

Members: Chris Whamond, Aditya Karan, Shawn Lee, Jackie Chan
team31
Stage 3 Submission

Database Implementation

```
flight_price_info=> \dt
                    List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | LegInfo         | table | postgres
 public | PricingInfo     | table | postgres
 public | ProfileInfo     | table | postgres
 public | RoundTripInfo   | table | postgres
 public | humanuser       | table | postgres
(5 rows)

flight_price_info=> █
```

We implemented five tables: note that `humanuser` is not trivial (it contains some login details, but also critical attribute information used in matching particular price results with specific traits for comparison in a key project component).

DDL Commands

```
CREATE TABLE IF NOT EXISTS LegInfo (
    leg_id SERIAL,
    date DATE,
    start_location TEXT,
    end_location TEXT,
    carrier TEXT,
    time TEXT,
    leg_length TEXT,
    stop_info TEXT,
    PRIMARY KEY (leg_id)
);
```

```
CREATE TABLE IF NOT EXISTS PricingInfo (
    price INT,
    seller TEXT,
    search_time TIMESTAMP,
    round_trip_id INT,
    profile_id INT,
    FOREIGN KEY (profile_id) REFERENCES ProfileInfo(profile_id),
```

```

        FOREIGN KEY (round_trip_id) REFERENCES RoundTripInfo(route_id)
    );

CREATE TABLE IF NOT EXISTS RoundTripInfo (
    route_id SERIAL,
    name TEXT,
    leg_id1 INT,
    leg_id2 INT,
    start_location TEXT,
    end_location TEXT,
    PRIMARY KEY (route_id),
    FOREIGN KEY (leg_id1) REFERENCES LegInfo(leg_id),
    FOREIGN KEY (leg_id2) REFERENCES LegInfo(leg_id)
);

CREATE TABLE IF NOT EXISTS ProfileInfo (
    profile_id SERIAL NOT NULL,
    name TEXT,
    PRIMARY KEY (profile_id)
);

CREATE TABLE IF NOT EXISTS HumanUser (
    id INT NOT NULL,
    name VARCHAR(100),
    gender VARCHAR(10),
    age INT,
    zip VARCHAR(10),
    website_visited VARCHAR(100),
    password VARCHAR(100),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES ProfileInfo(profile_id) ON DELETE CASCADE
);

```

1,000 Rows Per Table Requirement

Per instructions, three of our four tables have at least 1000 rows:

```
flight_price_info=> SELECT COUNT(*) FROM "LegInfo";
count
-----
17065
(1 row)
```

Proof for LegInfo table.

```
flight_price_info=> SELECT COUNT(*) FROM "PricingInfo";
count
-----
237178
(1 row)
```

Proof for PricingInfo table.

```
flight_price_info=> SELECT COUNT(*) FROM "RoundTripInfo";
count
-----
16505
(1 row)
```

Proof for RoundTripInfo table.

Advanced Queries

AQ1: Average Round Trip Length for Each Departure/Starting Location

```
1 SELECT
2     start_location,
3     AVG(leg_1_length + leg_2_length) AS avg_roundtrip_length
4 FROM (
5     SELECT
6         start_location,
7         (
8             SELECT
9                 CAST(leg_length AS INTERVAL)
10            FROM
11                "LegInfo"
12            WHERE
13                leg_id = leg_id1
14        ) AS leg_1_length,
15        (
16            SELECT
17                CAST(leg_length AS INTERVAL)
18            FROM
19                "LegInfo"
20            WHERE
21                leg_id = leg_id2
22        ) AS leg_2_length
23    FROM
24        "RoundTripInfo"
25 ) as t
26 GROUP BY
27     start_location
28 LIMIT
29     15;
```

This query joins the roundtrip and its two travel leg information. And then, it will calculate the average leg_length(travel duration) of round trips for each start_location.

The query above satisfies the following advanced query requirements: joins multiple relations, aggregates via GROUP BY, and contains subqueries.

start_location	avg_roundtrip_length
9MY	12:28:51.428571
AFW	47:27:13.333333
ATL	05:57:05.806452
AUS	06:28:09.375
AZA	19:14:00
BDF	10:03:34.064516
BNA	07:56:43.231445
BUF	12:15:27.931034
BUR	23:18:20.917431
BWI	12:32:10.953347
BZN	16:37:40.754717
CHI	20:46:32.307692
CLT	03:00:07.272727
CMH	09:06:42
DEN	08:52:17.053942

(15 rows)

First 15 rows from the first advanced query.

AQ2: Calculate the Average Round Trip Price Each Sock Puppet Account Pays

```

1 SELECT
2     name,
3     ROUND(AVG(price / 100), 2) AS avg_price,
4     COUNT(DISTINCT round_trip_id) AS num_round_trips
5 FROM
6     "ProfileInfo"
7 NATURAL JOIN
8     "PricingInfo"
9 GROUP BY
10    profile_id
11 ORDER BY
12    avg_price
13 LIMIT
14    15;

```

This query joins the ProfileInfo and PricingInfo relation by joining on the shared column they have: profile_id. Once it's joined, group by using the profile_id to average the prices on all the round trips and count the number of unique round trips each profile queried. Finally, sort in descending order by the average price.

name	avg_price	num_round_trips
senior_patagon	547.09	9438
youth_patagon	563.53	9484
senior	574.03	9543
male_patagon	576.07	9642
male	577.00	9493
youth	584.47	9522
control	585.52	9637
female_patagon	589.11	9660
female	591.47	9529
(9 rows)		

The output of the second advanced query. Only 9 rows because there are only 9 sock puppet accounts.

Note: there are only 9 sock puppet accounts that have queried for round trips so the limit clause is pointless.

The query above satisfies the following requirements as specified on the rubric: joins multiple relations and aggregates via GROUP BY.

Indexing Analysis for AQ1

```

QUERY PLAN
-----
Limit (cost=1443.28..65770.28 rows=15 width=20) (actual time=10.824..23.209 rows=15 loops=1)
  -> GroupAggregate (cost=1443.28..275922.23 rows=64 width=20) (actual time=10.822..23.205 rows=15 loops=1)
    Group Key: "RoundTripInfo".start_location
    -> Sort: (cost=1443.28..1484.54 rows=16505 width=12) (actual time=10.683..10.930 rows=2935 loops=1)
      Sort Key: "RoundTripInfo".start_location
      Sort Method: quicksort Memory: 1542kB
      -> Seq Scan on "RoundTripInfo" (cost=0.00..287.05 rows=16505 width=12) (actual time=0.019..2.857 rows=16505 loops=1)
        SubPlan 1
          -> Index Scan using "LegInfo_pkey" on "LegInfo" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.002 rows=1 loops=1)
            Index Cond: (leg_id = "RoundTripInfo".leg_id1)
          SubPlan 2
            -> Index Scan using "LegInfo_pkey" on "LegInfo" "LegInfo_1" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.002 rows=1 loops=1)
              Index Cond: (leg_id = "RoundTripInfo".leg_id2)
        Planning Time: 0.822 ms
        Execution Time: 23.426 ms
        (15 rows)

```

Create an index start_location

```
QUERY PLAN
-----
Limit (cost=0.29..64509.96 rows=15 width=20) (actual time=0.144..12.354 rows=15 loops=1)
  -> GroupAggregate (cost=0.23..275241.55 rows=64 width=20) (actual time=0.143..12.351 rows=15 loops=1)
    Group Key: "RoundTripInfo".start_location
    -> Index Scan using idx_start_location on "RoundTripInfo" (cost=0.25..803.86 rows=16505 width=12) (actual time=0.033..0.784 rows=2935 loops=1)
      SubPlan 1
        -> Index Scan using "LegInfo_pkey" on "LegInfo" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.001 rows=1 loops=2934)
          Index Cond: (leg_id = "RoundTripInfo".leg_id1)
        SubPlan 2
          -> Index Scan using "LegInfo_pkey" on "LegInfo" "LegInfo_1" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.001 rows=1 loop=1)
            Index Cond: (leg_id = "RoundTripInfo".leg_id2)
      Planning Time: 0.360 ms
      Execution Time: 12.430 ms
(12 rows)
```

After creating an index on the start_location, the query plan was more simplified and the performance was enhanced. In the query plan, sorting when grouping rows was removed because index already explains where each row is.

Create an index on start_location, leg_id1

```
QUERY PLAN
-----
Limit (cost=0.29..64509.96 rows=15 width=20) (actual time=0.103..12.134 rows=15 loops=1)
  -> GroupAggregate (cost=0.29..275241.55 rows=64 width=20) (actual time=0.103..12.134 rows=15 loops=1)
    Group Key: "RoundTripInfo".start_location
    -> Index Scan using idx_start_location on "RoundTripInfo" (cost=0.29..803.86 rows=16505 width=12) (actual time=0.012..0.679 rows=2935 loops=1)
      SubPlan 1
        -> Index Scan using "LegInfo_pkey" on "LegInfo" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.001 rows=1 loops=2934)
          Index Cond: (leg_id = "RoundTripInfo".leg_id1)
        SubPlan 2
          -> Index Scan using "LegInfo_pkey" on "LegInfo" "LegInfo_1" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.001 rows=1 loop=1)
            Index Cond: (leg_id = "RoundTripInfo".leg_id2)
      Planning Time: 0.277 ms
      Execution Time: 12.191 ms
(12 rows)
```

After adding an additional index on leg_id1, the query plan stayed the same, and the performance showed a negligible change.

Create an index on start_location, leg_id1, leg_id2

```
QUERY PLAN
-----
Limit (cost=0.29..64509.96 rows=15 width=20) (actual time=0.225..12.833 rows=15 loops=1)
  -> GroupAggregate (cost=0.29..275241.55 rows=64 width=20) (actual time=0.224..12.829 rows=15 loops=1)
    Group Key: "RoundTripInfo".start_location
    -> Index Scan using idx_start_location on "RoundTripInfo" (cost=0.29..803.86 rows=16505 width=12) (actual time=0.123..0.921 rows=2935 loops=1)
      SubPlan 1
        -> Index Scan using "LegInfo_pkey" on "LegInfo" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.002 rows=1 loops=2934)
          Index Cond: (leg_id = "RoundTripInfo".leg_id1)
        SubPlan 2
          -> Index Scan using "LegInfo_pkey" on "LegInfo" "LegInfo_1" (cost=0.29..8.31 rows=1 width=16) (actual time=0.001..0.001 rows=1 loop=1)
            Index Cond: (leg_id = "RoundTripInfo".leg_id2)
      Planning Time: 0.193 ms
      Execution Time: 12.889 ms
(12 rows)
```

After adding an additional index on leg_id2, the query plan stayed the same, and the performance showed a negligible change.

The final design for this query should be adding a single index on start_location field in table RoundTrip. Sorting by a grouping key is necessary to group a table if that key is indexed. However, if we index that key in the table, it is possible to group rows by that key without sorting. This makes a huge improvement on the performance because otherwise the sorting would have been done with the entire rows. However, indexing leg_id1 or leg_id2 had no impact in the optimization. Adding an index might be beneficial in the query performance, but it also degrades the performance when inserting/updating rows. Therefore, the final design should be only indexing start_location field.

Indexing Analysis for AQ2

```

QUERY PLAN
-----
Limit (cost=35481.25..35481.29 rows=15 width=76) (actual time=218.588..218.592 rows=9 loops=1)
  -> Sort (cost=35481.25..35484.43 rows=1270 width=76) (actual time=218.586..218.589 rows=9 loops=1)
        Sort Key: (round(avg(("PricingInfo".price / 100)), 2))
        Sort Method: quicksort Memory: 25kB
        -> GroupAggregate (cost=29495.24..35450.09 rows=1270 width=76) (actual time=107.659..218.560 rows=9 loops=1)
              Group Key: "ProfileInfo".profile_id
              -> Merge Join (cost=29495.24..33059.26 rows=237178 width=44) (actual time=93.619..160.119 rows=237178 loops=1)
                    Merge Cond: ("PricingInfo".profile_id = "ProfileInfo".profile_id)
                    -> Sort (cost=29407.07..30000.02 rows=237178 width=12) (actual time=93.585..117.551 rows=237178 loops=1)
                          Sort Key: "PricingInfo".profile_id
                          Sort Method: external merge Disk: 5128kB
                          -> Seq Scan on "PricingInfo" (cost=0.00..4175.78 rows=237178 width=12) (actual time=0.008..41.635 rows=237178 loops=1)
                    -> Sort (cost=88.17..91.35 rows=1270 width=36) (actual time=0.028..0.037 rows=9 loops=1)
                          Sort Key: "ProfileInfo".profile_id
                          Sort Method: quicksort Memory: 25kB
                          -> Seq Scan on "ProfileInfo" (cost=0.00..22.70 rows=1270 width=36) (actual time=0.013..0.016 rows=13 loops=1)
              1)
Planning Time: 0.189 ms
Execution Time: 220.141 ms
(18 rows)

```

Create an index for profile_id in PricingInfo.

```

QUERY PLAN
-----
Limit (cost=16976.53..16976.57 rows=15 width=76) (actual time=146.873..146.876 rows=9 loops=1)
  -> Sort (cost=16976.53..16979.70 rows=1270 width=76) (actual time=146.872..146.874 rows=9 loops=1)
        Sort Key: (round(avg(("PricingInfo".price / 100)), 2))
        Sort Method: quicksort Memory: 25kB
        -> GroupAggregate (cost=0.45..16945.37 rows=1270 width=76) (actual time=17.135..146.837 rows=9 loops=1)
              Group Key: "ProfileInfo".profile_id
              -> Merge Join (cost=0.45..14554.54 rows=237178 width=44) (actual time=0.050..87.498 rows=237178 loops=1)
                    Merge Cond: ("ProfileInfo".profile_id = "PricingInfo".profile_id)
                    -> Index Scan using "ProfileInfo_pkey" on "ProfileInfo" (cost=0.15..67.20 rows=1270 width=36) (actual time=0.008..0.023 rows=10 loops=1)
                    -> Index Scan using profile_id on "PricingInfo" (cost=0.29..11519.44 rows=237178 width=12) (actual time=0.037..44.057 rows=237178 loops=1)
              1)
Planning Time: 0.327 ms
Execution Time: 146.984 ms
(12 rows)

```

Creating an index for profile_id in PricingInfo substantially reduces the execution time of the second advanced query. Looking at the produced execution tree, we can see that the merge condition is simplified from the tree produced with no added indices. That is indicative of the performance gains we see in the final time.

Create an index for profile_id and price in PricingInfo.


```

QUERY PLAN
-----
Limit (cost=16976.53..16976.57 rows=15 width=76) (actual time=146.212..146.216 rows=9 loops=1)
  -> Sort (cost=16976.53..16979.70 rows=1270 width=76) (actual time=146.210..146.213 rows=9 loops=1)
    Sort Key: (round(avg(("PricingInfo".price / 100)), 2))
    Sort Method: quicksort Memory: 25kB
    -> GroupAggregate (cost=0.45..16945.37 rows=1270 width=76) (actual time=16.509..146.156 rows=9 loops=1)
      Group Key: "ProfileInfo".profile_id
      -> Merge Join (cost=0.45..14554.54 rows=237178 width=44) (actual time=0.021..87.677 rows=237178 loops=1)
        Merge Cond: ("ProfileInfo".profile_id = "PricingInfo".profile_id)
        -> Index Scan using "ProfileInfo_pkey" on "ProfileInfo" (cost=0.15..67.20 rows=1270 width=36) (actual time=0.006..0.023 rows=10 loops=1)
        -> Index Scan using profile_id on "PricingInfo" (cost=0.29..11519.44 rows=237178 width=12) (actual time=0.011..43.269 rows=237178 loops=1)
      Planning Time: 0.397 ms
      Execution Time: 146.292 ms
(12 rows)

```

There seems to be no improvement when we add a price index to the PricingInfo relation. When calculating the average, there's not really a need to read the prices in a specific order because you're going to read all the prices for a given user profile no matter what to get an accurate average. It makes sense that the performance is nearly identical to the previous design that uses only the profile_id index.

Create an index for profile_id, price, and round_trip_id in PricingInfo.

```

QUERY PLAN
-----
Limit (cost=16976.53..16976.57 rows=15 width=76) (actual time=144.471..144.475 rows=9 loops=1)
  -> Sort (cost=16976.53..16979.70 rows=1270 width=76) (actual time=144.470..144.472 rows=9 loops=1)
    Sort Key: (round(avg(("PricingInfo".price / 100)), 2))
    Sort Method: quicksort Memory: 25kB
    -> GroupAggregate (cost=0.45..16945.37 rows=1270 width=76) (actual time=16.589..144.425 rows=9 loops=1)
      Group Key: "ProfileInfo".profile_id
      -> Merge Join (cost=0.45..14554.54 rows=237178 width=44) (actual time=0.019..86.631 rows=237178 loops=1)
        Merge Cond: ("ProfileInfo".profile_id = "PricingInfo".profile_id)
        -> Index Scan using "ProfileInfo_pkey" on "ProfileInfo" (cost=0.15..67.20 rows=1270 width=36) (actual time=0.006..0.023 rows=10 loops=1)
        -> Index Scan using profile_id on "PricingInfo" (cost=0.29..11519.44 rows=237178 width=12) (actual time=0.010..42.842 rows=237178 loops=1)
      Planning Time: 0.378 ms
      Execution Time: 144.535 ms
(12 rows)

```

Similar to what we saw in Design 2, we see negligible differences in execution time and the trees produced during the planning process. It seems that only profile_id helped. One theory for this outcome could be that all prices and round trip IDs need to be read, so there's not really a point in having those indexed because we're not looking for a particular subset. However, that justification could also be used with profile_id, but the profile_id could still be helpful because that's the field we're grouping by, so the index still could provide some utility when performing other operation in the query.

For the final implementation, we will keep only the profile_id in the PricingInfo relation for now given the results we saw above with the second advanced query. It may be helpful to re-add the fields above if new queries are going to be regularly used, only profile_id seems useful. All other indices were negligible. We can see that the profile_id index is being used in the last two steps where an index scan on profile_id is being performed. Without that index, the database couldn't have performed an index scan resulting in a more intensive algorithm to produce the same results. In short, given the near identical performance of Design 1, 2, and 3, we will only keep the profile_id index unless other queries call for new indices.