Amazing SQL your ORM can (or can't) do

Louise Grandjonc - pgconfEU 2019

@louisemeta

citusdata

Microsoft

About me

Software Engineer at Citus Data / Microsoft

Python developper

Postgres enthusiast

@louisemeta and @citusdata on twitter

www.louisemeta.com

louise.grandjonc@microsoft.com

citusdata

Microsoft

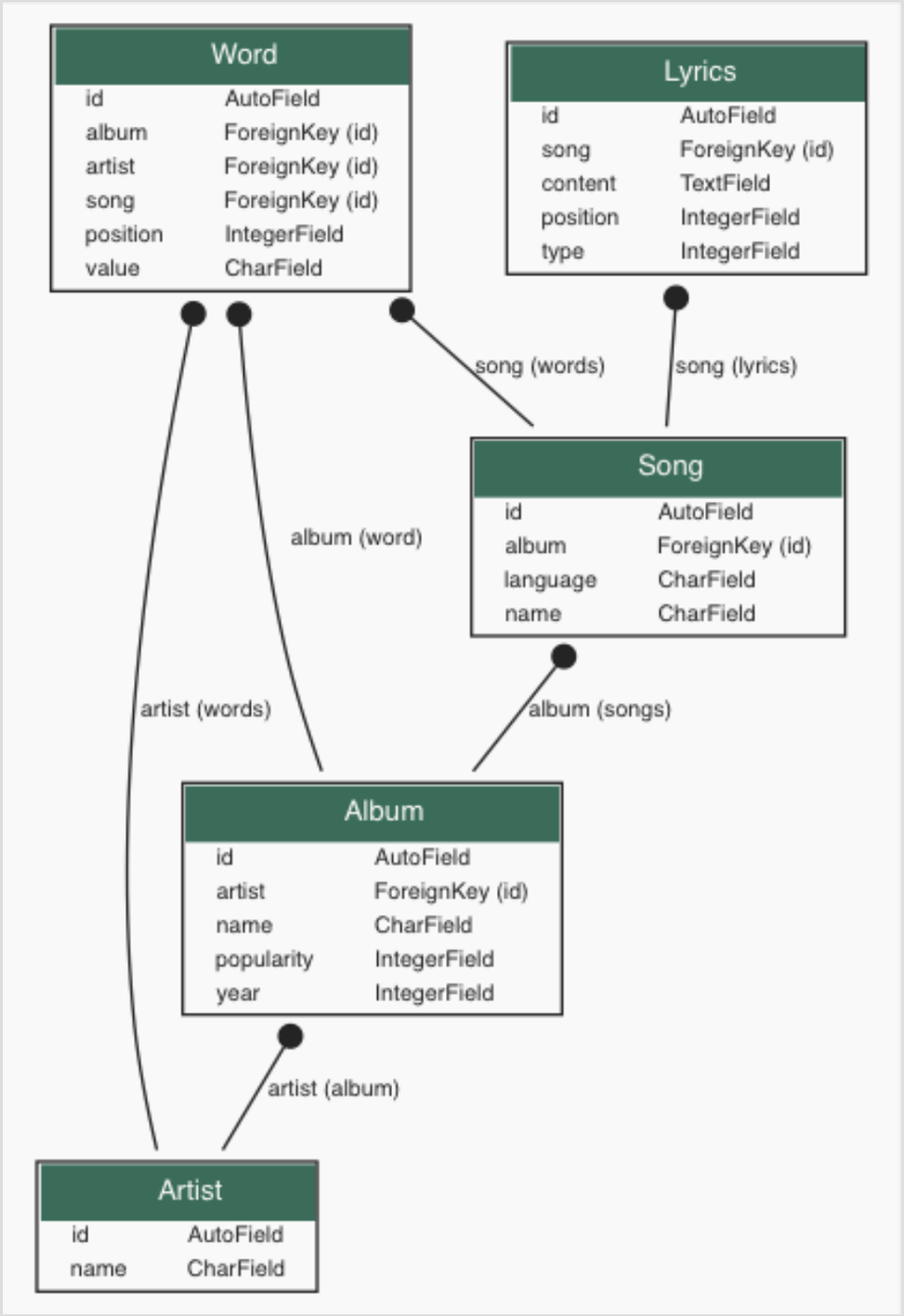
Why this talk

- I am a developper working with other developers
- I write code for applications using postgres
- I love both python and postgres
- I use ORMs
- I often attend postgres conferences
- A subject that I always enjoy is complex and modern SQL
- I want to use my database at its full potential

Today's agenda

1. Basic SQL reminder
2. Aggregate functions
3. GROUP BY
4. EXISTS
5. Subquery
6. LATERAL JOIN
7. Window Functions
8. GROUPING SETS / ROLLUP / CUBE
9. CTE (Common Table Expression)

Data model


Word
id AutoField
album ForeignKey (id)
artist ForeignKey (id)
song ForeignKey (id)
position IntegerField
value CharField
Lyrics
id AutoField
song ForeignKey (id)
content TextField
position IntegerField
type IntegerField
song (words)
song (lyrics)
Song
id AutoField
album ForeignKey (id)
language CharField
name CharField
album (word)
artist (words)
album (songs)
Album
id AutoField
artist ForeignKey (id)
name CharField
popularity IntegerField
year IntegerField
artist (album)
Artist
id AutoField
name CharField

Dataset

- **10 artists:** Kyo, Blink-182, Maroon5, Jonas Brothers, Justin Timberlake, Avril Lavigne, Backstreet Boys, Britney Spears, Justin Bieber, Selena Gomez

- **72 albums**

- **1012 songs**

- **120,029 words**: transformed the lyrics into tsvector and each vector (> 3 letters) went into a kyo_word row with its position.

ORMs we'll talk about today

- Django ORM (python)
- SQLAlchemy (python)
- Activerecord (ruby)
- Sequel (ruby)

A quick tour on basic SQL

SELECT columns
FROM table_a
(INNER, LEFT, RIGHT, OUTER) JOIN table_a ON table_a.x = table_b.y
WHERE filters
ORDER BY columns
LIMIT X

@louisemeta

citusdata

Microsoft

A quick tour on basic SQL

Django ORM (python)

```
Artist.objects.get(pk=10)
```

Sqlalchemy (python)

```
session.query(Artist).get(10)
```

Activerecord (ruby)

```
Artist.find(10)
```

Sequel (ruby)

```
Artist[10]
```

```
SELECT id, name
FROM kyo_artist
WHERE id=10;
```

A quick tour on basic SQL

Django ORM (python)

```
Album.objects
        .filter(year_gt=2009)
        .order_by('year')
```

Sqlalchemy (python)

```
session.query(Album)
.filter(Album.year > 2009)
.order_by(Album.year)
```

Activerecord (ruby)

```
Album.where('year > ?', 2009)
        .order(:year)
```

Sequel (ruby)

```
Album.where(Sequel[:year] > 2009)
        .order(:year)
```

```
SELECT id, name, year, popularity
FROM kyo_album
WHERE year > 2009
ORDER BY year;
```

A quick tour on basic SQL

Django ORM (python)

```
Album.objects
.filter(year_gt=2009)
.order_by('year')
.select_related('artist')
```

Sqlalchemy (python)

```
session.query(Album)
.join('artist')
.filter(Album.year > 2009)
.order_by(Album.year)
```

Activerecord (ruby)

```
Album.where('year > ?', 2009)
     .joins(:artist)
     .order(:year)
```

Sequel (ruby)

```
Album.where(Sequel[:year] > 2009)
     .join(:artist)
     .order(:year)
```

```
SELECT id, name, year, popularity
FROM kyo_album
INNER JOIN kyo_artist ON
kyo_artist.id=kyo_album.artist_id
WHERE year > 2009
ORDER BY year;
```

A quick tour on basic SQL

Django ORM (python)

```
Album.objects
.filter(artist_id=12)[5:10]
```

Sqlalchemy (python)

```
session.query(Album)
.filter(Album.artist_id == 12)
.offset(5).limit(10)
```

Activerecord (ruby)

```
Album.where(artist_id: 12)
     .limit(10).offset(5)
```

Sequel (ruby)

```
Album.where(artist_id: 12)
     .limit(10).offset(5)
```

```
SELECT id, name, year, popularity
FROM kyo_album
WHERE artist_id = 12
ORDER BY id
OFFSET 5 LIMIT 10;
```

To go further in pagination:

https://www.citusdata.com/blog/2016/03/30/five-ways-to-paginate/

Executing RAW SQL queries

Django

```
Word.objects.raw(query, args)
```

```
with connection.cursor() as cursor:
    cursor.execute(query)
    data = cursor.fetchall(cursor)
```

Executing RAW SQL queries

SQLAlchemy

```
engine = create_engine('postgres://localhost:5432/kyo_game')

with engine.connect() as con:
    rs = con.execute(query, **{'param1': 'value'})
    rows = rs.fetchall()
```

Executing RAW SQL queries

Activerecord

```
words = Word.find_by_sql ['SELECT * FROM words WHERE
artist_id=:artist_id', {:artist_id => 14}]
```

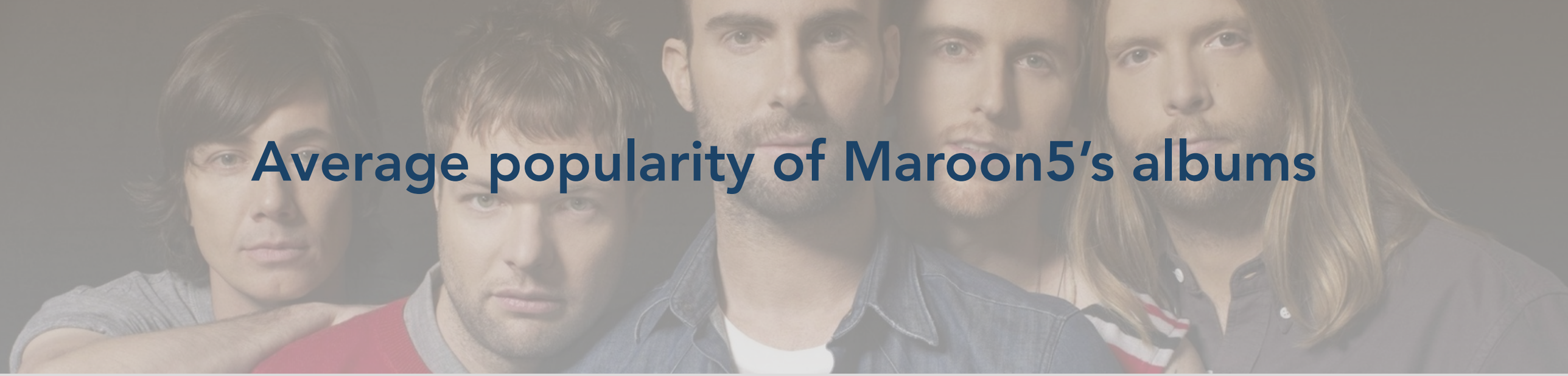
```
rows = ActiveRecord::Base.connection.execute(sql,  params)
```

The functions select/where/group also can take raw SQL.

Executing RAW SQL queries

Sequel

```
DB = Sequel.connect('postgres://localhost:5432/kyo_game_ruby')
DB['select * from albums where name = ?', name]
```



Average popularity of Maroon5's albums

```
SELECT AVG(popularity) FROM kyo_album WHERE artist_id=9;
```

```
# Django
popularity = Album.objects.filter(artist_id=9).aggregate(value=Avg('popularity')
```

```
# sqlalchemy
session.query(func.avg(Album.popularity).label('average'))
.filter(Album.artist_id == 9)
```

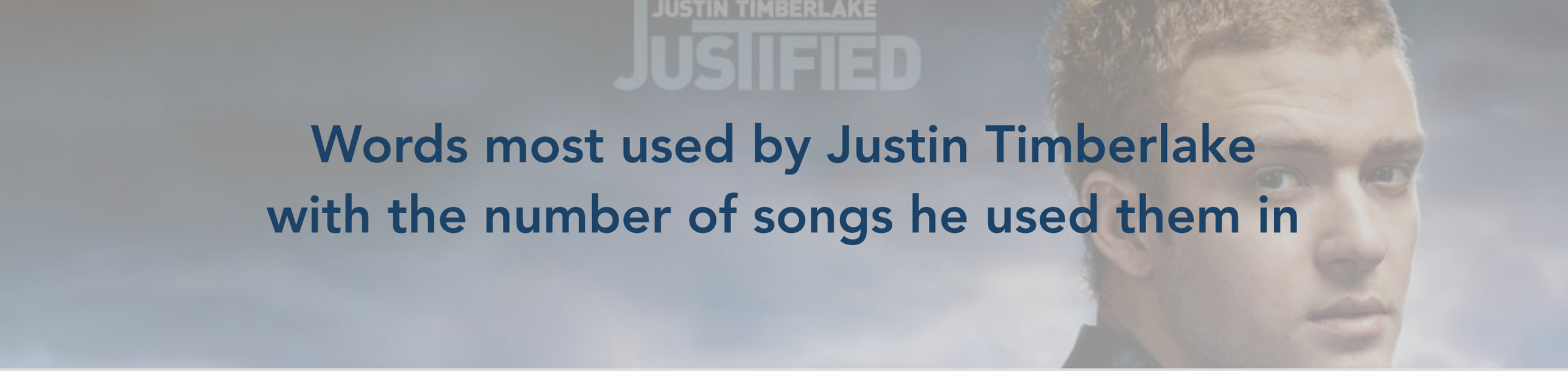
```
{'value': 68.16666666667}
```

Average popularity of Maroon5's albums
Ruby - ActiveRecord/Sequel

```
SELECT AVG(popularity) FROM kyo_album WHERE artist_id=9;
```

```
#Activerecord
Album.where(artist_id: 9).average(:popularity)
```

```
# Sequel
Album.where(artist_id: 9).avg(:popularity)
```



Words most used by Justin Timberlake with the number of songs he used them in

Word	Number of occurrences	Number of song
love	503	56
know	432	82
like	415	68
girl	352	58
babi	277	59
come	227	58
caus	225	62
right	224	34
yeah	221	54

Words most used by Justin Timberlake

```
SELECT COUNT(id) AS total,
  FROM kyo_word
 WHERE artist_id = 11;
```

17556

```
SELECT  value,
        COUNT(id) AS total
  FROM kyo_word
 WHERE artist_id = 11
 GROUP BY value
 ORDER BY total DESC
 LIMIT 10
```

Word	Number of
love	503
know	432
like	415
girl	352
babi	277
come	227
caus	225
right	224
yeah	221

Words most used by Justin Timberlake

Django

```
SELECT  value,
        COUNT(id) AS total,
        COUNT(DISTINCT song_id) AS total_songs
  FROM kyo_word
 WHERE artist_id = 11
 GROUP BY value ORDER BY total DESC
 LIMIT 10
```

```
Word.objects.filter(artist_id=self.object.pk)
                    .values('value')
                    .annotate(total=Count(‘id'),
                              total_song=Count('song_id', distinct=True))
                    .order_by('-total')[:10])
```

Words most used by Justin Timberlake

SQLAlchemy

```
SELECT  value,
        COUNT(id) AS total,
        COUNT(DISTINCT song_id) AS total_songs
  FROM kyo_word
 WHERE artist_id = 11
 GROUP BY value ORDER BY total DESC
 LIMIT 10
```

```
session.query(
   Word.value,
   func.count(Word.value).label('total'),
   func.count(distinct(Word.song_id)))
.group_by(Word.value)
.order_by(desc('total')).limit(10).all()
```

Words most used by Justin Timberlake

Activerecord

```
Word.where(artist_id: 11)
.group(:value)
.count(:id)
```

```
Word.where(artist_id: 11)
.group(:value)
.select('count(distinct song_id), count(id)')
.order('count(id) DESC')
.limit(10)
```

Words most used by Justin Timberlake

Sequel

```
Word.where(artist_id: 11)
.group_and_count(:value)
```

```
Word.group_and_count(:value)
.select_append{count(distinct song_id).as(total)}
.where(artist_id: 11)
.order(Sequel.desc(:count))
```

To go further with aggregate functions

AVG

COUNT (DISTINCT)

Min

Max

SUM

Support with ORMs

	Django	SQLAlchemy	Activerecord (*)	Sequel
AVG	Yes	Yes	Yes	Yes
COUNT	Yes	Yes	Yes	Yes
Min	Yes	Yes	Yes	Yes
Max	Yes	Yes	Yes	Yes
Sum	Yes	Yes	Yes	Yes

* Activerecord: to cumulate operators, you will need to use select() with raw SQL

Words that Avril Lavigne only used in the song "Complicated"

Word
dress
drivin
foolin
pose
preppi
somethin
strike
unannounc

Words that Avril Lavigne only used in the song “Complicated” - Django

```
SELECT *
FROM kyo_word word
WHERE song_id=342
AND NOT EXISTS (
    SELECT 1 FROM kyo_word word2  WHERE
    word2.artist_id=word.artist_id
    word2.song_id <> word.song_id
    AND word2.value=word.value);
```

Filter the result if the subquery returns no row

Subquery for the same value for a word, but different primary key

```
same_word_artist = (Word.objects
                    .filter(value=OuterRef(‘value’), artist=OuterRef('artist'))
                    .exclude(song_id=OuterRef(‘song_id’))

context['unique_words'] = Word.objects.annotate(is_unique=~Exists(same_word_artist))
                                      .filter(is_unique=True, song=self.object)
```

Words that Avril Lavigne only used in the song “Complicated” - SQLAlchemy

```
word1 = Word
word2 = aliased(Word)

subquery  = session.query(word2).filter(value == word1.value, artist_id ==
word1.artist_id, song_id != word1.song_id)

session.query(word1).filter(word1.song_id == 342, ~subquery.exists())
```

Words that Avril Lavigne only used in the song “Complicated” - Activerecord

There is an exists method:
Album.where(name: ‘Kyo').exists?

But in a subquery, we join the same table and they can’t have an alias

```
Word.where(song_id: 342).where(
'NOT EXISTS (SELECT 1 FROM words word2  WHERE  word2.artist_id=words.artist_id
AND word2.song_id <> words.song_id AND word2.value=words.value)’
)
```

An example where EXISTS performs better

I wanted to filter the songs that had no value in the table kyo_word yet.
A basic version could be

```
Song.objects.filter(words__isnull=True)
```

```
SELECT "kyo_song"."id", "kyo_song"."name",
"kyo_song"."album_id", "kyo_song"."language"
FROM "kyo_song"
LEFT OUTER JOIN "kyo_word" ON ("kyo_song"."id" =
"kyo_word"."song_id")
WHERE "kyo_word"."id" IS NULL
```

17-25ms

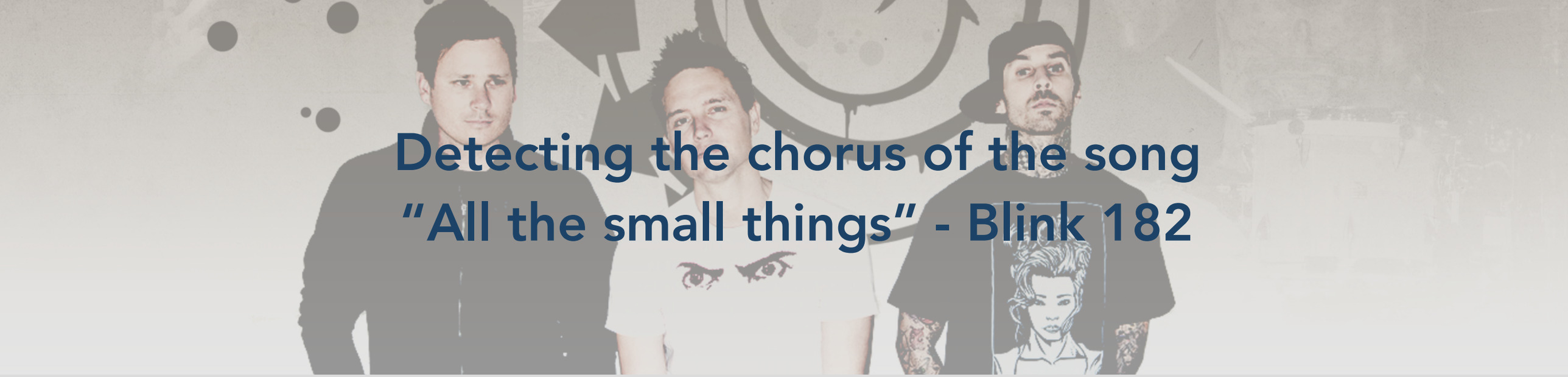
An example where EXISTS performs better

And with an EXISTS

```
Song.objects.annotate(processed=Exists(Word.objects.filter(song_id=OuterRef('pk'))))
.filter(processed=False)
```

```
SELECT * ,
EXISTS(SELECT * FROM "kyo_word" U0 WHERE U0."song_id" = ("kyo_song"."id"))
AS "processed"
FROM "kyo_song"
WHERE EXISTS(SELECT * FROM "kyo_word" U0 WHERE U0."song_id" =
("kyo_song"."id")) = False
```

4-6ms



Detecting the chorus of the song
“All the small things” - Blink 182

To make it simple, we want to know what group of two words is repeated more than twice.

“Say it ain’t so
I will not go
Turn the lights off
Carry me home”

Word	Next word	Occurences
turn	light	4
light	carri	4
carri	home	4

Detecting the chorus of a song
Subquery

Step 1: Getting the words with their next word

```
SELECT value,
    (
     SELECT U0.value
       FROM kyo_word U0
      WHERE (U0.position > (kyo_word.position) AND U0.song_id = 441)
      ORDER BY U0.position ASC
      LIMIT 1
    ) AS "next_word",

FROM "kyo_word"
 WHERE "kyo_word"."song_id" = 441
```

→ Subquery

Detecting the chorus of a song
Subquery

Step 2: Getting the words with their next word, with counts

```
SELECT kyo_word.value,
       (
        SELECT U0.value
          FROM kyo_word U0
         WHERE (U0.position > (kyo_word.position) AND U0.song_id =
441)
         ORDER BY U0.position ASC
         LIMIT 1
       ) AS next_word,
       COUNT(*) AS total
  FROM kyo_word
 WHERE kyo_word.song_id = 441
GROUP BY 1, 2
HAVING COUNT(*) > 2
```

We want thee count grouped by the word and its following word

A chorus should appear more than twice in a song

Detecting the chorus of a song
Subquery - Django

```
next_word_qs = (Word.objects
                  .filter(song_id=self.object.pk,
                          position__gt=OuterRef('position'))
                .order_by("position")
                .values('value'))[:1]

context['words_in_chorus'] = (Word.objects
                              .annotate(next_word=Subquery(next_word_qs))
                              .values('value', 'next_word')
                              .annotate(total=Count('*'))
                              .filter(song=self.object, total__gt=2))
                              .order_by('-total')
```

Word	Next word	Occurences
turn	light	4
light	carri	4
carri	home	4

Detecting the chorus of a song
Subquery - SQLAlchemy

```
word1 = Word
word2 = aliased(Word)

next_word_qs = session.query(word2.value)
.filter(word2.song_id==word1.song_id,
        word2.position > word1.position)
.order_by(word2.position).limit(1)

word_in_chorus = session.query(word1.value,
next_word_qs.label('next_word'),
func.count().label('total'))
.filter(word1.song_id == 441, func.count() > 2)
.group_by(word1.value, 'next_word')
```

Word	Next word	Occurences
turn	light	4
light	carri	4
carri	home	4

Detecting the chorus of a song
Subquery - ActiveRecord

```
Word.select('value,
(SELECT U0.value FROM words U0 WHERE U0.position >
(words.position) AND U0.song_id = words.song_id ORDER BY
U0.position ASC LIMIT 1) as next_word,
count(*) as total’)
.where(song_id: 441)
.having('count(*) > 2’)
.group('1, 2')
```

Word	Next word	Occurences
turn	light	4
light	carri	4
carri	home	4

Detecting the chorus of a song
Subquery - Sequel

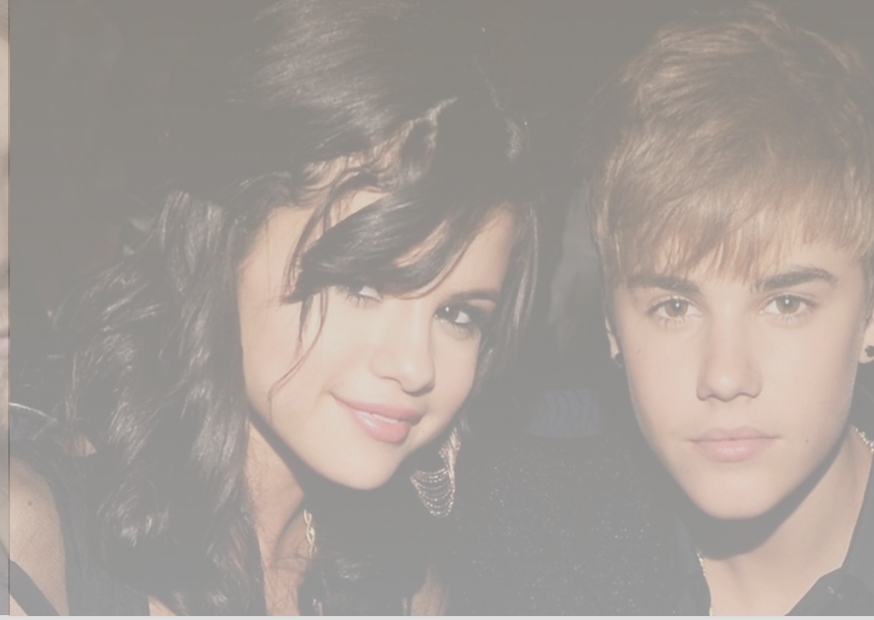
```
Word.from(DB[:words].where(Sequel[:song_id] =~ 441).as(:word_2))
.select{[
    Sequel[:word_2][:value],
    Word.select(:value).where{(Sequel[:position] > Sequel[:word_2]
[:position]) & (Sequel[:song_id] =~ 441)}.
    order(Sequel[:position]).limit(1).as(:following_word),

count(:id).as(:total)]}
.group(:value, :following_word)
.having(:total > 2)
```

Word	Next word	Occurences
turn	light	4
light	carri	4
carri	home	4

Support with ORMs

	Django	SQLAlchemy	Activerecord	Sequel
Subquery into column	Yes	Yes	No	Yes
FROM subquery	No	Yes	Yes	Yes
Subquery in WHERE clause	Yes	Yes	Yes-ish	Yes





LATERAL JOIN

We want all the artists with their last album.

Artist id	Artist Name	Album id	Album name	Year	Popularity
6	Kyo	28	Dans a peau	2017	45
8	Blink-182	38	California	2016	
9	Maroon5	44	Red Pill Blues	2017	82
10	Jonas Brothers	51	Happiness Begins	2019	
11	Justin Timberlake	61	Man Of The Woods	2018	
12	Avril Lavigne	69	Head Above Water	2019	
13	Backstreet Boys	90	DNA	2019	68
14	Britney Spears	87	Glory	2016	
15	Justin Bieber	104	Purpose	2015	
16	Selena Gomez	98	Revival	2015	

LATERAL JOIN

Why can’t we do it with a subquery?

```
SELECT artist.*,
(SELECT * FROM kyo_album album
WHERE album.artist_id = artist_id
ORDER BY year DESC
LIMIT 1) AS album
FROM kyo_artist artist;


ERROR:  subquery must return only one column
```

LATERAL JOIN

Why can’t we do it by joining subqueries?

```
SELECT artist.*,
album.*
FROM kyo_artist artist
INNER JOIN (
    SELECT * FROM kyo_album
    ORDER BY year DESC
    LIMIT 1
) AS album ON artist.id = album.artist_id;


 id |      name      | id |       name       | year | artist_id | popularity
----+----------------+----+------------------+------+-----------+------------
 10 | Jonas Brothers | 51 | Happiness Begins | 2019 |        10 |
(1 row)
```

LATERAL JOIN

Here is the solution

```
SELECT artist.*, album.*
FROM kyo_artist artist
INNER JOIN LATERAL (
   SELECT * FROM kyo_album
   WHERE kyo_album.artist_id = artist.id
   ORDER BY year DESC LIMIT 1) album on true
```

LATERAL JOIN

So we get our wanted result

Artist id	Artist Name	Album id	Album name	Year	Popularity
6	Kyo	28	Dans a peau	2017	45
8	Blink-182	38	California	2016	
9	Maroon5	44	Red Pill Blues	2017	82
10	Jonas Brothers	51	Happiness Begins	2019	
11	Justin Timberlake	61	Man Of The Woods	2018	
12	Avril Lavigne	69	Head Above Water	2019	
13	Backstreet Boys	90	DNA	2019	68
14	Britney Spears	87	Glory	2016	
15	Justin Bieber	104	Purpose	2015	
16	Selena Gomez	98	Revival	2015	

LATERAL JOIN
SQLAlchemy

It won't return an object, as the result can't really be matched to one. So you will use the part of sqlalchemy that's not ORM like

```
    subquery = select([
        album.c.id,
        album.c.name,
        album.c.year,
        album.c.popularity])
        .where(album.c.artist_id==artist.c.id)
    .order_by(album.c.year)
    .limit(1)
    .lateral('album_subq')

query = select([artist, subquery.c.name, subquery.c.year,
subquery.c.popularity])
.select_from(artist.join(subquery, true()))
```

Support with ORMs

	Django	SQLAlchemy	Activerecord	Sequel
LATERAL	No	Yes	No	Yes

For each album of the backstreet boys: Words ranked by their frequency

Word	Album	Frequency	Rank
roll	Backstreet Boys	78	1
know	Backstreet Boys	46	2
heart	Backstreet Boys	43	3
babi	Backstreet Boys	42	4
wanna	Backstreet Boys	36	5
everybodi	Backstreet Boys	33	6
parti	Backstreet Boys	30	7
girl	Backstreet Boys	28	8
nobodi	Backstreet Boys	26	9
…	…	…	…
crazi	Backstreet Boys	8	24
wish	Backstreet Boys	8	24
shake	Backstreet Boys	7	25
…	…	…	…
love	Backstreet's Back	33	3
yeah	Backstreet's Back	27	4
babi	Backstreet's Back	23	5
…	…	…	…

For each album of the backstreet boys: Words ranked by their frequency

```
SELECT value as word,
       name as album_name,
       COUNT(word.value) AS frequency,
       DENSE_RANK() OVER (
            PARTITION BY word.album_id
            ORDER BY COUNT(word.value) DESC
        ) AS ranking
  FROM kyo_word word
  INNER JOIN kyo_album album
    ON (word.album_id = album.id)
WHERE word.artist_id = 13
GROUP BY word.value,
       album.name,
       word.album_id
ORDER BY word.album_id ASC
```

Dense rank is the function we use for the window function

We indicate the partition by album_id, because we want a rank per album

We indicate the order to use to define the rank

For each album of the backstreet boys:
Window Functions - Django

```
from django.db.models import Count, Window, F
from django.db.models.functions import DenseRank

dense_rank_by_album = Window(
        expression=DenseRank(),
        partition_by=F("album_id"),
        order_by=F("frequency").desc())

ranked_words = (Word.objects
        .filter(artist_id=self.object)
        .values('value', 'album__name')
        .annotate(frequency=Count('value'),
                  ranking=dense_rank_by_album)
        .order_by('album_id'))
```

Ranking backstreet boys vocabulary by frequency

The result being very long, we want to limit it to the words ranked 5 or less for each album.

Problem: This won't work

```
SELECT value as word,
       name as album_name,
       COUNT(word.value) AS frequency,
       DENSE_RANK() OVER (
           PARTITION BY word.album_id
           ORDER BY COUNT(word.value) DESC
       ) AS ranking
 FROM kyo_word word
 INNER JOIN kyo_album album
   ON (word.album_id = album.id)
WHERE word.artist_id = 13 AND ranking < 6
GROUP BY word.value,
      album.name,
      word.album_id
ORDER BY word.album_id ASC
```

Ranking backstreet boys vocabulary by frequency

```
SELECT * FROM (
  SELECT value,
      name as album_name,
      COUNT(word.value) AS frequency,
      DENSE_RANK() OVER (
          PARTITION BY word.album_id
          ORDER BY COUNT(word.value) DESC
      ) AS ranking
  FROM kyo_word word
  INNER JOIN kyo_album album
    ON (word.album_id = album.id)
  WHERE word.artist_id = 13
  GROUP BY word.value,
                album.name,
                word.album_id
  ORDER BY word.album_id ASC) a
WHERE a.ranking < 6 AND a.frequency > 5;
```

Instead of selecting FROM a table, we select from the query

This is our previous query wrapped into a subquery

We can now filter on ranking

Ranking backstreet boys vocabulary by frequency

Window Functions + Subquery - Django

In Django, you can't do a SELECT … FROM (subquery)

```
query = """
SELECT * FROM (…)
WHERE a.ranking < %s AND a.frequency > %s;" " "
"""

queryset = Word.objects.raw(query, [13, 6, 5])

for word in queryset:
    print(word.value, word.ranking)
```

Ranking backstreet boys vocabulary by frequency

Window Functions + Subquery - SQLAlchemy

```
subquery = session.query(
    Word.value,
    Album.name,
    func.count(Word.value).label('frequency'),
    func.dense_rank().over(
        partition_by=Word.album_id,
        order_by=desc(func.count(Word.value))))
.join('album').filter(Word.artist_id==13)
.group_by(Word.value, Album.name, Word.album_id)
.subquery('c)


session.query(Word).select_entity_from(subquery)
.filter(subquery.c.ranking <5)
```

For each album of the backstreet boys:
Window Functions + subquery - Activerecord

```
my_subquery =  Word
.where(artist_id: 13)
.joins(:album)
.group(:value, :name, :album_id)
.select('DENSE_RANK() OVER(PARTITION BY album_id ORDER BY
COUNT(value) DESC) AS ranking')

Word.select('*').from(my_subquery, :subquery).where('ranking < 6')
```

For each album of the backstreet boys:

Window Functions + subquery - Sequel

```
Word
.where(artist_id: 13)
.group(:value, :album_id)
.select{dense_rank.function.over(partition: :album_id,
                                 :order=>Sequel.desc(count(:value))).as(:ranking)}
.select_append(:album_id)
.from_self(:alias => 'subquery')
.where(Sequel[:ranking] > 5)
```

To go further

Window Functions: performs a **calculation across a set of rows**. Comparable to aggregate functions though each row remains in the result of the query.

AVG

RANK / DENSE_RANK

SUM

COUNT

Support with ORMs

	Django	SQLAlchemy	Activerecord	Sequel
Avg	Yes	Yes	No	Yes
Rank	Yes	Yes	No	Yes
Dense Rank	Yes	Yes	No	Yes
Sum	Yes	Yes	No	Yes
Count	Yes	Yes	No	Yes

GROUPING SETS

The goal of grouping sets is to have sub result in a query with different group by.

Here we want for all bands, the number of songs:

- Per album
- Per artist
- Per year, for all artist
- Total

GROUPING SETS

Result expected:

Artist	Album	Year	Number of songs
Maroon5	Songs About Jane	2002	12
Maroon5	It Won't Be Soon Before Long	2007	17
Maroon5	Hands All Over	2010	17
Maroon5	Overexposed	2012	16
Maroon5	V	2014	14
Maroon5	All albums		76
Selena Gomez	Kiss & Tell	2009	14
Selena Gomez	A Year Without Rain	2010	11
Selena Gomez	When The Sun Goes Down	2011	13
Selena Gomez	Stars Dance	2013	15
Selena Gomez	For You	2014	15
Selena Gomez	Revival	2015	16
Selena Gomez	All albums		84
All artists	All albums	1992	8
All artists	All albums	1994	30
All artists	All albums	1995	13
All artists	All albums	1997	26
All artists	All albums	1999	38
All artists	All albums	2000	39
All artists	All albums	2001	34
All artists	All albums	2002	40
…	…	…	…
All artists	All albums	2019	43
All artists	All albums		1012

GROUPING SETS

```
SELECT artist.name as artist_name,
       album.name as album_name,
       album.year,
       count(song.id) as nb_songs
  FROM kyo_artist artist
 INNER JOIN kyo_album album
    ON album.artist_id=artist.id
INNER JOIN kyo_song song
   ON song.album_id=album.id
GROUP BY GROUPING  SETS (
                (artist_name, album_name, year),
                (artist_name),
                (year),
                ())
ORDER BY 1, 3 ,4;
```

GROUPING SETS

SQLAlchemy

```
session.query(
    Album.name,
    Artist.name,
    Album.year,
    func.count(Song.id))
.join('songs')
.join('artist')
.group_by(func.grouping_sets((
      tuple_(Artist.name, Album.name, Album.year),
      tuple_(Artist.name),s
      tuple_(Album.year),
      tuple_())))
```

GROUPING SETS

Sequel

```
Album
.join(:artists, id: Sequel[:albums][:artist_id)]
.join(:songs, album_id: Sequel[:albums][:id])
.select{[
        Sequel[:artists][:name].as(:artist_name),
        Sequel[:albums][:name].as(:album_name),
        Sequel[:albums][:year].as(:year),
        count(Sequel[:songs][:id])]}
.group([:artist_name, :album_name, :year], [:artist_name], [:year], [])
.grouping_sets()
```

To go further

GROUPING SETS

CUBE (a, b, c) <-> GROUPING SETS ((a, b, c),
(a, b),
(a, c),
(a),
(b, c),
(b),
(c),
())

ROLLUP (a, b, c) <-> GROUPING SETS ((a, b, c),
(a, b),
(a),
())

Support with ORMs

	Django	SQLAlchemy	Activerecord	Sequel
GROUPING SETS	No	Yes	No	Yes
ROLLUP	No	Yes	No	Yes
CUBE	No	Yes	No	Yes

Common Table Expression (CTE)

- Defined by a WITH clause
- You can see it as a temporary table, private to a query
- Helps break down big queries in a more readable way
- A CTE can reference other CTEs within the same WITH clause (Nest). A subquery cannot reference other subqueries
- A CTE can be referenced multiple times from a calling query. A subquery cannot be referenced.

Common Table Expression (CTE)

```
WITH last_album AS (
SELECT album.id FROM kyo_album album
WHERE artist_id = 15
ORDER BY year DESC LIMIT 1),

older_songs AS (
SELECT song.id FROM kyo_song song
INNER JOIN kyo_album album ON (album.id = song.album_id)
WHERE album_id NOT IN (SELECT id FROM last_album)
AND album.artist_id=15
)

SELECT value, COUNT(*) FROM kyo_word
INNER JOIN last_album ON kyo_word.album_id=last_album.id
WHERE value NOT IN (
    SELECT value
    FROM kyo_word
    INNER JOIN older_songs ON kyo_word.song_id=older_songs.id)
GROUP BY value ORDER BY 2 DESC;
```

Common Table Expression (CTE)

value	count
sorri	21
journey	9
mark	8
blame	6
direct	4
wash	4
children	4
serious	4
human	3
delusion	3
disappoint	2
confus	2

Support with ORMs

	Django	SQLAlchemy	Activerecord	Sequel
WITH	No	Yes	No	Yes

Need them? Any ORM allows you to do raw SQL

Things I haven't talked about

- Indexes
- Constraints
- Fulltext search (fully supported in the Django ORM)
- ´Recursive CTE
- INSERT / UPDATE / DELETE
- INSERT … ON CONFLICT
- …

Conclusion

Here are the features we saw today and their compatibility with Django ORM

	Django	SQLAlchemy	Activerecord	Sequel
Aggregations	Yes	Yes	Yes	Yes
DISTINCT	Yes	Yes	Yes	Yes
GROUP BY	Yes	Yes	Yes	Yes
(NOT) EXISTS	Yes	Yes	Yes	Yes
Subqueries in SELECT	Yes	Yes	No	Yes
Subqueries in FROM	No	Yes	Yes	Yes
Subqueries in WHERE	Yes	Yes	Kind of	Yes
LATERAL JOIN	No	Yes	No	Yes
Window functions	Yes	Yes	No	Yes
GROUPINGS SETS	No	Yes	No	Yes
CTE	No	Yes	No	Yes

Thank you for your attention!

louisemeta.com

citusdata.com/newsletter

@louisemeta @citusdata