SQL SYNTAXS

CREATE TABLE:

```
CREATE TABLE table_name (
  column1 type1,
  column2 type2,
  ...
);
CREATE TABLE player (
  name VARCHAR (200),
  age INTEGER,
  score INTEGER
);
```

Data Type	Syntax
Integer	INTEGER / INT
Float	FLOAT
String	VARCHAR
Text	TEXT
Date	DATE
Time	TIME
Datetime	DATETIME

Data Type	Syntax
Boolean	BOOLEAN

1. Boolean values are stored as integers 0 (FALSE) and 1 (TRUE).
2. Date object is represented as: 'YYYY-MM-DD'
3. Datetime object is represented as: 'YYYY-MM-DD HH:MM:SS'

INSERTING ROWS:

INSERT clause is used to insert new rows in a table.

```
INSERT INTO
   table_name (column1, column2,..., columnN)
VALUES
   (value11, value12,..., value1N),
   (value21, value22,..., value2N),
...;
```

```
INSERT INTO
  player (name, age, score)
VALUES
  ("Rakesh", 39, 35),
  ("Sai", 47, 30);
```

Let's see the added data: we can Retrive the inserted data by using the following command.

```
SELECT * FROM table_name;
```

RETRIVING DATA: (SELECT CLAUSE)

SELECT clause is used to retrive rows from the table.

```
SELECT
   column1,
   column2, ...,
   columnN
FROM
   table_name;
```

EXAMPLE:

```
SELECT   name, age
FROM   player;
```

SELECTING SPECIFIC ROWS:

WHERE clause used to retrive only specific rows.

```
SELECT *
FROM table_name
WHERE condition;
```

EXAMPLE:

```
SELECT * FROM player WHERE name="Sai";
```

UPDATE ROWS:

UPDATE Clause is used to update the data of an existing table in database. We can update all the rows or only specific rows as per the requirement.

```
UPDATE                    (UPDATE ALL ROWS)
  table_name
SET
  column1 = value1;
```

```
UPDATE
  player
SET
  score = 100;
```

UPDATE SPECIFIC ROWS:

```
UPDATE
  table_name
SET
  column1 = value1
WHERE
  column2 = value2;
```

```
UPDATE
  Player   SET score = 150 WHERE name = "Ram";
```

DELETE ROWS: DELETE clause is used to delete existing records from a table.

DELETE ALL ROWS:

```
DELETE FROM table_name;
```

DLETE SPECIFIC ROWS:

```
DELETE FROM
    table_name
WHERE
    column1 = value1;
```

```
DELETE FROM
    player
WHERE
    name = "Shyam";
```

ALTER CLAUSE: clause is used to add, delete, or modify columns in an existing table. Let's learn more about ALTER clause using the following database.

ADD COLUMN:

```
ALTER TABLE
  table_name
ADD
  column_name datatype;
```

RENAME COLUMN:

```
ALTER TABLE
  table_name RENAME COLUMN c1   TO   c2;
```

```
ALTER TABLE
  player RENAME COLUMN jersey_num TO jersey_number;
```

DROP COLUMN:

```
ALTER TABLE
  table_name DROP COLUMN column_name;
```

```
ALTER TABLE
  player DROP COLUMN jersey_number;
```

LIKE OPERATOR: operator is used to perform queries on strings. This operator is especially used to WHERE clause to retrieve all the rows that match the given pattern.

```
SELECT * FROM table_name
WHERE
    c1 LIKE matching_pattern;
```

```
SELECT * FROM product WHERE category LIKE “Gadgets”;
```

Common Patterns

Pattern	Example	Description
Exact Match	WHERE name LIKE "mobiles"	Retrieves products whose name is exactly equals to "mobiles"
Starts With	WHERE name LIKE "mobiles%"	Retrieves products whose name starts with "mobiles"
Ends With	WHERE name LIKE "%mobiles"	Retrieves products whose name ends with "mobiles"
Contains	WHERE name LIKE "%mobiles%"	Retrieves products whose name contains with "mobiles"
Pattern Matching	WHERE name LIKE "a_%"	Retrieves products whose name starts with "a" and have at least 2 characters in length

LOGICAL OPERATORS:

AND : Used to fetch rows that satisfy two or more conditions.

OR : Used to fetch rows that satisfy at least one of the given condition

NOT : Used to negate a condition in the WHERE clause.

```
SELECT * FROM table_name
WHERE
  condition1
  operator condition2
  operator condition3
  ...;
```

```
SELECT   *   FROM
  product
WHERE
  category = "Clothing"
  AND price <= 1000;
```

EX:

```
SELECT    *    FROM
  product
WHERE
  brand = "Redmi"
  AND rating > 4
  OR brand = "OnePlus";
```

IN OPERATOR: Retrieves the corresponding rows from the table if the value of column(c1) is present in the given values(v1,v2,..).

```
SELECT    *   FROM
  table_name
WHERE
  c1 IN (v1, v2,..);
```

EX:

```
SELECT   *   FROM   product
WHERE    brand IN ("Puma", "Levi's", "Mufti", "Lee", "Denim");
```

BETWEEN OPERATOR: Retrieves all the rows from table that have column (c1) value present between the given range (v1 and v2).

```
SELECT  *  FROM
 table_name
WHERE
 c1  BETWEEN  v1
 AND v2;
```

EX:

```
SELECT  name, price, brand
FROM    product
WHERE   price
BETWEEN 1000
 AND 5000;
```

ORDER BY: We use ORDER BY clause to order rows. By default, ORDER BY sorts the data in the ASCENDING ORDER.

```
SELECT  column1, column2, column N
FROM
 table_name [WHERE condition]
ORDER BY
 column1 ASC / DESC,
 cloumn2 ASC / DESC;
```

EXAMPLE:

```
SELECT   name, price, rating
FROM    product
WHERE    brand = "Puma"
ORDER BY
  price ASC,
  rating DESC;
```

DISTINCT clause is used to return the distinct i.e unique values.

```
SELECT
  DISTINCT column1, column2,columnN
FROM    table_name
WHERE
  [condition];
```

EXAMPLE:

```
SELECT
  DISTINCT brand
FROM
  product
ORDER BY
  brand;
```

LIMIT: Clause is used to specify the number of rows(n) we would like to have in result.

```
SELECT    column1, column2,column N
FROM
  table_name
LIMIT n;
```

EXAMPLE:

```
SELECT    name, price, rating
FROM     product
WHERE
  brand = "Puma"
ORDER BY
  rating DESC
LIMIT 2;
```

OFFSET: clause is used to specify the position (from nth row) from where the chunk of the results are to be selected.

```
SELECT    column1, column2,column N
FROM     table_name
OFFSET n;
```

EXAMPLE:

```
SELECT    name, price, rating
FROM      product
ORDER BY  rating DESC
LIMIT 5
OFFSET 5;
```

AGGREGATION FUNCTIONS: Combining multiple values into a single value.

Aggregate Functions	Description
COUNT	Counts the number of values
SUM	Adds all the values
MIN	Returns the minimum value
MAX	Returns the maximum value
AVG	Calculates the average of the values

- We can calculate multiple aggregate functions in a single query.

```
SELECT
  aggregate_function(c1),
  aggregate_function(c2)
FROM
    TABLE;
```

```
SELECT    SUM(score)
FROM      player_match_details
WHERE     name = "Ram";
```

```
SELECT
  MAX(score),
  MIN(score)
FROM    player_match_details
WHERE   year = 2011;
```

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

```
SELECT COUNT()
 FROM player_match_details;
```

```
SELECT COUNT(1)
 FROM player_match_details;
```

ALIAS: Using the keyword AS, we can provide alternate temporary names to the columns in the ouput.

```
SELECT    c1 AS a1, c2 AS a2, ...
FROM      table_name;
```

EXAMPLE:

```
SELECT   name AS player_name
FROM    player_match_details;
```

```
SELECT   AVG(score) AS avg_score
FROM    player_match_details;
```

GROUP BY: The GROUP BY Clause in SQL is used to group rows which have same values for the mentioned attributes.

```
SELECT   c1, aggregate_function(c2)
FROM     table_name
GROUP BY   c1;
```

```
SELECT    name, SUM(score) as total_score
FROM     player_match_details
GROUP BY name;
```

- GROUP BY WITH WHERE:

```
SELECT
   c1,
   aggregate_function(c2)
FROM    table_name
WHERE   c3 = v1
GROUP BY c1;
```

HAVING: Clause is used to filter the resultant rows after the application of GROUP BY Clause.

```
SELECT    c1, c2, aggregate_function(c1)
FROM     table_name
GROUP BY c1, c2
HAVING
  condition;
```

```
SELECT    name, count (*) AS half_centuries
FROM     player_match_details
WHERE   score >= 50
GROUP BY    name
HAVING    half_centuries > 1;
```

DATE FUNCTIONS: Date Functions are used to extract the date or time from a datetime field. One import function in date functions is the strftime() function.

strftime() function is used to extract year, month, day, hour, etc. from a date (or) datetime field based on a specified formats as strings.

```
strftime(format, field_name)
```

```
strftime("%Y", release_date)
```

format	description	output format	Function	Behavior
%Y	Year	1990, 2021 etc.	strftime("%Y", field_name)	Extract Year
%m	Month	01 - 12	strftime("%m", field_name)	Extract Month
%d	Day of the month	01 - 31	strftime("%d", field_name)	Extract Day
%H	Hour	00 - 24	strftime("%H", field_name)	Extract Hour

```
SELECT   name, strftime ('%Y', release_date)
FROM   movie;
```

CAST FUNCTION: In database management systems, the CAST function is used to convert a value from one data type to another data type.

```
CAST (value AS data_type);
```

```
CAST (strftime('%Y', release_date) AS INTEGER)
```

EXAMPLE:

```
SELECT
  strftime ('%m', release_date) AS MONTH,
  COUNT (*) AS total_movies
FROM    movie
WHERE CAST (strftime('%Y', release_date) AS INTEGER) = 2010
GROUP BY strftime('%m', release_date);
```

ARITHEMETIC FUNCTIONS: In SQL are used to perform mathematical operations on numeric values. Some commonly used arithmetic functions are FLOOR, CEIL, ROUND.

FLOOR FUNCTION: It's Rounds a number to the nearest integer below its current value.

```
FLOOR (number)
```

```
SELECT FLOOR (2.3);
```

ROUND FUNCTION: The ROUND function rounds a number to a specified number of decimal places.

```
ROUND (number, decimal_places)
```

```
SELECT ROUND (2.345, 2);
SELECT ROUND (2.345, 1);
```

```
SELECT
  name,
  ROUND (collection_in_cr, 1) AS RoundedValue,
  CEIL (collection_in_cr) AS CeilValue,
  FLOOR (collection_in_cr) AS FloorValue
FROM
  movie;
```

STRING FUNCTIONS: String functions in SQL are used to manipulate and operate on string values or character data.

SQL Function	Behavior
UPPER ()	Converts a string to upper case
LOWER ()	Converts a string to lowercase

```
SELECT   name
FROM    movie
WHERE
  UPPER(name) LIKE UPPER("%avengers%");
```

Usaually,UPPER() AND LOWER() functions can help you to perform case-insensitive searches.

CASE Clause: SQL Provides CASE Clause to perform conditional operations. This is similar to the switch case/if-else conditions in other programming languages.

Each condition in the CASE clause is evaluated and result in corresponding value when the first condition is met.

```
SELECT c1, c2
CASE
   WHEN condition1 THEN value1
   WHEN condition2 THEN value2
```

```
    ELSE value
    END AS cn
FROM table;
```

In CASE clause, if no condition is satisfied, it returns the value in the ELSE part. If we do not specify the ELSE part, CASE clause result in NULL.

We can use CASE in various clauses like SELECT, WHERE, HAVING, ORDER BY, GROUP BY.

SET OPERATIONS: The SQL SET operation is used to combine the two or more SQL Queries. [INTERSECT, MINUS, UNION, UNION ALL]

```
SELECT    c1, c2
FROM     table_name
SET_OPERATOR
SELECT    c1, c2
FROM     table_name;
```

```
SELECT actor_id
FROM cast
WHERE movie_id=6
INTERSECT
SELECT actor_id
FROM cast
WHERE movie_id=15;
```

```
SELECT actor_id
FROM cast
WHERE movie_id=6
EXCEPT
SELECT actor_id
FROM cast
WHERE movie_id=15;

SELECT actor_id
FROM cast
WHERE movie_id=6
UNION
SELECT actor_id
FROM cast
WHERE movie_id=15;

SELECT actor_id
FROM cast
WHERE movie_id=6
UNION ALL
SELECT actor_id
FROM cast
WHERE movie_id=15;
```

Clauses	How to Use It	Functionality
CREATE TABLE	CREATE TABLE table_name ...	Creates a new table
INSERT	INSERT INTO table_name ...	Used to insert new data in the table
SELECT	SELECT col1, col2 ..	Retrieves the selected columns
SELECT	SELECT * FROM ...	Retrieves all the columns from a table
FROM	FROM table_name	FROM clause specifies the table(s) in which the required data columns are located
WHERE	WHERE col > 5	Retrieves only specific rows based on the given conditions
UPDATE, SET	UPDATE table_name SET column1 = value1;	Updates the value of a column of all the rows (or only specific rows using WHERE clause)
DELETE	DELETE FROM table_name	Deletes all the rows from the table
DROP	DROP TABLE table_name	Deletes the table from the database
ALTER	ALTER TABLE table_name ...	Used to add, delete or modify columns in a table
ORDER BY	ORDER BY col1 ASC/DESC..	Sorts the table based on the column(s) in the ascending or descending orders
DISTINCT	SELECT DISTINCT col, ...	Gets the unique values of given column(s)
LIMIT	LIMIT 10	Limits the number of rows in the output to the mentioned number
OFFSET	OFFSET 5	Specifies the position (from nth row) from where the chunk of the results are to be retrieved
GROUP BY	GROUP BY col ...	Groups the rows that have same values in the given columns

Clauses	How to Use It	Functionality
HAVING	HAVING col > 20	Filters the resultant rows after the application of GROUP BY clause
CASE	CASE WHEN condition1 THEN value1 WHEN .. ELSE .. END	Returns a corresponding value when the first condition is met

Functions

Functions	How to Use It	Functionality
COUNT	SELECT COUNT(col) ...	Counts the number of values in the given column
SUM	SELECT SUM(col) ...	Adds all the values of given column
MIN	SELECT MIN(col) ...	Gets the minimum value of given column
MAX	SELECT MAX(col) ...	Gets the maximum value of given column
AVG	SELECT AVG(col) ...	Gets the average of the values present in the given column
strftime()	strftime("%Y", col) ...	Extracts the year from the column value in string format. Similarly, we can extract month, day, week of the day and many.
CAST()	CAST(col AS datatype) ...	Converts the value to the given datatype
FLOOR()	FLOOR(col)	Rounds a number to the nearest integer below its current value
CEIL()	CEIL (col)	Rounds a number to the nearest integer above its current value

Functions	How to Use It	Functionality
ROUND()	ROUND(col)	Rounds a number to a specified number of decimal places
UPPER()	UPPER(col)	Converts a string to upper case
LOWER()	Lower(col)	Converts a string to lower case

ENTITY RELATIONSHIP MODEL (ER MODEL):

ONE-TO-ONE REALTIONSHIP: [Person has a passport.]

ONE-TO-MANY or MANY-TO-ONE RELATIONSHIP: [Person can have many cars.]

MANY-TO-MANY RELATIONSHIPS: [Each student can register for many courses, and a course can have many students.]

PRIMARY KEY: Following syntax creates a table with c1 as the primary key.

```
CREATE TABLE table_name (
   c1 t1 NOT NULL PRIMARY KEY,
   cn tn,
);
```

FOREIGN KEY: IN case of foreign key, we just create a foreign key constraint.

```
CREATE TABLE table2(
 c1 t1 NOT NULL PRIMARY KEY, c2 t2,
 FOREIGN KEY(c2) REFERENCES table1(c3) ON DELETE CASCADE
);
```

JOINS: we use JOIN Clause to combine rows from two or more tables, based on a related column between them. There are various types of joins are NATURAL JOIN, INNER JOIN, FULL JOIN, CROSS JOIN, LEFT JOIN, RIGHT JOIN.

NATURAL JOIN: It's combines the tables based on the common columns.

```
SELECT *
FROM table1
  NATURAL JOIN table2;


SELECT course.name,
 instructor.full_name
FROM course
 NATURAL JOIN instructor
WHERE instructor.full_name = "Alex";
```

INNER JOIN: It's combines the rows from both the tables if they meet a specified condition.

```
SELECT *
FROM table1
  INNER JOIN table2
ON table1.c1 = table2.c2;


SELECT student.full_name,
  review.content,
  review.created_at
FROM student
```

```
  INNER JOIN review
ON student.id = review.student_id
WHERE review.course_id = 15;
```

LEFT JOIN: In LEFT JOIN, for each row in the left table, matched rows from the right table are combined. If there is no match, NULL values are assigned to the right half of the rows in the temporary table.

```
SELECT *
FROM table1
  LEFT JOIN table2
ON table1.c1 = tabl2.c2;
```

```
SELECT student.full_name
FROM student
  LEFT JOIN student_course
ON student.id = student_course.student_id
WHERE student_course.id IS NULL;
```

RIGHT JOIN (RIGHT OUTER JOIN): for each row in the right table, matched rows from the left table are combined. If there is no match, NULL values are assigned to the left half of the rows in the temporary table.

```
SELECT *
FROM table1
  RIGHT JOIN table2
ON table1.c1 = table2.c2;
```

```
SELECT *
FROM table2
  LEFT JOIN table1
ON table1.c1 = table2.c2;

SELECT course.name,
   instructor.full_name
FROM course
  RIGHT JOIN instructor
ON course.instructor_id = instructor.instructor_id;
```

FULL JOIN (FULL OUTER JOIN): It's the result of both RIGHT JOIN and LEFT JOIN.

```
SELECT *
FROM table1
  FULL JOIN table2
ON c1 = c2;

SELECT course.name,
   instructor.full_name
FROM course
  FULL JOIN instructor
ON course.instructor_id = instructor.instructor_id;
```

CROSS JOIN (CARTESIAN JOIN): In CROSS JOIN, each row from the first table is combined with all rows in the second table.

Cross Join is also called as CARTESIAN JOIN.

```
SELECT *
FROM table1
  CROSS JOIN table2;
```

```
SELECT course.name AS course_name,
   instructor.full_name AS instructor_name
FROM course
  CROSS JOIN instructor;
```

SELF JOIN: We can also combine a table with itself. This kind of join is called SELF-JOIN.

```
SELECT t1.c1,
  t2.c2
FROM table1 AS t1
  JOIN table1 AS t2
ON t1.c1 = t2.cn;
```

```
SELECT sc1.student_id AS student_id1,
 sc2.student_id AS student_id2, sc1.course_id
FROM
```

```
  student_course AS sc1
  INNER JOIN student_course sc2 ON sc1.course_id = sc2.course_id
WHERE
   sc1.student_id < sc2.student_id;
```

Join Type	Use Case
Natural Join	Joins based on common columns
Inner Join	Joins based on a given condition
Left Join	All rows from left table & matched rows from right table
Right Join	All rows from right table & matched rows from left table
Full Join	All rows from both the tables
Cross Join	All possible combinations

VIEW: A view can simply be considered as a name to a SQL Query.

CREATE VIEW: To create a view in the database, use the CREATE VIEW statement.

```
CREATE VIEW user_base_details AS
SELECT id, name, age, gender, pincode
FROM user;
```

DELETE VIEW: To remove a view from a database, use the DROP VIEW statement.

DROP VIEW view_name;

ADVANTAGES: Views are used to write COMPLEX QUERIES that involves MULTIPLE JOINS, GROUP BY JOINS and can be used whenever needed.

SUBQUERIES: We can write nested queries, i.e., a query inside another query.

```
SELECT
  name, (
    SELECT AVG (rating)
    FROM product
    WHERE category = "WATCH"
  ) - rating AS rating_variance
FROM product
WHERE category = "WATCH";
```

```
SELECT *
FROM product
WHERE rating > (
    SELECT AVG (rating)
```

```
    FROM product
);
```

TRANSCATIONS: A transaction is a logical group of one or more SQL statements. Transactions are used in various scenarios such as banking, ecommerce, social networks, booking tickets, etc.

A transaction has four important properties.

1) Atomicity: Either all SQL statements or none are applied to the database.

2) Consistency: Transactions always leave the database in a consistent state.

3) Isolation: Multiple transaction can occur at the same time without adversely affecting the other.

4) Durability: Changes of a successful transaction persist even after a system crash.

These four properties are commonly Acronym as ACID.

Atomicity Consistency Isolation Durable.

INDEXES: In scenarios like, searching for a word in dictionary, we use index to easily search for the word.
Similarly, in databases, we maintain indexes to speed up the search for data in a table.

CREATE INDEX:

CREATE INDEX index

ON TABLE column;

FOR MULTIPLE COLUMNS:

CREATE INDEX index

ON TABLE (column1, column2, …);

UNIQUE INDEXS:

CREATE UNIQUE INDEX index

ON TABLE column;