Julia Evans
@b0rk

row-oriented vs column-oriented

Some SQL databases are row-oriented and some are column-oriented (aka "columnar")

row-oriented

a row's values are stored together on disk

Examples:

PostgreSQL, MySQL, SQLite, SQL Server

Mainly used for:

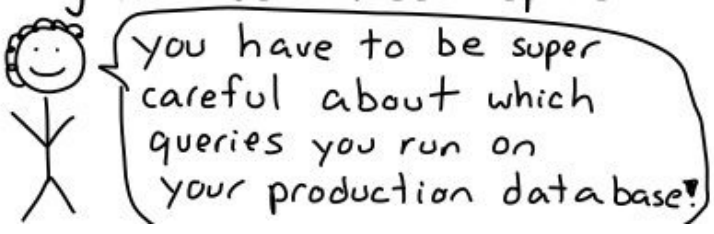
records that users need to look up/change all the time (eg a web service)

Pros:

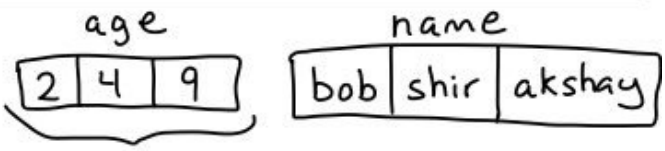
→ easy to look up or update a single row (it's in one place!)

Cons:

→ big expensive queries make your website less responsive



column-oriented



a column's values are stored together

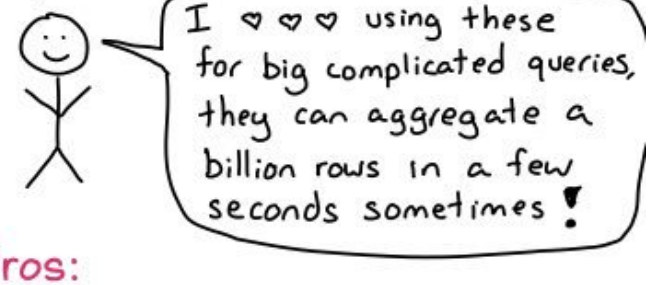
Examples:

presto is open source, the others aren't

Redshift, Presto, BigQuery, Vertica, Snowflake

Mainly used for:

data analysis ("run this huge query on a billion rows")



Pros:

→ querying all data in a column to do analysis is way faster

→ usually distribute data across many machines so 1000 machines can run your query

Cons:

→ SELECT * can be SUPER SLOW if you have 100 columns (avoid doing it!)

→ Updating a row is slow (do batch imports instead)

Julia Evans
@b0rk

ways to count

Here are three ways to count rows:

① COUNT(*): count all rows

This counts every row, regardless of the values in the row. Often used with a GROUP BY to get common values, like in this "top 50 baby names" query:

```
SELECT name, COUNT(*)
FROM baby_names
GROUP BY name
ORDER BY COUNT(*) DESC
LIMIT 50
```

② COUNT(DISTINCT column): get the number of distinct values

Really useful when a column has duplicate values.
For example, this query finds out how many species every animal genus has:

```
SELECT genus, COUNT(DISTINCT species)
FROM all_animals
GROUP BY 1 ORDER BY 2 DESC
```

③ SUM(CASE WHEN expression THEN 1 ELSE 0 END)

Want to know many dogs are named 'boxer'? You can use SUM and CASE WHEN to count them!

I like to put commas at the start for big queries

```
SELECT owner
, SUM(CASE WHEN type = 'dog' then 1 else 0 end) AS num_dogs
, SUM(CASE WHEN type = 'cat' then 1 else 0 end) AS num_cats
, SUM(CASE WHEN type NOT IN ('dog', 'cat) then 1 else 0
end) AS num_other
FROM pets GROUP BY 1
```

owner	type
1	dog
1	cat
2	dog
2	parakeet

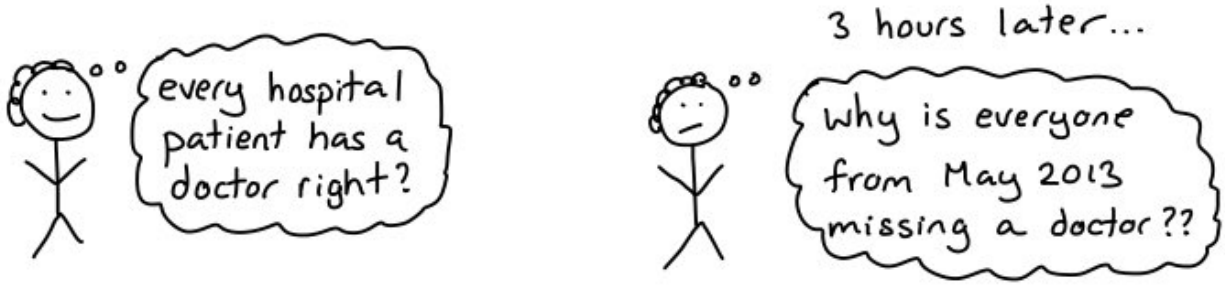
⇒

owner	num_dogs	num_cats	num_other
1	1	1	0
2	1	0	1

Julia Evans
@b0rk

questions to ask about your data

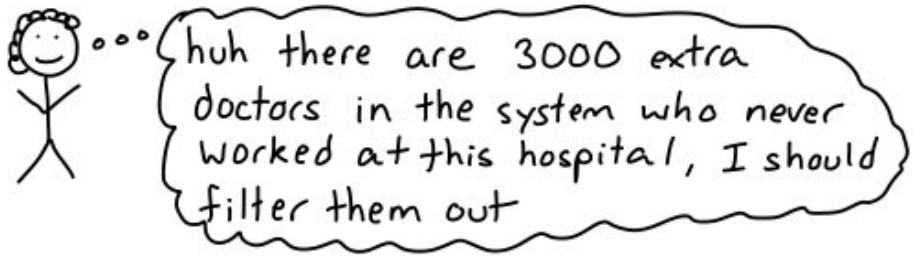
It's really easy to make incorrect assumptions about the data in a table:



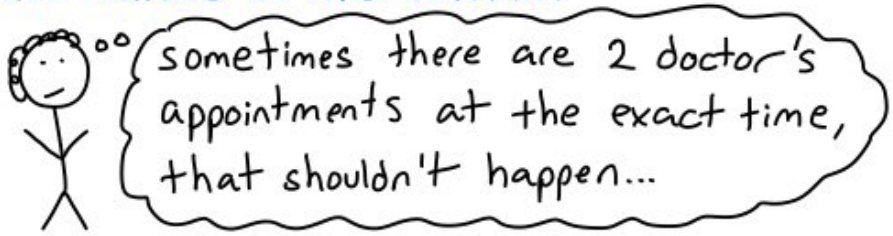
Some questions you might want to ask:

Does this column have NULL or 0 or "" values?

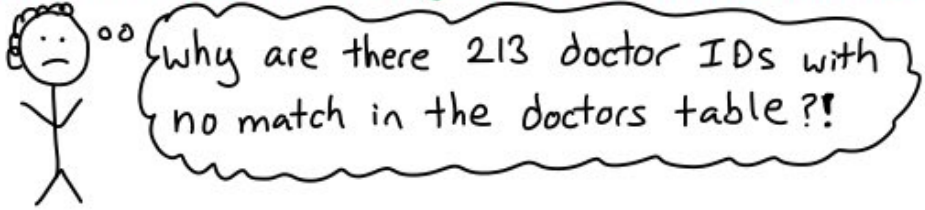
How many different values does this column have?



Are there duplicate values in this column?



Does the id column in table A always have a match in table B?



Julia Evans
@b0rk

SQL query steps

When this SQL query runs, here's how I think of what happens: every line in the query changes a table into another table

```
5  SELECT owner, count(*)
1  FROM cats
2  WHERE owner != 3
3  GROUP BY owner
4  HAVING count(*) = 2
6  ORDER BY owner DESC
```

① FROM cats

owner	name
1	libra
2	cinnamon
2	chanceuse
3	astra
4	lime
4	nikola

② WHERE owner != 3

owner	name
1	libra
2	cinnamon
2	chanceuse
~~3~~	~~astra~~ ← filter out this row
4	lime
4	nikola

③ GROUP BY owner

owner	name
1	libra
2	cinnamon
2	chanceuse
4	lime
4	nikola

④ HAVING count(*) = 2

owner	name
2	cinnamon
2	chanceuse
4	lime
4	nikola

⑤ SELECT owner, count(*)

owner	count(*)
→ 2	2
→ 4	2

⑥ ORDER BY owner DESC

	owner	count(*)
sort ↓	4	2
	2	2

Julia Evans
@b0rk

how LEFT JOIN works

Let's run this join:

```
cats LEFT JOIN people ON owner_id = id
```

Here are the 2 tables:

cats

owner_id	name
1	bella
2	luna
3	lime

people

id	name
1	juan
2	ahmed
4	ryan

① Combine every cats row with every people row

owner_id	cats.name	id	people.name
1	bella	1	juan
1	bella	2	ahmed
1	bella	4	ryan
2	luna	1	juan
2	luna	2	ahmed
2	luna	4	ryan
3	lime	1	juan
3	lime	2	ahmed
3	lime	4	ryan

same

② Find rows where the ON condition is true

owner_id = id

owner_id	cats.name	id	people.name
1	bella	1	juan
2	luna	2	ahmed

③ Add any missing rows from the left table (cats)

cats owner_id	cats.name	people id	people.name
1	bella	1	juan
2	luna	2	ahmed
3	lime	NULL	NULL

lime was missing so we add him back & put NULLs for the people columns

Find duplicate emails with HAVING

This query finds duplicate email addresses in a clients table:

```
SELECT email, count(*), group_concat(name, ',') AS names
FROM clients
GROUP BY email
HAVING count(*) > 1
```

Here's how it breaks down:

① FROM clients

id	name	email
1	mr darcy	darcy@pemberley.com
2	luna	luna@mice.com
3	nala	me@cartoon.com
4	tigger	me@cartoon.com

② GROUP BY email

id	name	email
1	mr darcy	darcy@pemberley.com

id	name	email
2	luna	luna@mice.com

id	name	email
3	nala	me@cartoon.com
4	tigger	me@cartoon.com

③ SELECT email, count(*), group_concat(name, ',') AS names

email	count(*)	names
darcy@pemberley.com	1	mr darcy
luna@mice.com	1	luna
me@cartoon.com	2	nala,tigger

④ HAVING count(*) > 1

email	count(*)	names
me@cartoon.com	2	nala,tigger

A simple LEFT JOIN

Here's a join query to figure out which treats Luna bought

```
SELECT clients.name AS client_name, sales.item
FROM sales
LEFT JOIN clients ON sales.client_id = clients.id
WHERE clients.name = 'luna'
```

Let's go through that query one step at a time

① FROM sales

client_id	item
1	catnip
1	blanket
1	tuna
2	tuna
5	laser pointer

② LEFT JOIN clients

Here's the clients table:

id	name	email
1	mr darcy	darcy@pemberley.com
2	luna	luna@mice.com
3	nala	me@cartoon.com
4	tigger	me@cartoon.com

③ LEFT JOIN clients ON sales.client_id = clients.id

Sales data on the left, clients on the right

client_id	item	id	name	email
1	catnip	1	mr darcy	darcy@pemberley.com
1	blanket	1	mr darcy	darcy@pemberley.com
1	tuna	1	mr darcy	darcy@pemberley.com
2	tuna	2	luna	luna@mice.com
5	laser pointer	NULL	NULL	NULL

④ WHERE clients.name = 'luna'

client_id	item	id	name	email
2	tuna	2	luna	luna@mice.com

⑤ SELECT cats.name AS

Get the time between thunderstorms with LAG()

Window functions let you reference values in other rows, like the previous row! This means you can subtract the day in the previous row to get the time between thunderstorms.

```
SELECT type, day, day - lag(day) OVER(PARTITION BY type
ORDER BY day ASC) as days_since_prev
FROM weather
ORDER BY day ASC
```

Let's go through that query one step at a time:

① FROM weather

type	day
rain	9
thunderstorm	11
rain	13
rain	21
thunderstorm	22
rain	30
thunderstorm	36
rain	38
thunderstorm	41
rain	48

② PARTITION BY type

type	day
rain	9
rain	13
rain	21
rain	30
rain	38
rain	48

type	day
thunderstorm	11
thunderstorm	22
thunderstorm	36
thunderstorm	41

③ ORDER BY day ASC

In this case the rows already look ordered, but you should always use an ORDER BY if you expect a specific order

type	day
rain	9
rain	13
rain	21
rain	30
rain	38
rain	48

type	day
thunderstorm	11
thunderstorm	22
thunderstorm	36
thunderstorm	41

④ SELECT type, day, day - lag(day) OVER(PARTITION BY type ORDER BY day ASC) as days_since_prev

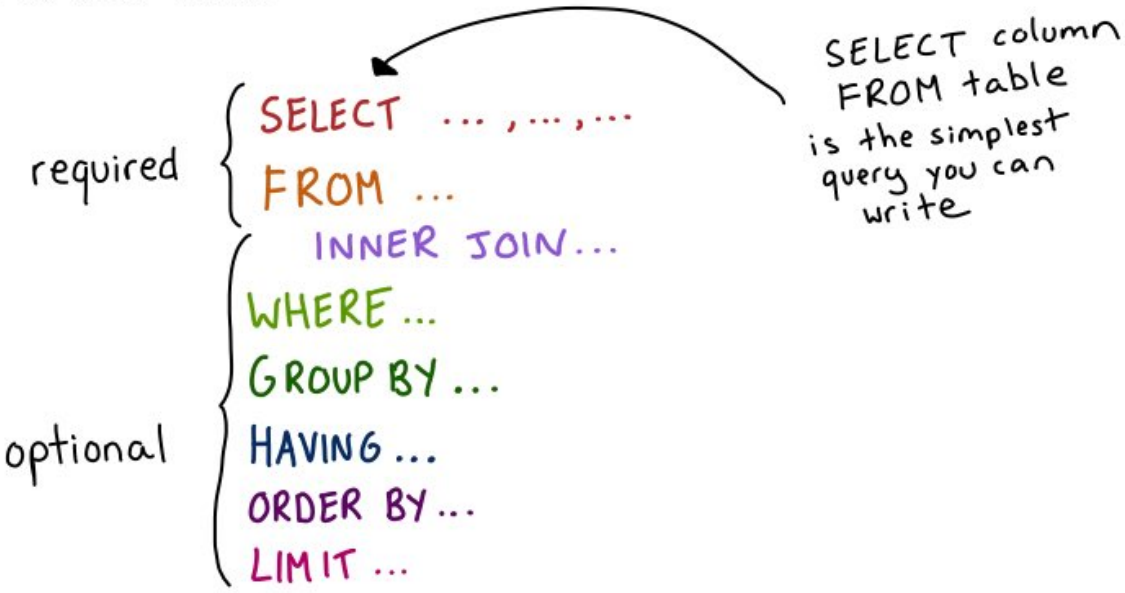
type	day	days_since_prev
rain	9	
rain	13	4
rain	21	8
rain	30	9
rain	38	8
rain	48	10
thunderstorm	11	
thunderstorm	22	11
thunderstorm	36	14
thunderstorm	41	5

Julia Evans
@b0rk

anatomy of a SQL query

Every SQL database contains a bunch of tables

Every SELECT query takes data from those tables and generates a new table



A few basic facts to start out:

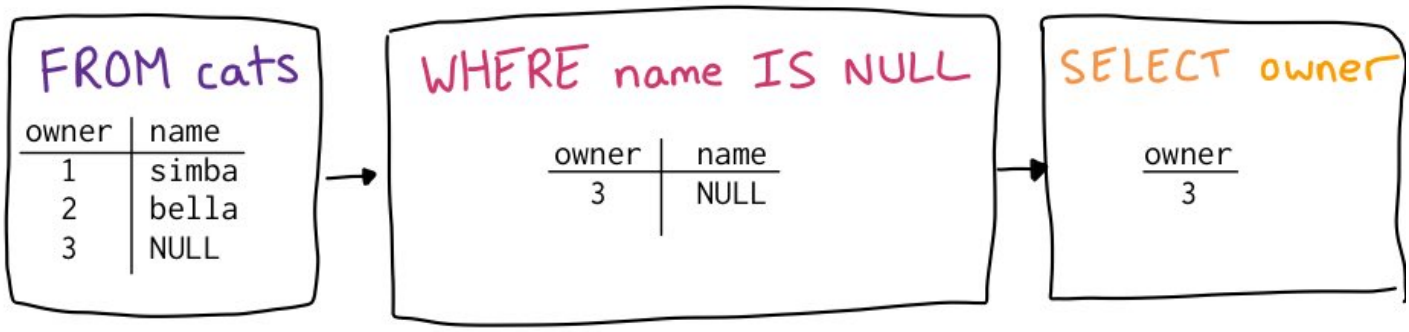
- You always need to use the order `SELECT ... FROM ... WHERE ... GROUP BY`
- SQL isn't case sensitive: `select * from table` is fine too

Julia Evans
@b0rk

WHERE

WHERE filters the table you start with.
For example, let's break down this simple query:

```
SELECT owner
FROM cats
WHERE name IS NULL
```



What you can put in a WHERE:

expr LIKE '...'

Check if a string contains a substring!

```
WHERE name LIKE '%darcy%'
```

% is a wildcard

= != < >=

These work the way you'd guess

```
WHERE revenue - costs >= 0
```

expr IN (...)

Check if an expression is in a list of values

```
WHERE name IN ('bella', 'simba')
```

expr IS NULL
expr IS NOT NULL

= NULL doesn't work, you need to use IS NULL

AND OR NOT

You can AND together as many conditions as you want

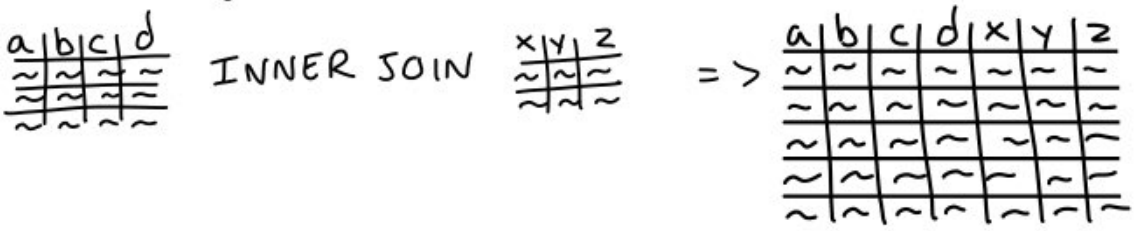

```
(....)
AND (.....)
AND (.....)
```

put all the ORs in the brackets

Julia Evans
@b0rk

rules for simple JOINs

Joins in SQL let you take 2 tables and combine them into one.



Joins can get really complicated, so we'll start with the simplest way to join. Here are the rules I use for 90% of my joins:

Rule 1: only use LEFT JOIN and INNER JOIN

This is every join type:

Rule 2: Only include 1 condition in your join

Here's the syntax for a join:

```
table1 LEFT JOIN table2 ON <arbitrary boolean condition>
```

I usually stick to a very simple condition, like this:

```
table1 LEFT JOIN table2
  ON table1.some_column = table2.other_column
```

Rule 3: One of the joined columns should be unique

If neither of the columns is unique, you'll get strange results like this:

Julia Evans
@b0rk

SELECT

`SELECT` is where you pick the final columns that appear in output of the query. Here's the syntax:

```
SELECT expression_1 [AS alias],
       expression_2 [AS alias2],
       ...
FROM ...
```

Some useful things to know about SELECT:

① You can combine many columns with SQL expressions

A few examples:

```
CONCAT(first_name, ' ', last_name)
MAX(last_year_profit, this_year_profit)
DATE_TRUNC('month', created)
```

← this is PostgreSQL syntax for rounding a date, other SQL dialects have different syntax

② Alias an expression with `AS`

`CONCAT(first_name, ' ', last_name)` is a mouthful! It's nice to give your complicated expressions an easy-to-read alias, like:

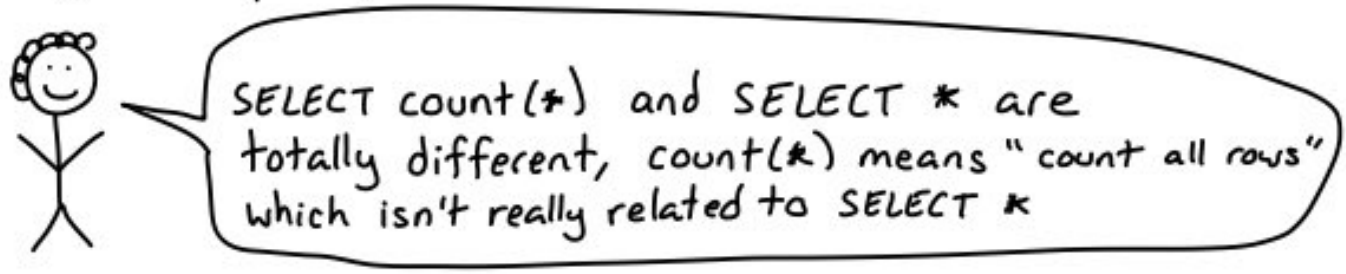
```
SELECT CONCAT(first_name, ' ', last_name) AS full_name
```

③ Select all columns with `SELECT *`

When I'm starting to figure out a query, I'll often write something like

```
SELECT * from some_table LIMIT 10
```

just to quickly see what the columns in the table look like



ORDER BY and LIMIT

Julia Evans
@b0rk

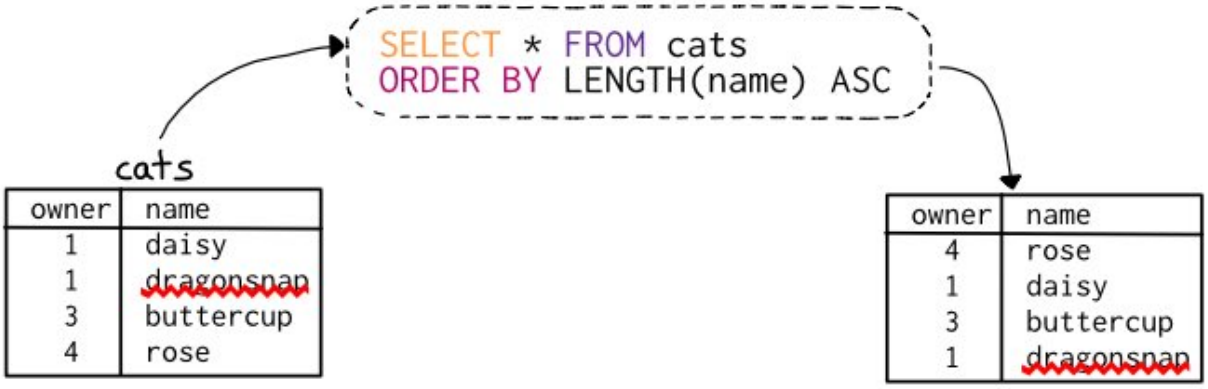
ORDER BY and LIMIT happen at the end and affect the final output of the query.

ORDER BY lets you sort by anything you want!
The syntax:

```
ORDER BY [expression] ASC or DESC
```

ASC ← stands for ascending

Example:



LIMIT lets you limit the number of rows output.
The syntax:

```
LIMIT 2
```

must be a number