

SQL

Remy Wang

April 2026

SQL is the language for databases. A SQL program is also called a *query*. Each query takes one or more tables as input, and produces one table as output. A table is a collection of rows, where each row has the same number of columns. Each column has a name and a type, and stores the same type of data.

name	age	weight
Casa	9	12
Kira	7	10
Toby	17	40
Maya	10	34

Here's the simplest SQL query:

```
SELECT name  
FROM pets
```

It returns a table containing a single column, namely `name`.

We can also return multiple columns, or evaluate expressions over them:

```
SELECT name, weight / (age + 5)
FROM pets
```

Conditions in the WHERE clause can be used to filter out rows:

```
SELECT name
FROM pets
WHERE age < 10
```

The query can be understood as a for-loop:

```
for (name, age, weight) in pets:
    if age < 10:
        print(name)
```

An aggregate function takes in a whole column and returns a single value:

-- also try MAX, SUM, or COUNT

```
SELECT MIN(age)
FROM pets
```

Aggregate functions can also be applied by group:

name	age	weight	kind
Casa	9	12	cat
Kira	7	10	cat
Toby	17	40	dog
Maya	10	34	dog

```
SELECT kind, max(age)
FROM pets
GROUP BY kind
```

The rows will get collected into different groups, one per different value of `kind`. Here we will get two groups:

name	age	weight	kind
Casa	9	12	cat
Kira	7	10	cat

name	age	weight	kind
Toby	17	40	dog
Maya	10	34	dog

Then, the aggregate function is applied to each group, reducing it to one row each:

kind	max(age)
cat	12
dog	17

Note that every column in the GROUP BY clause *must* also appear in the SELECT clause, and any other column in the SELECT clause *must* be aggregated. The following query is invalid because weight neither appears in GROUP BY nor is aggregated.

```
SELECT kind, weight, max(age)
FROM pets
GROUP BY kind
```

A more formal way to represent database queries is *relational algebra*. Relational algebra consists of a set of *relation* (a.k.a. table) operators. Each operator takes one or more tables as input, and produces an output table.

The *selection* operator $\sigma_p(t)$ filters table t by condition p ,
e.g. $\sigma_{age < 10}(\text{pets})$ corresponds to:

```
SELECT *  
  FROM pets  
 WHERE age < 10
```

Here SELECT * means “output all columns”.

The *projection* operator $\pi_e(t)$ evaluates (“maps”) an expression e over each row of t . For example, $\pi_{\text{age}+5, \text{weight}*2}(t)$ corresponds to:

```
SELECT age + 5, weight * 2
FROM pets
```

The *aggregation* operator $\gamma_{x,y,F(z)}$ groups a table by columns x, y and aggregates each group by function F . $\gamma_{\text{age},\text{max}(\text{weight})}(\text{pets})$ corresponds to:

```
SELECT age, MAX(weight)
FROM pets
GROUP BY age
```

Note that grouping and aggregation is done in one operator, because grouping alone does not produce a valid table (it produces a “mini table” per group).

Here are the relevant operators so far, and which part of a SQL query they correspond to. Confusingly, although σ is called the *selection* operator, it actually corresponds to the WHERE clause, whereas the *projection* π corresponds to the SELECT clause.¹

Name	Notation	Meaning	SQL
selection	$\sigma_p(t)$	filter by condition p	WHERE
projection	$\pi_{e(x,y)}(t)$	map an expression e over x, y	SELECT
aggregation	$\gamma_{x,F(y)}(t)$	group by x , aggregate over y using F	GROUP BY & SELECT

¹I would much rather call σ “filter” and π “map”

So far we've only been using one table in the FROM clause. We can also supply multiple tables. Suppose we break down our *pets* table into separate ones linked by ID:

ID	name
1	Casa
2	Kira

ID	age
1	9
2	7

To query across tables, we need to *join* them:

```
SELECT name, age
  FROM Pname, Page
 WHERE Pname.ID = Page.ID
```

Note that we can just use `name` and `age` in the output because it is unambiguous which table each column comes from, but in the `WHERE` clause we have to use `Pname.ID` and `Page.ID` to specify which ID columns we are comparing.

Let's break down the query to understand what's going on. First, if we drop the `WHERE` clause and `SELECT` all columns:

```
SELECT *  
FROM Pname, Page
```

This takes the *Cartesian product* of the tables. Namely it produces a table containing all possible pairings of the rows:

Pname.ID	name	Page.ID	age
1	Casa	1	9
1	Casa	2	7
2	Kira	1	9
2	Kira	2	7

Cartesian product has its own relational algebra operator \times ; the current query computes exactly $\text{Pname} \times \text{Page}$.

```
SELECT *  
  FROM Pname, Page  
 WHERE Pname.ID = Page.ID
```

Adding back the WHERE clause, we filter down the Cartesian product to keep only the rows where the IDs match:

Pname.ID	name	Page.ID	age
1	Casa	1	9
2	Kira	2	7

In relational algebra: $\sigma_{\text{Pname.ID}=\text{Page.ID}}(\text{Pname} \times \text{Page})$

```
SELECT name, age
  FROM Pname, Page
 WHERE Pname.ID = Page.ID
```

Finally, we keep only the name and age columns in the output:

name	age
Casa	9
Kira	7

In relational algebra: $\pi_{\text{name,age}}(\sigma_{\text{Pname.ID=Page.ID}}(\text{Pname} \times \text{Page}))$

Joins can be understood as running nested loops over the tables:

```
for Pname.ID, name in Pname:
    for Page.ID, age in Page:
        if Pname.ID == Page.ID:
            print(name, age)
```

If the column being *joined on* is a unique identifier, you can also think of the join as looping over one table while looking up from the other:

```
for ID, name in Pname:  
    age = Page[ID]  
    print(name, age)
```

So far we've been implementing our computation with single queries. Sometimes it's useful, or necessary, to compose multiple *subqueries* into a large one.

Suppose we add an `owner` column to our `pets` table. The following query finds people with either cats or dogs:

```
SELECT owner
  FROM pets
 WHERE pets.kind = 'cat' OR pets.kind = 'dog'
```

But the following query does *not* find people with both cats and dogs (why?):

```
SELECT owner
  FROM pets
 WHERE pets.kind = 'cat' AND pets.kind = 'dog'
```

Instead, we can INTERSECT two subqueries:

```
SELECT owner FROM pets WHERE kind = 'cat'  
INTERSECT  
SELECT owner FROM pets WHERE kind = 'dog'
```

Similarly, we can also take the union/difference of two tables:

```
-- also try EXCEPT  
SELECT owner FROM pets WHERE kind = 'cat'  
UNION  
SELECT owner FROM pets WHERE kind = 'dog'
```

When working with subqueries, it can be helpful to store the query result in variables. There are several different ways to do that in SQL.

First, we can just create a new table storing the query result:

```
CREATE TABLE catpeople AS
    SELECT owner FROM pets WHERE kind = 'cat';
CREATE TABLE dogpeople AS
    SELECT owner FROM pets WHERE kind = 'dog';
```

```
SELECT * FROM catpeople
INTERSECT
SELECT * FROM dogpeople
```

This stores the query result in “global variables” that are also accessible to other queries.

To store the result in a local variable accessible to a single query only, use WITH:

```
WITH catpeople AS (  
    SELECT owner FROM pets WHERE kind = 'cat'  
), dogpeople AS (  
    SELECT owner FROM pets WHERE kind = 'dog'  
)  
SELECT * FROM catpeople  
INTERSECT  
SELECT * FROM dogpeople
```

Note there are no semicolons after the WITH clauses because the above is considered one single query.

Sometimes we want to keep data stored in variables automatically updated as the underlying table changes. For this we can create *views*:

```
CREATE VIEW catpeople AS
    SELECT owner FROM pets WHERE kind = 'cat';
CREATE VIEW dogpeople AS
    SELECT owner FROM pets WHERE kind = 'dog';
```

Below, Lisa will appear in the result when the query runs the second time after the insertions, even though we didn't explicitly update catpeople and dogpeople.

```
SELECT * FROM catpeople  
INTERSECT  
SELECT * FROM dogpeople;
```

```
INSERT INTO pets values ('Carlo', 5, 20, 'dog', 'Lisa');  
INSERT INTO pets values ('Zoe', 3, 8, 'cat', 'Lisa');
```

```
SELECT * FROM catpeople  
INTERSECT  
SELECT * FROM dogpeople;
```

Subqueries are very helpful for complex queries. Suppose we want to find pet kinds with average age > 10 . The following query is wrong because aggregate cannot appear in the WHERE clause:²

```
SELECT kind, AVG(age) FROM pets
WHERE AVG(age) > 10
GROUP BY kind
```

²The condition is evaluated per row and we can't aggregate a single row.

Instead, we can first calculate the average age per kind:

```
CREATE TABLE averages AS
  SELECT kind, AVG(age) AS a
  FROM pets
  GROUP BY kind;
```

Then we filter that table by the average age:

```
SELECT *
  FROM averages
 WHERE averages.a > 10.0;
```

Or we can use the WITH syntax:

```
WITH averages AS (  
    SELECT kind, AVG(age) AS a  
    FROM pets GROUP BY kind  
)  
SELECT * FROM averages WHERE averages.a > 10.0;
```

For this particular query, SQL also has a convenient HAVING clause:

```
SELECT kind, AVG(age)
FROM pets
GROUP BY kind
HAVING AVG(age) > 10
```

It's like WHERE, but the condition is evaluated on each group *after* grouping so we can use aggregate functions.

Subqueries are even more powerful when *nested* inside another query. To appreciate this, let's try to find the oldest cat (“argmax”). We'll first do this the “hacky way” to pick up some SQL features:

```
SELECT name, age
  FROM pets
 WHERE kind = 'cat'
ORDER BY age DESC
LIMIT 1
```

The new bits here are `ORDER BY age DESC` – which sorts the output by age from large to small – and `LIMIT 1` – which limits the output to the first row. This is hacky because SQL is agnostic to ordering in general, but here we rely on the order.

To do this properly, we can use a subquery. First, we find the age of the oldest cat:

```
SELECT max(age)
  FROM pets
 WHERE kind = 'cat'
```

Then, we store this in a variable for use in another query:

```
WITH oldage AS (  
    SELECT max(age) AS a  
    FROM pets  
    WHERE kind = 'cat'  
)  
SELECT name, age  
FROM pets, oldage  
WHERE kind = 'cat'  
AND age = oldage.a
```

In this case, because the `oldage` subquery stores a single value (a.k.a. a *scalar*), we can use it directly as a *nested subquery*:

```
SELECT name, age
  FROM pets
 WHERE kind = 'cat'
    AND age = (SELECT max(age)
              FROM pets
              WHERE kind = 'cat')
```

Remember, this only works because the inner query is a scalar! Also note that, unlike the hacky query, this query returns *all* pets that are oldest.

Now, let's move on to the more general problem: how can we find the oldest pet(s) *per kind*? First we can remove the `kind = 'cat'` filter:

```
SELECT name, age
  FROM pets
 WHERE age = (SELECT max(age) FROM pets)
```

But that won't work, because the inner query now returns the *overall* oldest age.

To fix it, we need to filter the inner query by *the current pet kind* considered by the outer query:

```
SELECT name, age
  FROM pets as p1
 WHERE age = (SELECT max(age)
              FROM pets as p2
              WHERE p2.kind = p1.kind)
```

The idea is that, the inner query is evaluated once *per row* of the outer query, because the WHERE clause of the outer query runs once per row. Just like a nested loop.

We can also use nested queries in the SELECT clause:

```
SELECT kind, (SELECT max(age)
              FROM pets as p2
              WHERE p2.kind = p1.kind)
FROM pets as p1
```

Which is a convoluted way to implement “max age per kind”.³

³But this query is different from the normal GROUP BY – how?

Let's see two other ways to find old pets, while introducing a bit more SQL features along the way.

```
SELECT name, age
  FROM pets as p1
 WHERE NOT EXISTS (SELECT *
                   FROM pets as p2
                   WHERE p2.age > p1.age
                   AND p2.kind = p1.kind)
```

This query is rather self-explanatory: an oldest cat is one where there is no cat older than it.

We can also use the ALL keyword:

```
SELECT name, age
  FROM pets as p1
 WHERE age >= ALL (SELECT age
                   FROM pets as p2
                   WHERE p2.kind = p1.kind)
```

Here, `age >= ALL (SELECT ...)` is true if and only if `age >= x` for *all* values of `x` returned by the subquery.

Similarly, we can also use ANY:

```
SELECT name, age
  FROM pets as p1
 WHERE not age < ANY (SELECT age
                       FROM pets as p2
                       WHERE p2.kind = p1.kind)
```

Naturally, the oldest cat shouldn't be younger than any other cat.

There's *nothing* left to learn in SQL, and we'll cover exactly that now. Wait, what?

Consider the following queries. What do you expect them to return?

```
SELECT COUNT(*) FROM r;
```

```
SELECT COUNT(*) FROM r WHERE r.x = r.x;
```

```
SELECT COUNT(*) FROM r WHERE NOT (r.x <> r.x);
```

```
SELECT COUNT(*) FROM r WHERE r.x = r.x OR r.x = r.x;
```

Normally, all queries should return the size of the input table *r*, but not if *r* contains *nothing*, and by *nothing*, I mean NULL.

In SQL, the NULL value stands for the absence of a value. There are many occasions when you might want to use NULL:

- ▶ You don't know the true value of an entry (age of a shelter cat)
- ▶ The column is not applicable to this row ("owner" of a stray)
- ▶ The entry should be treated as a default value

In general, NULL means "missing information".

SQL treats NULL specially during query evaluation. To understand that, we need to look at SQL operations more formally.

First, there are two kinds of values in SQL:

- ▶ **Data values**
- ▶ **Logical values**

Data values are like numbers, dates, strings, etc.—the stuff you’re actually interested in storing in your tables. A special data value is the NULL value, representing “nothingness” or “unknown”.

Logical values are your familiar booleans true and false, used by the engine in intermediate calculations. A special logical value is the UNKNOWN value, representing “unknown”. You can think of it as just the logical counterpart to NULL.

There are 3 kinds of operators in SQL:

1. Data operators
2. Predicates
3. Logical connectives

Logical connectives:

 true AND false
<logical value> <logical operator> <logical value>

The entire expression evaluates to itself another **logical value**.

Conveniently summarized:

Operator kind	Example	Input -> Output
Data	+ - * /	data → data
Predicate	> < =	data → logic
Logical	AND OR NOT	logic → logic

Some systems support “if/case statements” that goes from logic to data, but you won’t need those very often.

These operators all behave as you would expect if NULL is *not* involved. When NULL *is* involved, there are special rules:

Kind	Example	Output on missing input
Data	+ - * /	NULL
Predicate	> < =	UNKNOWN
Logical	AND OR NOT	<i>3 Valued Logic</i>

For **data**: if either operand is NULL, the result will also be NULL

For **predicate**: if any argument is NULL, the output is UNKNOWN⁴

For **logical**, we apply **3 valued logic**

⁴a special case is IS NULL, which of course returns true if the argument is NULL.

3 Valued Logic

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	T	F	U
	F	T	U

Don't memorize these – they fall naturally out of “short-circuiting”:
if we can already deduce the result from the non-missing operand,
we use that, otherwise it's UNKNOWN.

Finally, SQL will return a row only if the condition in WHERE is true on that row. In other words, it discards rows when the condition is false or UNKNOWN.

Let's now revisit the examples using these rules.

```
SELECT *  
FROM R  
WHERE R.x=R.x; -- NULL=NULL => UNKNOWN
```

Where $R.x$ is NULL, the WHERE clause evaluates to UNKNOWN, so that row is *excluded*. The return table is *not* necessarily the same as R.

```
SELECT *  
FROM R  
WHERE R.x = R.x  
OR R.x <> R.x; -- NULL <> NULL => UNKNOWN
```

Same as above.

```
SELECT *  
FROM R  
WHERE null = null; -- NULL=NULL => UNKNOWN
```

Same as above. It's just explicit this time.

```
SELECT *  
FROM R  
WHERE null <> null; -- NULL <> NULL => UNKNOWN
```

Same as above. It's just explicit this time.

```
SELECT *  
FROM R  
-- NULL <> NULL => UNKNOWN  
-- NOT UNKNOWN => still UNKNOWN  
WHERE NOT null <> null;
```

This time we have a nested operation. Just follow through with the rules and you'll find when WHERE evaluates to UNKNOWN.