Notes on SQL*

June 21, 2025

*Working From https://www.sqltutorial.org/

1 Getting Started

1.1 What Is SQL

SQL is a language used to interact with databases. Databases are exactly what you expect, they store data, and potentially structural information about that datas relations.

Two common types are

1. Relational Database Management Systems (RDMBS - PostgreSQL, MySQL, MariaDB, ...)
   (a) Data is stored in Rows and Columns
   (b) Essentialy a large spreadsheet
2. Document Databases (No SQL)
   (a) Data is stored as documents, no SQL

SQL stands for Structured Query Language, and is used to interface with rleational database systems, altering data, gathering data, removing data etc. based on constraints. There are three main types of commands or parts,

1. DDL - Data Definition Language, database structure creation and modification
   (a) CREATE TABLE
   (b) ALTER TABLE
   (c) DROP TABLE
2. DML - Data Manipulation Language, queries data in structures
   (a) SELECT
   (b) INSERT
   (c) UPDATE
   (d) DELETE
3. DCL - Data Control Language, handles authorization
   (a) GRANT
   (b) REVOKE

An RDBMS table consists of rows and columns, each column represents a field, and each row represents a record.

While some SQL commands are universal between the various database implementations and are mandated by the language standards there are plenty of variants with local tools and quirks.

1.2 SQL Syntax

In SQL you tell the language what you want, and the language works out how, a statement in SQL will often follow a pattern of [Verb - Subject - Condition]

A verb gives the action that you want the database to take, eg. SELECT, INSERT, UPDATE, or DELETE, with natural results.

The subject stipulates which database you want to work on, eg. a table.

The condition filters your statement down to only elements satysfying some condition.

An example statement would then be

```
SELECT first_name
FROM employees
WHERE YEAR(hire_date) = 1999;
```

SQL statements include tokens, the most importnat are keywords, which are reserved for SQL use. Another type is literals, as usual these are constants and declared, they can take the usual form of Strings, Numbers, Bools, but also Times, and Timestamps as well as more.

Further there are identifiers which refer to objects such as tables columns and indexes.

Finally expressions can be combined from identifiers, literals and operators.

2 Selecting Data

2.1 SELECT

The basic syntax of SELECT follows the structure

```
SELECT
   select_list
FROM
   table_list;
```

where `select_list` is a comma seperated list of columns to select and `table_list` is the table to select from. Naturally then the command is evaluated "backwards", where the table is grabbed first.

You can use `*` as a generic keyword for all available columns. Using `SELECT *` should in general be avoided for proper use since you should know what you want to grab.

An example on selecting information from a table named *employees* is

```
SELECT
  employee_id,
  first_name,
  last_name,
  hire_date
FROM
  employees;
```

You can perform calculations as part of a SELECT statement such as

```
SELECT
  salary,
  salary*1.05
FROM
  employees;
```

which will give you the salary and the salary increased by 5%. This creates a temporary column with a name depending on the implementation.

You can pick the name of this temporary column by using AS. EG.

```
SELECT
  salary,
  salary*1.05 AS new_salary
FROM
  employees;
```

3 Sorting Rows

3.1 SORT BY

To get the output in a certain order, for instance ordering employees by wage you can use ORDER BY to sort the output of a SELECT command. This type of command would look like

```
SELECT
  select_list
FROM
  table_list
ORDER BY
  sort_expression [ASC | DESC];
```

where you can use ASC or DESC to select an ascending or descending sorting. By default the sort uses ASC. If you wish to use a secondary sort you can sort first by one expression then by another using

```
SELECT
  select_list
FROM
  table_list
ORDER BY
  sort_expression_1 [ASC | DESC],
  sort_expression_2 [ASC | DESC];
```

For our employee example this could look like

```
SELECT
  first_name,
  last_name
FROM
  employees
ORDER BY
  salary DESC,
  hire_date DESC;
```

to get the employee names in order of descending salry, with ties broken by who was hired most recently.

You can terminate an ORDER BY clause with NULLS FIRST to bring any NULL values to the front, or NULLS LAST to send them to the back.

This could look like

```
SELECT
  first_name,
  last_name,
  salary
FROM
  employees
ORDER BY
  salary NULLS FIRST;
```

4 Limiting Rows

4.1 SQL DISTINCT

By adding DISTINCT to a SELECT command you select only entries with distinct entries in the first selected column. On our fake dataset set we can use

```
SELECT DISTINCT
  salary FROM
  employees
ORDER BY
  salary DESC;
```

which would give you a list of the salary levels that exist in the data set in descending order.

DISTINCT treats all NULL values as being identical / selects only one. (usually).

4.2 SQL LIMIT

If you want to grab only the first k elements in a list you can use LIMIT, and if you want to skip some number of rows at the beginning you can use OFFSET. Together this would look like

```
SELECT
  column_list
FROM
  table1
ORDER BY
  column_list DESC
LIMIT
  row_count
OFFSET
  row_to_skip;
```

4.3 SQL FETCH

LIMIT is not actually enforced by the SQL standard, but is widely supported. Modern SQL (post SQL:2008) standards include the OFFSET FETCH clause which behaves similarily. The syntax is

```
OFFSET rows_to_skip { ROW | ROWS }
FETCH { FIRST | NEXT } [ row_count ] { ROW | ROWS } ONLY
```

An example of this syntax would be

```
SELECT
   first_name,
   salary
FROM
   employees
ORDER BY
   salary DESC
FETCH FIRST 5 ROWS ONLY;
```

5 Filtering Data

5.1 WHERE

By adding a WHERE clause to the end of a statement you can select only rows satysfying some condition. In SQL it is possible for a condition to return NULL, true or false; WHERE only selects true evaluations.

WHERE can take a number of comparators / condtions including

Symbol	Comparator
=	Equal To
<>	Not Equal To
<	Less than
>	Greater than
<=	Less than equal to
>=	Greater than or equal

An example of how you could apply this is

```
SELECT
   first_name,
   salary
FROM
   employees
WHERE
   salary > 14000
ORDER BY
   salary DESC;
```

This will grab employee names and salaries for employees earning more than 14,000.

5.2 AND

An AND operator combines two boolean expressions that are being passed to another operator, only true and true combined will make true. and example of how AND is applied would be:

```
SELECT
   first_name,
   salary
FROM
   employees
WHERE
   salary > 14000
   AND job_id = 9
ORDER BY
   salary DESC;
```

SQL evaluates comparisons in order which means that you can order them to avoid eg. division by zero errors.

5.3 OR

OR acts just like AND but takes an or evaluation, if either condition is true, regardless of if the other is NULL. OR short circuits if the first is true which can be useful.

5.4 BETWEEN

The between operator acts as a shortcut for taking both a <= and a >=, the syntax is

```
SELECT
   first_name
FROM
   employees
WHERE salary BETWEEN 10000 AND 14000;
```

5.5 NOT

The NOT operator is the negation of a truth value.

5.6 IN

IN checks if a value is in a discrete set of values, the syntax is `expression IN (value1, value2, ...)`

5.7 LIKE

LIKE performs pattern matching, where the % character matches any length of free characters, the _matches a single character. For instance then `LIKE 'Kim%;` would match any string beginning with Kim. You can choose the escape character freely using ESCAPE excape_character. Then any special character preceded by your escape character will be presented.

5.8 IS NULL

IS NULL returns true if a value is NULL, this is nessecary because NULL is not equal to NULL.

6 Joining Tables

6.1 INNER JOIN

Using a join lets you poll two tables at once, an INNER JOIN selects and joins those elements who have a matching selected key across the two tables. So if you have for instance a table of hired employees and a table of fired employees you could take:

```
SELECT
   first_name
   last_name
FROM
   hiredemployees
INNER JOIN firedemployees ON hiredemployees.id = hiredemployees.id;
```

By the use of table aliases we could shorten the above by renaming `hired_employees` to `h`, and `fired_employees` to h which would make the join `SELECT hired_employees h INNER JOIN fired_employees f ON h.id = f.id`

6.2 LEFT JOIN

A LEFT JOIN will take all elements from the primary table, and join in the additional columns from the extra table on a match condition. If the match condition isn't met, to fill in the new columns they are padded with null values.

6.3 RIGHT JOIN

The RIGHT JOIN behaves exactly as a LEFT JOIN but uses the tables in reverse order.

You can stack JOINs to merge as many table as needed, by simply applying them sequentially.

6.4 Self Joining

You can apply a JOIN between a table and itself to shuffle around elements according to eachother or do special selections. An example of this would be

```
SELECT
   e.first_name
   m.first_name
FROM
   employees e
   LEFT JOIN employees m ON m.employee_id = e.manager_id;
ORDER BY
   manager NULLS FIRST;
```

6.5 FULL OUTER JOIN

A FULL OUTER JOIN takes any entries from either of the tables, merging those that match, and null padding those that don't.

6.6 CROSS JOIN

A CROSS JOIN merges every row in the first table with every table in the second table leading to an n * m number of new rows.

7 Grouping Rows

7.1 GROUP BY

Suppose that you have a list that gives multiple rows that share identical entries in some column of interest. This could be for instance if many employees are employed by the same manager, if you then want to return the names of the managers whose employees meet some criteria, but don't want a large number of duplicate manager entries you can use GROUP BY.

The syntax is:

```
SELECT
   column
FROM
   table_name
GROUP BY
   column;
```

or for multiple columns and with some aggregating function,

```
SELECT
   column1
   column2
   aggregate_function (column3)
FROM
   table_name
GROUP BY
   column;
```

an example aggregate function here would be COUNT, which would give you an extra column detailing how many entries were grouped.

7.2 HAVING

HAVING allows you to filter on the results of a GROUP, if run without a group it behaves like a WHERE would.

7.3 GROUPING SETS

GROUPING SETS lets you construct multiple groupings in the same way as GROUP BY constructs one.

The syntax would be eg.

```
SELECT
   column1
   column2
   aggregate_function (column3)
FROM
   table_name
GROUP BY
   GROUPING SETS
      (column1),       (column1, column 2),       ();
```

7.4 ROLLUP

ROLLUP works much like a plain group by, but includes a hierarchy, where the results are presented subdivided by subgroups of column2 matches within column1 matches, then all column1 matches.

```
SELECT
   column1
   column2
   aggregate_function (column3)
FROM
   table_name
GROUP BY
   ROLLUP (column1, column2);
```

7.5 CUBE

CUBE behaves like a standard GROUP BY, but takes all possible combinations of subgroups between the columns presented. The syntax is:

```
SELECT
   column1
   column2
   aggregate_function (column3)
FROM
   table_name
GROUP BY
   CUBE (column1, column2);
```

8 Set Operations

8.1 UNION

A UNION operation lets you combine two select statements into one output so long as the SELECTs give output tables with the same number of columns, the syntax for UNION is:

```
SELECT
   column1
   column2
FROM
   table1
UNION
SELECT
   column3
   column4
FROM
   table2;
```

This kind of operation would let you for instance grab all the ids from two tables, or wages and names from two seperate department tables. The output takes the column names from the final SELECT, UNION also removes any full duplicate rows, if this isn't the desired behaivour you can use UNION ALL.

8.2 INTERESECT

INTERESECT behaves like UNION but takes only the intersection between the two tables, ie. the rows that are present in both.

8.3 MINUS

MINUS allows you to take the entries of the first SELECT that don't exist in the second SELECT, again with the same syntax as a UNION.

9 Managing Tables

9.1 CREATE TABLE

To create a new table you use the CREATE TABLE command, which takes column names, column datatypes and potentially restrictions on the data stored in the column in the form of a condition. The syntax is:

```
CREATE TABLE table_name (
   column1 datatype constraint,
   column2 datatype constraint,
   ...
);
```

If you run CREATE TABLE with a table name that already exists you'll get an error, but you can use IF NOT EXIST to condition the creation of the new table on that no table with that name exists already, and then you're free to decide how to handle that case yourself.

An example of a common constriant would be NOT NULL which will prevent any NULLS from being inserted into that column, and any attempts to alter the table in such a way that NULLs would be present will fail.

9.2 Primary Key

A primary key is the column, or set of columns that acts as a unique ID / hash for each row, a key then obviously has to be unique and not NULL (also each element in the key), but it also not allowed to be mutated. To sepcify a row or multiple rows as primary keys you can use the PRIMARY KEY specifier inside the table creation.

This could look like:

```
CREATE TABLE employees (
   id INT PRIMARY KEY,
   first_name STRING NOT NULL,
   last_name STRING NOT NULL
);
```

You can create a table without a PRIMARY KEY, but it's a bad habit, once you have a table without a PRIMARY KEY you can add it using ALTER TABLE with the ADD PRIMARY KEY subcommand as,

```
ALTER TABLE employees
ADD PRIMARY KEY (id);
```

9.3 DROP TABLE

To delete a table you use the DROP TABLE statement, simply write:

```
DROP TABLE table_name;
```

This is very dangerous since the table is gone forever as well as any information that isn't stored in another table.

9.4 IDENTITY

An IDENTITY column assigns a unique iterated INTEGER in the column which can be a very convienient way to get a good PRIMARY KEY, you can pass a number of arguments, just as usual the name and datatype (A kind of INTEGER), but also select if it's mutable or not, what value to start on, increment, wether to restart,

a minimum value, a maximum value, and if it's caching. The syntax is

```
column_name datatype GENERATED  ALWAYS | BY DEFAULT  ?AS IDENTITY? [options]
```

10 Subqeuries

10.1 Correlated Subqeuries

If you want each element to be subject to some criteria according to it's properties you end up with a correlated subquery. Take for instance the example below:

```
SELECT
   name,
   id,
   salary
FROM
   employees e1
WHERE
   salary > (
      SELECT
         AVG(salary) avg_salary
      FROM
         employees e2
      WHERE
         e2.department_id = e1.department_id
   )
```

In the query above, each entry is checked by it's department id, which is used to find the average wage within it's department, then only records who'se salaries are above the average for their department are selected.

10.2 ALL

If you want to check if a comparison holds for a given selected list of results you can use ALL. The syntax of ALL is:

value ALL comparison_operator (subquery)

An example use case would be if you want to check wether or not all of the managers make more than 100,000$. Then you could run the comparison

```
100000 ALL <= (
   SELECT
      salary
   FROM
      employees
   WHERE
      id = manager_id;
)
```

You could also use an ALL as part of making a query. This could be done for instance as below

```
SELECT
   name FROM
   employees
WHERE
   salary > ALL (        SELECT
         salary
      FROM
         employees
```

```
    WHERE
        department_id = 3
)
```

Which would return a list of all workers who'se salaries exceed all of the salaries in department 3, which is of course the same as exceeding the highest salary in department 3.

10.3 ANY

ANY runs the same type of check as ALL but returns true if any of the checks returns true.

10.4 EXISTS

EXISTS works very similarily to ALL and ANY but instead checks wether or not a query retruns at least one row. The syntax is

`EXISTS (query)`

Just like the above two this can also be used for a WHERE condition within a larger selection, where it would be checking some subquery.

11 Aggregating Functions

11.1 AVG

AVG takes, as the name implies, the average value of a set. The syntax for AVG is

`AVG([ALL|DISTINCT] expression)`

When run as part of a SELECT AVG can look like this:

```
SELECT
    AVG(averaged_column)
FROM
    table
...
```

AVG can also benifit from a GROUP BY, for instance

```
SELECT
    AVG(salary) avg_salary
FROM
    employees
GROUP BY
    department_id
```

which would give the average salary for each department.

11.2 ANY_VALUE

ANY VALUE, just like AVG is an aggregating function, so they work in very similar ways, but ANY VALUE will select an arbitrary value from the group, rather than perform any fancy operations.

11.3 COUNT

COUNT as the name implies counts entries in it's group.

11.4 MAX/MIN

MAX and MIN take the maximum and minimum value in a group represectively.

11.5 SUM

SUM takes the SUM of every (non NULL) entry in it's group.