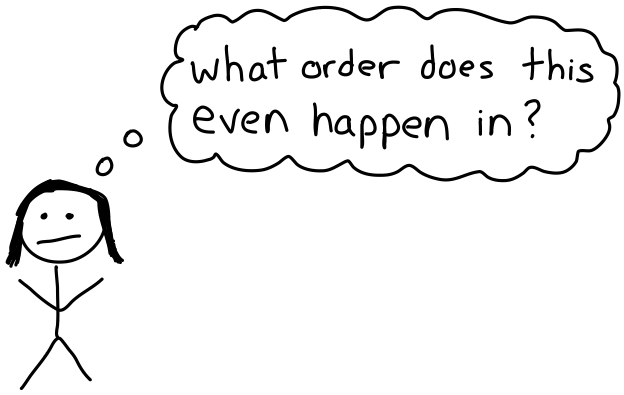
BECOME A SELECT STAR

BY JULIA EVANS

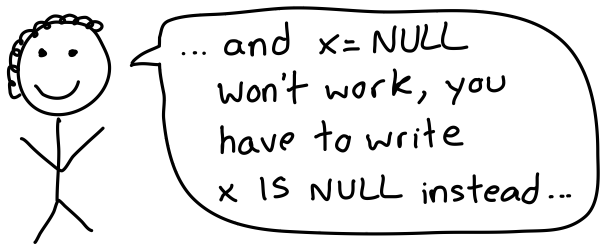
about this zine

SQL isn't like most programming languages.

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

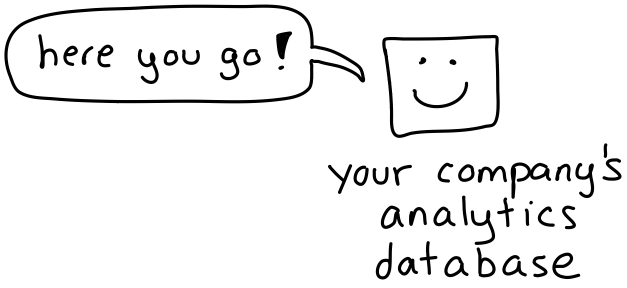


and it has quite a few weird gotchas:



but knowing it lets you use really powerful database software to answer questions FAST:

.. 10 seconds later...



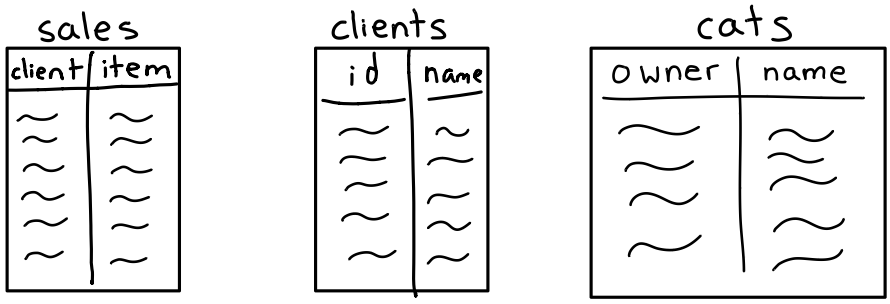
This zine will get you started with `SELECT` queries so you can get the answers to any question you want about your data. ♥
You can run any query from the zine here:

https://sql-playground.wizardzines.com

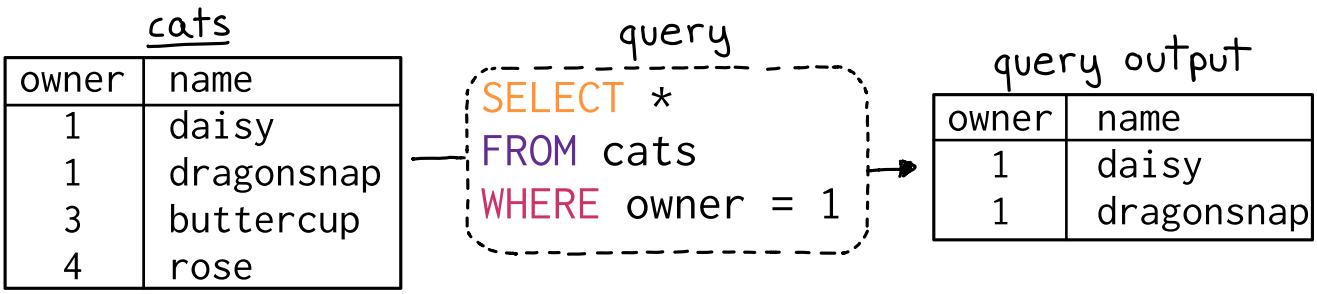
Table of Contents

getting started with SELECT

A SQL database contains a bunch of tables



Every SELECT query takes data from those tables and outputs a table of results.

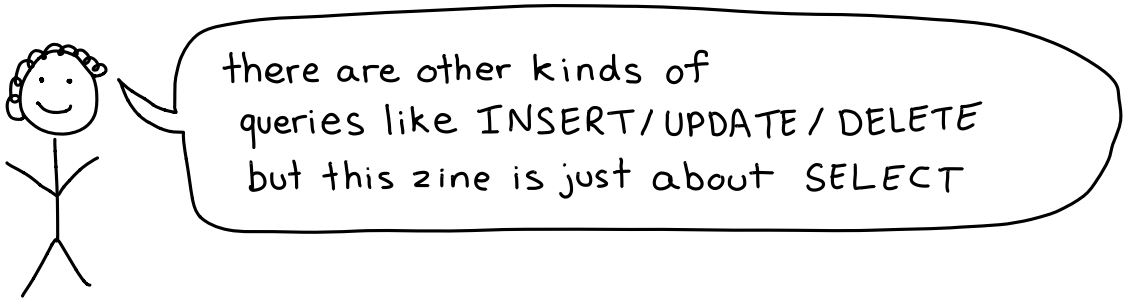


A few basic facts to start out:

→ SELECT queries have to be written in the order:

```
SELECT ... FROM ... WHERE ... GROUP BY .. HAVING ...
    ORDER BY ... LIMIT
```

→ SQL isn't case sensitive: `select * from table` is fine too. This zine will use ALL CAPS for SQL keywords like FROM.



SELECT queries start with FROM

Conceptually, every step (like "WHERE") of a query transforms its input, like this:

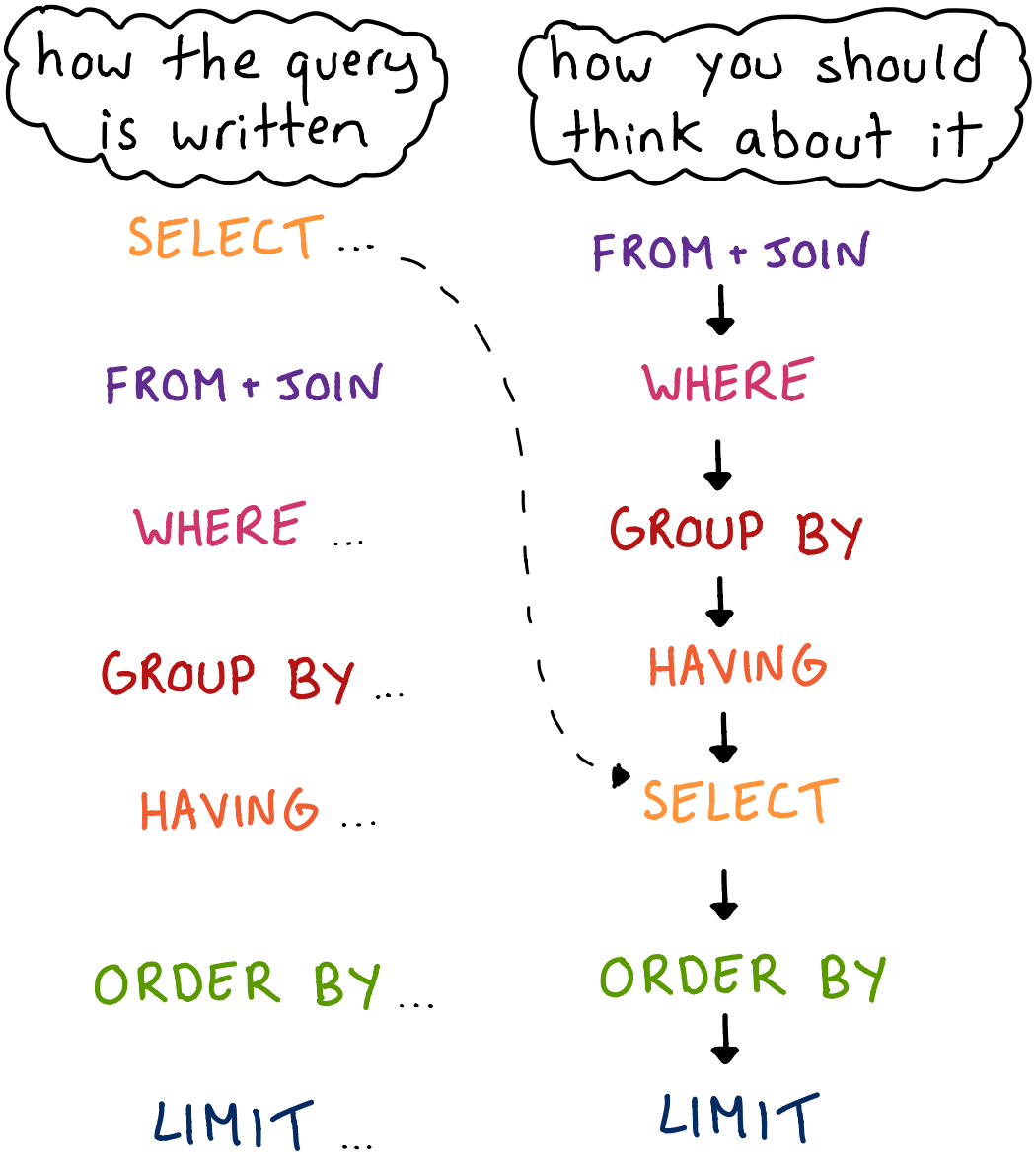
cats

owner	name
1	daisy
1	dragonsnap
3	buttercup
4	rose

WHERE owner = 1

owner	name
1	daisy
1	dragonsnap

The query's steps don't happen in the order they're written:



(In reality query execution is much more complicated than this. There are a lot of optimizations.)

SELECT

SELECT is where you pick the final columns that appear in the table the query outputs. Here's the syntax:

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

Some useful things you can do in a SELECT:

★ Combine many columns with SQL expressions

A few examples:

```
CONCAT(first_name, ' ', last_name)
DATE_TRUNC('month', created)
```

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

★ Alias an expression with AS

`||` is a concatenation operation

`first_name || ' ' || last_name` is a mouthful! If you alias an expression with AS, you can use the alias elsewhere in the query to refer to that expression.

```
SELECT first_name || ' ' || last_name AS full_name
FROM people
ORDER BY full_name DESC
```

refers to first_name || ' ' || last_name

★ Select all columns with SELECT *

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

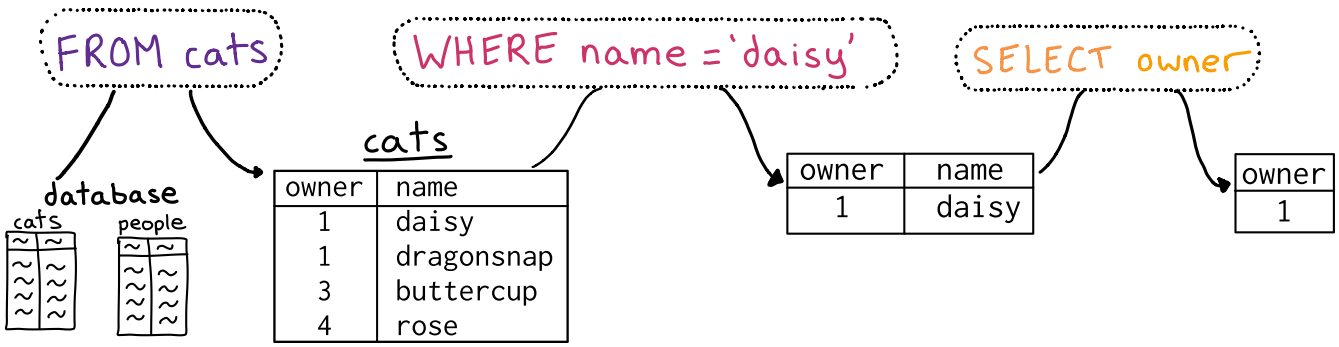
```
SELECT * FROM some_table LIMIT 10
```

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

WHERE

WHERE filters the table you start with. For example, let's break down this query that finds all owners with cats named "daisy".

```
SELECT owner
FROM cats
WHERE name = 'daisy'
```



What you can put in a WHERE:

expr LIKE '...'

Check if a string contains a substring!

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

% is a wildcard like * in your shell

= != < >=

These work the way you'd guess, except when NULL is involved.

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

expr IN (...)

Check if an expression is in a list of values

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

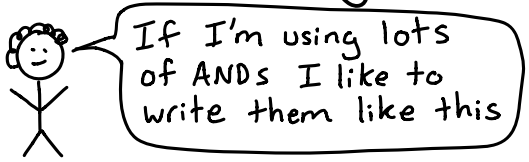
expr IS NULL
expr IS NOT NULL

more about NULL on pages 15-17

~~= NULL~~ IS NULL

AND OR NOT

You can AND together as many conditions as you want




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

put all the ORs in the parentheses

GROUP BY

GROUP BY combines multiple rows into one row. Here's how it works for this table & query:

```
SELECT item, COUNT(*), MAX(price)
FROM sales
GROUP BY item
```

aggregates

sales

item	price
catnip	5
laser	8
tuna	4
tuna	3

query output

item	COUNT(*)	MAX(price)
catnip	1	5
laser	1	8
tuna	2	4

① Split the table into groups for each value that you grouped by:

item='catnip'

item	price
catnip	5

item='laser'

item	price
laser	8

item='tuna'

item	price
tuna	4
tuna	3

② Calculate the aggregates from the query for each group:

item	price
catnip	5

COUNT(*) = 1

MAX(price) = 5

item	price
laser	8

COUNT(*) = 1

MAX(price) = 8

item	price
tuna	4
tuna	3

COUNT(*) = 2

MAX(price) = 4

③ Create a result set with 1 row for each group

item	COUNT(*)	MAX(price)
catnip	1	5
laser	1	8
tuna	2	4

HAVING

This query uses HAVING to find all emails that are shared by more than one user:

```
SELECT email, COUNT(*)
FROM users
GROUP BY email
HAVING COUNT(*) > 1
```

users

id	email
1	asdf@fake.com
2	bob@builder.com
3	asdf@fake.com

query output

email	COUNT(*)
asdf@fake.com	2

HAVING is like WHERE, but with 1 difference: HAVING filters rows AFTER grouping and WHERE filters rows BEFORE grouping.

Because of this, you can use aggregates (like COUNT(*)) in a HAVING clause but not with WHERE.

Here's another HAVING example that finds months with more than $6.00 in income:

```
SELECT month
FROM sales
GROUP BY month
HAVING SUM(price) > 6
```

sales

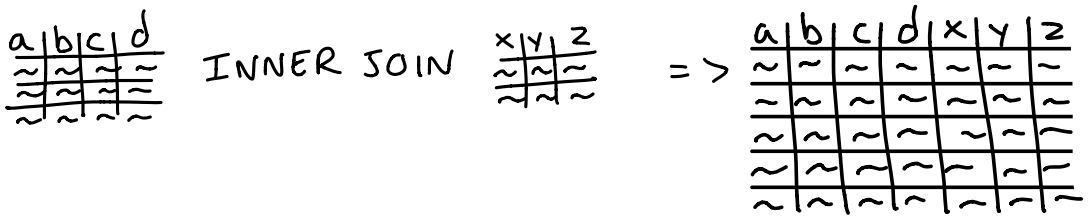
month	item	price
Jan	catnip	5
Feb	laser	8
March	food	4
March	food	3

query output

month
Feb
March

my 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`

There are other kinds of joins (`RIGHT JOIN`, `CROSS JOIN`, `FULL OUTER JOIN`), but 95% of the time I only use `LEFT JOIN` and `INNER JOIN`.

Rule 2: refer to columns as `table_name.column_name`

You can leave out the table name if there's just one column with that name, but it can get confusing

Rule 3: Only include 1 condition in your join

Here's the syntax for a `LEFT JOIN`:

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

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

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

Rule 4: One of the joined columns should have unique values

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

owners_bad

name	age
maher	16
maher	32
rishi	21

INNER JOIN

cats_bad

owner	name
maher	daisy
maher	dragonsnap
rishi	buttercup

```
owners_bad INNER JOIN cats_bad
  ON owners_bad.name = cats_bad.owner
```

=>

owner	name	age
maher	daisy	16
maher	dragonsnap	16
maher	daisy	32
maher	dragonsnap	32
rishi	buttercup	21

(these are "bad" versions of the owners and cats tables that don't JOIN well)

INNER JOIN and LEFT JOIN

Here are examples of how `INNER JOIN` and `LEFT JOIN` work:

`INNER JOIN` only includes rows that match the ON condition. This query combines the `cats` and `owners` tables:

```
SELECT *
FROM owners INNER JOIN cats
   ON cats.owner = owners.id
```

owners

id	name
1	maher
2	rishi
3	chandra

cats

owner	name
1	daisy
1	dragonsnap
3	buttercup
4	rose

these 2 rows don't have matches so they're not in the output.

query output

cats.owner	owners.name	cats.name
1	maher	daisy
1	maher	dragonsnap
3	chandra	buttercup

`LEFT JOIN` includes every row from the left table (`owners` in this example), even if it's not in the right table. Rows not in the right table will be set to `NULL`.

```
SELECT *
FROM owners LEFT JOIN cats
   ON cats.owner = owners.id
```

owners

id	name
1	maher
2	rishi
3	chandra

cats

owner	name
1	daisy
1	dragonsnap
3	buttercup
4	rose

query output

cats.owner	owners.name	cats.name
1	maher	daisy
1	maher	dragonsnap
2	rishi	NULL
3	chandra	buttercup

Rishi has no cats :(

This is a classic example of a join that follows my 4 guidelines from the previous page:

1) it's an `INNER JOIN` / `LEFT JOIN`
2) it includes the table name in `cats.owner` and `owners.id`
3) the condition `ON cats.owner = owners.id` is simple
4) it joins on a unique column (the `id` column in the owners table)

example : LEFT JOIN + GROUP BY

This query counts how many items every client bought (including clients who didn't buy anything):

```
SELECT name, COUNT(item) AS items_bought
FROM owners LEFT JOIN sales
                ON owners.id = sales.client
GROUP BY name
ORDER BY items_bought DESC
```

FROM owners LEFT JOIN sales...

owners

id	name
1	maher
2	rishi
3	chandra

sales

item	client
catnip	1
laser	1
tuna	1
tuna	2

ON owners.id = sales.client

id	name	item
1	maher	catnip
1	maher	laser
1	maher	tuna
2	rishi	tuna
3	chandra	NULL

GROUP BY name

id	name	item
1	maher	catnip
1	maher	laser
1	maher	tuna
2	rishi	tuna
3	chandra	NULL

SELECT name, COUNT(item) AS items_bought

name	items_bought
rishi	1
chandra	0
maher	3

COUNT(item) doesn't count NULLs

ORDER BY items_bought DESC

name	items_bought
maher	3
rishi	1
chandra	0

ORDER BY and LIMIT

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 is:

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

ASC stands for ascending

For example, this query sorts cats by the length of their name (shortest first):

```
SELECT * FROM cats
ORDER BY LENGTH(name) ASC
```

cats

owner	name
1	daisy
1	dragonsnap
3	buttercup
4	rose

owner	name
4	rose
1	daisy
3	buttercup
1	dragonsnap

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

```
LIMIT [integer]
```

For example, this is the same as the previous query, but it limits to only the 2 cats with the shortest names:

```
SELECT * FROM cats
ORDER BY LENGTH(name) ASC
LIMIT 2
```

cats

owner	name
1	daisy
1	dragonsnap
3	buttercup
4	rose

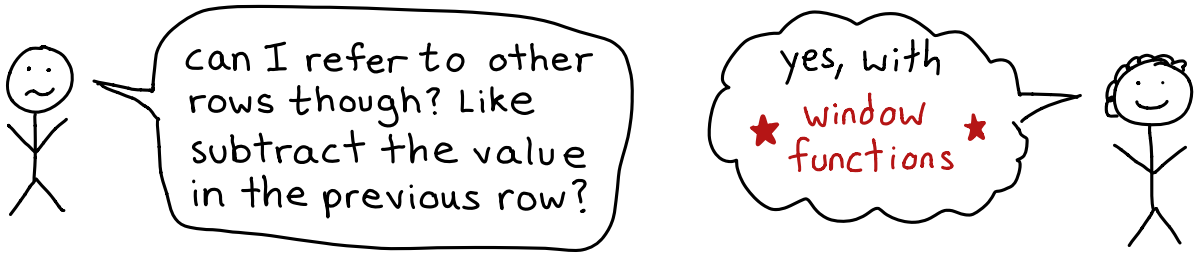
owner	name
4	rose
1	daisy

refer to other rows with ★ window functions ★

Let's talk about an advanced SQL feature: window functions! Normally SQL expressions only let you refer to information in a single row.

2 columns from the same row

```
SELECT CONCAT(firstname, ' ', lastname) as full_name
```



Window functions are SQL expressions that let you reference values in other rows. The syntax (explained on the next page!) is:

```
[expression] OVER ([window definition])
```

Example: use `LAG()` to find how long since the last sale

```
SELECT item,
       day - LAG(day) OVER (ORDER BY day)
FROM sales
```

sales

item	day
catnip	2
laser	40
tuna	70
tuna	72

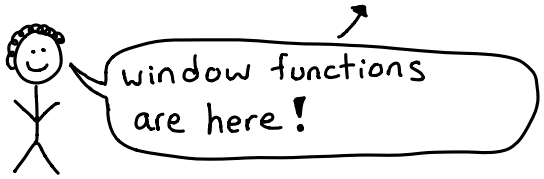
query output

item	day - LAG(day) OVER (ORDER BY day)
catnip	NULL
laser	38
tuna	30
tuna	2

2 - NULL
40 - 2
70 - 40
72 - 70

They're part of `SELECT`, so they happen after `HAVING`:

FROM + JOIN → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT



OVER() assigns every row a window

A "window" is a set of rows.

name	class	grade
juan	1	93
lucia	1	98

← a window!

A window can be as big as the whole table (an empty `OVER()` is the whole table!) or as small as just one row.

`OVER()` is confusing at first, so here's an example! Let's run this query that ranks students in each class by grade:

```
SELECT name, class, grade,
       ROW_NUMBER() OVER (PARTITION BY class
                          ORDER BY grade DESC)
        AS rank_in_class
FROM grades
```

Step 1: Assign every row a window. `OVER(PARTITION BY class)` means that there are 2 windows: one each for class 1 and 2

grades

name	class	grade
juan	1	93
lucia	1	98
raph	2	88
chen	2	90

name	class	grade
juan	1	93
lucia	1	98

name	class	grade
raph	2	88
chen	2	90

Step 2: Run the function. We need to run `ROW_NUMBER()` to find each row's rank in its window:

query output

name	class	grade	rank_in_class
juan	1	93	2
lucia	1	98	1
raph	2	88	2
chen	2	90	1

example: get the time between baby feedings

This query finds the time since a baby's last feeding/diaper change.

```
SELECT event, hour,
       hour - LAG(hour) OVER(PARTITION BY event ORDER BY hour ASC)
          AS time_since_last
FROM baby_log
WHERE event in ('feeding', 'diaper')
ORDER BY hour ASC
```

① FROM baby_log

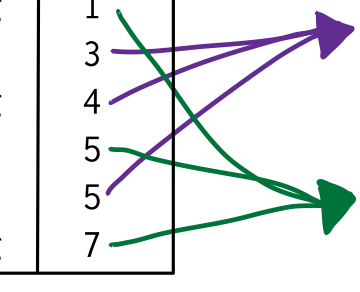
event	hour
feeding	1
cough	1
diaper	3
feeding	4
diaper	5
diaper	5
feeding	7
cough	7

② WHERE event IN ('diaper', 'feeding')

event	hour
feeding	1
diaper	3
feeding	4
diaper	5
diaper	5
feeding	7

③ OVER (PARTITION BY event ORDER BY hour ASC)

event	hour
feeding	1
diaper	3
feeding	4
diaper	5
diaper	5
feeding	7



event	hour
feeding	1
feeding	4
feeding	7

event	hour
diaper	3
diaper	5
diaper	5

this ORDER BY only affects the windows, not the query output

④ SELECT type, hour, hour - LAG(hour)

event	hour	time_since_last
feeding	1	NULL
feeding	4	3
feeding	7	3
diaper	3	NULL
diaper	5	2
diaper	5	0

LAG() is NULL for the first row in the window

⑤ ORDER BY hour ASC

event	hour	time_since_last
feeding	1	NULL
diaper	3	NULL
feeding	4	3
diaper	5	2
diaper	5	0
feeding	7	3

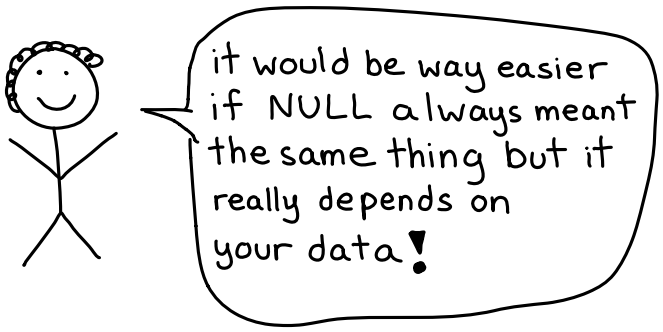
NULL: unknown or missing

NULL is a special state in SQL. It's very commonly used as a placeholder for missing data ("we don't know her address!")

What NULL means exactly depends on your data. For example, it's really important to know if `allergies IS NULL` means

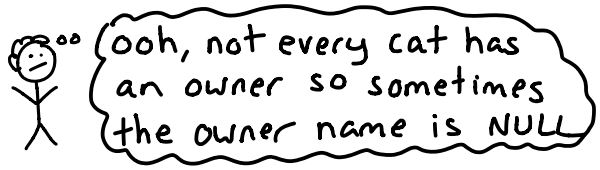
→ "no allergies" or
→ "we don't know if she has allergies or not"

NULL "should" mean "unknown" but it doesn't always



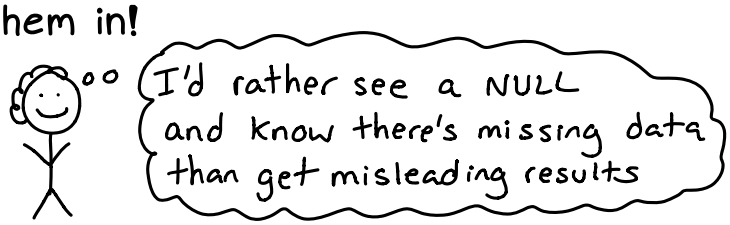
★ where NULLs come from ★

→ There were already NULL values in the table
→ The window function LAG() can return NULL
→ You did a LEFT JOIN and some of the rows on the left didn't have a match for the ON condition



★ ways to handle NULLs ★

→ Leave them in!



→ Filter them out!

```
... WHERE first_name IS NOT NULL ...
```

→ Use COALESCE or CASE to add a default value

NULL surprises

`NULL` isn't equal (or not equal!) to anything in SQL (`x = NULL` and `x != NULL` are never true for any `x`). This results in 2 behaviours that are surprising at first:

Surprise! `x = NULL` doesn't work

fish

name	owner
NULL	bob
nemp	ahmed

```
SELECT * FROM fish
WHERE name = NULL
```

name	owner

no results!

You need to use `x IS NULL` instead.

works

```
name IS NULL
name IS NOT NULL
```

doesn't work

```
name = NULL
name != NULL
```

surprise! `name != 'betty'` doesn't match NULLs

fish

name	owner
NULL	bob
nemo	ahmed

```
SELECT * FROM fish
WHERE name != 'betty'
```

name	owner
nemo	ahmed

To match `NULL`s as well, I'll often write something like `WHERE name != 'betty' OR name IS NULL` instead.

more surprising truths

More operations with `NULL` which might be surprising:

```
2 + NULL              => NULL
NULL * 10             => NULL
CONCAT('hi', NULL)    => NULL
```

NULL isn't even equal to itself!

```
NULL = NULL    =>  NULL
2 = NULL       =>  NULL
2 != NULL      =>  NULL
```

handle NULLs with COALESCE

COALESCE is a function that returns the first argument you give it that isn't NULL

```
COALESCE(NULL, 1, 2) => 1
COALESCE(NULL, NULL, NULL) => NULL
COALESCE(4, NULL, 2) => 4
```

2 ways you might want to use COALESCE in practice:

① Set a default value

In this table, a NULL discount means there's no discount, so we use COALESCE to set the default to 0:

```
SELECT name,
       price - COALESCE(discount, 0) as net_price
FROM products
```

products

name	price	discount
orange	200	NULL
apple	100	23
lemon	150	NULL

query output

name	net_price
orange	200
apple	77
lemon	150

② Use data from 2 (or more!) different columns

This query gets the best guess at a customer's state:

mailing address, most accurate

if not, try billing address

as a last resort, use their IP address

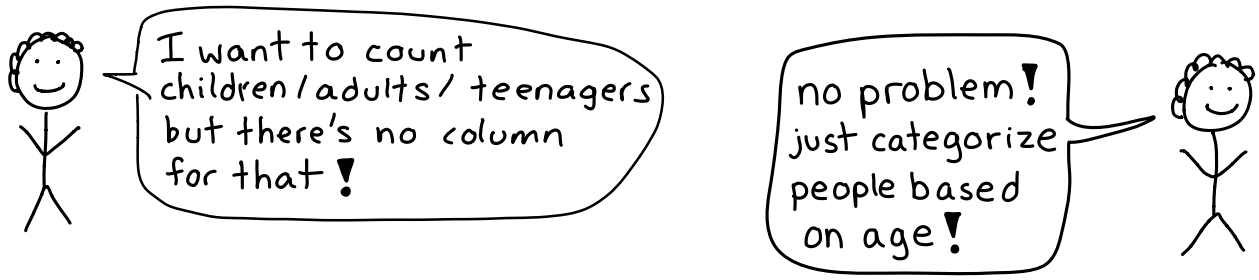
```
SELECT customer,
    COALESCE(mailing_state, billing_state, ip_address_state) AS state
FROM addresses
```

customer	mailing_state	billing_state	ip_address_state
1	Bihar	Bihar	Bihar
2	NULL	Kerala	Kerala
3	NULL	NULL	Punjab
4	Gujarat	Punjab	Gujarat

state
Bihar
Kerala
Punjab
Gujarat

♥ CASE ♥

Often I want to categorize by something that isn't a column:



`CASE` is how to write an `if` statement in SQL. Here's the syntax:

```
CASE
  WHEN <condition>       THEN <result>
  WHEN <other-condition> THEN <result>
  ...
  ELSE <result>
END
```

★ example ★

Here's how to categorize people into age ranges!

```
SELECT first_name, age, CASE
   WHEN age < 13 THEN 'child'
   WHEN age < 20 THEN 'teenager'
   ELSE 'adult' END AS age_range
FROM people
```

returns first THEN where the condition matches

people

first_name	age
ahmed	5
marie	17
akira	60
pablo	15

first_name	age	age_range
ahmed	5	child
marie	17	teenager
akira	60	adult
pablo	15	teenager

ways to count rows

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 "most popular names" query:

```
SELECT first_name, COUNT(*)
FROM people
GROUP BY first_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 plant genus has:

"GROUP BY 1" means group by the first expression in the SELECT

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

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

This trick using SUM and CASE lets you count how many cats vs dogs vs other animals each owner has:

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 owner
```

pets

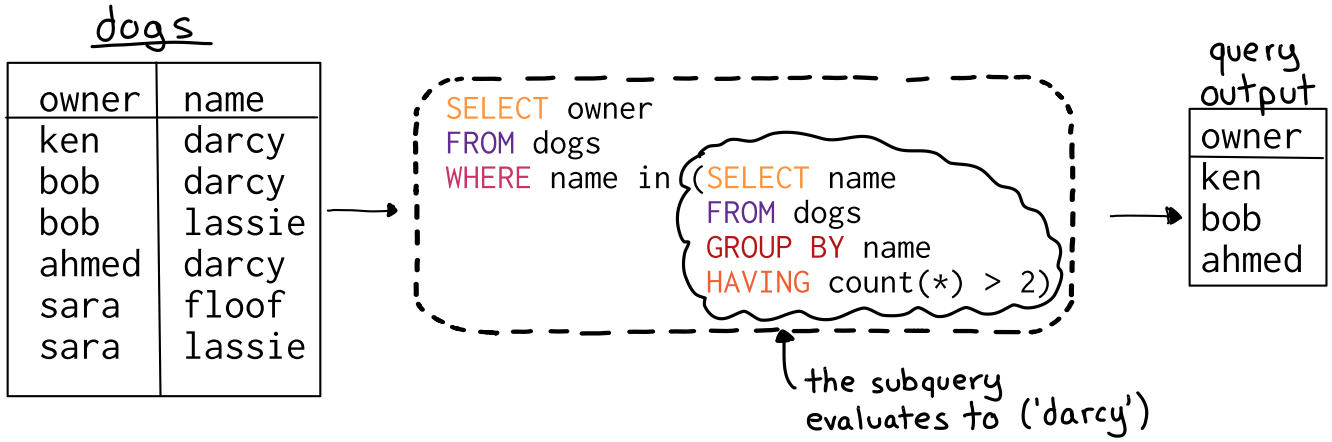
owner	type
1	dog
1	cat
2	dog
2	parakeet

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

subqueries

Some questions can't be answered with one simple SQL query.

For example, this query finds owners who have named their dogs popular names: ("boring" owners :))



★ common table expressions ★

"Common table expressions" (or CTEs) let you name a query so people reading it can understand what it's for.
Here's the query above rewritten using a CTE:

```
WITH popular_dog_names AS (
    SELECT name
    FROM dogs
    GROUP BY name
    HAVING count(*) > 2
)
SELECT owner
FROM dogs INNER JOIN popular_dog_names
     ON dogs.name = popular_dog_names.name
```

★ where you can use a subquery/CTE ★

♥ in a FROM ♥

```
SELECT .....
FROM (<subquery or CTE>)
GROUP BY .....
```

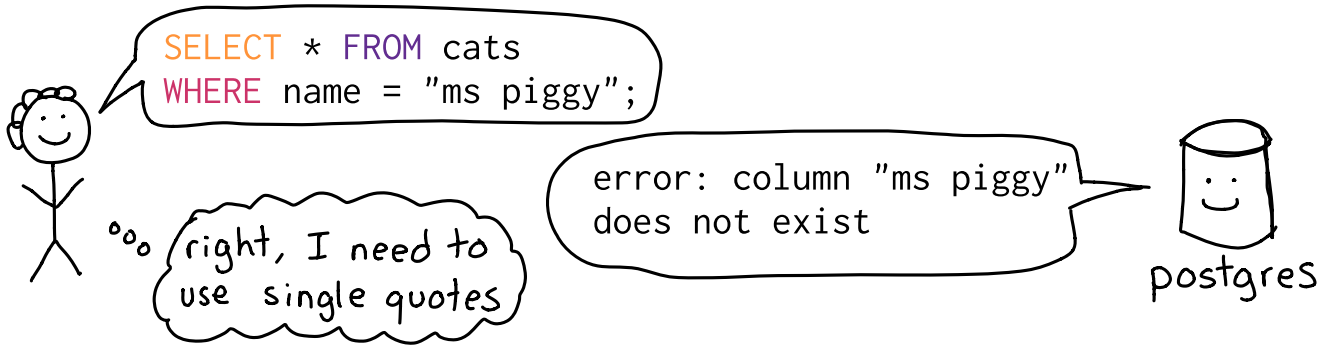
♥ in a WHERE ♥

```
SELECT ...

WHERE name IN (<subquery>)
```

tip: single quote strings

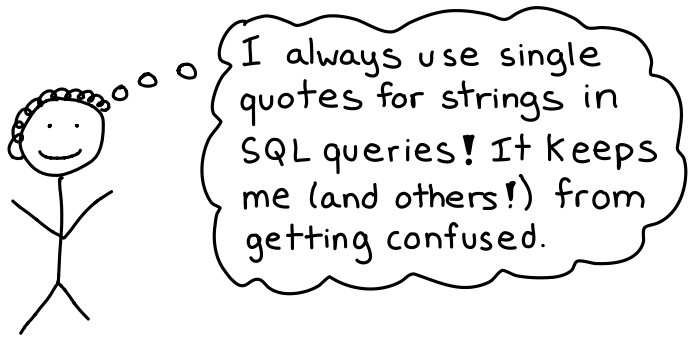
In some SQL implementations (like PostgreSQL), if you double quote a string it'll interpret it as a column name:



Here's a table explaining what different quotes mean in different SQL databases. "Identifier" means a column name or table name.

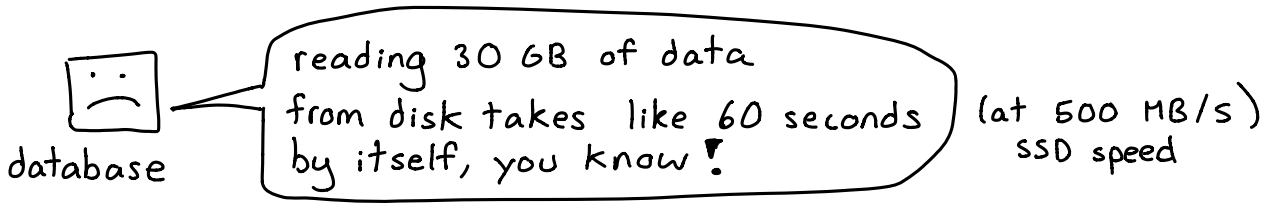
Sometimes table names have special characters like spaces in them so it's useful to be able to quote them.

	single quotes 'ms piggy'	double quotes "ms piggy"	backticks `ms piggy`
MySQL	string	string or identifier	identifier
PostgreSQL	string	identifier	invalid
SQLite	string	string or identifier	identifier
SQL server	string	string or identifier	invalid

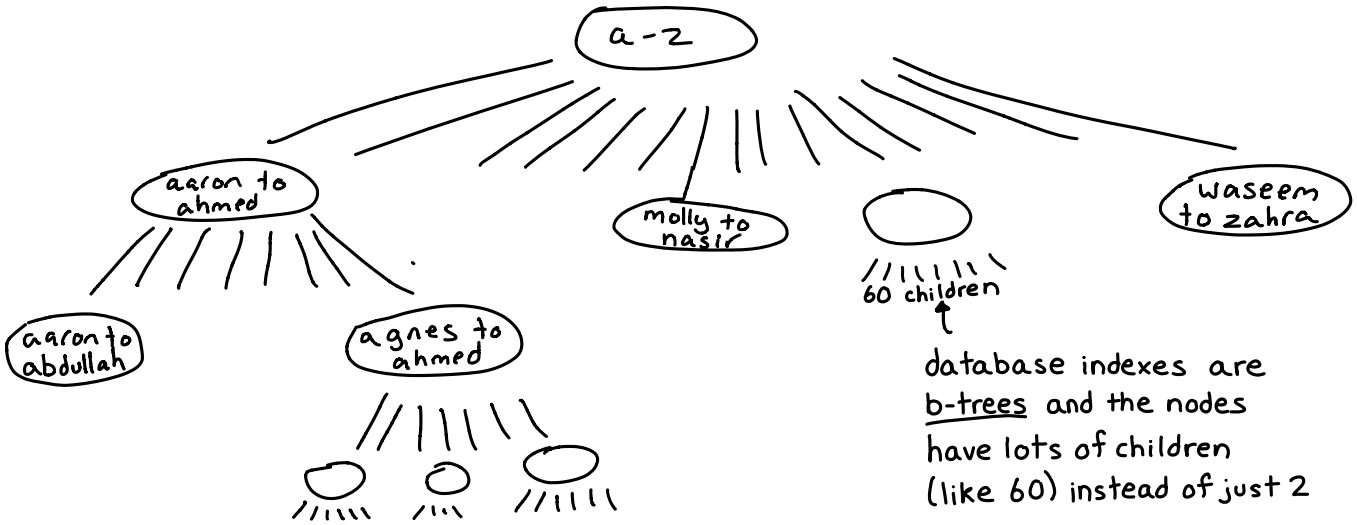


how indexes make your queries fast

By default, if you run `SELECT * FROM cats WHERE name = 'mr darcy'` the database needs to look at every single row to find matches

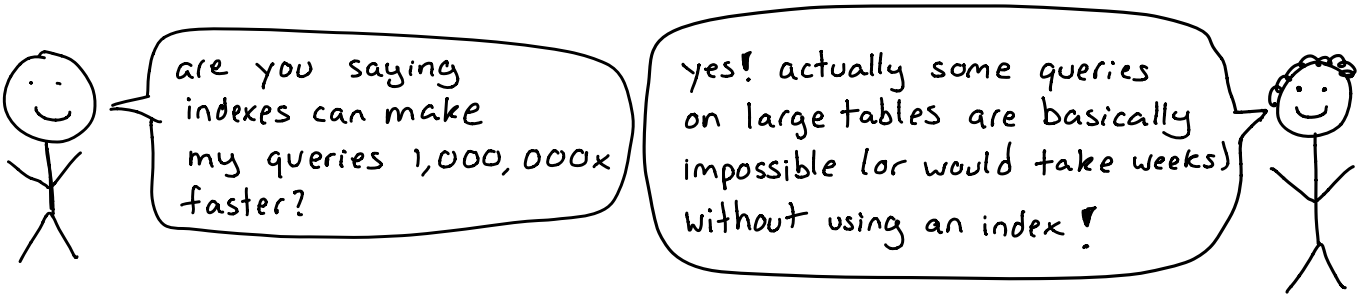


Indexes are a tree structure that makes it faster to find rows. Here's what an index on the 'name' column might look like.



$\log_{60}(1{,}000{,}000{,}000) = 5.06$

This means that if you have 1 billion names to look through, you'll only need to look at maybe (5) nodes in the index to find the name you're looking for (5 is a lot less than 1 billion!!!).

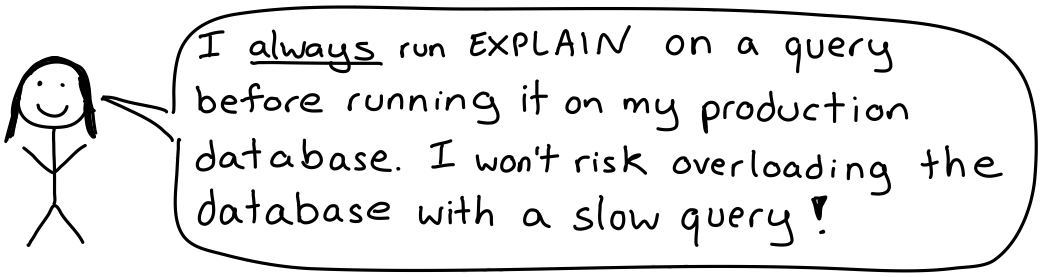


EXPLAIN your slow queries

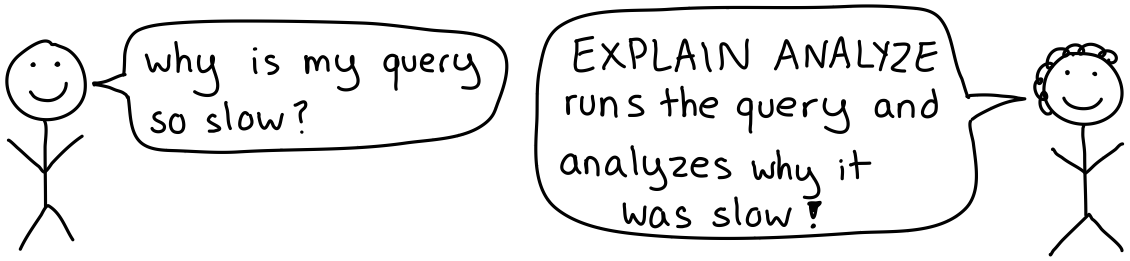
Sometimes queries run slowly, and EXPLAIN can tell you why!

2 ways you can use EXPLAIN in PostgreSQL: (other databases have different syntax for this)

① Before running the query (EXPLAIN SELECT ... FROM ...)
This calculates a query plan but doesn't run the query.



② After running the query (EXPLAIN ANALYZE SELECT ... FROM ...)



Here are the EXPLAIN ANALYZE results from PostgreSQL for the same query run on two tables of 1,000,000 rows: one table that has an index and one that doesn't

```
EXPLAIN ANALYZE SELECT * FROM users WHERE id = 1
```

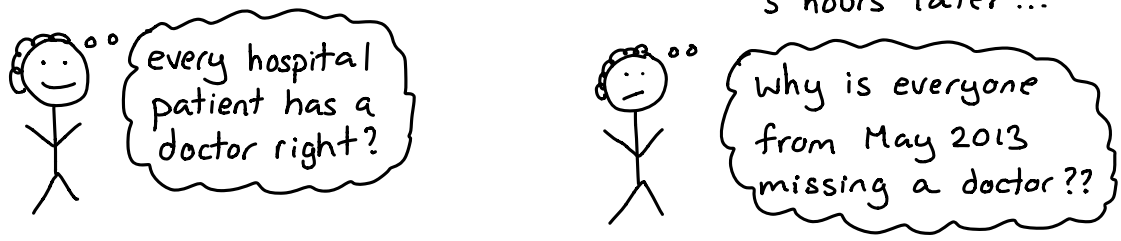
unindexed table	indexed table
Seq Scan on users	Index Only Scan using users_id_idx on users
Filter: (id = 1)	Index Cond: (id = 1)
Rows Removed by Filter: 999999	Heap Fetches: 1
Planning time: 0.185 ms	Planning time: 3.411 ms
Execution time: 179.412 ms	Execution time: 0.088 ms

"Seq Scan" means it's looking at each row (slow!)

the query runs 50 times faster with an index

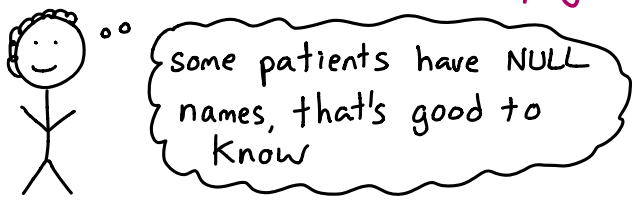
questions to ask about your data

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

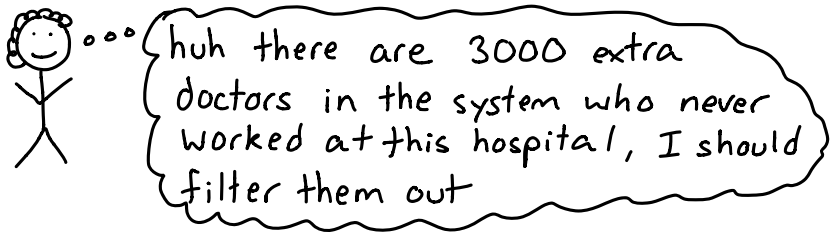


Some questions you might want to ask:

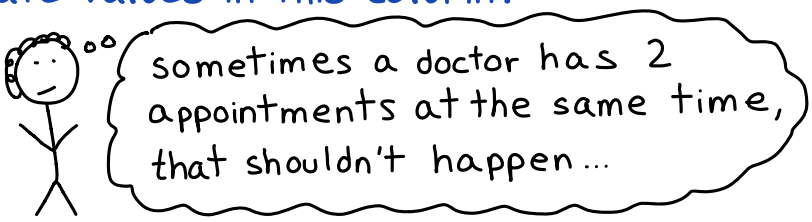
Does this column have `NULL` or `0` or empty string values?



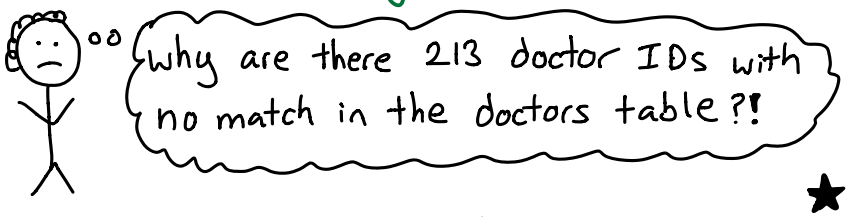
How many different values does this column have?



Are there duplicate values in this column?



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



★ A lot of these can also be enforced by `NOT NULL` or `UNIQUE` or `FOREIGN KEY` constraints on your tables. ★ ★

♥ thanks for reading ♥

When you're learning it's important to experiment! So you can try out any of the queries in this zine and run your own in an SQL playground:

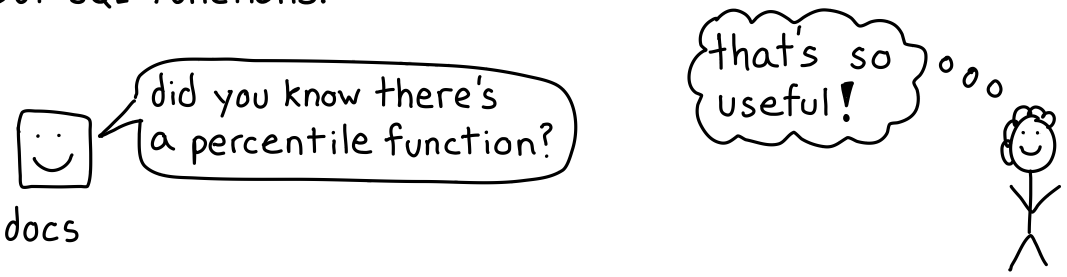
★ https://sql-playground.wizardzines.com ★

Here are a few more great SQL resources:

★ Use the index, LUKE! (https://use-the-index-luke.com) is a very in-depth explanation of how to use indexes to make your queries fast.

★ There are several visualizers that will help you understand the output of an EXPLAIN. For example: https://explain.depesz.com/ for PostgreSQL!

★ The official documentation is always GREAT for learning about SQL functions.



★ credits ★

Cover illustration: Deise Lino
Editing: Dolly Lanuza, Kamal Marhubi, Samuel Wright
Reviewers: Anton Dubrau, Arielle Evans

love this?
more zines at
★ wizardzines.com ★