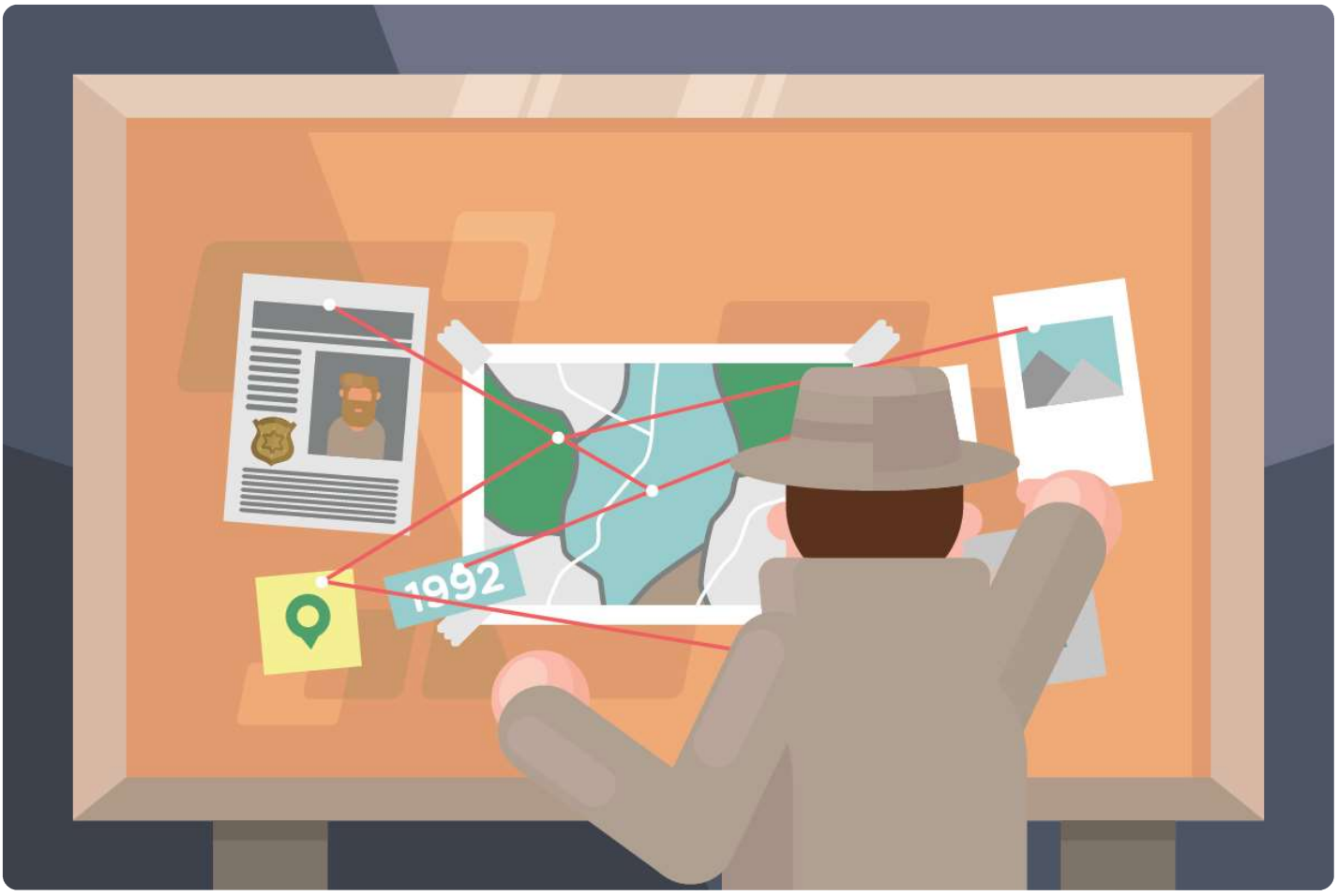
SQL Murder Mystery

Can you find out whodunnit?



There's been a Murder in SQL City! The SQL Murder Mystery is designed to be both a self-directed lesson to learn SQL concepts and commands and a fun game for experienced SQL users to solve an intriguing crime.

Walkthrough for SQL Beginners

If you're comfortable with SQL, you can skip these explanations and put your skills to the test! Below we introduce some basic SQL concepts, and just enough detail to solve the murder. If you'd like a more complete introduction to SQL, try Select Star SQL.

A crime has taken place and the detective needs your help. The detective gave you the crime scene report, but you somehow lost it. You vaguely remember that the crime was a **murder** that occurred sometime on **Jan.15, 2018** and that it took place in **SQL City**. Start by retrieving the corresponding crime scene report from the police department's database.

All the clues to this mystery are buried in a huge database, and you need to use SQL to navigate through this vast network of information. Your first step to solving the mystery is to retrieve the corresponding crime scene report from the police department's database. Below we'll explain from a high level the commands you need to know; whenever you are ready, you can start adapting the examples to create your own SQL commands in search of clues -- you can run any SQL in any of the code boxes, no matter what was in the box when you started.

Some Definitions

What is SQL?

SQL, which stands for Structured Query Language, is a way to interact with relational databases and tables in a way that allows us humans to glean specific, meaningful information.

Wait, what is a relational database?

There's no single definition for the word *database*. In general, databases are systems for managing information. Databases can have varying amounts of structure imposed on the data. When the data is more structured, it can help people and computers work with the data more efficiently.

Relational databases are probably the best known kind of database. At their heart, relational databases are made up of *tables*, which are a lot like spreadsheets. Each column in the table has a name and a data type (text, number, etc.), and each row in the table is a specific instance of whatever the table is "about." The "relational" part comes with specific rules about how to connect data between different tables.

What is an ERD?

ERD, which stands for Entity Relationship Diagram, is a visual representation of the relationships among all relevant tables within a database. You can find the ERD for our SQL Murder Mystery database below. The diagram shows that each table has a name (top of the box, in bold), a list of column names (on the left) and their corresponding data types (on the right, in all caps). There are also some gold key icons, blue arrow icons and gray arrows on the ERD. A gold key

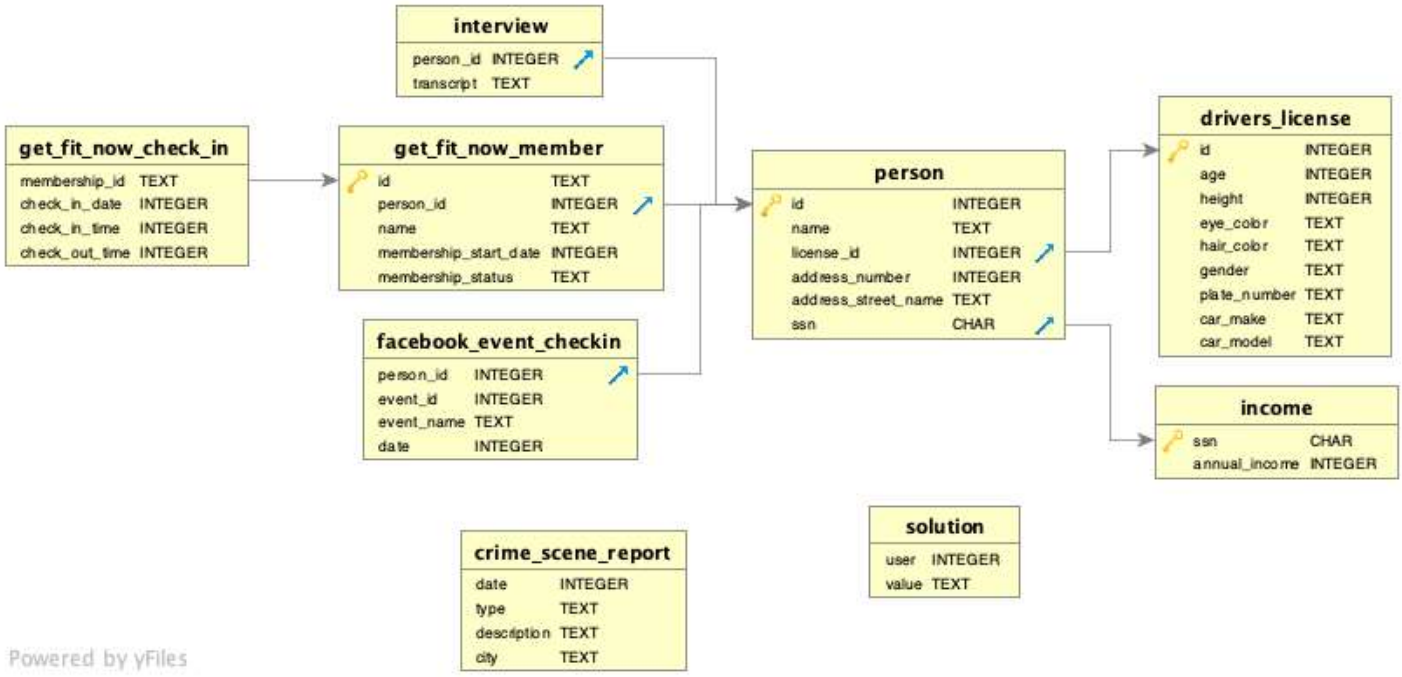
indicates that the column is the primary key of the corresponding table, and a blue arrow indicates that the column is the foreign key of the corresponding table.

Primary Key:
a unique identifier for each row in a table.

Foreign Key:
used to reference data in one table to those in another table.

If two tables are related, the matching columns, i.e. the common identifiers of the two tables, are connected by a gray arrow in the diagram.

Here is the ERD for our database:



What is a query?

If you were to look at the data in this database, you would see that the tables are huge! There are so many data points; it simply isn't possible to go through the tables row by row to find the information

we need. What are we supposed to do?

This is where queries come in. Queries are statements we construct to get data from the database. Queries read like natural English (for the most part). Let's try a few queries against our database. For each of the boxes below, click the "run" to "execute" the query in the box. You can edit the queries right here on the page to explore further. (Note that SQL commands are not case-sensitive, but it's conventional to capitalize them for readability. You can also use new lines and white space as you like to format the command for readability. Most database systems require you to end a query with a semicolon (';') although the system for running them in this web page is more forgiving.)

Just how many people are in this database?

Don't worry about exactly what this means, but know that you can change 'person' to any other table from the ERD to learn how many rows that table has. Try it!

```
1 SELECT count(*)
2 FROM person;
```

RUN ⇩ RESET

What do we know about those people?

If you want all the data for each row in a table, use '*' after 'SELECT'. As you just learned, there are thousands of people, so rather than see all of them, we'll limit the results to just the first 10. Run this, then change the table name or limit number and see what happens.

```
1 SELECT * FROM person LIMIT 10;
```

RUN ⇩ RESET

What are possible values for a column?

When working with data, always see if you can find documentation that explains the database structure (like the ERD) and valid values. But sometimes that's not available. Here we show how the DISTINCT keyword can give you a quick look at which values are in the database. After you run it, delete the word DISTINCT and run it again. See the difference? (After you try that, you may want to click the "reset" button and run one more time before you continue.)

```
1 SELECT DISTINCT type FROM crime_scene_report;
```

RUN ⇩ RESET

What elements does a SQL query have?

A SQL query can contain:

- SQL keywords (like the SELECT and FROM above)
- Column names (like the name column above)
- Table names (like the person table above)
- Wildcard characters (such as `%`)
- Functions
- Specific filtering criteria
- Etc

SQL Keywords

SQL keywords are used to specify actions in your queries. SQL keywords are not case sensitive, but we suggest using all caps for SQL keywords so that you can easily set them apart from the rest of the query. Some frequently used keywords are:

SELECT

SELECT allows us to grab data for specific columns from the database:

- * (asterisk): it is used after SELECT to grab all columns from the table;
- column_name(s): to select specific columns, put the names of the columns after SELECT and use commas to separate them.

FROM

FROM allows us to specify which table(s) we care about; to select multiple tables, list the table names and use commas to separate them. (But until you learn the JOIN keyword, you may be surprised at what happens. That will come later.)

WHERE

The WHERE clause in a query is used to filter results by specific criteria.

Let's try some of these things.

Here's a simple query to get everything about a specific person. (Don't worry—all of the SSNs are made up.)

Note that you need to use single straight quotes (') around literal text so the database can tell it apart from table and column names. After you run it, try again with Yessenia Fossen, Ted Denfip or Davina Gangwer.

```
1 SELECT * FROM person WHERE name = 'Kinsey Erickson'
```

RUN ⇩ RESET

The AND keyword is used to string together multiple filtering criteria so that the filtered results meet each and every one of the criteria. (There's also an OR keyword, which returns rows that match any of the criteria.)

This query only returns reports about a specific type of crime in a specific city. Notice that when querying for text values, you must match the data as it is in the database. Try changing 'Chicago' to 'chicago' and running the statement. Then, see if you can edit this SQL statement to find the first clue based on the prompt above.

```
SELECT * FROM crime_scene_report
WHERE type = 'murder'
AND city = 'SQL City';
/* SELECT * FROM crime_scene_report
WHERE type = 'murder'
AND city = 'SQL City';
```

RUN ⇩ SHOW SOLUTION RESET

Correct

date	type	description	city
20180215	murder	REDACTED REDACTED REDACTED	SQL City
20180215	murder	Someone killed the guard! He took an arrow to the knee!	SQL City
20180115	murder	Security footage shows that there were 2 witnesses. The first witness lives at the last house on "Northwestern Dr". The second witness, named Annabel, lives somewhere on "Franklin Ave".	SQL City

If you haven't found the right crime scene report yet, click "show solution" above, and replace the contents of the box with just the revealed command. (Leave out the initial `/*`) If you figured out the query that shows the single crime scene report instead of a few for the same city and type, then congratulations and disregard the word "incorrect". You'll know if you got it!

Wildcards and other functions for partial matches

Sometimes you only know part of the information you need. SQL can handle that. Special symbols that represent unknown characters are called "wildcards," and SQL supports two. The most common is the `%` wildcard.

When you place a `%` wildcard in a query string, the SQL system will return results that match the rest of the string exactly, and have anything (or nothing) where the wildcard is. For example, `'Ca%a'` matches `Canada` and `California`.

The other, less commonly used wildcard, is `_`. This one means 'match the rest of the text, as long as there's exactly one character in exactly the position of the `_`, no matter what it is. So, `'B_b'` would match `'Bob'` and `'Bub'` but not `'Babe'` or `'Bb'`.

Important: When using wildcards, you don't use the `=` symbol; instead, you use `LIKE`.

Try out some wildcards.

Once you run this command, try variations like 'Irvin_' and 'I%e' -- and then explore some more.

```
1 SELECT DISTINCT city
2 FROM crime_scene_report
3 WHERE city LIKE 'I%';
```

RUN ⇩ RESET

city
Irving
Indianapolis
Irvine
Inglewood
Independence

SQL also supports numeric comparisons like `<` (less than) and `>` (greater than). You can also use the keywords `BETWEEN` and `AND` -- and all of those work with words as well as numbers.

Try out some comparison operators.

```
SELECT DISTINCT city
FROM crime_scene_report
WHERE city BETWEEN 'W%' AND 'Z%';
```

RUN ⇩ RESET

city
Wilmington
Waterbury
West Valley City
Winter Haven
Youngstown
Wichita
West Covina
Yakima
Washington
Winston
Westminster
Waco
Yonkers
Warren
Worcester
Waterloo
York

We've mentioned that SQL commands are not case-sensitive, but `WHERE` query values for `=` and `LIKE` are. Sometimes you don't know how the text is stored in the database. SQL provides a couple of functions which can smooth that out for you. They're called `UPPER()` and `LOWER()`, and you can probably figure out what they do, especially if you explore in the box below.

Try out UPPER() and LOWER().

```
1 SELECT DISTINCT city
2 FROM crime_scene_report
3 WHERE LOWER(city) ='sql city';
```

RUN ⇩ RESET

city
SQL City

Digging deeper

SQL Aggregate Functions

Sometimes the questions you want to ask aren't as simple as finding the row of data that fits a set of criteria. You may want to ask more complex questions such as "Who is the oldest person?" or "Who is the shortest person?" *Aggregate functions* can help you answer these questions. In fact, you learned an aggregate function above, `COUNT`.

How old is the oldest person with a drivers license? With a small amount of data, you might be able to just eyeball it, but there thousands of records in the `drivers_license` table. (Try `COUNT` if you want

to know just how many!) You can't just look over that list to find the answer.

Here are a few useful aggregate functions SQL provides:

MAX
finds the maximum value

MIN
finds the minimum value

SUM
calculates the sum of the specified column values

AVG
calculates the average of the specified column values

COUNT
counts the number of specified column values

Run this query, then try some of the other aggregate functions.

```
1 SELECT max(age) FROM drivers_license;
```

RUN ⇩ RESET

max(age)
89

There's another way to find minimum and maximum values, while also seeing more of the data. You can control the sort order of results you get. It's really quite intuitive: just use `ORDER BY` followed by a column name. It can be challenging when there's a lot of data! (When people get serious about working with SQL, they use better tools than

this web-based system.) By default, ORDER BY goes in "ascending" (smallest to largest, or A to Z) order, but you can be specific with `ASC` for ascending, or you can reverse it with `DESC`.

Run this query to see how to control sort order.

After you've run it, change ASC to DESC, or take that part out completely to see how the results differ. Try sorting on other columns.

```
1 SELECT * FROM drivers_license ORDER BY age ASC LIMIT 10
```

RUN ⇩ RESET

id	age	height	eye_color	hair_color	gender	plate_number	car_make	car_
101255	18	79	blue	grey	female	5162Z1	Lexus	GS
108374	18	63	brown	red	male	X2KE6N	Ford	Esc
112201	18	57	green	green	male	HW66XJ	BMW	325
115674	18	74	blue	blue	female	2OGQIP	Mitsubishi	Dia
122161	18	73	black	black	male	2H3Y1S	BMW	M5
127288	18	59	black	black	female	A20YSP	Ford	Fre
131246	18	57	brown	white	female	VOU6R8	Suzuki	Gra Vita
141220	18	70	green	blue	male	215AK2	Lexus	LX
152848	18	58	brown	blonde	male	4Y00IK	Mazda	Mil
160151	18	67	blue	grey	female	O1G724	Mitsubishi	Mo

By now, you know enough SQL to identify the two witnesses. Give it a try!

Write a query that identifies the first witness.

There's more than one way to do it, so you may learn the answer even if the results say 'Incorrect'

```
SELECT * FROM person
WHERE address_street_name = 'Northwestern Dr'
ORDER BY address_number DESC LIMIT 1;



/* SELECT * FROM person
WHERE address_street_name = 'Northwestern Dr'
ORDER BY address_number DESC LIMIT 1;
```

RUN ⇩ SHOW SOLUTION RESET

Write a query that identifies the second witness.

There's more than one way to do it, so you may learn the answer even if the results say 'Incorrect'

```
SELECT * FROM person
WHERE name like '%Annabel%'
AND address_street_name = 'Franklin Ave';

/* SELECT * FROM person
WHERE name like '%Annabel%'
AND address_street_name = 'Franklin Ave';
```

RUN ⇩ SHOW SOLUTION RESET

Correct

id	name	license_id	address_number	address_street_name	ssn
16371	Annabel Miller	490173	103	Franklin Ave	318771143

Making connections

Joining tables

Until now, we've been asking questions that can be answered by considering data from only a single table. But what if we need to ask more complex questions that simultaneously require data from two different tables? That's where JOINcomes in.

More experienced SQL folks use a few different kinds of JOIN -- you may hear about INNER, OUTER, LEFT and RIGHT joins. Here, we'll just talk about the most common kind of JOIN, the INNER JOIN. Since it's common, you can leave out INNER in your SQL commands.

The most common way to join tables is using primary key and foreign key columns. Refer back to the Entity Relationship Diagram (ERD) above if you don't remember what those are, or to see the key relationships between tables in our database. You can do joins on any columns, but the key columns are optimized for fast results. It is probably easier to show how joins work with our interactive SQL system than to write them.

Run this query to identify the biggest annual earners in our database. (Again, trust us, the SSNs are all made up.)

Try editing the query to return more data, or to find people with different incomes. Try joining other tables together. You can use * in between SELECT and FROM here like with any other query, so try that too.

```sql
SELECT person.name, income.annual_income
FROM income
JOIN person
  ON income.ssn = person.ssn
WHERE annual_income > 450000
```

RUN ⇩ RESET

Sometimes you want to connect more than one table. SQL lets you join as many tables in a query as you like.

Let's see if big earners have other characteristics in common.

This example shows how to join more than one table. It also, incidentally, shows how you can use 'aliases' for the various tables in your query so that you don't have to type as much. Finally, it shows how you can change how a column name shows up in the results, which can be handy.

```sql
SELECT name, annual_income as income,
gender, eye_color as eyes, hair_color as hair
FROM income i
JOIN person p
  ON i.ssn = p.ssn
JOIN drivers_license dl
  ON p.license_id = dl.id
WHERE annual_income > 450000
```

RUN ⇩ RESET

Now that you know how to join tables, you should be able to find the interview transcripts for the two witnesses you identified before. Give it a try!

Write a query that shows the interview transcripts for our two subjects.

There's more than one way to do it, so you may learn the answer even if the results say 'Incorrect'. The official solution does it in one query, but you don't have to. Technically you don't even need to use a JOIN to get the transcripts, but give it a try.

```sql
SELECT person.name, interview.transcript
FROM person JOIN interview
ON person.id = interview.person_id
WHERE person.id = 14887 OR person.id = 16371;



/* SELECT person.name, interview.transcript
FROM person JOIN interview
ON person.id = interview.person_id
WHERE person.id = 14887 OR person.id = 16371;
```

RUN ⇩ SHOW SOLUTION RESET

Correct

name	transcript
Morty Schapiro	I heard a gunshot and then saw a man run out. He had a "Get Fit Now Gym" bag. The membership number on the bag started with "48Z". Only gold members have those bags. The man got into a car with a plate that included "H42W".
Annabel Miller	I saw the murder happen, and I recognized the killer from my gym when I was working out last week on January the 9th.

Go Get 'em!

Now you know enough SQL to solve the mystery. You'll need to read the ERD and make some reasonable assumptions, but there's no other syntax that you need!

Find the murderer!

It will take more than one query to find the killer, but you can just keep editing this box, keeping notes on your results along the way. When you think you know the answer, go to the next section.

```
select person.name
from get_fit_now_member
join person on get_fit_now_member.person_id = person.id
join drivers_license on person.license_id = drivers_license.id
where membership_status = 'gold'
and get_fit_now_member.id like '48Z%'
and plate_number like '%H42W%';

```

RUN ⇩ RESET

name
Jeremy Bowers

Check your solution

Did you find the killer?

```
INSERT INTO solution VALUES (1, 'Jeremy Bowers');

SELECT value FROM solution;
```

RUN ⇩ RESET

value
Congrats, you found the murderer! But wait, there's more... If you think you're up for a challenge, try querying the interview transcript of the murderer to find the real villain behind this crime. If you feel especially confident in your SQL skills, try to complete this final step with no more than 2 queries. Use this same INSERT statement with your new suspect to check your answer.

Credits

The SQL Murder Mystery was created by Joon Park and Cathy He while they were Knight Lab fellows. See the GitHub repository for more information.

Adapted and produced for the web by Joe Germuska.

This mystery was inspired by a crime in the neighboring Terminal City.

Web-based SQL is made possible by SQL.js

SQL query custom web components created and released to the public domain by Zi Chong Kao, creator of Select Star SQL.

Detective illustration courtesy of Vectors by Vecteezy

The Northwestern University Knight Lab is a team of technologists and journalists working at advancing news media innovation through exploration and experimentation.

(847) 467-4971 | 1845 SHERIDAN ROAD , FISK #109 & #111, EVANSTON, IL 60208