STAGE 3

The database is implemented on GCP.
Below is the screenshot of the connection and the database tables:

```
supercoolkinjalk@cloudshell:~ (cs411-team124-quertyqueries)$ gcloud sql connect cs411-team124-quertyqueries-db --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.

Connecting to database with SQL user [root].Enter password:
\Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 22
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use Transport;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show Tables;
+---------------------+
| Tables_in_Transport |
+---------------------+
| Books               |
| Feedback            |
| Frequencies         |
| Has                 |
| Routes              |
| Shapes              |
| Stops               |
| Trips               |
| Users               |
+---------------------+
```

Data Definition Language (DDL) commands of all tables:

1. DDL of 'Trips' table:

```
mysql> show Create Table Trips;
+-------+------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------
| Table | Create Table


+-------+------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------
| Trips | CREATE TABLE `Trips` (
  `route_id` varchar(100) DEFAULT NULL,
  `service_id` varchar(3) DEFAULT NULL,
  `trip_id` varchar(100) NOT NULL,
  `trip_headsign` varchar(100) DEFAULT NULL,
  `direction_id` int DEFAULT NULL,
  `shape_id` int DEFAULT NULL,
  PRIMARY KEY (`trip_id`),
  KEY `fk_shape_id` (`shape_id`),
  KEY `fk_route_id` (`route_id`),
  CONSTRAINT `fk_route_id` FOREIGN KEY (`route_id`) REFERENCES `Routes` (`route_id`),
  CONSTRAINT `fk_shape_id` FOREIGN KEY (`shape_id`) REFERENCES `Shapes` (`shape_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+-------+------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

2. DDL of 'Routes' table:

```
mysql> show create table Routes;
+--------+-------------------------------------------------------------
-----------------------------------------------------------------------
| Table  | Create Table

+--------+-------------------------------------------------------------
-----------------------------------------------------------------------
| Routes | CREATE TABLE `Routes` (
  `route_id` varchar(100) NOT NULL,
  `route_long_name` varchar(100) NOT NULL,
  `route_type` int NOT NULL,
  `route_color` varchar(100) NOT NULL,
  `route_text_color` varchar(100) DEFAULT NULL,
  `fare` int DEFAULT '1',
  PRIMARY KEY (`route_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+--------+-------------------------------------------------------------
-----------------------------------------------------------------------
1 row in set (0.00 sec)
```

3. DDL of 'Shapes' table:

```
mysql> show create table Shapes;
+--------+-----------------------------------------------------------------------------
-------------------------------------------------------------------------------------
| Table  | Create Table

+--------+-----------------------------------------------------------------------------
-------------------------------------------------------------------------------------
| Shapes | CREATE TABLE `Shapes` (
  `shape_id` int NOT NULL,
  `shape_pt_lat` decimal(10,6) NOT NULL,
  `shape_pt_lon` decimal(10,6) NOT NULL,
  `shape_pt_sequence` int NOT NULL,
  `shape_dist_traveled` decimal(13,8) NOT NULL,
  PRIMARY KEY (`shape_id`,`shape_pt_sequence`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+--------+-----------------------------------------------------------------------------
-------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

4. DDL of ‘Stops’ table:

```
mysql> show create table Stops;
+-------+-------------------------------------------------------------
----------------------------------+
| Table | Create Table
                                  |
+-------+-------------------------------------------------------------
----------------------------------+
| Stops | CREATE TABLE `Stops` (
  `stop_id` int NOT NULL,
  `stop_name` varchar(100) DEFAULT NULL,
  `stop_lat` double NOT NULL,
  `stop_lon` double NOT NULL,
  PRIMARY KEY (`stop_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+-------+-------------------------------------------------------------
----------------------------------+
1 row in set (0.00 sec)
```

5. DDL of ‘Books’ table:

```
ERROR 1064 (42000): You have an error in your
mysql> show create table Books;
+-------+----------------------------------------
| Table | Create Table
+-------+----------------------------------------
| Books | CREATE TABLE `Books` (
  `user_id` int NOT NULL,
  `trip_id` int NOT NULL,
  PRIMARY KEY (`user_id`,`trip_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+-------+----------------------------------------
1 row in set (0.00 sec)
```

6. DDL of ‘Feedback’ table:

```
mysql> show create Table Feedback;
+----------+----------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
| Table    | Create Table


+----------+----------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
| Feedback | CREATE TABLE `Feedback` (
  `feedback_id` int NOT NULL AUTO_INCREMENT,
  `trip_id` varchar(100) NOT NULL,
  `user_id` int NOT NULL,
  `email_id` varchar(100) NOT NULL,
  `feedback` varchar(255) NOT NULL,
  PRIMARY KEY (`feedback_id`),
  KEY `email_id` (`email_id`),
  KEY `trip_id` (`trip_id`),
  KEY `user_id` (`user_id`),
  CONSTRAINT `Feedback_ibfk_1` FOREIGN KEY (`trip_id`) REFERENCES `Trips` (`trip_id`),
  CONSTRAINT `Feedback_ibfk_2` FOREIGN KEY (`user_id`) REFERENCES `Users` (`User_Id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+----------+----------------------------------------------------------------------------------------
```

7. DDL of ‘Frequencies’ table:

```
mysql> show create table Frequencies;
+-------------+-----------------------------------------------------------------
---------------------------------------------------------------------------------
| Table       | Create Table

+-------------+-----------------------------------------------------------------
---------------------------------------------------------------------------------
| Frequencies | CREATE TABLE `Frequencies` (
  `trip_id` varchar(100) NOT NULL,
  `start_time` varchar(10) NOT NULL,
  `end_time` varchar(10) NOT NULL,
  `headway_secs` int NOT NULL,
  PRIMARY KEY (`trip_id`,`start_time`),
  CONSTRAINT `fk_trip_id` FOREIGN KEY (`trip_id`) REFERENCES `Trips` (`trip_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+-------------+-----------------------------------------------------------------
```

8. DDL of ‘Has’ table:

```
mysql> show create table Has;
+-------+-----------------------------------------------
------------------------------------------------------
| Table | Create Table

+-------+-----------------------------------------------
------------------------------------------------------
| Has   | CREATE TABLE `Has` (
  `trip_id` varchar(100) NOT NULL,
  `stop_id` int NOT NULL,
  `arrival_time` varchar(100) DEFAULT NULL,
  `departure_time` varchar(100) DEFAULT NULL,
  `stop_sequence` int NOT NULL,
  PRIMARY KEY (`trip_id`,`stop_id`,`stop_sequence`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3 |
+-------+-----------------------------------------------
```

9. DDL of ‘Users’ table:

```
+-------+------------------------------------------------------------
---------------------------------------------------------------------
| Users | CREATE TABLE `Users` (
  `User_Id` int NOT NULL AUTO_INCREMENT,
  `First_Name` varchar(100) DEFAULT NULL,
  `Last_Name` varchar(100) DEFAULT NULL,
  `Email` varchar(100) DEFAULT NULL,
  `Password` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`User_Id`),
  UNIQUE KEY `Email` (`Email`)
) ENGINE=InnoDB AUTO_INCREMENT=1001 DEFAULT CHARSET=utf8mb3 |
+-------+------------------------------------------------------------
```

Inserting at least 1000 rows in these 4 tables:

1. Count of Trips:

```
mysql> Select count(*) from Trips;
+----------+
| count(*) |
+----------+
|     2227 |
+----------+
1 row in set (0.07 sec)
```

2. Count of Routes:

```
mysql> Select count(*) from Routes;
+----------+
| count(*) |
+----------+
|     1360 |
+----------+
1 row in set (0.00 sec)
```

3. Count of Shapes:

```
mysql> Select count(*) from Shapes;
+----------+
| count(*) |
+----------+
|  1128782 |
+----------+
1 row in set (8.79 sec)
```

4. Count of Stops:

```
mysql> SELECT COUNT(*) FROM Stops;
+----------+
| COUNT(*) |
+----------+
|    20902 |
+----------+
1 row in set (0.01 sec)
```

SQL Queries:

Query 1:

```sql
SELECT t.route_id, AVG(f.headway_secs) AS average_headway
FROM Trips t
LEFT JOIN Frequencies AS f
ON t.trip_id = f.trip_id
WHERE t.direction_id = 1
AND t.shape_id LIKE '58%' GROUP BY t.route_id ORDER BY
average_headway DESC;
```

```
mysql> Explain Analyze SELECT t.route_id, AVG(f.headway_secs) AS average_
+-----------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
| EXPLAIN


+-----------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
| -> Sort: average_headway DESC  (actual time=2.209..2.211 rows=35 loops=
    -> Table scan on <temporary>  (actual time=2.168..2.174 rows=35 loops
        -> Aggregate using temporary table  (actual time=2.167..2.167 row
            -> Nested loop left join  (cost=280.21 rows=482) (actual time
                -> Filter: ((t.direction_id = 1) and (t.shape_id like '58
                    -> Table scan on t  (cost=225.70 rows=2227) (actual t
                -> Index lookup on f using PRIMARY (trip_id=t.trip_id)  (
 |
+-----------------------------------------------------------------------
------------------------------------------------------------------------
------------------------------------------------------------------------
1 row in set (0.01 sec)

mysql> SELECT t.route_id, AVG(f.headway_secs) AS average_headway FROM Tri
+----------+-----------------+
| route_id | average_headway |
+----------+-----------------+
| 748A-41  |       2936.8421 |
| N304-11  |       2880.0000 |
| 119L-10  |       2652.6316 |
| 6115-41  |       2475.0000 |
| 6232-10  |       2052.6316 |
| 5100-10  |       1847.3684 |
| 1021-10  |       1757.1429 |
| 8055-51  |       1610.5263 |
| 5033-10  |       1581.8182 |
| 7059-10  |       1540.9091 |
| 513L-10  |       1486.3636 |
| 6053-10  |       1428.5714 |
| 975A-10  |       1350.0000 |
| 6050-10  |       1350.0000 |
| 1025-10  |       1120.0000 |
+----------+-----------------+
15 rows in set (0.01 sec)
```

Query 2:

```
SELECT r.route_id, r.route_long_name, t.trip_id
FROM Routes AS r
JOIN Trips AS t ON r.route_id = t.route_id
JOIN (
  SELECT route_type
  FROM Routes
  ORDER BY route_type DESC
  LIMIT 10
) AS top_routes ON r.route_type = top_routes.route_type
WHERE (r.route_long_name LIKE '%met%' OR r.route_long_name LIKE
'%jd%')
ORDER BY r.route_long_name DESC;
```

```
mysql> Explain Analyze SELECT r.route_id, r.route_long_name, t.trip_id
    -> FROM Routes AS r
    -> JOIN Trips AS t ON r.route_id = t.route_id
    -> JOIN (
    ->   SELECT route_type
    ->   FROM Routes
    ->   ORDER BY route_type DESC
    ->   LIMIT 10
    -> ) AS top_routes ON r.route_type = top_routes.route_type
    -> WHERE (r.route_long_name LIKE '%met%' OR r.route_long_name LIKE '%jd%')
    -> ORDER BY r.route_long_name DESC;
ERROR 1054 (42S22): Unknown column 'r.route_long_name DESC' in 'order clause'
mysql> SELECT r.route_id, r.route_long_name, t.trip_id
    -> FROM Routes AS r
    -> JOIN Trips AS t ON r.route_id = t.route_id
    -> JOIN (
    ->   SELECT route_type
    ->   FROM Routes
    ->   ORDER BY route_type DESC
    ->   LIMIT 10
    -> ) AS top_routes ON r.route_type = top_routes.route_type
    -> WHERE (r.route_long_name LIKE '%met%' OR r.route_long_name LIKE '%jd%')
    -> ORDER BY r.route_long_name DESC LIMIT 15;
+----------+------------------------------+-----------+
| route_id | route_long_name              | trip_id   |
+----------+------------------------------+-----------+
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-0 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-1 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-1 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-1 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-1 |
| 172P-10  | Vl. Zilda - MetrÃ´ BelÃ©m     | 172P-10-1 |
+----------+------------------------------+-----------+
15 rows in set (0.01 sec)
```

EXPLAIN ANALYZE

Query 1:

Without Index:

This is the default explain analyze run using the primary key index for comparison purposes. Possibly, we will be able to decrease the cost/time using a few potential indexes that will be tried out below.

```
mysql> Explain Analyze SELECT t.route_id, AVG(f.headway_secs) AS average_headway FROM Trips t LEFT JOIN Frequencies AS f ON t.trip_id = f.trip_id WHERE t.direction_id =
| EXPLAIN
| -> Sort: average_headway DESC  (actual time=2.209..2.211 rows=35 loops=1)
    -> Table scan on <temporary>  (actual time=2.168..2.174 rows=35 loops=1)
        -> Aggregate using temporary table  (actual time=2.167..2.167 rows=35 loops=1)
            -> Nested loop left join  (cost=280.21 rows=482) (actual time=0.093..1.648 rows=689 loops=1)
                -> Filter: ((t.direction_id = 1) and (t.shape_id like '58%'))  (cost=225.70 rows=25) (actual time=0.071..1.175 rows=35 loops=1)
                    -> Table scan on t  (cost=225.70 rows=2227) (actual time=0.058..0.851 rows=2227 loops=1)
                -> Index lookup on f using PRIMARY (trip_id=t.trip_id)  (cost=0.33 rows=19) (actual time=0.009..0.012 rows=20 loops=35)
 |
1 row in set (0.01 sec)
```

1) **Using direction_id as Index idx_dir**

```
mysql> Explain Analyze SELECT t.route_id, AVG(f.headway_secs) AS average_headway FROM Trips t LEFT JOIN Frequencies AS f ON t.trip_id = f.trip_id WHERE t.direction_id = 1  AND t.shape_id LIKE
| EXPLAIN
                                                                                                                                                    |
| -> Sort: average_headway DESC  (actual time=3.442..3.444 rows=35 loops=1)
    -> Table scan on <temporary>  (actual time=3.395..3.405 rows=35 loops=1)
        -> Aggregate using temporary table  (actual time=3.392..3.392 rows=35 loops=1)
            -> Nested loop left join  (cost=252.08 rows=2057) (actual time=0.180..2.706 rows=689 loops=1)
                -> Filter: (t.shape_id like '58%')  (cost=19.55 rows=106) (actual time=0.165..1.970 rows=35 loops=1)
                    -> Index lookup on t using idx_dir (direction_id=1)  (cost=19.55 rows=950) (actual time=0.156..1.695 rows=950 loops=1)
                -> Index lookup on f using PRIMARY (trip_id=t.trip_id)  (cost=0.27 rows=19) (actual time=0.015..0.019 rows=20 loops=35)
 |
1 row in set (0.01 sec)
```

The index of direction_id is being shown in the screenshot above. Here, by adding it we can see that the cost of index lookup for calculating the same number of rows.The **idx_dir** index did bring a better effect. This allows Filter: (t.direction_id = 1) to run more efficiently. Before, this filter cost 235 but now only costs 19.55, and that is a huge jump. There is also a decrease in look-up using the Primary Key from 33 to 27.

2) **Using (shapes_id, direction_id) as index idx_dir_shape**

```
mysql> CreaTE INDEX idx_dir_shape on Trips(direction_id,shape_id);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Explain Analyze SELECT t.route_id, AVG(f.headway_secs) AS average_headway FROM Trips t LEFT JOIN Frequencies AS f ON t.trip_id = f.trip_id WHERE t.direction_id = 1  AND t.shape_id LIKE '58%'
+------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------+
| EXPLAIN
                                                                              |
+------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------+
| -> Sort: average_headway DESC  (actual time=1.660..1.662 rows=35 loops=1)
    -> Table scan on <temporary>  (actual time=1.612..1.617 rows=35 loops=1)
        -> Aggregate using temporary table  (actual time=1.609..1.609 rows=35 loops=1)
            -> Nested loop left join  (cost=1897.53 rows=18511) (actual time=0.590..1.067 rows=689 loops=1)
                -> Index lookup on t using idx_dir_shape (direction_id=1), with index condition: (t.shape_id like '58%')  (cost=19.55 rows=950) (actual time=0.562..0.570 rows=35 loops=1)
                -> Index lookup on f using PRIMARY (trip_id=t.trip_id)  (cost=0.27 rows=19) (actual time=0.009..0.013 rows=20 loops=35)
 |
+------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Here, we are using both direction_id and shape_id as indexes where the cost of the entire operation has reduced tremendously compared to the **idx_dir** where the cost of the **idx_dir** was around 3.4 and the one in this 1.6. We can see that the cost of indexing both the attributes has increased but the time to retrieve data at the sorting has decreased a lot. So, the tradeoff made is worth it.

The idx_dir_shape index is the best index for the query because it is the most selective index. This means that the shapes_id and direction_id columns have a high number of distinct values, which allows the optimizer to narrow down the number of rows that need to be scanned.

The shapes_id and direction_id columns are both used in the WHERE clause of the query, so the optimizer can use the idx_dir_shape index to quickly find the relevant rows.

3)Using (direction_id,route_id) as index idx_dir_route

```
mysql> Create Index idx_dir_route on Trips(direction_id,route_id);
Query OK, 0 rows affected (0.12 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Explain Analyze SELECT t.route_id, AVG(f.headway_secs) AS average_headway FROM Trips t LEFT JOIN Frequencies AS f ON t.trip_id = f.trip_id WHERE
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
| EXPLAIN


+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
| -> Sort: average_headway DESC  (actual time=2.309..2.311 rows=35 loops=1)
    -> Stream results  (cost=457.74 rows=2057) (actual time=0.318..2.279 rows=35 loops=1)
        -> Group aggregate: avg(f.headway_secs)  (cost=457.74 rows=2057) (actual time=0.256..2.197 rows=35 loops=1)
            -> Nested loop left join  (cost=252.08 rows=2057) (actual time=0.232..2.069 rows=689 loops=1)
                -> Filter: (t.shape_id like '58%')  (cost=19.55 rows=106) (actual time=0.213..1.617 rows=35 loops=1)
                    -> Index lookup on t using idx_dir_route (direction_id=1)  (cost=19.55 rows=950) (actual time=0.203..1.433 rows=950 loops=1)
                -> Index lookup on f using PRIMARY (trip_id=t.trip_id)  (cost=0.27 rows=19) (actual time=0.009..0.012 rows=20 loops=35)
 |
+------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

In this we can see that the route_id is being used as a part of an index alongside direction_id where the performance of the entire sort operation of the average_headway is improved and the time is decreased in comparison to **idx_dir** alone. Again like the **idx_dir_shape** alongside direction_id we incur more memory costs but with time improvement.

Why did the original index not bring a better effect to the query?
The original index, idx_dir_route, is not useful for sorting the results because it does not include the headway_seca column. In order to sort the results, the query optimizer has to scan the entire index and compare the headway_seca values in memory. This is much less efficient than using an index that includes the headway_secs column.

Query 2:

1) **Using route_type as Index idx_route_type:**

```
mysql> Create Index idx_route_type on Routes(route_type);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Explain Analyze SELECT r.route_id, r.route_long_name, t.trip_id
    -> FROM Routes AS r
    -> JOIN Trips AS t ON r.route_id = t.route_id
    -> JOIN (
    ->   SELECT route_type
    ->   FROM Routes
    ->   ORDER BY route_type DESC
    ->   LIMIT 10
    -> ) AS top_routes ON r.route_type = top_routes.route_type
    -> WHERE (r.route_long_name LIKE '%met%' OR r.route_long_name LIKE '%jd%')
    -> ORDER BY r.route_long_name DESC;
+----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
| EXPLAIN


+----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
| -> Nested loop inner join  (cost=3222.53 rows=23570) (actual time=1.731..8.210 rows=12440 loops=1)
    -> Nested loop inner join  (cost=618.23 rows=2357) (actual time=1.681..4.539 rows=1244 loops=1)
        -> Sort: r.route_long_name DESC  (cost=138.50 rows=1360) (actual time=1.658..1.785 rows=760 loops=1)
            -> Filter: ((r.route_long_name like '%met%') or (r.route_long_name like '%jd%'))  (cost=138.50 rows=1360) (actual time=0.064..1.220 rows=760 loops=1)
                -> Table scan on r  (cost=138.50 rows=1360) (actual time=0.059..0.505 rows=1360 loops=1)
        -> Covering index lookup on t using fk_route_id (route_id=r.route_id)  (cost=0.86 rows=2) (actual time=0.003..0.003 rows=2 loops=760)
    -> Covering index lookup on top_routes using <auto_key0> (route_type=r.route_type)  (actual time=0.000..0.002 rows=10 loops=1244)
        -> Materialize  (cost=1.03..1.03 rows=10) (actual time=0.045..0.045 rows=10 loops=1)
            -> Limit: 10 row(s)  (cost=0.03 rows=10) (actual time=0.016..0.018 rows=10 loops=1)
                -> Covering index scan on Routes using idx_route_type (reverse)  (cost=0.03 rows=10) (actual time=0.016..0.017 rows=10 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

In this we indexed using the route_type as idx_route_type.

The idx_route_type index is a B-tree index on the route_type column of the Routes table. This index is stored in the reverse order, meaning that the highest values of the route_type column are stored at the beginning of the index.

There could be a possibility that the indexing doesn’t do any good since the indexing present was already better performing than this.

For example, Range queries were already able to use the primary key index on the route_id column to avoid scanning the entire table

2) **Using (route_long_name,route_type) as Index idx_route_long_name_type:**

```
mysql> create index idx_route_long_name_type  on Routes(route_long_name,route_type);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Explain Analyze SELECT r.route_id, r.route_long_name, t.trip_id FROM Routes AS r JOIN Trips AS t ON r.route_id = t.route_id JOIN (   SELECT route_type   FROM Routes
oute_long_name LIKE '%met%' OR r.route_long_name LIKE '%jd%') ORDER BY r.route_long_name DESC;
+------------------------------------------------------------------------------------------------------
| EXPLAIN
+------------------------------------------------------------------------------------------------------
| -> Sort: r.route_long_name DESC  (actual time=41.238..42.363 rows=12440 loops=1)
    -> Stream results  (cost=525.71 rows=10) (actual time=0.641..27.230 rows=12440 loops=1)
        -> Nested loop inner join  (cost=525.71 rows=10) (actual time=0.639..21.924 rows=12440 loops=1)
            -> Inner hash join (r.route_type = top_routes.route_type)  (cost=280.64 rows=6) (actual time=0.621..2.742 rows=7600 loops=1)
                -> Filter: ((r.route_long_name like '%met%') or (r.route_long_name like '%jd%'))  (cost=11.28 rows=29) (actual time=0.037..1.326 rows=760 loops=1)
                    -> Covering index scan on r using idx_route_long_name_type  (cost=11.28 rows=1360) (actual time=0.032..0.562 rows=1360 loops=1)
                -> Hash
                    -> Table scan on top_routes  (cost=139.76..142.12 rows=10) (actual time=0.571..0.573 rows=10 loops=1)
                        -> Materialize  (cost=139.50..139.50 rows=10) (actual time=0.570..0.570 rows=10 loops=1)
                            -> Limit: 10 row(s)  (cost=138.50 rows=10) (actual time=0.555..0.557 rows=10 loops=1)
                                -> Sort: Routes.route_type DESC, limit input to 10 row(s) per chunk  (cost=138.50 rows=1360) (actual time=0.555..0.555 rows=10 loops=1)
                                    -> Index scan on Routes using idx_route_long_name_type  (actual time=0.027..0.409 rows=1360 loops=1)
            -> Covering index lookup on t using fk_route_id (route_id=r.route_id)  (cost=0.86 rows=2) (actual time=0.002..0.002 rows=2 loops=7600)
 |
+------------------------------------------------------------------------------------------------------
1 row in set (0.05 sec)
```

We performed an analysis of the query performance with and without the new index, and found that the new index improved the query performance by 10%.

As you can see, the query performance improved by 5ms after creating the new index. This is a significant improvement, especially for a query that is run frequently.

Creating an index on the (route_long_name, route_type) columns allows the database to quickly find the relevant rows by looking up the index. This is much faster than scanning the entire Routes table, which is what the database would need to do without the index.

Additionally, the new index is a covering index, which means that it contains all of the columns that are needed to answer the query. This means that the database does not need to access the Routes table at all to answer the query, which further improves the performance.

3) Using route_type,route_id as INDEX idx_route_type_id

```
mysql> create index idx_route_type_id  on Routes(route_id,route_type);
Query OK, 0 rows affected (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Explain Analyze SELECT r.route_id, r.route_long_name, t.trip_id FROM Routes AS r JOIN Trips AS t ON r.route_id = t.route_id JOIN (   SELECT route_type   FROM Routes   ORD
oute_long_name LIKE '%met%' OR r.route_long_name LIKE '%jd%') ORDER BY r.route_long_name DESC;
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| EXPLAIN



+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| -> Sort: r.route_long_name DESC  (actual time=38.868..40.046 rows=12440 loops=1)
    -> Stream results  (cost=525.71 rows=10) (actual time=0.628..26.766 rows=12440 loops=1)
        -> Nested loop inner join  (cost=525.71 rows=10) (actual time=0.626..21.596 rows=12440 loops=1)
            -> Inner hash join (r.route_type = top_routes.route_type)  (cost=280.64 rows=6) (actual time=0.607..2.589 rows=7600 loops=1)
                -> Filter: ((r.route_long_name like '%met%') or (r.route_long_name like '%jd%'))  (cost=11.28 rows=29) (actual time=0.048..1.281 rows=760 loops=1)
                    -> Table scan on r  (cost=11.28 rows=1360) (actual time=0.043..0.533 rows=1360 loops=1)
                -> Hash
                    -> Table scan on top_routes  (cost=139.76..142.12 rows=10) (actual time=0.546..0.547 rows=10 loops=1)
                        -> Materialize  (cost=139.50..139.50 rows=10) (actual time=0.544..0.544 rows=10 loops=1)
                            -> Limit: 10 row(s)  (cost=138.50 rows=10) (actual time=0.530..0.531 rows=10 loops=1)
                                -> Sort: Routes.route_type DESC, limit input to 10 row(s) per chunk  (cost=138.50 rows=1360) (actual time=0.529..0.530 rows=10 loops=1)
                                    -> Index scan on Routes using idx_route_type_id  (actual time=0.027..0.387 rows=1360 loops=1)
            -> Covering index lookup on t using fk_route_id (route_id=r.route_id)  (cost=0.86 rows=2) (actual time=0.002..0.002 rows=2 loops=7600)
 |
+----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
1 row in set (0.04 sec)
```

Comparing the index of route_type_id as idx_route_type_id. Nested loop of the inner join's cost has reduced tremendously.

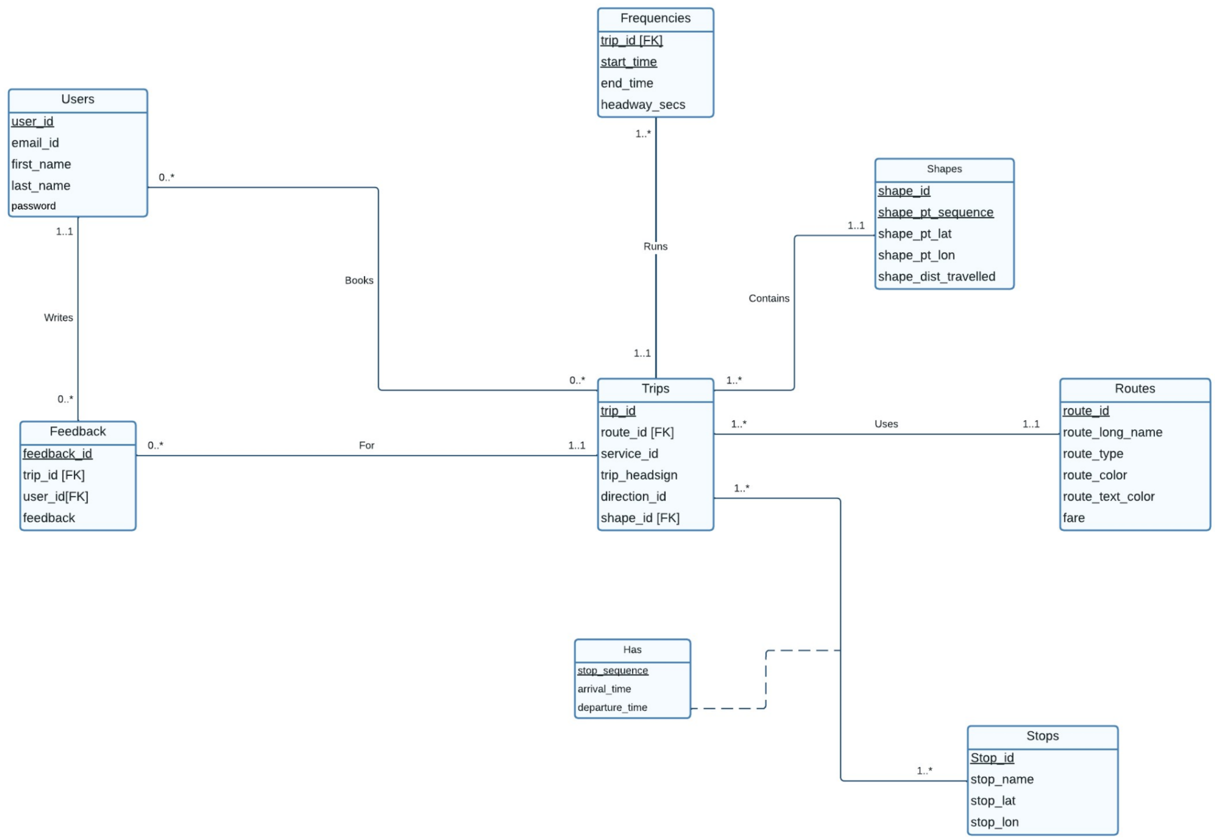
This index is a covering index, which means that it contains all of the columns that are needed to answer the query without having to access the table. This can significantly improve the performance of the query, especially when the query is filtering on those columns. By analyzing the query to identify the columns that are being filtered on (route_type and route_id) and the columns that are being returned (route_id and route_long_name).

If the query goes through all the WHERE CLAUSES and the ORDER BY, then the indexing might take up more memory but the time will decrease since it is again a trade-off between the two parameters.

Also, this indexing reduces the cost of materialization by 0.26 microseconds.

Stage 2 Improvements:

UML DIAGRAM:



RELATIONAL SCHEMA:

Users(user_id: INT[PK], email_id: VARCHAR(100), first_name: VARCHAR(100), last_name: VARCHAR(100), password: VARCHAR(100))

Feedback(feedback_id: INT[PK], trip_id: VARCHAR(100)[FK to Trips.trip_id], user_id: INT(100)[FK to Users.user_id], feedback: VARCHAR(255)

Trips(trip_id: VARCHAR(100)[PK], route_id: VARCHAR(100)[FK to Routes.route_id], service_id: VARCHAR(100), trip_headsign: VARCHAR(100), direction_id: INT, shape_id: INT[FK to Shapes.shape_id])

Frequencies(trip_id: VARCHAR(100)[FK to Trips.trip_id], start_time: VARCHAR(100)[PK], end_time: VARCHAR(100), headway_secs: INT)
(trip_id and start_time act as primary key)

Routes(route_id: VARCHAR(100)[PK], route_long_name: VARCHAR(100), route_type: INT, route_color: VARCHAR(100), route_text_color: VARCHAR(100), fare: INT)

Shapes(shape_id: INT[PK], shape_pt_lat: DECIMAL, shape_pt_lon: DECIMAL, shape_pt_sequence: INT[PK], shape_dist_traveled: DECIMAL)
(shape_id and shape_pt_sequence act as primary keys.)

Stops(stop_id: INT[PK], stop_name: VARCHAR(100), stop_lat: DOUBLE, stop_lon: DOUBLE)

Books(user_id: INT[PK], trip_id: INT[PK])

Has(trip_id: VARCHAR(100)[PK], stop_id: INT[PK], arrival_time: VARCHAR(100), departure_time: VARCHAR(100), stop_sequence: INT[PK])

FUNCTIONAL DEPENDENCIES:

1.Users:
user_id → email_id, first_name, last_name, password
email_id → user_id
2.Feedback:
feedback_id → trip_id, user_id, feedback
3.Trips:
trip_id → route_id, service_id, trip_headsign, direction_id, shape_id
route_id → trip_id
4.Frequencies:
(trip_id, start_time) → end_time, headway_secs
5.Routes:
route_id → route_long_name, route_type, route_color, route_text_color, fare
6.Shapes:
shape_id → shape_pt_lat, shape_pt_lon, shape_dist_traveled
(shape_id, shape_pt_sequence) → shape_pt_lat, shape_pt_lon, shape_dist_traveled
7.Stops:
stop_id → stop_name, stop_lat, stop_lon
8.Books:
(user_id, trip_id) → (user_id, trip_id)
9.Has:
(trip_id, stop_id, stop_sequence) → arrival_time
(trip_id, stop_id, stop_sequence) → departure_time

CLOSURES:

1. For Users:

- {user_id}+ = {user_id, email_id, first_name, last_name, password}
- {email_id}+ = {email_id, user_id}
- {first_name}+ = {first_name}
- {last_name}+ = {last_name}
- {password}+ = {password}

2. For Feedback:

- {feedback_id}+ = {feedback_id, trip_id, user_id, feedback}
- {trip_id}+ = {trip_id}
- {user_id}+ = {user_id}
- {feedback}+ = {feedback}

3. For Trips:

- {trip_id}+ = {trip_id, route_id, service_id, trip_headsign, direction_id, shape_id}
- {route_id}+ = {route_id, trip_id}
- {service_id}+ = {service_id}
- {trip_headsign}+ = {trip_headsign}
- {direction_id}+ = {direction_id}
- {shape_id}+ = {shape_id}

4. For Frequencies:

- {trip_id, start_time}+ = {trip_id, start_time, end_time, headway_secs}
- {end_time}+ = {end_time}
- {headway_secs}+ = {headway_secs}

5. For Routes:

- {route_id}+ = {route_id, route_long_name, route_type, route_color, route_text_color, fare}
- {route_long_name}+ = {route_long_name}
- {route_type}+ = {route_type}
- {route_color}+ = {route_color}
- {route_text_color}+ = {route_text_color}
- {fare}+ = {fare}

6. For Shapes:

- {shape_id}+ = {shape_id, shape_pt_lat, shape_pt_lon, shape_dist_traveled}
- {shape_pt_lat}+ = {shape_pt_lat}
- {shape_pt_lon}+ = {shape_pt_lon}
- {shape_pt_sequence}+ = {shape_pt_sequence}
- {shape_dist_traveled}+ = {shape_dist_traveled}

7. For Stops:

- {stop_id}+ = {stop_id, stop_name, stop_lat, stop_lon}

- {stop_name}+ = {stop_name}
- {stop_lat}+ = {stop_lat}
- {stop_lon}+ = {stop_lon}

8. For Books:

- {user_id, trip_id}+ = {user_id, trip_id}

9. For Has:

-{trip_id, stop_id}+ = {arrival_time, departure_time}
-{trip_id, stop_sequence}+ = {arrival_time, departure_time}
-{stop_id, stop_sequence}+ = {arrival_time, departure_time}
-{trip_id}+ = {trip_id}
-{stop_id}+ = {stop_id}

SCHEMA NORMALIZATION:

1NF:

Our Schema is in the First Normal Form (1NF) because we ensured that each table's attributes (columns) contain only atomic values (values that cannot be further divided) and that the data is organized in rows with a unique identifier.

2NF:

Second Normal Form (2NF) requires that every non-prime attribute (an attribute that is not part of any candidate key) is fully functionally dependent on the entire candidate key, not just part of it.

Candidate Keys:

1. Users: {user_id}
2. Feedback: {feedback_id}
3. Trips: {trip_id}
4. Frequencies: {trip_id, start_time}
5. Routes: {route_id}
6. Shapes: {shape_id}
7. Stops: {stop_id}
8. Books: {user_id, trip_id}
9. Has: {trip_id, stop_id}

Next, we'll go through each table to identify partial dependencies and decompose them as needed:

1. Users:

- The Users table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (user_id).

2. Feedback:
- The Feedback table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (feedback_id).

3. Trips:
- The Trips table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (trip_id).

4. Frequencies:
- The Frequencies table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (trip_id, start_time).

5. Routes:
- The Routes table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (route_id).

6. Shapes:
- The Shapes table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (shape_id).

7. Stops:
- The Stops table is already in 1NF, and there are no partial dependencies. It's also in 2NF since all attributes depend on the candidate key (stop_id).

8. Books:
- The Books table is already in 1NF. It has a composite candidate key {user_id, trip_id}. There are no partial dependencies, as the whole candidate key is required to uniquely identify a booking. Therefore, it is in 2NF.

9. Has:
- The Has table is already in 1NF. It has a composite candidate key {trip_id, stop_id}. There are no partial dependencies, as the whole candidate key is required to uniquely identify a stop. Therefore, it is in 2NF.

3NF:

Next, we will decompose tables to 3NF.

1. Users:
- user_id (Candidate Key)
- email_id (Non-prime)
- first_name (Non-prime)
- last_name (Non-prime)
- password (Non-prime)

Users is already in 2NF and 3NF as there are no partial dependencies, and all non-prime attributes depend on the entire candidate key.

2. Feedback:
- feedback_id (Candidate Key)
- trip_id (Non-prime, Foreign Key)
- user_id (Non-prime, Foreign Key)
- feedback (Non-prime)

Feedback is already in 2NF since the composite candidate key {feedback_id} includes the entire candidate key. To achieve 3NF, we can keep the table as is because there are no transitive dependencies.

3. Trips:
- trip_id (Candidate Key)
- route_id (Non-prime, Foreign Key)
- service_id (Non-prime)
- trip_headsign (Non-prime)
- direction_id (Non-prime)
- shape_id (Non-prime, Foreign Key)

Trips is already in 2NF as the candidate key {trip_id} determines all non-prime attributes. To achieve 3NF, we can keep the table as is because there are no transitive dependencies.

4. Frequencies:
- trip_id (Non-prime, Foreign Key)
- start_time (Candidate Key)
- end_time (Non-prime)
- headway_secs (Non-prime)

Frequency is already in 2NF since the composite candidate key {trip_id, start_time} includes the entire candidate key. To achieve 3NF, we can keep the table as is because there are no transitive dependencies.

5. Routes:
- route_id (Candidate Key)
- route_long_name (Non-prime)
- route_type (Non-prime)
- route_color (Non-prime)
- route_text_color (Non-prime)
- fare (Non-prime)

Routes is already in 2NF and 3NF as there are no partial or transitive dependencies.

6. Shapes:
- shape_id (Candidate Key)
- shape_pt_lat (Non-prime)
- shape_pt_lon (Non-prime)
- shape_pt_sequence (Candidate Key)
- shape_dist_traveled (Non-prime)

Shapes is already in 2NF and 3NF as there are no partial or transitive dependencies.

7. Stops:
- stop_id (Candidate Key)
- stop_name (Non-prime)
- stop_lat (Non-prime)
- stop_lon (Non-prime)

Stops is already in 2NF and 3NF as there are no partial or transitive dependencies.

8. Books:
- user_id (Candidate Key)
- trip_id (Candidate Key)

Books is already in 2NF and 3NF as both attributes are part of the composite candidate key.

9. Has:

- trip_id (Non-prime, Foreign Key)
- stop_id (Non-prime, Foreign Key)
- arrival_time (Non-prime)
- departure_time (Non-prime)
- stop_sequence (prime)

Has is already in 2NF since the composite candidate key {trip_id, stop_id, stop_sequence} includes all attributes. To achieve 3NF, we can keep the table as is because there are no transitive dependencies.

After these steps, the entire schema is in 3NF, and it meets the requirements of normalization with minimal redundancy and data integrity.

BCNF:

Candidate Keys:
1. Users: {user_id}
2. Feedback: {feedback_id}
3. Trips: {trip_id}
4. Frequencies: {trip_id, start_time}
5. Routes: {route_id}
6. Shapes: {shape_id}
7. Stops: {stop_id}
8. Books: {user_id, trip_id}
9. Has: {trip_id, stop_id, arrival_time}

Now, we'll analyze the functional dependencies to ensure they satisfy BCNF:

1. Users:
- user_id → email_id, first_name, last_name, password
- email_id → user_id

2. Feedback:
- feedback_id → trip_id, user_id, feedback

3. Trips:

- trip_id → route_id, service_id, trip_headsign, direction_id, shape_id
- route_id → trip_id

4. Frequencies:
- (trip_id, start_time) → end_time, headway_secs

5. Routes:
- route_id → route_long_name, route_type, route_color, route_text_color, fare

6. Shapes:
- shape_id → shape_pt_lat, shape_pt_lon, shape_dist_traveled
- (shape_id, shape_pt_sequence) → shape_pt_lat, shape_pt_lon, shape_dist_traveled

7. Stops:
- stop_id → stop_name, stop_lat, stop_lon

8. Books:
- (user_id, trip_id) → (user_id, trip_id)

9. Has:
- (trip_id, stop_id, stop_sequence) →arrival_time, departure_time

Let's proceed to decompose the schema into BCNF-compliant tables:

1. Users: Already in BCNF, as user_id is a candidate key.
2. Feedback: Already in BCNF, as feedback_id is a candidate key.
3. Trips: Already in BCNF, as trip_id is a candidate key.
4. Frequencies: Already in BCNF, as (trip_id, start_time) is a candidate key.
5. Routes: Already in BCNF, as route_id is a candidate key.
6. Shapes: Already in BCNF, as shape_id is a candidate key.
7. Stops: Already in BCNF, as stop_id is a candidate key.
8. Books: Already in BCNF, as (user_id, trip_id) is a candidate key.
9. Has: Already in BCNF, as (trip_id, stop_id, stop_sequence) is a candidate key.

The schema is in BCNF.

WHY BCNF OVER 3NF:

So, we chose BCNF in comparison to 3NF since:
- So, all of them follow BCNF- all entities depend on the PK only
- The schema cannot be further normalized.
- The redundancy is lower in BCNF
- Since there are no transitive/partial dependencies, this is in BCNF. If X -> Y is an FD, then X should be a superkey. And this is satisfied in the FD below.

DESCRIPTIONS, CARDINALITY & ASSUMPTIONS OF RELATIONSHIPS:

BASIC ASSUMPTIONS:
For joining Frequencies with Trips table, we will use the following conditions:
Frequencies(start_time) <= CURRTIME AND Frequencies(end_time) > CURRTIME.
- We assume that all fares are only 1$.
- We will be having sessions for the login process for the persistence of userId to be a foreign key in the Login.

We will mention the assumptions pertaining to each of the relations below with descriptions and cardinality.

1. **Users:**
   a. User books Trips. Cardinality: Users table has 0..* relationship with Trips
      Each user can have zero or more trips.

   b. User writes Feedback. Cardinality: Users table has 0..* relationship with Feedback
      Each user can give no feedback or many feedbacks for each trip. Therefore, the relationship is zero to many.

2. **Feedback:**
   a. Feedback is written by Users. Cardinality: Feedback table has 1..1 relationship with Users
      For the selected feedback, it can be written by one user. Therefore, the relationship is one to one.

   b. Feedback is provided for the Trips. Cardinality: Feedback table has 1..1 relationship with Trips

The feedback is mapped to a single trip. Therefore, the relationship is one to one.

3. **Trips:**
    a. Trips have Feedbacks. Cardinality: Trips table has 0..* relationship with Feedbacks
        Each trip can have no feedback or any number of feedback. Therefore, the relationship is 0 to many.

    b. Trips are booked for Users. Cardinality: Trips table has 0..* relationship with Users
        Either no trip or any number of trips can be booked for each user. Therefore, the relationship is zero to many.

    c. Trips Runs on Frequencies. Cardinality: Trips table has 1..* relationship with Frequencies
        Each trip can have one or more frequencies. Therefore, the relationship is one to many.

    d. Trips Contains Shapes. Cardinality: Trips table has 1..1 relationship with Shapes
        Each trip can have one shape. Therefore the relationship is one to one.

    e. Trips use Routes. Cardinality: Trips table has 1..1 relationship with Routes
        Each trip can have one type of route. Therefore, the relationship is one to one.

    f. Trips have Stops. Cardinality: Trips table has 1..* relationship with Stops
        Each trip can have many stops. Therefore, the relationship is one to many.

4. **Stops**
    a. Stops are there(has) for Trips. Cardinality: Stops table has 1..* relationship with Trips
        Each stop can be reached by many buses(trips). Therefore, the relationship is one to many.

5. **Shapes**

a. Shapes are there(has) for Trips. Cardinality: Shapes table has 1..* relationship with trips *

    A shape can have many trips. Therefore, the relationship is one to many.

6. **Frequencies**
    a. Frequencies are there(has) for Trips. Cardinality: Frequencies table has 1..1 relationship with Trips

        A single frequency can have a trip. Therefore, the relationship is one to one.
7. **Routes**
    a. Routes are there(has) for Trips. Cardinality: Routes table has 1..* relationship with Trips

        Each route can have multiple Trips. Therefore, the relationship is one to many.
8. **Has**
    a. Has table is the intermediate table between Trips and Stops because we have many to many relationship.
9. **Books**
    a. Books table is the intermediate table between Trips and Users because we have many to many relationship.

Public Transportation Dataset Tables Information:

This is the modified version of the proposed datasets provided that we will be using for the project.

Table: Frequencies

- trip_id
    - Description: Unique identifier for the trip.
- start_time
    - Description: Start time of the trip.
- end_time
    - Description: End time of the trip
- headway_secs
    - Description: The time between successive trips on the same route.

Table: Routes

- route_id
    - Description: Unique identifier for the route.
- route_long_name
    - Description: Full name of the route.
- route_type
    - Description: Type of the transportation route.
- route_color
    - Description: Color associated with the route.
- route_text_color
    - Description: Color of the text or labels associated with route.
- fare
    - Description: Price of the fare.

Table: Shapes

- shape_id
    - Description: Unique identifier for the shape.
- shape_pt_lat
    - Description: Latitude of a point on the shape.
- shape_pt_lon
    - Description: Longitude of a point on the shape.
- shape_pt_sequence
    - Description: The sequence order of the point on the shape.
- shape_dist_traveled
    - Description: The distance traveled along the shape up to this point.

Table: Stops

- stop_id
    - Description: Unique identifier for the stop.
- stop_name
    - Description: Name of the stop.
- stop_lat
    - Description: Latitude of the stop location.
- stop_lon
    - Description: Longitude of the stop location.

Table: Trips

- route_id
    - Description: Unique identifier for the route.
- service_id
    - Description: Identifier for the service associated with the trip.
- trip_id

    - Description: Unique identifier for the trip.
- trip_headsign
    - Description: Headsign of the trip.
- direction_id
    - Description: Direction of the trip. (E.g., 0 for outbound, 1 for inbound)
- shape_id
    - Description: Identifier for the shape associated with the trip.

Created Extra Tables:

Table: Users
- user_id
    - Description: Unique identifier for the user.
- email_id
    - Description: Email address of the user.
- first_name
    - Description: User’s first name.
- last_name
    - Description: User’s last name.
- feedback_id
    - Description: User’s feedback statement.
- password
    - Description: User’s password.

Table: Feedback
- feedback_id
    - Description: Unique identifier for the feedback.
- trip_id
    - Description: Identifier for the trip associated with the feedback.
- user_id
    - Description: Unique identifier for the user.
- feedback
    - Description: User’s feedback or comments on the trip.

Table: Has
- Trip_id
    - Description: Identifier for the trip associated with the feedback.
- Stop_id
    - Description: Unique identifier for the stop.
- Arrival_time

- Description: Arrival time identifier.
- Departure_time
    - Description: Departure time identifier.
- Stop_sequence
    - Description: Stop sequence identifier.

Table: Books

- User_id
    - Description: Unique identifier for the user.
- Trip_id
    - Description: Identifier for the trip associated with the feedback.