SQL (Structure Query Language)

What is SQL?

SQL or Structure Query Language, is a language designed to allow both technical and non-technical users query, manipulate and transform data from a relational database. And due to its simplicity, SQL databases provide safe and scalable storage for millions of website and mobile applications.

→ SELECT Queries

To retreive data from a SQL database, we need to write SELECT statements, which are often colloquially refered to as queries. A query in itself is just a statement which declares what data we are looking for, where to find in a database, and optionally how to transform it before it is turned.

Syntax:

(1) Select query ~~from~~ for a specific column from table myTable

```
SELECT columnName FROM myTable; //Selecting one column
SELECT column1, column2 FROM myTable; //Selecting multiple column.
```

(2) Select query for all columns

```
SELECT * FROM myTable;
```

→ Queries with constraints (Pt-1)

In order to filter certain ~~values~~ results from being returned, we need to use a WHERE clause in the query. The clause is applied to each row of data by checking specific column values to determine whether it should be included in the results or not.

Syntax - SELECT column, anotherColumn...
FROM myTable;
WHERE condition
AND/OR anotherCondition,
AND/OR ...;

More complex clause can be constructed by joining numerous AND or OR logical keyword (i.e. num_wheels >= 4 AND doors <= 2).

Operator	Condition	SQL Example
=, !=, <, <=, >, >=	Standard numerical operator	col_name != 4
BETWEEN...AND...	Number is within range of two values	col_name BETWEEN 1.5 AND 10.5
NOT BETWEEN...AND...	Number is not within range of two values	col_name NOT BETWEEN 1 AND 10
IN(...)	Number exists in a list	col_name IN(2,4,6)
NOT IN(...)	Number doesn't exist in a list	col_name NOT IN(2,4,6)

→ Queries with constraints (Pt-2)

When writing WHERE clauses with columns containing text data, SQL supports a number of useful operators to do things like case-insensitive string comparison and wildcard pattern matching.

Operator	Condition	Example
=	Case sensitive exact string comparison	col_name = "abc"
!= or <>	Case sensitive exact string inequality comparison	col_name != "abcd"
LIKE	Case insensitive exact string comparison	col_name LIKE "ABC"
NOT LIKE	Case insensitive exact string inequality comparison	col_name NOT LIKE "ABCD"
%	Used anywhere in a string to match a sequence of zero or more characters (only with like or not like)	col_name LIKE "%AT%" (matches, "AT", "ATTIC", "CAT", or even "BATS")
_ (underscore)	Used anywhere in a string to match a single character (only with LIKE or NOT LIKE)	col_name LIKE "AN_" (matches "AND" but not "AN")
IN(...)	String exists in a list	col_name IN ("A","B","C")
NOT IN(...)	String does not exist in a list	col_name NOT IN ("D","E","F")

5432

→ Filtering and Sorting Query results

SQL provides a convenient way to discard rows that have a duplicate column value by using the DISTINCT keyword.

```
SELECT DISTINCT column, another_column,...
FROM myTable
WHERE condition(s);
```

~~OR~~

DISTINCT keyword will blindly remove duplicate rows, but there are time when we need to discard duplicate based on specific columns using grouping and GROUP BY clause.

→ ORDERING RESULTS

Most data in real databases are added in no particular column order. SQL provides a way to sort your results by a given column in ascending or descending order using the ORDER BY clause.

```
SELECT column, another_column,...
FROM myTable
WHERE condition(s)
ORDER BY column ASC/DESC;
```

When an ORDER BY clause is specified, each row is sorted alpha-numerically based on the specified column's value.

→ Limiting results to a subset

Another clause which is commonly used with ORDER BY clause are the LIMIT and OFFSET clauses, which are a useful optimization to indicate to the database the subset of the result you care about. LIMIT reduce the number of rows to return, and (optional) OFFSET will specify where to begin counting no. of rows from.

```
SELECT column FROM myTable WHERE condition(s)
ORDER BY column ASC/DESC LIMIT num_limit OFFSET num_offset;
```

→ Multi-Table queries with JOINS

Entity data in real world is often broken down into pieces and stored across multiple orthogonal tables using a process known as normalization.

→ Database Normalization

Database normalization is useful because it minimizes duplicate data in any single table, and allows for in the database to grow independently of each other. But, queries get slightly more complex since they have to be able to find data from different parts of the database.

→ Multiple ~~queries~~-table queries with JOINS

Table that share information about a single entity need to have a primary key that identifies that entity uniquely across a database. One comman primary key-tape is an auto-incrementing integer (because they are space efficient), but it can also be a string, hashed value, so long as it is unique.

Using JOIN in query, we can combine row data across two separate tables using this unique key.

* Here using INNER JOIN

```
SELECT ~~JOIN~~ column, another_table_column,...
FROM myTable
INNER JOIN another_table
        on mytable.id = another_table.id
WHERE condition(s)
ORDER BY column,... ASC/DESC
LIMIT num_limit OFFSET num_offset;
```

The INNER JOIN is a process that matches rows from the first table and the second table which have the same key to create a result row with the combined columns from both tables.

→ OUTER JOINs

INNER JOIN we used might not be sufficient because the resulting table only contains data that belongs in both the table.

If the two tables have asymmetric data which can easily happen when data is entered in different stages, then we will use LEFT JOIN, RIGHT JOIN or FULL JOIN instead to ensure that the data you need is not left out of the results.

```
SELECT column, another_column...
FROM myTable
INNER/LEFT/RIGHT/FULL JOIN another_table
    ON mytable.id = another_table.matching_id
WHERE condition(s)
ORDER BY column,.. ASC/DESC;
LIMIT num_limit OFFSET num_offset;
```

When joining Table A to Table B, a LEFT JOIN includes rows from A regardless of whether a matching row is found in B. The RIGHT JOIN is same like LEFT JOIN but vice versa. A FULL JOIN simply means that rows from both tables are kept, regardless of whether a matching row exists in the other table.

When using any of these JOINS, we have to NULL also.

→ Short note on NULLs

It's always good to reduce the possibility of NULL values in databases ~~and~~ because they require special attention when constructing queries, constraints (certains functions behave differently with null values) and when processing the results.

Sometimes, it's not possible to avoid NULL values, like joining two tables with asymmetric data. In these cases, we can test a column for NULL values in a WHERE clause by using either IS NULL OR IS NOT NULL constraint.

```
SELECT column, another_column,...
FROM myTable.
WHERE column IS/IS NOT NULL
AND/OR another_condition ....;
```

→ Queries with expressions

Each database has its own supported set of mathematical, string, and date functions that can be used in a query.

```
SELECT particle_speed/2.0 AS half_particle_speed
FROM physics_data
WHERE ABS(particle_position)*10.0 > 500;
```

AS keyword is ~~is~~ used to give descriptive alias to make regular column easier to reference.

```
SELECT column AS better_column_name,...
FROM a_long_widgets_table_name AS myWidgets
INNER JOIN widget_sales
    ON myWidgets.id = widget_sales.widget_id;
```

→ Queries with Aggregates

(i) Select query with aggregate function over all rows

```
SELECT AGG-FUNC (column_or_expression) AS
       aggregate-description, ....
FROM myTable
WHERE constraint_expression;
```

COMMAN AGGREGATE FUNCTION

FUNCTION	Description
COUNT(*), COUNT(column)	A comman function used to counts the number of rows in the group if no column is specified, otherwise, count the number of rows in the group with non-NULL values in the specified column.
MIN(column)	Find the smallest numerical value in the specified column for all rows in the group.
MAX(column)	Largest numerical value in the specified column for all rows in the group.
AVG(column)	Average numerical value in the specified column for all the rows in group.
SUM(column)	Sum of all numerical values in the specified column for rows in group.

GROUP AGGREGATE FUNCTION

```
SELECT AGG_FUNC (column_or_expression) AS aggregate_description,...
FROM myTable
WHERE constraint_expression
GROUP BY column;
```

→ Queries with Aggregate (Pt. 2)

GROUP BY clause is executed after the WHERE clause which filters the rows which are to be grouped, then how exactly do we filter the grouped rows?

SQL allow us to do this by adding HAVING clause which is used specifically with the GROUP BY clause to allow us to filter grouped rows from the result set.

```
SELECT group_by_column , AGG_FUNC(column_expression) AS
                                aggregate_result_alias, ....
FROM myTable
WHERE condition
GROUP BY column
HAVING group_condition;
```

→ ORDER of execution of query

```
SELECT DISTINCT column, AGG_FUNC(column_or_expression), ...
FROM myTable
    JOIN another_table AS table2
        ON myTable.column = table2.column
    WHERE constraint_expression
    GROUP BY column
    HAVING constraint_expression
    ORDER BY column ASC/DESC
    LIMIT count OFFSET count;
```

SQL

→ Create Table

- An SQL relation is defined using the create table command

  create table r(A1D1, A2D2, ..., AnDn),
  (Integrity-constraint1),
  ...
  (Integrity-constraintk));

  - r is the name of relation
  - each Ai is an attribute name in the schema of relation r.
  - Di is the datatype of values in the domain of attribute Ai.

{ create table if not exists myTable (
  column DataType TableConstraint DEFAULT default_value,
  another_column " " " "
  ...
);

DataType : (1) INTEGER (Any integer), BOOLEAN (0 and 1),
(2) FLOAT, DOUBLE, REAL
(3) CHARACTER(num_chars), VARCHAR(num_chars), TEXT

varies no. of character that can be store (longer value will be truncated).

(4) DATE, DATETIME

(5) BLOB - Binary data

Table constraint

Contraint

→ Primary Key — means value in this column are unique

→ AUTOINCREMENT — For integer values, means that the value is automatically filled in and incremented with row insertion.

→ UNIQUE — value in the column has to be unique. Differs from 'PRIMARY KEY' in that it doesn't have to be a Key for a row in the table

→ NOT NULL — inserted value cannot be NULL

→ CHECK (expression) :- allow to run more complex expression to test whether value inserted is valid.

→ FOREIGN KEY — This is a consistency check which ensures that each value in this column corresponds to another value in a column in another table.

→ Altering Tables (FOR SQL)

(1) Adding Column

```
ALTER TABLE myTable
ADD column DataType OptionalTableConstraint
    DEFAULT default_value;
```

→ MySQL, In MySQL we can even specify where to insert the new column using the FIRST or AFTER clauses.

(2) Removing column

```
ALTER TABLE myTable
DROP column_to_be_deleted;
```

(3) Renaming The table

```
ALTER TABLE myTable
RENAME TO new_Table_name;
```

Note: Each database implementation supports different methods of altering their tables.

→ Dropping Tables

To remove an entire table including all of its data and metadata. It removes table schema from the database entirely.

```
DROP TABLE IF EXISTS myTable;
```

→ Selecting no. of tuples in output

```
SELECT TOP 10 DISTINCT name FROM myTable;
```

* MySQL supports the LIMIT clause
* ORACLE uses FETCH FIRST n ROWS ONLY

→ SET OPERATIONS

```
(SELECT COLUMN FROM myTable WHERE condition)
UNION / INTERSECT / EXCEPT
(SELECT ... )
```

→ This automatically eliminates all duplicate values

In order to retain all duplicate values use

UNION ALL, INTERSECT ALL, EXCEPT ALL