Short Circuit

SQL

Summary by Emiel Bos

1 Introduction

SQL, or Structured Query Language, is a declarative, domain-specific programming language designed to query, manipulate, and transform data from a relational database. A *relational database* represents a collection of related (two-dimensional) tables, each with a fixed number of named columns (the attributes or properties of the table) and any number of rows of data. The database *schema* describes the structure of each table, and the datatypes that each column of the table can contain.

All popular *relational database management systems* (RDBMSs) – SQLite, MySQL, Postgres, Oracle and Microsoft SQL Server – support SQL. Because the SQL standard is large, complex, does not specify the database behavior in some important areas, defers some decisions to individual implementations, and is ambiguous for some construct semantics, all these RDBMS vendors use different implementations of SQL and therefore differ slightly in syntax, ruleset, and the kind of operators and functions they support. In particular, date and time syntax, string concatenation, `NULL`s, and comparison case sensitivity vary from vendor to vendor. PostgreSQL and Mimer SQL strive for standards compliance, though PostgreSQL does not adhere to the standard in all cases. The most popular commercial and proprietary RDBMSs are Oracle (whose `DATE` behaves as `DATETIME` and lacks a `TIME` type) and MS SQL Server (before the 2008 version). As a result, SQL code can rarely be ported between database systems without modifications.

SQL may be informally partitioned into sublanguages, which are commonly: data query language (DQL), data manipulation language (DML), data definition language (DDL), and data control language (DCL). Each SQL statement can be broadly classified as belonging to one of these, and we will use this classification.

2 Data query language

A *data query language* (DQL) are for performing queries on a database. SQL supports this with the `SELECT` statement. DQL is often considered a part of a data manipulation language (DML), i.e., the `SELECT` statement can also be considered a DML statement.

2.1 `SELECT`

To *query* a database, use the `SELECT` and `FROM` clauses, which yields a table:

```
SELECT col, another_col, ...
FROM table;
```

You can use $\star$ to select all columns. Capitalizing commands is simply a convention that makes queries easier to read, but isn't enforced. The SQL standard defines that statements end with the semicolon statement terminator, though it's not required on every platform. Also, SQL treats one space, multiple spaces, or a line break as being the same thing. Columns names are typically lower case, and use underscores instead of spaces, because else you need to use double quotes. You can define aliases for column and table names in your result with the `AS` keywords, although this keyword is optional and the SQL standard lets you use aliases with or without `AS`, but without `AS`, you cannot use a PostgreSQL keyword as alias. You can also use expressions to transform raw column values in a query with mathematical/arithmetic and string functions, and combine different column values (within the same row):

```
SELECT col AS Column, another_col AS "Another Column",
     (some_numerical_col + another_numerical_col) / 2.0 AS avg_col
FROM long_table_name AS table;
```

You can round floating point numbers with `ROUND(num, <precision>)`, e.g. `ROUND(num, 2)` for rounding to three decimals. You can get the length of a string with `CHAR_LENGTH(content)`. The `DATEDIFF(date1, date2)` function returns the difference between two dates as an integer. You can convert between datatypes in queries using two types of syntax that do the same: `CAST(column AS integer)` or `column::integer`.

You may specify multiple comma-separated table names in the `FROM` clause (and you may specify the same name multiple times, though you need aliases in that case), which will result in a Cartesian product of the given tables, i.e. every row from any table is paired with every row from any other table.

You can discard duplicate rows (taking only the values in the specified column(s) into account) by using the `DISTINCT` keyword (you add one `DISTINCT` keyword behind `SELECT`, so not per individual column name), so you get all unique combinations of values across those columns:

```
SELECT DISTINCT column, another_column, ...
FROM mytable
```

2.2 LIMIT

The less results returned, the faster a query runs. The `LIMIT` clause will reduce the number of rows to return, and the optional `OFFSET` clause will specify where to begin counting the number rows from (useful for pagination of results):

```
SELECT column, another_column, ...
FROM table
LIMIT 100 OFFSET 200;
```

2.3 WHERE

To filter results, use a `WHERE` clause with one or more conditions (concatenated with `AND` or `OR`) that are applied to each row:

```
SELECT column, another_column, ...
FROM table
WHERE
/* Numerical conditions */
     column >= 4 -- Standard numerical operators are supported
  AND column BETWEEN 1.5 AND 10.5 -- Number is within range of two values (inclusive)
  AND column >= 1.5 AND column <= 10.5 -- Same thing
  AND column NOT BETWEEN 1 AND 10 -- Number is not within range of two values (inclusive)
  AND column IN (2, 4, 6) -- Number exists in a list
  AND column NOT IN (1, 3, 5); -- Number does not exist in a list
/* Textual conditions */
  AND column = "abc" -- Case sensitive exact string comparison (notice the single equals)
  AND column != "abcd" -- Case sensitive exact string inequality comparison
  AND column LIKE "ABC" -- Case insensitive exact string comparison
  AND column NOT LIKE "ABCD" -- Case insensitive exact string inequality comparison
  AND column LIKE "%AT%" -- % matches a sequence of zero or more characters (e.g. "AT", "ATTIC")
  AND column LIKE "AN_" -- _matches exactly one character (e.g. "AND", but not "AN")
  AND column IN ("A", "B", "C") -- String exists in a list
  AND column NOT IN ("D", "E", "F") -- String does not exist in a list
```

In some variants of SQL `LIKE` is case-sensitive, and `ILIKE` is case-insensitive.

Full-text search is best left to dedicated libraries like Apache Lucene or Sphinx.

2.4 ORDER

You can sort your results by one or more columns in ascending (`ASC`, which is the default, so you can omit it) or descending (`DESC`) order using the `ORDER BY` clause. The results are sorted first by the column first specified, then the second, etc.:

```
SELECT column, another_column, ...
FROM table
ORDER BY column DESC, another_column ASC, ...;
```

2.5 JOIN

Databases are often broken down into pieces and stored across multiple orthogonal tables using a process known as *normalization*. The term "relational database" refers to the fact that the tables within it "relate" to one another; they contain common identifiers (*keys*) that allow information from multiple tables to be combined easily. This minimizes duplicate data in any single table, and allows for data in the database to grow independently of each other. Tables that share information about a single entity need to have a *primary key* – a column – that identifies that entity uniquely across the database. `JOIN` clauses in a query combine row data across two separate tables using this unique key. `INNER JOIN` matches rows from the first table and the second table which have the same key (as defined by the `ON` constraint) to create a result row with the combined columns from both tables. `INNER JOIN` is the default, so you can use `INNER JOIN` and `JOIN` interchangeably, but it's more readable if you write it out. A `LEFT JOIN` includes all rows from the table in the `FROM` clause regardless of whether a matching row is found in the table in the `LEFT JOIN` clause. Vice versa for `RIGHT JOIN`. In a `FULL JOIN`, rows from both tables are kept, regardless of whether a matching row exists in the other table. You might see queries with `LEFT/RIGHT/FULL OUTER JOIN`, but the `OUTER` keyword is really kept for SQL-92 compatibility and is usually left out. There is also `CROSS JOIN` for the Cartesian product, and this is the same as specifying multiple tables in the `FROM` clause. It has no `ON` clause because you're just joining everything to everything.

```
SELECT column, another_table_col, ...
FROM table AS t
   INNER/LEFT/RIGHT/FULL JOIN another_table AS at ON t.id = at.id AND at.id > 5;
```

Conditions in the `WHERE` clause are applied to the joined tables after the join, but you can filter one or both of the tables before joining them and create matches between the tables only under certain circumstances by adding additional conditions after `ON`, as above. You can also match on multiple foreign keys, which is mainly used for complex performance reasons. You may also join a table with itself, e.g. by matching every row on a date column with a row from a day before.

When working with multiple tables, column names across tables may have the same name. As demonstrated above, you can always prefix column names with the corresponding table or alias names, which is often necessary to prevent ambiguity errors, e.g. `t.id` or `table.*`.

`NULL`s indicate the absence of a value. When using `LEFT/RIGHT/FULL JOIN`, you will likely have to write additional logic to deal with `NULL`s in the result and constraints. An alternative to `NULL`s is to have data-type appropriate default values in your database, e.g. 0 for numerical data, empty strings for text data, etc. You can test a column for `NULL` values in a `WHERE` clause with `IS (NOT)NULL`. The `IFNULL(expression, alt_value)` function is useful for dealing with `NULL`s, which simply returns the given `expression` if it isn't `NULL`, and `alt_value` if it is, e.g. `IFNULL(maybenull, 0)`.

You can also concatenate multiple (different types of) `JOIN`s. `JOIN`s are left associative, i.e. each subsequent join joins a new table to the derived table that is the result of all the joins before it.

2.6 GROUP

You can use aggregate expressions (or functions) that allow you to summarize information about groups of rows, i.e. vertically, across rows:

```
SELECT AGG_FUNC(column_or_expression) AS aggregate_description, ...
FROM table
GROUP BY column1, column2, ...;
```

The `GROUP BY` clause works by grouping rows that have the same value(s) in the column(s) specified (so the order of column names doesn't matter, contrary to `ORDER BY`). Each aggregate function runs on each group, so you get as many results as there are unique groups/values in the specified column. Without a specified grouping, each aggregate function is going to run on the whole set of result rows and return a single value. `AGG_FUNCTION` can be any of `COUNT` (counts the number of rows with non-`NULL` values in the specified column; you can also use `COUNT(*)` to count all rows), `MIN`, `MAX`, `AVG` (ignores `NULL`s), `SUM` (treats `NULL`s as 0s).
The `GROUP BY` clause is executed after the `WHERE` clause, which filters the rows which are to be grouped. You can similarly filter the grouped rows with a `HAVING` clause, of which the constraints are written the same way as `WHERE` clauses; they're just applied to the grouped rows after the grouping rather instead of before.

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

You can use `DISTINCT` when performing an aggregation, e.g. `COUNT(DISTINCT month)` counts the unique values in the `month` column and would return 12. It's worth noting that using `DISTINCT`, particularly in aggregations, can slow your queries down quite a bit.

Every column specified in the `SELECT` clause must appear in `GROUP BY`, except columns which appear inside aggregate functions. This makes sense, because SQL wouldn't know how to combine values within each group that are not grouped on but which are `SELECT`ed, or which of those values within each groups to show.

2.7 UNION

When working with multiple tables, the `UNION (ALL)` operator allows you to append the rows of one query to those of another assuming that they have the same column count, order and data type. In other words, where `JOIN`s allow you to combine two tables side-by-side, `UNION`s allows you to stack one table on top of the other. Without the `ALL`, duplicate rows between the tables will be removed from the result. The `INTERSECT` operator will ensure that only rows that are identical in both result sets are returned, and the `EXCEPT` operator will ensure that only rows in the first result set that aren't in the second are returned (i.e. it's query order-sensitive, like `LEFT JOIN` and `RIGHT JOIN`). Both `INTERSECT` and `EXCEPT` also discard duplicate rows after their respective operations, though some databases also support `INTERSECT/EXCEPT ALL`. These operators are rarely used in practice.

```
SELECT column, another_column
FROM table

UNION (ALL)/INTERSECT/EXCEPT

SELECT other_column, yet_another_column
FROM another_table
```

2.8 Order of execution

Combining all the above ingredients in one query, we get the following blueprint, in which we denote the order of execution:

```
SELECT DISTINCT column, AGG_FUNC(column_or_expression), ... -- 5: expressions here are only
    executed at step 5
FROM mytable -- 1a: first executed to determine the total working set of data that is being queried
   JOIN another_table -- 1b: first executed to determine the total working set of data being queried
   ON mytable.column = another_table.column
WHERE constraint_expression -- 2: rows are filtered according to the applied constraints. Aliases
    in the SELECT part of the query are not yet accessible (in most RDBMSs); table aliases from
    FROM are available
GROUP BY column -- 3: the reminaing rows are grouped based on common values in the specified column
HAVING constraint_expression -- 4: groups are filtered according to the applied constraints.
    Aliases are still not yet accessible
-- 6: of the remaining rows, rows with duplicate values in the column marked as DISTINCT will be
    discarded
UNION (ALL)/INTERSECT/EXCEPT SELECT other_column, yet_another_column FROM another_table -- 7:
    append results
ORDER BY column ASC/DESC -- 8: rows are then sorted by the specified data. You can reference SELECT
    aliases in this clause
LIMIT count OFFSET COUNT; -- 9: rows that fall outside the range are discarded
```

2.9 CASE

SQL offers if-then-else functionality with `CASE` expressions, which can occur in `SELECT`, `GROUP BY` and `ORDER BY` clauses:

```
SELECT OrderID, Quantity,
(CASE
   WHEN Quantity > 30 THEN 'The quantity is greater than 30'
   WHEN Quantity = 30 THEN 'The quantity is 30'
   ELSE 'The quantity is under 30'
END AS QuantityText)
FROM OrderDetails
ORDER BY
(CASE
   WHEN City IS NULL THEN Country
   ELSE City
```

```
END);
```

2.10 Subqueries

You can use SQL *subqueries* for additional pre or post processing. A subquery must be fully enclosed in parentheses and can be referenced anywhere a normal table can be referenced: inside a `FROM` clause, as a `JOIN`, inside a `WHERE` or `HAVING` condition. When used in a condition, the subquery generally returns a one-cell result, except when using an `IN` condition. You can alias subqueries to reference their results. You can test expressions against the results of the subquery, and you can use subqueries in expressions in the `SELECT` clause, which allow you to return data directly from the subquery. They are generally executed in the same logical order as the part of the query that they appear in.

```
SELECT *
FROM sales_associates
WHERE salary >
   (SELECT AVG(revenue_generated)
    FROM sales_associates);
```

Correlated subqueries reference and are dependent on a column or alias from the outer query. Unlike regular subqueries, each of these inner queries need to be run for each of the rows in the outer query, since the inner query is dependent on the current outer query row:

```
SELECT *
FROM employees
WHERE salary >
   (SELECT AVG(revenue_generated)
    FROM employees AS dept_employees
    WHERE dept_employees.department = employees.department); -- The subquery needs to know what
        department each employee is in
```

3 Data manipulation language

A *data manipulation language* (DML) is for adding (inserting), deleting, and modifying (updating) data in a database. Querying is sometimes also considered a part of a DML.

3.1 `INSERT`

You can insert new rows into a table with `INSERT`, which declares which table to write into, the columns of data that we are filling, and one or more rows of data to insert. The number of values need to match the number of columns specified. If you don't specify any columns, each new row of data you insert should contain values for every corresponding column in the table. you can use mathematical and string expressions with the values that you are inserting:

```
INSERT INTO table (column1, column2) -- Specifying columns is optional
VALUES (row1_value_or_expr1, row1_value_or_expr2),
       (row2_value_or_expr1, row2_value_or_expr2 / 10); -- Can be any number of rows
```

3.2 `UPDATE`

`UPDATE` takes multiple column-value pairs and applies those changes to each and every row that satisfies the constraint in the WHERE clause. If you leave out the `WHERE` clause, you will update all rows.

```
UPDATE table
SET column = value_or_expr, other_column = another_value_or_expr
WHERE condition;
```

3.3 `DELETE`

`DELETE` is similar to `UPDATE`, but doesn't need `SET`. Omitting the `WHERE` clears the whole table.

```
DELETE FROM table
WHERE condition;
```

4 Data definition language

A *data definition language* (DDL) is a syntax for creating and modifying database objects such as tables, indices, and users. A *database schema* (but colloquially also referred to simply as a 'schema') is the structure/blueprint of a database; the tables, their columns/attributes and associated constraints and data types, and the relationships between tables. This kind of schema is typically defined in a `.sql` or `.ddl` file containing `CREATE TABLE` statements. Another example of a DDL is XML.

4.1 `CREATE TABLE`

You can create a new table with `CREATE TABLE`. Use `IF NOT EXISTS` to suppress errors and skip creating the table if one with the same name already exists. Every line inside the statement defines a column (or a table-wide constraint).

```
CREATE TABLE [IF NOT EXISTS] table (
   <column1> <datatype> <column_constraint>,
   <column2> <datatype> <column_constraint>
   [, <table_contraint>]
);
```

A `<datatype>` can be any of `INTEGER`, `BOOLEAN`, `FLOAT`, `DOUBLE`, `REAL`, `TEXT`, `CHARACTER(max_num_chars)`, `VARCHAR(max_num_chars)`, `DATE`, `DATETIME`, `DEFAULT <default_value>`, `BLOB` (Binary Large OBject, often opaque to the database, so you usually have to store them with the right metadata to requery them).

Constraints can be column-specific (in which case they're defined on the corresponding line of the column) or apply to the whole table (in which case they're defined on a separate line. Column constraints can refer to only one column, table constraints can refer to multiple columns. With the exception of `NOT NULL`, both types of constraint can be any of:

- `PRIMARY KEY`; the values in this column are unique, and each value can be used to identify a single row in this table. These are implicitly `NOT NULL` and `UNIQUE`. You can only have one primary key, but you can have multiple columns in your primary key by declaring it as a table constraint like `PRIMARY KEY(<some_column>, <another_column)`.
- `AUTOINCREMENT`; integer values are automatically filled in and incremented with each row insertion. Not supported in all databases.
- `UNIQUE`; the values in this column have to be unique, so you can't insert another row with the same value in this column as another row in the table.
- `NOT NULL`; inserted values can't be `NULL`. If they can be `NULL`, you can optionally specify the `NULL` column constraint to make this explicit.
- `CHECK(<expression>)`; checks the expression on inserted rows for validity. You can also put a `CHECK` on a separate row to enforce a condition on an entire record.
- `FOREIGN KEY`; a consistency check which ensures that each value in this column corresponds to another value in a column in another table.

You can optionally name constraints with the `CONTRAINT <contraint_name>` keyword right before the constraint definition. Whenever you want to remove a constraint with the `ALTER TABLE` command, you specify it with this name. If a name isn't supplied, the DBMS will automatically generate a constraint name, but you will have to look up its generated name in the database's metadata first if you ever want to remove it.

4.2 `ALTER TABLE`

You can update an existing table's schema with `ALTER TABLE`, which supports a number of operations. You can add columns:

```
ALTER TABLE table
ADD <another_column> <datatype> [column_constraint];
```

Default value constraints are applied to existing and new rows.
In some databases like MySQL, you can specify where to insert the new column using the `FIRST` or `AFTER` clauses, though this is not standard.

You can drop a constraint:

```
ALTER TABLE table
DROP CONSTRAINT <constraint_name>;
```

You can drop columns:

```
ALTER TABLE table
DROP column_to_be_deleted;
```

Some databases (including SQLite) don't support this feature, so you may have to create a new table and migrate the data over.

You can rename tables:

```
ALTER TABLE table
RENAME TO new_table_name;
```

4.3 DROP TABLE

You delete a table with:

```
DROP TABLE [IF EXISTS] table;
```

If you have another table that is dependent on columns in the table you are removing, then you will have to either remove the dependent rows from those tables or remove them entirely.

4.4 CREATE SCHEMA

Besides being used in a broader design sense to refer to the structure of a database's tables, the word 'schema' is also confusingly used to refer to the concept of a namespace in a database. MySQL uses the word 'schema' interchangeably with 'database', such that `CREATE SCHEMA` and `CREATE DATABASE` are aliases for the same command. In SQL Server and PostgreSQL, `CREATE DATABASE` really does create a database and `CREATE SCHEMA` creates a new schema entity in the current database.
In PostgreSQL a database contains one or more named schemas, which in turn contain tables and other kinds of named objects (e.g. data types, functions, views, operators, etc.). A schema can be considered a namespace; within one schema, two objects of the same type cannot have the same name. Whereas a client connection to a server can only access data in a single database (the one specified in the connection request), schemas are not rigidly separated, and therefore a user can access objects in any of the schemas in the database they are connected to (if they have the priviliges). You can use schemas to allow many users to use one database without interfering with each other, to organize database objects into logical groups to make them more manageable, or for putting third-party applications into separate schemas so they do not collide with the names of other objects. Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

```
CREATE SCHEMA some_schema;
```

You can refer to tables as `schema.table`, or, even more generally, as `database.schema.table`.

5 Data control language

A *data control language* (DCL) is used to control access to data stored in a database (authorization).

5.1 GRANT

Gives specified permissions for the table (and other objects) to, or assigns a specified role with certain permissions to, specified groups or users of a database.

```
GRANT SELECT, INSERT, UPDATE, DELETE on Employees TO User1
```

5.2 REVOKE

Takes away specified permissions for the table (and other objects) to, or takes away a specified role with certain permissions to, specified groups or users of a database.

```
REVOKE INSERT On Employees TO User1
```

5.3 DENY

Denies a specified permission to a security object.

```
DENY UPDATE On Employees TO User1
```