_name(s) FROM table\_name   
WHERE expression| value operator expression| value;

**Single Comparison Operators**

=, !=, <, >, <= and >=

**DCL**

**GRANT**

Grant is used to grant privileges on tables or views to users.

Grant select on TABLE\_NAME|VIEW\_NAME to USER;

**Revoke**

Revoke is used to revoke already granted privileges on tables or views from users.

REVOKE select on TABLE\_NAME|VIEW\_NAME from USER;

**Listing Information on Privileges**

To view information on privileges use following queries

To see which table privileges are granted by you to other users.

SELECT \* FROM USER\_TAB\_PRIVS\_MADE;

To see which table privileges are granted to you by other users

SELECT \* FROM USER\_TAB\_PRIVS\_RECD;

To see which column level privileges are granted by you to other users.

SELECT \* FROM USER\_COL\_PRIVS\_MADE;

To see which column level privileges are granted to you by other users

SELECT \* FROM USER\_COL\_PRIVS\_RECD;

To see which privileges are granted to roles

SELECT \* FROM USER\_ROLE\_PRIVS;

**DDL**

## **The CREATE DATABASE Statement**

The CREATE DATABASE statement is used to create a database.

CREATE DATABASE database\_name;

## **The CREATE TABLE Statement**

The CREATE TABLE statement is used to create a table in a database.

CREATE TABLE table\_name  
(  
column\_name1 data\_type [constraint],  
column\_name2 data\_type,  
column\_name3 data\_type,  
....  
);

Constraints:

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

### The CREATE INDEX Syntax

Creates an index on a table

CREATE INDEX index\_name  
ON table\_name (column\_name)

CREATE UNIQUE INDEX index\_name  
ON table\_name (column\_name)

## **The ALTER TABLE Statement**

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

ALTER TABLE table\_name  
ADD column\_name datatype;

ALTER TABLE table\_name  
DROP COLUMN column\_name;

ALTER TABLE table\_name  
MODIFY column\_name datatype;

**The DROP Statement**

Indexes, tables, and databases can easily be deleted/ removed with the DROP statement.

DROP INDEX index\_name;

DROP TABLE table\_name;

DROP DATABASE database\_name;

**The Truncate Statement**

If we only want to delete the data inside the table, and not the table itself, Then use the TRUNCATE TABLE statement.

TRUNCATE table table\_name;

**Data Types**

### Character strings

* CHARACTER(n) or CHAR(n) — fixed-width n-character string, padded with spaces as needed
* CHARACTER VARYING(n) or VARCHAR(n) — variable-width string with a maximum size of n characters
* NATIONAL CHARACTER(n) or NCHAR(n) — fixed width string supporting an international character set
* NATIONAL CHARACTER VARYING(n) or NVARCHAR(n) — variable-width NCHAR string

### Bit strings

* BIT(n) — an array of n bits
* BIT VARYING(n) — an array of up to n bits

### Numbers

* INTEGER and SMALLINT
* FLOAT, REAL and DOUBLE PRECISION
* NUMERIC(precision, scale) or DECIMAL(precision, scale)

### Date and time

* DATE — for date values (e.g., 2011-05-03)
* TIME — for time values (e.g., 15:51:36). The granularity of the time value is usually a *tick* (100 nanoseconds).
* TIME WITH TIME ZONE or TIMETZ — the same as TIME, but including details about the time zone in question.
* TIMESTAMP — This is a DATE and a TIME put together in one variable (e.g., 2011-05-03 15:51:36).
* TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ — the same as TIMESTAMP, but including details about the time zone in question.

**Analytical Functions**

Introduced in Oracle 8i, analytic functions allowed developers to perform tasks in SQL that were previously confined to procedural languages. Whatever an analytic function does can be done by native SQL, with join and sub-queries. But the same routine done by analytic function is always faster, or at least as fast, when compared to native SQL.

Analytic functions give aggregate result, but they do not group the result set. They return the group value multiple times with each record.

The general syntax of analytic function is:

**SELECT column\_name(s),**

***Function (arg1,..., argn) OVER ( [PARTITION BY <...>] [ORDER BY <....>] [<window\_clause>] )***

**FROM emp;**

The clause ***PARTITION BY*** is used to break the result set into groups***. PARTITION BY*** can take any non-analytic SQL expression.

Some functions support the <window\_clause> inside the partition to further limit the records they act on. In the absence of any <window\_clause> analytic functions are computed on all the records of the partition clause.

The functions **SUM, COUNT, AVG, MIN, MAX** are the common analytic functions the result of which does not depend on the order of the records.

Functions like **LEAD, LAG, RANK, DENSE\_RANK, ROW\_NUMBER, FIRST, FIRST VALUE, LAST, LAST VALUE** depends on order of records.

To specify the order of the records in the partition we use the "***ORDER BY***" clause inside the ***OVER( )*** clause. This is different from the ***ORDER BY*** clause of the main query which comes after ***WHERE***.

**LEAD and LAG**

LEAD has the ability to compute an expression on the next rows (rows which are going to come after the current row) and return the value to the current row. Same goes for LAG. The general syntax of LEAD|LAG is:

***LEAD|LAG (<sql\_expr>, <offset>, <default>) OVER (<analytic\_clause>)***

<sql\_expr> is the expression to compute from the leading row.  
< offset> is the index of the leading row relative to the current row.  
< offset> is a positive integer with default 1.  
< default> is the value to return if the <offset> points to a row outside the partition range.

**ROW\_NUMBER, RANK and DENSE\_RANK**

All the above three functions assign integer values to the rows depending on their order. That is the reason of clubbing them together.

**ROW\_NUMBER( )** gives a running serial number to a partition of records. It is very useful in reporting, especially in places where different partitions have their own serial numbers.

**RANK() and DENSE\_RANK()** both provide rank to the records based on some column value or expression. In case of a tie of 2 records at position N, RANK declares 2 positions N and skips position N+1 and gives position N+2 to the next record. While DENSE\_RANK declares 2 positions N but does not skip position N+1.

**FIRST VALUE and LAST VALUE function**

The general syntax is:

***FIRST\_VALUE(<sql\_expr>) OVER (<analytic\_clause>)***

The FIRST\_VALUE analytic function picks the first record from the partition after doing the ORDER BY. The <sql\_expr> is computed on the columns of this first record and results are returned. The LAST\_VALUE function is used in similar context except that it acts on the last record of the partition.

**FIRST and LAST function**

The FIRST function (or more properly KEEP FIRST function) is used in a very special situation. Suppose we rank a group of record and found several records in the first rank. Now we want to apply an aggregate function on the records of the first rank. KEEP FIRST enables that.

The general syntax is:

***Function( ) KEEP (DENSE\_RANK FIRST ORDER BY <expr>) OVER (<partitioning\_clause>)***

Please note that FIRST and LAST are the only functions that deviate from the general syntax of analytic functions. They do not have the ORDER BY inside the OVER clause. Neither do they support any <window> clause. The ranking done in FIRST and LAST is always DENSE\_RANK.

**LISTAGG**

For a specified measure, LISTAGG orders data within each group specified in the ORDER BY clause and then concatenates the values of the <measure> column.

LISTAGG (measure\_expression, 'DELIMETER ') WITHIN GROUP (ORDER BY clause)

[OVER (PARTITION BY clause)]