from zero to query

a sql primer

oskar 2023-02-16

sql - a fundamental tool for the data professional

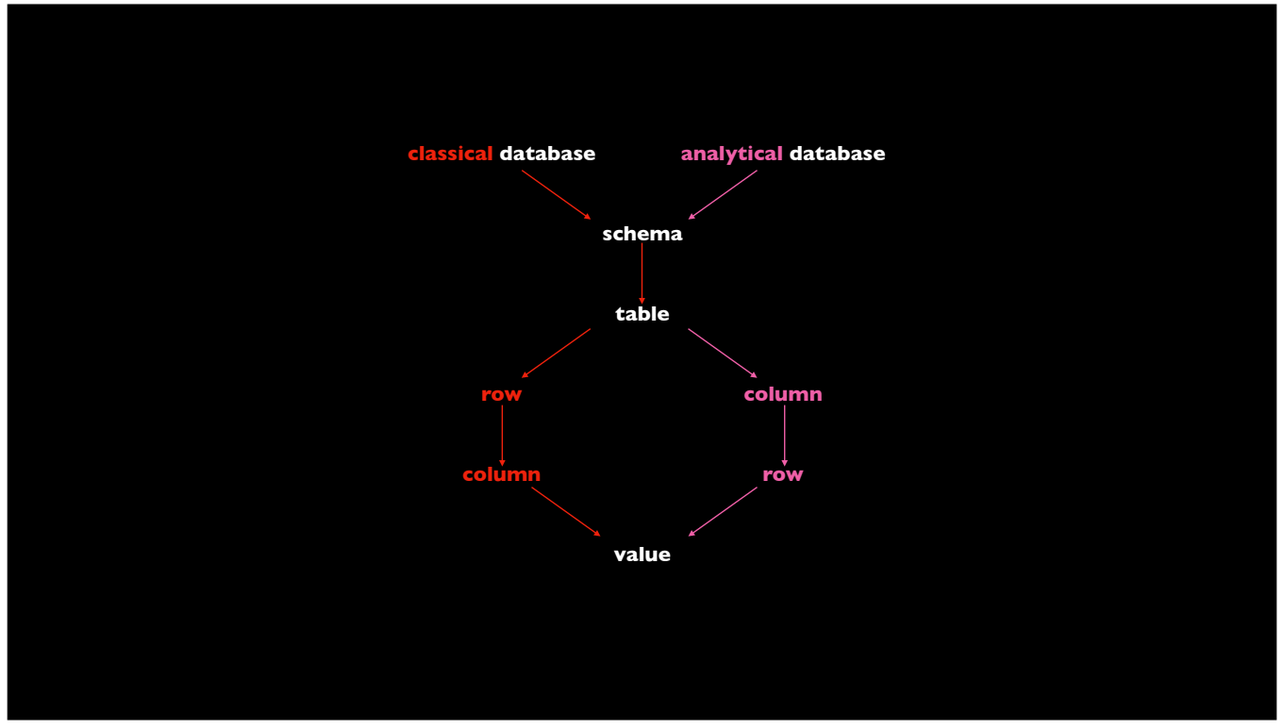
- database management
- data pipeline engineering
- data modeling
- data designing
- big data (parallel, distributed)
- data querying
- data analytics

the first thing to know about sql is that it is fundamental for various data professionals, whether they are for instance database managers, data pipeline engineers, schema designers, data reporters, data analysts. it is the ubiquitous, default language used to interact with any big data tool.

sql - a fundamental tool for the data professional

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

today you will learn about the latter two aspects, querying and analytical computations.



sql’s natural environment is databases and rectangular tables are the central concept of databases.
a database contain schemas, and each schema comprises of a set of tables.
- in a classical database, tables are comprised of rows and each row is a list of columns, each column stores a value.
- in analytical databases, tables are comprised of columns, each column is a list of rows, each row stores a value.

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				

in reality, a production database is comprised of a lot more components than this, there are users, user groups, access controls, procedures, functions, schedulers etc, etc. sql is a 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				

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

a note on sqlite

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

we will be using sqlite to access the training database with. not much to say about it: sqlite is a the minimal, **the simplest** application that queries 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 the minimum commands for sqlite. note that these are not sql commands, they operate on the sqlite environment and help you manage the database (as opposed to define it, or query it, or perform analytical computations on it).

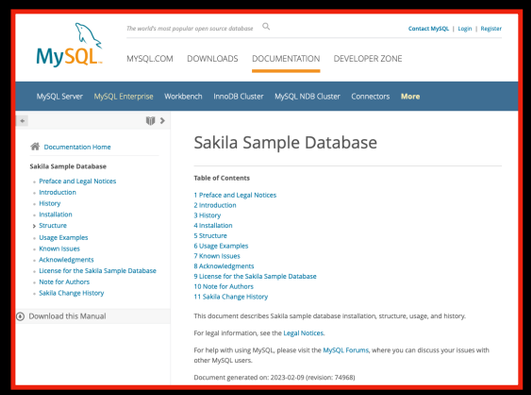
sql commands

- these run on the database
- they end with a ‘;’
- you can add comments with ‘-- a comment’
- they operate on the data tables
- example:
  - SELECT column1, column2 FROM table; -- a+b

now. a note about sql commands. they always end with a semi colon. 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



next, a note on the training data. this is a classic database made for training purposes. if you are familiar with r or python you will know these classic datasets: the titanic dataset, iris, mtcars, penguins, etc. in sql there are a few of those too. this one is called sakila

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

.header ON
.mode qbox
.show

.tables
```

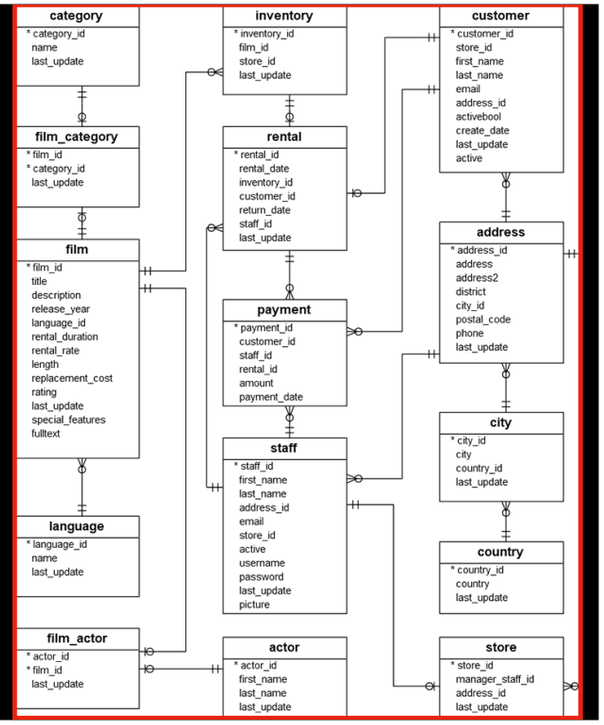
.tables

```
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>
```

in sqlite the command to list the available tables isn’t a data query but rather a sqlite command, so it begins with a period. try it now!

the sakila training data

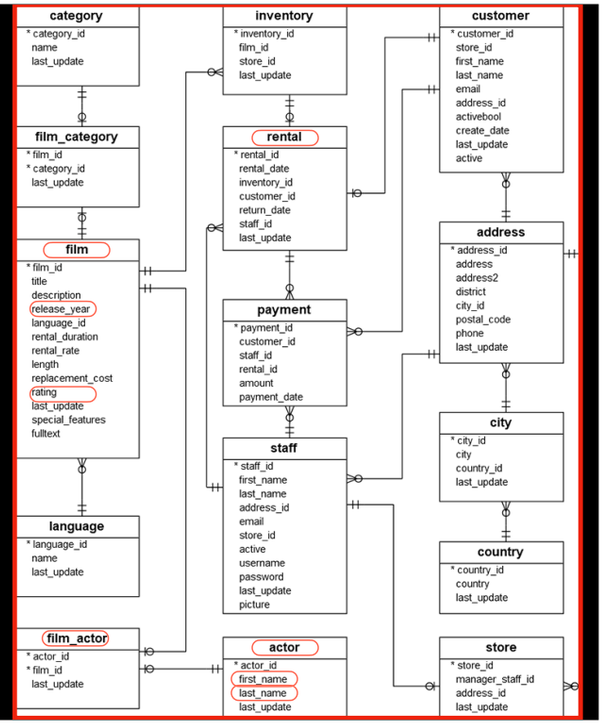
- classic, fictional data
- dvd rental company
- 20 relational tables:
  - normalised: no repetition
  - stores
  - inventory
  - films
  - film casting
  - actors
  - film ratings



next, a note on the training data. this is a classic database made for training purposes. if you are familiar with r or python you will know these classic datasets: the titanic dataset, iris, mtcars, penguins, etc. in sql there are a few of those too. this one is called sakila

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 question: you are running a chain of dvd rental stores, and you are deciding how much stock of upcoming film to procure. you have no data on the popularity of the new films since they are not out yet, but you know their cast. and you have noticed that the dvds that make you the most money tend to be the ones with popular actors. some actors are rented 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 data, but in order to conjure it out requires a very specific query. the rest of today is about generating the query that brin

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

here is the question and the list of sql *components* required to answer the question. they are presented in a pedagocial order, from the most basic, to the more complex. 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};
- + LIMIT num
- + WHERE {a_condition}
- + ORDER BY {columns}
- + INNER JOIN {table_2} ON {col1}={col2}
- + GROUP BY {columns}
- + HAVING {a_condition}
```

now that we have seen what tables are available in our database, we next start asking ourselves what data these tables contain. and here is where the venerable select statement comes in. 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 latter, we can use the asterisk wildcard (indicating all available columns) a la "SELECT * FROM a_tablename;" the output of a select statement is a table of results.

SELECT … FROM …;

```
- SELECT * FROM {table};
  -- returns all columns and all rows from {table}

- SELECT col2, col1 FROM example_table;
  -- returns columns ‘col2’ and ‘col1’ (in that order) from example_table

- SELECT t.col2, t.col1 FROM table_name t;
  -- introduces an alias for table_name, refers to its columns, ‘col2’, ‘col1’

- SELECT price + tax AS total_cost FROM sales;
  -- returns a single column, sum of price+tax, calls the output ‘total_cost’
```

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. — comments your code
3. you can alias results and table names

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

some of our tables have 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. we can use a limit clause within our sql statement to achieve that.

SELECT … FROM … LIMIT …;

```
- SELECT * FROM {table} LIMIT {n};
  -- returns {n{ unspecified rows of all columns from {table}

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

- SELECT sale_date, sale_cost FROM sales LIMIT 15;
  -- returns 15 unspecified rows of two columns from sales table

- SELECT id AS region_id, name AS region_name FROM regions LIMIT 10;
  -- returns region id and region name for 10 unspecified rows
```

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. note that in many database systems

SELECT (aggregate function) FROM … ;

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

- SELECT SUM(s.sale_cost) AS total_sales FROM sales s;
  -- returns the sum of the sale cost column from the sales table

- SELECT AVG(s.sale_cost) AS average_sales FROM sales s;
  -- returns the average of the sale cost column from sales

- SELECT MAX(s.sale_cost) AS highest_value_sale FROM sales s;
  -- returns the highest value sale from sales

- SELECT MIN(s.sale_date) AS earliest_sale FROM sales s;
  -- returns the date of the earliest sale from sales
```

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.

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

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 is by adding a WHERE clause, in the SELECT statement, *right after* the FROM clause.

SELECT … FROM … WHERE … [LIMIT n];

```
- SELECT * FROM {table} WHERE {column}={expression};
  -- returns only rows where the value in {column} equals {expression}

- SELECT * FROM table_name WHERE column1<>{expression};
  -- returns only rows where the value in column1 is not {expression}

- SELECT name AS item_name FROM items WHERE item_price>=10;
  -- returns names of items whose price is greater than or equal to £10

- SELECT name FROM items WHERE item_price>=10 LIMIT 8;
  -- returns 8 of the items whose price is greater or equal to £10
```

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

comparison operators

operator 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.
3. IN, LIKE, and BETWEEN…AND are useful, special comparison operators.

SELECT … FROM … WHERE …;

```
- SELECT * FROM sales WHERE sale_date BETWEEN ‘2023-02-01’ AND ‘2023-02-04’;
  -- returns only sales occurring between feb 1st and feb 4th, inclusive

- SELECT * FROM sales WHERE region_id IN (14,56,43) ;
  -- returns only sales in regions with id 14, 56, or 43

- SELECT * FROM region WHERE region_name LIKE ‘%new%’;
  -- returns only regions whose name contains ‘new’
```

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

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

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 1th 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 items ORDER BY item_cost LIMIT 10;
  -- return only the top 10 least expensive items in the catalog

- SELECT * FROM items ORDER BY item_cost DESC LIMIT 10;
  -- return only the top 10 most expensive items in the catalog
```

try it!

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

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.

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 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

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

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

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.
likewise ‘aden’ is in yemen. but we don’t want to have to do the look 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	?	?

what we want is something like this: a single table which, for each city reports contains the city name and the name of the associated country.

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

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

so we add a `JOIN` to the `WHERE` clause

city-and-country

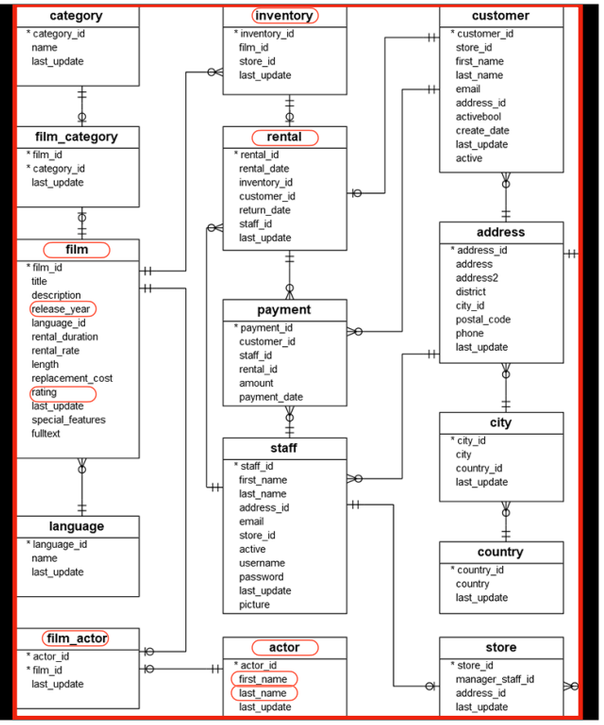
city_id	city	country
1	A Corua (La Corua)	Spain
2	Abha	Saudi Arabia
3	Abu Dhabi	United Arab Emirates
4	Acua	Mexico
5	Adana	Turkey
6	Addis Abeba	Ethiopia
7	Aden	Yemen
8	Adoni	India

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

it turns out that sql is brilliant at producing tables like that. the syntax magic to make something like this work all happens in the FROM clause. they all work similarly: you have FROM table_a JOIN table_b ON table_a.key_column=table_b.key_column…

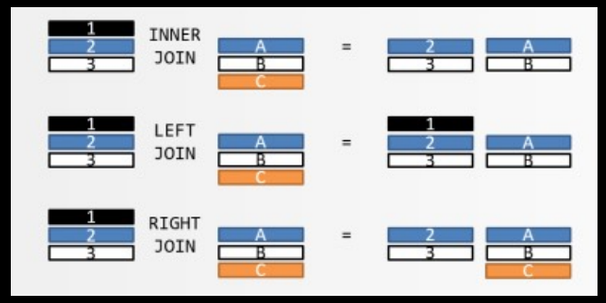
today's objective:

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



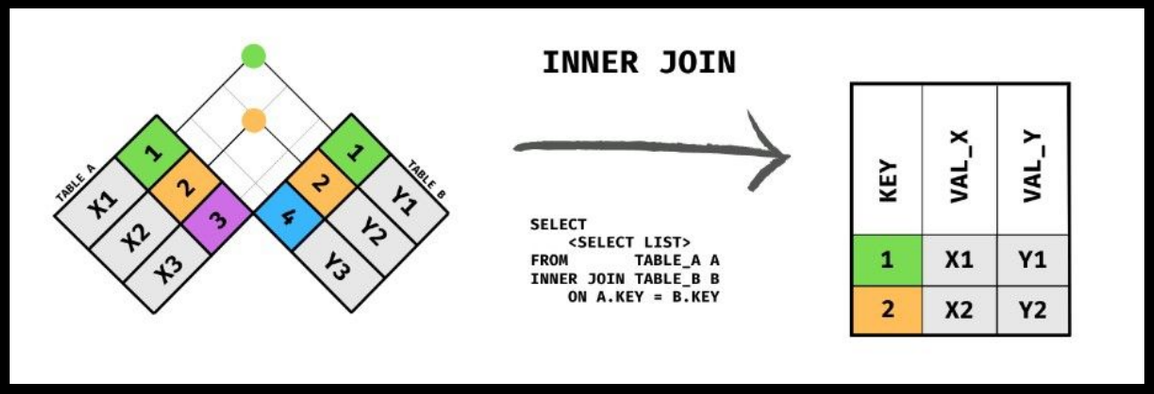
to answer the question at hand we are going to have to join the tables shown here: the actor table, the film_actor, the film table the inventory, the rental

FROM a <type> JOIN b ON a.col=b.col



there are multiple kinds of join. for a properly set up relational database, like we have, the inner join is the most useful, followed by a left join. right joins are rarely if ever needed. there are others, but we will focus on these.

FROM a INNER JOIN b



an INNER JOIN between 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. if there were 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.

SELECT … FROM a INNER JOIN b ON …;

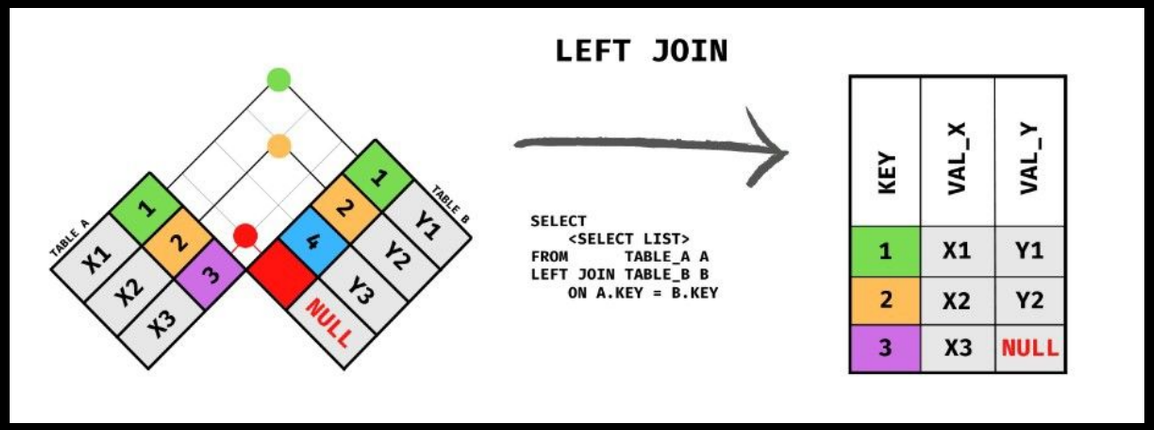
```
- SELECT a.city, b.country
  FROM city a
    INNER JOIN country b ON a.country_id=b.country_id
  ; -- output a table with city-country names

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

-
```

try it!

LEFT OUTER JOIN



a very common join is the left join, also known as the left outer join. 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 each row from table_a is guaranteed to be found in the output (whether or not it matches anything in table_b). the information from table_b (right hand table) won’t be included in the output unless they keys’ match.

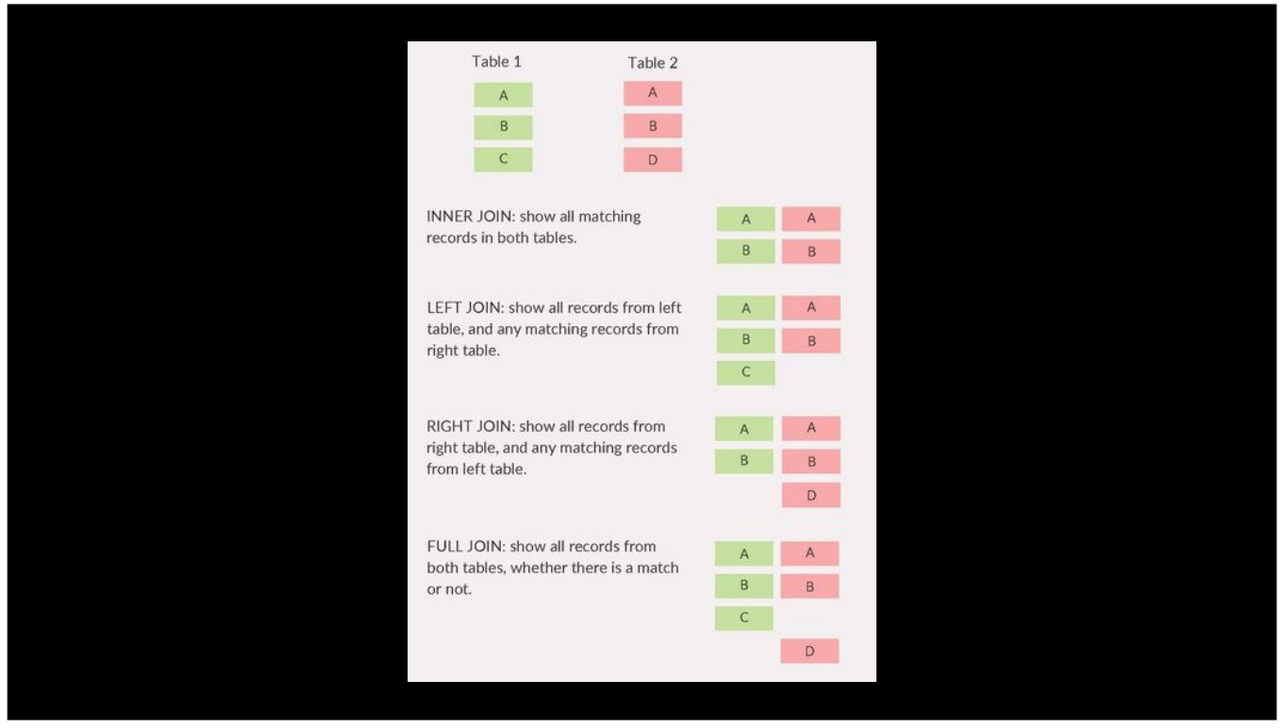


Table 1
A
B
C
Table 2
A
B
D
INNER JOIN: show all matching records in both tables.
A
B
A
B
LEFT JOIN: show all records from left table, and any matching records from right table.
A
B
C
A
B
RIGHT JOIN: show all records from right table, and any matching records from left table.
A
B
A
B
D
FULL JOIN: show all records from both tables, whether there is a match or not.
A
B
C
A
B
D

how can i aggregate select 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};
- + LIMIT num
- + WHERE {a_condition}
- + ORDER BY {columns}
- + INNER JOIN {table_2} ON {col1}={col2}
- + GROUP BY {columns}
- + HAVING {a_condition}
```

we 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. but very often we want to treat groups of rows as separate segments and sum together only rows within the same segment?

SELECT {col}, … FROM … GROUP BY {col};

```
- SELECT region_id, COUNT(*) FROM sales GROUP BY region_id;
  -- return each region's number of records from the sales table

- SELECT region_id, AVG(item_price) FROM items GROUP BY item_type;
  -- return the average price of items of each type from the items table

- SELECT item_type, MAX(item_price) FROM items GROUP BY item_type;
  -- returns the price of the priciest item of each type from the item table
```

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 the distinct value, you also get a

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

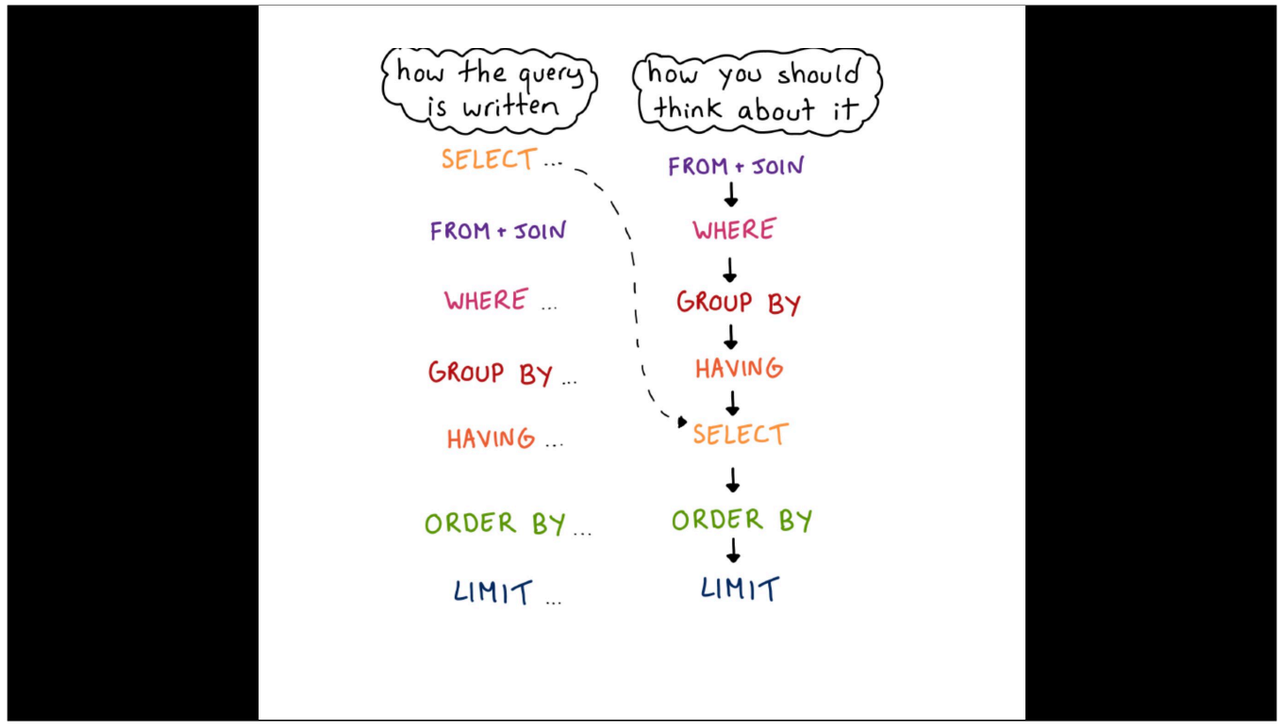
when we want to aggregate over specific groups of rows, but are only interested in some of the outcomes we can filter on the aggregated rows (one row per segment

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

excellent. now we have covered **all the components that we need** to answer the question. the dql component of sql really is this simple. all that remains is combining them 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 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.

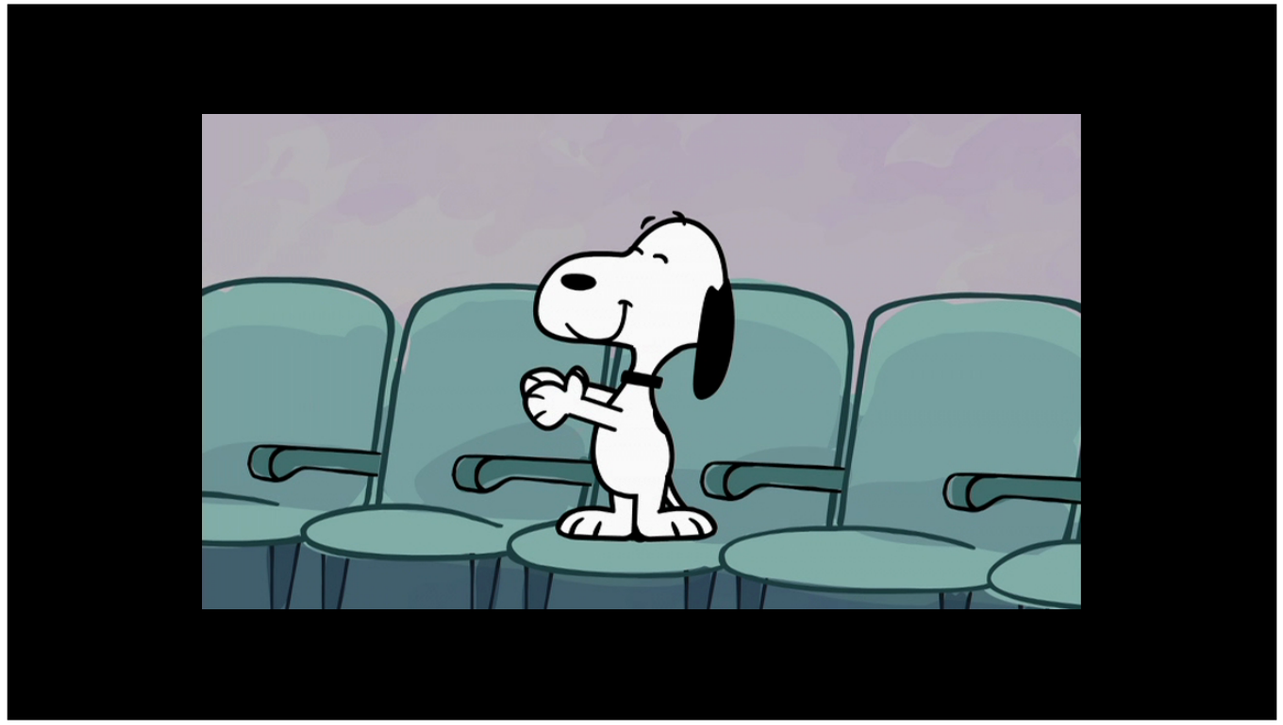
that's simple! however, sql demands we arrange the components in the slightly different order as shown on the left hand column. the steps are all conceptually the same, and the output is the same, it is just the syntax that requires the SELECT clause in front of all the rest.

hint: structure of the solution

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

we have 15 more minutes. if you are finding the assignment hard, take a look at this structure. this is the structure of the query you must make. you just need to compose the expressions in place of the curly brackets.

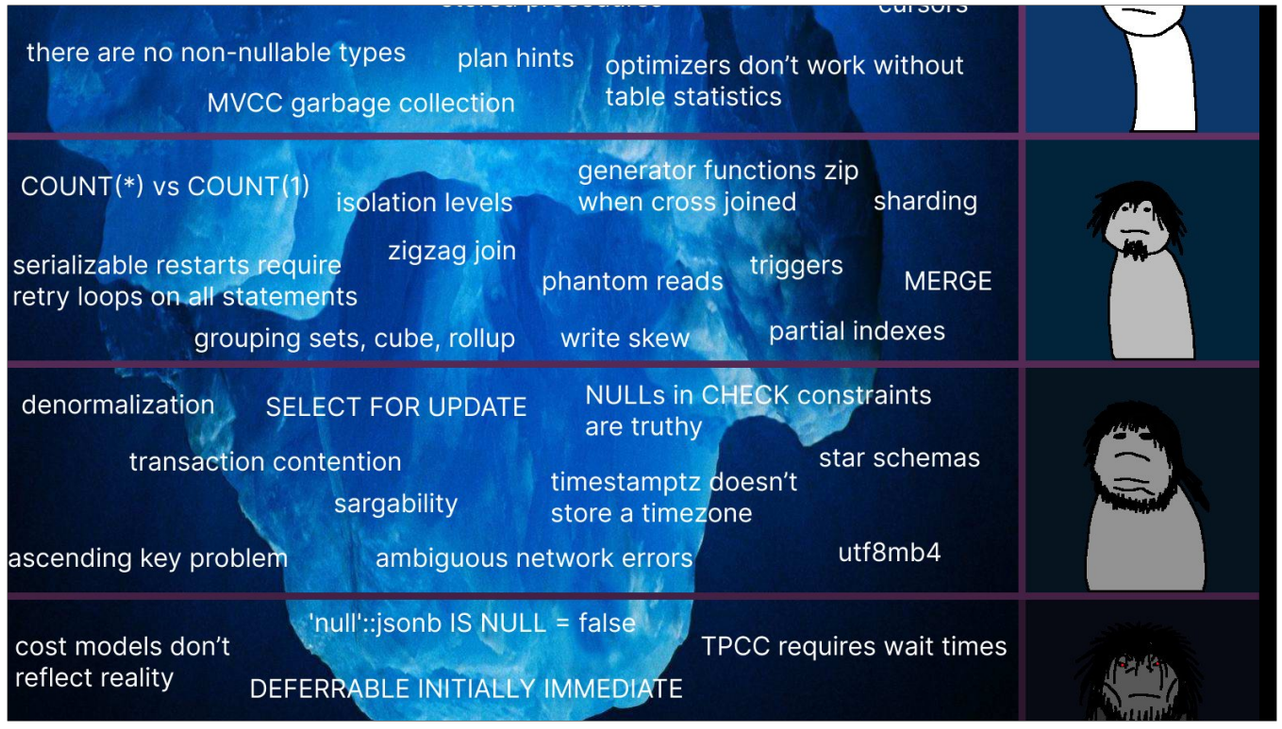
we hope that this workshop has sparked your curiosity about sql.
if you keep digging you will find that sql has plenty more to explore.



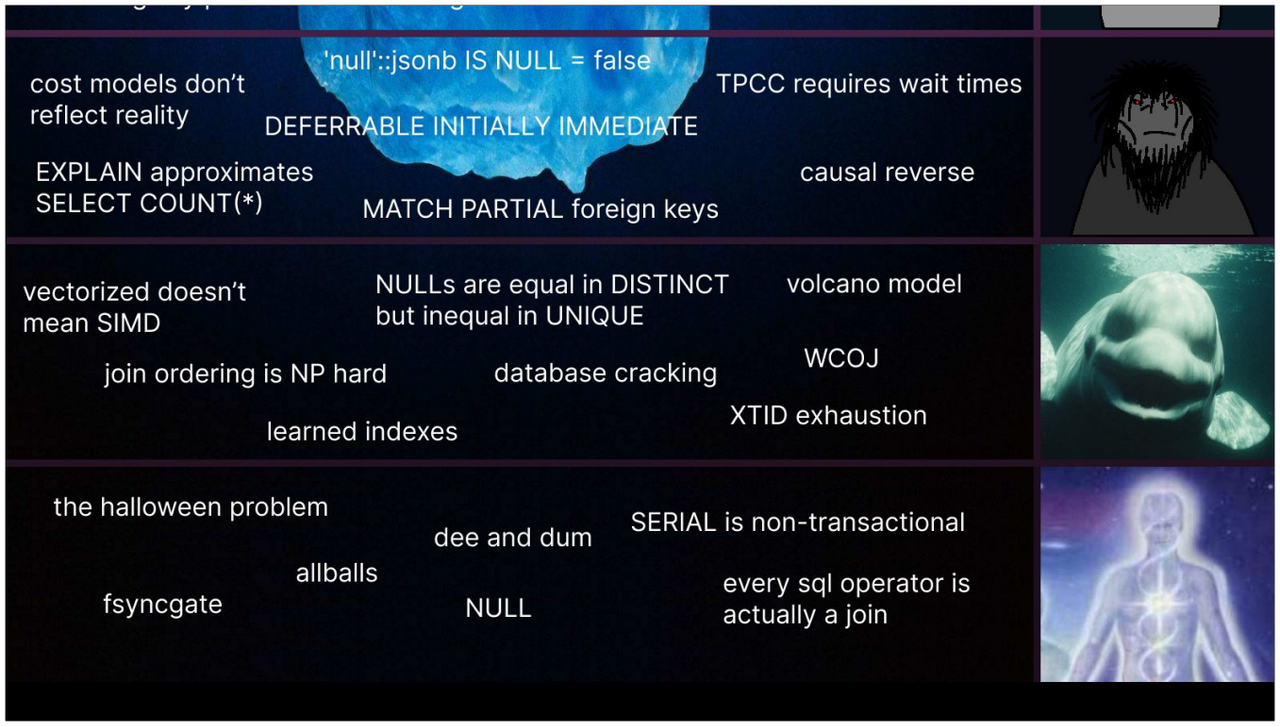
congratulations! give yourselves a round of applause. you have learned a lot today:
we covered how to query a database for quite specific information, from multiple tables.

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.



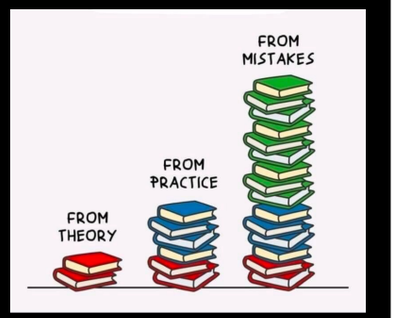
beyond that, you will find a lot of sql's power, usefulness, and ubiquity stems from how it handles the complexities of database management,



i have no idea what most of these are. but i suspect that you will never run out of adventures to be had with 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
- two week free course online starting 2023-02-20: https://corise.com/course/sql-crash-course



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.