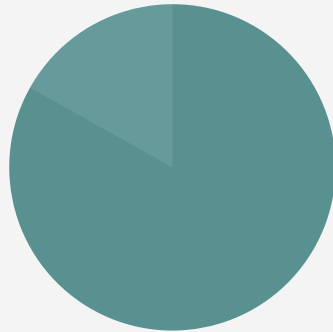


# SQL Next Steps: Optimization

Getting the most out of your database

By **Haki Benita**





# About Me

## Haki Benita

- DBA (Oracle and PostgreSQL)
- Full Stack Developer (Python / Django / Javascript)
- Team leader
- Data plumber
- Website: <https://hakibenita.com/>
- Twitter: [@haki\\_be](https://twitter.com/haki_be)





# What You'll Learn

- How to avoid common mistakes in SQL
- How to improve performance of SQL
- How to be more productive writing SQL



# SQL History

- SQL = **S**tructured **Q**uery **L**anguage
- Pronounced **S - Q - L** (not SEQUEL!)
- Used to interact with **relational databases** (RDBMS)
- Invented in the early 70s at IBM based on work by Edgar F. Codd
- Became a standard in 1986 (ANSI-86)
- Standard revised in 1992 (ANSI-92)



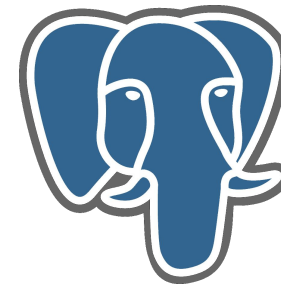
[Important Papers: Codd and the Relational Model](#)



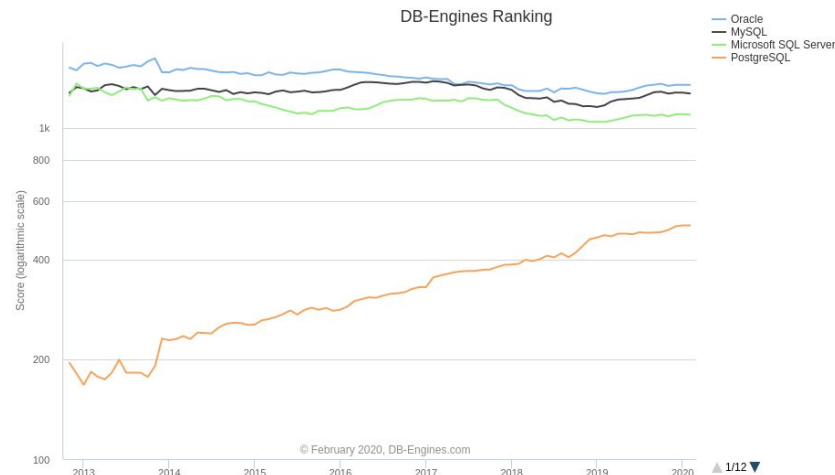
[Comparing Database Types: How Database Types Evolved to Meet Different Needs](#)



# PostgreSQL



- Based on the Berkeley POSTGRES project from 1986
- Released under the name Postgres95
- Open Source
- Free
- Growing fast!



Source: [https://db-engines.com/en/ranking\\_trend](https://db-engines.com/en/ranking_trend)



# Anatomy of an SQL Query

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

LIMIT

**[Q]** In which order are the parts of the query executed?



# Anatomy of an SQL Query

In order of execution

FROM

WHERE

SELECT

GROUP BY

HAVING

ORDER BY

LIMIT






## Question

What's wrong with this query?

```
SELECT
    department,
    count(*) as number_of_employees
FROM
    employees
WHERE
    number_of_employees > 10
GROUP BY
    department
```

Order of execution:

FROM  
WHERE  
SELECT  
GROUP BY  
HAVING  
ORDER BY  
LIMIT

A vertical line with a downward-pointing arrowhead, indicating the sequence of execution from top to bottom.




## Question

What's wrong with this query?

```
SELECT
    department,
    count(*) as number_of_employees
FROM
    employees
WHERE
    number_of_employees > 10
GROUP BY
    department
```

```
SELECT
    department,
    count(*) as number_of_employees
FROM
    employees
GROUP BY
    department
HAVING
    number_of_employees > 10
```



### Aggregate results can only be used in the HAVING clause

When conditions in the WHERE clause are being evaluated, the result of the aggregation function is not yet available.




## Question

What's wrong with this query?

```
SELECT
  count(*) as number_of_employees
FROM
  employees
LIMIT
  1
```

Order of execution:

FROM  
WHERE  
SELECT  
GROUP BY  
HAVING  
ORDER BY

A vertical line with a downward-pointing arrow at the bottom, indicating the sequence of execution for the clauses listed to its right.



## Question

What's wrong with this query?

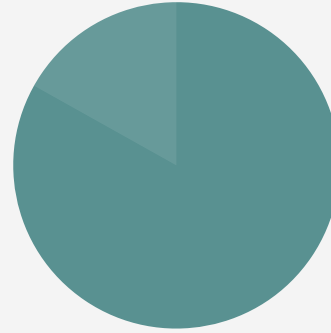
```
SELECT
  count(*) as number_of_employees
FROM
  employees
LIMIT
  1
```

**Limit has no effect**

LIMIT is executed on the aggregated results, so it has not effect in this case (if might cause confusion, so better to omit!)

# Common Mistakes In SQL

Writing correct SQL





# Sales Database Setup

- PostgreSQL 12
- Use [DB Fiddle](#)
- Use local database
- Follow along with presentation



# Sales Database

```
select * from sale
```

id	branch	sold_at	customer	product	price	discount
1	NY	2020-03-31 20:15:00-07	Bill	Shoes	10000	1000
2	NY	2020-03-31 21:00:00-07		Shoes	5000	0
3	LA	2020-03-31 23:15:00-07	Lily	Shoes	15000	0
4	LA	2020-04-01 02:10:00-07	John	Shoes	5000	2500
5	NY	2020-03-31 20:15:00-07		Shirt	1500	0
6	NY	2020-03-31 19:07:00-07	John	Shirt	1850	0
7	LA	2020-03-31 02:55:00-07	Bill	Shirt	1250	0
8	LA	2020-03-31 03:45:00-07	Lily	Shirt	1850	100
9	NY	2020-03-31 00:45:00-07	Lily	Pants	5200	0
10	LA	2020-03-31 03:45:00-07	John	Pants	5200	0
11	LA	2020-04-01 00:01:00-07	David	Pants	4500	0
12	LA	2020-04-01 23:01:00-07		Hat	8000	8000
13	LA	2020-04-01 23:01:00-07	Bill	Give Away	0	0
14	NY	2020-03-31 10:01:00-07		Give Away	0	0
15	LA	2020-04-01 03:45:00-07		Give Away	0	0



## Exercise

What is the discount rate on **Shoes**?

```
select * from sale where product = 'Shoes';
```

id	branch	sold_at	customer	product	price	discount
1	NY	2020-04-01 03:15:00+00	Bill	Shoes	10000	1000
2	NY	2020-04-01 04:00:00+00		Shoes	5000	0
3	LA	2020-04-01 06:15:00+00	Lily	Shoes	15000	0
4	LA	2020-04-01 09:10:00+00	John	Shoes	5000	2500

10%





## Exercise

What is the discount rate on **Shoes**?

```
SELECT price, discount, discount / price * 100 as discount_rate
FROM sale
WHERE product = 'Shoes';
```

price	discount	discount_rate
10000	1000	0
5000	0	0
15000	0	0
5000	2500	0

How is this possible?

# Be Careful When Dividing Integers

## Integer division truncates the result

```
SELECT 1000 / 10000;  
?column?  
-----  
0  
  
SELECT 1000 / 10000::float;  
?column?  
-----  
0.1  
  
SELECT 1000 / 10000::float * 100;  
?column?  
-----  
10
```

Casting the denominator to float produces the expected result



[Mathematical Functions and Operators](#)



## Exercise

# What is the discount rate on **Shoes**?

Put it together:

```
SELECT price, discount, discount / price::float as discount_rate
FROM sale
WHERE product = 'Shoes';
```

price	discount	discount_rate
10000	1000	0.1
5000	0	0
15000	0	0
5000	2500	0.5



## Exercise

# What is the discount rate on **Shoes**?

Multiple by 100:

```
SELECT price, discount, discount / price::float * 100 as discount_rate
FROM sale
WHERE product = 'Shoes';
```

price		discount		discount_rate
-----	+	-----	+	-----
10000		1000		10
5000		0		0
15000		0		0
5000		2500		50



## Exercise

Find the **average** discount rate **by product**



## Exercise

Find the **average** discount rate **by product**

```
SELECT
    product,
    AVG(discount / price::float) * 100 as discount_rate
FROM sale
GROUP BY product;
```

```
ERROR:  division by zero
```



# Guard Against "division by zero" Errors

```
SELECT id, product, price, discount
FROM sale
WHERE price = 0;
```

id	product	price	discount
13	Give Away	0	0
14	Give Away	0	0
15	Give Away	0	0

Product "Give Away" price is zero and it causes the division to fail

# Guard Against "division by zero" Errors

**NULLIF(value1, value2)**

The NULLIF function returns a null value if value1 equals value2; otherwise it returns value1.

 [NULLIF Documentation](#)

The adjusted price is NULL when the price equals zero

```
SELECT
    id, product, price, discount,
    NULLIF(price, 0) AS adjusted_price
FROM sale
WHERE price = 0;
```

id	product	price	discount	adjusted_price
13	Give Away	0	0	
14	Give Away	0	0	
15	Give Away	0	0	



# Dividing by null produces null, not an error

No error!

```
SELECT
  id,
  product,
  discount / NULLIF(price, 0) AS discount_rate
FROM sale
WHERE price = 0;
```

id	product	discount_rate
----	---------	---------------

13	Give Away	
14	Give Away	
15	Give Away	

Result of division  
is now NULL.

## Exercise

Find the average discount rate **by product**

```
SELECT
    product,
    AVG(discount / NULLIF(price, 0)::float) * 100 AS discount_rate
FROM sale
GROUP BY product;
```

product	discount_rate
Shirt	1.3513513513513513
Pants	0
Hat	100
Shoes	15
Give Away	

What is the real discount here?



## Exercise

# Find the average discount rate **by product**

**COALESCE**(*value1*, ..., *valueN*)

Returns the first of its arguments  
that is not NULL



[COALESCE Documentation](#)

Make discount rate zero for  
products with price zero

```
SELECT
  product,
  COALESCE(AVG(discount / NULLIF(price, 0)::float), 0) * 100
FROM sale
GROUP BY product;
```

product	discount_rate
Shirt	1.3513513513513513
Pants	0
Hat	100
Shoes	15
Give Away	0



## Exercise

How many **unique users** purchased **each product**?

## Exercise

How many unique users purchased each product?

Is this correct?

```
SELECT
    product,
    COUNT(DISTINCT customer) AS customers
FROM sale
GROUP BY product;
```

product	customers
Give Away	1
Hat	0
Pants	3
Shirt	3
Shoes	3



# Be Careful When Aggregating Nullable Column

Aggregate functions ignore null values!

```
SELECT * FROM sale WHERE product = 'Hat';
```

id	sold_at	customer	product	price	discount
12	2020-04-01 18:01:00+03		Hat	8000	8000

Why was this customer not counted?

# Be Careful When Aggregating Nullable Column

Aggregate functions  
ignore NULL values!

This can also  
be useful!

```
SELECT
  product,
  COUNT(*) AS cnt,
  COUNT(customer) AS cnt_customer
FROM sale
GROUP BY product;
```

product	cnt	cnt_customer
Shirt	4	3
Pants	3	3
Hat	1	0
Shoes	4	3
Give Away	3	1



## Exercise

How many **known customers** purchased each product?



Hint: Use the fact that aggregate functions ignore null!





## Exercise

How many **known customers** purchased each product?

```
SELECT
    product,
    COUNT(customer) as known_customers,
    COUNT(*) - COUNT(customer) as unknown_customers
FROM sale
GROUP BY product;
```

product	known_customers	unknown_customers
Shirt	3	1
Pants	3	0
Hat	0	1
Shoes	3	1
Give Away	1	2



## Exercise

Write a query to find the sales made by the customer **Bill**



## Exercise

Write a query to find the sales made by the customer **Bill**

```
SELECT * FROM sale WHERE customer = 'Bill';
```

id	branch	sold_at	customer	product	price	discount
1	NY	2020-03-31 20:15:00-07	Bill	Shoes	10000	1000
7	LA	2020-03-31 02:55:00-07	Bill	Shirt	1250	0
13	LA	2020-04-01 23:01:00-07	Bill	Give Away	0	0



## Exercise

Write a query to find the sales made by an **unknown customer**



## Exercise

Write a query to find the sales made by an **unknown customer**

```
SELECT * FROM sale WHERE customer IS NULL;
```

id	branch	sold_at	customer	product	price	discount
2	NY	2020-03-31 21:00:00-07		Shoes	5000	0
5	NY	2020-03-31 20:15:00-07		Shirt	1500	0
12	LA	2020-04-01 23:01:00-07		Hat	8000	8000
14	NY	2020-03-31 10:01:00-07		Give Away	0	0
15	LA	2020-04-01 03:45:00-07		Give Away	0	0



## Exercise

Write a query to find the sales made by a customer, **using a parameter**

```
SELECT * FROM sale WHERE customer = :name;
```

Using psql \set command to set the values of this parameter

 [psql Documentation](#)



Very common when writing queries using a reporting tool



## Exercise

Write a query to find the sales made by a customer, **using a parameter**

```
\set name '\''Bill'\''
```

```
SELECT * FROM sale WHERE customer = :name;
```

id	branch	sold_at	customer	product	price	discount
1	NY	2020-03-31 20:15:00-07	Bill	Shoes	10000	1000
7	LA	2020-03-31 02:55:00-07	Bill	Shirt	1250	0
13	LA	2020-04-01 23:01:00-07	Bill	Give Away	0	0



## Exercise

Write a query to find the sales made by a customer, **using a parameter**

```
\set name null
```

```
SELECT * FROM sale WHERE customer = :name;
```

```
id | branch | sold_at | customer | product | price | discount
```

```
---+-----+-----+-----+-----+-----+-----
```

← Is this correct?





# Comparing NULL values

To compare null, we need to use **IS**

```
SELECT NULL IS NULL;
```

```
?column?
```

```
-----
```

```
t
```

```
SELECT NULL = NULL;
```

```
?column?
```

```
-----
```

```
SELECT (NULL = NULL) IS NULL;
```

```
?column?
```

```
-----
```

```
t
```



# Comparing NULL values

How to compare both null and literal values?

```
SELECT *  
FROM sale  
WHERE
```

```
(:name IS NULL AND customer IS NULL)
```

Handle NULL case

```
OR
```

```
(:name IS NOT NULL AND customer = :name);
```

Handle Literal values



# Comparing NULL values

How to compare both null and literal values?

```
\set name '\''Bill'\''
SELECT * FROM sale
WHERE (:name IS NULL AND customer IS NULL)
OR (:name IS NOT NULL AND customer = :name);
```

id	branch	sold_at	customer	product	price	discount
1	NY	2020-03-31 20:15:00-07	Bill	Shoes	10000	1000
7	LA	2020-03-31 02:55:00-07	Bill	Shirt	1250	0
13	LA	2020-04-01 23:01:00-07	Bill	Give Away	0	0



# Comparing NULL values

How to compare both null and literal values?

```
\set name NULL
SELECT * FROM sale
WHERE (:name IS NULL AND customer IS NULL)
OR (:name IS NOT NULL AND customer = :name);
```

id	branch	sold_at	customer	product	price	discount
2	NY	2020-03-31 21:00:00-07		Shoes	5000	0
5	NY	2020-03-31 20:15:00-07		Shirt	1500	0
12	LA	2020-04-01 23:01:00-07		Hat	8000	8000
14	NY	2020-03-31 10:01:00-07		Give Away	0	0
15	LA	2020-04-01 03:45:00-07		Give Away	0	0



# Comparing NULL values

There must be a better way!



There is!



# Comparing NULL values IS DISTINCT FROM

```
SELECT *  
FROM sale  
WHERE customer IS NOT DISTINCT FROM :name;
```

*a IS DISTINCT FROM b*

*a IS NOT DISTINCT FROM b*

Treating null like an ordinary value



[Comparison Functions and Operators](#)



[The Many Faces of DISTINCT in PostgreSQL](#)



## Exercise

Write a query to find the sales made by a customer, **using a parameter**

```
\set name '\''Bill'\''
```

```
SELECT * FROM sale WHERE customer IS NOT DISTINCT FROM :name;
```

id	branch	sold_at	customer	product	price	discount
1	NY	2020-03-31 20:15:00-07	Bill	Shoes	10000	1000
...						

```
\set name NULL
```

```
SELECT * FROM sale WHERE customer IS NOT DISTINCT FROM :name;
```

id	branch	sold_at	customer	product	price	discount
2	NY	2020-03-31 21:00:00-07		Shoes	5000	0
...						



# IS DISTINCT FROM

## All cases

```
WITH t AS (  
    SELECT 1      as a, 1      as b UNION ALL  
    SELECT 1      as a, 2      as b UNION ALL  
    SELECT 1      as a, NULL as b UNION ALL  
    SELECT NULL as a, NULL as b  
)  
SELECT  
    a,  
    b,  
    a = b as equal,  
    a IS NOT DISTINCT FROM b AS is_not_distinct_from  
FROM  
    t;
```

a	b	equal	is_not_distinct_from
1	1	t	t
1	2	f	f
1	NULL	NULL	f
NULL	NULL	NULL	t





# SUMMARY

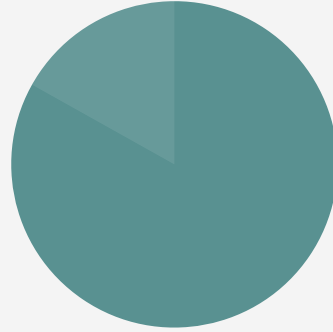
## Mistakes you can now avoid!



- **Be Careful when dividing integers**  
Dividing by an integer truncates the result
- **Guard against "division by zero" errors**  
Use NULLIF and COALESCE
- **Be Careful when aggregating nulls**  
Aggregate functions ignore NULL values
- **Be Careful when comparing nulls**  
Use IS DISTINCT FROM to treat NULL like a value

# Common Mistakes In SQL

Working with dates and times





## Exercise

Find the amount of sales during each month

month		total_sales
2020-03-01		?\$
2020-04-01		?\$



Hint: Use [date\\_trunc](#) to get the month



# Working with dates and times

How is date represented in the database?

- EPOCH: 01/01/1970 00:00:00 UTC
- **Unix time**: Seconds since EPOCH

```
SELECT extract('epoch' FROM now());
```

```
date_part
```

```
-----  
1583585129.063067
```



# Working with dates and times

## The year 2038 problem (Y2k38)

- On 32-bit systems, signed integer can only go as far as 2038-01-19 03:14:07 UTC
- Remember Y2K?

```
SET TIME ZONE UTC;  
SET  
  
SELECT  
  '1970-01-01 UTC'::timestampz  
+ interval '1 second' * (pow(2, 31) - 1) AS y28k;  
  
y28k | 2038-01-19 03:14:07+00
```



[Year 2038 Problem](#)



[IMAGE CREDIT](#)



# Working with dates and times

## Time zones

- **UTC, Coordinated Universal Time:** the primary time standards, not adjusted to daylight savings.
- **GMT, Greenwich Mean Time:** Synonym for UTC (time at Greenwich, England).

```
SELECT  
  now() at time zone 'UTC' as utc,  
  now() at time zone 'GMT' as gmt;
```

```
utc | 2020-03-12 14:07:03.139617
```

```
gmt | 2020-03-12 14:07:03.139617
```



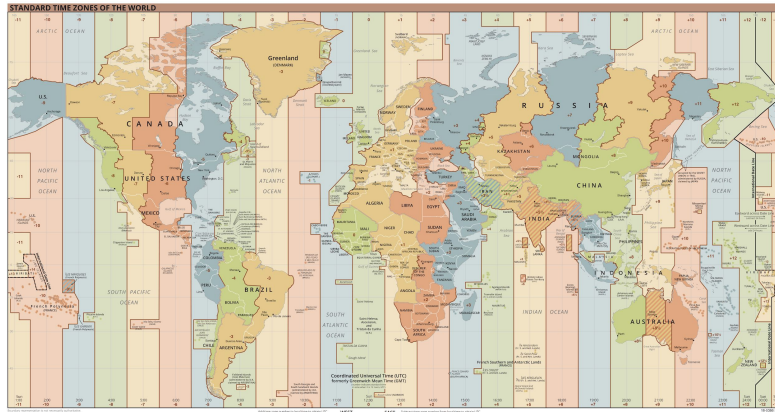
[Timezones: Documentation](#)

# Time Zone

Represented as an offset from UTC

```
select *  
from pg_timezone_names  
where name ~* 'Sydney|Tel_Aviv|Paris|London|New_York'  
and name not like 'posix%'  
order by utc_offset;
```

name	abbrev	utc_offset	is_dst
America/New_York	EST	-05:00:00	f
Europe/London	GMT	00:00:00	f
Europe/Paris	CET	01:00:00	f
Asia/Tel_Aviv	IST	02:00:00	f
Australia/Sydney	AEDT	11:00:00	t



[IMAGE CREDIT](#)



[IANA: Database of all the time zones](#)



## Exercise

Find your time zone



Use the table [pg\\_timezone\\_names](#)





# Working with dates and times

## Using time zones

- Specify the time zone of a timestamp

```
SELECT '2020-03-22 17:00:00 America/New_York'::timestampz;
```

- Convert a timestamp to a different time zone

```
SELECT '2020-03-22 17:00:00 America/New_York'::timestampz AT TIME ZONE 'Australia/Sydney';  
timezone
```

```
-----  
2020-03-23 08:00:00
```

```
SELECT now() AT TIME ZONE 'Australia/Sydney'  
timezone
```

```
-----  
2020-03-08 00:13:37.826917
```



## Exercise

What is the time in **your local time zone**?



Use `NOW()` at time zone 'YOUR LOCAL TIME ZONE NAME'



# Working with dates and times

## Daylight Saving

- **DST, Daylight saving:** Changes to the clock to extend daylight (in warm seasons)
- **For example:** In the US, clock moves on the second Sunday in March (8/3/2020):

```
-- "Standard Time"
SELECT '2020-03-01 12:00:00 America/New_York' AT TIME ZONE 'UTC';
      timezone
-----
2020-03-01 17:00:00

-- "Daylight Time" (AKA "Summer Time"):
SELECT '2020-03-22 12:00:00 America/New_York' AT TIME ZONE 'UTC';
      timezone
-----
2020-03-22 16:00:00
```



# Working with dates and times

## Daylight Saving

- Daylight saving can produce surprising results:

```
select '2020-03-07 12:00:00 America/New_York'::timestampz -  
'2020-03-08 12:00:00 America/New_York'::timestampz as diff;  
diff  
-----  
-23:00:00
```

The day the clock moves there's only 23 hours!



# Working with dates and times

## Database types

- **date**: date without time
- **timestamp**: date and time without time zone
- **timestamp with time zone (timestampz)**: date and time with time zone

```
\d sale
      Table "public.sale"
  Column |          Type
  -----+-----
   id    | integer
  branch | text
  sold_at | timestamp with time zone
customer | text
product  | text
price    | integer
discount | integer
```



[Date/Time Types Documentation](#)



# Working with dates and times

## Interval

```
SELECT '2020-03-22 17:00 UTC'::timestamp + INTERVAL '3 hours';  
2020-03-22 20:00:00+00
```

```
SELECT '2020-03-22 17:00 UTC'::timestamp + INTERVAL '3 hours 2 minutes 55 seconds';  
2020-03-22 20:02:55+00
```

```
SELECT '2020-03-22 17:00 UTC'::timestamp - INTERVAL '1 days' * 3;  
2020-03-19 17:00:00+00
```



[Interval Documentation](#)



# Working with dates and times

## Useful functions

```
SELECT now();  
now | 2020-03-12 14:41:18.910296+00
```

```
SELECT now()::date;  
now | 2020-03-12
```

```
SELECT date_trunc('month', now());  
date_trunc | 2020-03-01 00:00:00+00
```

```
SELECT date_trunc('hour', now());  
date_trunc | 2020-03-12 14:00:00+00
```

```
SELECT date_part('month', now());  
date_part | 3
```

```
SELECT date_part('dow', now());  
date_part | 4
```

```
SELECT extract('month' from now());  
date_part | 3
```

```
SELECT extract('day' from now());  
date_part | 12
```



## Exercise

Find the amount of sales during each month  
*What went wrong...*

```
SET TIME ZONE 'America/New_York';
```

```
SELECT date_trunc('month', sold_at) AS month,  
sum(price) AS total_sales  
FROM sale  
GROUP BY month;
```

month	total_sales
2020-04-01 00:00:00-04	37500
2020-03-01 00:00:00-05	26850



```
SET TIME ZONE 'America/Los_Angeles';
```

```
SELECT date_trunc('month', sold_at) AS month,  
sum(price) AS total_sales  
FROM sale  
GROUP BY month;
```

month	total_sales
2020-04-01 00:00:00-07	17500
2020-03-01 00:00:00-08	46850







## Take Away

Unless explicitly mentioned, timezone is usually set by the client application

```
show time zone;
```

```
      TimeZone
```

```
-----
```

```
America/New_York
```



## Exercise

Find the amount of sales during each month

**\*Assuming that billing for all branches is according to time zone "America/New\_York"**



Hint: Explicitly set the time zone for the sold\_at date.

## Exercise

Find the amount of sales during each month

**\*Assuming that billing for all branches is according to time zone "America/New\_York"**

```
SELECT
    date_trunc('month', sold_at at time zone 'America/New_York') AS month,
    SUM(price) AS total_sales
FROM sale
GROUP BY month;
```

month	total_sales
2020-04-01 00:00:00	37500
2020-03-01 00:00:00	26850

Time zone is  
set explicitly



## Exercise

What is the busiest **hour of the day** in all branches?



Hint: Which time zone should you use to extract the hour?



## Exercise

What is the busiest **hour of the day** in all branches?

```
SELECT extract('hour' FROM sold_at) AS hour_of_day, COUNT(*) AS sales
FROM sale
GROUP BY hour_of_day
ORDER BY sales desc;
```

hour_of_day	sales
-------------	-------

-----+-----	
-------------	--

10	3
----	---

6	3
---	---

9	2
---	---

3	2
---	---

7	2
---	---

...

Is this correct?



## Exercise

What is the busiest **hour of the day** in all branches?

```
SELECT extract('hour' FROM sold_at AT TIME ZONE CASE
        WHEN branch = 'NY' then 'America/New_York'
        WHEN branch = 'LA' then 'America/Los_Angeles'
      END) AS hour_of_day,
       count(*) AS sales
FROM sale
GROUP BY hour_of_day
ORDER BY sales DESC;
```

hour_of_day	sales
23	5
3	4
0	2
2	2
22	1
13	1



## When working with dates and times be explicit about the time zone

Wrong!	Right!
<code>date_part('month', sold_at)</code>	<code>date_part('month', sold_at at time zone 'America/New_York')</code>
<code>extract('month' from sold_at)</code>	<code>extract('month' from sold_at at time zone 'America/New_York')</code>
<code>sold_at::date</code>	<code>(sold_at at time zone 'America/New_York')::date</code>
<code>'2020-03-22 11:00'</code>	<code>'2020-03-22 11:00 America/New_York'</code>



## Exercise

How many sales were there in **March**?

**\*In "America/New\_York" time**





## Exercise

How many sales were there in **March**?

**\*In "America/New\_York" time**

```
SELECT count(*)
FROM sale
WHERE sold_at BETWEEN '2020-03-01 America/New_York' AND '2020-04-01 America/New_York';

count
-----
9
```



## Exercise

How many sales were there in **April**?

\*In "America/New\_York" time

```
SELECT count(*)
FROM sale
WHERE sold_at BETWEEN '2020-04-01 America/New_York' AND '2020-05-01 America/New_York';

count
-----
7
```



## Exercise

What can possibly go wrong?

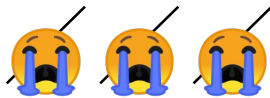
```
SELECT count(*) FROM sale;
```

```
count
```

```
-----
```

```
15
```

**March (9) + April (7) = 16**





## **BETWEEN is Inclusive!**

One sale is counted twice

```
SELECT * FROM sale WHERE sold_at = '2020-04-01 America/New_York';
```

id	branch	sold_at	customer	product	price	discount
2	NY	2020-03-31 21:00:00-07		Shoes	5000	0



# Use Half Open Ranges

## Ranges that don't overlap

```
SELECT count(*)
FROM sale
WHERE sold_at >= '2020-03-01 America/New_York'
AND sold_at < '2020-04-01 America/New_York';
count
-----
      8
```

```
SELECT count(*)
FROM sale
WHERE sold_at >= '2020-04-01 America/New_York'
AND sold_at < '2020-05-01 America/New_York';
count
-----
      7
```

**March (8) + April (7) = 15** 🧐



# BETWEEN is Inclusive!

## Not only timestamps...

- **Timestamps**
  - Unlikely if data is created by users (odd times)
  - More likely when data is created automatically (by batch jobs etc.)
- **Integers**
  - Binning
  - Dividing to buckets
  - Search for ranges...
- **Date** (with no time)
  - Very common when filtering on range of dates

```
-- Search for 90's babies...  
SELECT * FROM birthdates WHERE birthdate BETWEEN '1990-01-01' AND '2000-01-01';
```



# SUMMARY

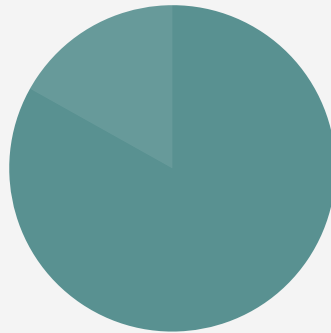
## Working with date and time



- **Dates and times in PostgreSQL**
  - How is date represented in the database?
  - Using time zones
  - Daylight saving
  - Database types `date`, `timestamp`, `timestampz`
  - Interval
  - Useful functions `now`, `date_trunc`, `date_part`, `extract`
- **Be explicit about the time zone when working with times**  
Otherwise you might get incorrect results
- **BETWEEN is inclusive**  
Use half open ranges to avoid overlap

# Writing Faster SQL

Performance Tips







# The Path of a Query



[Official Documentation](#)

## Connection

Client application creates a connection to the database server

## Parser Stage

Check the query for syntax errors

## Rewrite System

Make adjustments to the query for the database internal needs (for example, inline views).  
This stage does not change the logic of the query

## Planner / Optimizer

Evaluate the query and find the **best execution plan**

## Executor

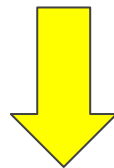
Execute the query according to the execution plan



# The Optimizer



[Official Documentation](#)



Connection

Parser Stage

Rewrite System

Planner /  
Optimizer

Executor

- **Generate all possible execution plans**
  - This can take some time, depending on the query.
- **Estimate the cost for each plan**
  - Cost is measured in arbitrary units
  - Cost mostly measures disk page fetches (IO)
  - Using statistics obtained from analysing tables and indexes
  - Cost can be used to compare execution plans
- **Choose the plan with the lowest cost**
  - The lower the cost the faster is execution is (expected) to be



# The Optimizer

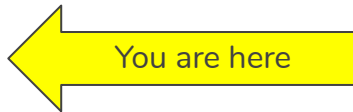
## Rule vs. Cost

### Rule based optimizer

- Using rules and heuristics to come up with the best plan
- Intuitive
- Easier to implement

### Cost based optimizer

- Score all possible execution plans based on statistics
- Less intuitive
- Harder to implement
- Expected to produce better plans for most queries



[Cockroach Labs: How We Built a Cost-Based SQL Optimizer](#)



# Generate Some Data

Can't optimize with just 15 rows...

```
INSERT INTO sale (id, branch, sold_at, customer, product, price, discount)
SELECT
  (SELECT MAX(id) FROM sale) + generate_series(1, 99985) as id,
  (ARRAY['NY', 'LA'])[ceil(random() * 2)] AS branch,
  '2020-03-01 00:00:00 UTC'::timestampz + interval '1 hour' * random() * 24 * 30 * 6 AS sold_at,
  (ARRAY['Bill', 'David', 'John', 'Lily'])[ceil(random() * 30)] AS customer,
  (ARRAY['Shoes', 'Shirt', 'Pants', 'Hat', 'Give Away'])[ceil(random() * 4)] AS product,
  round(random() * 150 * 100)::integer / 10 * 10 as price,
  0 as discount;

ANALYZE sale;
```



# Generate Some Data

## Tricks to generate random data...

```
INSERT INTO sale (id, branch, sold_at, customer, product, price, discount)
SELECT
  (SELECT MAX(id) FROM sale) + generate_series(1, 9985) as id,
  (ARRAY['NY', 'LA'])[ceil(random() * 2)] AS branch,
  '2020-03-01 00:00:00 UTC'::timestamp + interval '1 hour' * random() * 24 * 30 * 6 AS sold_at,
  (ARRAY['Bill', 'David', 'John', 'Lily'])[ceil(random() * 30)] AS customer,
  (ARRAY['Shoes', 'Shirt', 'Pants', 'Hat', 'Give Away'])[ceil(random() * 4)] AS product,
  round(random() * 150 * 100)::integer / 10 * 10 as price,
  0 as discount;
```

**Generate a range of 9985 consecutive numbers starting for the last sale\_id**

`generate_series` is a “Set Returning Function”: A function that returns more than one rows. This is what’s making the query return many rows.



[Set Returning Functions](#)



# Generate Some Data

## Tricks to generate random data...

```
INSERT INTO sale (id, branch, sold_at, customer, product, price, discount)
SELECT
  (SELECT MAX(id) FROM sale) + generate_series(1, 9985) as id,
  (ARRAY['NY', 'LA'])[ceil(random() * 2)] AS branch,
  '2020-03-01 00:00:00 UTC'::timestamp + interval '1 hour' * random() * 24 * 30 * 6 AS sold_at,
  (ARRAY['Bill', 'David', 'John', 'Lily'])[ceil(random() * 30)] AS customer,
  (ARRAY['Shoes', 'Shirt', 'Pants', 'Hat', 'Give Away'])[ceil(random() * 4)] AS product,
  round(random() * 150 * 100)::integer / 10 * 10 as price,
  0 as discount;
```

### Pick a random value from an array from values

To produce both known and unknown customers, the customer field is using a range greater than the length of the array (index outside the array will result in NULL).

Arrays in PostgreSQL starts at index 1



# Generate Some Data

## Tricks to generate random data...

```
INSERT INTO sale (id, branch, sold_at, customer, product, price, discount)
SELECT
  (SELECT MAX(id) FROM sale) + generate_series(1, 9985) as id,
  (ARRAY['NY', 'LA'])[ceil(random() * 2)] AS branch,
  '2020-03-01 00:00:00 UTC'::timestamp + interval '1 hour' * random() * 24 * 30 * 6 AS sold_at,
  (ARRAY['Bill', 'David', 'John', 'Lily'])[ceil(random() * 30)] AS customer,
  (ARRAY['Shoes', 'Shirt', 'Pants', 'Hat', 'Give Away'])[ceil(random() * 4)] AS product,
  round(random() * 150 * 100)::integer / 10 * 10 as price,
  0 as discount;
```

**Use interval arithmetics to produce random dates in the next 6 months**

1 hour \* 24 = 1 day \* 30 = ~one month \* 6 = six months!



# Generate Some Data

## Tricks to generate random data...

```
INSERT INTO sale (id, branch, sold_at, customer, product, price, discount)
SELECT
  (SELECT MAX(id) FROM sale) + generate_series(1, 9985) as id,
  (ARRAY['NY', 'LA'])[ceil(random() * 2)] AS branch,
  '2020-03-01 00:00:00 UTC'::timestamp + interval '1 hour' * random() * 24 * 30 * 6 AS sold_at,
  (ARRAY['Bill', 'David', 'John', 'Lily'])[ceil(random() * 30)] AS customer,
  (ARRAY['Shoes', 'Shirt', 'Pants', 'Hat', 'Give Away'])[ceil(random() * 4)] AS product,
  round(random() * 150 * 100)::integer / 10 * 10 as price,
  0 as discount;
```

### Produce random prices in range 0 - 150\$

Use the fact PostgreSQL truncate integers to produces prices in multiples of 10 cents

For example:  $\text{round}(\text{random}() * 150 * 100)::\text{integer} = 4589 / 10 = 458 * 10 = 4580$





# Generate Some Data

## Analyze the table

```
INSERT INTO sale (id, branch, sold_at, customer, product, price, discount)
SELECT
  (SELECT MAX(id) FROM sale) + generate_series(1, 99985) as id,
  (ARRAY['NY', 'LA'])[ceil(random() * 2)] AS branch,
  '2020-03-01 00:00:00 UTC'::timestamp + interval '1 hour' * random() * 24 * 30 * 6 AS sold_at,
  (ARRAY['Bill', 'David', 'John', 'Lily'])[ceil(random() * 30)] AS customer,
  (ARRAY['Shoes', 'Shirt', 'Pants', 'Hat', 'Give Away'])[ceil(random() * 4)] AS product,
  round(random() * 150 * 100)::integer / 10 * 10 as price,
  0 as discount;

ANALYZE sale;
```

Collect stats on the table

 [ANALYZE command](#)



# Execution Plan

## Using the EXPLAIN command

```
EXPLAIN SELECT * FROM sale;  
          QUERY PLAN  
-----  
Seq Scan on sale  (cost=0.00..1751.00 rows=10000 width=33)
```

- View execution plan
- Will not execute the query
- Shows estimates



[Using EXPLAIN](#)



# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale;  
QUERY PLAN  
-----  
Seq Scan on sale (cost=0.00..1751.00 rows=10000 width=33)
```

Database planning a  
**sequential scan** on the  
entire sale table

The estimated total cost  
for this plan node

Estimated the query  
will return 10,000 rows



# Execution Plan

## Useful statistics

- **reltuples**  
Number of rows in the table
- **relpages**  
Number of blocks in the table

 pg\_class

```
SELECT relname, relpages, reltuples
FROM pg_class
WHERE relname = 'sale';
```

relname	relpages	reltuples
sale	751	10000

```
SHOW block_size;
8192
```

```
SELECT pg_size_pretty(pg_relation_size('sale'));
6008 kB
```



# Execution Plan

A closer look *at the cost*

```
cost =  
relpages * seq_page_cost  
+  
reltuples * cpu_tuple_cost  
-----  
751 * 1  
+  
100000 * 0.01  
-----  
cost = 1,751
```

- Cost can vary between platforms
- Can vary because of sampling

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'sale';  
relpages | 751  
reltuples | 100000  
  
SHOW seq_page_cost;  
seq_page_cost | 1  
  
SHOW cpu_tuple_cost;  
cpu_tuple_cost | 0.01  
  
EXPLAIN SELECT * FROM sale;  
QUERY PLAN  
  
-----  
Seq Scan on sale (cost=0.00..1751.00 rows=100000 width=33)
```



# Execution Plan

## Estimated vs. Actual Rows

```
EXPLAIN (ANALYZE ON, TIMING ON) SELECT * FROM sale;
```

QUERY PLAN

```
-----  
Seq Scan on sale (cost=0.00..1751 rows=100000 width=33) (actual time=0.026..11.209 rows=100000 loops=1)
```

Planning Time: 0.105 ms

Execution Time: 15.568 ms

- Executes the query
- View **estimated rows vs actual rows**
- Time execution



# Execution Plan

## Useful statistics

- **null\_frac**  
% of null value in field
- **n\_distinct**  
+ Number of distinct values (closed set)  
- Distinct values / num rows (sequential, unique  
-1 = unique)
- **correlation**  
Correlation between physical row ordering and  
logical ordering of the column values.  
1 = rows stored sorted (in ascending order) on  
disk, index scans are cheaper.

```
SELECT attname, null_frac, n_distinct, correlation
FROM pg_stats
WHERE tablename = 'sale';
```

attname	null_frac	n_distinct	correlation
sold_at	0	-1	-0.004219238
customer	0.8694	4	0.2778079
product	0	5	0.2626033
price	0	1501	0.0019420487
discount	0	1	1
branch	0	2	0.4909097
id	0	-1	1



# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale WHERE id = 1000;
```

QUERY PLAN

```
-----  
Index Scan using sale_pkey on sale (cost=0.29..8.31 rows=1 width=33)
```

Index Cond: (id = 1000)

Database **used the index** on the primary key to access only the relevant blocks

The estimated cost is 8

Database estimated that the query will return only one row because the field has a unique constraint



[Using EXPLAIN](#)





# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale WHERE customer IS NULL;  
  
QUERY PLAN  
  
-----  
Seq Scan on sale (cost=0.00..1751.00 rows=86940 width=33)  
  Filter: (customer IS NULL)
```

$\text{rows} = \text{reltuples} * \text{null\_frac}$

$86,940 = 100,000 * 0.8694$



# Execution Plan

## Useful statistics

- **most\_common\_vals**  
Common values in the column
- **most\_common\_freqs**  
Corresponding frequency

```
SELECT attname, most_common_vals, most_common_freqs
FROM pg_stats
WHERE tablename = 'sale'
AND attname IN ('branch', 'product', 'customer');
```

attname	most_common_vals	most_common_freqs
customer	{Lily,David,Bill,John}	{0.03516,0.0319,0.0317,0.03176}
product	{Shoes,Pants,Hat,Shirt}	{0.25396,0.2505,0.2491,0.24636}
branch	{NY,LA}	{0.50433,0.4956}



# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale WHERE branch = 'NY';  
Seq Scan on sale (cost=0.00..2001.00 rows=50433 width=33)  
  Filter: (branch = 'NY'::text)
```

NY freq = 0.50433

```
EXPLAIN SELECT * FROM sale WHERE branch = 'LA';  
Seq Scan on sale (cost=0.00..2001.00 rows=49567 width=33)  
  Filter: (branch = 'LA'::text)
```

LA freq = 0.49567

```
EXPLAIN SELECT * FROM sale WHERE branch = 'FOO';  
Seq Scan on sale (cost=0.00..2001.00 rows=1 width=33)  
  Filter: (branch = 'FOO'::text)
```

Unknown value



# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale WHERE product IN ('Shoes', 'Pants');  
QUERY PLAN
```

```
-----  
Seq Scan on sale (cost=0.00..2001.00 rows=50447 width=33)  
  Filter: (product = ANY ('{Shoes,Pants}'::text[]))
```

Shoes freq = 0.25396666

Pants freq = 0.2505

Shoes OR Pants = 0.25396666 + 0.2505 = 0.50447

**Estimated rows = 10,000 \* 0.50447 = 50,447**



# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale WHERE branch = 'NY' AND product = 'Shoes';  
QUERY PLAN
```

```
-----  
Seq Scan on sale (cost=0.00..2251.00 rows=12808 width=33)  
  Filter: ((branch = 'NY'::text) AND (product = 'Shoes'::text))
```

NY freq = 0.5043333

Shoes freq = 0.25396666

NY AND Pants = 0.5043333 \* 0.25396666 = 0.12808

Estimated rows = 10,000 \* 0.12808 = 12,808




# Execution Plan

## A closer look

```
EXPLAIN SELECT * FROM sale WHERE branch = 'NY' OR product = 'Shoes';  
QUERY PLAN
```

```
-----  
Seq Scan on sale (cost=0.00..2251.00 rows=63022 width=33)  
  Filter: ((branch = 'NY'::text) OR (product = 'Shoes'::text))
```

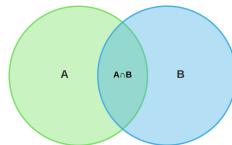
 [How the Planner Uses Statistics](#)

NY freq = 0.5043333

Shoes freq = 0.25396666

NY OR Pants =  $0.5043333 + 0.25396666 - (0.5043333 * 0.25396666) = 0.63022$

Estimated rows =  $10,000 * 0.63022 = 63,022$





# Execution Plan

## How Accurate is it?

```
EXPLAIN (ANALYZE ON) SELECT * FROM sale WHERE branch = 'NY' OR product = 'Shoes';
```

QUERY PLAN

```
Seq Scan on sale (cost=0.00..2251.00 rows=62683 width=33) (actual time=0.026..36.181 rows=62678 loops=1)
```

```
Filter: ((branch = 'NY'::text) OR (product = 'Shoes'::text))
```

```
Rows Removed by Filter: 37322
```

```
Planning Time: 0.180 ms
```

```
Execution Time: 40.728 ms
```

Pretty accurate!



# Execution Plan

*Plan-reading is an art  
that requires some  
experience to master...*







# KEY TAKEAWAYS

## Execution Plans



- 💡 **Databases are really good at optimizing queries!**
  - The optimizer is only as good as its stats
  - Incorrect estimates can indicate a possible performance problem
  - Provide the database with information about your data by analyzing and using constraints
- 💡 **Use cost to compare execution plans**
  - The lower the cost, the faster the query is expected to be
  - Cost is estimated based on statistics, so it can be inaccurate
  - Cost is not an absolute measure, use it only for comparison



# Working With Indexes Setup

1. Create an index on the field customer

```
CREATE INDEX sale_customer_ix ON sale(customer);
```

2. Create an index on the field sold\_at

```
CREATE INDEX sale_sold_at_ix ON sale(sold_at);
```

3. Create an index on the field product

```
CREATE INDEX sale_product_ix ON sale(product);
```

 [PostgreSQL Indexes: Documentation](#)

 [Indexes in PostgreSQL](#)

 [Indexes in PostgreSQL: B-Tree](#)

```
\d sale
      Column          |          Type          | Nullable |
-----+-----+-----+-----+
id                    | integer                | not null |
branch               | text                   |          |
sold_at              | timestamp with time zone |          |
customer             | text                   |          |
product              | text                   |          |
price                | integer                |          |
discount             | integer                |          |
```

Indexes:

```
"sale_pkey" PRIMARY KEY, btree (id)
"sale_customer_ix" btree (customer)
"sale_sold_at_ix" btree (sold_at)
"sale_product_ix" btree (product)
```



# Nulls are Not Indexed

Null values are not indexed by a B-Tree index

```
EXPLAIN SELECT * FROM sale WHERE customer = 'Bill';
```

```
-----  
Bitmap Heap Scan on sale (cost=61.04..851.75 rows=3177 width=33)
```

```
  Recheck Cond: (customer = 'Bill'::text)
```

```
    -> Bitmap Index Scan on sale_customer_ix (cost=0.00..60.24 rows=3177 width=0)
```

```
      Index Cond: (customer = 'Bill'::text)
```

```
EXPLAIN SELECT * FROM sale WHERE customer IS NULL;
```

```
-----  
Seq Scan on sale (cost=0.00..1751 rows=86940 width=33)
```

```
  Filter: (customer IS NULL)
```



# Avoid Transformations on Indexed Fields

## Common mistakes

Indexes cannot be used with transformations

```
EXPLAIN SELECT * FROM sale WHERE lower(customer) = 'bill';
```

```
-----  
Seq Scan on sale (cost=0.00..2251.00 rows=500 width=33)  
  Filter: (lower(customer) = 'bill'::text)
```



[Avoid transformations on indexed fields](#)



# Avoid Transformations on Indexed Fields

## Common mistakes

Simple arithmetics on indexed field (id):

```
EXPLAIN SELECT * FROM sale WHERE id + 1 = 100;
```

```
-----  
Seq Scan on sale (cost=0.00..226.22 rows=50 width=33)  
  Filter: ((id + 1) = 100)
```

```
EXPLAIN SELECT * FROM sale WHERE id = 100 - 1;
```

```
-----  
Index Scan using sale_pkey on sale (cost=0.29..8.30 rows=1 width=33)  
  Index Cond: (id = 99)
```



# Avoid Transformations on Indexed Fields

## Common mistakes

Apply timezone on indexed field in comparison:

```
EXPLAIN SELECT * FROM sale WHERE sold_at at time zone 'America/New_York' > '2021-01-01';
```

```
-----  
Seq Scan on sale (cost=0.00..226.22 rows=3338 width=33)
```

```
  Filter: (timezone('America/New_York'::text, sold_at) > '2021-01-01 00:00:00'::timestamp  
without time zone)
```

```
EXPLAIN SELECT * FROM sale WHERE sold_at > '2021-01-01 America/New_York';
```

```
-----  
Index Scan using sale_sold_at_idx on sale (cost=0.29..8.30 rows=1 width=33)
```

```
  Index Cond: (sold_at > '2021-01-01 05:00:00+00'::timestamp with time zone)
```



# Avoid Transformations on Indexed Fields

## Common mistakes

Date arithmetics on the indexed field:

```
EXPLAIN SELECT * FROM sale WHERE sold_at - interval '1 day' > '2021-01-01 America/New_York'::timestampz;
```

```
-----  
Seq Scan on sale (cost=0.00..226.22 rows=3338 width=33)  
  Filter: ((sold_at - '1 day'::interval) > '2021-01-01 05:00:00+00'::timestamp with time zone)
```

```
EXPLAIN SELECT * FROM sale WHERE sold_at > '2021-01-01 America/New_York'::timestampz + interval '1 day';
```

```
-----  
Index Scan using sale_sold_at_idx on sale (cost=0.29..8.30 rows=1 width=33)  
  Index Cond: (sold_at > ('2021-01-01 05:00:00+00'::timestamp with time zone + '1 day'::interval))
```



# Avoid Transformations on Indexed Fields

## Common mistakes

Change string case (lower / upper)

Dont:

```
SELECT * FROM users WHERE lower(email) = 'me@hakibenita.com'
```

Do:

```
SELECT * FROM users WHERE email = lower('ME@HakiBenita.com')
```



[Indexes on Expressions](#) (if you really have to)





# Avoid Transformations on Indexed Fields

## Common mistakes

String concatenation

Dont:

```
SELECT * FROM users WHERE first_name || ' ' || last_name = 'Haki Benita'
```

Do:

```
SELECT * FROM users WHERE first_name = 'Haki' AND last_name = 'Benita'
```



# Common Misconception

## Indexes are not always the best plan

```
EXPLAIN SELECT * FROM sale WHERE product in ('Shoes');  
-----  
Bitmap Heap Scan on sale (cost=474.14..1537.61 rows=24997 width=33)  
  Recheck Cond: (product = 'Shoes'::text)  
-> Bitmap Index Scan on sale_product_ix (cost=0.00..467.89 rows=24997 width=0)  
    Index Cond: (product = 'Shoes'::text)  
  
EXPLAIN SELECT * FROM sale WHERE product in ('Shoes', 'Pants');  
-----  
Seq Scan on sale (cost=0.00..2001.00 rows=49473 width=33)  
  Filter: (product = ANY ('{Shoes,Pants}'::text[]))
```

Few rows  
Index used

Many rows  
Index NOT used



Accessing a large portion of the table using an index is inefficient



# SUMMARY

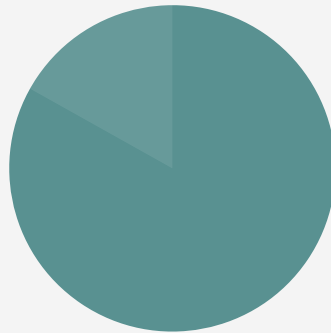
## Make the most of your DB!



- **How the database processes a query**  
Parse, rewrite, plan and execute
- **How to produce and compare execution plans**  
Using EXPLAIN
- **How the database is using statistics**  
To estimate row count and produce a plan
- **How row estimates are calculated**  
For different type of predicates
- **Nulls are not indexed**  
by B-TREE indexes
- **Avoid transformations on indexed fields**  
With examples using arithmetics, date and string manipulation
- **When using an index is not the best plan**  
Fetching a lot of rows using an index is inefficient

# Writing SQL Faster

Productivity Tips





# CASE Can Take Many Forms

Avoid repetition

```
SELECT
  CASE
    WHEN fruit = 'apple' THEN 'red'
    WHEN fruit = 'pear'  THEN 'green'
    WHEN fruit = 'orange' THEN 'orange'
    ELSE '?'
  END AS color
FROM
  fruit;
```



```
SELECT
  CASE fruit
    'apple' THEN 'red'
    'pear'  THEN 'green'
    'orange' THEN 'orange'
    ELSE '?'
  END AS color
FROM
  fruit;
```



# Reference Column in GROUP BY & ORDER BY

Use position or alias to avoid repetition

```
SELECT
    first_name || ' ' || last_name as full_name,
    count(*) as sales_by_user
FROM
    sale
GROUP BY
    first_name || ' ' || last_name
ORDER BY
    count(*) DESC
```



```
SELECT
    first_name || ' ' || last_name as full_name,
    count(*) as sales_by_user
FROM
    sale
GROUP BY
    1
ORDER BY
    sales_by_user DESC
```



It's best to avoid positional column reference in code, and use it only for ad-hoc queries.



# Use Selective Aggregates

## Where you used to use CASE



```
SELECT
  COUNT(*),
  SUM(CASE WHEN customer IS NULL THEN 1 ELSE 0 END),
  SUM(CASE WHEN customer IS NOT NULL THEN 1 ELSE 0 END)
FROM
  sale;
```



```
SELECT
  COUNT(*) AS sales,
  COUNT(*) FILTER (WHERE customer IS NULL),
  COUNT(*) FILTER (WHERE customer IS NOT NULL)
FROM
  sale;
```



# UNION vs. UNION ALL

## When to use each one...

Eliminating  
duplicates  
requires a sort,  
which can take  
some time...

```
SELECT 1 UNION SELECT 1;  
?column?  
-----  
1
```

UNION: Concatenate results **and**  
removes duplicates

```
SELECT 1 UNION ALL SELECT 1;  
?column?  
-----  
1  
1
```

UNION ALL: Concatenate results



[Know the Difference Between UNION and UNION ALL](#)





# Symmetric Range

When you aren't sure about the order

```
SELECT 5 BETWEEN 0 AND 10;  
true  
  
SELECT 5 BETWEEN 10 AND 0;  
false  
  
SELECT 5 BETWEEN SYMMETRIC 10 AND 0;  
true  
  
SELECT 5 BETWEEN SYMMETRIC 0 AND 10;  
true
```





# Use **DISTINCT ON**

Get the first / last **row** in a group

One of the most  
useful features of  
PostgreSQL!

```
SELECT DISTINCT ON (customer)
  *
FROM
  sale
ORDER BY
  customer,
  sold_at
```

- Can be used instead of RANK / ROW\_NUMBER
- DISTINCT ON clause can accept multiple fields
- Field in DISTINCT ON must be in ORDER BY
- Can control first / last using sort order (ASC / DESC)



[SELECT: Documentation](#)



[The many faces of distinct in PostgreSQL: DISTINCT ON](#)



# More Awesome Features

Some *must know* PostgreSQL features

## [JSON Functions and Operators](#)

Store, query and manipulate JSON documents straight in the database.

## [Window Functions](#)

Write complicated analytics reports using advanced window functions.

## [Full Text Search](#)

Search, index and rank results using a fully featured text search in the database.

# What You've Learned



How to avoid common mistakes in SQL



How to write faster SQL



How to write SQL faster



# Where to Find Me

If you didn't have enough...



<https://hakibenita.com>

I write about SQL, Python, Django, Testing and Performance. I publish an article about once a month.

[Join the mailing list for updates >>](#)



[@be\\_haki](#)

I tweet random tips on SQL, tuning, performance and other random stuff I learn on my day to day.



# THANK YOU!

Haki Benita

