from zero to query

a sql primer

oskar 2023-09-13

welcome to this workshop. happy to see your interest.

a note on sqlite

- small (<2mb)
- open source
- serverless
- self-contained
- fast
- complete
- in-memory
- cross-platform
- ubiquitous

we will need an interpreter, a program that reads your queries and converts them into actions onto a database, to retrieve the desired data. today we will be using sqlite to access the training database.
it is an obvious choice: sqlite is the minimal, **the simplest** application to query a database. it is the most popular database tool in the world, and it is built into countless applications. i hope you all have sqlite already installed on your machines, but it is quick to install if not.

sqlite commands

- these are not sql commands!
- they start with a '.'
- they operate on the environment, not the data
- examples:
  - .quit
  - .open <path-to-database>
  - .show
  - .help
  - .cd <directory>
  - .shell CMD ARGS…

you should know a minimum commands for sqlite. note that these are not sql commands, they operate on the interpreter itself, the sqlite environment.
(whereas sql commands define, query, or perform analytical computations on the data in the database).
sqlite commands start with a period. the first one you should know is .quit which quits the app.
.open followed by the path to a database file in the sqlite format will pull in the database and tell sqlite that your sql commands/statements should operate on this database.

```
.open data/sqlite-sakila.db
.tables


.header ON
.mode qbox
```

after starting up sqlite, you will want to run these commands like so. don’t forget the dot in front!
try it now.
first, you open up the training database inside sqlite.
then, some useful but optional command for display purposes.
and finally list all the available tables with the dot-tables command.

.tables

```
sqlite> .tables
actor                   film                    payment
address                 film_actor              rental
category                film_category           sales_by_film_category
city                    film_list               sales_by_store
country                 film_text               staff
customer                inventory               staff_list
customer_list           language                store
sqlite>
```

when you run the dot-tables command, you should be seeing a list of 21 tables, like this.
please let a nearby instructor know if you are not getting this response, as nothing else will make sense if we don't all start from the same starting point.

ok? that is a lot of tables! we won't need **all** of them, but we will be working with many of these.

sql - a fundamental data tool

- database management
- data pipeline engineering
- data modeling
- data designing
- **big** data (parallel, distributed)
- data querying
- data analytics

i won’t try to motivate you that you can benefit from knowing sql. what is worth knowing about sql, is that it is fundamental for various data professions.
whether database managers, data pipeline engineers, data modelers, schema designers, data reporters, data analysts, data scientists, data end users. sql is also the ubiquitous, default language used to interact with any big data tool.

data definition	data management	data querying	data control	transaction control
to operate on entire tables	to operate on table values, rows, columns	to fetch data from tables	to control access to schemas + tables	for transactional atomicity, dev
CREATE	INSERT	SELECT	GRANT	COMMIT
DROP	UPDATE		REVOKE	ROLLBACK
ALTER	DELETE			SAVE POINT
TRUNCATE				

sql is the collective term for 5 components: a data definition language, data management language, data querying language, data control language, and a transaction control language.

data definition	data management	data querying	data control	transaction control
to operate on entire tables	to operate on table cells, rows, columns	to fetch data from tables	to control access to schemas + tables	for transactional atomicity, dev
CREATE	INSERT	SELECT	GRANT	COMMIT
DROP	UPDATE		REVOKE	ROLLBACK
ALTER	DELETE			SAVE POINT
TRUNCATE				

this workshop is aimed for beginners and we are going to ignore most of that. today we focus one the data querying language, which is the feature of sql used for extracting information from database tables (querying) and for performing analytical computations. the good news is that today, you only need to learn to use a single command, the SELECT statement, which lies at the heart of sql. it is what any user of sql needs to know, so it is an obvious starting point.

sql commands

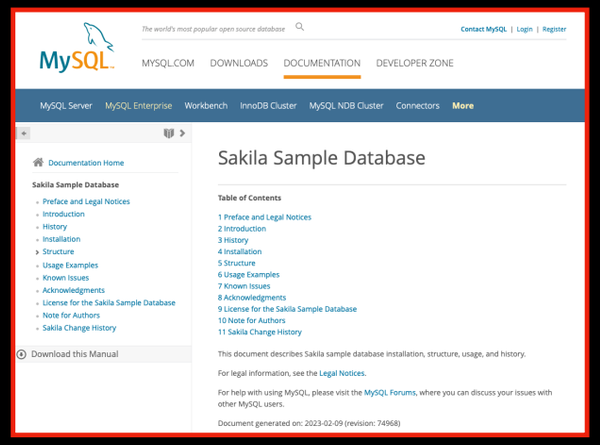
- run on the database
- operate on the data tables
- output a table
- start with SELECT … clause
- contain a FROM … clause
- end with a ‘;’
- you can add comments with ‘-- a comment’
- example:

```
SELECT name FROM category; -- film categories
```

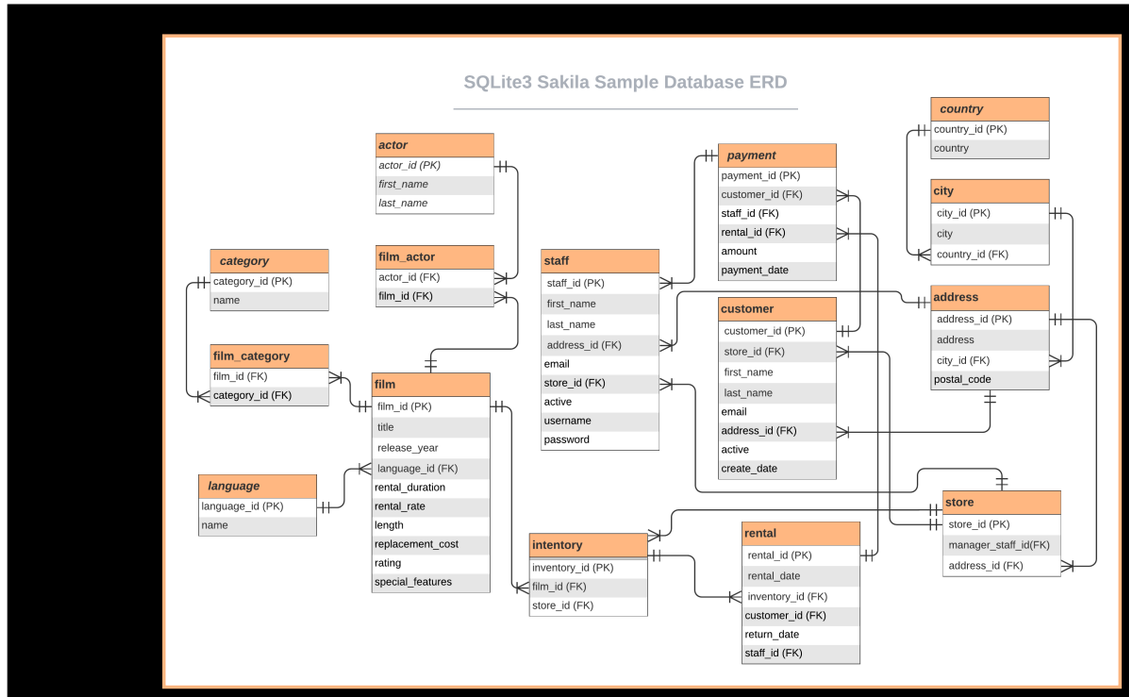
a note about sql commands. you will write a number of these today. they do not start with a period. they always end in a semicolon. you will undoubtedly forget the semicolon at some point today. that’s ok. we are here to point it out. but start now getting used to adding a semi colon to every sql statement. (just the sql statements, and not the sqlite commands!)

the sakila training data

- classic, fictional data
- dvd rental company
- 20 relational tables:
  - normalised, no repetition
  - stores
  - inventory
  - films
  - film casting
  - actors
  - film ratings



now, let’s talk about the training data. this is a classic, synthetic database made for training purposes. if you are familiar with r or python you will know these classic datasets: the titanic dataset, mpg, iris, mtcars, penguins, etc. in sql there are a few of those too. this the smallest one with full features is called sakila and describes the database of a chain of video rental stores. yes, this is an outdated concept. just pretend that its is 1999.



i have printed a handout for you and you can also find this entity relationship diagram in the repo under img/ note that for each link between two tables table there is a corresponding relationship between the records of the two tables. this will be a lot more clear once we actually work with the data, so refer back to this diagram later.

today's objective:

"which **top 10** *actors* were *rented* out the greatest number of times, counting only '**R**' *rated films* made in **2006**?"

now consider this dilemma: imagine you are running a chain of dvd rental stores, and you are deciding how much stock of upcoming films to procure. you have no data on the popularity of the new films since they are not out yet. but you do know their cast. and you have noticed that films with certain actors are rented out more than others. some actors might be rented out a lot more than others even if their individual films are not the most popular films. so you ask yourself: "who are the actors that get rented out the most number of times? i must ensure i will get plenty of stock of the upcoming films featuring those actors."
the answer to this question lies in the database, but in order to conjure it out of there requires a specific query. the rest of this workshop is about building the skill required to generate that query.

today's plan:

"which top 10 actors were rented out the greatest number of times, counting only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table};
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
```

here again is the question we want to learn how to answer, and the list of sql *component clauses* required to answer the question. by the end of this workshop, you will be able to assemble these components together to construct a sql query to answer any such question of the data in your database.

what do the tables contain?

"which top 10 actors were rented out the greatest number of times, counting only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table};
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
```

we have seen what tables are available in the training database. next, we start asking ourselves what data these tables contain. and here is where the venerable SELECT statement comes in. the output of a SELECT statement is a *table* of results.
at a *minimum* we must specify 2 pieces of information to the select statement.

1) from which table we want to query, and
2) which of those table's columns we want returned.

for the columns, we can use the asterisk wildcard (indicating all available columns) a la "SELECT * FROM a_tablename;" choose a table in the database and try it out!

SELECT {columns} FROM {table};

```
- SELECT * FROM staff;
  -- returns all columns and all rows from the staff table

- SELECT title, rating FROM film;
  -- returns title and rating (in order) from the table film

- SELECT c.first_name AS customer_name FROM customer c;
  -- sets an alias for table customer, renames column to 'customer_name'

- SELECT title, replacement_cost/rental_rate AS break_even_count FROM film;
  -- returns the number of rentals a film needs to break even

- SELECT DISTINCT a.last_name FROM actor a;
  -- returns all the first names in the actor table, with no duplicates
```

this is the minimal SELECT statement: here are examples on how to fetch information from tables. go ahead try these commands or versions of them, replacing the table and column names with tables you found in the previous step.
note

1. every command ends with a semicolon
2. you can alias results and table names

exercises

- show all the columns of the category table
- rename the name column to category_name in the output

now you try:

SELECT {aggregate function} FROM {table};

```
- SELECT COUNT(*) AS num_records FROM actor;
  -- returns the number of rows in table actor, names the output ‘num_records’

- SELECT COUNT(DISTINCT rating) FROM film;
  -- returns a count of distinct values in the rating column

- SELECT AVG(replacement_cost) AS avg_cost FROM film;
  -- returns the average replacement cost of a film

- SELECT AVG(rental_rate + replacement_cost) AS average_total_cost FROM film;
  -- returns the average rate of rental from film table

- SELECT MAX(rental_rate) AS highest_rental_rate FROM film;
  -- returns the most expensive rental_rate from film

- SELECT MIN(length) AS shortest_length FROM film;
  -- returns the length of the shortest film
```

in addition to extracting each value from each row, we can also request sql deliver us aggregates of all the rows, e.g. counts, sums, averages, min, max, etc. try these examples. each should output a single value. note that you can either ask for a column value, OR for an aggregate value, but it does not make sense to mix the two.

exercises

- what is the maximum number of times that a film needs to be rented out to break even?
- what is the average number of times that a film needs to be rented out to break even?

try your hand at one of these.

that's too many rows!

"which top 10 actors were rented out the greatest number of times, counting only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table}
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
    ;
```

some of our tables contain a large number of records. we don't want to overwhelm ourselves or our screens with hundreds, thousands, millions of rows. that is not useful to see. so we can use a LIMIT clause within our SELECT statement to achieve that. note limit only trims the output at the limit count. it doesn't specify which rows to return, only that the returned rows should not be greater than <num> in number

SELECT … FROM … LIMIT …;

```
- SELECT * FROM film LIMIT 5;
  -- returns all columns of 5 unspecified rows from film

- SELECT * FROM category LIMIT 5;
  -- returns 5 unspecified rows of all columns from category

- SELECT title, release_year FROM film LIMIT 15;
  -- returns 15 unspecified rows of two columns from film table

- SELECT r.rental_id, r.rental_date FROM rental r LIMIT 10;
  -- returns rental id and date of rental for 10 unspecified rows

- SELECT first_name||' '||last_name AS fullname FROM actor LIMIT 10;
  - returns the full names of 10 unspecified actor
```

here are examples on how to fetch limited information from tables. go ahead try these commands or versions of them, replacing the table and column names, note that limit 5 just tells sql that we want no more than 5 rows. we have not specified in any way which 5 rows we want. they could be any rows in the table. sql gets to decide which rows to serve us, and it normally chooses whatever rows it can produce in the fastest time. in many database systems

exercises

- show the first 5 records of any table, using a LIMIT clause
- output the full name of 5 customers, using || to paste strings

try your hand at one of these.

but i only want the most extreme rows!

"which top 10 actors were rented out
the greatest number of times, counting
only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table}
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
    ;
```

next we introduce the ORDER BY clause to the SELECT STATEMENT. it simply specifies in what order we want our result table sorted.
say if we want the top 10 most expensive items we cannot just put that in a WHERE clause without knowing what the price of the 11th most expensive item is. so instead we can use another way to specify which records we want returned: by sorting the records by some criteria and then LIMITing the number of records to just a few of rows.

SELECT … FROM … ORDER BY … LIMIT …;

```
- SELECT * FROM payment ORDER BY payment_date LIMIT 7;
  -- return the earliest 7 payments in the payment table

- SELECT * FROM payment ORDER BY payment_date DESC LIMIT 7;
  -- return the latest 7 payments in the payment table

- SELECT * FROM payment ORDER BY amount DESC LIMIT 5;;
  -- return only the top 5 highest payment amounts from the payment table
```

ok let us try it. first, let us take a look at the payment table. the first command orders the table by the payment date, and returns only the 7 earliest records.
the next one is almost the same, with only one small difference. to order a table in the opposite order we we ORDER BY column DESC (for descending order, ascending is default and assumed).

but i only want specific rows!

"which top 10 actors were rented out the greatest number of times, counting only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table}
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
    ;
```

we will now start to get more specific about what information we want to extract from the tables. the first limitation we applied was by selecting specific *columns*. then we learned how to only retrieve a limited number of unspecified rows. now, let us select *specific rows* from the table, namely rows where some specific column values are found. the way to do that in sql is by adding a WHERE clause, in the SELECT statement, *right after* the FROM clause.

comparison operators

comparison syntax	meaning
{column} = {expression}	column value is equal to expression value
{column} <> {expression}	column value is not equal to expression value
{column} != {expression}	column value is not equal to expression value
{column} < {expression}	column value is less than expression value
{column} <= {expression}	column value is less than or equal to expression value
{column} > {expression}	column value is greater than expression value
{column} >= {expression}	column value is greater than or equal to expression value
{column} IN ({exp1,exp2,...})	column value is one of exp1, exp2, …
{column} LIKE '%expr%'	(string) column contains substring 'expr'
{column} BETWEEN {exp1} AND {exp2}	{exp1} <= column value <= {exp2}

here are the comparison operators you can use to make a condition. some notes:

1. if you are used to c, r, python, and some other programming languages you may have expected equality being represented as '==', but that is not the case for sql. in sql "a=b" evaluates to true if a and b are equal.
2. '<>' and '!=' are synonyms for "not equal" .

SELECT {column} FROM {table} WHERE {cond}

```
- SELECT * FROM actor WHERE LENGTH(last_name) = 3;
  -- returns only records of actors whose last name is three characters

- SELECT title AS name FROM film f WHERE rating <>'R AND;
  -- returns only rows where the value in column1 is not {expression}

- SELECT title AS film_name, rental_rate FROM film WHERE rental_rate<=1.0;
  -- returns titles of films whose rental price is at most £1

- SELECT first_name FROM staff WHERE store_id=2;
  -- returns the first names of staff at store with id 2
```

we will now start to get more and more specific about what information we want to extract from the the tables. the first limitation was selecting specific columns. now let the next limitation we set on the data be about selecting specific rows. the way to do that is by adding a WHERE clause in the SELECT statement. try it.
also note that you can still add a LIMIT clause as well.

exercises

- show the first_names of inactive customers (active = '0')
- how many payments have amounts greater than $10.00?

try your hand at one of these.

comparison operators

comparison syntax	meaning
{column} = {expression}	column value is equal to expression value
{column} <> {expression}	column value is not equal to expression value
{column} != {expression}	column value is not equal to expression value
{column} < {expression}	column value is less than expression value
{column} <= {expression}	column value is less than or equal to expression value
{column} > {expression}	column value is greater than expression value
{column} >= {expression}	column value is greater than or equal to expression value
{column} IN ({exp1,exp2,…})	column value is one of exp1, exp2, …
{column} LIKE '%expr%'	(string) column contains substring 'expr'
{column} BETWEEN {exp1} AND {exp2}	{exp1} <= column value <= {exp2}

1. IN (),
2. LIKE "%",
3. and BETWEEN…AND

   are special, sql specific, and useful comparison operators.

SELECT … FROM … WHERE …;

```
- SELECT * FROM rental WHERE rental_date BETWEEN '2005-08-16' AND '2005-08-17';
  -- returns only rentals occurring in

- SELECT * FROM payment WHERE amount IN (7.98, 8.97);
  -- returns info on all payments of a specific amount

- SELECT * FROM city WHERE city LIKE 'Ok%';
  -- returns info on all cities whose name begins with ‘Ok’

- SELECT last_name AS full_name FROM customer WHERE first_name LIKE 'AL%';
  -- returns the last name of all customers whose first name begins with ‘AL’
```

here are some examples of how to use BETWEEN, IN, LIKE:

exercises

- what city names begin with a 'Q'?
- which actors' first_names end with 'K'?

try your hand at one of these.

how can i aggregate groups of rows into a single row?

"which top 10 actors were rented out
the greatest number of times, counting
only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table};
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {a_condition}
    GROUP BY {columns}
    HAVING {a_condition}
    ORDER BY {columns}
    LIMIT {num}
```

we will often want a summary of a table, e.g. the sum or an average of a column. we saw before how we could do that across all the rows in a table.
very often we want to treat groups of rows as separate, isolated segments, and sum together some measure only among rows within the same segment. that is what the GROUP BY clause is for.

SELECT {col} FROM {tab} GROUP BY {col};

```
- SELECT city_id, COUNT(*) AS num_address FROM address GROUP BY city_id;
  -- return number of addresses in each city in address table

- SELECT rating, AVG(length) AS avg_len FROM film GROUP BY rating ORDER BY avg_len;
  -- returns the average length of a movie in each rating category

- SELECT country_id, COUNT(*) AS num_cities
  FROM city
  GROUP BY country_id
  ORDER BY num_cities DESC
  LIMIT 5;
  -- return top 5 country ids, by number of cities assigned to each
```

when you run a group by query, you do not get a response table with one row per row in the input. rather you get one row per segment, per group, per distinct value in the group by column. alongside each distinct value of the grouping variable, you can also get a summary of other columns. (recall the aggregate functions from above: SUM(), COUNT(), COUNT(DISTINCT), AVG(), MAX(), MIN().

how do i report only some aggregated groups?

"which top 10 actors were rented out
the greatest number of times, counting
only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table};
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {column}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
    ;
```

when we want to aggregate over specific groups of rows, but are only interested in some of the outcomes we can filter on the aggregated rows (one row per segment or per *category*). for instance,
there is an actor's name in the actors table that is repeated. that is: two records in that table have the same first and last name. how to find a repeated name in the list?
if you counted the number of films with each rating, but excluded from the output, ratings where counts .

SELECT … FROM … GROUP BY … HAVING …;

```
- SELECT col1, COUNT(*) AS num FROM table GROUP BY col1 HAVING num>9;
  -- count instances of each value of col1, but only output rows with count>9

- SELECT rating, AVG(length) AS len FROM film GROUP BY rating HAVING len<115;
  -- the film rating categories with average length of film under 115 minutes

- SELECT actor_id, COUNT(*) AS n FROM film_actor GROUP BY actor_id HAVING n<15;
  -- which actor ids have appeared in fewer than 15 films?
```

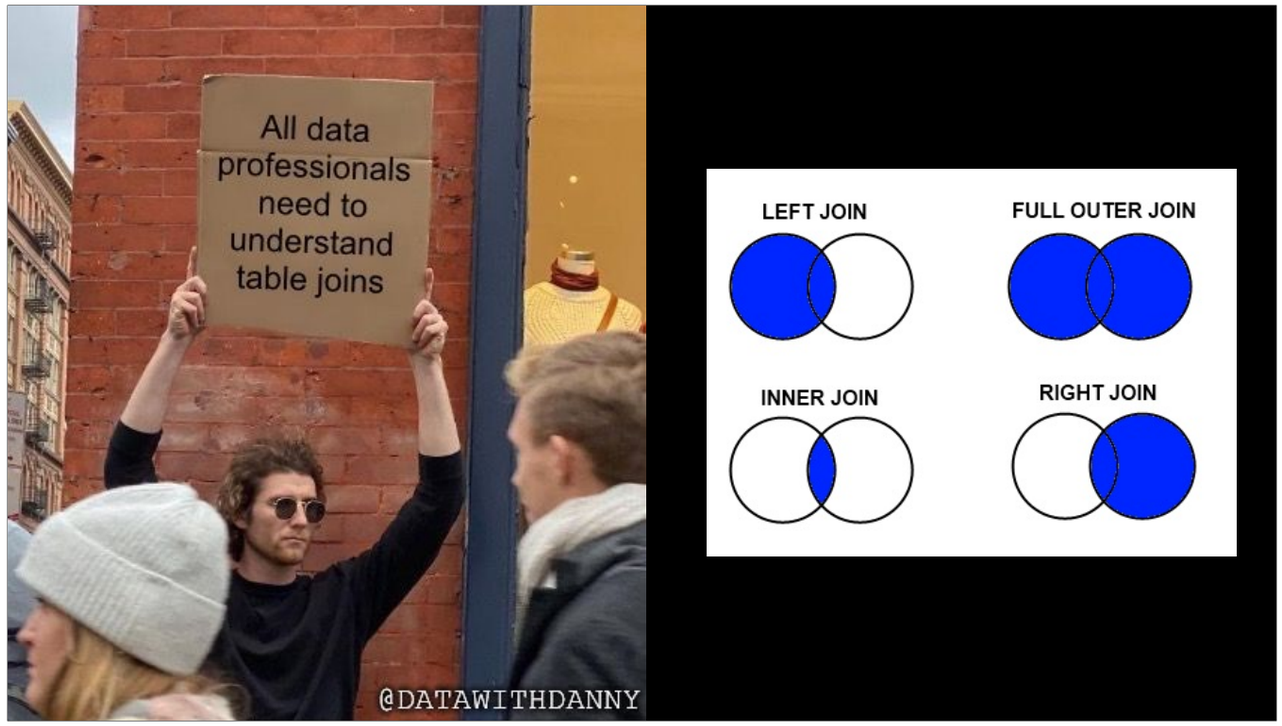
HAVING clause is like a WHERE clause: it places a condition on the rows returned. but unlike WHERE clause, it does not filter the rows of the input table, but rather filters the rows of the aggregated table, before it is returned and can be used to filter out specific values of the aggregated values.
note that it doesn’t make sense to include a HAVING clause in a SELECT statement that doesn’t have aggregation functions..

but my information is spread over two tables!

"which top 10 actors were rented out
the greatest number of times, counting
only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table};
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
    ;
```

often, the data we need is an amalgam of information spread across multiple tables. in order to get precisely the data we need, we must JOIN multiple tables. the JOIN clause is the trickiest concept we will talk about today. let's take it slow.



there are multiple types of joins. which to use depends on what you want the output to be. you will most often encounter either left join or an inner join. you use a left join whenever you have a table that already has most of the information you need, but you need to add a column to that table with values from another table.
an inner join is appropriate if the table you already have and the other table, share a unique and complete primary key.

JOIN

city

city_id	city	country_id	last_update
1	A Corua (La Corua)	87	2021-03-06 15:51:49
2	Abha	82	2021-03-06 15:51:49
3	Abu Dhabi	101	2021-03-06 15:51:49
4	Acua	60	2021-03-06 15:51:49
5	Adana	97	2021-03-06 15:51:49
6	Addis Abeba	31	2021-03-06 15:51:49
7	Aden	107	2021-03-06 15:51:49
8	Adoni	44	2021-03-06 15:51:49

...

country

country_id	country	last_update
1	Afghanistan	2021-03-06 15:51:49
2	Algeria	2021-03-06 15:51:49
3	American Samoa	2021-03-06 15:51:49
4	Angola	2021-03-06 15:51:49
5	Anguilla	2021-03-06 15:51:49
6	Argentina	2021-03-06 15:51:49
7	Armenia	2021-03-06 15:51:49
8	Australia	2021-03-06 15:51:49
9	Austria	2021-03-06 15:51:49

...

consider these two tables, just the top few rows are shown. the first one lists a number of cities in the world. the second one lists countries.

JOIN

city

city_id	city	country_id	last_update
1	A Corua (La Corua)	87	2021-03-06 15:51:49
2	Abha	82	2021-03-06 15:51:49
3	Abu Dhabi	101	2021-03-06 15:51:49
4	Acua	60	2021-03-06 15:51:49
5	Adana	97	2021-03-06 15:51:49
6	Addis Abeba	31	2021-03-06 15:51:49
7	Aden	107	2021-03-06 15:51:49
8	Adoni	44	2021-03-06 15:51:49

...

country

country_id	country	last_update
1	Afghanistan	2021-03-06 15:51:49
2	Algeria	2021-03-06 15:51:49
3	American Samoa	2021-03-06 15:51:49
4	Angola	2021-03-06 15:51:49
5	Anguilla	2021-03-06 15:51:49
6	Argentina	2021-03-06 15:51:49
7	Armenia	2021-03-06 15:51:49
8	Australia	2021-03-06 15:51:49
9	Austria	2021-03-06 15:51:49

...

note that one of the properties given for each city is the country it is in.but it doesn’t say the country name, just the id of the country, which is a look up key in the country table.

JOIN

city

city_id	city	country_id	last_update
1	A Corua (La Corua)	87	2021-03-06 15:51:49
2	Abha	82	2021-03-06 15:51:49
3	Abu Dhabi	101	2021-03-06 15:51:49
4	Acua	60	2021-03-06 15:51:49
5	Adana	97	2021-03-06 15:51:49
6	Addis Abeba	31	2021-03-06 15:51:49
7	Aden	107	2021-03-06 15:51:49
8	Adoni	44	2021-03-06 15:51:49

...

Turkey

country

country_id	country	last_update
1	Afghanistan	2021-03-06 15:51:49
2	Algeria	2021-03-06 15:51:49
3	American Samoa	2021-03-06 15:51:49
4	Angola	2021-03-06 15:51:49
5	Anguilla	2021-03-06 15:51:49
6	Argentina	2021-03-06 15:51:49
7	Armenia	2021-03-06 15:51:49
8	Australia	2021-03-06 15:51:49
9	Austria	2021-03-06 15:51:49

...

if we are interested in finding out which country that city called adana is in, we need to find the country with id ’97’. that country happens to be turkey.

JOIN

city

city_id	city	country_id	last_update
1	A Corua (La Corua)	87	2021-03-06 15:51:49
2	Abha	82	2021-03-06 15:51:49
3	Abu Dhabi	101	2021-03-06 15:51:49
4	Acua	60	2021-03-06 15:51:49
5	Adana	97	2021-03-06 15:51:49
6	Addis Abeba	31	2021-03-06 15:51:49
7	Aden	107	2021-03-06 15:51:49
8	Adoni	44	2021-03-06 15:51:49

...

Turkey

Yemen

country

country_id	country	last_update
1	Afghanistan	2021-03-06 15:51:49
2	Algeria	2021-03-06 15:51:49
3	American Samoa	2021-03-06 15:51:49
4	Angola	2021-03-06 15:51:49
5	Anguilla	2021-03-06 15:51:49
6	Argentina	2021-03-06 15:51:49
7	Armenia	2021-03-06 15:51:49
8	Australia	2021-03-06 15:51:49
9	Austria	2021-03-06 15:51:49

...

likewise ‘aden’ is in yemen. and so on and so forth. but we don’t want to have to do the looking up. we want sql to do that for us.

we want this

city-and-country

city_id	city	country
1	?	?
2	?	?
3	?	?
4	?	?
5	?	?
6	?	?
7	?	?
8	?	?

...

you want to produce something like this. here is a hint on how to work use the JOIN clause: always picture what you expect and want the output to look like first, then write the code. make sure you have a clear image of what you are trying to achieve first.

we want this

city-and-country

city_id	city	country
1	A Corua (La Corua)	Spain
2	Abha	Saudi Arabia
3	Abu Dhabi	United Arab Emirates
4	Acua	Mexico
5	Adana	Turkey
6	Addis Abeba	Ethiopia
7	Aden	Yemen
8	Adoni	India

...

now, it turns out that sql is brilliant at producing tables like that. the syntax magic to make something like this work all happens in an addition to the FROM clause.

so we add a JOIN to the WHERE clause

city

city_id	city	country_id	last_update
1	A Corua (La Corua)	87	2021-03-06 15:51:49
2	Abha	82	2021-03-06 15:51:49
3	Abu Dhabi	101	2021-03-06 15:51:49
4	Acua	60	2021-03-06 15:51:49
5	Adana	97	2021-03-06 15:51:49
6	Addis Abeba	31	2021-03-06 15:51:49
7	Aden	107	2021-03-06 15:51:49
8	Adoni	44	2021-03-06 15:51:49

...

+

country

country_id	country	last_update
1	Afghanistan	2021-03-06 15:51:49
2	Algeria	2021-03-06 15:51:49
3	American Samoa	2021-03-06 15:51:49
4	Angola	2021-03-06 15:51:49
5	Anguilla	2021-03-06 15:51:49
6	Argentina	2021-03-06 15:51:49
7	Armenia	2021-03-06 15:51:49
8	Australia	2021-03-06 15:51:49
9	Austria	2021-03-06 15:51:49

...

```
SELECT
    city_id, city.city, country.country
FROM
    city
    INNER JOIN country ON city.country_id=country.country_id
;
```

so you have FROM table_a JOIN table_b ON table_a.key_column=table_b.key_column…

so we add a JOIN to the WHERE clause

city-and-country

city_id	city	country
1	A Corua (La Corua)	Spain
2	Abha	Saudi Arabia
3	Abu Dhabi	United Arab Emirates
4	Acua	Mexico
5	Adana	Turkey
6	Addis Abeba	Ethiopia
7	Aden	Yemen
8	Adoni	India

=

...

```
SELECT
    city_id, city.city, country.country
FROM
    city
    INNER JOIN country ON city.country_id=country.country_id
;
```

and here is the output of that

SELECT … FROM a INNER JOIN b ON a.key=b.key;

```
- SELECT a.city, b.country
  FROM city a
    INNER JOIN country b ON a.country_id=b.country_id
  LIMIT 10
  ; -- output a table with city-country names

- SELECT f.title, f.length, l.name
  FROM film f
    INNER JOIN language l ON f.language_id=l.language_id
  WHERE f.rating='R'
  LIMIT 10; -- output a sample of films and the name of the language it is in
```

try it!

SELECT … FROM a INNER JOIN b ON a.key=b.key;

```
SELECT
    f.title AS film_title,
    c.name  AS category
FROM film f
    INNER JOIN film_category fc ON f.film_id=fc.film_id
    INNER JOIN category c ON fc.category_id=c.category_id
WHERE f.rating IN ('G', 'PG') AND f.length BETWEEN 85 AND 90
;
```

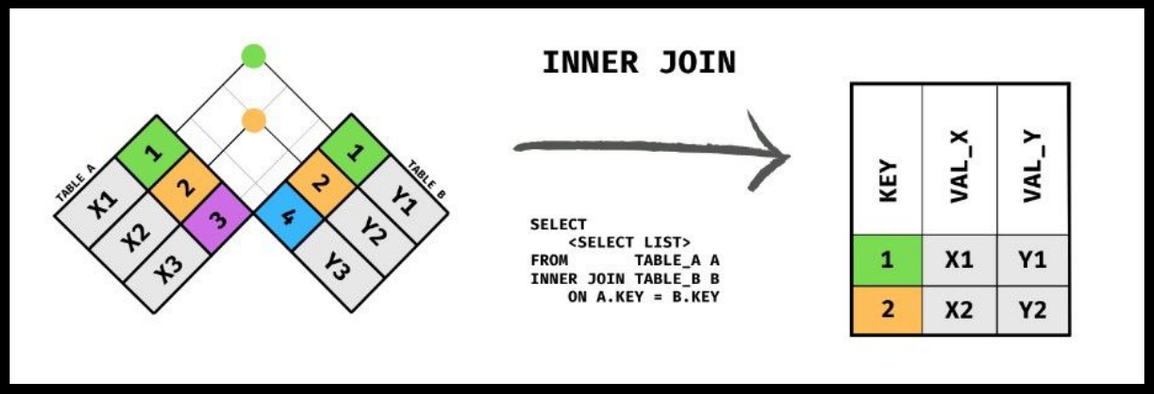
try it!

exercises

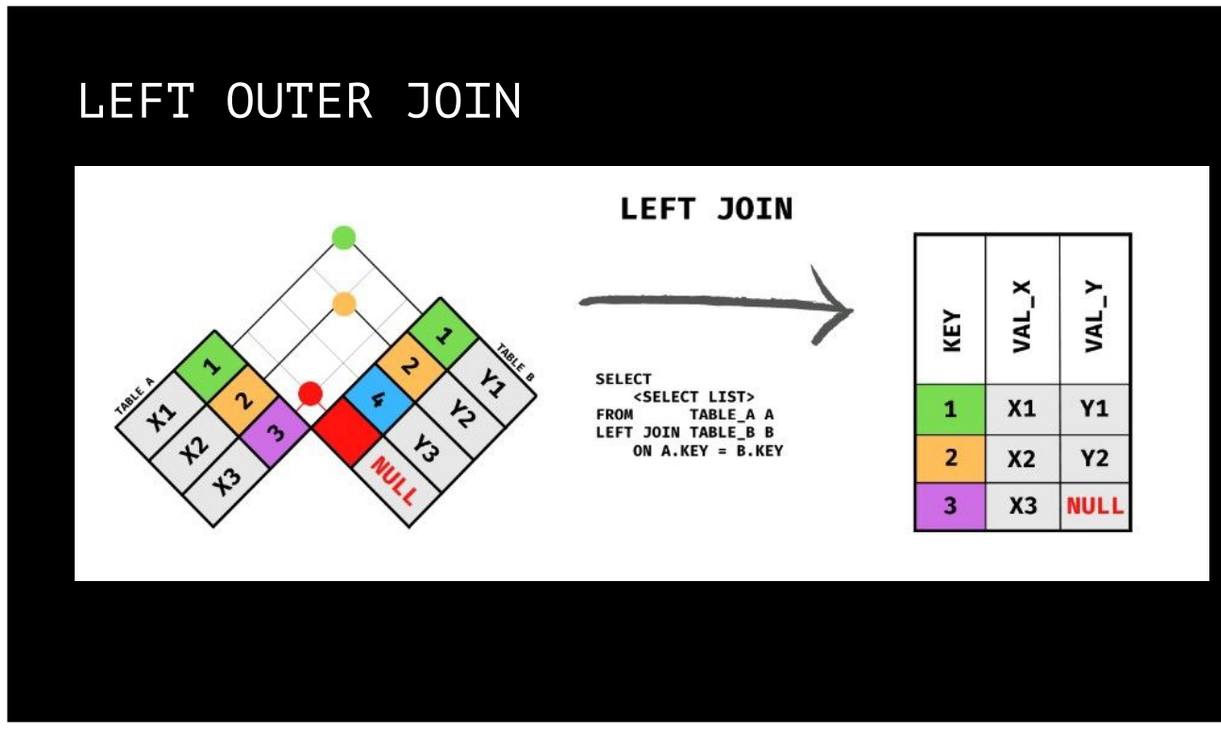
- what countries are the cities whose name starts with 'Q' in?

try your hand at one of these.

FROM a INNER JOIN b

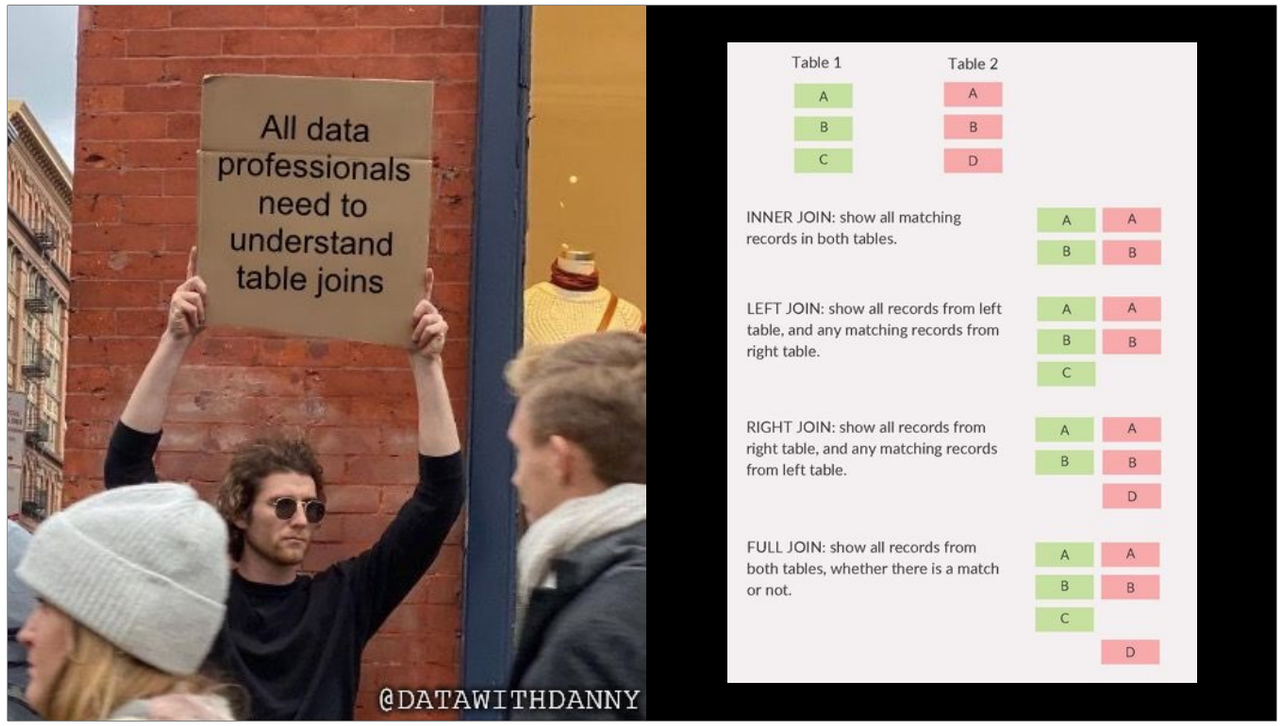


we just showed you an INNER JOIN. that is when the join two tables only returns only the rows where the keys from both tables match. for our city + country table example, we are assuming that **there is no city** in that table *with an associated country_id that does not exist in the country table*. had that been the case, then that city would not occur in the output. likewise, if there were a country in the country table with a country_id that no city in the city table was associated with, then that record would no appear in the output of an INNER JOIN.



the left join is also known as the left outer join, and is very common.
FROM table_a LEFT JOIN table_b.
this method of merging tables treats the rows from table_a (the left hand table) with preference, and every row from table_a is guaranteed to be found in the output (whether or not it matches anything in table_b). the information in a row from table_b (right hand table) is only included if its key matches a row in the left hand table, otherwise it is ignored.
(i.e. if there were a country in the country list that no city in the city list belonged to, we would not include that country in the output).

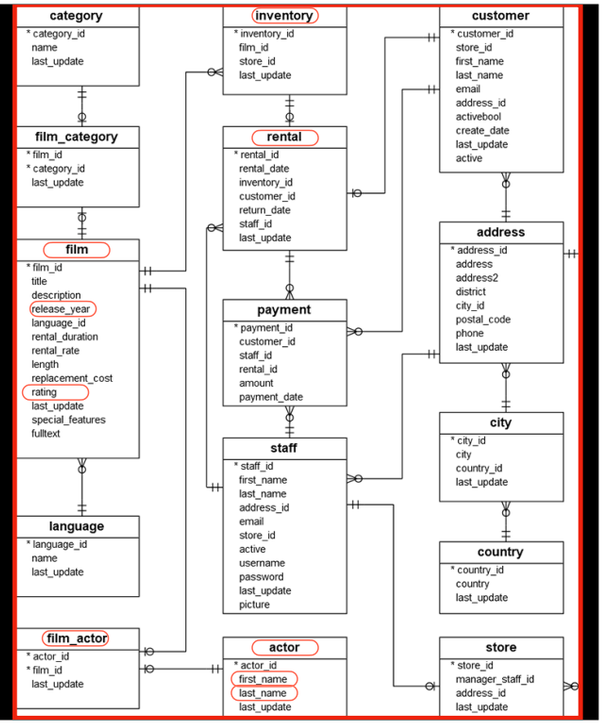
the point is that joins can be tricky and often trip people up. be careful and always, always, always start by thinking careful about what the output should look like and what should happen if your key has duplicate instances in either table.



as you can see, joining tables is a crucial data skill. if you have a background in r, python, excel: you might call this by a different name: merge, mesh, vlookup. they are all related. now we just showed you inner join. it will only return rows for keys that are found in both tables.
there are multiple kinds of join. for a properly set up relational database, like we have, the inner join is most useful, that is because we can trust our primary keys to be unique and present in every row. if our database had some wonky data, with broken primary keys, primary keys missing or duplicated, we would need to be more careful.
an inner join only returns rows where the join key value is found in both tables.
we also often use a left join. in a left join we ensure that all the rows that in the table on the left of the join clause are present, whether or not the corresponding key is found in the right hand table.
right joins are rarely if ever needed, you can just turn the join around as a left join.
there are other join types, but we will focus on these.

today's objective:

"which **top 10** *actors* were *rented* out the greatest number of times, counting only '**R**' *rated films* made in **2006**?"



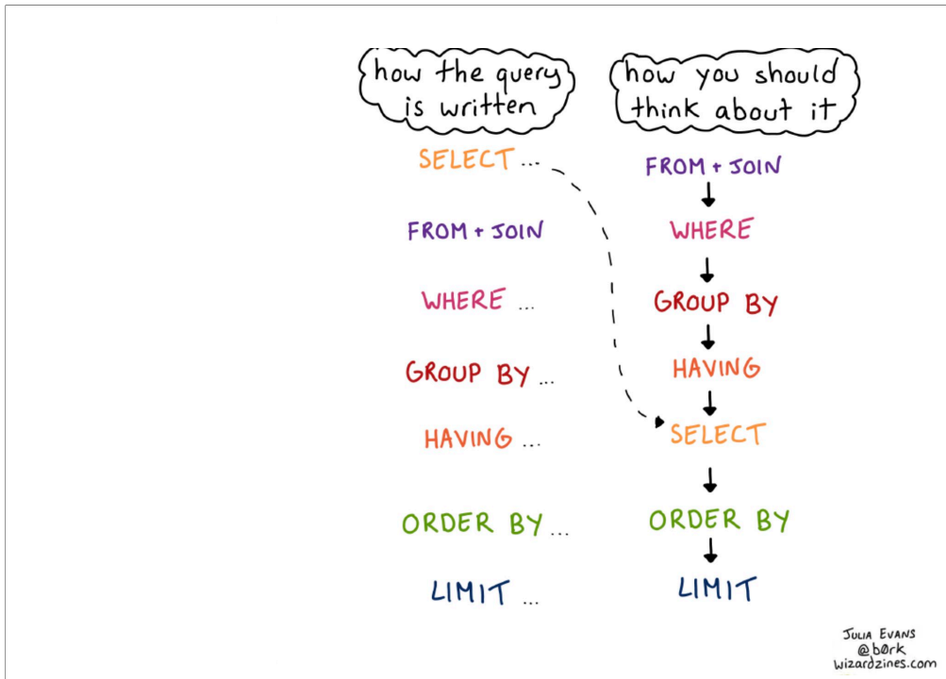
to answer the question at hand we are going to have to join the tables shown here: the actor table, the film_actor, the film table the inventory, the rental

how do i combine the components of a SELECT?

"which top 10 actors were rented out
the greatest number of times, counting
only 'R' rated films made in 2006?"

```
- SELECT {columns} FROM {table};
    INNER JOIN {table_2} ON {col1}={col2}
    WHERE {condition}
    GROUP BY {columns}
    HAVING {condition}
    ORDER BY {columns}
    LIMIT {num}
    ;
```

excellent. now we have covered **all the components that we need** to answer the question. all that remains is combining these components all in the particular configuration that gives us the right answer. sql is strict about the order in which the components are combined.



but the order that sql demands us to type in the command is not the same as the order we usually think about data transformations from source data to solution.
follow the right hand side pathway:
- you start with a table, which you may **join** with another table to get a **merged** table,
- you then **filter** out some of the rows from that table,
- then you **group** the remaining rows into segments,
- then you **filter out the segments** that you want,
- then you **select** columns (the group categories and aggregations within each segment, computations on the columns etc) that you want in the output.
- and then you (optionally) **sort** the resulting table of segments.
- finally you (optionally) **trim** the output to the desired length.

that's simple! however, sql demands we arrange the components in the slightly different order as shown on the left hand column. the steps are all conceptually the same, and the output is the same, it is just the syntax that requires the SELECT clause in front of all the rest.

your turn! compose a query to answer:

"which top 10 actors were rented out the greatest number of times, counting only 'R' rated films made in 2006?"

that's it! now just work on the solution

hint: structure of the solution

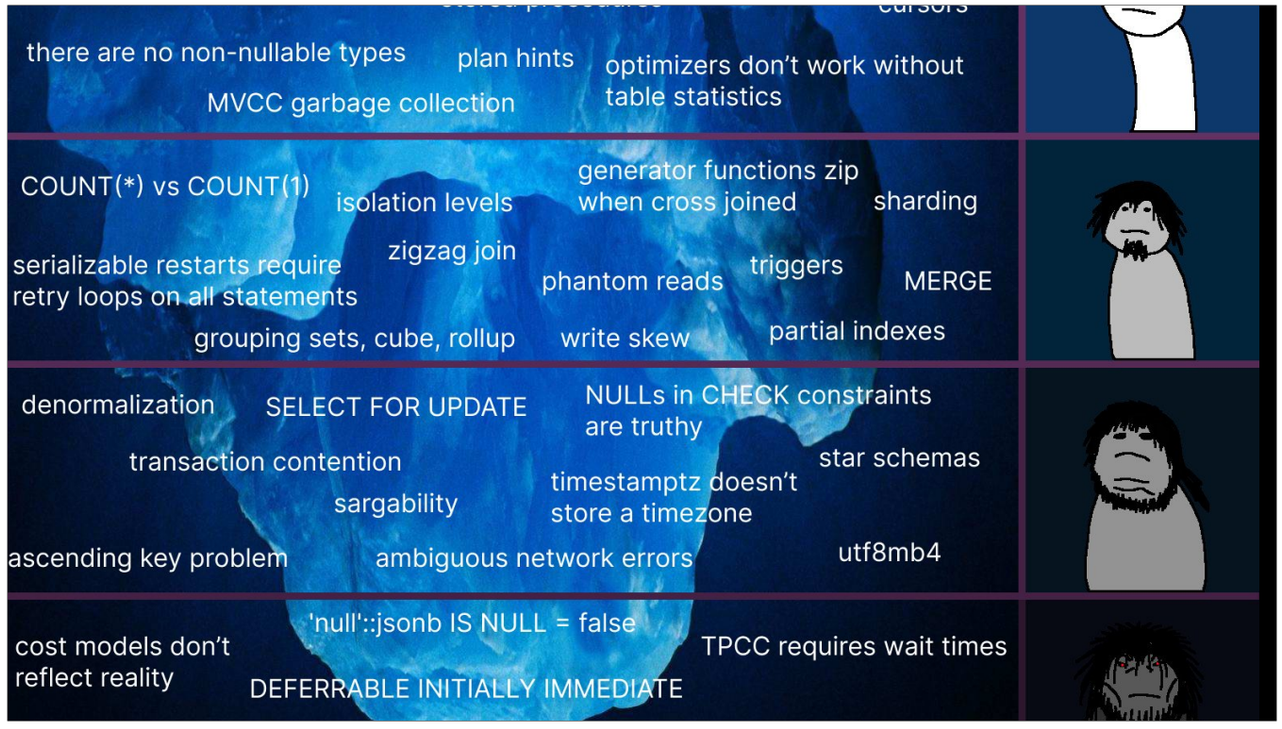
```
SELECT
   {}        AS actor_name,
   COUNT({}) AS num_rentals
FROM {table1}
    INNER JOIN {table2} ON {join-condition}
    INNER JOIN {table3} ON {join-condition}
    INNER JOIN {table4} ON {join-condition}
    INNER JOIN {table5} ON {join-condition}
WHERE {row condition1}
    AND {row condition2}
GROUP BY {column1}
ORDER BY {column} DESC
LIMIT {num}
;
```

we have 15 minutes left. if you are finding the assignment hard, take a look at this code. this is the structure of the solution. you just need to replace the curly brackets with an appropriate expression

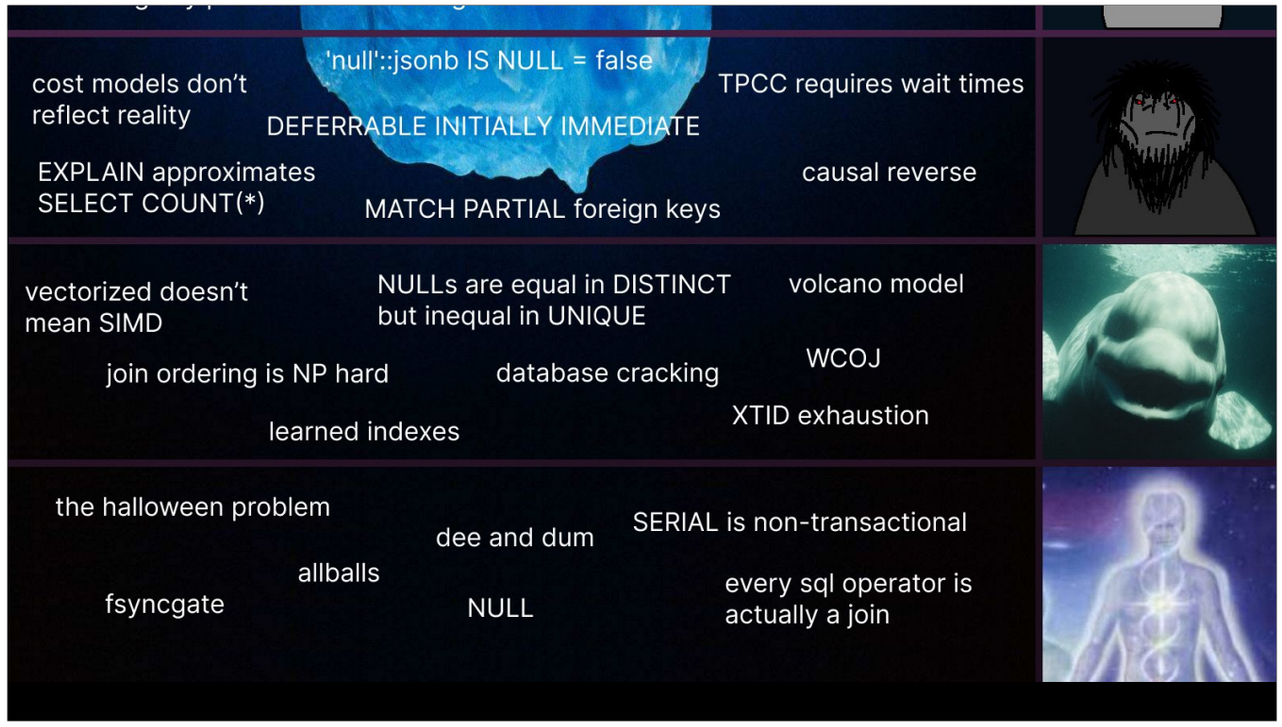
you made it! congratulations! give yourselves a round of applause. you have learned a lot today:
we covered how to query a database for quite specific answer, collecting information from across multiple tables.

we hope that this workshop has sparked your curiosity about sql.
if you keep digging you will find that sql has plenty more to explore.
there is a lot more to sql than we were able to cover here.

here are some of the topics covered today.
a sensible next step from here would be learn more analytics functions and how use something called window functions in sql, and common table expressions.



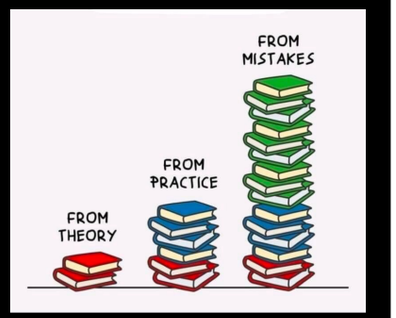
beyond that, you will find a lot of sql's power, usefulness, and ubiquity stems from how it handles the complexities of database management,



i have no idea what most of these mean. but i suspect that you will never run out of adventures to be had with learning sql!

further learning

- refresher: https://www.youtube.com/watch?v=kbKty5ZVKMY
- pandas experts note: https://www.youtube.com/watch?v=fmrmwFPMMaM
- more discussion: https://www.youtube.com/watch?v=OV6Mh2Jl9zQ
- deeper learning: https://app.datacamp.com/learn/career-tracks/data-analyst-in-sql



there are endless resources out there to help you on your journey. the most valuable method is to try things out, so go ahead and attempt something with sql, and if you make mistakes you will learn tons from that. good luck.