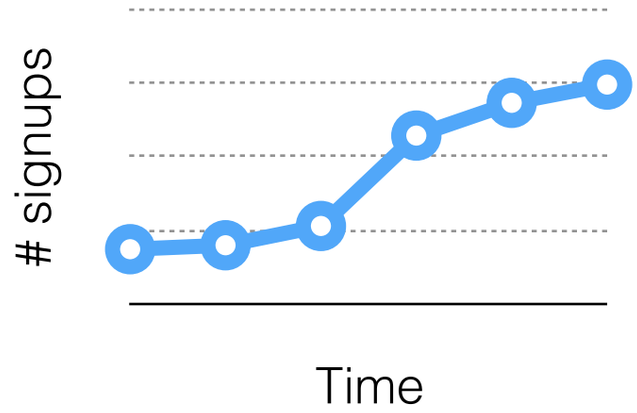
github.com/wxgeorge/pycon-ca-2016/blob/master/slides.pdf

Databases 201

The power of the relational algebra
and APIs beyond the ORM

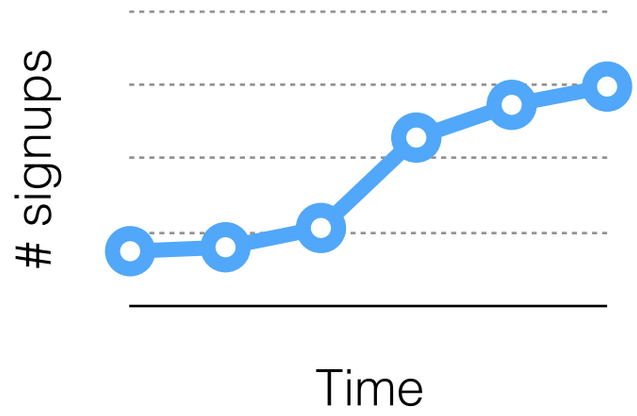
Wesley George
Technical Lead, Clearbanc

Talk Origin




```
months = [ June2016, July2016, ... November2016 ]
counts_by_month = { k:0 for k in months }

for u in User.objects.all:
    signup_month = get_month(u.signup_date)
    counts_by_month[signup_month]+=1
```

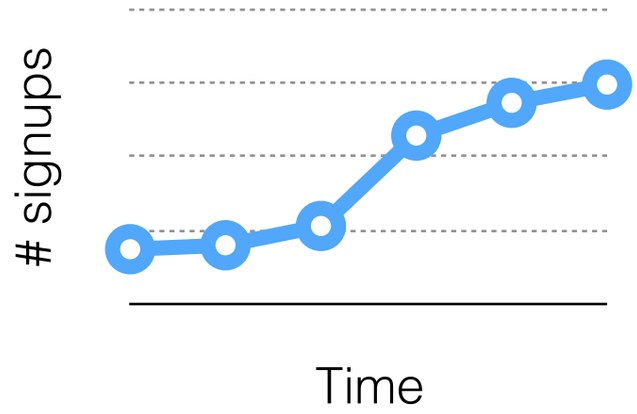
Talk Origin




```
months = [ June2016, July2016, ... November2016 ]
counts_by_month = { k:0 for k in months }

for u in User.objects.all:
    signup_month = get_month(u.signup_date)
    counts_by_month[signup_month]+=1
```

ending with …

Talk Origin




```
months = [ June2016, July2016, ... November2016 ]
counts_by_month = { k:0 for k in months }

for u in User.objects.all:
  signup_month = get_month(u.signup_date)
  counts_by_month[signup_month]+=1
```

ending with …

a light-weight system
for business intelligence

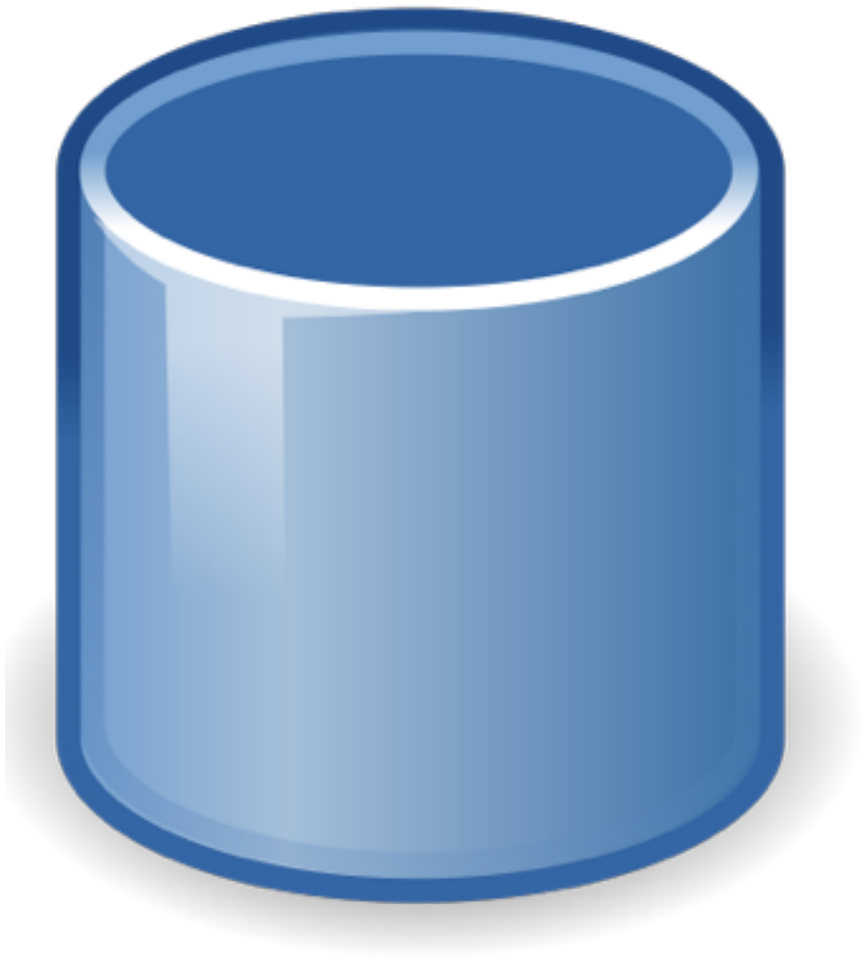
Agenda

- the capabilities of the (relational) databases (Along with what we're tap into with the ORM)

- explore of the power of the database as a compute engine through some examples of sophisticated computations done only in SQL

- explore tools for managing the complexity of query collections and reporting pipelines, and when these should be broken down

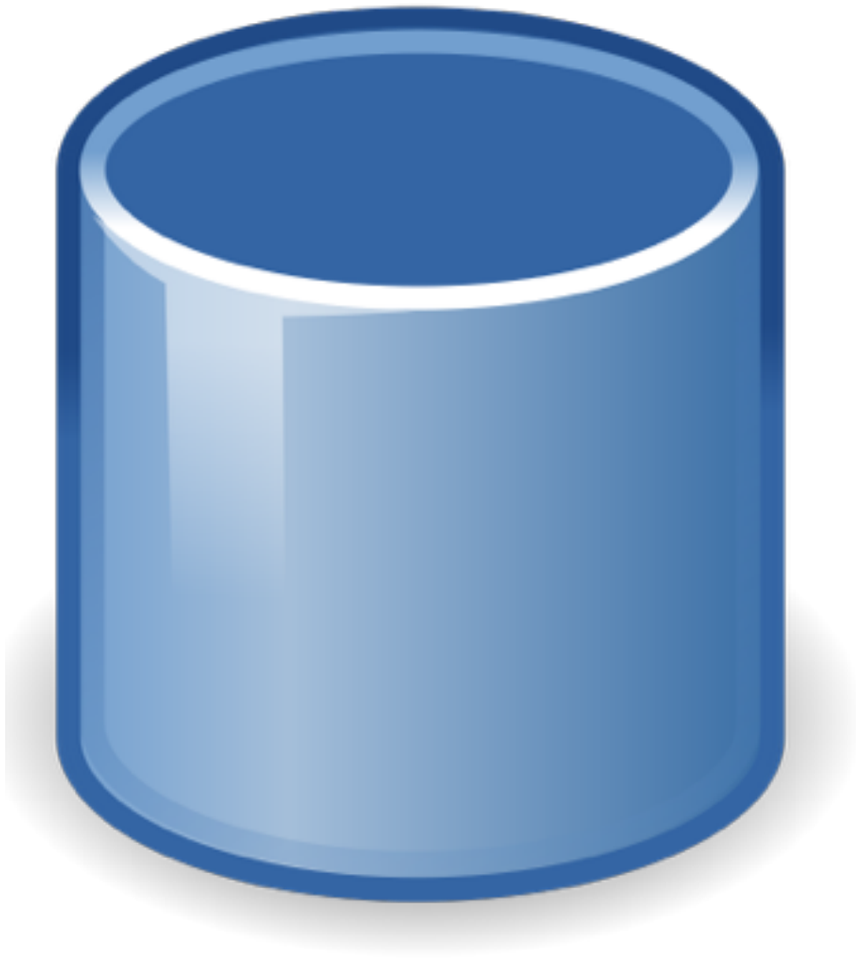
Databases



**https://commons.wikimedia.org/wiki/File:Database.svg

Databases

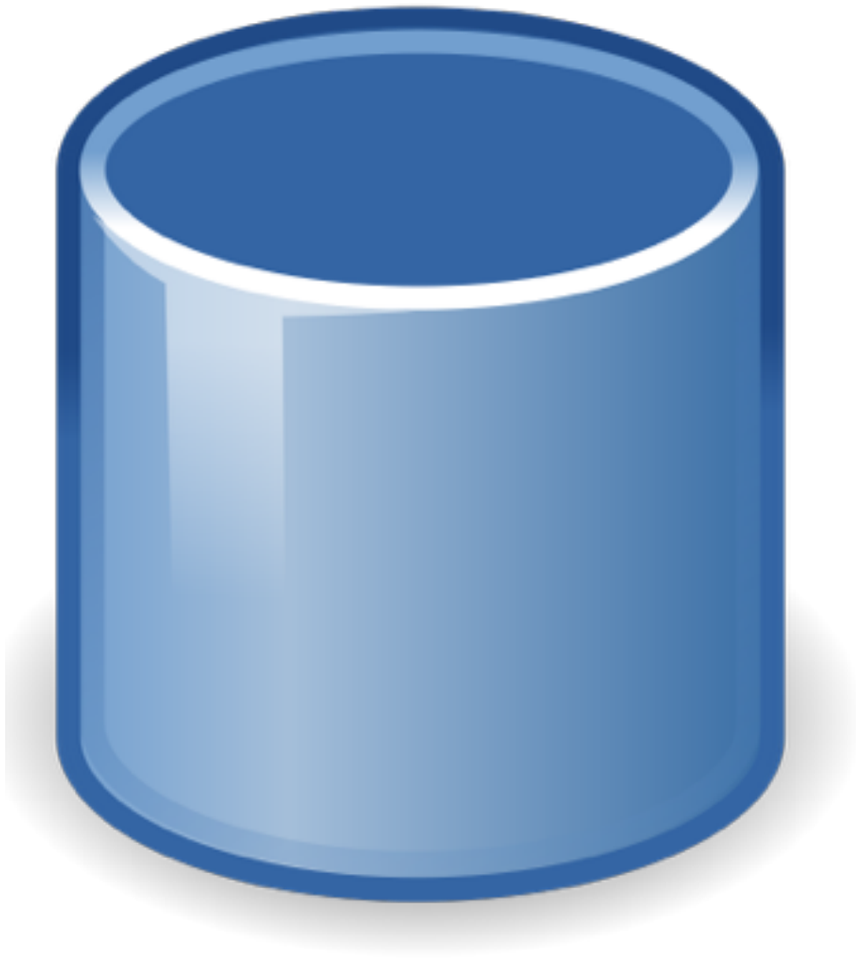
- Persistence



**https://commons.wikimedia.org/wiki/File:Database.svg

Databases

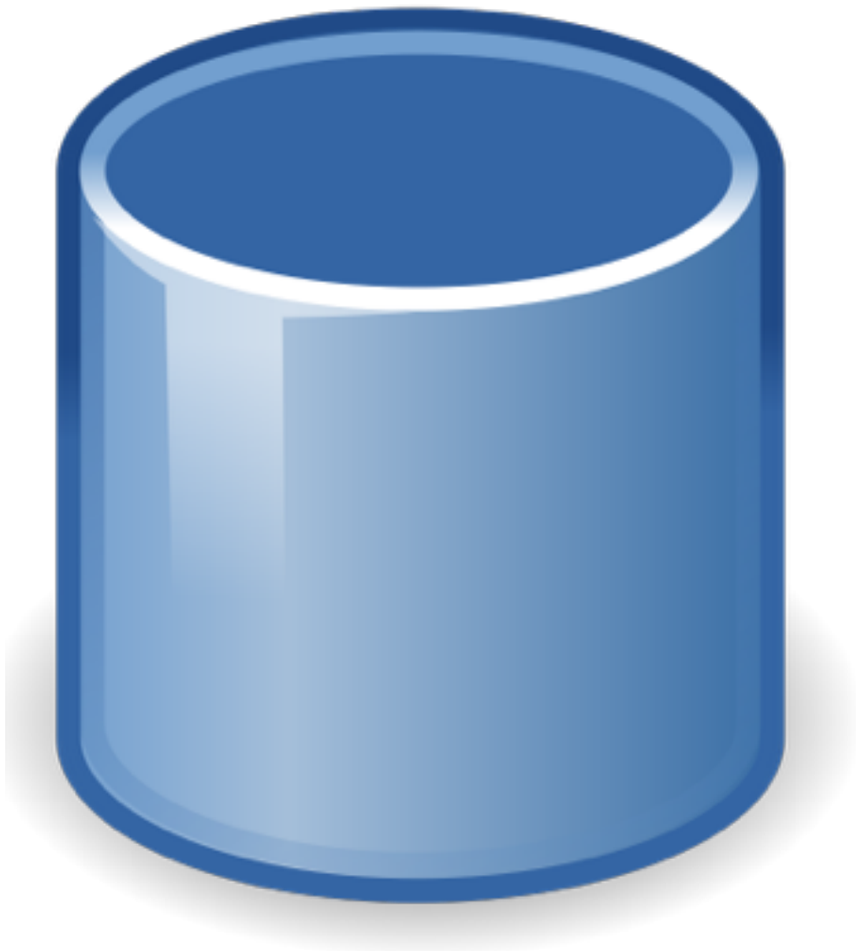
- Persistence
- Concurrency



**https://commons.wikimedia.org/wiki/File:Database.svg

Databases

- Persistence
- Concurrency
- Computing
  (i.e. reporting, summarizing)

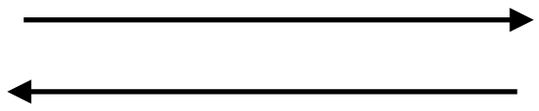


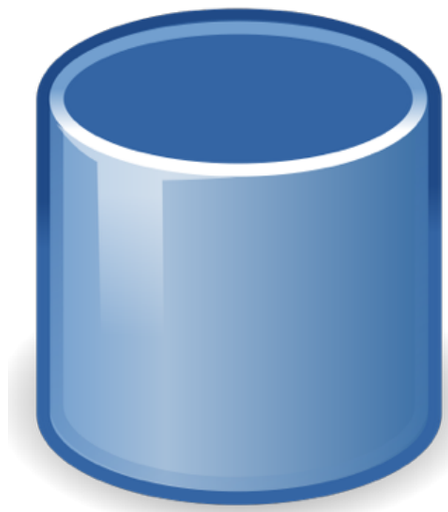
**https://commons.wikimedia.org/wiki/File:Database.svg

the ORM

```python
class User(models.Model):
  name=models.CharField(max_length=100)
  email=models.CharField(max_length=200)


User.new(
  name="Wes",
  email="wes@clearbanc.com")

user.name = "Wesley"
user.save

user.delete
```

```sql
CREATE TABLE user
  name          varchar(100) NOT NULL,
  email         varchar(200) NOT NULL,
);

INSERT INTO user VALUES (name, email) (
  ('Wes', 'wes@clearbanc.com')
);

UPDATE user SET name='Wesley' WHERE id=1;


DELETE FROM user WHERE id=1;
```



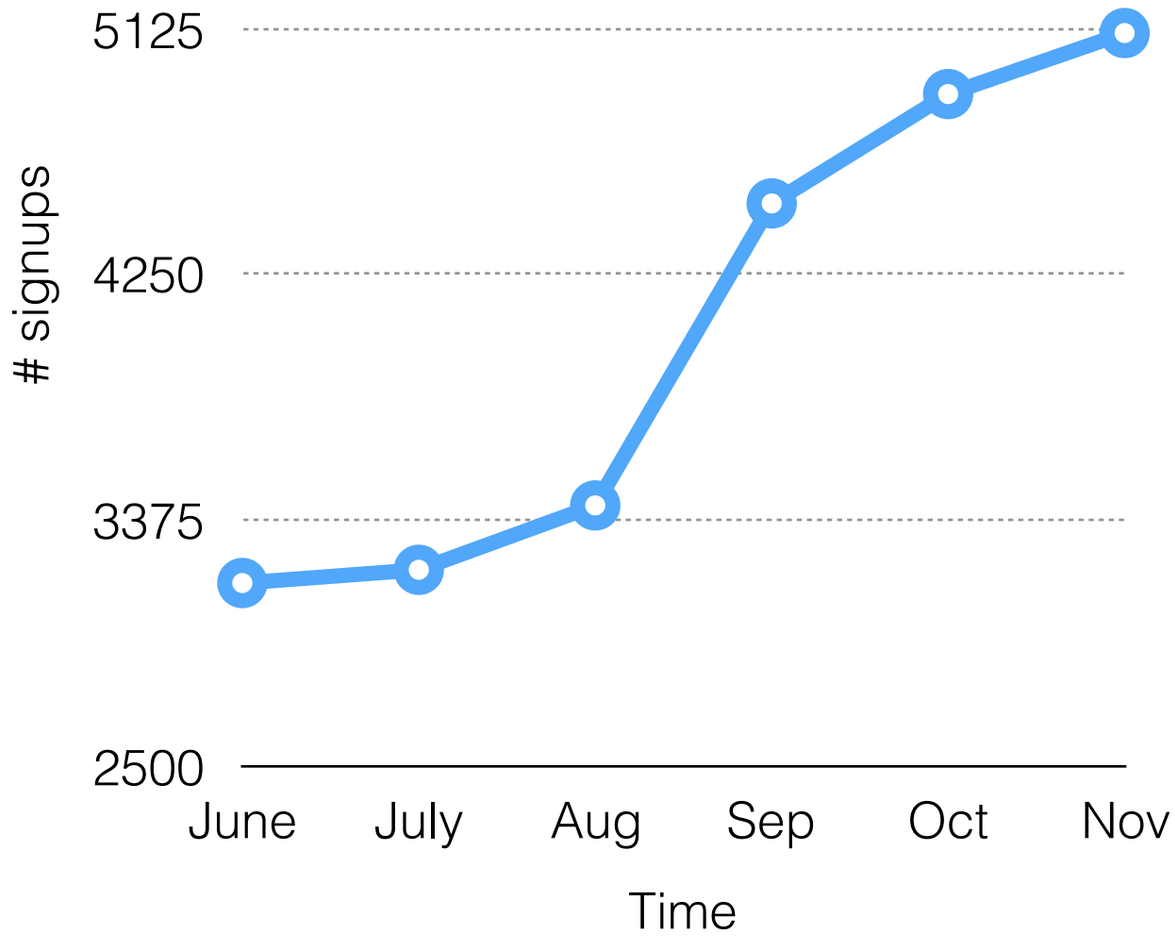


the ORM

```
# Some examples of aggregation from the Django tutorial:
Book.objects.aggregate(Avg('price'))
Book.objects.annotate(Count('authors'), Count('store'))
Store.objects.aggregate(youngest_age=Min('books__authors__age'))

# There is power here,
# and this plays nicely Django's association,
# but these are all are rooted in single tables and pre-existing columns
```

Example 1

user	created_at
1	2016-07-01 14:34:26
2	2016-07-02 17:18:23
3	2016-07-02 19:44:91



SQL building blocks

```
SELECT COUNT(*)
  FROM users
```

```
 count
--------
 100000
(1 row)
```

SQL building blocks

```
SELECT COUNT(*), state
  FROM users
  GROUP BY state
```

```
 count |     address_state
-------+------------------------
   267 | AK
   704 | AL
  4719 | AZ
       ...
  5313 | WA
  1245 | WI
(52 rows)
```

SQL building blocks

```
SELECT COUNT(*), age - age % 10
  FROM users
  GROUP BY age - age % 10
```

```
 count  | bin
--------+-----
   100  |  10
 24000  |  20
 56000  |  30
 82300  |  40
 34000  |  50
 18600  |  60
   100  | 110
(7 rows)
```

SQL building blocks

```
WITH
  users_with_age AS (
    SELECT *, EXTRACT('years' FROM clock_timestamp() - birthday) AS age
      FROM users
  )
SELECT COUNT(*), age - age % 10
  FROM users_with_age
  GROUP BY age - age % 10
```

```
 count  | bin
--------+-----
   100  |  10
 24000  |  20
 56000  |  30
 82300  |  40
 34000  |  50
 18600  |  60
   100  | 110
(7 rows)
```

SQL building blocks

```sql
WITH
  users_with_age AS (
    SELECT *, EXTRACT('years' FROM clock_timestamp() - birthday) AS age
      FROM users
  ),
  users_with_bin AS (
    SELECT *, age - age % 10 as bin
      FROM users_with_age
  )
SELECT COUNT(*), bin
  FROM users_with_bin
  GROUP BY bin
```

```
 count  | bin
--------+-----
   100  |  10
 24000  |  20
 56000  |  30
 82300  |  40
 34000  |  50
 18600  |  60
   100  | 110
(7 rows)
```

SQL building blocks

```
SELECT users.name, addresses.state
  FROM users
  JOIN addresses ON users.address_id = addresses.id
```

```
 id |
----
  1
  2
```

JOIN

```
 user_id | tag
---------+-----
       1 | 'A'
       2 | 'B'
```

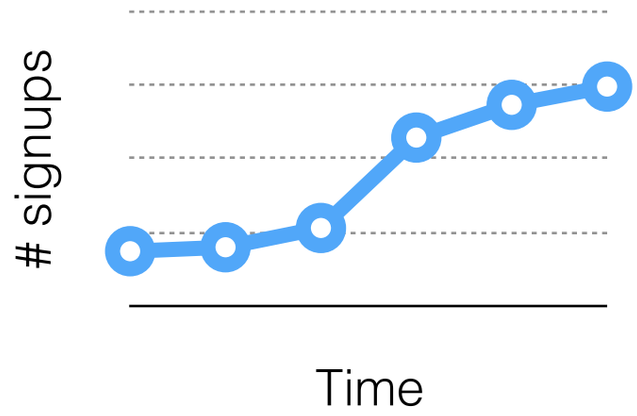
=

```
 id | user_id | tag
----+---------+-----
  1 |       1 | 'A'
  2 |       1 | 'A'
  1 |       2 | 'B'
  2 |       2 | 'B'

        |
        V

 ON id = user_id

        |
        V

 id | user_id | tag
----+---------+-----
  1 |       1 | 'A'
  1 |       2 | 'B'
```

SQL building blocks

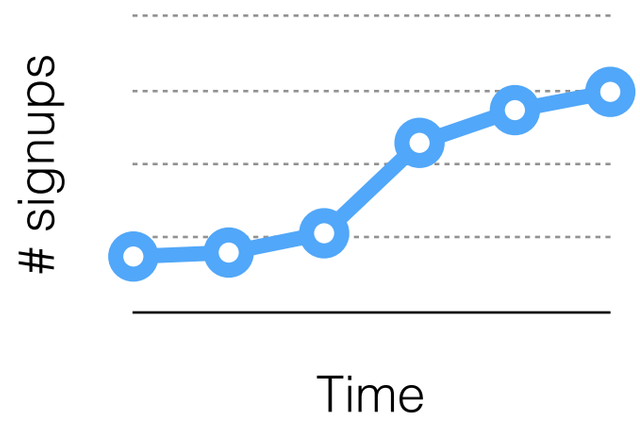
```
SELECT *
  FROM users
  JOIN events ON events.user_id = users.id
             AND events.created_at > users.time_of_last_outreach
```

Example 1




```
months = [ June2016, July2016, ... November2016 ]
counts_by_month = { k:0 for k in months }

for u in User.objects.all:
    signup_month = get_month(u.signup_date)
    counts_by_month[signup_month]+=1
```

Example 1: SQL



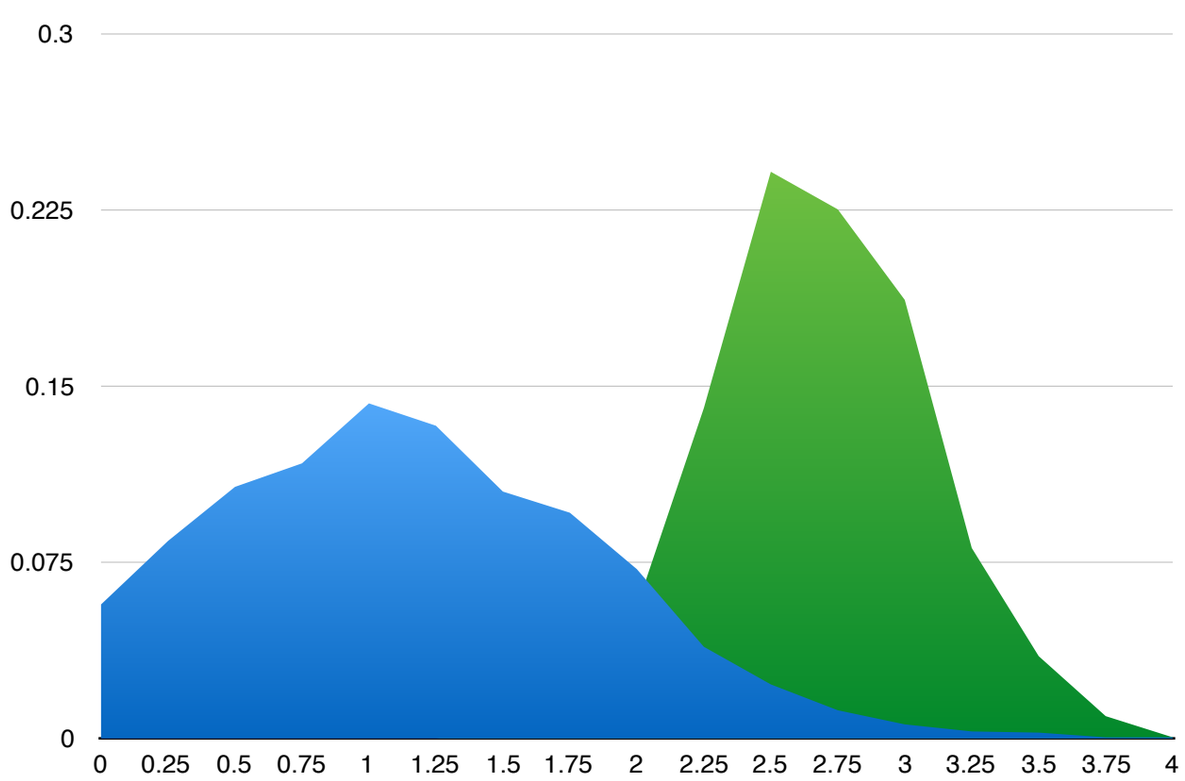

```sql
WITH
  months AS (
    VALUES (name, start_date, end_date)
    (
      ("jun2016", "2016-06-01", "2016-06-30"),
      ("jul2016", "2016-07-01", "2016-07-31"),
      ("aug2016", "2016-08-01", "2016-08-31"),
      ("sep2016", "2016-09-01", "2016-09-30"),
      ("oct2016", "2016-10-01", "2016-10-31"),
      ("nov2016", "2016-11-01", "2016-11-30"),
    )
  )
,
users_with_month AS (
  SELECT months.name AS signup_month, users.*
    FROM users
    JOIN months ON months.start_date <= users.signup_date
               AND                     users.signup_date <= months.end_date
)
SELECT signup_month, COUNT(*)
  FROM users_with_month
  GROUP BY signup_month
```

Example 2

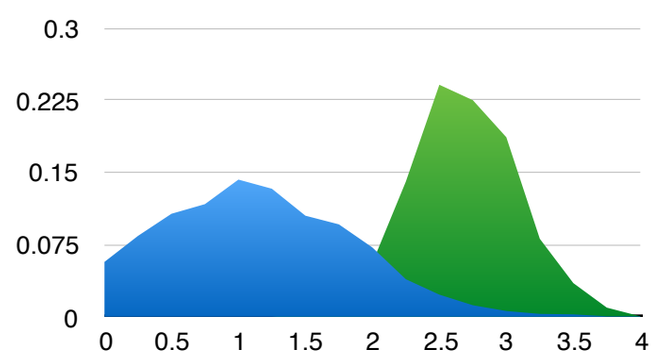
(distribution of)
user engagement change
segmented by campaign

user	treatment	treatment_time
1	A	2016-07-01
2	B	2016-07-21

user	event	created_at
1	1	2016-07-01

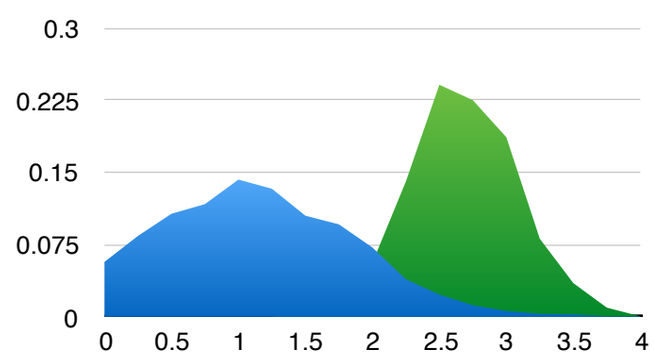


Example 2



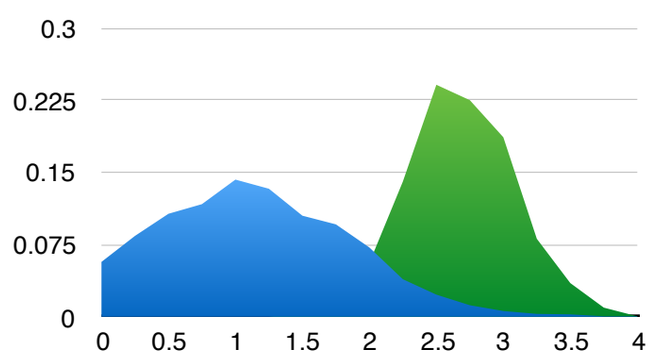

```
WITH
  users AS (
    SELECT users.*, id AS user_id, marketing_event.created_at AS treatment_time
      JOIN marketing_events USING (user_id)
      WHERE campaign = 'birthday-email-blast';
  ),
  events_pre AS (
    SELECT user_id, events.created_at::date, COUNT(*) AS n_events
      FROM users
      JOIN events
        ON events.user_id = user_id
       AND (
            events.created_at <  treatment_time AND
            events.created_at >= treatment_time - '30 days'::interval
           )
    GROUP BY users.id, events.created_at::date
),
events_post AS (
  SELECT user_id, events.created_at::date, COUNT(*) AS n_events
    FROM users
    JOIN events
      ON events.user_id = user_id
     AND (
          events.created_at >  treatment_time AND
          events.created_at <= treatment_time + '7 days'::interval
         )
    GROUP BY users.id, events.created_at::date
),
```

Example 2



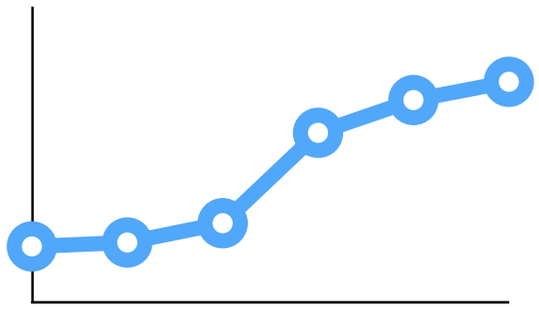

```
events_post AS (
  SELECT user_id, events.created_at::date, COUNT(*) AS n_events
    FROM users
    JOIN events
      ON events.user_id = user_id
     AND (
           events.created_at >  treatment_time AND
           events.created_at <= treatment_time + '7 days'::interval
         )
    GROUP BY users.id, events.created_at::date
),

activity_pre  AS (SELECT user_id, AVG(n_events) AS activity_pre FROM events_pre  GROUP BY user_id),
activity_post AS (SELECT user_id, AVG(n_events) AS activity_pre FROM events_post GROUP BY user_id),
activity_changes AS (
  SELECT *, activity_pre/activity_post AS increase_factor
    FROM users
    JOIN activity_pre  USING (user_id)
    JOIN activity_post USING (user_id)
),
```

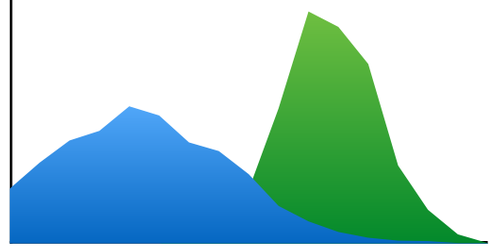
Example 2

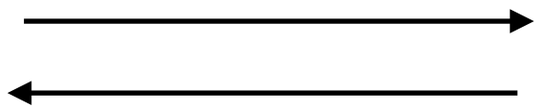


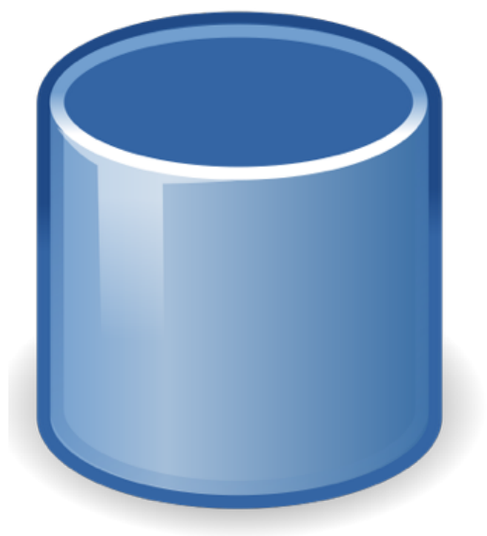

```
increase_factor_distribution_pre_1 AS (
  SELECT *,
    increase_factor * 10 - ((increase_factor * 10) % 5) / 10.0 AS bin
    FROM activity_changes_by_user
),

increase_factor_distribution_pre_2 AS (
  SELECT treatment, bin, COUNT(*) AS count
    FROM increase_factor_distribution_pre_1
    GROUP BY treatment, bin
),

increase_factor_distribution AS (
  SELECT treatment, bin, count, count/population.count AS mass
    FROM increase_factor_distribution_pre_2
    JOIN (
            SELECT treatment, COUNT(*) FROM users GROUP BY treatment
         ) AS population USING (treatment)
)

SELECT * FROM increase_factor_distribution
```

Concerning Efficiency









Managing Complexity

```
events_pre AS (
  SELECT user_id, events.created_at::date, COUNT(*) AS n_events
    FROM users
    JOIN events
      ON events.user_id = user_id
     AND (
           events.created_at <  users.birthday AND
           events.created_at >= users.birthday - '30 days'::interval
         )
    GROUP BY users.id, events.created_at::date
),
events_post AS (
  SELECT user_id, events.created_at::date, COUNT(*) AS n_events
    FROM users
    JOIN events
      ON events.user_id = user_id
     AND (
           events.created_at >  users.birthday AND
           events.created_at <= users.birthday + '7 days'::interval
         )
    GROUP BY users.id, events.created_at::date
),
```

Managing Complexity

```
# SQLAlchemy Core gives different objects to work with.
# We get Python objects for the range of types that exist in a database.

events = select([users.id, func.date(events.created_at), func.count()]).
         select_from(user.join(events, events.c.user_id = users.c.id )).
         group_by([users.id, func.date(events.created_at)])
```

```
mt = users.marketing_time
events_pre  = events.where([ mt - '30 days', mt             ])
events_post = events.where([ mt            , mt + '7 days' ])

c = [users.id, func.avg(events_pre.count)]
activity_pre  = select(c).select_from(events_pre ).group_by(users.id)
activity_post = select(c).select_from(events_post).group_by(users.id)

activity_increase = select([users.id, ..., activity_pre.avg/activity_post.avg]).
                    select_from(users.join(activity_pre).join(activity_post))
```

Managing Complexity

```
# Some examples of aggregation from the Django tutorial:
Book.objects.aggregate(Avg('price'))
Book.objects.annotate(Count('authors'), Count('store'))
Store.objects.aggregate(youngest_age=Min('books__authors__age'))

# There is power here,
# and this plays nicely Django's association,
# but these are all are rooted in single tables and pre-existing columns
```

Thanks!

- wesley.george@gmail.com

- @wxalistair