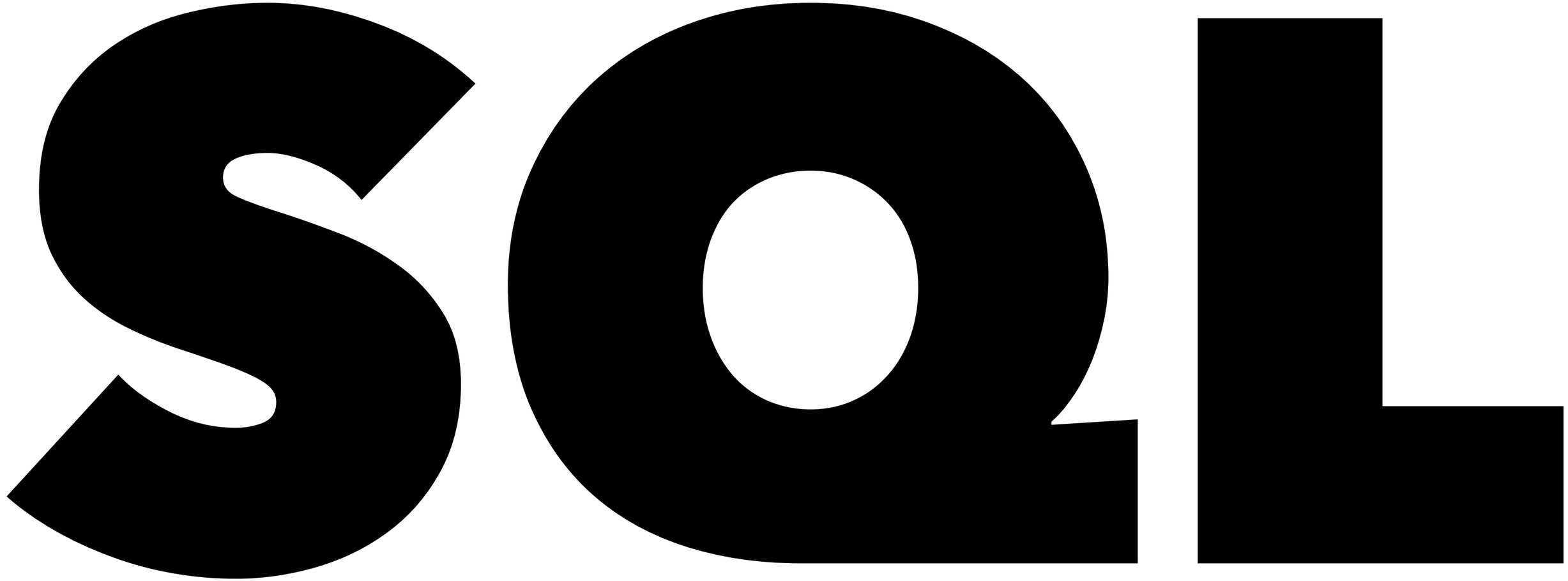
SQL

SQL (Structured Query Language) is a descriptive computer language designed for updating, retrieving, and calculating data in table-based databases.

Mozilla Developer Network

Been around since the early 70s.

Has features we've been looking for

- Structured
- Random Access (don't have to rewrite or append)
- Can store multiple related data in one place
- Supports multiple users

Has other features: ACID

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity

Updates to the database allow for multiple changes to execute at once. Either all of them fail, or all of them succeed.

Consistency

Any rules the database enforces are applied when asking for changes.

Isolation

Isolation allows for multiple database requests to be handled concurrently (e.g., reading and writing to multiple tables/rows at the same time).

Durability

Durability ensures that once data is written to the database, the database does not lose the information in the case of a database, computer, or system crash.

Once a database says that row updates, a power loss to the system should not lose the updated data.

SQL Basics

- SQL (relational) databases store and arrange data into tables.
- These tables consist of rows where each row has the same set of columns.

For an analogy, you can think of a database as an Excel spreadsheet, a table as "sheet" within that spreadsheet, rows as the rows of the sheet, and columns as named versions of the familiar "Column A", "Column B" style of spreadsheets.

Tables

- Tables are the containers for our data.
- Usually, a database has multiple tables, one for each "thing" we are storing.

For instance, let's imagine we are designing and building a system to manage books for a library. We could create a database named `Library` and in that database, there would be one table, called `Books`.

In this `Books` table, we would design columns representing the specific data about books we wish to track. Each row in the `Books` table would represent a unique book in our collection.

Columns

- Columns are the part of the table that defines the structure (what we often call the schema).
- This is where we define the attributes of the "thing" represented by the table.
- Every column has a data type that defines and restricts what type of data we can place into each column.

- In our `Books` table, we will want to store specific details:
- `Title`
- `PrimaryAuthor`
- `YearPublished`
- `Genre`

In a language such as JavaScript, Ruby, or C# we would naturally define some of these as a `string`.

In a database, we have a few choices for the data type.

`char(N)`

The `N` represents the largest number of characters this column can store. If we supply *less* than `N` characters, spaces will fill the remaining space. The appended spaces ensures the column is *always* `N` characters long.

`varchar(N)`

Again the `N` represents the largest number of characters the column can store. However, the width of the data is variable. If we supply *less* than `N` characters, the column is *not* filled with spaces.

`text`

Allows for a variable number of characters but has a much larger limit than what a `char` or `varchar` can support. In some cases many megabytes, or gigabytes of text.

In our case, `varchar` or `text` make the most sense for our columns.

See the Postgres docs for more details.

Rows

Rows are where our data is stored.

Each row represents one instance of a "thing", in our case one "book".

Much like one `object` *is an instance of a class (this is an important concept).*

In our example, a row of data would contain 'The Cat in the Hat' (`Title`), 'Dr. Suess' (`Author`), and 'kids' (`Genre`).

Getting started with Postgres

You may choose to follow along

All the commands are in the `Introduction to SQL` *lesson, so feel free to copy + paste.*

Create our first database

First, we need to create a database:

```
createdb Library
```

We can run this command from any directory, it doesn't matter.

This creates a new, empty database.

But where is it?

The databases, tables, columns, rows, relationships, and data are stored and managed by the database.

We do not see where it keeps the data or how the database structures the data.

Eventually, the data won't even be on the same computer where our code runs!

Connecting

To connect to that database and start running queries against it, use the command:

```
pgcli Library
```

REPL

- Read
- Evaluate
- Print
- Loop

Like our command terminal/shell but now the commands go to the database instead of our operating system.

Queries

SQL databases use the Structured Query Language to both define the schema (structure) of our database and as a way to create, read, update, and delete data within it.

We call the statements we ask a database to do for us a query, even if the statement's purpose is to create tables, or delete rows.

CREATE TABLE

Let's start by creating the table with only the `Title`, `PrimaryAuthor`, and the `YearPublished` columns for our `Books`.

```
CREATE TABLE "Books" (
    "Title"           TEXT NOT NULL,
    "PrimaryAuthor"   TEXT,
    "YearPublished"   INT,
    "Id"              SERIAL PRIMARY KEY
);
```

Break it down

- `TEXT`
- `NOT NULL`
- `INT`
- `SERIAL PRIMARY KEY`

Other types

Type	Description
BOOLEAN	Stores a `true` or `false` value.
DATE	Stores a year, month, and day together. Use `YYYY-MM-DD` format such as `'2020-07-04'` when adding data.
TIMESTAMP	Stores a precise time, Use `YYYY-MM-DD HH:MM:DD` format such as `'2020-07-04 15:45:12` when adding data.

ALTER TABLE

The structure of our tables is not set in stone.

Can be modified at a later date by using the `ALTER TABLE` query.

```
ALTER TABLE "Books" ADD COLUMN "Genre" VARCHAR(15);
```

INSERT

To create a new row in our database, we need to use `INSERT`.

The format for `INSERT` statements is similar to:

EXAMPLE ONLY

```
INSERT INTO "TableName" ("ColumnA", "ColumnB", "ColumnC")
VALUES ('columnAValue', 'columnBValue', 'columnCValue');
```

Break it down

- Table Name
- List of columns
- List of values in the same order

Sample data

```
INSERT INTO Books (Title,  PrimaryAuthor, YearPublished, Genre)
VALUES ('Night of the Living Dummy', 'R. L. Stine', 1993, 'horror');

INSERT INTO Books (Title,  PrimaryAuthor, YearPublished, Genre)
VALUES ('A Shocker on Shock Street', 'R. L. Stine', 1995, 'horror');

INSERT INTO Books (Title,  PrimaryAuthor, YearPublished, Genre)
VALUES ('The Lost World', 'Michael Crichton', 1995, 'sci-fi');
```

SELECT

SELECT statements allow us to query and return a new view of the data.

```
SELECT * FROM "Books";
```

This query will give us back all the columns (*) from all the rows in the Books table.

Regardless if there are ten rows or ten million rows, this statement will return them all.

SELECT (some of the columns)

Often we do not want *all* the columns from the table. We can specify specific columns.

```
SELECT "Title", "PrimaryAuthor" FROM "Books";
```

While this will still return all the rows, we will only see the `Title` and `PrimaryAuthor` columns for all those rows.

Look familiar?

SQL

```sql
SELECT "Title" FROM "Books";
```

C#

```
books.Select(book => book.Title);
```

Wait. It gets better.

We can also do computations with `SELECT`

See the number of books:

```
SELECT COUNT(*)
FROM "Books";
```

See the average, largest, and smallest year of publication.

```
SELECT AVG("YearPublished"), MAX("YearPublished"), MIN("YearPublished")
FROM "Books";
```

`SELECT` and `WHERE` together!

We can use the `WHERE` clause to help filter down our table to only see rows that satisfy the conditions supplied.

SQL

```sql
SELECT "Title", "PrimaryAuthor" FROM "Books" WHERE "Genre" = 'horror';
```

C#

```
books.
    Where(book => book.Genre == "horror").
    Select(book => book.Title);
```

```sql
SELECT * FROM "Books";

SELECT "Title", "PrimaryAuthor" FROM "Books";
SELECT "Title", "PrimaryAuthor" FROM "Books" WHERE "Genre" = 'horror';
SELECT "Title", "PrimaryAuthor" FROM "Books" WHERE "Genre" = 'fantasy' OR "Genre" = 'sci-fi';
SELECT "Title", "PrimaryAuthor" FROM "Books" WHERE "Genre" = 'horror' ORDER BY "Title";
SELECT "Title", "PrimaryAuthor" FROM "Books" WHERE "Title" LIKE 'The Lord of the Rings%' ORDER BY "Title";
```

Aliases using the "AS" keyword

Sometimes we want to use a different name for a column than the name in the table itself. SQL allows us to do this with the AS keyword.

We can use an alternate name for a column by using the AS keyword.

```
SELECT "Title" AS "BookTitle" FROM "Books";
```

We can also rename the table itself:

```
SELECT "Title" AS "BookTitle" FROM "Books" AS "BooksTable";
```

UPDATE

To change data in our database, we use an `UPDATE` statement. The general structure of an `UPDATE` is:

```
UPDATE "TableName"
SET "ColumnA" = 'new value'
WHERE "ColumnB" = 'some value';
```

The above update statement will update all rows that have `ColumnB = 'some value'` and change `ColumnA` to contain `'new value'`.

NOTE

The same syntax for `WHERE` clauses from `SELECT` apply here. We can use multiple conditions and combine them with `AND` and `OR`.

Word of warning. If the `WHERE` *clause is left off, it will update all rows.*

This might cause a **bad day**

Example:

```
UPDATE "Books" SET "Genre" = 'children horror' WHERE "PrimaryAuthor" = 'R. L. Stine';
```

DELETE

To remove one **or many** rows, we can use the `DELETE` statement.

```
DELETE FROM "Books" WHERE "YearPublished" = 1995;
```

As with the `UPDATE` statement, the `DELETE` statement can use the same `WHERE` syntax to filter rows to delete.

Word of warning. If the `WHERE` *clause is left off, it will delete all rows.*

This might cause a **bad day**

References

Here are some useful postgres database references about SQL

- PostgreSQL Data Types
- PostgreSQL Documentation