# SQL FOR INTERVIEWS

*February 21, 2017*

*Columbia Data Science Society*

# Agenda

01 Introduction

02 Setup

03 Useful commands

04 General Tips

05 Interview Questions

06 Q&A

# Introduction

## *Background*

- Structured Query Language (SQL) has been around for decades and is the foundation for working with databases.
- There are several different types (T-SQL, PL/SQL, PostgreSQL) depending on the database that you're working with, but most are very similar.

## *For this workshop*

- We've chosen to work with a Postgres database because has lots of nifty features but is still free
- The next slide provides some setup information for Postgres, its GUI interface Postico, and the example database that we'll be working with

# Setup

## *Instructions*

- https://eggerapps.at/postico/docs/v1.1.1/

## *Postgres*

- http://postgresapp.com/

## *Postico (Mac-specific)*

- https://eggerapps.at/postico/

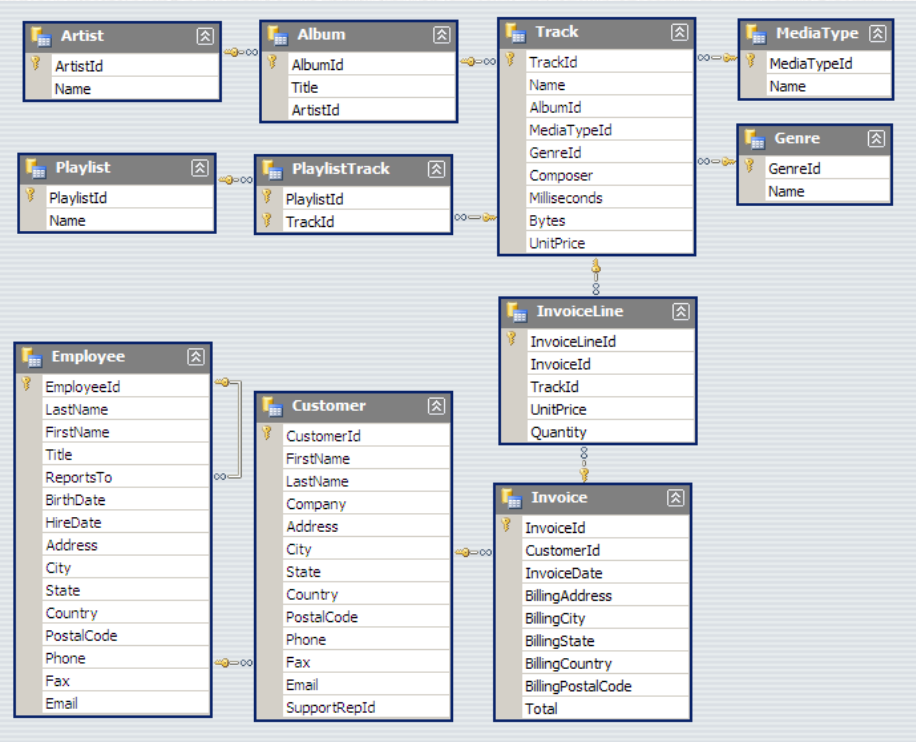
## *Chinook Database*

- https://raw.githubusercontent.com/xivSolutions/ChinookDb_Pg_Modified/master/chinook_pg_serial_pk_proper_naming.sql
- Just paste the above script into Postico and execute

# Concepts

## *Tables*

- Databases store data in tables, where each row is a sample and each column is a feature
- The tables, their columns, and the data types need to be defined in advance
- All tables should have a *primary key* column, or a unique identifier for every row



# Examples

## *Selecting Data*

- Say you want to know everything in the "Track" table
- Use * to grab all the columns from the table

```
1 select * from track
```

Load Query... Save Query...

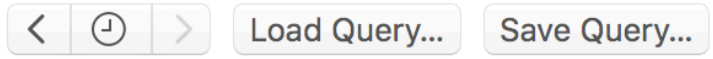
| track_id | name | album_id | media_type_id | genre_id | composer |
|---|---|---|---|---|---|
| 2819 | Battlestar Galactica: The Story So Far | 226 | 3 | 18 | NULL |
| 2820 | Occupation / Precipice | 227 | 3 | 19 | NULL |
| 2821 | Exodus, Pt. 1 | 227 | 3 | 19 | NULL |
| 2822 | Exodus, Pt. 2 | 227 | 3 | 19 | NULL |
| 2823 | Collaborators | 227 | 3 | 19 | NULL |

# Examples

## *Selecting Data*

- Now suppose you want only the tracks with a price > 0.99.
- Use the “where” clause

```
1 select * from track
2 where unit_price > 0.99
```




| track_id | name | album_id | media_type_id | genre_id | composer |
|---|---|---|---|---|---|
| 2819 | Battlestar Galactica: The Story So Far | 226 | 3 | 18 | NULL |
| 2820 | Occupation / Precipice | 227 | 3 | 19 | NULL |
| 2821 | Exodus, Pt. 1 | 227 | 3 | 19 | NULL |
| 2822 | Exodus, Pt. 2 | 227 | 3 | 19 | NULL |

# Examples

## *Selecting Data*

- There are many options for filtering using where clauses

```
1 select * from track
2 where track_id in (1,2)
```

Use “in” to select from among a list

```
1 select * from track
2 where composer like '%Dave%'
```

Use “like” to search for parts of strings

```
1 select count(*) from track
```

Use “count” for the number of rows rather than the actual results

```
1 select count(distinct composer) from track
```

Or use “count distinct” for the number of unique entries

```
1 select composer from track
2 order by composer
```

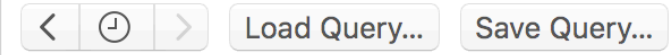
Use “order by” to sort the results

# Examples

## *Aggregating*

- Let's say that instead of a count of the distinct composers, you also want to know how many songs each composer has
- Use a "group by" clause to aggregate results on your chosen field

```
1 select composer, count(*) from track
2 group by composer
3 order by count(*) desc
```




| composer | count |
|---|---|
| NULL | 978 |
| Steve Harris | 80 |
| U2 | 44 |
| Jagger/Richards | 35 |

# Examples

## *Aggregating*

- You can also use a number of other aggregating functions similarly– avg, max, min, etc.

```
1 select composer, max(milliseconds) from track
2 group by composer
3 order by max(milliseconds) desc
```

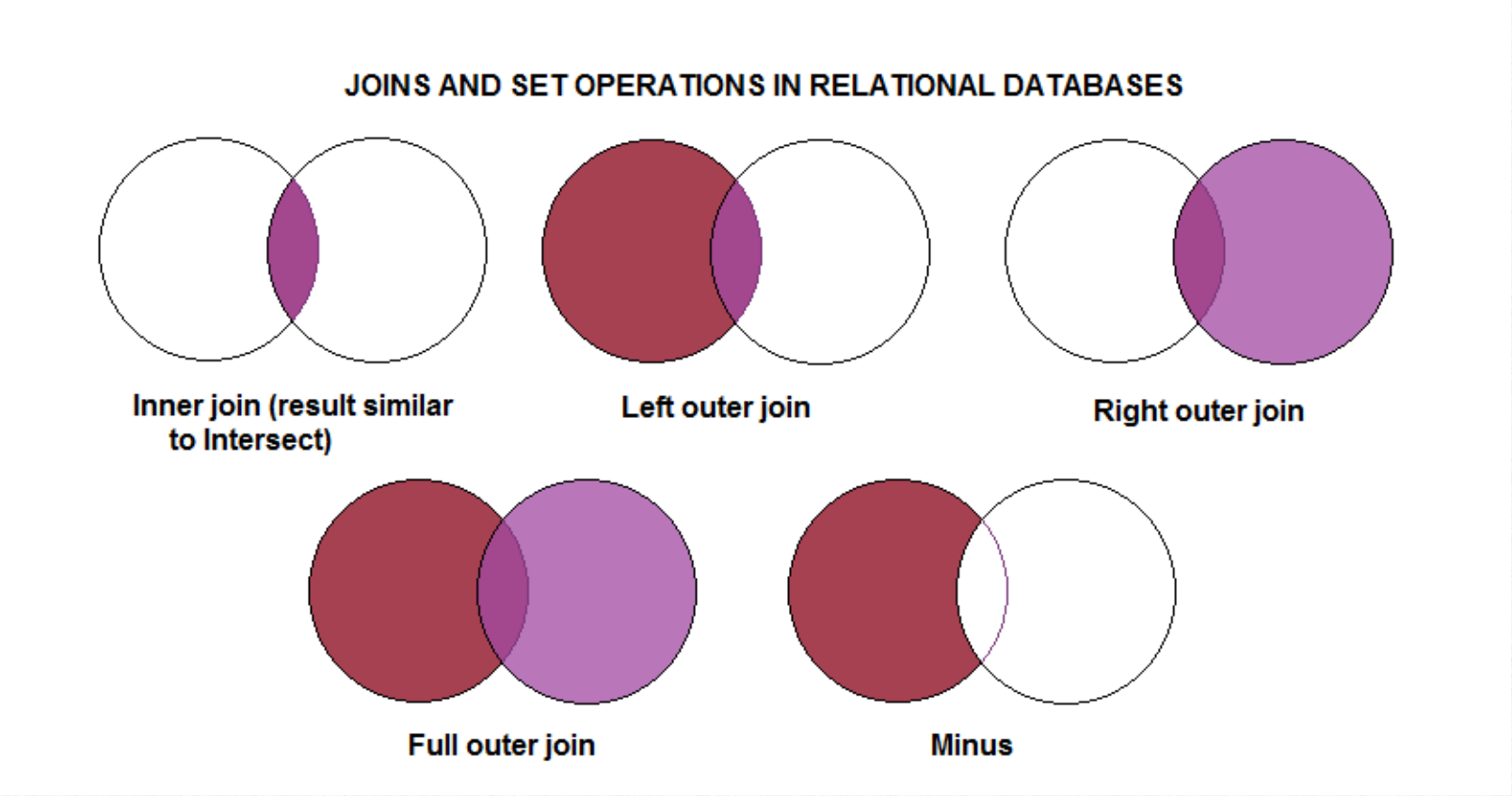
Load Query... Save Query...

| composer | max |
|---|---|
| NULL | 5286953 |
| Jimmy Page | 1612329 |
| Blackmore/Gillan/Glover/Lord/Paice | 1196094 |
| Jimmy Page/Led Zeppelin | 1116734 |

# Examples

## *Joins!*

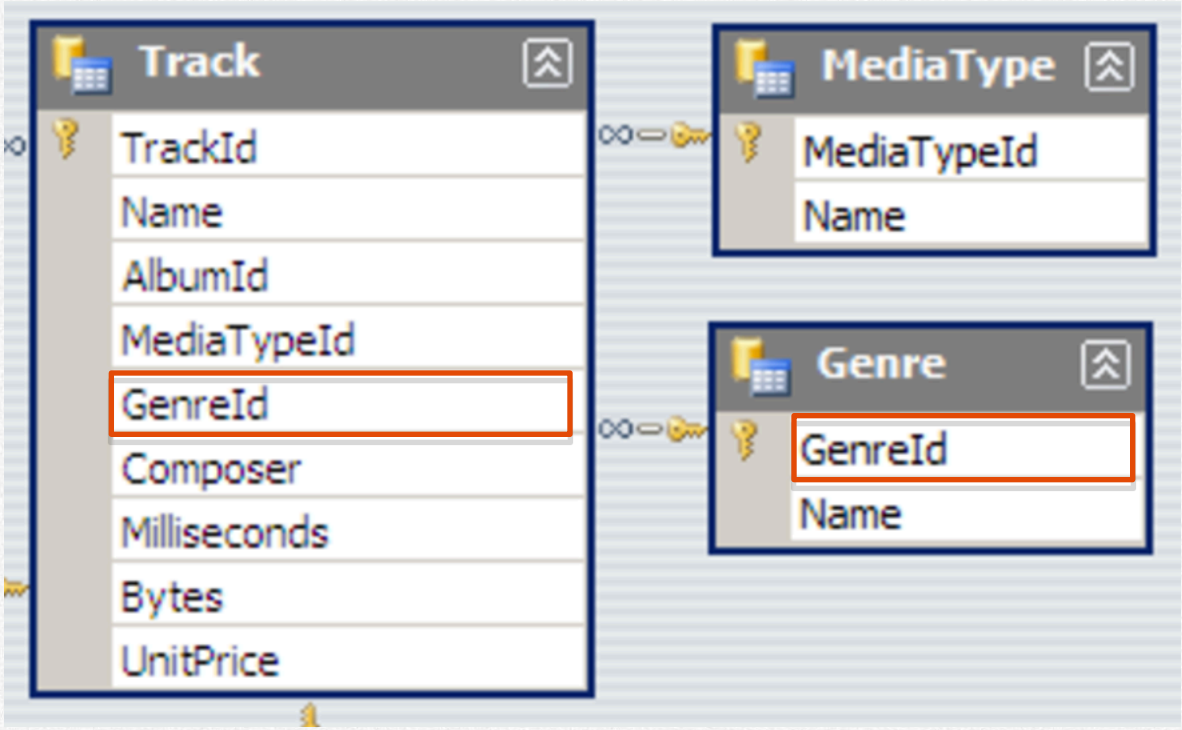
- Joins are used to combine tables
- This is where primary keys become important– they are generally used to link two tables



# Examples

## *Joins!*

- Let's say you want to know how many tracks there are per genre
- The Track table only has genreid, not genre, so we'll need to join to the Genre table



# Examples

*Joins!*

```
1 select g.genre_id, g.name, count(*) from track t
2 left join genre g on t.genre_id = g.genre_id
3 group by g.genre_id, g.name
4 order by count(*) desc
```

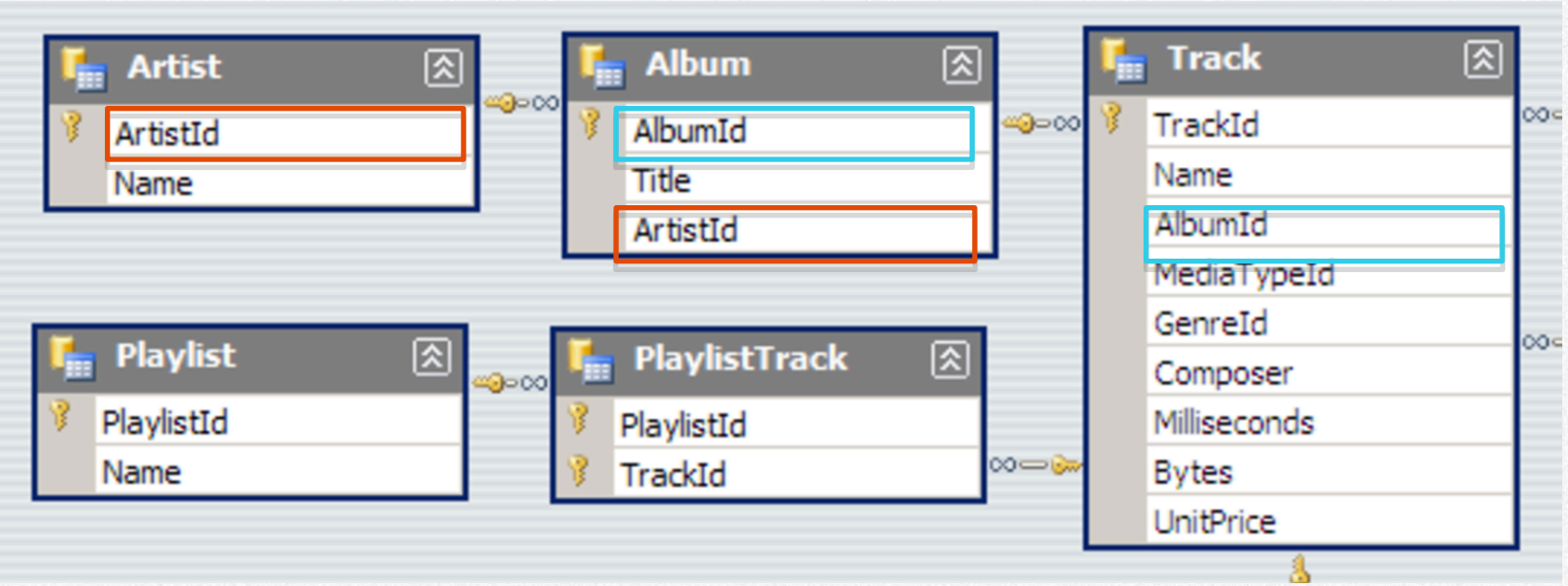
Load Query... Save Query...

| genre_id | name | count |
|---|---|---|
| 1 | Rock | 1297 |
| 7 | Latin | 579 |
| 3 | Metal | 374 |

# Examples

## *Nested Queries*

- Now let's say you want the number of tracks per artist. Artist is two tables away from track, but because we have keys linking all the tables, we can write a subquery

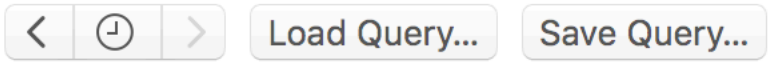


# Examples

## *Nested Queries*

- Just write a query, then treat it the same way you would treat a table

```
1 select a.name, count(*) from track t
2 left join (
3     select * from album alb
4     left join artist art on alb.artist_id = art.artist_id) a
5 on t.album_id = a.album_id
6 group by a.name
7 order by count(*) desc
```




| name | count |
|---|---|
| Iron Maiden | 213 |
| U2 | 135 |
| Led Zeppelin | 114 |
| Metallica | 112 |

# General Tips

## *Deleting Data*

- There are 3 different ways to delete data, with *very* different results:
  - Delete: Removes rows that meet a given where clause
  - Truncate: Delete all rows in a table
  - Drop: Remove a table entirely

## *Optimization*

- Getting the right data is pretty easy, but getting it quickly can be a problem
  - Only query data you need
  - Filter, then aggregate/join
  - If you want even more performance improvements, learn about indexing

# CASE EXAMPLE

# Hungry for Data Science

Given the following table schema:

What is the month-over-month trend in takeout sales from restaurants that are both:

1) Good for groups

and do *not*

2) Take reservations

???

```
sql_interviews=> \d+ restaurants
                                Table "public.restaurants"
    Column     |       Type        | Modifiers | Storage  | S
---------------+-------------------+-----------+----------+--
 restaurant_id | integer           | not null  | plain    |
 name          | character varying |           | extended |
 location      | character varying |           | extended |
Indexes:
    "restaurants_pkey" PRIMARY KEY, btree (restaurant_id)

sql_interviews=> \d+ restaurant_features
                           Table "public.restaurant_features"
    Column     |       Type        | Modifiers | Storage  | S
---------------+-------------------+-----------+----------+--
 restaurant_id | integer           |           | plain    |
 feature       | character varying |           | extended |

sql_interviews=> \d+ takeout_orders
                           Table "public.takeout_orders"
    Column     |  Type   | Modifiers | Storage | Stats target
---------------+---------+-----------+---------+-------------
 order_id      | integer | not null  | plain   |
 restaurant_id | integer |           | plain   |
 order_date    | date    |           | plain   |
 order_amount  | integer |           | plain   |
Indexes:
    "takeout_orders_pkey" PRIMARY KEY, btree (order_id)
```

# Step 1:

***Key Idea:***

Identify the relevant subset of restaurants

```
sql_interviews=> select * from restaurant_features order by restaurant_id;
 restaurant_id |        feature
---------------+-----------------------
             1 | Cash Only
             1 | Takes Reservations
             1 | Good for Groups
             2 | Outdoor Seating
             2 | Good for Groups
             3 | Coat Check
             3 | Has TV
             4 | Good for Groups
             4 | Wheelchair Accessible
             5 | Has TV
             5 | Outdoor Seating
             5 | Good for Groups
             5 | Coat Check
(13 rows)

sql_interviews=>
sql_interviews=>
sql_interviews=>
sql_interviews=> SELECT restaurant_id
FROM restaurant_features
WHERE feature = 'Good for Groups';
 restaurant_id
---------------
             1
             2
             4
             5
(4 rows)
```

# Step 1, cont.

***Key Idea:***

Self-join and only take rows that are NULL for the value you want to exclude

| restaurant_id | rf1.feature | rf2.feature |
|---|---|---|
| 1 | Good for Groups | Takes Reservations |
| 2 | Good for Groups | NULL |
| 4 | Good for Groups | NULL |
| 5 | Good for Groups | NULL |

```
sql_interviews=> SELECT rf1.restaurant_id
sql_interviews-> FROM restaurant_features rf1
sql_interviews-> LEFT JOIN restaurant_features rf2 ON rf1.restaurant_id = rf2.restaurant_id
sql_interviews-> AND rf2.feature = 'Takes Reservations'
sql_interviews-> WHERE rf1.feature = 'Good for Groups' AND rf2.feature IS NULL;
 restaurant_id
---------------
             2
             4
             5
(3 rows)
```

# Step 2

**Key Idea:**

Explore the data, keeping in mind that we want to aggregate takeout sales by month

```
sql_interviews=>
sql_interviews=> select * from takeout_orders
sql_interviews-> ;
 order_id | restaurant_id | order_date | order_amount
----------+---------------+------------+--------------
        1 |             1 | 2016-01-01 |            8
        2 |             1 | 2016-02-01 |           15
        3 |             1 | 2016-02-01 |            3
        4 |             1 | 2016-03-01 |           37
        5 |             1 | 2016-03-01 |           48
        6 |             1 | 2016-03-01 |            5
        7 |             2 | 2016-01-01 |           17
        8 |             2 | 2016-02-01 |           43
        9 |             2 | 2016-02-01 |           15
       10 |             2 | 2016-03-01 |           35
       11 |             2 | 2016-03-01 |           22
       12 |             2 | 2016-03-01 |           50
       13 |             3 | 2016-01-01 |           16
       14 |             3 | 2016-02-01 |            9
       15 |             3 | 2016-02-01 |           26
       16 |             3 | 2016-03-01 |            2
       17 |             3 | 2016-03-01 |           43
       18 |             3 | 2016-03-01 |           18
       19 |             4 | 2016-01-01 |           38
       20 |             4 | 2016-02-01 |           22
       21 |             4 | 2016-02-01 |            1
       22 |             4 | 2016-03-01 |            9
       23 |             4 | 2016-03-01 |            1
       24 |             4 | 2016-03-01 |           43
       25 |             5 | 2016-01-01 |           13
       26 |             5 | 2016-02-01 |           36
       27 |             5 | 2016-02-01 |           18
       28 |             5 | 2016-03-01 |           47
       29 |             5 | 2016-03-01 |           22
       30 |             5 | 2016-03-01 |            3
(30 rows)
```

# Step 2, cont.

## *PostgreSQL — date_trunc()*

- Truncate a datetime to a specific precision
- ex: date_trunc(‘month’, now())
- Use ‘Extract’ in MySQL and ‘strftime’ in SQLite.

```
SELECT date_trunc('month', order_date) AS order_month,
       SUM(order_amount) AS current_month_total
FROM takeout_orders
GROUP BY date_trunc('month', order_date)
```

# Step 3

**Key Idea:**
Build the trend

## *PostgreSQL — lag()*

- Returns the value at the specified row offset of the current row

```
SELECT order_month, current_month_total,
   lag(current_month_total, 1) OVER (order by order_month) AS last_month_total
FROM step_2
```

# Putting it all together

```
WITH restaurant_subset AS (
        SELECT rf1.restaurant_id
                FROM restaurant_features rf1
                LEFT JOIN restaurant_features rf2 ON rf1.restaurant_id = rf2.restaurant_id
                        AND rf2.feature = 'Takes Reservations'
                WHERE rf1.feature = 'Good for Groups' AND rf2.feature IS NULL
    ), order_trend AS (
        SELECT order_month, current_month_total,
                        lag(current_month_total, 1) OVER (ORDER BY order_month) AS last_month_total
                FROM (SELECT date_trunc('month', order_date) AS order_month,
                                SUM(order_amount) AS current_month_total
                        FROM takeout_orders t
                        JOIN restaurant_subset r ON r.restaurant_id = t.restaur
                        GROUP BY date_trunc('month', order_date)) AS monthly_orders
    )
SELECT order_month, current_month_total, last_month_total,
                (current_month_total - last_month_total) / last_month_total::DOUBLE PRECISION AS
perc_change
FROM order_trend
ORDER BY order_month;
```

Step 1

Step 2

Step 3

# Tada!

```
sql_interviews=>
sql_interviews=> WITH restaurant_subset as
sql_interviews-> (
sql_interviews(> SELECT rf1.restaurant_id
sql_interviews(> FROM restaurant_features rf1
sql_interviews(> LEFT JOIN restaurant_features rf2 ON rf1.restaurant_id = rf2.restaurant_id
sql_interviews(> AND rf2.feature = 'Takes Reservations'
sql_interviews(> WHERE rf1.feature = 'Good for Groups' AND rf2.feature IS NULL
sql_interviews(> ),
sql_interviews-> order_trend as
sql_interviews-> (
sql_interviews(> SELECT order_month, current_month_total,
sql_interviews(>     lag(current_month_total, 1) over (order by order_month) as last_month_total
sql_interviews(> FROM (SELECT date_trunc('month', order_date) AS order_month,
sql_interviews(>         SUM(order_amount) AS current_month_total
sql_interviews(>     FROM takeout_orders t
sql_interviews(>     JOIN restaurant_subset r ON r.restaurant_id = t.restaurant_id
sql_interviews(>     GROUP BY date_trunc('month', order_date)) as monthly_orders
sql_interviews(> )
sql_interviews-> SELECT order_month, current_month_total, last_month_total,
sql_interviews-> (current_month_total - last_month_total) / last_month_total::DOUBLE PRECISION as mom_perc_change
sql_interviews-> FROM order_trend
sql_interviews-> ORDER BY order_month;
      order_month       | current_month_total | last_month_total |  mom_perc_change
------------------------+---------------------+------------------+-------------------
 2016-01-01 00:00:00-05 |                  68 |                  |
 2016-02-01 00:00:00-05 |                 135 |               68 | 0.985294117647059
 2016-03-01 00:00:00-05 |                 232 |              135 | 0.718518518518519
(3 rows)
```