# Guide to SQL

This is a basic SQL reference guide. It is meant as a companion to the SQL lecture that will be delivered in class.

## 1 What is SQL?

Structured Query Language, or SQL, is a language used to communicate with a database. It is the standard language for a relational database, and is used for structuring, storing, manipulating, and retriving data. While SQL is a standard language, every database implements its own version of SQL with slightly different behaviours; in this class, we use SQLite3 since it is the easiest to set up and use. SQL usage between databases doesn't vary widely, so the skills you learn through SQLite3 will transfer readily to other database systems.

SQL is split into two main sublanguages; a Data Definition Language (DDL) and a Data Manipulation Language (DML). It can also be generated programmatically through other applications; for example, you may have worked with JDBC in the past.

For more help on SQL, check out the course textbook or W3Schools.

## 2 Tables

All data in a relational database is stored in **tables**. A table is not unlike an Excel table; each column defines an attribute name and type, and each row is an instance of the table's schema. A **schema** is a blueprint for what data in the given table should look like, including types and constraints.

To create a table in SQL, we use the `CREATE TABLE` syntax to define our table name, attribute names and types, as well as any constraints on our data. A sample table creation is below:

```
CREATE TABLE Person (
    id int PRIMARY KEY,
    full_name varchar(255) NOT NULL,
    age int
);
```

Here, we've defined a table called `Person` with three attributes, `id`, `full_name`, and `age`. The `id` attribute is of type `int` and is a **primary key** - this means that it uniquely identifies a row in the `Person` table and cannot be null. The `full_name` attribute if of type `varchar(255)` - a string of up to length 255 characters - and cannot be null. The `age` attribute is of type `int` and can be null.

If we define multiple tables, we can define **foreign key** relationships between them. A foreign key in one table references the primary key of another, and is the key behind the relationality of a relational database:

```
CREATE TABLE Dogs (
    id int PRIMARY KEY,
    pet_name varchar(255) NOT NULL,
    owner_id int REFERENCES Person(id)
);
```

To drop a table, use the `DROP TABLE <table_name>` syntax. This will delete the table and all of the data in it.

To alter a table, use the `ALTER TABLE <table_name>` syntax. You can `ADD`, `DROP`, and `RENAME` columns and constraints.

## 3 Constraints

Constraints are a way for SQL to check that certain properties always hold for your data. You've already seen some examples of constraints: `NOT NULL` and `PRIMARY KEY` are two such constraints. Constraints on a single attribute can appear after the attribute in table creation. Constraints on multiple attributes can appear outside of any column declaration. The following is an example of some common constraints:

```
CREATE TABLE Cats (
    id int PRIMARY KEY, -- UNIQUE and NOT NULL
    num_legs int NOT NULL, -- Cannot be NULL
    pet_name varchar(255) UNIQUE, -- Must be unique
    owner_id int DEFAULT 0, -- Specify a default value
```

Constraints can come after DDL:

```
CREATE TABLE Cats (
    id int,
    owner_id int,
    PRIMARY KEY (id),
    FOREIGN KEY (owner_id) REFERENCES Person(id)
```

And here's an example of table `CHECK` constraints:

```
CREATE TABLE Cats (
    id int PRIMARY KEY,
    num_legs int NOT NULL,
    CHECK (num_legs < 5)
```

## 4 Selects

Now that we understand how data is organized into tables, we can start selecting data from tables. Select queries in SQL follow a basic syntax:

```
SELECT <columns...>
    FROM <tables...>
    WHERE <filters...>;
```

For example, let's say I had the `Person` table above. I could select all of the rows and columns:

```
SELECT * FROM Person;
```

Only the `id` and `age` columns:

```
SELECT id, age FROM Person;
```

I could order my output or limit the number of rows emitted:

```
SELECT * FROM Person ORDER BY age [ASC|DESC];

SELECT * FROM Person LIMIT 10;
```

I can ask for non-duplicate output:

```
SELECT DISTINCT full_name FROM Person;
```

I can also rename columns when outputting:

```
SELECT id AS person_id FROM Person;
```

---

## 5 Inserts, Updates, and Deletes

To insert data into a table, use the `INSERT INTO` syntax, optionally specifying the order of the columns you'll be inserting:

```
INSERT INTO Person (id, full_name, age) VALUES (1, "Barry", 19);
INSERT INTO Person VALUES (1, "Barry", 19);
```

Updates can be done like so:

```
UPDATE Person SET age=20 WHERE id=1;
```

And deletes:

```
DELETE FROM Person WHERE id=1;
```

To delete all data, but leave the table intact:

```
DELETE FROM Person;
```

---

## 6 Filters and Aggregation

Specifying filters on your data is the job of the `WHERE` clause. You can use comparison operations (=, <, <=, >, >=), text operators (LIKE), range operators (BETWEEN), or many more (IS NOT NULL, IN). Each filter can be joined using logical operators `AND`, `OR`, and `NOT`. The following are some examples:

```
SELECT id
    FROM Person
    WHERE id < 10;

SELECT id
    FROM Person
    WHERE full_name = "Barry␣the␣Bee";

SELECT id
    FROM Person
    WHERE id BETWEEN 10 AND 100;

SELECT id
    FROM Person
    WHERE age IS NOT NULL;
```

```
SELECT id
    FROM Person
    WHERE full_name LIKE "B%"; /* Everything that starts with 'B' */
```

You can also aggregate your data using a suite of aggregation functions: `AVG`, `COUNT`, `MAX`, `MIN`, `SUM`, and more. To get the average of each subset of data, use the `GROUP BY` clause. The attribute you group by **must** be a selected column. To filter data based on aggregations, use the `HAVING` clause. The following are some examples:

```
/* Get the average age across all people. */
SELECT AVG(age) FROM Person;

/* Get the number of people in each age group. */
SELECT COUNT(*), age
    FROM Person
    GROUP BY age;

/* Get the age groups with more than 10 people in them. */
SELECT COUNT(*) AS num, age
    FROM Person
    GROUP BY age
    HAVING num > 10;
```

---

## 7 Joins

Given multiple tables, it may be beneficial to combine tables on some common attribute. Given the `Cat` and `Person` relations above, it might be useful to join the `Person(id)` attribute with the `Cat(owner_id)` attribute. To do so, SQL exposes a number of `JOIN` operations, each one preserving a different amount of data on either side of the join.

We'll focus on `CROSS JOIN` and `INNER JOIN`. The `CROSS JOIN` returns every possible combination of rows from the left and right table combined. If the left table had $m$ rows, and right right $n$ rows, expect to have $mn$ rows in the joined table. An example is below:

```
-- Get all pairs of people and cats
SELECT * FROM Person, Cat;
```

The `INNER JOIN` returns pairs of rows that share some common attribute. If a row on one of the two tables has no corresponding match in the other table, then it will simply not be included in the result. Other joins (`LEFT JOIN`, `RIGHT JOIN`, `OUTER JOIN`) will include elements with no matches. An example is below:

```
-- Get all owners and pets together
SELECT * FROM Person INNER JOIN Cat ON Person.id = Cat.owner_id;
```

The `NATURAL (INNER)JOIN` is an `INNER JOIN` that infers the columns that should be joined on.

---

## 8 Set Operations

Given the results of multiple subqueries, SQL allows you to combine the results using **set operations**. The `UNION (ALL)` operator adds the two result sets together, returning rows that are in either set (adding `ALL` preserves duplicates). The `INTERSECT` operator intersects the result sets, returning rows that are in both sets. The `MINUS` operator returns rows that are in the first set, but not the second. An example of how to use one of these operators is below:

```
-- Gets everybody below 10 years of age and above 18 years of age.
(SELECT *
    FROM Person
    WITH age < 10)
UNION -- Add the top set to the bottom set
(SELECT *
    FROM Person
    WITH age > 18)
```

---

## 9 Where to now?

There is a whole host of advanced SQL functions, including subqueries, views, triggers, indices, functions, recursion, and more. The textbook contains a decent overview of some more advanced SQL features and techniques. We will go over some of these more advanced features in class - see you there!

---

## Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!