SQL (and Node.js)

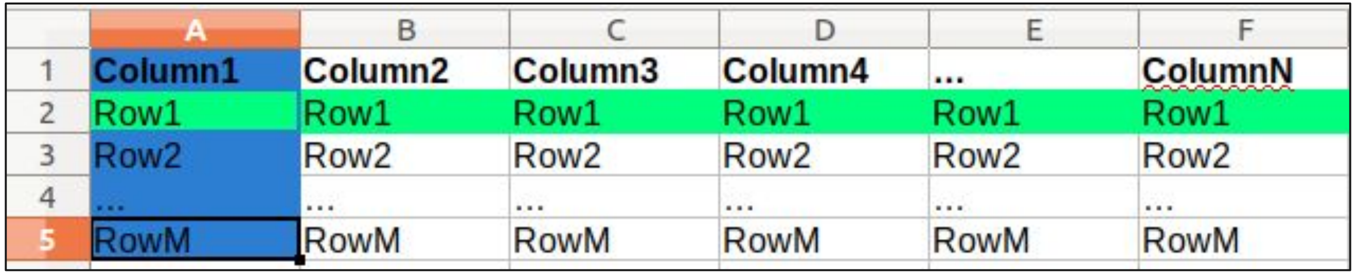
SQL

A Relational Database

Relational Database: A method of structuring data as tables associated by shared attributes

A table row corresponds to a unit of data called a **record**; a column corresponds to an attribute of that record

In Excel-speak:

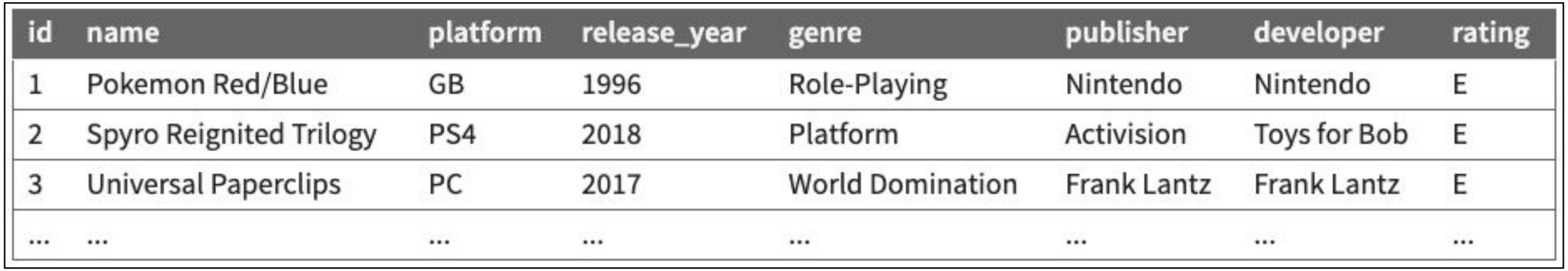


	A	B	C	D	E	F
1	**Column1**	**Column2**	**Column3**	**Column4**	**...**	**ColumnN**
2	Row1	Row1	Row1	Row1	Row1	Row1
3	Row2	Row2	Row2	Row2	Row2	Row2
4	...	...	...	...	...	...
5	RowM	RowM	RowM	RowM	RowM	RowM

In the above image, the cells highlighted blue are the first "column" and the cells highlighted green are of the first "row" (or "record", or "tuple")

Tables can be visualized just like an Excel sheet, just with different terminology, and more programmatic capabilities.

Example Database



id	name	platform	release_year	genre	publisher	developer	rating
1	Pokemon Red/Blue	GB	1996	Role-Playing	Nintendo	Nintendo	E
2	Spyro Reignited Trilogy	PS4	2018	Platform	Activision	Toys for Bob	E
3	Universal Paperclips	PC	2017	World Domination	Frank Lantz	Frank Lantz	E
...	...	...	...	...	...	...	...

Structured Query Language (SQL)

A "domain-specific language" (HTML is also a DSL) designed specifically for data access and management.

Also like HTML, SQL is a **declarative** language: describes what data you are seeking, not exactly how to find it.

In SQL, you write statements. The main different types of statements we'll look at:

- **Data Definition**: Generally, what does your data look like?
- **Data Manipulation**: Change or access the data.

(There are others, but we won't be talking about them.)

SQL Basics

```
SELECT name FROM menu WHERE qty > 0;
INSERT into menu (name, category, qty, image)
VALUES ("Cookie", "Desserts", 154, "cookie.png");
```

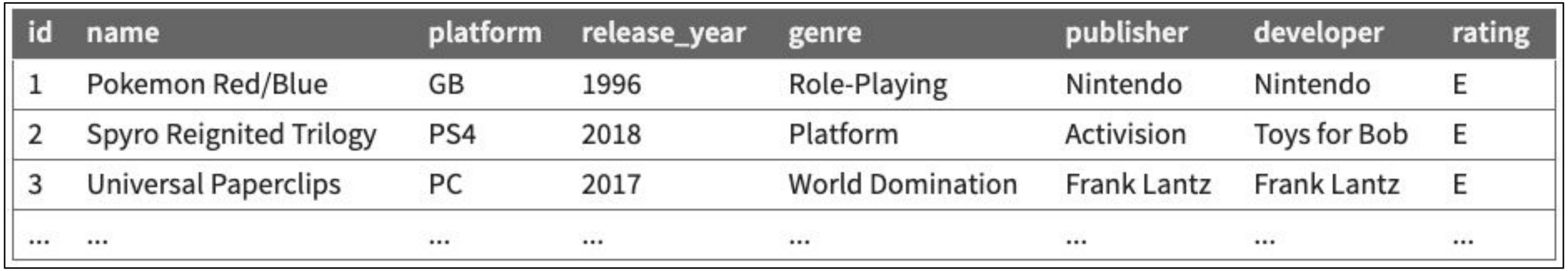
Structured Query Language (SQL): A language for searching/updating a database.
A standard syntax that is used by all database software (with minor variations). Generally case-insensitive.

Some Basic SQL Statements

- SELECT
- DISTINCT
- WHERE
- LIKE
- ORDER BY
- LIMIT

These are the basic "building-blocks" of forming "questions" (queries) in SQL

Example Database



id	name	platform	release_year	genre	publisher	developer	rating
1	Pokemon Red/Blue	GB	1996	Role-Playing	Nintendo	Nintendo	E
2	Spyro Reignited Trilogy	PS4	2018	Platform	Activision	Toys for Bob	E
3	Universal Paperclips	PC	2017	World Domination	Frank Lantz	Frank Lantz	E
...	...	...	...	...	...	...	...

The SQL `SELECT` Statement

Syntax:

```
SELECT column(s) FROM table;
```

Example:

```
SELECT name, release_year FROM Games;
```

Example output:

name	release_year
Pokemon Red/Blue	1996
Spyro Reignited Trilogy	2018
Universal Paperclips	2017
Super Mario Bros.	1985
...	...

The `SELECT` statement is used to return data from a database.
It returns the data in a result table containing the row data for column name(s) given. Table and column names are case-sensitive.

The `DISTINCT` Modifier

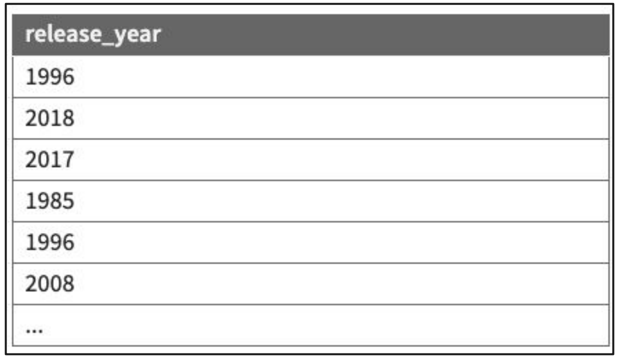
Syntax:

```
SELECT DISTINCT column(s) FROM table;
```

The `DISTINCT` modifier eliminates duplicates from the result set.

Example (without `DISTINCT`):

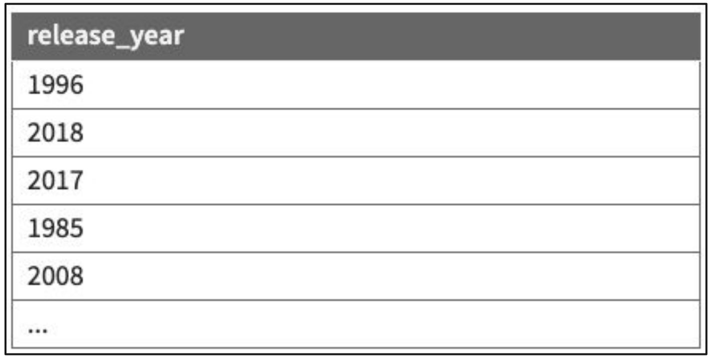
```
SELECT release_year
FROM Games;
```



release_year
1996
2018
2017
1985
1996
2008
...

Example (with `DISTINCT`):

```
SELECT DISTINCT release_year
FROM Games;
```



release_year
1996
2018
2017
1985
2008
...

The SQL **WHERE** Statement

Syntax:

```
SELECT column(s) FROM table WHERE condition(s);
```

Example:

```
SELECT name, release_year FROM Games WHERE genre = 'puzzle';
```

Example output:

name	release_year
Tetris	1989
Brain Age 2: More Training in Minutes a Day	2005
Pac-Man	1982
...	...

The `WHERE` clause filters out rows based on their columns' data values. In large databases, it's critical to use a `WHERE` clause to reduce the result set in size.
Suggestion: When trying to write a query, think of the `FROM` part first, then the `WHERE` part, and lastly the `SELECT` part.

More about the `WHERE` Clause

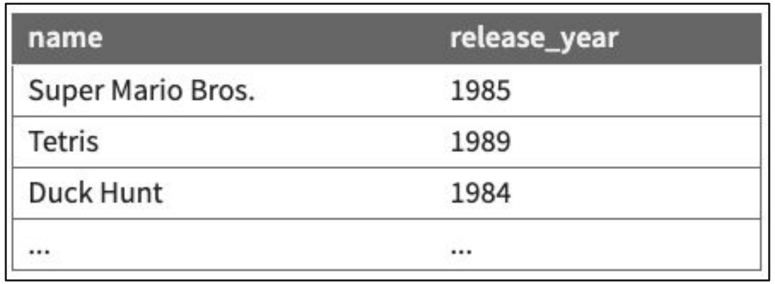
Syntax:

```
WHERE column operator value(s)
```

Example:

```
SELECT name, release_year
FROM Games
WHERE release_year < 1990;
```

Example result:



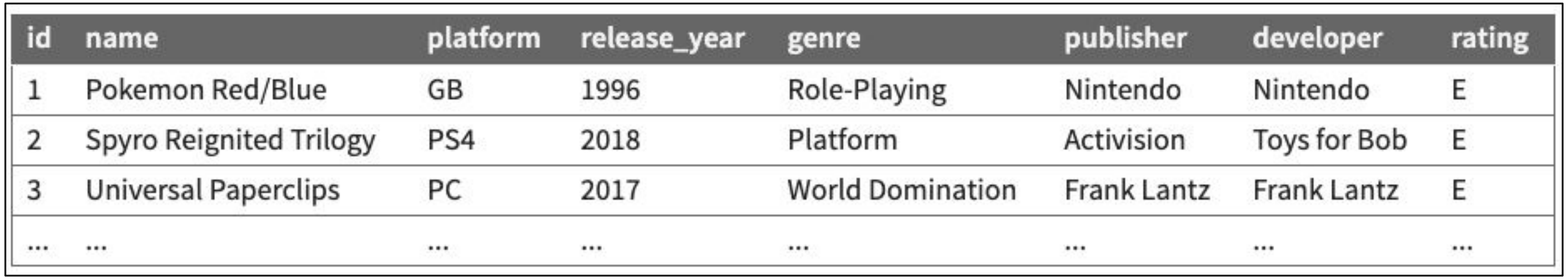
name	release_year
Super Mario Bros.	1985
Tetris	1989
Duck Hunt	1984
...	...

The `WHERE` portion of a `SELECT` statement can use the following properties:

- =, >, >=, < <=
- <> or != (not equal)
- `BETWEEN` min `AND` max
- `LIKE` pattern
- `IN` (value, value, ..., value)

Check your Understanding

Write a SQL query that returns the `name` and `platform` of all games with a `release_year` before 2000.



id	name	platform	release_year	genre	publisher	developer	rating
1	Pokemon Red/Blue	GB	1996	Role-Playing	Nintendo	Nintendo	E
2	Spyro Reignited Trilogy	PS4	2018	Platform	Activision	Toys for Bob	E
3	Universal Paperclips	PC	2017	World Domination	Frank Lantz	Frank Lantz	E
...	...	...	...	...	...	...	...

```
SELECT name, platform
FROM Games
WHERE release_year < 2000;
```

Multiple WHERE Clauses; AND, OR

Example:

```
SELECT name, release_year FROM Games
WHERE release_year < 1990 AND genre='puzzle';
```

Example output:

name	release_year
Tetris	1989
Pac-Man	1982
Dr. Mario	1989
...	...

Multiple `WHERE` conditions can be combined using `AND` or `OR`.

Approximate Matches with `LIKE`

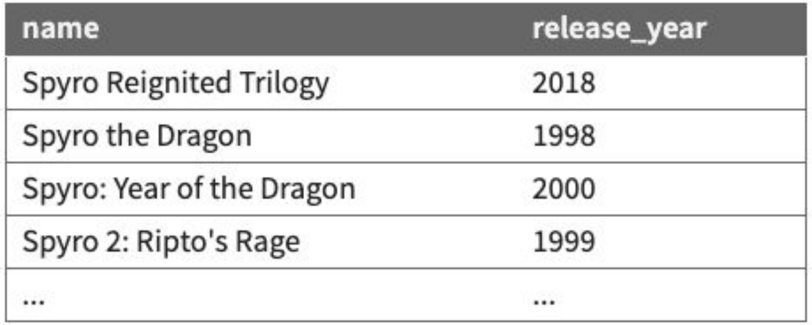
Syntax:

```
WHERE column LIKE pattern
```

Example:

```
SELECT name, release_year
FROM Games
WHERE name LIKE 'Spyro%'
```

Example result:



name	release_year
Spyro Reignited Trilogy	2018
Spyro the Dragon	1998
Spyro: Year of the Dragon	2000
Spyro 2: Ripto's Rage	1999
...	...

- `LIKE 'text%'` searches for text that starts with a given prefix
- `LIKE '%text'` searches for text that ends with a given suffix
- `LIKE '%text%'` searches for text that contains a given substring

- Note: In SQLite, the text in the LIKE string is case-insensitive.

Sorting By a Column: `ORDER BY`

Syntax:

```
SELECT column(s) FROM table
ORDER BY column(s) ASC|DESC;
```

Example (ascending order by default):

```
SELECT name FROM Games
ORDER BY name
```

name
'98 Koshien
007 Racing
007: Quantum of Solace
007: The World is not Enough
...

Example (descending order):

```
SELECT name FROM Games ORDER BY
name DESC;
```

name
Zyuden Sentai Kyoryuger: Game de Gaburincho
Zwei
Zumba Fitness: World Party
Zumba Fitness Rush
...

The `ORDER BY` keyword is used to sort the result set in ascending or descending order (ascending if not specified)

Limiting Rows with `LIMIT`

Syntax:

```
LIMIT number
```

Example:

```
SELECT name FROM Games
WHERE genre='puzzle'
ORDER BY name
LIMIT 3;
```

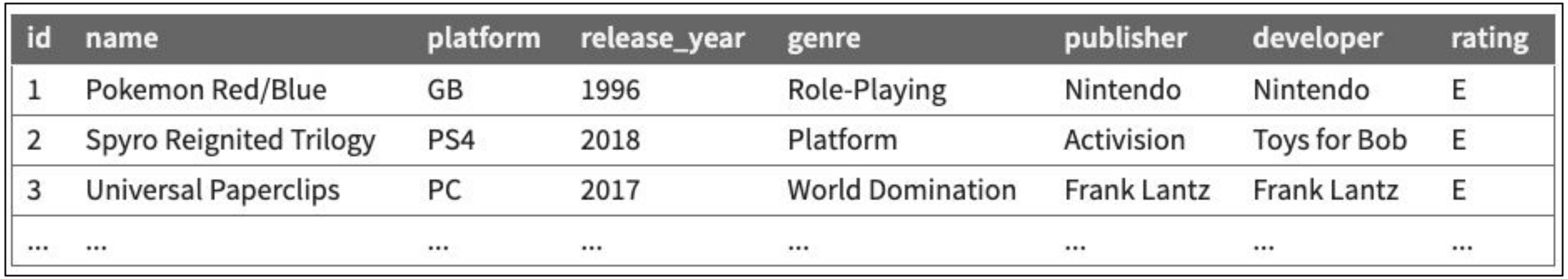
Example result:

name
100 All-Time Favorites
101-in-1 Explosive Megamix
3D Lemmings

LIMIT can be used to get the top-N of a given category. It can also be useful as a sanity check to make sure you query doesn't return 100000 rows.

Check your Understanding

Write a SQL query that returns the `name` and `platform` of all games with the word 'dragon' in them, ordered by `release_year`.



id	name	platform	release_year	genre	publisher	developer	rating
1	Pokemon Red/Blue	GB	1996	Role-Playing	Nintendo	Nintendo	E
2	Spyro Reignited Trilogy	PS4	2018	Platform	Activision	Toys for Bob	E
3	Universal Paperclips	PC	2017	World Domination	Frank Lantz	Frank Lantz	E
...	...	...	...	...	...	...	...

```
SELECT name, genre
FROM Games
WHERE name LIKE '%dragon%'
ORDER BY release_year;
```

Additional Practice with SQL Queries

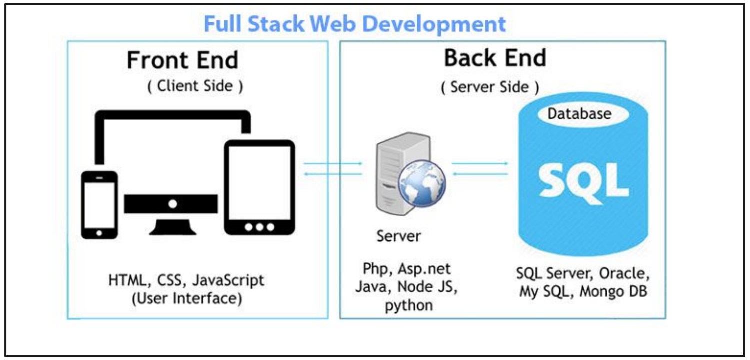
SQLZoo has multiple exercises (with built-in databases you don't need to worry about setting up) for practicing `SELECT`, `WHERE`, `ORDER BY`, `LIMIT`, `LIKE`, etc. We recommend you go through these for additional practice!

The Node.js and SQL Connection

Full-stack website organization

A full-stack website consists of three layers:

- Web Browser (client): HTML, CSS, JS
- Web Server: Node.js
- Database Server: SQL



The web server makes requests of the database server much like our client-side JS used AJAX
The `sqlite` module will handle most of this for us.

`sqlite` vs `sqlite3`

- `sqlite3` is the most popular module used to connect to a SQLite database in Node.js
- Similar to the `fs` module, `sqlite3` functions use callbacks for results and errors
- We will use `sqlite` which is a wrapper for sqlite that adds promises

Using SQL in Node.js

First install the `sqlite3` and `sqlite` modules in your project.

- `npm install sqlite3 sqlite`

Then require both modules with the rest of your modules in your Node.js program.

- `const sqlite3 = require("sqlite3");`
- `const sqlite = require("sqlite");`

SQL Connection

```
'use strict';
const sqlite3 = require('sqlite3');
const sqlite = require('sqlite');

* Establishes a database connection to the wpl database and returns the database
object.
* Any errors that occur during con. should be caught in the function that calls this
one.
* @returns {Object} - The database object for the connection.
*/
async function getDBConnection() {
    const db = await sqlite.open({
        filename: 'data/wpl.db',
        driver: sqlite3.Database
    });

    return db;
```

Executing SQL queries with the `db` object

Once you have the `db` object, you can now execute SQL queries with `db.all`
This function takes a SQL query string and an optional array of parameters/placeholder values and returns the resulting rows.

```
let rows = await db.all(sqlString)
```

```
let rows = await db.all("SELECT name, category FROM menu ORDER BY
name;");
```

More about db.all(sqlString)

The query function returns an array of JS objects, which represent information for each row matching the query. In the below example, we limit at most 2 rows in the result.

```
let qry = "SELECT name, type FROM pokedex LIMIT 2;";
let rows = await db.all(qry);
console.log(rows);
```

```
[
  { name: 'Bulbasaur', type: 'grass' },
  { name: 'Ivysaur', type: 'grass' }
]
```

Extracting the data

The column (field) names for each row (record) can be accessed using dot notation (it's just an object!)

```
...
let qry = "SELECT name, type FROM pokedex LIMIT 2;";
let rows = await db.all(qry);
let firstRow = rows[0];
console.log("Name: " + firstRow.name + "(" + firstRow.type + ")");
```

```
Name: Bulbasaur (grass)
```

Note that only the column names specified in the SELECT statement will be accessible (for this example, `name` and `type`)

`try/catch`

Database connections can have different problems: the database server could be down, the database could be missing or corrupted, the user/password credentials may be incorrect.

You can find a good review of SQLite error codes here - useful when you are debugging and/or want to handle errors differently (similar to how we can use `err.code === "ENOENT"` in our response logic)

`try/catch` helps us catch and identify when errors occur so we can handle the error correctly.

Using try/catch with `sqlite` functions

You can `try/catch` just like you would for `fs.readFile()`, catching any errors that occur in the `db.all` function.

```
try {
  let rows = await db.all("SELECT name FROM pokedex"); // error could
happen here
  // process the result rows somehow
} catch (error) {
  res.status(500).send("Error on the server. Please try again
later.");
}
```

Using SQL in Node

In order to connect and open our database so that we have access to query the tables, we need to include the lines of code below. Remember to also `npm install` so that your package.json is updated with the proper dependencies

```
// top of app.js
const sqlite3 = require('sqlite3');
const sqlite = require('sqlite');

// somewhere in your app.js (after all
// endpoint definitions)
async function getDBConnection() {
  const db = await sqlite.open({
      filename: '<db-file-name>.db',
      driver: sqlite3.Database
  });
  return db;
}
```

←

Make sure to replace this with the name of your database file!