An Introduction to SQL

SQL (Structured Query Language) is a programming language for working with relational data. Relational data is stored within Relational Database Management Systems (RDBMS) such as MySQL, Oracle, sqlite DB2 and more. Each of these RDBMS have their own unique spin on SQL syntax, so in your progression learning the language, you will *always* need to keep an eye out for minor syntax changes depending on which RDBMS you are using.

Assuming you are working on a JavaScript based tech stack, you will probably not be doing a lot of writing raw SQL and/or interacting directly with some RDBMS. At the least you will be using already existing JavaScript libraries that allow you to write SQL statements inside your JavaScript, and most often you will use an ORM technology like Sequelize, which will provide you with a syntax for interacting with an RDBMS that looks and feels much more like JavaScript than SQL.

All that said, it is important you have some small amount of exposure and experience with raw SQL, and this short series of exercises will expose you to some of the basics of writing raw SQL.

High Level Goals of this minisprint

Feedback

- Gain exposure to the basics of raw SQL syntax
- Gain a small amount of experience reading and writing raw SQL
- Gain a small amount of experience thinking about relational data, and wrestling with getting the kind of data you want out of a relational schema

SQLite

In this minisprint you will be using the SQLite RDBMS. SQLite is serverless, and while you don't need to understand all the implications of this, you should know that it will allow you to interact directly with the RDBMS with very little configuration, making it a good choice for our purpose of exposing you to basic SQL.

Please remember that each RDBMS has its own specific implementation of SQL syntax and therefore that you may need to slightly modify some of what you learn here when working with another RDBMS like MySQL.

Setup

Do `which sqlite3`. You should see the path to the `sqlite3` binary. If you do not, do `brew install sqlite3` to install it.

The SQLite database is just a file. For these exercises you will use the `school.db` file in the root directory of this repo. From inside the root directory of this repo do `sqlite3 school.db`. This will start up an interactive SQLite terminal, with the already populated tables from `school.db` loaded into the session.

From inside the `sqlite` terminal you just opened, do `.table` to display the tables in the database. You should see several table names (more on *tables* later) including `classes`, `teachers` and more.

Exit the terminal with `CTRL + d`.

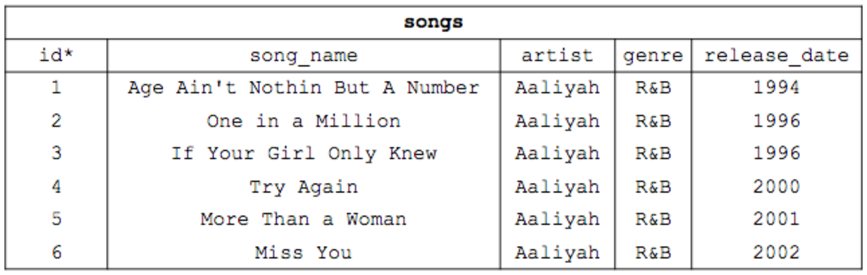
If you should happen to modify the database in a way that becomes problematic for you, a pristine copy of it exists in the `dbs/` directory of this repo. If needed, simply make a new copy of `dbs/school.db` and load it into `sqlite3`.

Feedback

Schema

Relational data lives inside *tables* that contain *columns* and *rows*. The columns of a table define the properties or attributes of the kind of 'thing' the table is storing, and each row is an instance of this kind of 'thing'.

For example, a given table might be meant to store data on songs. The columns of the table are `song_name`, `artist`, `genre` and `release_date`. Each row represents a given song and contains an entry for each of the columns, like so:



songs				
id*	song_name	artist	genre	release_date
1	Age Ain't Nothin But A Number	Aaliyah	R&B	1994
2	One in a Million	Aaliyah	R&B	1996
3	If Your Girl Only Knew	Aaliyah	R&B	1996
4	Try Again	Aaliyah	R&B	2000
5	More Than a Woman	Aaliyah	R&B	2001
6	Miss You	Aaliyah	R&B	2002

Tables are defined with a *schema*. A schema defines the name of the table, and also, the details about what kind of data should be stored at each column for that table. The syntax for creating a new table schema is:

```
CREATE TABLE <table-name> (
  <name-of-column-1> <data-type-of-column> [ADDITIONAL-INFO-ABOUT-THIS-COL
  <name-of-column-2> <data-type-of-column> [ADDITIONAL-INFO-ABOUT-THIS-COL
                              ...
  <name-of-column-n> <data-type-of-column> [ADDITIONAL-INFO-ABOUT-THIS-COL
  [OTHER-SCHEMA-DEFINITION-COMMANDS]
);
```

Most SQL implementations ignore insignificant whitespace and also are case insensitive. With that in mind you shouldn't be surprised to see the above in a form that looks more like `create table <table-name> (<name-of-column1> <data-type-of-column> [additional-info-about-this-column]);`. Notice the `;` at the end of these statements. The `;` indicates that the statement is complete and is part of the reason why an SQL author can choose to write their SQL on one or many lines.

Your turn

Feedback

In these *Your turn* sections you'll be given questions and exercises to answer on your own. Interact with the database until you are able to answer each of them. For more complicated technical questions, you'll be given a target answer.

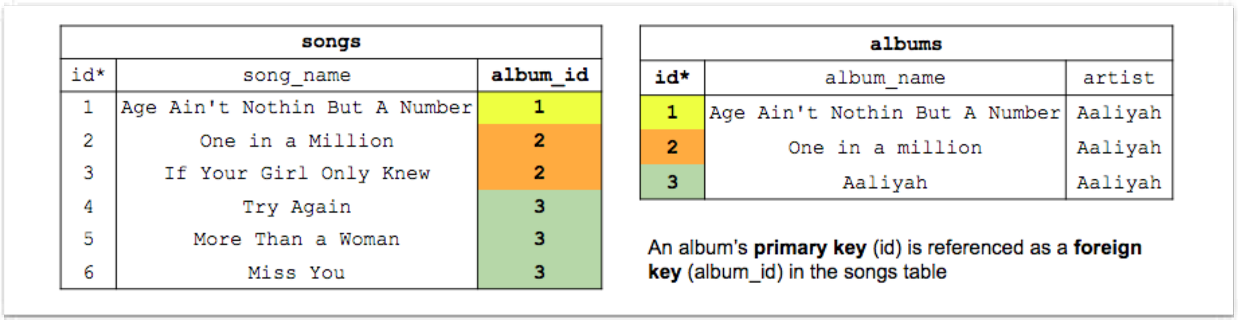
In SQLite, you can view the schema of a table with the `.schema <table-name>` command. For example, to see the schema for a table called `jobs` you would issue the `.schema jobs` command.

- [ ] Look at the `departments` table in the `school` database provided to you. How many columns does it have and what are the column names?

By including `PRIMARY KEY` in the definition of a column, you tell the RDBMS that for each row, the value of this column should be uniquely identifying. RDBMSs will enforce the uniqueness of primary keys. If you should try to modify the database in a way that violates the uniqueness of a primary key, you will get an error.

- [ ] Which column in the `departments` table is intended to be uniquely identifying?

Primary keys from one table are often referred to in other tables. When the value of a column is intended to refer to a column in another table, it is called a `FOREIGN KEY`.



songs		
id*	song_name	album_id
1	Age Ain't Nothin But A Number	1
2	One in a Million	2
3	If Your Girl Only Knew	2
4	Try Again	3
5	More Than a Woman	3
6	Miss You	3

albums		
id*	album_name	artist
1	Age Ain't Nothin But A Number	Aaliyah
2	One in a million	Aaliyah
3	Aaliyah	Aaliyah

An album's **primary key** (id) is referenced as a **foreign key** (album_id) in the songs table

The technique of using foreign keys to point to data that lives elsewhere is one of the most quintessential aspects of relational data. By referring to data in columns of other tables, rather than storing the data in two different tables, you can store the data in a more memory efficient way since you do not need to store the same data more than once. Making changes to your data is also simplified on account of only ever having to change it in one place.

- [ ] Look at the `teachers` schema. Which column is being used as a foreign key? Why might we be using a foreign key here rather than storing the data directly in this table? Review the last paragraph if you are unclear.

- [ ] Look at the remaining table schemas in this database. Whiteboard each of the 5 tables, representing them as simple spreadsheet like grids, using arrows to indicate where a particular column is referring to data stored in another t

Feedback

Querying the database with `SELECT`

By far the most common SQL command is `SELECT`, which is used to query the database using the following syntax `SELECT <column-name> [, <additional-column-names>] FROM <table-name>;` If you wish to select every column in a table you can use the `*` character in place of a column name to indicate 'every column'. `SELECT` will return the columns requested for every row in the table.

Your turn

SQLite allows you to set a 'mode' for viewing the results of queries, and also provides you the option for whether or not you would like to view column headers in query results. For pleasant viewing, set the mode to column mode with the `.mode column` command and turn headers on with `.header on`.

- [ ] Display the `name` column for every row in the `students` table
- [ ] Display every column for the `teachers` table. The `department` column just contains numbers, what are these numbers referencing? (Look at the `teachers` schema if you need to).
- [ ] Display every column for the `teachers` table and then every column for the `departments` table. Just by looking at the tables, what is the name of the department that the teacher `beth` is a part of?

Filtering rows with `WHERE`

By default `SELECT` will return every row in a table, however, you will frequently wish to trim down which rows match your query. One way to do this is with `WHERE`, using the following syntax:

```
SELECT <column-name> FROM <table-name>
  WHERE <some-condition-to-limit-by>;
```

A simple `condition-to-limit-by` is a value for a given column. You can evaluate this value using `=`, `!=`, `<`, `>`, `<=`, and `>=`, amongst others. Chain togeher your conditions using `AND` and `OR`, which behave logically as you are used to from your work with

Feedback

JavaScript. For example, if I wanted to display all the columns from the teachers table where the department id is 1 or 2:

```sql
SELECT * FROM teachers
  WHERE department = 1 OR department = 2;
```

Your turn

- [ ] Display just the name column for all the students whose names are not naomi. (Note, `naomi` being text, should be placed in single quotes)
- [ ] Display the name and department id of teachers whose own id is greater than 2 or whose name is 'fred'

`WHERE` pattern matching with `LIKE`

Aside from the comparison operators above, you can also use the `LIKE` keyword, in conjunction with the `%` wildcard symbol to select rows based on patterns you write. `%` will match 0 or more of any character. Using `NOT LIKE` instead of `LIKE` will select rows that do not match the pattern.

For example, to select all the class names that start with the letter 'c':

```sql
SELECT name FROM CLASSES
  WHERE name LIKE 'c%';
```

Your turn

- [ ] Display the id and name of all the students whose names end in 'm'
- [ ] Display all columns for students whose names do not contain the letter 'a'. HINT: a more long-winded way to say "includes the letter 'a'" is "includes 0 or more of any letter followed by an 'a' followed by 0 or more of any letter."

Limiting `WHERE` to a defined set with `IN`

Using `IN` you can filter your query against a set of matches you define, for example, to get the entries for the teachers whose names are either 'pamela' or 'sunny':

Feedback

```
SELECT * FROM teachers
  WHERE name IN ('pamela', 'sunny');
```

Perhaps you're thinking you could have done this so:

```
SELECT * FROM teachers
  WHERE name = 'pamela' OR name = 'sunny';
```

This also is correct and discloses the truth about SQL (which is true in most all programming languages), when your work begins to get more complicated there often is not a single correct way to proceed.

Your turn

- [ ] Display just the names of all the teachers whose id is `NOT` either 1, 2, or 4
- [ ] Display just the names of all the teachers whose department is either 1 or 4

Using `IN` to compose subqueries

Set it up so that you can view all the teacher and department entries while reading this section.

In the last exercise you were able to print out teachers who had particular department ids. The `department` field in the `teachers` table, if you recall (look at the `teacher` schema if you do not), is a foreign key, referencing the `id` column (which is a uniquely identifying primary key) in the `departments` table. You might imagine wanting to answer a question like "who are all the teachers in the 'cs' department", instead of "all the teachers in the department with an id of 1". In order to do this, you would first need to find out what the `id` field is for the department with the `name` 'cs', and then, with that id, ask for any of the teachers whose `department` foreign key equals the id you just retrieved.

Thus we could first:

```
SELECT id FROM departments
  WHERE name = 'cs';
```

And then with our result, 1, issue a second query:

Feedback

```sql
SELECT name FROM teachers
  WHERE department = 1;
```

The following is not legitimate SQL but represents what we would like to be able to do, namely, filter our second query by the results of the first:

```sql
SELECT name FROM teachers
  WHERE department = (SELECT id FROM departments
                       WHERE name = 'cs');
```

The above can be only slightly modified to be valid SQL. Review the syntax for `IN` above, you'll notice the use of `(` and `)`, similar to our imaginary query above. In order to make this nested selection, simply use the `IN` command passing in a separate query as the set of possibilities to filter by. Thus:

```sql
SELECT name FROM teachers
  WHERE department IN (SELECT id FROM departments
                       WHERE name = 'cs');
```

Your turn

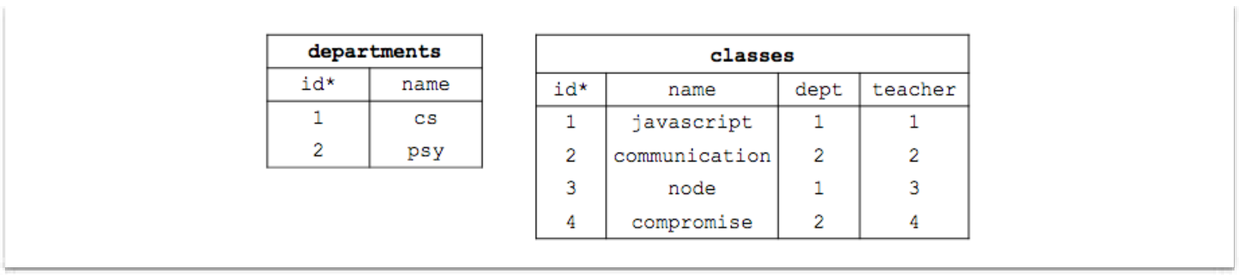
- [ ] Display the name and id of all the teachers in the 'psy' department (should be pamela and sunny, with their respective ids)
- [ ] Display the name of the department that 'sunny' teaches for (should be 'psy')

Selecting from multiple tables

You can select from multiple tables in SQL with the following syntax : `SELECT * FROM <table-1>, <table-2> [,<table-n>];`.

To help gain an intuition about what will happen during a multi-table select statement, display all the rows from `departments` and then, in a seperate query, all the rows from `classes`.

Feedback



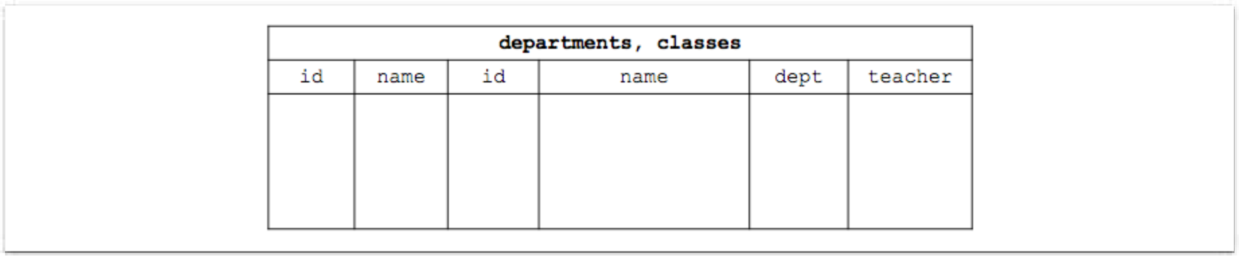
departments

id*	name
1	cs
2	psy

classes

id*	name	dept	teacher
1	javascript	1	1
2	communication	2	2
3	node	1	3
4	compromise	2	4

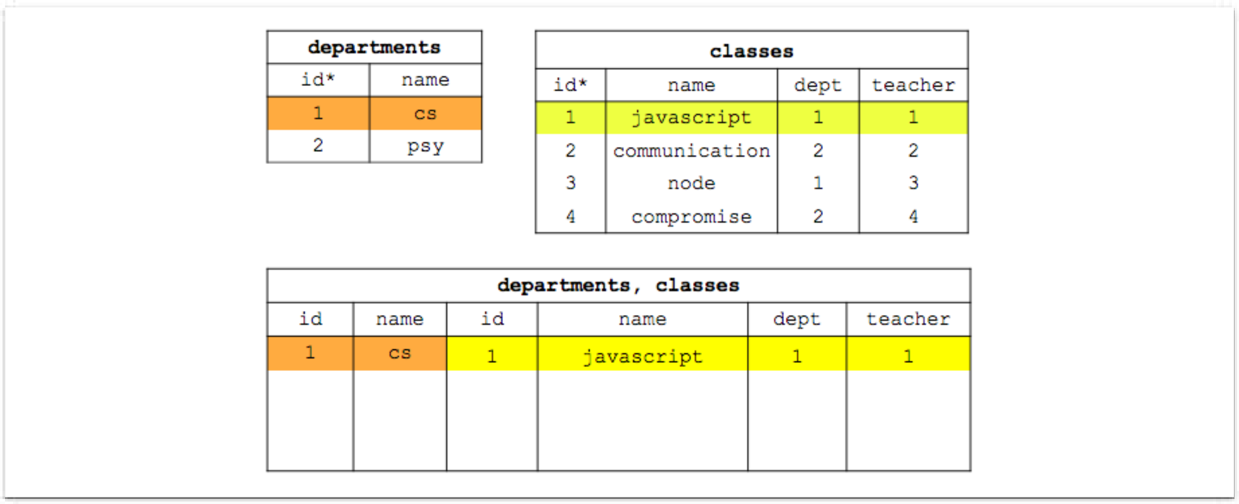
Imagine creating a new table with one column for every column in the `departments` table (there are 2) and one column for every column in the `classes` table (there are 4). This would result in a 6 column table.



departments, classes

id	name	id	name	dept	teacher

Now, in order to populate this new 6 column table, imagine taking the first row from the `departments` table and combining it with the first row in the `classes` table.



departments

id*	name
1	cs
2	psy

classes

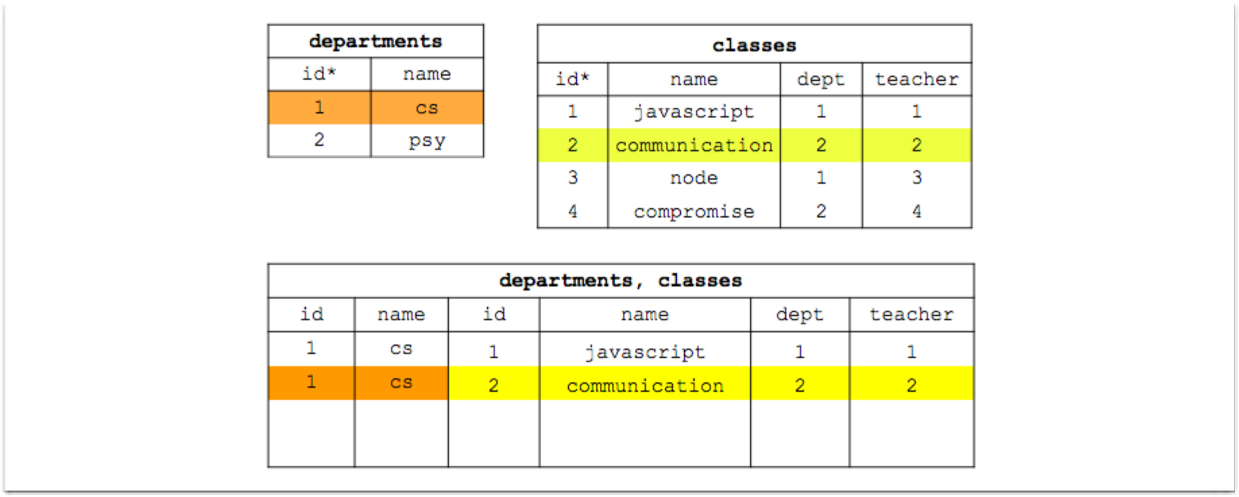
id*	name	dept	teacher
1	javascript	1	1
2	communication	2	2
3	node	1	3
4	compromise	2	4

departments, classes

id	name	id	name	dept	teacher
1	cs	1	javascript	1	1

Then take the first row of the `departments` table and combine it with the *second* row of the `classes` table.

Feedback

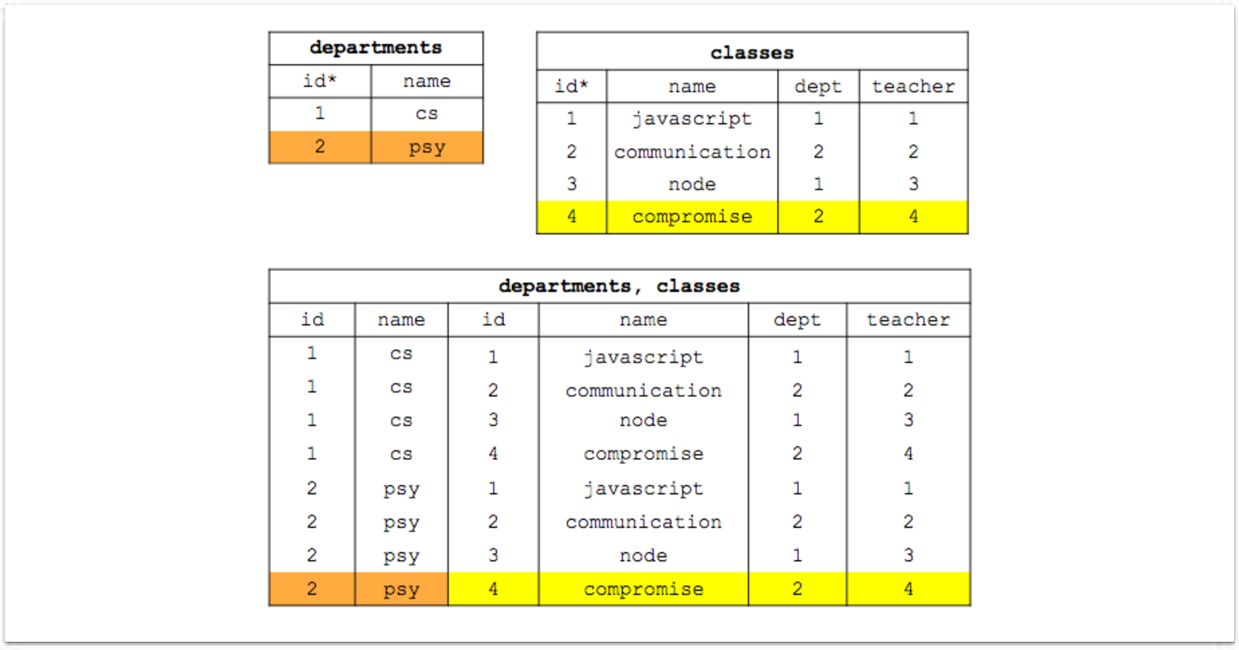


departments	
id*	name
1	cs
2	psy

classes			
id*	name	dept	teacher
1	javascript	1	1
2	communication	2	2
3	node	1	3
4	compromise	2	4

departments, classes					
id	name	id	name	dept	teacher
1	cs	1	javascript	1	1
1	cs	2	communication	2	2

Imagine continuing in this way until you have combined each possible row in `departments` with each possible row in `classes`.



departments	
id*	name
1	cs
2	psy

classes			
id*	name	dept	teacher
1	javascript	1	1
2	communication	2	2
3	node	1	3
4	compromise	2	4

departments, classes					
id	name	id	name	dept	teacher
1	cs	1	javascript	1	1
1	cs	2	communication	2	2
1	cs	3	node	1	3
1	cs	4	compromise	2	4
2	psy	1	javascript	1	1
2	psy	2	communication	2	2
2	psy	3	node	1	3
2	psy	4	compromise	2	4

Because there are 2 rows in `departments` and 4 rows in `classes` this would be a grand total of 8 (2 * 4) rows, each with 6 columns.

If you run `SELECT * FROM departments, classes;` you will see just this occur, please do so now.

Your turn

Of course you don't have to select all columns (`*`). However, because you will be selecting from multiple tables, for each column you wish to select, you need to

Feedback

preface the column name with the table name. For example, `SELECT departments.id, classes.id FROM departments, classes;`.

- [ ] Use the same thought process from above to make a prediction about what the following queries will return. How many columns will there be? How many rows will there be? Run each of the queries to check your work.
    - `SELECT departments.id, classes.id FROM departments, classes;`
    - `SELECT students.*, teachers.name FROM students, teachers;`

Advanced filtering with multi-table `SELECT`

Recall above we used `IN` in conjunction with a sub query to answer the question "Who are all the teachers in the 'cs' department?" Let's use multi-table selection to answer the question in a different manner.

Take a look at the results from the following query. If you're at all suprised by what you see, please revisit the previous section:

```
SELECT * FROM teachers, departments;
```

Since we are only interested in the "cs" department, let's trim this down a bit by only showing columns where the department name is "cs":

```
SELECT * FROM teachers, departments
  WHERE departments.name = "cs";
```

Given that we know the correct answer to our query is "fred" and "beth", can you identify what the rows containing "fred" and "beth" have in common with each other that would help us make a query that only gave us these 2 rows?

You perhaps noticed that for the rows containing "fred" and "beth", the `teachers.department` column entry (which is a foreign key referencing a `department.id`) matches the `departments.id` column entry. Reflect on the meaning of a row where a foreign key matches the primary key that it references.

Therefore:

Feedback

```sql
SELECT * FROM teachers, departments
  WHERE departments.name = "cs" AND teachers.department = departments.id;
```

And to only return the `name` of the teachers, as was our original intent:

```sql
SELECT teachers.name FROM teachers, departments
  WHERE departments.name = "cs" AND teachers.department = departments.id;
```

Your turn

You already completed the following with subqueries, now do them without subqueries, using multiple table selection instead.

- [ ] Display the name and id of all the teachers in the 'psy' department (should be pamela and sunny, with their respective ids)
- [ ] Display the name of the department that 'sunny' teaches for (should be 'psy')

`INNER JOIN`

Selecting accross multiple tables and then filtering based on the rows that have shared entries is known as an *inner join*. The following query, which is already familiar to you, is an inner join:

```sql
SELECT teachers.name FROM teachers, departments
  WHERE teachers.department = departments.id;
```

SQL provides a special syntax for inner joins. Using this syntax the above could be refactored as:

```sql
SELECT teachers.name FROM teachers INNER JOIN departments
  ON teachers.department = departments.id;
```

Your turn

- [ ] What is the difference between the return from the following two statements:
    - `SELECT * FROM students, teachers;`

Feedback

    - `SELECT * FROM students INNER JOIN teachers;`

You already completed the following with subqueries, and then with multiple table selection, now do them using inner join syntax instead.

- [ ] Display the name and id of all the teachers in the 'psy' department (should be pamela and sunny, with their respective ids)
- [ ] Display the name of the department that 'sunny' teaches for (should be 'psy')

Other joins

Inner joins are by *far* the most prevalent kind of joining that you will do. There are however, other kinds of joins.

The next most common kind of join is the *left outer join* (there are also *right outer joins*). The left outer join will return all the same information as an inner join, but will also return all entries from the first table in the query, even the ones that don't meet the `ON` condition.

Run the following familiar query:

```
SELECT teachers.name FROM teachers INNER JOIN departments
  ON departments.name = "cs" AND teachers.department = departments.id;
```

Notice the difference when refactoring to use a `LEFT OUTER` join instead:

```
SELECT teachers.name FROM teachers LEFT OUTER JOIN departments
  ON departments.name = "cs" AND teachers.department = departments.id;
```

Some RDBMS support other kinds of outer joins, but not SQLite. To do a *right* `OUTER JOIN` in SQLite, you just need to change the order of the tables in the query so that the one you want to be on the 'LEFT' is on the left:

```
SELECT teachers.name FROM departments LEFT OUTER JOIN teachers
  ON departments.name = "cs" AND teachers.department = departments.id;
```

Your turn

Feedback

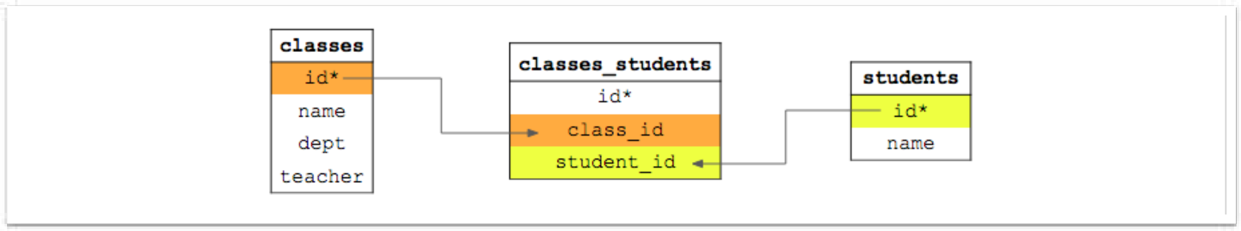
- [ ] Spend about 5 minutes reading this awesome Stack Overflow post on the differences between inner and outer joins. Keep an eye out for novel syntax and also the way Mark Harrison's response demonstrates multiple ways to make the same query.

Join Tables

You've been exploring tables that have entries which relate to each other. For example, each single entry in the `classes` table relates to a single entry in the `teachers` table. This kind of relationship is called a *one to one* relationship. Each single entry in the `departments` table maps to multiple entries in the `classes` table. This is called a *one to many* relationship.

Consider the relationship of `students` to `classes`. We would expect there to be many students in each class and also many classes for each student. This kind of relationship is called *many to many*. There is not a very direct way to handle many to many relationships between tables. Instead, when we want to support a many to many relationship between two tables, we create a third table, often referred to as a *join table* (or sometime *junction table*) that contains foreign keys referencing each of the other 2 tables.

The database you've been working inside of contains a table `classes_students` that serves as a join table between the `students` and `classes` tables:



In this way, each of the "original" tables can have its own one to one or one to many relationship with the join table. By combining all the techniques you've learned with an additional join table, you will be able to query data to expose many to many relationships.

Your turn

Take a look at the `classes_students` schema and answer the following:

- [ ] Which classes is 'sam' taking? (confirm your answer below)

Feedback

- [ ] What are the names of the students in the 'compromise' class?
- [ ] What are the names of the students taking any class in the 'cs' department?

Answers from prompts above

- 'sam' is taking the 'commumication' class
- The students in the 'compromise' class are 'naomi', 'chris', and 'kim'
- The names of the students taking any class in the 'cs' department are 'lauren', 'dan', 'naomi', 'kim' and 'chris'