Open in app

Get started

Published in Towards Data Science

GreekDataGuy Follow
Apr 6, 2020 · 14 min read · ✦ · Listen

Save

SQL Query Cheatsheet for Postgres

The SQL queries I use as a data scientist and software engineer

Photo by Markus Spiske on Unsplash

I write a lot of raw SQL whether I'm loading data in an app, or querying data to train a

As engineers who write lots of code, some commands will stick in our memory, but some we'll have to look up. As I heard once and like to repeat - good engineers are like indexes, they can look up information quickly.

This is my personal cheat sheet for SQL, written with Postgres in mind but roughly applicable to other relational databases. This exclusively covers queries. No inserts, deletes, indexing or other Postgres functionality. We'll start simple and work towards more interesting queries.

First, we'll create a database and insert some data, then we'll execute every SQL query on that data and investigate the output.

Contents:

Setup

Create a few related tables with different data types. I've imagined the data around a fictional company's CRM (customer relationship management) system so you can relate it to real life.

After you've created a database. Open your favorite SQL editor and run the following to create tables for `users`, `mailing_lists`, `products` and `sales`.

```
-- users whose information the company has
```

```
 created_at TIMESTAMP
);

-- users who are on the company mailing list
create table mailing_lists (
 id serial primary key,
 first_name varchar (50),
 email varchar (50),
 created_at TIMESTAMP
);

-- products the company sells
create table products (
 id serial primary key,
 name varchar(50),
 manufacturing_cost int,
 data jsonb,
 created_at TIMESTAMP
)

-- sales transactions of products by users
create table sales (
 id serial primary key,
 user_id int,
 product_id int,
 sale_price int,
 created_at TIMESTAMP
);
```

And let’s populate those tables.

```
insert into users (first_name, location, created_at)
values
 ('Liam', 'Toronto', '2010-01-01'),
 ('Ava', 'New York', '2011-01-01'),
 ('Emma', 'London', '2012-01-01'),
 ('Noah', 'Singapore', '2012-01-01'),
 ('William', 'Tokyo', '2014-01-01'),
 ('Oliver', 'Beijing', '2015-01-01'),
 ('Olivia', 'Moscow', '2014-01-01'),
 ('Mia', 'Toronto', '2015-01-01');

insert into mailing_lists (first_name, email, created_at)
values
 ('Liam', 'liam@fake.com', '2010-01-01'),
 ('Ava', 'ava@fake.com', '2011-01-01');
```

```
  ('smart phone', 200, '{"in_stock":10}', '2010-01-01'),
  ('TV', 1000, '{}', '2010-01-01');

insert into sales (user_id, product_id, sale_price, created_at)
values
  (1, 1, 900, '2015-01-01'),
  (1, 2, 450, '2016-01-01'),
  (1, 3, 2500, '2017-01-01'),
  (2, 1, 800, '2017-01-01'),
  (2, 2, 600, '2017-01-01'),
  (3, 3, 2500, '2018-01-01'),
  (4, 3, 2400, '2018-01-01'),
  (null, 3, 2500, '2018-01-01');
```

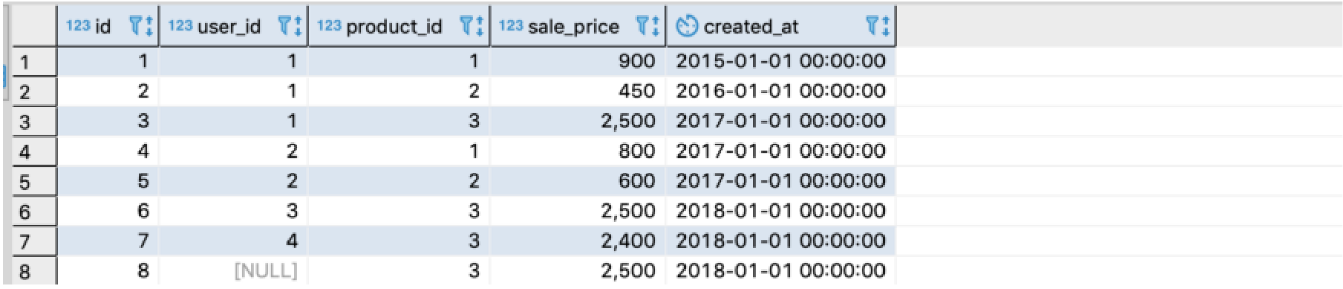
Selects and Counts

Select

This is the basic query around which everything later will be based.

Get all sales data without filtering or manipulating it. Simple.

```
select * from sales;
```

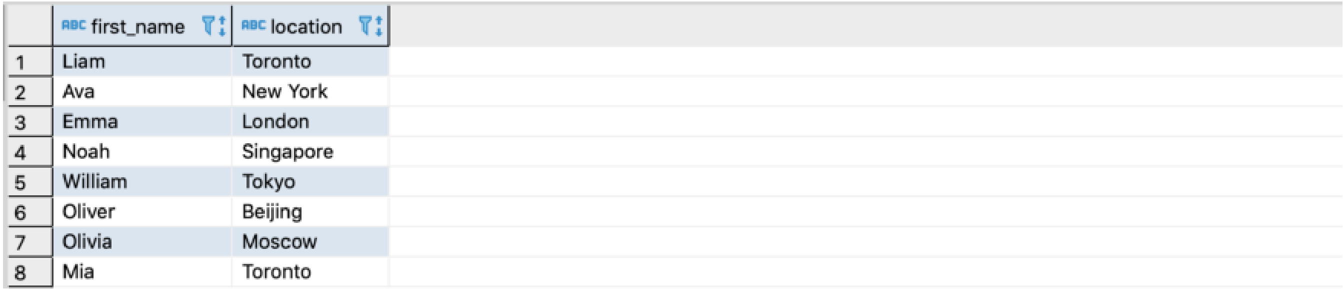


	123 id	123 user_id	123 product_id	123 sale_price	created_at
1	1	1	1	900	2015-01-01 00:00:00
2	2	1	2	450	2016-01-01 00:00:00
3	3	1	3	2,500	2017-01-01 00:00:00
4	4	2	1	800	2017-01-01 00:00:00
5	5	2	2	600	2017-01-01 00:00:00
6	6	3	3	2,500	2018-01-01 00:00:00
7	7	4	3	2,400	2018-01-01 00:00:00
8	8	[NULL]	3	2,500	2018-01-01 00:00:00

Select specific columns

Retrieve only the name and location of users, excluding other information like when a record was created.

```
select
  first_name,
  location
from users;
```



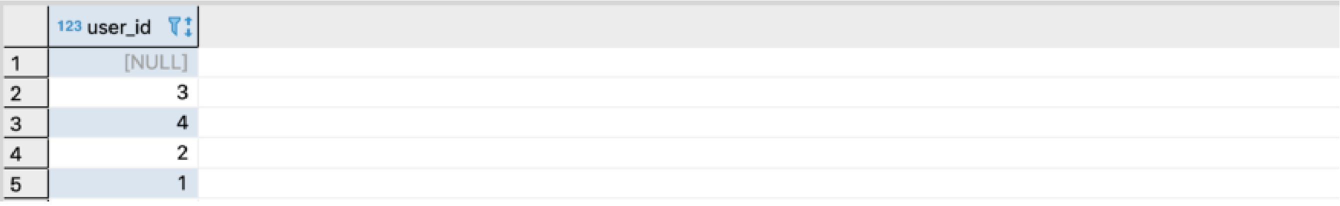
	first_name	location
1	Liam	Toronto
2	Ava	New York
3	Emma	London
4	Noah	Singapore
5	William	Tokyo
6	Oliver	Beijing
7	Olivia	Moscow
8	Mia	Toronto

Select Distinct

Removes duplicates in a specific column from the query.

Useful if you wanted to find all users who had ever bought something once, rather than a list of every transaction from the sales table.

```
select distinct user_id from sales;
```



	user_id
1	[NULL]
2	3
3	4
4	2
5	1

Count

Count the records in a table.

I use this all the time to get a sense of the size of a table before writing any other queries.

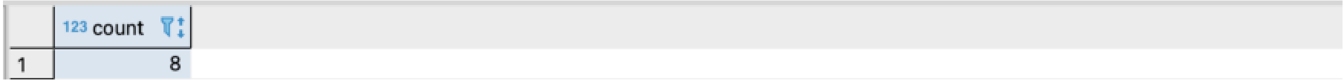
```
select count(*) from products;
```

Count a subquery

You can also wrap a whole query in `count()` if you want to see the number of records inclusive of a `join` or `where` clause.

Useful because sometimes the number of records can change by an order of magnitude after a join.

```
select count(*) from (
  select * from products
  left join sales on sales.product_id = products.id
) subquery;
```



	count
1	8

Limit, Offset and Order By

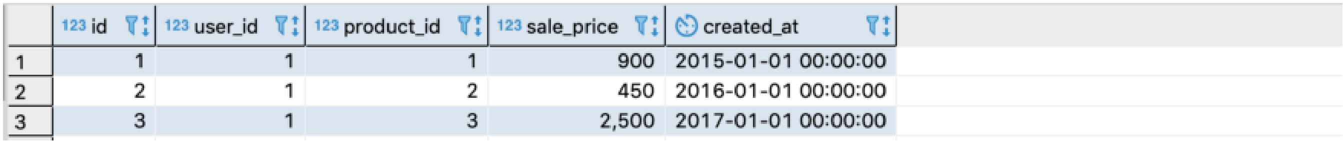
Limit

Limit the number of returned records to a specific count.

I've found this useful when I'm loading data-heavy records into a jupyter notebook and loading too many crashes my computer. In such a case I may add `limit 100`.

We'll just get the first 3 records from sales.

```
select * from sales limit 3;
```

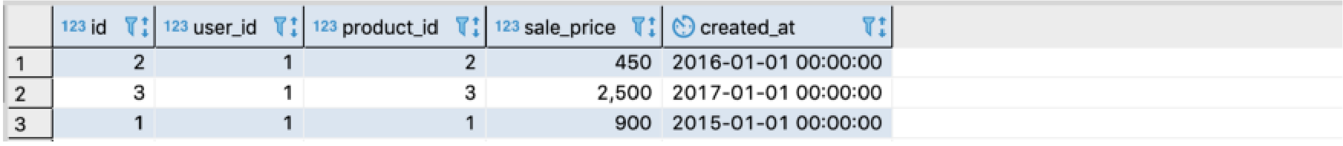


	id	user_id	product_id	sale_price	created_at
1	1	1	1	900	2015-01-01 00:00:00
2	2	1	2	450	2016-01-01 00:00:00
3	3	1	3	2,500	2017-01-01 00:00:00

Useful if you want user names in alphabetical order, or a table ordered by a foreign key.

Get sales ordered by `user_id`. Note we still have the `limit` here.

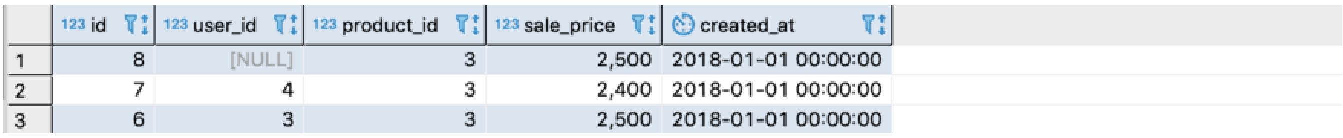
```
select
  *
from sales
order by user_id asc
limit 3;
```



	id	user_id	product_id	sale_price	created_at
1	2	1	2	450	2016-01-01 00:00:00
2	3	1	3	2,500	2017-01-01 00:00:00
3	1	1	1	900	2015-01-01 00:00:00

We can also order sales in the opposite direction, descending.

```
select
  *
from sales
order by user_id desc
limit 3;
```



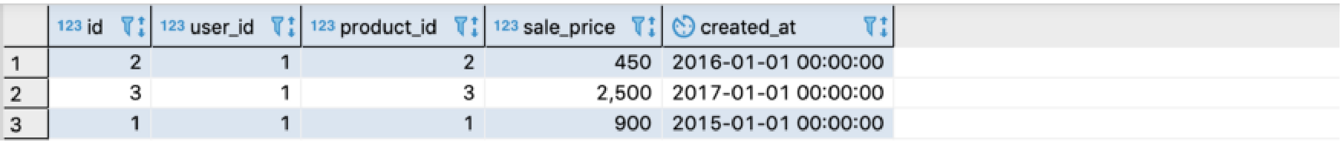
	id	user_id	product_id	sale_price	created_at
1	8	[NULL]	3	2,500	2018-01-01 00:00:00
2	7	4	3	2,400	2018-01-01 00:00:00
3	6	3	3	2,500	2018-01-01 00:00:00

Offset

Skip N records off the top, in a `select`.

I find this very useful when loading too many records at once crashes my jupyter notebook and I want to iterate over a finite number of records at a time.

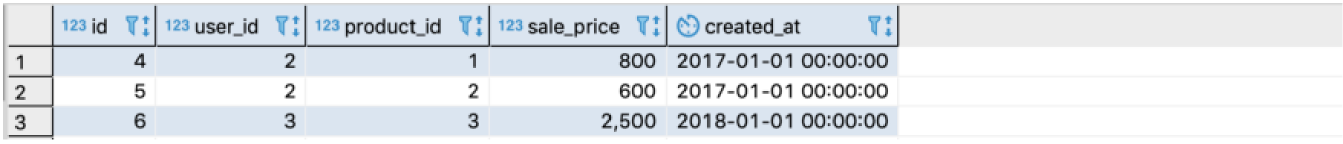
Grab the first 3 records from sales.



	id	user_id	product_id	sale_price	created_at
1	2	1	2	450	2016-01-01 00:00:00
2	3	1	3	2,500	2017-01-01 00:00:00
3	1	1	1	900	2015-01-01 00:00:00

Grab the next 3 records from sales.

```
select * from sales
order by user_id asc
limit 3 offset 3;
```



	id	user_id	product_id	sale_price	created_at
1	4	2	1	800	2017-01-01 00:00:00
2	5	2	2	600	2017-01-01 00:00:00
3	6	3	3	2,500	2018-01-01 00:00:00

Notice how we have records 1–3 in the first query and 4–6 in the second.

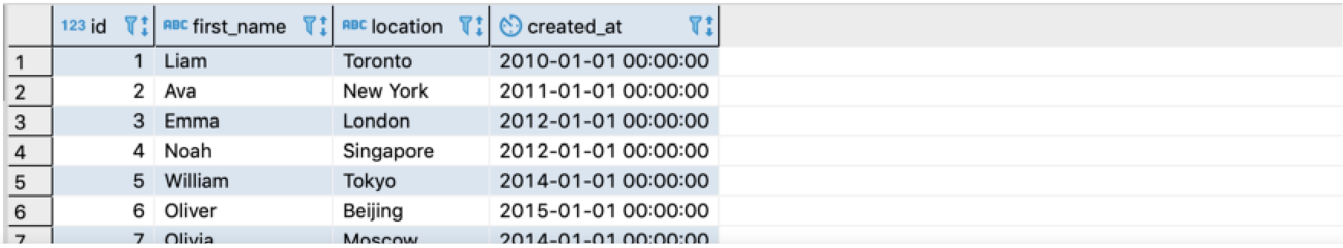
Joins

Left join

Starts with a base table (on the left) and tries to join records from the other table (on the right) based on a key.

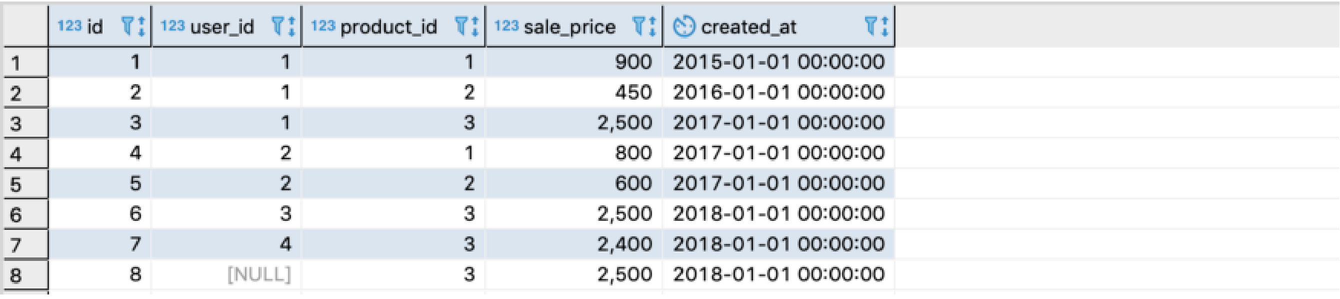
Before joining, let's just examine the left table.

```
select * from users
```



	id	first_name	location	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00
2	2	Ava	New York	2011-01-01 00:00:00
3	3	Emma	London	2012-01-01 00:00:00
4	4	Noah	Singapore	2012-01-01 00:00:00
5	5	William	Tokyo	2014-01-01 00:00:00
6	6	Oliver	Beijing	2015-01-01 00:00:00
7	7	Olivia	Moscow	2014-01-01 00:00:00

```
select * from sales
```

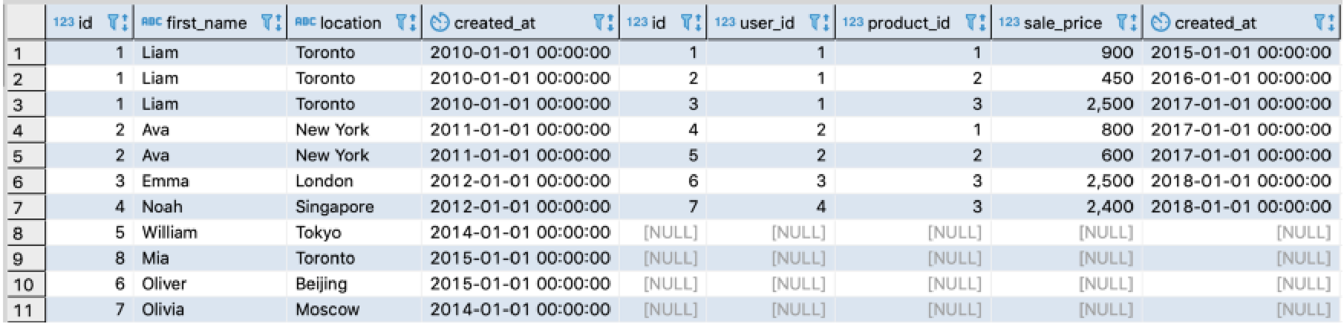


	id	user_id	product_id	sale_price	created_at
1	1	1	1	900	2015-01-01 00:00:00
2	2	1	2	450	2016-01-01 00:00:00
3	3	1	3	2,500	2017-01-01 00:00:00
4	4	2	1	800	2017-01-01 00:00:00
5	5	2	2	600	2017-01-01 00:00:00
6	6	3	3	2,500	2018-01-01 00:00:00
7	7	4	3	2,400	2018-01-01 00:00:00
8	8	[NULL]	3	2,500	2018-01-01 00:00:00

Now we'll do a left join. Join sales (right) on users (left).

Notice how records on the left are always returned regardless of if they match a record on the right. And that records on the left are duplicated if they match multiple records on the right.

```
select
  *
from users
left join sales on sales.user_id = users.id;
```



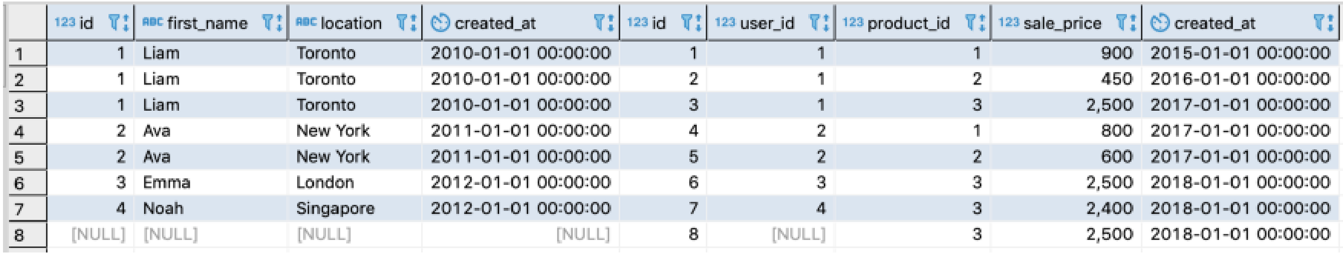
	id	first_name	location	created_at	id	user_id	product_id	sale_price	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00	1	1	1	900	2015-01-01 00:00:00
2	1	Liam	Toronto	2010-01-01 00:00:00	2	1	2	450	2016-01-01 00:00:00
3	1	Liam	Toronto	2010-01-01 00:00:00	3	1	3	2,500	2017-01-01 00:00:00
4	2	Ava	New York	2011-01-01 00:00:00	4	2	1	800	2017-01-01 00:00:00
5	2	Ava	New York	2011-01-01 00:00:00	5	2	2	600	2017-01-01 00:00:00
6	3	Emma	London	2012-01-01 00:00:00	6	3	3	2,500	2018-01-01 00:00:00
7	4	Noah	Singapore	2012-01-01 00:00:00	7	4	3	2,400	2018-01-01 00:00:00
8	5	William	Tokyo	2014-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
9	8	Mia	Toronto	2015-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
10	6	Oliver	Beijing	2015-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
11	7	Olivia	Moscow	2014-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]

`Left join` is probably used more than any other join by developers querying databases.

Right join.

It's the same as the `left join` but in the other direction. Start with the right table (sales) and join records on the left (users) if they exist.

```
from users
right join sales on sales.user_id = users.id;
```

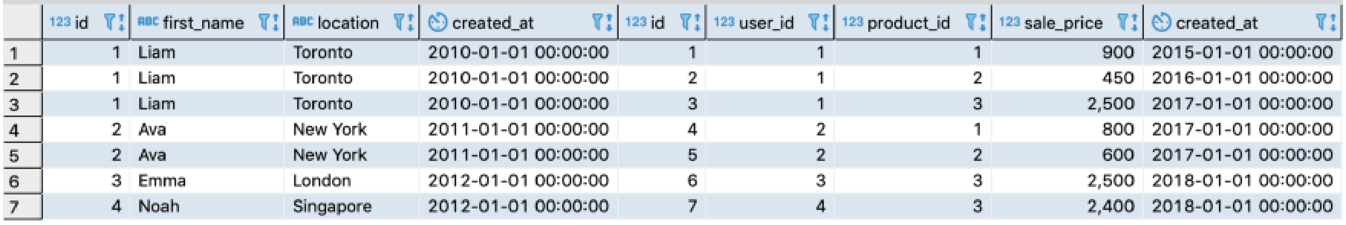


	id	first_name	location	created_at	id	user_id	product_id	sale_price	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00	1	1	1	900	2015-01-01 00:00:00
2	1	Liam	Toronto	2010-01-01 00:00:00	2	1	2	450	2016-01-01 00:00:00
3	1	Liam	Toronto	2010-01-01 00:00:00	3	1	3	2,500	2017-01-01 00:00:00
4	2	Ava	New York	2011-01-01 00:00:00	4	2	1	800	2017-01-01 00:00:00
5	2	Ava	New York	2011-01-01 00:00:00	5	2	2	600	2017-01-01 00:00:00
6	3	Emma	London	2012-01-01 00:00:00	6	3	3	2,500	2018-01-01 00:00:00
7	4	Noah	Singapore	2012-01-01 00:00:00	7	4	3	2,400	2018-01-01 00:00:00
8	[NULL]	[NULL]	[NULL]	[NULL]	8	[NULL]	3	2,500	2018-01-01 00:00:00

Inner join

Only return records if a match exists on both sides. Notice we have no empty data.

```
select
  *
from users
inner join sales on sales.user_id = users.id;
```

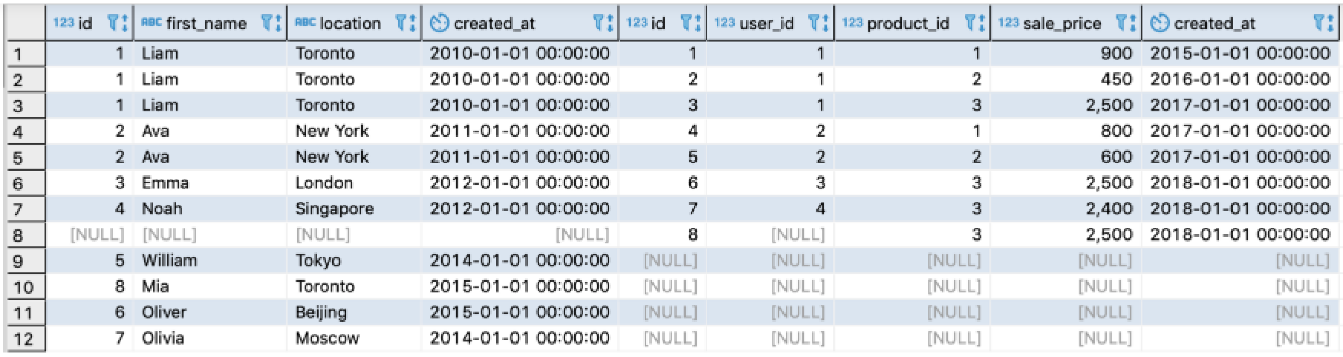


	id	first_name	location	created_at	id	user_id	product_id	sale_price	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00	1	1	1	900	2015-01-01 00:00:00
2	1	Liam	Toronto	2010-01-01 00:00:00	2	1	2	450	2016-01-01 00:00:00
3	1	Liam	Toronto	2010-01-01 00:00:00	3	1	3	2,500	2017-01-01 00:00:00
4	2	Ava	New York	2011-01-01 00:00:00	4	2	1	800	2017-01-01 00:00:00
5	2	Ava	New York	2011-01-01 00:00:00	5	2	2	600	2017-01-01 00:00:00
6	3	Emma	London	2012-01-01 00:00:00	6	3	3	2,500	2018-01-01 00:00:00
7	4	Noah	Singapore	2012-01-01 00:00:00	7	4	3	2,400	2018-01-01 00:00:00

Outer join

Return all records on left and right regardless of whether a they can be matched on a key. Some records on the left don’t have matching records on the right, and vice versa. But all are returned anyway.

```
select
 *
from users
full outer join sales on sales.user_id = users.id;
```



	id	first_name	location	created_at	id	user_id	product_id	sale_price	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00	1	1	1	900	2015-01-01 00:00:00
2	1	Liam	Toronto	2010-01-01 00:00:00	2	1	2	450	2016-01-01 00:00:00
3	1	Liam	Toronto	2010-01-01 00:00:00	3	1	3	2,500	2017-01-01 00:00:00
4	2	Ava	New York	2011-01-01 00:00:00	4	2	1	800	2017-01-01 00:00:00
5	2	Ava	New York	2011-01-01 00:00:00	5	2	2	600	2017-01-01 00:00:00
6	3	Emma	London	2012-01-01 00:00:00	6	3	3	2,500	2018-01-01 00:00:00
7	4	Noah	Singapore	2012-01-01 00:00:00	7	4	3	2,400	2018-01-01 00:00:00
8	[NULL]	[NULL]	[NULL]	[NULL]	8	[NULL]	3	2,500	2018-01-01 00:00:00
9	5	William	Tokyo	2014-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
10	8	Mia	Toronto	2015-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
11	6	Oliver	Beijing	2015-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
12	7	Olivia	Moscow	2014-01-01 00:00:00	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]

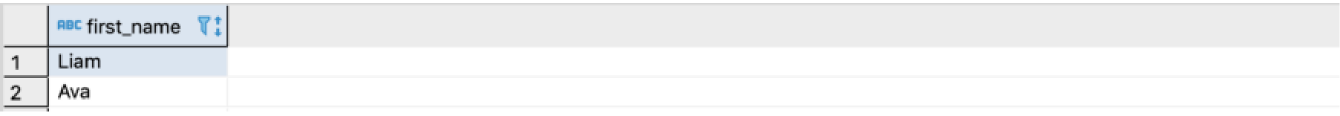
Intersect, Union and Except

Intersect

Not really a join but it can be used like one. It has the benefit of being able to match on `null` values, something inner join cannot do.

Here we'll just intersect names from `users` and the `mailing_lists`. Only names existing in both tables are returned.

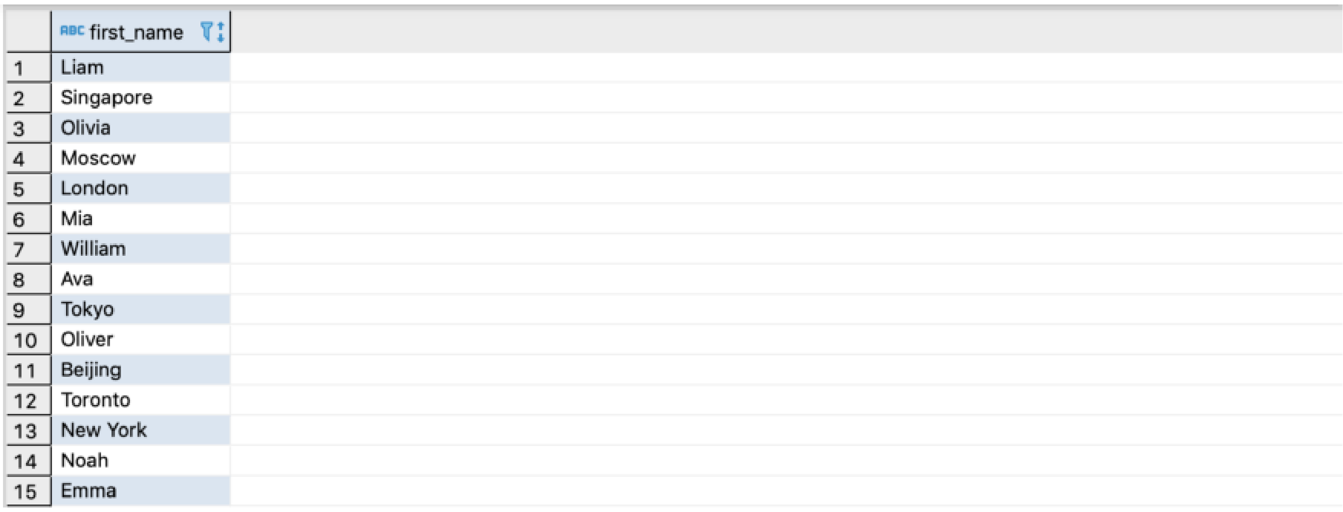
```
select
  first_name
from users
intersect
select
  first_name
from mailing_lists;
```



	first_name
1	Liam
2	Ava

Union

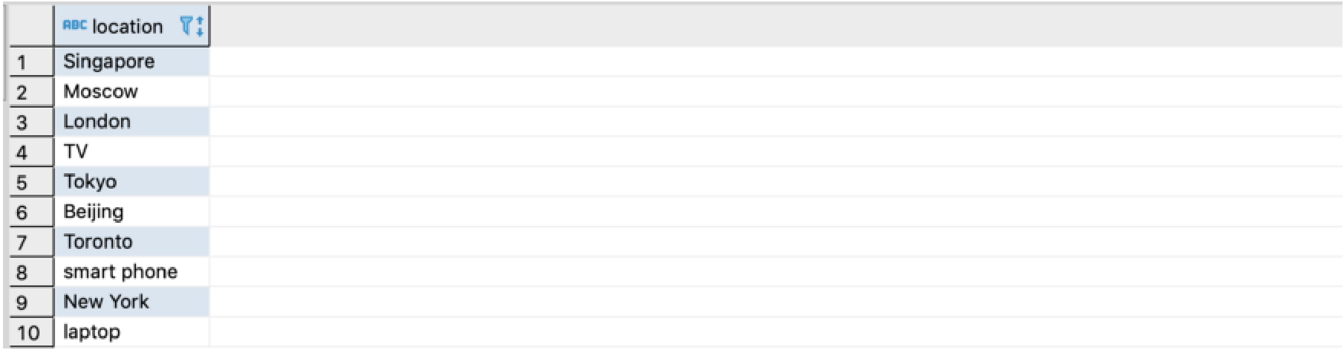
Allows you to return data from different columns, in the same column. Notice how `first_name` is a mix of user names and locations.



	first_name
1	Liam
2	Singapore
3	Olivia
4	Moscow
5	London
6	Mia
7	William
8	Ava
9	Tokyo
10	Oliver
11	Beijing
12	Toronto
13	New York
14	Noah
15	Emma

We can also stack 2 columns from the same table. Here we have user locations and product names.

```
select location from users
union
select name from products;
```



	location
1	Singapore
2	Moscow
3	London
4	TV
5	Tokyo
6	Beijing
7	Toronto
8	smart phone
9	New York
10	laptop

Union all

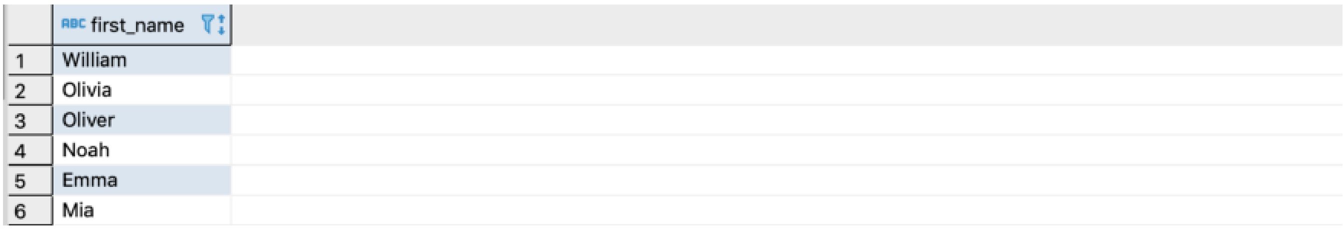
Use `union all` if you don’t want duplicates removed automatically.

```
select name from products
union all
select name from products
```

Except

We can exclude rows that exist in 2 tables, while returning the others. Return all names except those in both `users` and the `mailing_lists`.

```
select
  first_name
from users
except
select
  first_name
from mailing_lists;
```

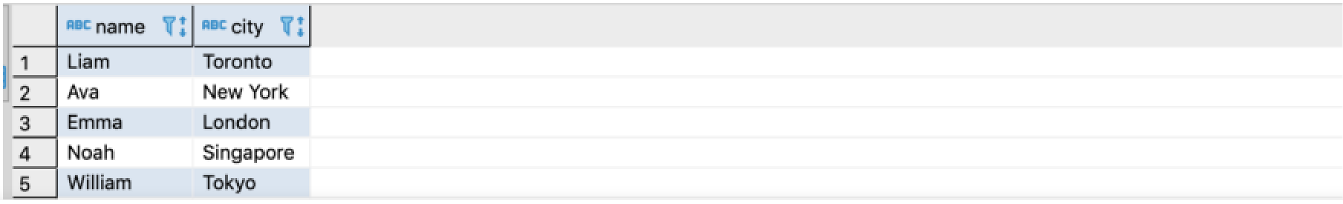


	first_name
1	William
2	Olivia
3	Oliver
4	Noah
5	Emma
6	Mia

Aliasing

Giving a columns an alias changes the headers at the top of returned columns. Notice how the first column's name is now `name` instead of `first_name`. Here we renamed 2 columns.

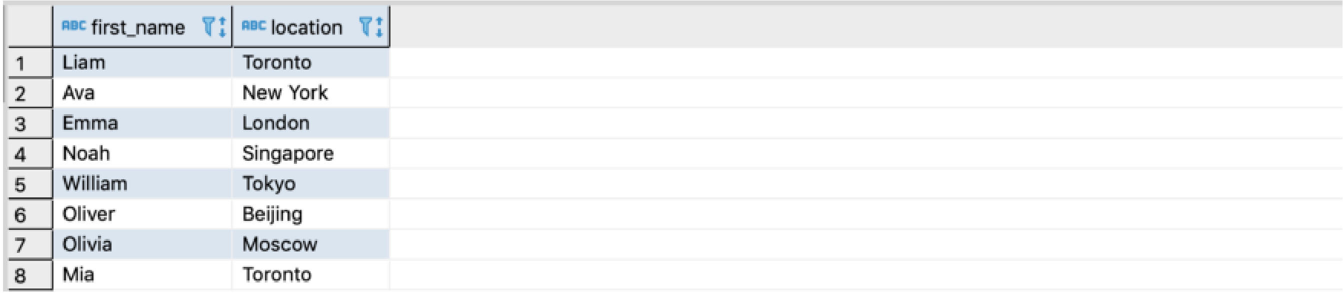
```
select
  first_name as name,
  location as city
from users;
```



	name	city
1	Liam	Toronto
2	Ava	New York
3	Emma	London
4	Noah	Singapore
5	William	Tokyo

We can also alias tables. We then need to refer to the alias name when selecting out columns. We've renamed `users` as `u`.

```
select
  u.first_name,
  u.location
from users as u;
```



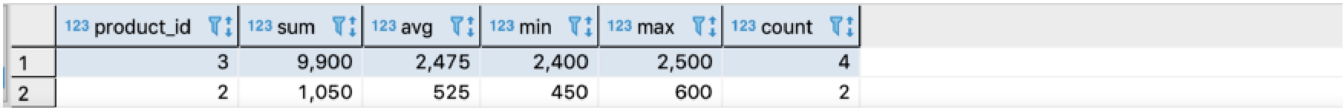
	first_name	location
1	Liam	Toronto
2	Ava	New York
3	Emma	London
4	Noah	Singapore
5	William	Tokyo
6	Oliver	Beijing
7	Olivia	Moscow
8	Mia	Toronto

Aggregating Data

Grouping and aggregating data is a pretty powerful feature. Postgres provides the standard functions like: `sum()`, `avg()`, `min()`, `max()` and `count()`.

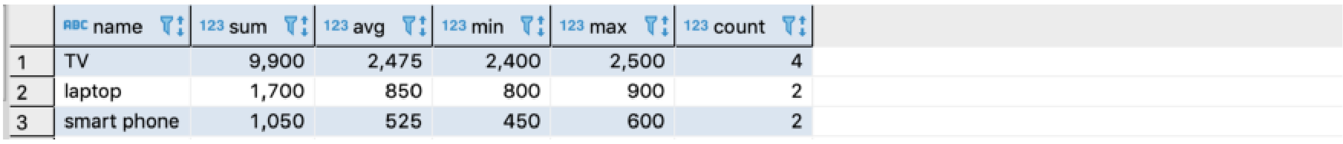
Here we'll calculate sum, avg, min and max of sale prices, on a per-product level.

```
select
  product_id,
  sum(sale_price),
  avg(sale_price),
  min(sale_price),
  max(sale_price),
  count(id)
from sales group by product_id;
```



	product_id	sum	avg	min	max	count
1	3	9,900	2,475	2,400	2,500	4
2	2	1,050	525	450	600	2

```
select
  products.name,
  sum(sale_price),
  avg(sale_price),
  min(sale_price),
  max(sale_price),
  count(sales.id)
from sales
left join products on products.id = sales.product_id
group by products.name;
```



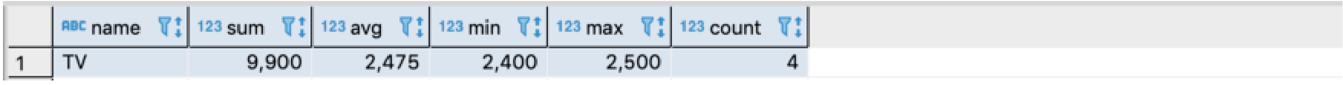
	name	sum	avg	min	max	count
1	TV	9,900	2,475	2,400	2,500	4
2	laptop	1,700	850	800	900	2
3	smart phone	1,050	525	450	600	2

Group by having

This allows filtering on grouped and aggregated data. Regular where clauses won't work here but we can use `having` instead.

Only return aggregated data for products which sold more than 2 items.

```
select
  products.name,
  sum(sale_price),
  avg(sale_price),
  min(sale_price),
  max(sale_price),
  count(sales.id)
from sales
left join products on products.id = sales.product_id
group by products.name
having count(sales.id) > 2;
```

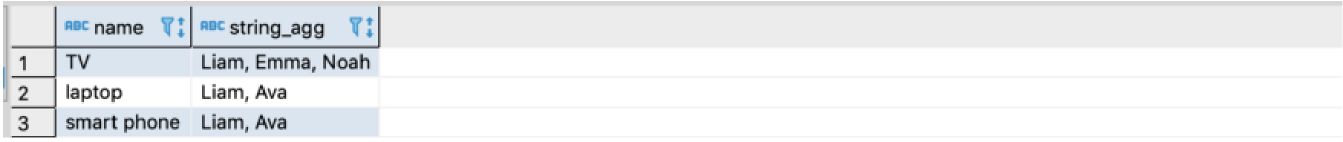


	name	sum	avg	min	max	count
1	TV	9,900	2,475	2,400	2,500	4

String_agg

Can also use `agg` functions (like `string_agg`) in combination with `group by` to build a

```
select
 products.name,
 string_agg(users.first_name, ‘, ‘)
from products
left join sales on sales.product_id = products.id
left join users on users.id = sales.user_id
group by products.name;
```



	name	string_agg
1	TV	Liam, Emma, Noah
2	laptop	Liam, Ava
3	smart phone	Liam, Ava

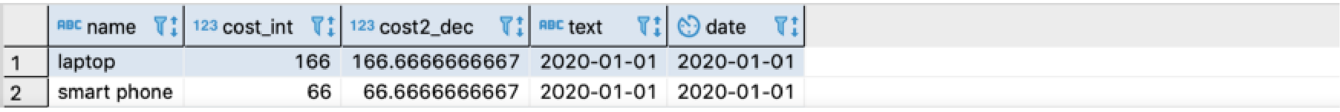
Modifying Selected Values

Casting

Casting means converting the type of data in a column. Not all data can be converted to all datatypes. For instance, trying to cast a string to an integer would throw an error.

But casting an integer to a decimal would work. We do this below so we can see decimals after dividing manufacturing cost by 3 (an arbitrary decision). Notice how when we divide an integer, we don’t get decimals places, but when we divide a decimal, we do.

```
select
  name,
  manufacturing_cost / 3 cost_int,
  manufacturing_cost::decimal / 3 as cost2_dec,
  '2020-01-01'::text,
  '2020-01-01'::date
from products;
```

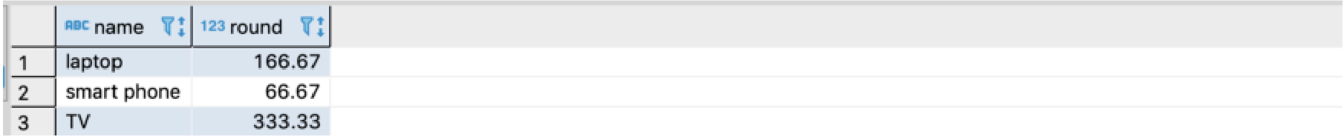


	name	cost_int	cost2_dec	text	date
1	laptop	166	166.6666666667	2020-01-01	2020-01-01
2	smart phone	66	66.6666666667	2020-01-01	2020-01-01

We can also round to a specified number of decimals. Sometimes we don't want a 10 decimal places.

This is a modified version of the above query with an added `round()` .

```
select
  name,
  round(
    manufacturing_cost::decimal / 3, 2
  )
from products;
```

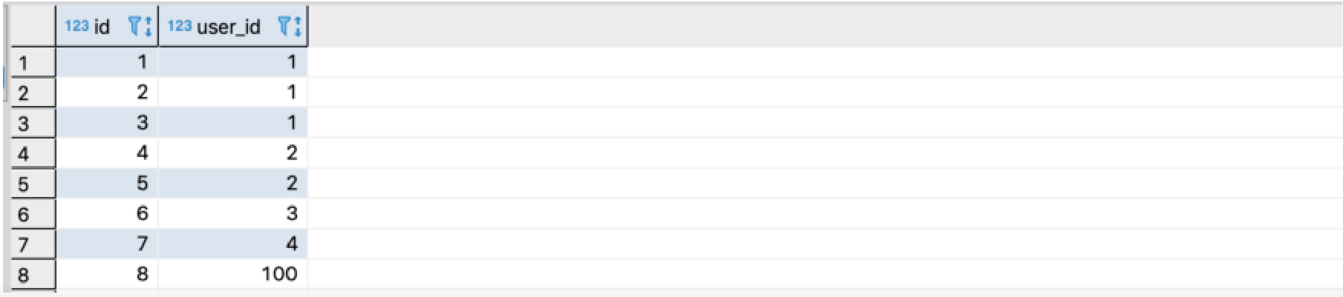


	name	round
1	laptop	166.67
2	smart phone	66.67
3	TV	333.33

Case

Case allows conditionally applying logic or returning a different values based on a cell's value. It's SQL's equivalent of `if/else` . Here we return the value `100` for cells where user_id is `null` .

```
select
  id,
  case
    when user_id is null then 100
    else user_id
  end
from sales;
```

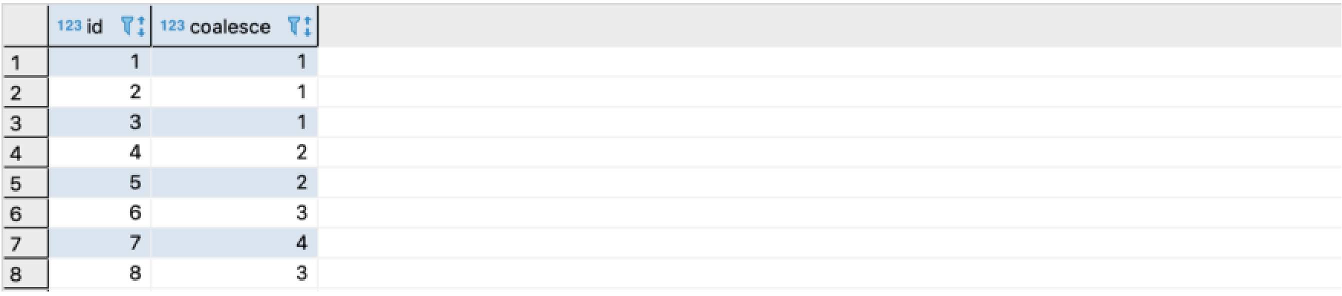


	id	user_id
1	1	1
2	2	1
3	3	1
4	4	2
5	5	2
6	6	3
7	7	4
8	8	100

Coalesce allows returning the value from a different column if the first column’s value is null.

Useful is data is really sparse or spread across multiple columns.

```
select
  id,
  coalesce(user_id, product_id)
from sales;
```

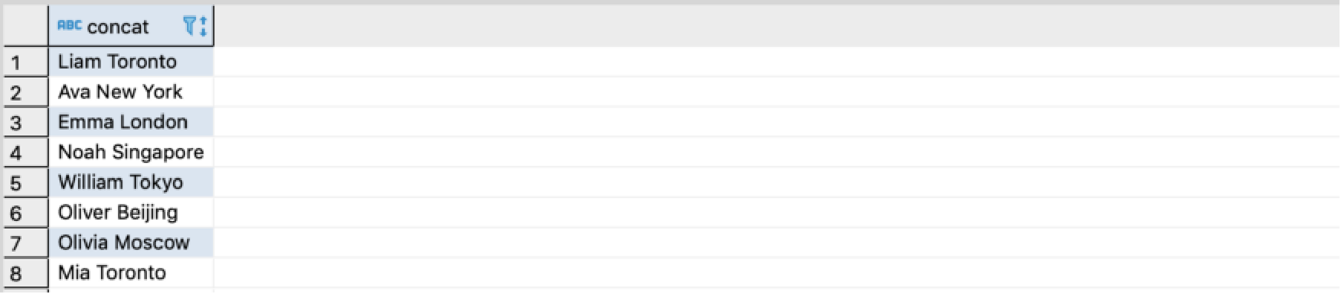


	id	coalesce
1	1	1
2	2	1
3	3	1
4	4	2
5	5	2
6	6	3
7	7	4
8	8	3

Concat

Concat simply concatenates strings. Here we concatenate names and locations.

```
select
 concat(first_name, ‘ ’, location)
from users;
```

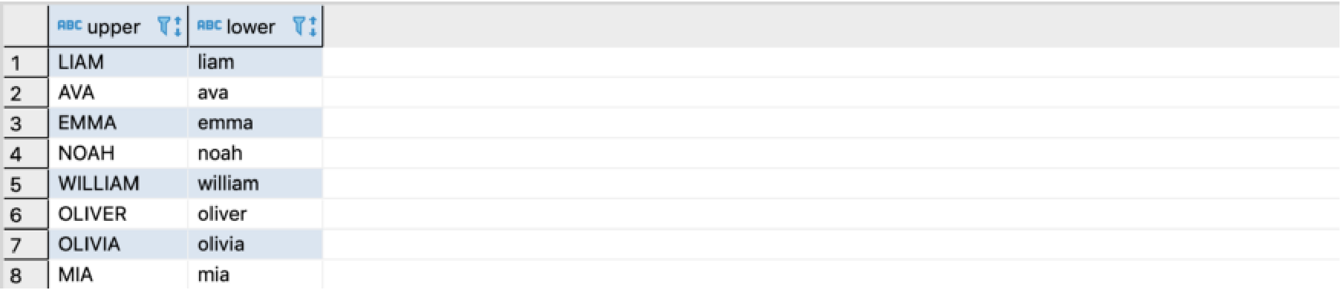


	concat
1	Liam Toronto
2	Ava New York
3	Emma London
4	Noah Singapore
5	William Tokyo
6	Oliver Beijing
7	Olivia Moscow
8	Mia Toronto

Upper and lower

Changes the case of a string.

```
select
  upper(first_name),
  lower(first_name)
from users;
```



	upper	lower
1	LIAM	liam
2	AVA	ava
3	EMMA	emma
4	NOAH	noah
5	WILLIAM	william
6	OLIVER	oliver
7	OLIVIA	olivia
8	MIA	mia

Where Clauses

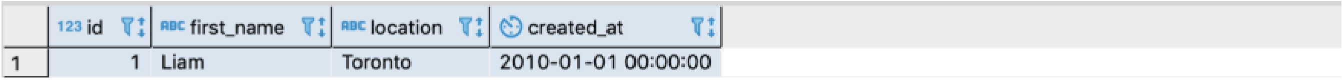
The big section.

Operators

We can use all the equality operators you'd expect in a where clause: `=`, `<>`, `!=`, `<`, `<=`, `>=`, `>`.

Find all records where the name is exactly "Liam".

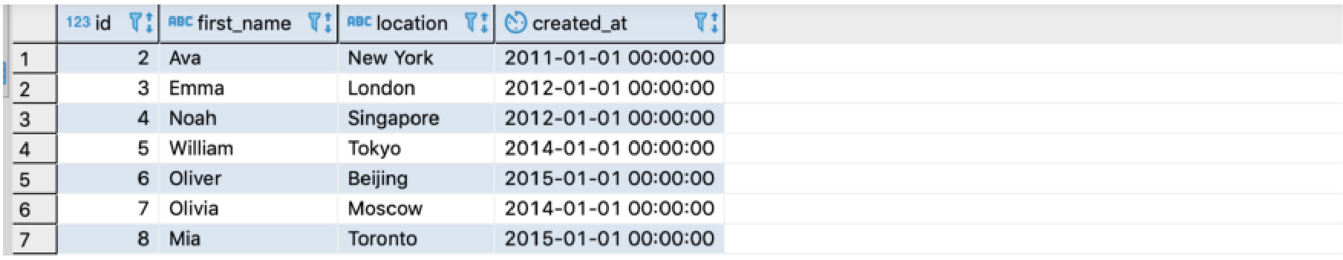
```
select * from users where first_name = 'Liam'
```



	id	first_name	location	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00

Find all records where the name is not "Liam".

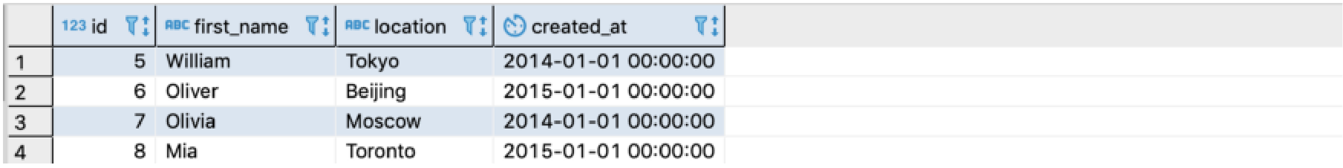
```
select * from users where first_name != 'Liam'
```



	id	first_name	location	created_at
1	2	Ava	New York	2011-01-01 00:00:00
2	3	Emma	London	2012-01-01 00:00:00
3	4	Noah	Singapore	2012-01-01 00:00:00
4	5	William	Tokyo	2014-01-01 00:00:00
5	6	Oliver	Beijing	2015-01-01 00:00:00
6	7	Olivia	Moscow	2014-01-01 00:00:00
7	8	Mia	Toronto	2015-01-01 00:00:00

Find all records where `id` is greater or equal to 5.

```
select * from users where id >= 5
```



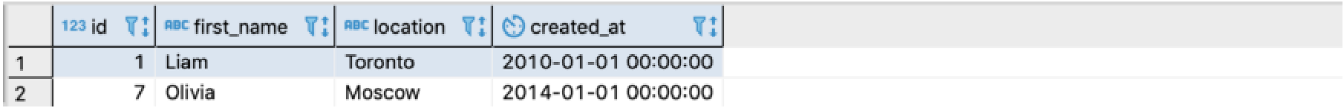
	id	first_name	location	created_at
1	5	William	Tokyo	2014-01-01 00:00:00
2	6	Oliver	Beijing	2015-01-01 00:00:00
3	7	Olivia	Moscow	2014-01-01 00:00:00
4	8	Mia	Toronto	2015-01-01 00:00:00

And, Or, Not

Chain multiple where clauses together with `and` , `or` and `not` . But notice we only write the `where` word once.

Select all records where the name is exactly “Liam” or “Olivia”.

```
select * from users where first_name = 'Liam' or first_name =
'Olivia';
```



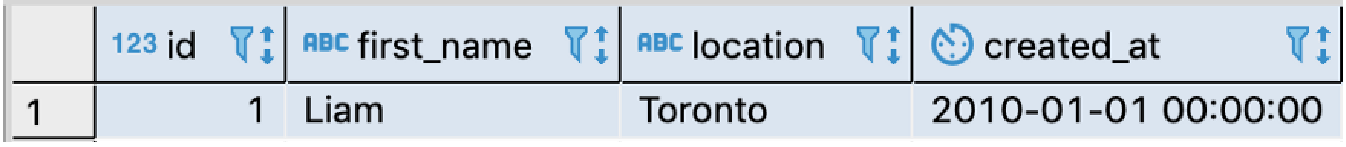
	id	first_name	location	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00
2	7	Olivia	Moscow	2014-01-01 00:00:00

Select all records where the name is exactly “Liam” AND the id is 5. This returns none because Liam’s id is not 5.

```
select * from users where first_name = 'Liam' and id = 5;
```

Select all records where the name is “Liam” AND the id is NOT 5. This returns Liam now.

```
select * from users where first_name = 'Liam' and not id = 5;
```

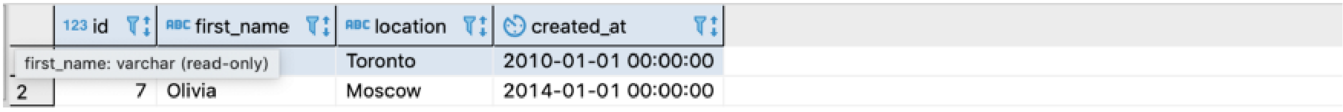


	123 id	ABC first_name	ABC location	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00

In

Rather than chaining clauses with or, or, or… you can find records where a value exists in a given array.

```
select * from users where first_name in ('Liam', 'Olivia');
```

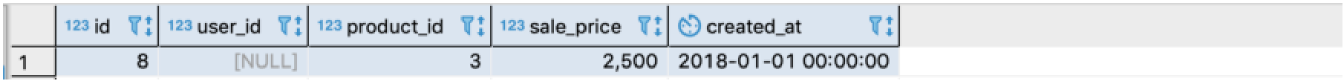


	123 id	ABC first_name	ABC location	created_at
			Toronto	2010-01-01 00:00:00
2	7	Olivia	Moscow	2014-01-01 00:00:00

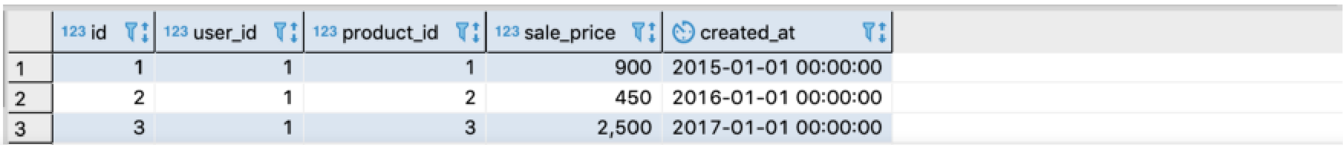
Null

We can also load records where a value is (or is not) null.

```
select * from sales where user_id is null;
select * from sales where user_id is not null;
```



	123 id	123 user_id	123 product_id	123 sale_price	created_at
1	8	[NULL]	3	2,500	2018-01-01 00:00:00



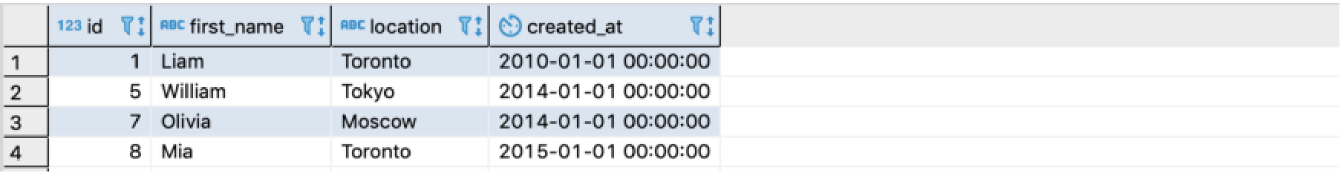
	123 id	123 user_id	123 product_id	123 sale_price	created_at
1	1	1	1	900	2015-01-01 00:00:00
2	2	1	2	450	2016-01-01 00:00:00
3	3	1	3	2,500	2017-01-01 00:00:00

Fuzzy matching

Sometimes we want to find values that roughly match a query. For this, we can search on partial strings or ignore capitalization.

Load any records with the characters “ia” in the name.

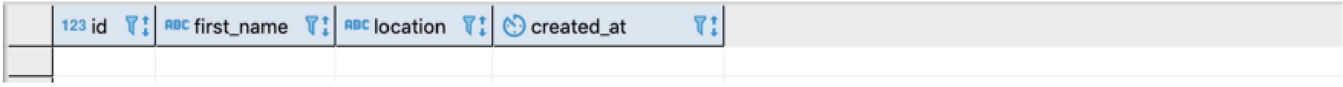
```
select * from users where first_name like '%ia%';
```



	123 id	ABC first_name	ABC location	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00
2	5	William	Tokyo	2014-01-01 00:00:00
3	7	Olivia	Moscow	2014-01-01 00:00:00
4	8	Mia	Toronto	2015-01-01 00:00:00

Load records with the characters “IA” in the name. This returns nothing because no names have capitalized “IA” in them.

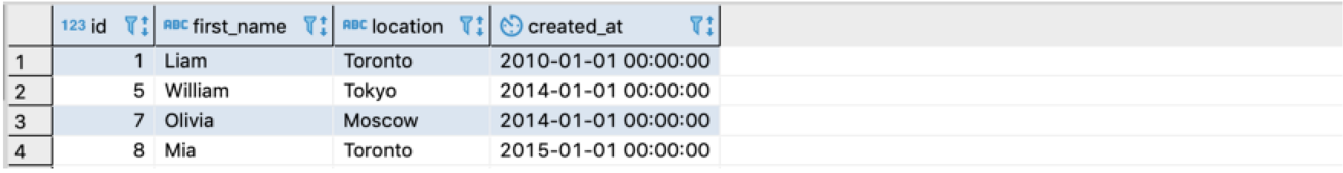
```
select * from users where first_name like '%IA%';
```



	123 id	ABC first_name	ABC location	created_at

So let’s do a search ignoring cases.

```
select * from users where first_name ilike ‘%IA%’;
```



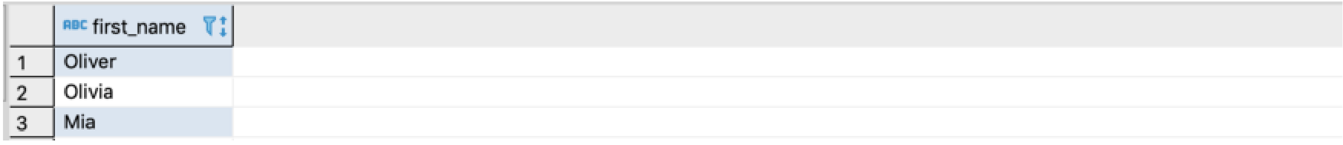
	123 id	ABC first_name	ABC location	created_at
1	1	Liam	Toronto	2010-01-01 00:00:00
2	5	William	Tokyo	2014-01-01 00:00:00
3	7	Olivia	Moscow	2014-01-01 00:00:00
4	8	Mia	Toronto	2015-01-01 00:00:00

Where in subqueries

We already know we can do this

But we can also select from this query! Note you need to provide an alias for a subquery to work or an error will be thrown.

```
select
  first_name
from (
  select * from users where id > 5
) subquery;
```

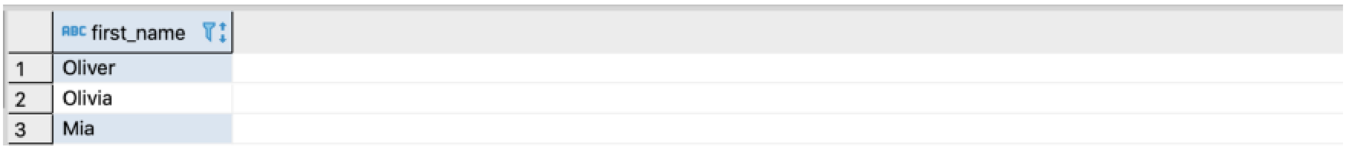


	first_name
1	Oliver
2	Olivia
3	Mia

With

Although we can query from another query, I prefer this approach. It feels much cleaner to define the subqueries in advance.

```
with cte as (
  select * from users where id > 5
)
select
  first_name
from cte
```



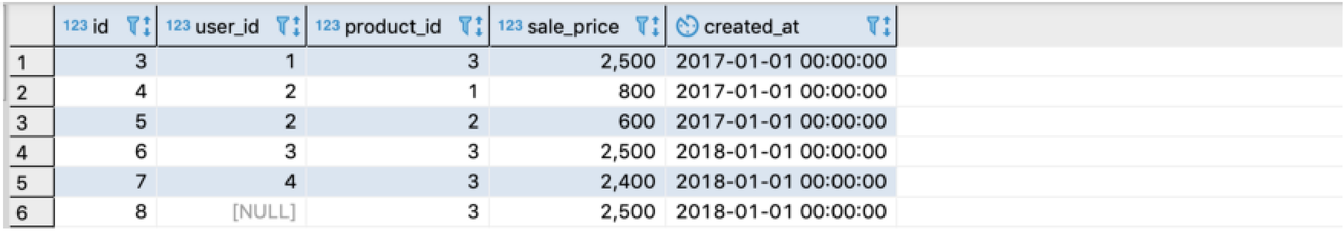
	first_name
1	Oliver
2	Olivia
3	Mia

Date filtering

We can filter by dates.

Useful if you want to find all the transactions that occurred after a specific date.

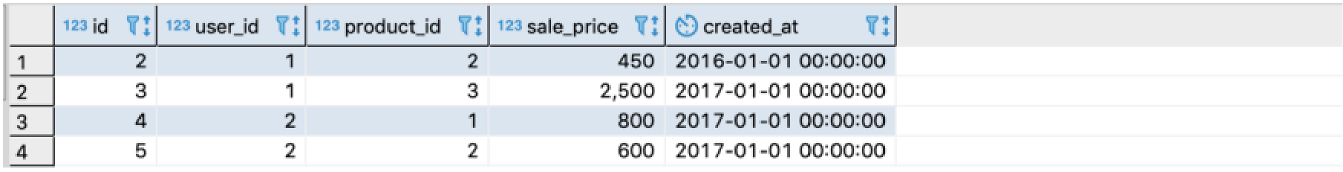
```
select * from sales where created_at > '2016-01-01';
```



	id	user_id	product_id	sale_price	created_at
1	3	1	3	2,500	2017-01-01 00:00:00
2	4	2	1	800	2017-01-01 00:00:00
3	5	2	2	600	2017-01-01 00:00:00
4	6	3	3	2,500	2018-01-01 00:00:00
5	7	4	3	2,400	2018-01-01 00:00:00
6	8	[NULL]	3	2,500	2018-01-01 00:00:00

We can also find transactions between 2 dates.

```
select
  *
from sales
where created_at between '2016-01-01' and '2017-01-01';
```



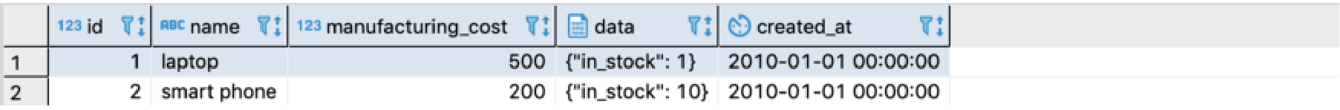
	id	user_id	product_id	sale_price	created_at
1	2	1	2	450	2016-01-01 00:00:00
2	3	1	3	2,500	2017-01-01 00:00:00
3	4	2	1	800	2017-01-01 00:00:00
4	5	2	2	600	2017-01-01 00:00:00

JSON(B)s

Postgres has some pretty awesome functionality for working with JSON.

Find records that have the key, `in_stock` in the `data` column.

```
select * from products where data -> 'in_stock' is not null;
```



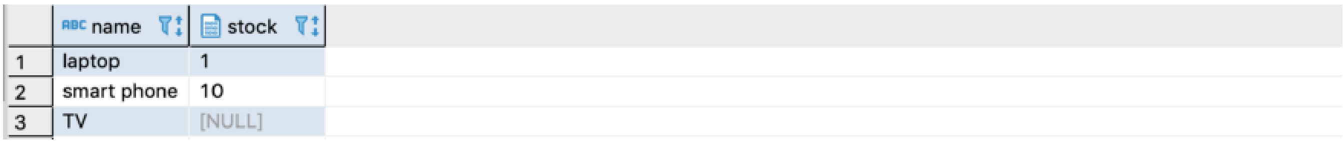
	id	name	manufacturing_cost	data	created_at
1	1	laptop	500	{"in_stock": 1}	2010-01-01 00:00:00
2	2	smart phone	200	{"in_stock": 10}	2010-01-01 00:00:00

Find records where the value of `in_stock` is greater than 5. Notice we need to cast JSONB to an integer to do the comparison.

```
select * from products where (data -> 'in_stock')::int > 5;
```

Select out data from the JSONB, as JSONB.

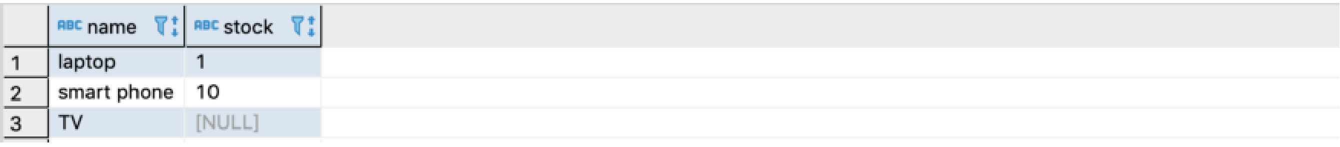
```
select name, data -> 'in_stock' as stock from products;
```



	name	stock
1	laptop	1
2	smart phone	10
3	TV	[NULL]

Select it out as text. The data type can have an impact in a more complex query where this value has other functions run on it.

```
select name, data ->> 'in_stock' as stock from products;
```



	name	stock
1	laptop	1
2	smart phone	10
3	TV	[NULL]

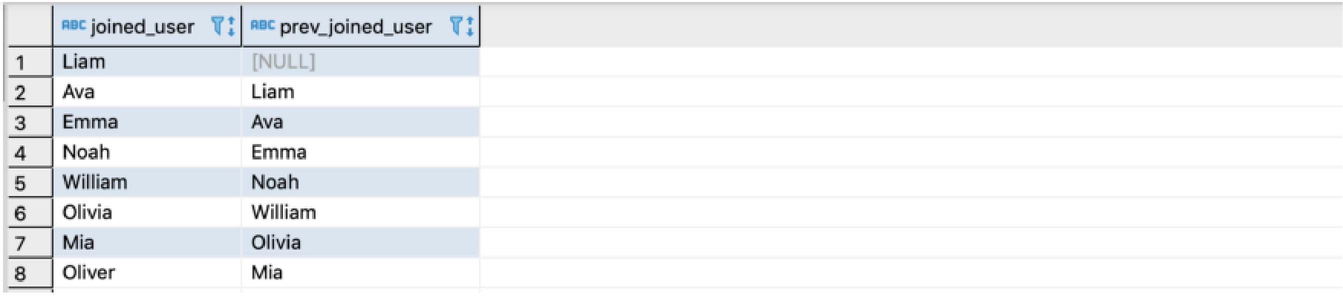
Lag

Get a record in the table and attach the record immediately before it.

Useful when looking at events over time where previous events affect future events. You might use data queried like this to train an ML model to predict a future state given the current state.

We'll use it to find the user added immediately before every other user.

```
select
 first_name as joined_user,
 lag(first_name) over (order by created_at) as prev_joined_user
from users;
```



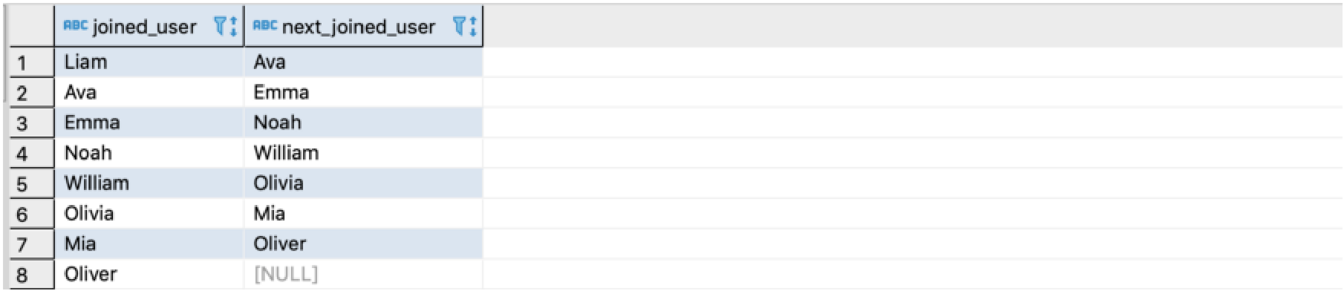
	joined_user	prev_joined_user
1	Liam	[NULL]
2	Ava	Liam
3	Emma	Ava
4	Noah	Emma
5	William	Noah
6	Olivia	William
7	Mia	Olivia
8	Oliver	Mia

We get `null` as the previous user for Liam because he was the first to be added to the database.

Lead

The opposite of above. Load the user that joined immediately after each other user.

```
select
 first_name as joined_user,
 lead(first_name) over (order by created_at) as next_joined_user
from users;
```



	joined_user	next_joined_user
1	Liam	Ava
2	Ava	Emma
3	Emma	Noah
4	Noah	William
5	William	Olivia
6	Olivia	Mia
7	Mia	Oliver
8	Oliver	[NULL]

Conclusion

Postgres can do a million things. So even though there's a lot of commands here, we're only covering the basics.

While it's not necessary to know these by heart to be effective, understanding how they work is almost a requisite for building advanced queries and knowing what's possible.

Are there any Postgres commands that were a game changer for you?