Project Phase-2
Group-3

Data Loading

We decided to use mongoDB for our document-oriented database. The fields in the database match those found in our relational database, more specifically we have the following collections with fields:

1. Trip:
    a. id
    b. passengercount
    c. tripdistance
    d. storeandfwdflag
    e. fareamount
    f. extra
    g. mtatax
    h. improvementsurcharge
    i. tipamount
    j. tollsamount
    k. totalamount
    l. congestionsurcharge
    m. airportfee
    n. vendor
    o. paymenttype
    p. ratecode
    q. pickuplocation
    r. Dropofflocation
2. Payment
    a. Id
    b. Description
3. Ratecode
    a. Id
    b. Description

4. Time
    a. Tripid
    b. Pickupdate
    c. Pickuptime
    d. Dropoffdate
    e. Dropofftime
    f. Dayofweek
    g. Isweekend
5. Location
    a. Id
    b. Borough
    c. Zone
6. Vendor
    a. Id
    b. name

We did not decide to change any collection or field names when changing over to the document database format, we determined that it was not necessary and the data was organized in a meaningful way that was transferable. Data was exported from the relational database using row_to_json, and stripping nulls. There were no extensive text fields so not much manipulation of the exported data was needed before loading. Data was then directly loaded into the mongoDB database collections using mongoDB Compasses built in import.

Queries

1.

Query details:

a. Description: The query identifies highest-earning zones in each borough for each day of the week.
b. Detailed steps:
    i. First it creates a Common Table Expression named EarningsPerZone that,
        1. Joins three tables: trip, time and location
        2. Groups data by borough, zone and day of the week
        3. Calculates total earnings for each group
    ii. The main query then selects from the CTE
        1. Uses a subquery to find the zones with maximum earnings

2. Compares earnings within the same borough and day of the week
3. Orders the results by day of week and total earnings descending

c. Query can be useful for
   i. Identifying prime location for taxi services on specific days
   ii. Optimizing driver deployment across zones
   iii. Understanding weekly earning patterns by location

```sql
 WITH EarningsPerZone AS (
   SELECT
      l.borough,
      l.zone,
      t.dayofweek,
      SUM(tr.totalamount) AS totalearnings
   FROM trip tr
   JOIN time t ON tr.id = t.tripid
   JOIN location l ON tr.pickuplocation = l.id
   GROUP BY l.borough, l.zone, t.dayofweek
)

SELECT
   epz.borough,
   epz.zone,
   epz.dayofweek,
   epz.totalearnings
FROM EarningsPerZone epz
WHERE epz.totalearnings = (
   SELECT MAX(totalearnings)
   FROM EarningsPerZone epz2
   WHERE epz2.dayofweek = epz.dayofweek
   AND epz2.borough = epz.borough
)
ORDER BY epz.dayofweek, epz.totalearnings DESC;
```

Data Output Messages Notifications

	borough character varying	zone character varying	dayofweek integer	totalearnings numeric
1	Queens	JFK Airport	0	11464909
2	Manhattan	Midtown Center	0	2844059
3	Unknown	[null]	0	292178
4	Brooklyn	Downtown Brooklyn/Metr...	0	38286
5	EWR	Newark Airport	0	32467
6	Bronx	Mott Haven/Port Morris	0	16204
7	Staten Island	Arrochar/Fort Wadsworth	0	1838
8	Queens	JFK Airport	1	12934300
9	Manhattan	Midtown Center	1	3570581
10	Unknown	[null]	1	307519
11	Brooklyn	Downtown Brooklyn/Metr...	1	41978
12	EWR	Newark Airport	1	32639
13	Bronx	Co-Op City	1	20850
14	Staten Island	Heartland Village/Todt Hill	1	715
15	Queens	JFK Airport	2	12471560
16	Manhattan	Midtown Center	2	3970912
17	Unknown	[null]	2	351554
18	Brooklyn	Downtown Brooklyn/Metr...	2	40201
19	EWR	Newark Airport	2	34638

Total rows: 49 of 49 Query complete 00:00:21.378 Ln 26, Col 48

Execution Time : 21.378 seconds

2.

Query details:

d. Description: This query analyzes trip data, calculating averages for trip distances and fares while grouping them by payment type and rate code.
e. Detailed steps:
    i. First it selects the descriptions of ratecode and paymenttype, along with average tripdistance and fareamounts, based on the joins of trip with ratecode and paymenttype.
    ii. Finally the results are grouped by payment and ratecode descriptions, calculating averages for each unique combination of payment type and rate code.
    iii. The results are then ordered by average fare amount in ascending order.
f. Query can be useful for
    i. How fare amounts vary across different payment methods and rate codes.
    ii. The relationship between trip distances and payment types.
    iii. Patterns in rate code usage across different payment methods.

```
select p.description as paymenttype, r.description as ratecode, avg(tr.tripdistance) as
avgtripdistance, avg(tr.fareamount) as avgfareamount
from trip tr
join payment p on tr.paymenttype = p.id
join ratecode r on tr.ratecode = r.id
group by p.description, r.description
order by avgfareamount;
```

Data Output Messages Notifications

	paymenttype character varying	ratecode character varying	avgtripdistance numeric	avgfareamount numeric
1	Unknown	Standard rate	0.00000000000000000000	0.00000000000000000000
2	Dispute	Group ride	1.8333333333333333	1.00000000000000000000
3	Dispute	Standard rate	2.7607561821886876	1.0638458996530264
4	Cash	Group ride	2.2727272727272727	2.1818181818181818
5	Dispute	JFK	10.8196235904924150	3.6176279711203050
6	Credit card	Group ride	2.0000000000000000	4.0000000000000000
7	Dispute	Newark	7.7838638045891932	4.5835183814458426
8	Dispute	Nassau or Westchester	21.9091201027617213	4.7755298651252408
9	Dispute	Negotiated fare	4.3197887970615243	5.0735766758494031
10	No charge	Standard rate	1.9729504459578833	5.1691790385546975

Total rows: 25 of 25 Query complete 00:00:05.932 Ln 55, Col 19

Execution Time: 5.932 seconds

3.

Query details:

a. Description: The query analyzes the trip data to find the top 10 highest-revenue zone-to-zone combinations.
b. Detailed steps:
    i. The query first selects the origin and destination zones and calculates the total revenue
    ii. Next it performs two joins of trip with location, first one links pickuplocation with location ids and the second join links dropofflocation with location ids
    iii. Group results by both origin and destination zones to get zone-pair totals
    iv. Finally, orders results by total revenue in descending order and takes only top 10 results
c. Query can be useful for
    i. Analyzing which zone-to-zone routes generate the most revenue, which could help with resource allocation or pricing strategies.

```
select originlocation.zone as originzone,
        destlocation.zone as destzone,
        sum(trip.totalamount) as totalrevenue
from trip
join location as originlocation on trip.pickuplocation = originlocation.id
join location as destlocation on trip.dropofflocation = destlocation.id
group by originzone, destzone
order by totalrevenue desc
limit 10;
```

Data Output Messages Notifications

	originzone character varying	destzone character varying	totalrevenue numeric
1	JFK Airport	Outside of NYC	5472556
2	JFK Airport	Times Sq/Theatre District	4687438
3	LaGuardia Airport	Times Sq/Theatre District	3573022
4	JFK Airport	Midtown South	2519009
5	JFK Airport	Clinton East	2433971
6	Upper East Side South	Upper East Side North	2337303
7	Times Sq/Theatre District	LaGuardia Airport	2253767
8	LaGuardia Airport	Midtown Center	2078031
9	Upper East Side North	Upper East Side South	2049610
10	JFK Airport	Midtown Center	2019320

Total rows: 10 of 10 Query complete 00:00:09.202 Ln 37, Col 1

Execution Time : 9.202 seconds

4.

Query details:

a. Description: The query performs monthly analysis of trip data, calculating trip counts , total revenue, and average tip amounts per month.
b. Detailed steps:
   i. Converts specific pickup dates to month-level groupings
   ii. Joins trip table with time table based on tripid. This join ensures that we have access to both timing and financial information for each trip
   iii. Performs aggregations, calculating total trips, total revenue and average tip amount per month
   iv. Finally, groups and orders results by months
c. Query can be useful for
   ii. Seasonal peaks and troughs in taxi demand
   iii. Correlation between trip volume and total revenue
   iv. Changes in customer tipping behavior over time
   v. Monthly revenue patterns for financial planning

```
select date_trunc('month', time.pickupdate) as monthyear,
       count(*) as tripcount,
       sum(trip.totalamount) as totalrevenue,
       avg(trip.tipamount) as avgtipamount
from trip
join time on time.tripid = trip.id
group by monthyear
order by monthyear;
```

Data Output Messages Notifications

	monthyear timestamp with time zone	tripcount bigint	totalrevenue numeric	avgtipamount numeric
1	2002-12-01 00:00:00-05	10	132	2.1000000000000000
2	2008-12-01 00:00:00-05	5	106	2.8000000000000000
3	2009-01-01 00:00:00-05	12	511	4.1666666666666667
4	2023-12-01 00:00:00-05	10	226	2.7000000000000000
5	2024-01-01 00:00:00-05	2824455	75833127	3.4314899688612493
6	2024-02-01 00:00:00-05	2821923	75760026	3.4311896532967058
7	2024-03-01 00:00:00-05	3156421	84366178	3.4212904425613693
8	2024-04-01 00:00:00-04	3105719	83118915	3.4248217562503240
9	2024-05-01 00:00:00-04	3319177	88840481	3.4264903619180297
10	2024-06-01 00:00:00-04	3128389	83681205	3.4235397196448396
11	2024-07-01 00:00:00-04	2797870	75134017	3.4299274090647528
12	2024-08-01 00:00:00-04	29	580	2.7586206896551724
13	2026-06-01 00:00:00-04	2	42	1.00000000000000000000

Total rows: 13 of 13 Query complete 00:00:20.331 Ln 48, Col 1

Execution Time : 20.331 seconds

5.

Query details:

a. Description: The query analyzes how congestion surcharges impact trip metrics by comparing trips with and without surcharges, providing insights into revenue patterns and trip volumes for both categories.

b. Detailed steps:

   i. First it categorizes the data into two categories, ‘with surcharge’ having congestion surcharge > 0 and ‘no surcharge’ where the congestion surcharge is 0

   ii. Then counts total number of trips and sums up total revenues and calculates average revenue per trip

   iii. Groups results by surcharge status and orders results by trip count in descending order to show most common category first

d. Query can be useful for

   i. Understanding how congestion pricing affects trip volumes

   ii. Understanding revenue implications of congestion pricing

```
select
	case when trip.congestionsurcharge > 0 then 'with surcharge' else 'no surcharge' end as surchargestatus,
	count(*) as tripcount,
	sum(trip.totalamount) as totalrevenue,
	avg(trip.totalamount) as avgtriprevenue
from trip
group by surchargestatus
order by tripcount desc;
```

Data Output | Messages | Notifications

	surchargestatus text	tripcount bigint	totalrevenue numeric	avgtriprevenue numeric
1	with surcharge	19226288	513122565	26.6885924625699979
2	no surcharge	1927734	75984225	39.4163432299269505

Total rows: 2 of 2 Query complete 00:00:02.481 Ln 58, Col 1

Execution Time : 2.481 seconds

Indexes

1.

Tried below indexes:

```
create index time_dayofweek_idx on time (dayofweek);
create index location_borough_zone_idx on location (borough, zone);
create index trip_pul_idx on trip (pickuplocation);
create index time_tripid_dayofweek_idx on time (tripid, dayofweek);
```

Effective indexes:

```
create index time_tripid_dayofweek_idx on time (tripid, dayofweek);
```

	borough character varying	zone character varying	dayofweek integer	totalearnings numeric
1	Queens	JFK Airport	0	11464909
2	Manhattan	Midtown Center	0	2844059
3	Unknown	[null]	0	292178
4	Brooklyn	Downtown Brooklyn/Metr...	0	38286
5	EWR	Newark Airport	0	32467
6	Bronx	Mott Haven/Port Morris	0	16204
7	Staten Island	Arrochar/Fort Wadsworth	0	1838
8	Queens	JFK Airport	1	12934300
9	Manhattan	Midtown Center	1	3570581
10	Unknown	[null]	1	307519
11	Brooklyn	Downtown Brooklyn/Metr...	1	41978
12	EWR	Newark Airport	1	32639
13	Bronx	Co-Op City	1	20850
14	Staten Island	Heartland Village/Todt Hill	1	715
15	Queens	JFK Airport	2	12471560
16	Manhattan	Midtown Center	2	3970912
17	Unknown	[null]	2	351554
18	Brooklyn	Downtown Brooklyn/Metr...	2	40201
19	EWR	Newark Airport	2	34638

Total rows: 49 of 49 Query complete 00:00:09.873 Ln 26, Col 48

Execution time after indexing - 9.873 seconds

Execution time before indexing - 21.378 seconds

Reason for improvement:

The index is particularly useful because it allows the database to efficiently retrieve rows from the time table by the combination of tripid and dayofweek. The scan accesses only the necessary indexed columns, avoiding a full table scan and reducing I/O overhead.

2.

Tried below indexes

```
create index trip_ratecode_idx on trip (ratecode);
create index trip_paymenttype_idx on trip (paymenttype);
create index ratecode_id_desc_idx on ratecode (id, description);
create index payment_id_desc_idx on payment (id, description);
create index trip_pttype_rtcode_fare_dist_idx on trip (paymenttype, ratecode, tripdistance,
fareamount);
```

None of the above indexes had any effect on the execution time of the query and the execution plan.

Possible reasons:

1. The indexes on ratecode and paymenttype do not significantly improve the performance of the join operations in this query. This is because both the ratecode and payment tables contain a relatively small number of rows (in the tens), which is negligible compared to the millions of rows in the trip table. As a result, these indexes don't contribute much to optimizing the join performance.
2. Furthermore, the ORDER BY clause in the query relies on the avgfareamount, which is an aggregation of trip.fareamount. Since this aggregation is computed dynamically during query execution, it is not possible to create a useful index for it.

3.

Tried below indexes

```
create index trip_puloc_idx on trip (pickuplocation);
create index trip_dloc_idx on trip (dropofflocation);
create index trip_puloc_dloc_idx on trip (pickuplocation, dropofflocation);
create index location_zone_idx on location (zone);
create index location_id_zone_idx on location (id, zone);
```

```sql
create index trip_totalamount_idx on trip (totalamount);
```

None of the above indexes had any effect on the execution time of the query and the execution plan.

Possible reasons:

1. The database optimizer ignores the zone index during joins because the low number of distinct zones (hundreds) compared to trip records (millions) means each zone lookup would require accessing such a large portion of data that a sequential table scan becomes more efficient.
2. The ORDER BY clause uses totalrevenue, which is calculated by SUM(totalamount). Since this is an aggregated value computed during query execution, no pre-built index can help optimize the sorting, as indices can only be built on actual stored table columns, not on computed values.
4.

Tried below indexes

```sql
create index time_tripid_idx on time (tripid);
create index time_tripid_pickupdate_idx on time (tripid, pickupdate);
```

Effective index

```sql
create index time_tripid_pickupdate_idx on time (tripid, pickupdate);
```

	monthyear timestamp with time zone	tripcount bigint	totalrevenue numeric	avgtipamount numeric
1	2002-12-01 00:00:00-05	10	132	2.1000000000000000
2	2008-12-01 00:00:00-05	5	106	2.8000000000000000
3	2009-01-01 00:00:00-05	12	511	4.1666666666666667
4	2023-12-01 00:00:00-05	10	226	2.7000000000000000
5	2024-01-01 00:00:00-05	2824455	75833127	3.4314899688612493
6	2024-02-01 00:00:00-05	2821923	75760026	3.4311896532967058
7	2024-03-01 00:00:00-05	3156421	84366178	3.4212904425613693
8	2024-04-01 00:00:00-04	3105719	83118915	3.4248217562503240
9	2024-05-01 00:00:00-04	3319177	88840481	3.4264903619180297
10	2024-06-01 00:00:00-04	3128389	83681205	3.4235397196448396
11	2024-07-01 00:00:00-04	2797870	75134017	3.4299274090647528
12	2024-08-01 00:00:00-04	29	580	2.7586206896551724
13	2026-06-01 00:00:00-04	2	42	1.00000000000000000000

Total rows: 13 of 13 Query complete 00:00:11.921 Ln 48, Col 1

Execution time after indexing - 11.921 seconds

Execution time before indexing - 20.331 seconds

Reason for improvement:

Here, the index ensures that the time table can be scanned quickly for the relevant rows based on tripid and pickupdate timestamp without the need for a full table scan. Since the query is focussed on grouping by pickupdate the index also helps by allowing the database to directly access the rows that need to be processed, making it much faster than scanning the entire table.

5.

Tried below indexes

```
create index trip_totalamount_idx on trip (totalamount);
create index trip_congestionsurcharge_idx on trip (congestionsurcharge);
```

None of the above indexes had any effect on the execution time of the query and the execution plan.

Possible reasons:

1. Aggregations functions used here like sum() and average() often require a full scan of the relevant columns, especially when calculating aggregates across all rows. Indexes are less effective for such operations because the database may determine that a sequential scan is more efficient than using an index.
2. Low selectivity on congestionsurcharge could be one of the possible reasons. If most of the rows have similar values, which is the case here, then an index on this column might not be selective enough to benefit the query.

Part 3: Valid Functional Dependencies

Output for Valid Functional Dependencies(valid_dependencies.txt file)

```
Valid Functional Dependencies

----------- Functional Dependencies for Location -----------
id -> borough
id -> zone
zone -> borough

----------- Functional Dependencies for RateCode -----------
id -> description
description -> id

----------- Functional Dependencies for Payment -----------
id -> description
description -> id

----------- Functional Dependencies for Vendor -----------
id -> name
name -> id

----------- Functional Dependencies for Trip -----------
id -> passengercount
id -> tripdistance
id -> storeandfwdflag
id -> fareamount
id -> extra
id -> tipamount
id -> tollsamount
id -> totalamount
id -> congestionsurcharge
id -> airportfee
id -> vendor
id -> paymenttype
id -> ratecode
id -> pickuplocation
id -> dropofflocation
```

Functional Dependencies and Normalization

Code Structure and Setup:

We utilized the script get_functional_dependencies.py to compute functional dependencies for the schema discussed. This script connects to a PostgreSQL database, analyzes attribute relationships in each table, and produces two outputs:

1. A file listing **valid functional dependencies.**
2. A file listing **pruned functional dependencies** (dependencies that were discarded).

The script imposes the following constraints during FD discovery:

- **LHS Restriction:** Limited to at most 2 attributes.
- **RHS Restriction:** Limited to 1 attribute.

Code Structure:

1. **FunctionalDependencyDiscovery Class**:
   - Manages FD discovery for individual tables.
   - Includes methods to:
     - Fetch and process table data.
     - Compute attribute partitions.
     - Validate functional dependencies using a lattice traversal algorithm.
     - Handle positive and negative pruning during FD discovery.
2. **DatabaseConnection Class**:
   - Manages PostgreSQL database connections and queries.
3. **Main Function**:
   - Hardcodes table names and their primary keys.
   - Iterates through tables to compute and write functional dependencies.

Key Observations for the Trip Table:

1. **mtatax:**
   - This attribute exhibits very few distinct values.
   - These limited values cause many irrelevant FDs, where random LHS combinations appear to "determine" it.
   - Excluded from RHS to avoid spurious dependencies.
2. **improvementsurcharge:**
   - Similarly, this attribute has only three distinct values (1, 0, -1) and was excluded from being in RHS.

Functional Dependencies and Normalization Discussion

Functional Dependencies for Location

- **Discovered FDs**:
  - id → borough
  - id → zone
  - zone → borough
- **Candidate Key**: id
- **Analysis**:
  - A transitive dependency exists: id → zone → borough.
- **3NF Decomposition**:
  - Decompose into two tables:
    1. {id, zone} with id → zone.
    2. {zone, borough} with zone → borough.

Functional Dependencies for RateCode

- **Discovered FDs**:
  - id → description
  - description → id
- **Candidate Keys**: Both id and description
- **Analysis**:
  - 1:1 relationship exists between id and description.
- **Normalization Status**: Already in BCNF.

Functional Dependencies for Payment

- **Discovered FDs**:
  - id → description
  - description → id
- **Candidate Keys**: Both id and description
- **Normalization Status**: Already in BCNF.

Functional Dependencies for Vendor

- **Discovered FDs**:
    - id → name
    - name → id
- **Candidate Keys**: Both id and name
- **Normalization Status**: Already in BCNF.

Functional Dependencies for Trip

- **Discovered FDs**:
    - id → passengercount
    - id → tripdistance
    - id → storeandfwdflag
    - id → fareamount
    - id → extra
    - id → tipamount
    - id → tollsamount
    - id → totalamount
    - id → congestionsurcharge
    - id → airportfee
    - id → vendor
    - id → paymenttype
    - id → ratecode
    - id → pickuplocation
    - id → dropofflocation
- **Candidate Key**: id
- **Analysis**:
    - The primary key id determines all other attributes.
    - No transitive dependencies exist.
- **Normalization Status**: Already in 3NF.

Normalization

1. **1NF Validation**:
    - All tables satisfy 1NF.
    - All attributes are atomic, and no repeating groups are present.
2. **2NF Validation**:
    - All tables satisfy 2NF:
        - Location and Trip tables:
            - The primary key (id) fully determines all non-prime attributes.
        - Other tables:
            - No partial dependencies exist as there are only two attributes, and each attribute fully determines the other.
3. **3NF Validation**:
    - All tables satisfy 3NF except for the **Location** table:
        - A transitive dependency (id → zone → borough) exists.
        - Decomposition into two tables resolves the issue, as discussed earlier.

Following these steps ensures the schema is properly normalized up to BCNF.