Lecture 9

Databases and MySQL

Introduction to Databases

- Database Types
- Database Systems
- MySQL Basics
- SQL Language

What is a Database?

A collection of related information, similar to JSON (but more tabular)

What are some examples of data you could store in a database?

- Pokedex
- Book Reviews
- Store Inventory
- User Information
- Movies/Games
- Cafe Menu
- ...

Why Learn Databases in this course?

Databases give us a great improvement in the way we can build, process, and retrieve large datasets. Most software companies will have a large group dedicated to database management.

Advantages of a database:

- **Powerful**: Can search it, filter data, combine data from multiple sources.
- **Fast**: Can search/filter a database very quickly compared to a file/JSON.
- **Big**: Scale well up to very large data sizes.
- **Safe**: Built-in mechanisms for failure recovery (e.g. transactions).
- **Multi-user**: Concurrency feature let many users view/edit data at the same time.
- **Abstract**: Provides layer of abstraction between stored data and app(s) - many database programs understand the same commands.

Database Software

- Oracle
- Microsoft SQL Server (powerful) and Microsoft Access (simple)
- PostgreSQL (powerful/complex free open-source database system)
- SQLite (transportable, lightweight free open-source database system)
- MySQL (simple free open-source database system)

Other types of databases:

- NoSQL
- Key/Value
- Blob/Document
- HBase, distributed databases
- Timeseries

Relational Database vs. Others

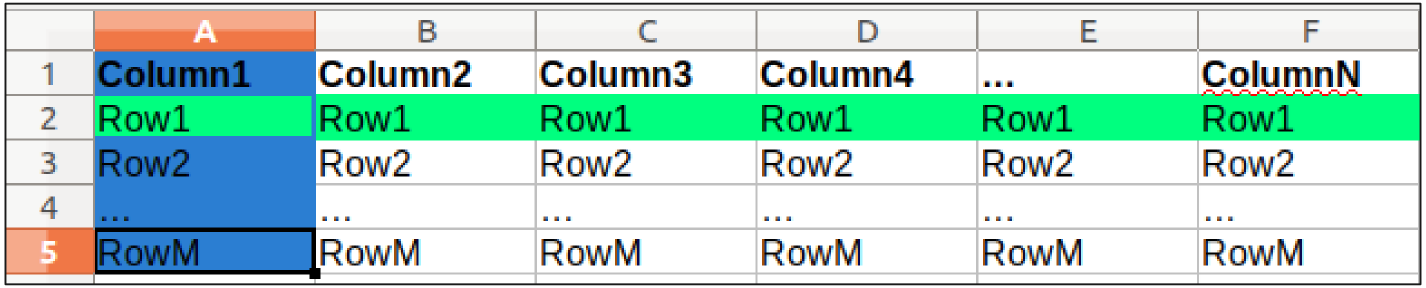
- Many different ways to store data (see previous slide)
- Relational databases define tabular data (data stored in tables), with operations that allow you to relate them to each other.
- Modern RDBMSs (**R**elational **D**ata**B**ase **M**anagement **S**ystem) are transactional.
- This allows the database to ensure that certain expectations about your data are met:
  - Each transaction is made up of 1+ commands, and they either *all* succeed or all fail. No in-between or partial success states.

A Relational Database

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

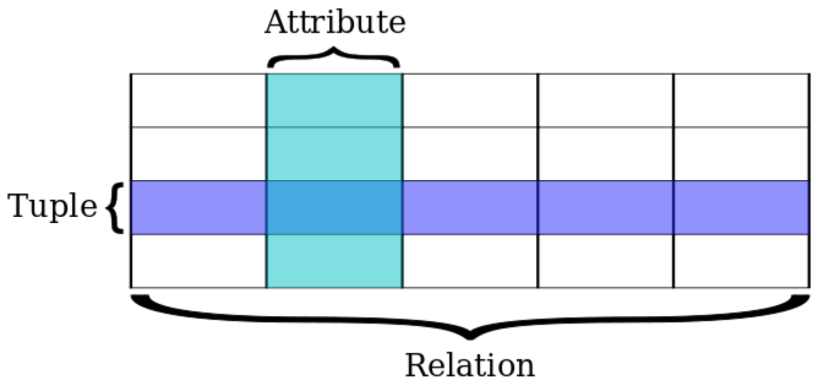
- A table row corresponds to a unit of data called a **record** (green) or tuple.
- a column corresponds to an attribute of that record (blue)

In Excel-speak:

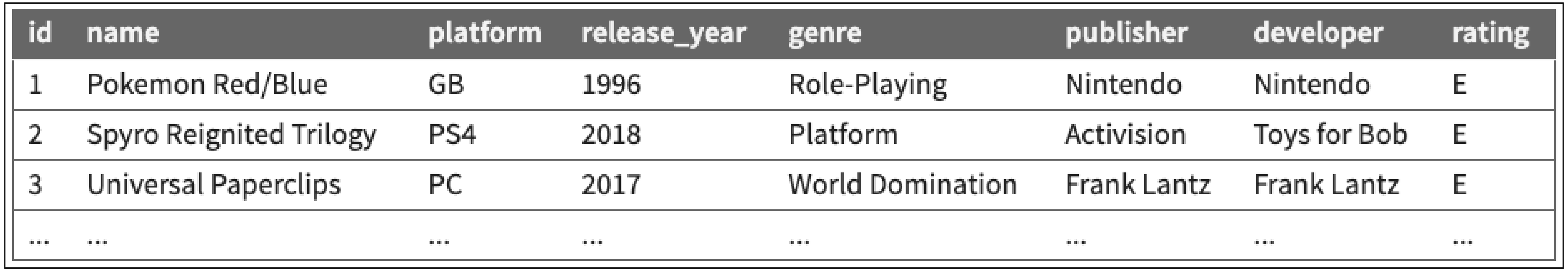


	A	B	C	D	E	F
1	**Column1**	**Column2**	**Column3**	**Column4**	**...**	**ColumnN**
2	Row1	Row1	Row1	Row1	Row1	Row1
3	Row2	Row2	Row2	Row2	Row2	Row2
4	...	...	...	...	...	...
5	RowM	RowM	RowM	RowM	RowM	RowM

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



Example Table



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.
- SQL is a **declarative** language: describes what data you are seeking, not exactly how to find it.
    - HTML: markup language, JavaScript: interpreted imperative language
- 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

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

`CREATE TABLE`: Syntax

`CREATE TABLE` is used to create a new table.

```
CREATE TABLE table_name(
  column1 datatype PRIMARY KEY,
  column2 datatype,
  column3 datatype,
  .....
  columnN datatype
);
```

Types in MySQL

MySQL has 5 many data types. But these types are commonly used:

- Numeric types: INT, TINYINT, FLOAT, DOUBLE, DECIMAL, ...
- String types: CHAR, VARCHAR, TEXT, ...
- Date & Time: DATE, DATETIME, TIMESTAMP, ...

See a full list on the official documentation.

MySQL Data Types

The following are commonly used MySQL types:

Type	Argument	Description
`INT`	none	An integer number
`VARCHAR(n)`	The string can be at most `n` characters (max of 65,535).	A text string
`TINYINT`	none	Stores only very small numbers (definition of "small" varies, but is usually between 0 and 255)
`DATETIME`	none	Stores dates and times for precise time information: YYYY-MM-DD HH:MM:SS
`DOUBLE`	Optionally `(m,d)`, specifying how many digits in the number.	A decimal number. Rounds based on the provided precision.
`TEXT`	none	A potentially very large text string.

`CREATE TABLE`: Syntax

What is this?

```
CREATE TABLE table_name(
  column1 datatype PRIMARY KEY,
  column2 datatype,
  column3 datatype,
  .....
  columnN datatype
);
```

Useful Column Constraints

The following are very common and useful in `CREATE TABLE` statements.

These are called constraints - they "constrain" the types of values you can insert in a column.

This reading on constraints is an excellent overview for more details.

- `PRIMARY KEY (keyname)`: Used to specify a column or group of columns uniquely identifies a row in a table.
- `AUTO_INCREMENT`: Used with an integer primary key column to automatically generate the "next" value for the key. In SQLite, only available for a primary key that's an `INTEGER`.
- `NOT NULL`: prevents NULL entries in a column, requires the value to be set in INSERT statements.
- `DEFAULT`: specifies default values for a column if not provided in an `INSERT` statement
- `UNIQUE`: requires an attribute to be unique (useful for fields that are not `PRIMARY KEY` but should still be unique)

Extracting the data

Every table should have a column which is used to uniquely identify each row. This improves efficiency and will prove very useful when using multiple tables.

```
CREATE TABLE students(
  id        INTEGER      PRIMARY KEY AUTO_INCREMENT,
  name      VARCHAR(60   NOT NULL,
  username  CHAR(32)     NOT NULL UNIQUE,
  email     VARCHAR(100) NOT NULL
);
```

Adding `PRIMARY KEY` makes it so that the code will error if that column ever has duplicates. It will use that column to identify each row quickly. This is usually an integer, but can also be other types. The primary key is conventionally named "`id`".

`AUTO_INCREMENT` will make it so that if you provide no input for that column, it will pick the highest unused (and never before used) value. Perfect for making it so you don't have to worry about what the next id is.

`CREATE TABLE`: example

```
CREATE TABLE queue (
   id              INTEGER PRIMARY KEY AUTO_INCREMENT,
   name            VARCHAR(60),
   email           VARCHAR(100),
   student_id      INT,
   length          INT,
   question        TEXT,
   creation_time   DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

The `.sql` File

Can be imported to execute SQL commands, often to create new tables (conventionally starting with setup in the file name)

```sql
-- this is a comment

CREATE TABLE queue(
   id              INTEGER PRIMARY KEY AUTOINCREMENT,
   name            TEXT,
   email           TEXT,
   student_id      INTEGER,
   length          INTEGER,
   question        TEXT,
   creation_time   DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Other table commands

`DROP TABLE table_name;` deletes a table

`DROP TABLE IF EXISTS table_name;` deletes a table only if it exists, which prevents an error if it doesn't

`CREATE TABLE IF NOT EXISTS blog_posts( ... )`

- The `IF NOT EXISTS` option is useful if you want the table to be created only if it is not already in a database when importing/executing table creation.

INSERT

To insert a new record into a table, we use the `INSERT INTO` keyword:

```
INSERT INTO queue(name, email, length, question)
  VALUES ("Maria", "m@edu.co", 2, "What's in a DB?");
```

First provide the table name, then optionally the list of columns you want to set (by default it sets all columns). Columns left out will be set to NULL, unless they have `AUTO_INCREMENT` set.

Then provide the values for each column, which must match the column names specified.

Quick questions:

- Where's the `id` column? What value should we insert for it?
- What about the `creation_time` and `student_id` columns?

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

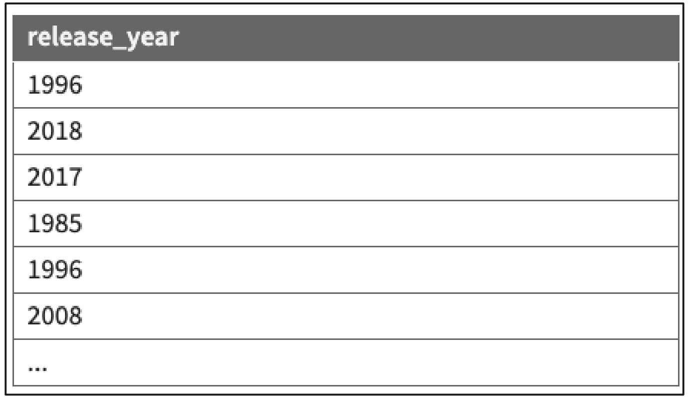
Syntax:

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

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

Example (without `DISTINCT`):

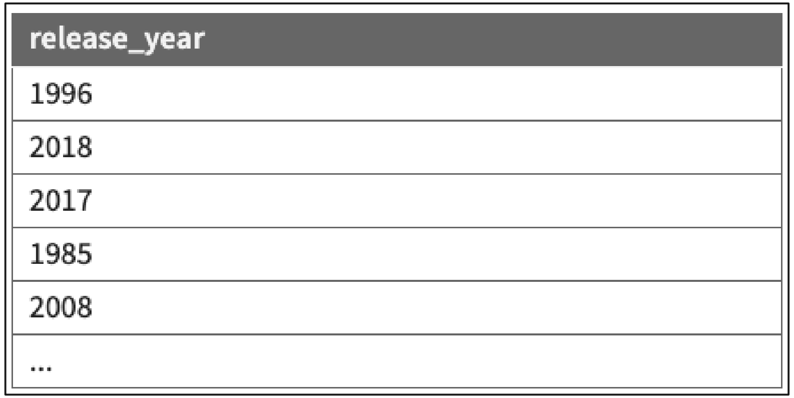
```
SELECT release_year
FROM Games;
```



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

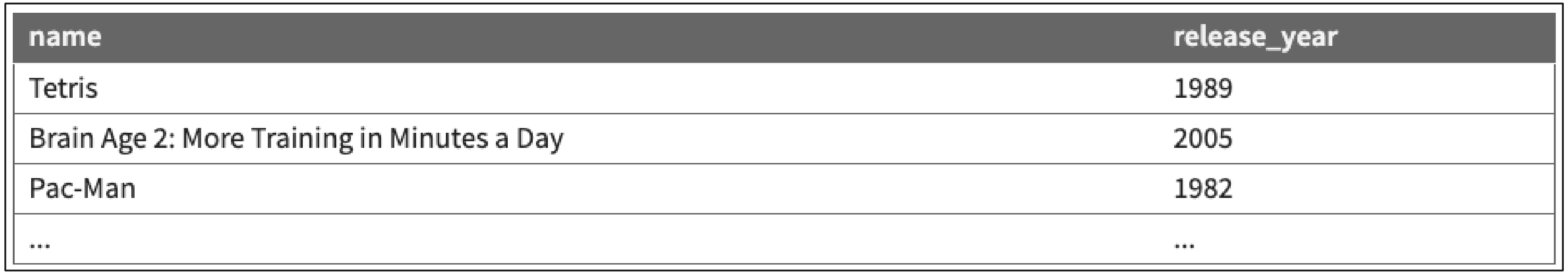
Syntax:

```
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

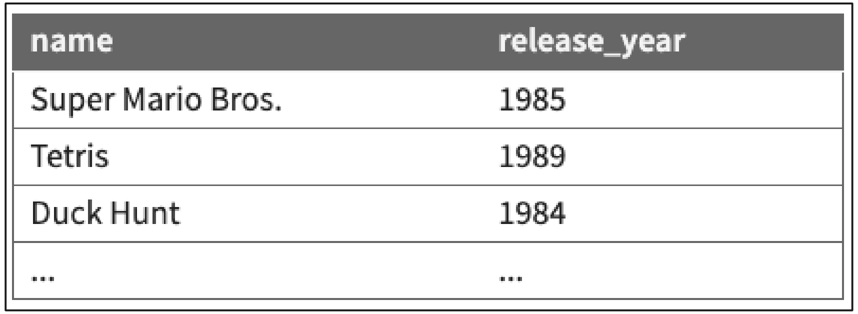
Syntax:

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

Example:

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

Example result:



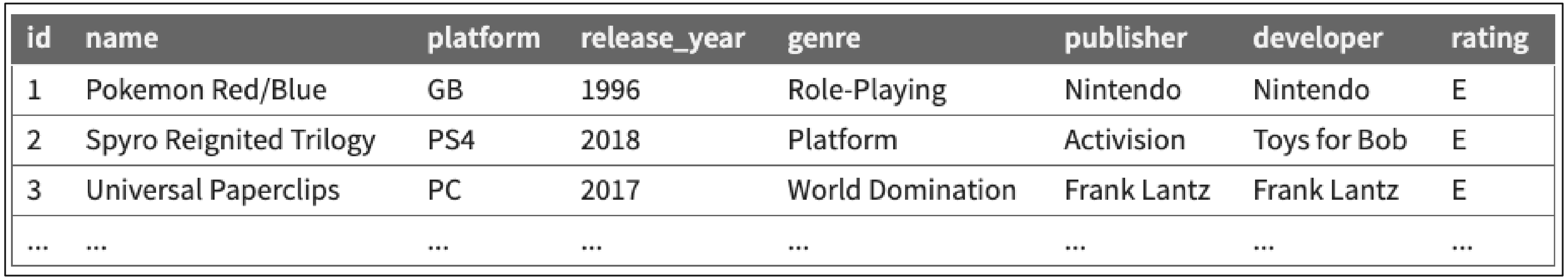
name	release_year
Super Mario Bros.	1985
Tetris	1989
Duck Hunt	1984
...	...

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

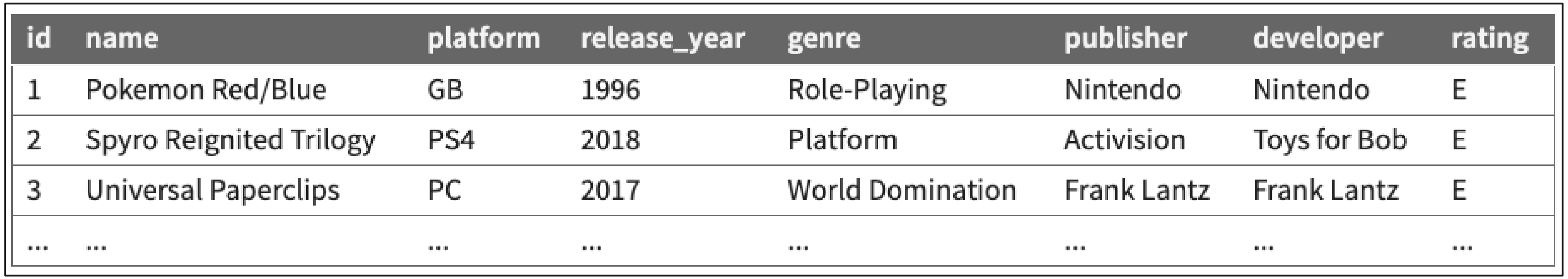
Write a SQL query that returns the `name` and `platform` of all games with a `release_year` before 2000.



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
...	...	...	...	...	...	...	...

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
...	...	...	...	...	...	...	...

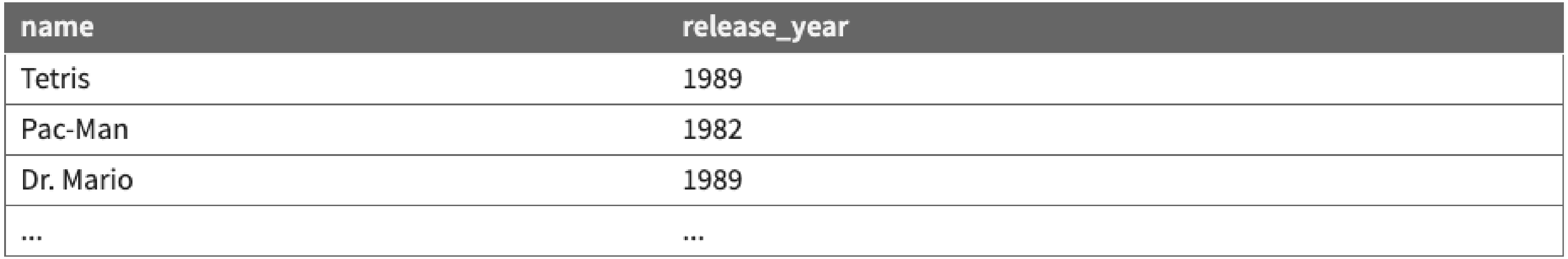
```
SELECT name, platform
FROM games
WHERE release_year < 2000;
```

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`

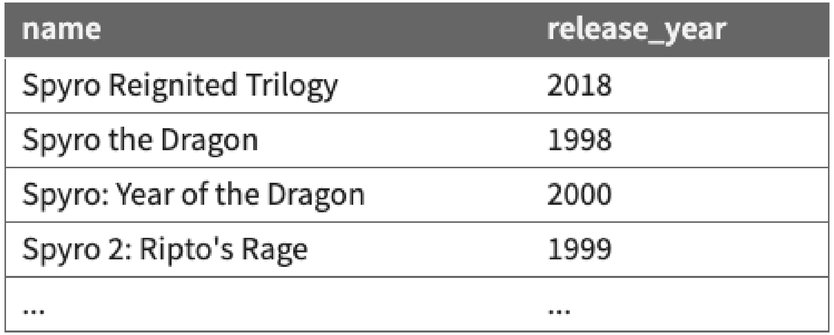
Syntax:

```
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 MySQL, 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;
```

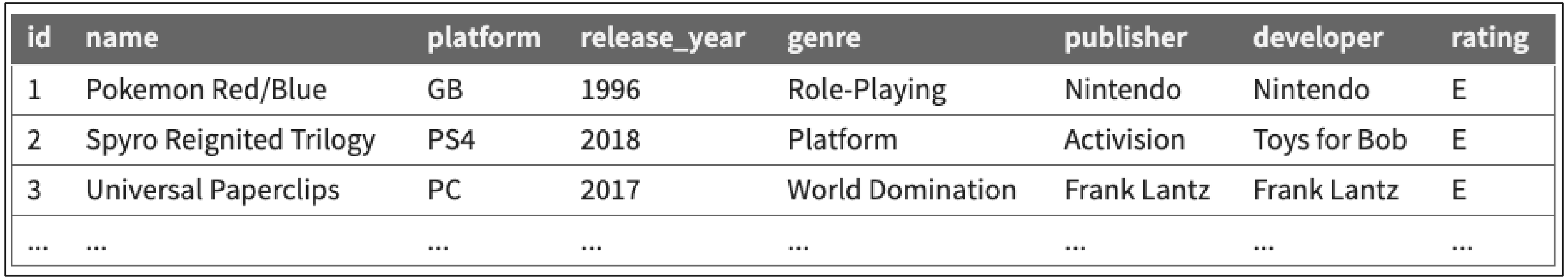
Example result:

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

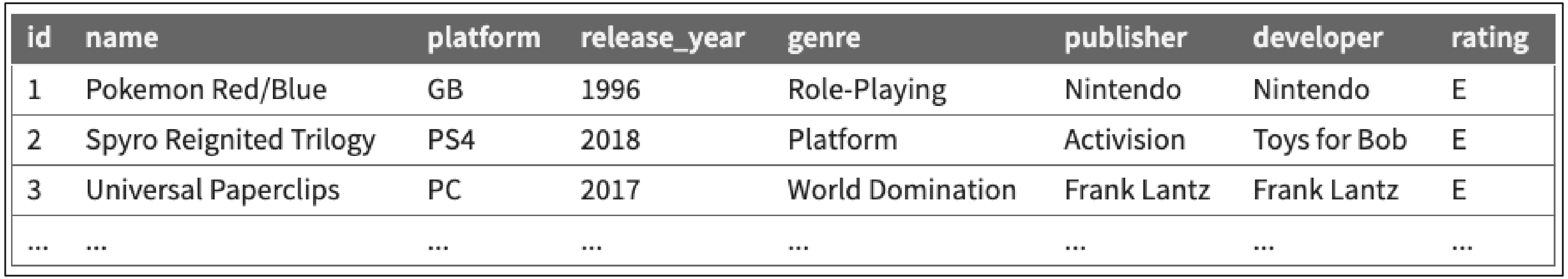
Write a SQL query that returns the `name` and `platform` of all games with the word ‘dragon’ in them, ordered by `release_year`.



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
...	...	...	...	...	...	...	...

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;
```

We can also modify the contents that exist in the table by adding, deleting, and/or updating them

DELETE

To delete a record from a table, we use the DELETE keyword:

```
DELETE
FROM cafemenu
WHERE name = 'Banana';
```

What happens if we forget to add a `WHERE` clause?

UPDATE

To update an existing record in a table, we use the `UPDATE` and `SET` keywords:

```
UPDATE cafemenu
SET description = 'yum yum, best banana ever'
WHERE name = 'Banana';
```

Additional Practice with SQL Queries

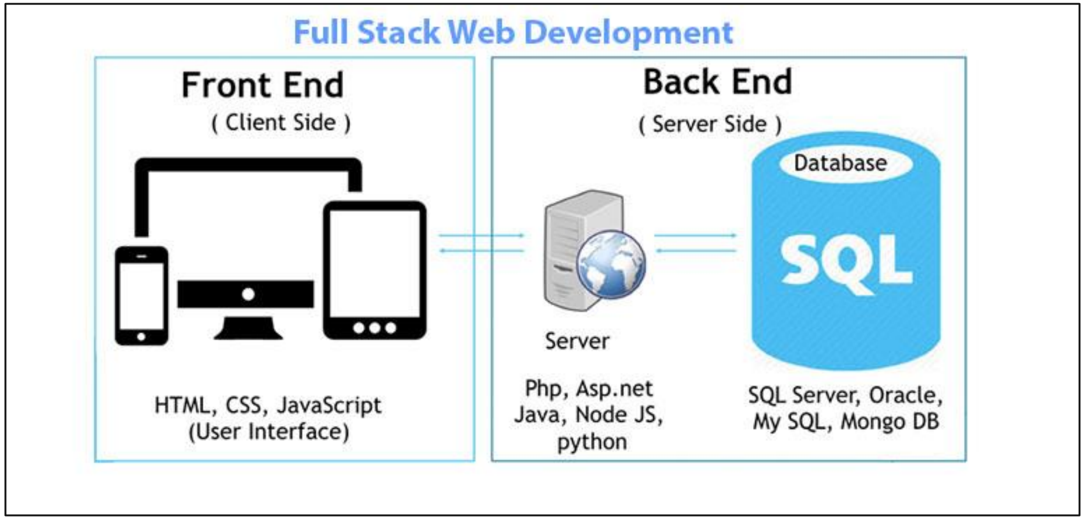
SQLZoo has multiple exercises (with built-in databases you don't need to worry about setting up) for practicing `SELECT`, `WHERE`, `ORDER BY`, `LIMIT`, `LIKE`, etc. You're recommended to go through these for additional practice!

Full-stack website architecture

A full-stack website consists of three layers:

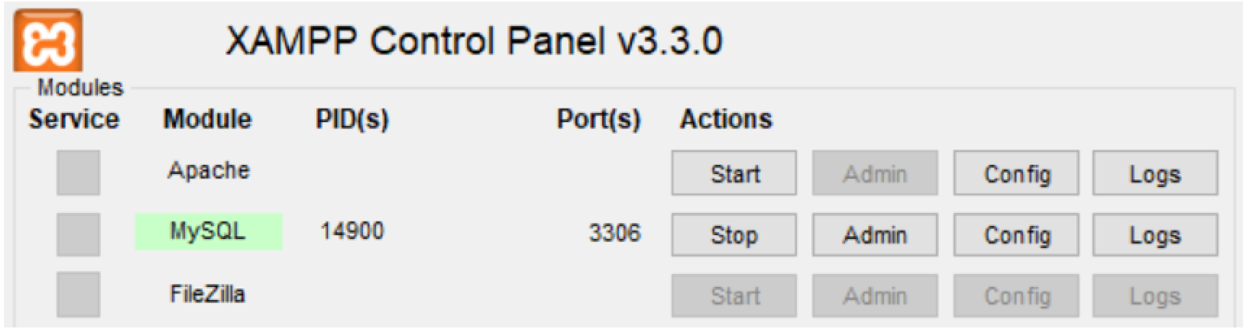
- Web Browser: HTML, CSS, JS
- Web Server: Node.js
- Database: SQL

- 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.



`mysql` vs `mysql2`

- `mysql2` has improved performance and better support for promises than `mysql` package
- `mysql2` is currently more popular than `mysql`
- You need to start the MySQL server before working with database
    - Use your XAMPP's MySQL installation



XAMPP Control Panel v3.3.0

Modules

Service	Module	PID(s)	Port(s)	Actions			
	Apache			Start	Admin	Config	Logs
	MySQL	14900	3306	Stop	Admin	Config	Logs
	FileZilla			Start	Admin	Config	Logs

Using MySQL in Node.js

First install the `mysql2` in your project:

```
npm install --save mysql2
```

Then require the `mysql2` module in your Node.js program:

```
const mysql = require('mysql2');
```

Establishing a MySQL Connection

```
const mysql = require('mysql2');
const connection = mysql.createConnection({
    host : '127.0.0.1',
    port: 3306,
    user: 'root',
    password: '',
    database: 'test'
});
```

- The default MySQL account of XAMPP is `'root'` with an empty password.

A simple SELECT query

```
let sql = "SELECT * FROM games WHERE release_year = 2018";
connection.query(sql, (err, rows) => {
    // loop through the results with a for loop
    for (let row of rows) {
        // display all game titles
        console.log(row.id + ", " + row.name;
    }
});
```

Output:

```
2, Spyro Reignited Trilogy
3002, Super Smash Bros Ultimate
3004, Super Mario Party
3005, Kirby Star Allies
```

A simple SELECT query *(asynchronously)*

The same query using asynchronous programming:

```
let sql = "SELECT * FROM games WHERE release_year = 2018";
let rs = await connection.promise().query(sql);
```

Alternatively, if you import the `mysql2/promise` module, you can omit the `.promise()` part. But you need to initialize the connection a bit differently.

```
const mysql = require("mysql2/promise");
const connection = null;
(async function () {
    connection = await mysql.createConnection({
        user: 'root', password: '', database: 'test'
    });
})();
```

```
let sql = "SELECT * FROM games WHERE release_year = 2018";
let rs = await connection.query(sql);
```

Result of SELECT query

```
let sql = "SELECT * FROM games WHERE release_year = 2018";
let rs = await connection.promise().query(sql);
let rows = rs[0];
for (let row of rows) {
    console.log(row.id + ", " + row.name);
}
```

Using placeholders in queries

The previous query can be used like below:

```
let sql = "SELECT * FROM games WHERE release_year = ?";
let rs = await connection.promise().query(sql, [2018]);
```

If the query only has a single placeholder (`?`), and the value is not `null`, `undefined`, or an array, it can be passed directly as the second argument to the `query` function:

```
let sql = "SELECT * FROM games WHERE release_year = ?";
let rs = await connection.promise().query(sql, 2018);
```

Using placeholders in queries is one of the ways to avoid SQL Injection attack.

INSERT queries

Consider this simple table:

```
CREATE TABLE `posts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(60) NOT NULL,
  `content` text NOT NULL,
  `created_at` datetime NOT NULL,
  PRIMARY KEY (`id`)
)
```

Here's the code to insert a row to this table:

```
let sql = "INSERT INTO posts (title, content, created_at) VALUES (?, ?, ?)";
let rs = await connection.promise().query(
    sql,
    ['My first post', 'This is some post contents.', new Date()]
);
console.log("A row has been inserted!");
```

Result of INSERT queries

Insert of returning an array of rows (like in SELECT queries), INSERT queries return an object of information about the executed query. For an INSERT query, we are usually interested in the generated id:

```
let sql = "INSERT INTO posts (title, content, created_at) VALUES (?, ?, ?)";
let rs = await connection.promise().query(
    sql,
    ['My first post', 'This is some post contents.', new Date()]
);
console.log("The auto id for this row is: " + rs[0].insertId);
```

UPDATE queries

For UPDATE queries, we're often interested in the number of affected rows.

```
let sql = "UPDATE posts SET title = ? WHERE id = ?";
let rs = await connection.promise().query(
    sql,
    ['Modified title', 8]
);
console.log("The number of affected rows: " + rs[0].affectedRows);
```

DELETE queries

INSERT/UPDATE/DELETE queries are similar so you should've figured out how to perform a DELETE query by now. But here's an example anyway:

```
let sql = "DELETE FROM posts WHERE id = ?";
let rs = await connection.promise().query(sql, 8);
console.log("The number of affected rows: " + rs[0].affectedRows);
```

Just avoid deleting the entire table by forgetting the WHERE clause:
(unless you actually want to erase the table)

```
let sql = "DELETE FROM posts";
let rs = await connection.promise().query(sql);
console.log("The number of affected rows: " + rs[0].affectedRows);
```

JavaScript Destructuring Assignment

JavaScript supports destructuring an array into variables:

```
let arr = ["apple", 3.0];
let [fruit, price] = arr;
```

The `query()` function returns an array of 2 objects. Many programmers destructure this array into two variables like following:

```
let sql = "SELECT * FROM games WHERE release_year = ?";
let [rows, fields] = await connection.promise().query(sql, 2018);
for (let row of rows) {
    // display game titles
    console.log(row.id + ", " + row.name);
}
```

You can read more about destructuring assignment here.

JavaScript Destructuring Assignment

Most of the times you don't need the `fields` object, so the below code will be sufficient:

```
let sql = "SELECT * FROM games WHERE release_year = ?";
let [rows] = await connection.promise().query(sql, 2018);
for (let row of rows) {
    // display game titles
    console.log(row.id + ", " + row.name);
}
```

Using `try/catch` with `mysql2` functions

You can `try/catch` just like you would for `fs.readFile()`, catching any errors that occur in any of the database querying functions.

```
let rows = [];
try {
    let sql = "'SELECT name FROM pokedex";
    [rows] = await connection.promise().query(sql);
} catch (err) {
    console.log(err); // ALWAYS delete this in production code
    res.status(500).send('Error on the server. Please try again later.');
    return;
}
for (let row of rows) {
    // do something with row
}
```

Escaping Table & Column Names

If you can't trust an SQL identifier (database / table / column name) because it is taken from user input, you should escape it with `mysql2.escapeId(identifier)`:

```
let rows = [];
let fields = [];
let tb = 'games', col = 'release_year';
let sql = "SELECT * FROM " + connection.escapeId(tb) +
    " WHERE " + connection.escapeId(col) + " = ?";
console.log(sql);
```

Output:

```
SELECT * FROM `games` WHERE `release_year` = ?
```

Placeholders for Table & Column Names

Alternatively, you can use `??` characters as placeholders for identifiers you would like to have escaped:

```
let sql = "SELECT * FROM ?? WHERE ?? = ?";
let [rows] = await connection.promise().query(
    sql,
    ['games', 'release_year', 2017]
);
for (let row of rows) {
    console.log(row.id + ", " + row.name);
}
```

Output:

```
3, Universal Paperclips
3000, Super Mario Odyssey
3001, Mario Kart 8 Deluxe
3006, Fire Emblem Warriors
```

mysql2 Type Casting

MySQL and JavaScript have different data types. So values from MySQL must be casted into JS types. The following is a mapping of such type casting:

- **Number**: TINYINT, SMALLINT,INT, MEDIUMINT, YEAR, FLOAT, DOUBLE
- **Date**: TIMESTAMP, DATE, DATETIME
- **String**: CHAR, VARCHAR, TINYTEXT, MEDIUMTEXT, LONGTEXT, TEXT, ENUM, SET, DECIMAL, BIGINT, TIME
- **Buffer**: TINYBLOB, MEDIUMBLOB, LONGBLOB, BLOB, BINARY, VARBINARY, BIT

(*) DECIMAL is cast into String because casting DECIMAL to JavaScript's Number might lead to loss of number precision.

Gracefully close the Node.js server

Pressing Ctrl + C will kill the Node.js process, but does not properly close the DB connection and the web server. Add the following to your code to make sure your web server shuts down properly.

```
process.on('SIGTERM', () => {
    console.log('SIGTERM signal received.');
    console.log('Closing HTTP server and stop receiving requests...');
    app.close();
    console.log('Closing DB connection...');
    connection.end(function (err) {
        // the connection is terminated now
        console.log('DB connection has been closed.');
        // exit program
        console.log("Goodbye!");
        process.exit(0);
    });
});
```

Example

A POST endpoint logging a user in:

```
app.use(express.urlencoded({ extended: true }));
app.post('/login', async (req, res) => {
    let user = req.body.username;
    let pw = req.body.password;
    let qry = "SELECT * FROM users WHERE username = ? AND password = ?";
    let [rows] = await connection.promise().query(qry, [user, pw]);
    if (rows.length === 0) {
        res.type("text");
        res.status(400).send("user or password not found");
    } else {
        // send the user info data (a single 1st row) as JSON
        res.json(rows[0]);
    }
});
```

Storing configurations in `.env`

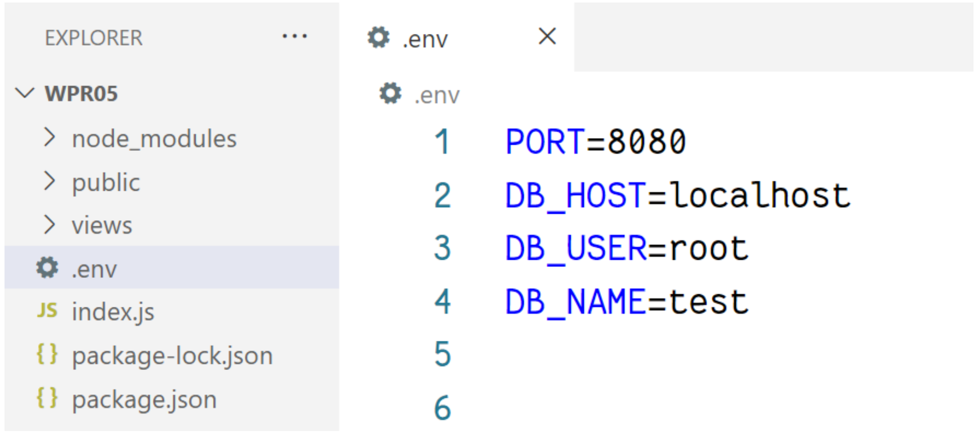
- An app’s config includes values that may vary between deployments.
- Examples:
  - Database connection parameters
  - External services credentials: API keys, SMTP accounts...
  - Per-deploy values: website's hostname, port number...
- In Node.js, app's config is usually stored in the `.env` file in the project root.
- Config can be loaded into the app using the `dotenv` module.
- Loaded config values are available in the `process.env` object.

`dotenv` usage instructions

- Install `dotenv` package:

```
npm install dotenv --save
```

- Create a file named `.env` with some values in it:



```
PORT=8080
DB_HOST=localhost
DB_USER=root
DB_NAME=test
```

`dotenv` usage instructions

- Import the package and execute the `config()` function:

```
require('dotenv').config();
```

- After that, environment variables are available in `process.env` object.
  - Including all variables defined in `.env`
  - And all of the OS's environment variables (such `Path`, `SystemRoot`, `JAVA_HOME`, `USERNAME`... on Windows)
- You can use these variables in your Node application. For example:

```
const conn = mysql.createConnection({
    host: process.env.DB_HOST,
    user: process.env.DB_USER,
    database: process.env.DB_NAME
}).promise();

app.listen(process.env.PORT);
```

SQL Setup files

- `.sql` files are often used to setup the database (create tables) and initialize website's data
- A `.sql` file contains SQL statements, separated by semi-colons (`;`)
- The file is meant to be executed by a script
    - The script needs to split the file into SQL statements
    - An SQL statement can be single-line or multi-line
    - Spliting the file by semi-colons is not a solution since there may be semi-colons inside an SQL statement

Example: Executing a `.sql` setup file

```
const mysql = require('mysql2');
const conn = mysql.createConnection({
    user: 'root',
    database: 'test'
}).promise();
const fs = require('fs').promises;

(async function () {
    let content = await fs.readFile('./setup.sql', { encoding: 'utf8' });
    let lines = content.split('\r\n'); // Windows line separator
    let tmp = '';
    for (let line of lines) {
        line = line.trim();
        tmp += line + '\r\n';
        if (line.endsWith(';')) { // statement detected
            await conn.execute(tmp);
            tmp = '';
        }
    }
    await conn.end();
})();
```