1. Write a query to find the names of the customers who were born after 1990-01-01, and the family name starts with 'G'

### Hint: See postgresql date operators and string functions
### Order: by name
### Output columns: name

This question shows how to use the basic date and string operators standardized by SQL and functions provided by the database implementation. `birthdate >= ` is a date comparison, while `like '% G%'` is a string comparison. `DATE 1990-01-01'` cast the string ‘1990-01-01’ to a date, so that it can be compared to the birthdate attribute (which was declared to be of type date).

```sql
SELECT name
FROM customers
WHERE birthdate >= DATE '1990-01-01' AND
      name LIKE '% G%'
ORDER BY name;
```

Date comparison
String comparison

2. Write a query to find unique customers who flew on the dates within one week before their birthday.

### Hint: See postgresql date functions and distinct operator
### Order: by name
### Output columns: all columns from customers

First, this question helps you get familiar with basic natural joins, which forms the building block for the rest of the assignment. More importantly, this question shows in detail how to work with date/timestamp data types, which are very common in real world queries. The answer shows how to a) extract elements from a date, b) construct a date from elements, and c) compare dates via intervals. We present two solutions below, the second one using fewer date specific functionality. (the :: syntax in the second solution is an alternative way to do casting in Postgres --- the birthdate is first cast to a char(10) in order to call the substring and concatenation functions on it, and then cast back to a date).

```sql
SELECT DISTINCT(customerid), name, birthdate, frequentflieron
FROM flewon f NATURAL JOIN customers
WHERE to_date(concat(2016, ' ', extract(month from birthdate), ' ',
              extract(day from birthdate)), 'YYYY MM DD')
      BETWEEN f.flightdate + interval '1 day' AND
              f.flightdate + interval '1 week'
ORDER BY name;
```

Construct the birthday in 2016 from a string

Offset date with interval

```sql
SELECT DISTINCT(customerid), name, birthdate, frequentflieron
FROM customers NATURAL JOIN flewon
WHERE ('2016-'||substring(birthdate::char(10) from 6 for 5))::date
      - flightdate
      BETWEEN 1 AND 7
ORDER BY name;
```

Shorter syntax to convert between string and date

Compare without interval

3. Write a query to find number of inbound flights by each airlines to any airport

### Output: (airport_city, airline_id, inbound_flights)
### Order: first by airport_city in the increasing order, then inbound_flights in decreasing order, and then airline_id in the increasing order.
### Note: You must generate the airport city names instead of airport codes.

This query shows a) a join with custom conditions, b) the concept of group by and aggregation, both are basic operators of SQL and building blocks of complex queries. It also shows ordering query results with multiple-criteria.

```
SELECT city AS airport_city, airlineid, count(*) AS inbound_flights
FROM flights, airports
WHERE flights.dest = airports.airportid
GROUP BY city, airlineid
ORDER BY city ASC, inbound_flights DESC, airlineid ASC;
```

Aggregation after group by

Join condition in where clause

Order by multiple criteria

4. Find the name of the customer who flew the most times with his/her frequent flier airline. For example, if customer X flew Delta (which is listed as X's frequent flier airline in the customers table) 100 times, and no other customer flew their frequent flyer airline more than 99 times, the only thing returned for this query is X's name.

### Hint: use `with clause` and nested queries
### Output: only the name of the customer. If multiple answers, return them all.
### Order: order by name.

From the basic concepts in the first three queries, this query starts to increase the complexity, and shows how to use the nested queries and temporary queries.

In the solution below, in the first part of the query, a temporary relation is created, corresponding to the number of times that a customer flies with his/her frequent flyer airline. The second part shows a solution for getting the max result (also called ‘top-1’ result) in a table.

Note, top-1, top-k, and ranking queries are important queries in practice. Simply taking the first result from the sorted list (e.g. by using a special SQL command such as ‘LIMIT 1’), to select the first row of the resulting table would not always return the desired result if there is more than one customer who took the maximum number of flights.

```
WITH customer_flewontimes_by_airline AS (
    SELECT customerid,
           substring(flightid from 1 for 2) AS airline_id,
           count(*) AS flewon_times
    FROM flewon
    GROUP BY customerid, airline_id
```

Find the number of *flights* taken by *customers* on any *airlines*

```
),
customer_flewontimes_ffairlines AS (
    SELECT b.name, a.flewon_times
    FROM customer_flewontimes_by_airline a, customers b
    WHERE a.customerid = b.customerid AND
          a.airline_id = b.frequentflieron
)
SELECT name FROM customer_flewontimes_ffairlines
WHERE flewon_times =
      (SELECT max(flewon_times) FROM customer_flewontimes_ffairlines)
ORDER BY name;
```

Join with *customers* table to get the customer name, and the frequent flyer airline, which are used to filter the previous results.

Use a nested query to find the maximum flewon times, and return the top customer(s)

5. Find all 1-stop flights from JFK to LAX having layover duration greater than or equal to 1 hour.

### Output: (1st flight id, 2nd flight id, connection airport id, layover duration).
### Order: by the duration hours.

This query shows self-joins. Self-join is an operation that joins the same table multiple times, which often encountered when the dataset can be thought of as a graph (in this case, the cities can be thought of as vertices in the graph, and edges are the flights between them).

In this query, we are interested in finding all 1-hop neighbors of the node JFK, which has the node LAX as their 1-hop neighbor. A self-join on the edge table renders the 1-hop neighborhood. Therefore, we must match the destination city of a flight originating from JFK with the source city of a flight that has LAX as its destination along with the other filter conditions for this query. Multiple self-joins let you analyze ancestors (parents) and descendants (children) in the graph.

Exercise: Now try to increase the self-join multiple times, how many rows are derived at each run? Is SQL and relational database systems good at graph analysis?

```
SELECT a.flightid AS flight1,
       b.flightid AS flight2,
       b.source AS connection_airport,
       (b.local_departing_time - a.local_arrival_time) AS duration
FROM flights a, flights b
WHERE a.dest = b.source AND
     (b.local_departing_time
          - a.local_arrival_time) >= interval '1 hour' AND
      a.source='JFK' AND b.dest='LAX'
ORDER BY duration;
```

Perform a self-join on the flights table by matching the destination of table a with the source of table b (Similarly, you can join twice to get 2-stops, n times to get n-stops)

6. Assuming each flight have 120 seats, from flewon, find all flights with passenger load factor (PLF) less than or equal to 1% on Aug 1st 2016. Note, PLF is defined as number of customers on board divided by total number of available seats.

### Output: (flightid, PLF).

### Order: first by PLF descreasing order, then flightid
### Note: a) Each flight flew daily during Aug1 and Aug9, 2016. There may be empty flights which are not in the flewon table (i.e. PLF=0). Please include those.
###   b) PLF should be rounded to 2 decimal places, e.g., 10% should be 0.10.
### Hint: SQL set operators union/except/intersect may be useful.

This question is based on a real world business intelligence (BI) query. Each business domain has their well-established schema, e.g., airlines, insurance, banking, etc. The analysts often look at the past data using SQL, make quantitative decisions based on some operation models, and report the analysis results using SQL. In this case, PLF is an important measurement in the transport services, and often used as an input to further analysis. To get the answer right (PLF <= 1%), we need to be aware of the empty data which were not kept in the database. This query shows how to think in SQL and come up with the solution to handle non-existing records. The answer key shows how the set operations (union/intersection) are used in practice.

```
WITH flewon_plf_0801 AS (
    SELECT flightid, round(count(*)/120.0, 2) AS plf
    FROM flewon
    WHERE flightdate = DATE '2016-08-01'
    GROUP BY flightid
), empty_flights0801 AS (
    SELECT flightid, round(0.0, 2) AS plf
    FROM flights
    WHERE flightid NOT IN
        (SELECT DISTINCT(flightid)
         FROM flewon WHERE flightdate = DATE '2016-08-01')
)
SELECT flightid, plf FROM (
    SELECT * FROM empty_flights0801
    UNION
    SELECT * FROM flewon_plf_0801 WHERE plf <= 0.01
) combined_plf_0801
ORDER BY plf DESC, flightid;
```

Find all flights and their PLF based on the records stored in flewon table

round for float precision

From the *flights* table, find the ones which are not present in the *flewon* table

NOTE: (NOT) IN is set membership operator.
- *Exercise*: is DISTINCT necessary here?

Finally combine the two results by an union and then apply the required filters

7. Write a query to find the customers who used their frequent flier airline the least when compared to all the airlines that this customer as flown on. For example, if customer X has Delta as X's frequent flyer airline in the customer table, but flew on Delta only 1 time, but every other airline at least 1 time, then X's id and name would be returned as part of this query.
### Output: (customerid, customer_name)
### Order: by customerid
### Note: a customer may have never flown on their frequent flier airlines.

This question emphasizes skills to handle empty records and top-1 queries. Compared with question 4, this question requires aggregating flewon records at an individual level, and getting the top-1 (least) results at individual level as well. To filter the individual-level top-1, the per-customer least flewon times need to be calculated and used in the join. Instead of using a temporary table, the first solution below shows how this can be done in a nested query with conditions containing variables (t1) from the hosted query. This is also called “correlated subqueries” --- see Section 3.8.3 from your textbook.

```
WITH customer_onboard_times_init AS (
   SELECT customerid, airlineid, 0 AS num_flights
   FROM customers, airlines
), customer_actual_onboard_times_by_airline AS (
    SELECT customerid, airlineid, count(*) AS num_flights
    FROM flewon NATURAL JOIN flights
    GROUP BY customerid, airlineid
), customer_onboard_times_combined AS (
    SELECT customerid, airlineid, sum(num_flights) AS total_flights
    FROM (SELECT * FROM customer_onboard_times_init
          UNION SELECT * FROM  customer_actual_onboard_times_by_airline
    ) onboard_combined
    GROUP BY customerid, airlineid
)
SELECT customerid, name
FROM customer_onboard_times_combined t1 NATURAL JOIN customers
WHERE airlineid = frequentflieron AND
      total_flights = (SELECT min(total_flights)
                       FROM customer_onboard_times_combined t2
                       WHERE t1.customerid = t2.customerid)
ORDER BY customerid;
```

Similar to Q6, to handle empty records in flewon, we initialize each customer’s on board time on each airline to 0.

Here we use Union to combine customers’ onboard times, and do a sum aggregation.

For top-1 at individual level, we show a way to use nested queries with where condition referring to the host query (Note *t1* is not mentioned in the nested query, but the host one)

If you prefer not using correlated subqueries, you can create an additional temporary table instead, as shown below . However, correlated subqueries are a useful technique that you should get familiar with. In Q10, we will see another example.

```
# reuse previous temporary tables
, customer_min_onboard_times(customerid, total_flights) AS (
    SELECT customerid, min(total_flights)
    FROM customer_onboard_times_combined
    GROUP BY customerid
)
SELECT customerid, name
FROM customer_onboard_times_combined NATURAL JOIN
     customer_min_onboard_times NATURAL JOIN
     customers
WHERE airlineid = frequentflieron
ORDER BY customerid;
```

Instead of the previous nested query, we can prepare the table first by scanning the onboard combined table once.

Then do a similar join to get the same results.

8. Write a query to find the flights which are empty on three consecutive days, but not empty on the other days, return the flight, and the start and end dates of those three days.

### Hint: postgres window functions may be useful

### Output: flightid, start_date, end_date

### Order: by start_date, then flightid

This question attempts to get you to think in SQL and solve time-series flavor arrays. Each flight's flewon record forms a 0/1 boolean array. Typically SQL is NOT good at reasoning about such data types. This means, if you try to solve this problem as if you are writing an array scanning algorithm, you probably will not be successful. On the other hand, SQL builds on top of relational algebra which is based on sets. Thinking about this problem in terms of sets and set membership is likely going to be more helpful. The solution below first constructs all empty flights, and then uses this information to filter all flightids to contain only those with exactly three empty entries. Then the check to see if these three empty days were consecutive can be done via examining the size of the gap of those empty days.

```
WITH empty_flights AS (
    SELECT flightid, flightdate
    FROM flights,
         (SELECT DISTINCT flightdate FROM flewon) all_dates
    EXCEPT
    SELECT flightid, flightdate
    FROM flewon
    ORDER BY flightid, flightdate
), flights_three_times_empty AS (
    SELECT * FROM empty_flights
    WHERE flightid IN (
        SELECT flightid FROM empty_flights
        GROUP BY flightid HAVING count(*) = 3
    )
)
SELECT flightid,
       min(flightdate) AS start_date,
       max(flightdate) AS end_date
FROM flights_three_times_empty
GROUP BY flightid
HAVING max(flightdate) - min(flightdate) = 2
ORDER BY flightid;
```

Find the flightid and date of all empty *flights*.
- This join has no join condition. Therefore, it acts as a Cartesian product and returns all possible combinations.
- Use EXCEPT to do set subtraction. I.e., subtract all flights that weren't empty from all possible flights. We are left with just the empty ones.

Filter out the *flights* which have three empty cases exactly.

Condition to check if the three days are consecutive.

9. Write a query to find the city name(s) which have the strongest connection with OAK. We define it as the total number of customers who took a flight that departures the city to OAK, or arrives the city from OAK.

### Output columns: city name

### Order by: city name

### Note: a) You can assume there is only one airport in a city.

### b) If there are ties, return all tied cities

Similar to question 5, this query shows a graph analysis query by aggregating the incoming and outgoing edges w.r.t a given node (OAK), and return the top-1 neighbors. As the incoming and outgoing edges have flipped orders of (dest, source) pair, you have to write two queries and assemble the results. The "from_oak" and "to_oak" relations below compute the respective counts. To sum the strength, the answer key shows the way to use union.

Exercise: In the first solution below, if the `direction (0,1)` column is not added, the union will result in less total strength, while in the second solution below, `union all` is used instead of 'union' and does not need the direction column. Can you understand what functionality the direction column is adding?

```
WITH from_oak AS (
    SELECT dest AS airportid, count(*) AS strength, 0 AS direction
    FROM flights NATURAL JOIN flewon
    WHERE source = 'OAK'
    GROUP BY dest
), to_oak AS (
    SELECT source AS airportid, count(*) AS strength, 1 AS direction
    FROM flights NATURAL JOIN flewon
    WHERE dest = 'OAK'
    GROUP BY source
), oak_connections AS (
    SELECT airportid, sum(strength) AS strength
    FROM (SELECT * FROM from_oak UNION SELECT * FROM to_oak) oak_edges
    GROUP BY airportid
)
SELECT city FROM oak_connections NATURAL JOIN airports
WHERE strength = (SELECT max(strength) FROM oak_connections)
ORDER BY city;
```

From OAK (source airport), list the destinations and their strength. Add a direction column and fill it with 0's.

To OAK (destination airport), list the sources and their strength. This time, use a '1' direction column.

Compute the sum of the strength using Union.

Report the city (associated with the airportid) with the maximum strength

The query above can be simplified a bit using UNION ALL instead of UNION. UNION ALL retains duplicate rows (multiset or bag semantics) while performing an union whereas UNION discards them (set semantics). The first three temporary tables can be written as:

```
WITH oak_connections AS (
    SELECT airportid, sum(strength) AS strength
    FROM (
            SELECT dest AS airportid, count(*) AS strength
            FROM flights NATURAL JOIN flewon
            WHERE source = 'OAK' GROUP BY dest
        UNION ALL
            SELECT source AS airportid, count(*) AS strength
            FROM flights NATURAL JOIN flewon
            WHERE dest = 'OAK' GROUP BY source
    ) oak_connections
    GROUP BY airportid
)
```

UNION ALL instead of UNION

10. Write a query that outputs the top 20 ranking of the most busy flights. We rank the flights by their average onboard customers, so the flight with the most average number of customers gets rank 1, and so on.

### Output: (flightid, flight_rank)

### Order: by the rank, then flightid

### Note: a) If two flights tie, then they should both get the same rank, and the next rank should be skipped. For example, if the top two flights have the same average number of customers, then there should be no rank 2, e.g., 1, 1, 3 ...
###   b) There may be empty flights.

This query shows how to come up with ranking using SQL. Newer versions of the SQL standard include partition/window functions which make calculating the rank easier. However, the solution below shows the preferable method of expressing this query without using window functions. It also shows how to avoid relying on the flight date assumptions (Aug.1~Aug.9) but still gets the average number correct.

```
WITH flight_customers_per_day AS (
    SELECT flightid, flightdate, count(customerid) AS onboard_cnt
    FROM flewon
    GROUP BY flightid, flightdate
), flight_avg_customers(flightid, avg_customer) AS (
    SELECT flightid,
           sum(onboard_cnt) / (SELECT max(flightdate)
                                        - min(flightdate) + 1.0 FROM
flewon)
    FROM flight_customers_per_day
    GROUP BY flightid
), ranked_flights AS (
    SELECT
       flightid,
       avg_customer,
       (SELECT count(*)
        FROM flight_avg_customers t2
        WHERE t2.avg_customer > t1.avg_customer
       ) + 1 AS flight_rank
    FROM flight_avg_customers t1
)
SELECT flightid, flight_rank FROM ranked_flights
WHERE flight_rank <= 20
ORDER BY flight_rank, flightid;
```

Prepare the basic data table for the ranking.

Get the average onboard customers.

Without assuming date ranges, calculate the total date.

The rank of a member x is 1 plus the number of members that are greater than x. This first solution shows how do calculate this using correlated subqueries. The solution below does the same thing without correlated subqueries.

Apply condition on the rank.

Below we show an alternative solution without using correlated subqueries.

Exercise: The following query has to use two parts and union to get the current rank. First, why in the previous answer using nested queries we don't have to do so? Second, one may think that if we had used ‘<=’ instead of ‘<’ in `flight_pairwise_order’, the second table (‘busiest_flights’) would not have been necessary. Is that correct? If not, try to use use ‘<=’ and get the correct results.

```
# reuse previous temporary tables
, flight_pairwise_order (fid1, avg1, fid2, avg2) AS (
    SELECT *
    FROM flight_avg_customers AS f1,
         flight_avg_customers AS f2
    WHERE f1.avg_customer < f2.avg_customer
```

We first prepare a pair-wise comparison table for the *flights*. This is the first time we’ve seen a join using a non-equality join predicate. We keep the pairs whose first flight (*fa*) is less busier than the second (*fb*). From this table, it is easy to see the count

<table>
<tr>
<td>

```sql
), busiest_flights (fid1, avg1) AS (
    SELECT *
    FROM flight_avg_customers
    WHERE avg_customer =
          (SELECT max(avg_customer) FROM flight_avg_customers)
), ranked_flights(flightid, flight_rank) AS (
    SELECT fid1, count(*) + 1 AS frank
    FROM flight_pairwise_order
    GROUP BY fid1
    UNION
    SELECT fid1, 1 AS frank
    FROM busiest_flights
)
SELECT *
FROM ranked_flights
WHERE flight_rank <= 20
ORDER BY flight_rank, flightid;
```

</td>
<td>

of such pairs for *fa* is its rank - 1 (according to previous definition). However using ‘<’ filters out the busiest flights. So we have to prepare the second table to calculate the busiest ones.

Use union, we combine the final ranked flights.

</td>
</tr>
</table>