SQL questions

1) book: cracking the coding interview
2) www.programmerinterview.com/index.php/database-sql/introduction/
3) good one!! www.wagonhq.com/sql-tutorial/creating-a-histogram-sql
4) Leetcode: Database
5) SQL questions in your stock
6) glassdoor interview questions

Window functions

1) partition by
   a) avg() over(partition by xx) as xxx

2) sliding window functions - rows between <start> and <finish>
   a) default setting: rows between *unbounded preceding* and *current row*
   b) keywords: preceding/ following/ unbounded preceding/ unbounded following/ current row
   c) numbers works as well between 1 and 1

3) ranking function***
   a) adding row index per row:
      i) row_number() over(order by xxx) as xxx - just adding row index per record
         e.g. 1 2 3 4 5 6
      ii) row_number() over (partition by xxx order by xxx) as xxx - adding index within each group
         e.g. 1 2 3 1 2 3 4 1 2 1 2
   b) adding ranks and skipping the same "orderly":
      i) rank() over(order by xxx) as xxx - adding rank for the same order by and skip,
         e.g. 1 1 1 4 4 6
   c) adding ranks without skipping the same order:
      i) dense_rank() over(order by xxx) as xxx - adding rank for the same order without skipping,
         e.g. 1 1 1 2 2 3

Optimization

1) group by 取代 distinct
2) having 取代 join

Table 1

customers		
customer_id	account_id	city_name
1	1	New York
2	1	Paris
1	2	New York
3	2	Tel Aviv
4	3	Paris
5	4	Tel Aviv

Table 2

credit_cards		
account_id	credit_limit	Ccredit_card_ind
1	1,000	1
2	5,000	1
3	2,300	0
4	30,000	1

1

Write a SQL query that returns customer's details (city_name & customer_id) for customers with a credit limit higher than 4,000 USD

2

What is the key of Customers table?	
1	account_id
2	customer_id
3	After joining Customers and Credit Cards tables, key is account_id
4	None of the above

3

Write a SQL query that returns for each customer_id :

customer_id
# of accounts
# of credit cards
# of credit cards with credit limit higher than 4,000 USD

投的工作岗位是advertising data analyst。一共三道题，在coderpade上写，这是example table,第二列的单位是秒

Country|Durantion
US | 300
US | 600
JP | 1800
US | 300

Q1： 很简单，mean of durations
Answer：
select AVG(duration)
from #ps

Q2： top 5 countries with highest total duration
Answer：
select top 5 country, AVG(duration) as Average
from #ps
group by country
order by Average

Q2.1 如果有相同的，如何全部返回 （with ties）
Q2.2 使用Windows function 来实现，rank()over(order by)， 在这里卡了一下
Answer:

[SQL] 纯文本查看 复制代码

```sql
select top 5 ps1.country, Sum_D
from (select distinct country, sum(duration) over (partition by country) as Sum_D
        from #ps) ps1
order by ps1.Sum_D asc
```

Table Schema

publisher_info
- publisher_id
- video_id
- video_duration (in minutes)

consumption_info
- video_id
- user_id
- user_timespent

Questions:

1) How many minutes worth of video does an average publisher have?

2) How many publishers have at least one user who watched their videos?

1.

```
SELECT SUM(video_duration)/count(distinct publisher_id)
FROM publisher_info
```

2.

```
SELECT COUNT(DISTINCT publisher_id)
FROM publisher_info a
INNER JOIN consumption_info b
ON a.video_id = b.video_id
```

Table Name: **trips**

Column Name	Datatype
id	integer
client_id	integer (Foreign keyed to events.rider_id)
driver_id	integer
city_id	Integer (Foreign keyed to cities.city_id)
client_rating	integer
driver_rating	integer
request_at	Timestamp with timezone
predicted_eta	Integer
actual_eta	Integer
status	Enum('completed', 'cancelled_by_driver', 'cancelled_by_client')

Table Name: **events**

Column Name	Datatype
device_id	integer
rider_id	integer
city_id	integer
event_name	Enum('sign_up_success', 'attempted_sign_up', 'sign_up_failure')
_ts	Timestamp with timezone

1. For each of the cities A and B, calculate 90th percentile difference between actual and predicted ETA for all completed trips within the last 30 days

1.

```sql
SELECT c.city_name, Percentile_disc(0.9) within GROUP (ORDER BY t.actual_eta - t.predicted_eta) AS p90_diff
FROM trips t
LEFT JOIN cities c
  ON c.city_id = t.city_id
WHERE t.status = 'completed'
  AND (c.city_name = 'Qarth' OR c.city_name = 'Meereen')
  AND request_at > now() - interval '30 days'
GROUP BY  c.city_name;
```

Employee

	employee_id	first_name	last_name	gender	position	department_id	salary
1	2002	Super	Man	M	Tester	1	75000
2	2003	Jessica	Liyers	F	Architect	1	60000
3	2004	Bonnie	Adams	F	Project Manager	1	80000
4	2005	James	Madison	M	Software Developer	1	55000
5	2006	Michael	Greenback	M	Sales Assistant	2	85000
6	2007	Leslie	Peters	F	Sales Engineer	2	76000
7	2008	Max	Powers	M	Sales Representative	2	59000
8	2009	Stacy	Jacobs	F	Sales Manager	2	730000
9	2010	John	Henery	M	Sales Director	2	90000

Department

	department_id	department_name
1	1	IT
2	2	Sales

- *Sample query questions*
  - return **employee record with highest salary**
  - Return the **highest salary** in **employee table**
  - Return the **2nd highest salary** from **employee table**
  - Select **range of employees** based on **id**
  - Return an **employee** with the **highest salary** and the **employee's department name**
  - Return **highest salary**, **employee_name**, **department_name** for **each department**

https://www.youtube.com/watch?v=uAWWhEA57bE

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/

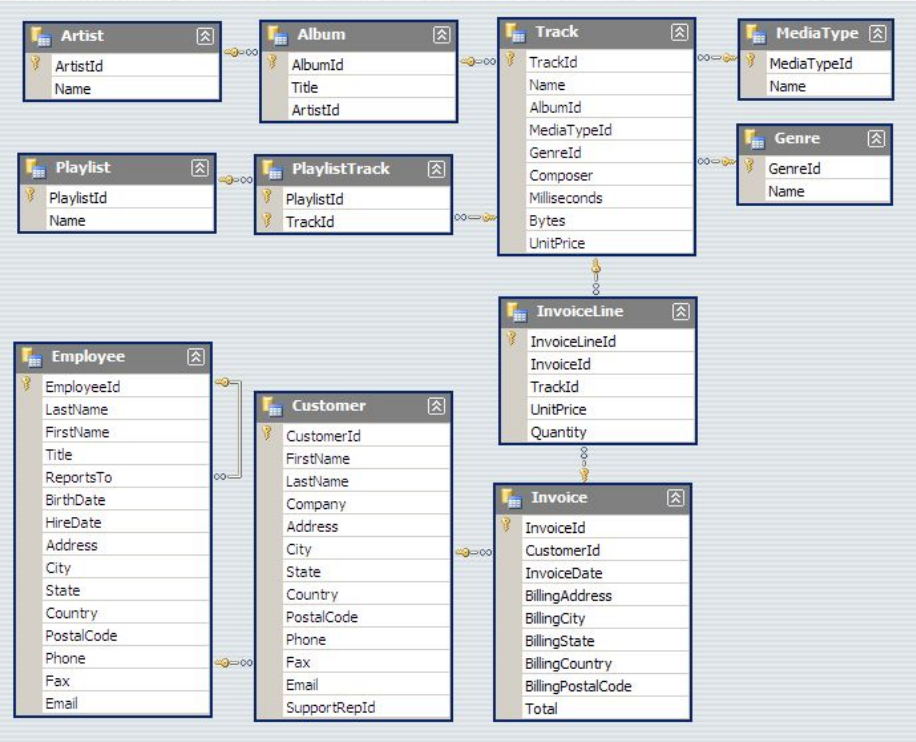
Chinook Database

- 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
```

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

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
```

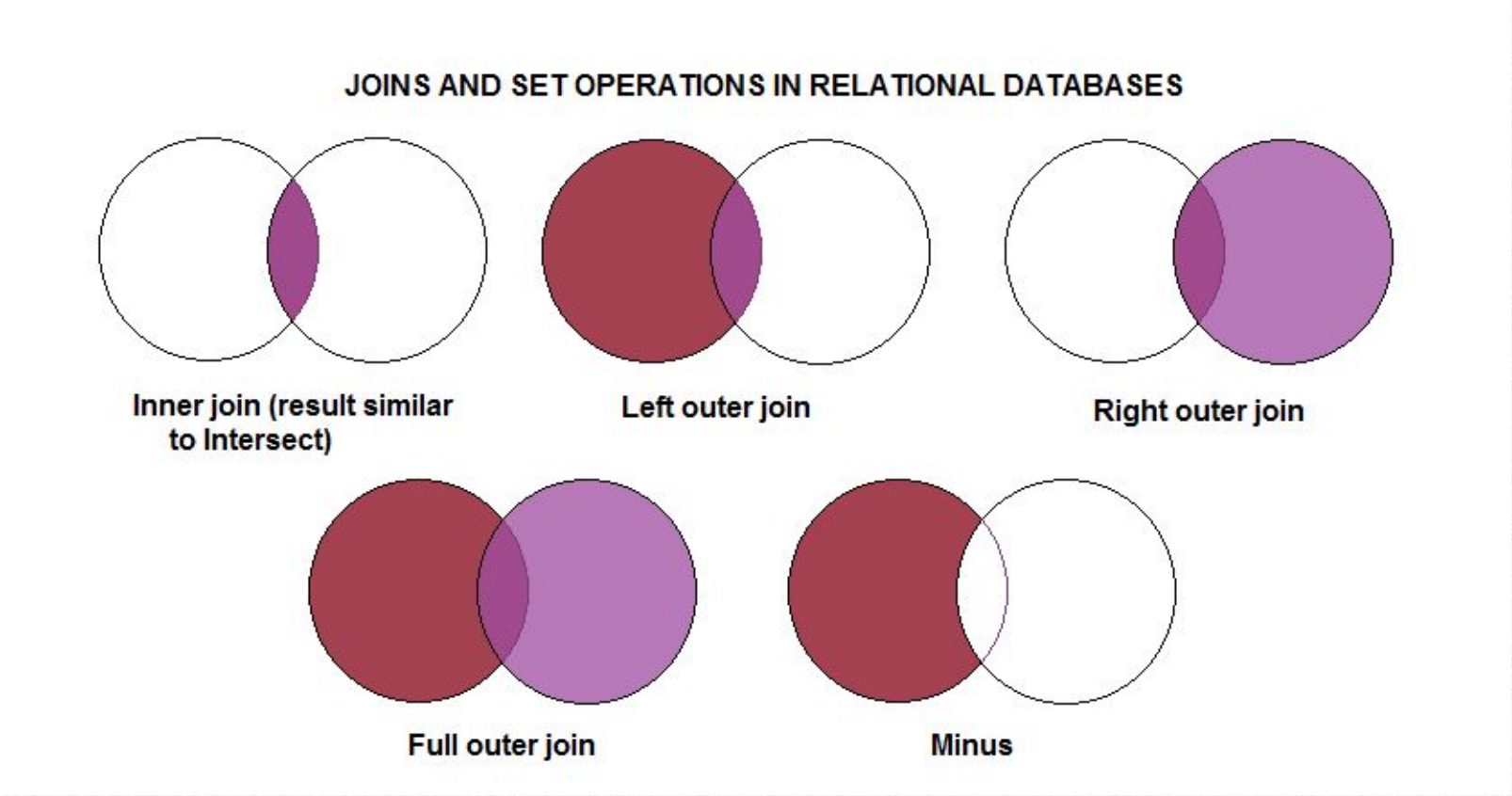
Load Query... Save Query...

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

Examples

Joins!

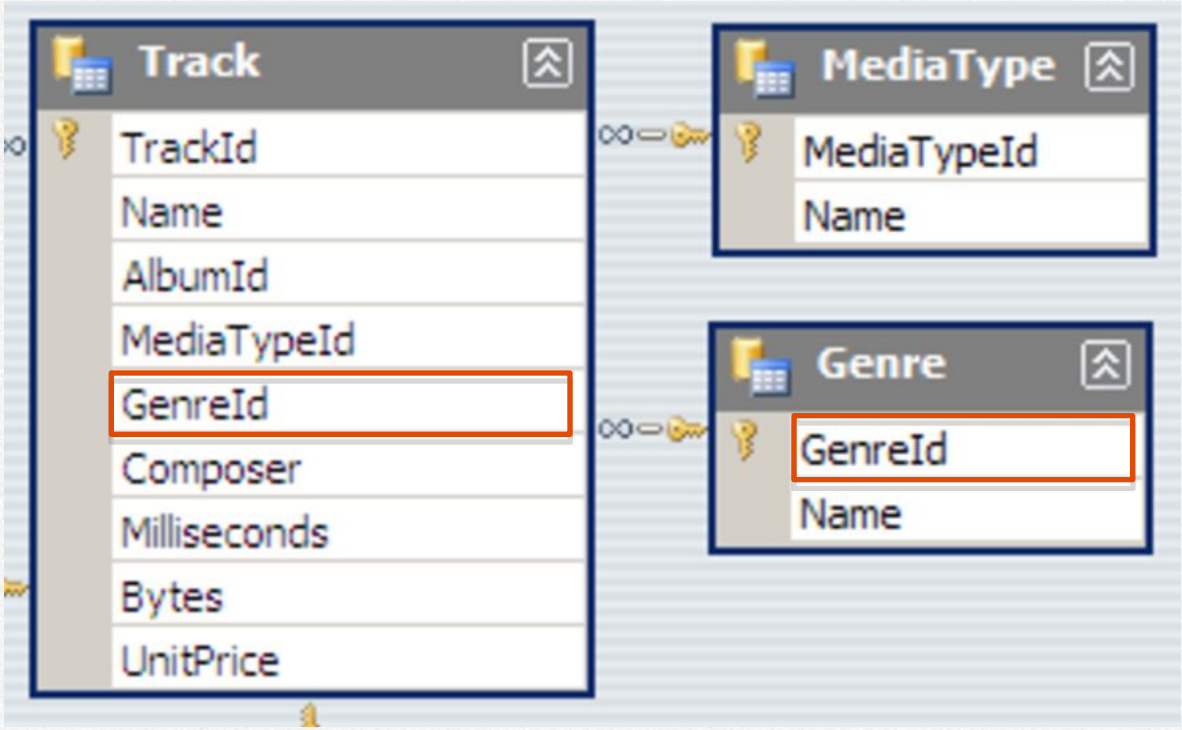
- Joins are used to combine tables
- This is where primary keys become important– they are generally used to link two tables



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
```

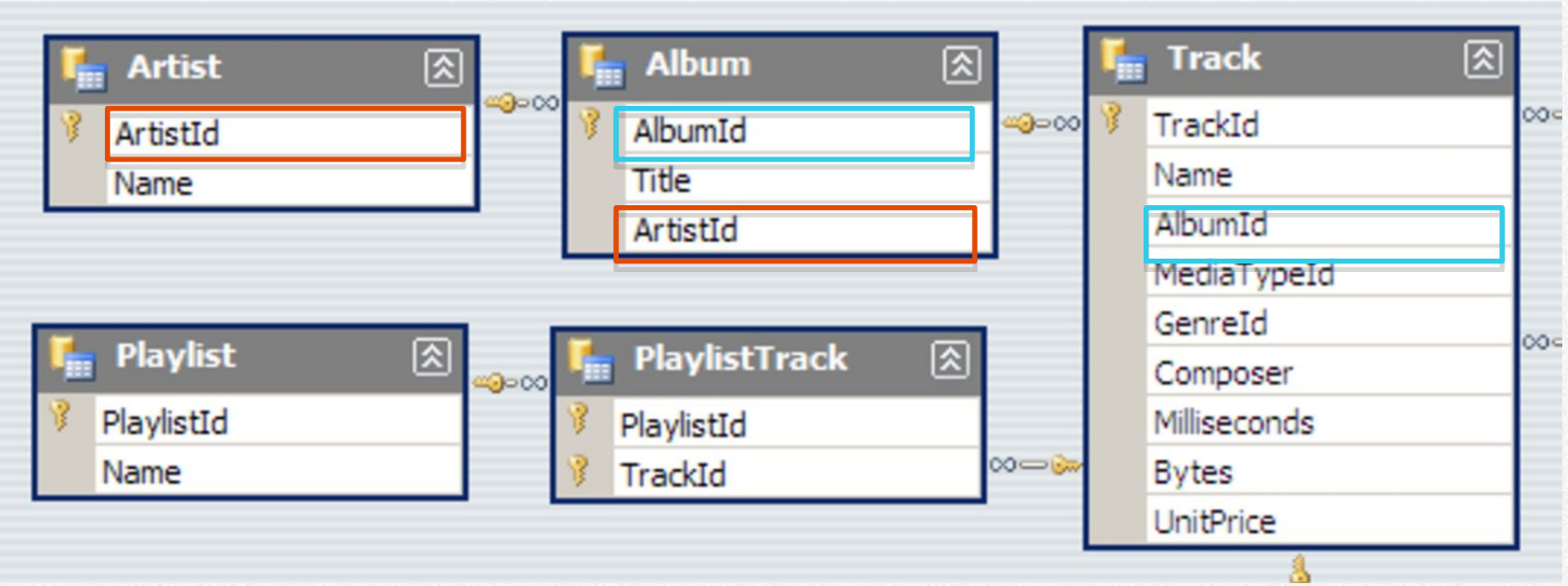
Load Query... Save Query...

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.restaurant_id
                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)
```