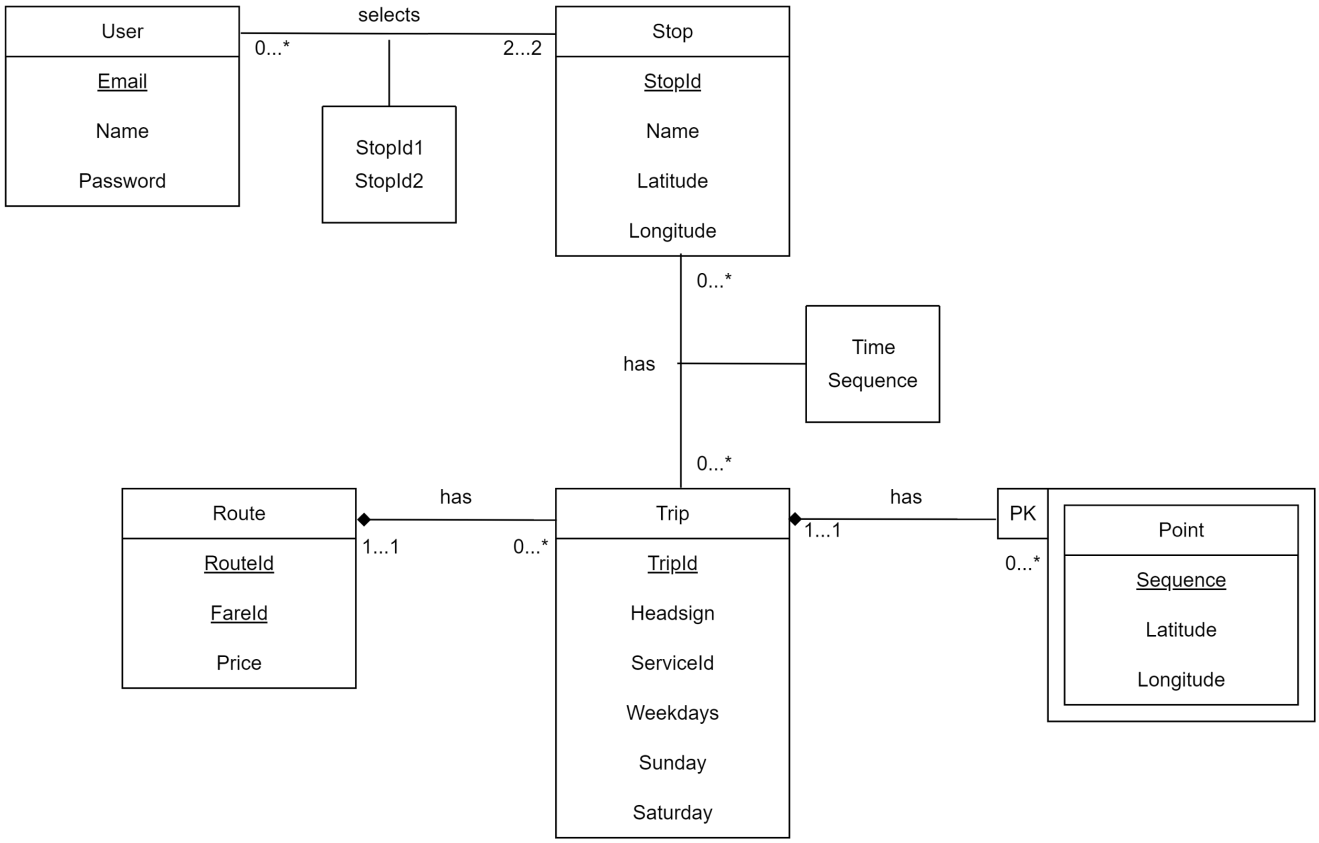
Part 1:

Tables & DDL commands:



User (Email, Name, Password)

Stop (StopId, Name, Latitude, Longitude)

StopsPicked (Email, StopId1, StopId2)

Trip (TripId, RouteId, Headsign, ServiceId, Weekdays, Sunday, Saturday)

Point (TripId, Sequence, Latitude, Longitude)

StopsToTrips (StopId, TripId, Sequence, Time)

Route (RouteId, FareId, Price)

```
CREATE TABLE  `411-project` . `User`(
    Email VARCHAR(32) NOT NULL,
    Password VARCHAR(32) NOT NULL,
    Name VARCHAR(32),
```

```
    PRIMARY KEY (`Email`)
    );

CREATE TABLE `411-project` . `Stop` AS (
    SELECT s.stop_id AS `StopId`, s.stop_name AS `Name`, s.stop_lat AS `Latitude`, s.stop_lon AS `Longitude`
    FROM `411-project`.`input_stops` as s
);

CREATE TABLE `411-project` . `Stop` (
    StopId INT NOT NULL,
    Name VARCHAR(100),
    Latitude REAL,
    Longitude REAL,
    PRIMARY KEY (`StopId`)
    );

CREATE TABLE `411-project` . `StopsPicked`(
    Email VARCHAR(32) NOT NULL,
    StopId1 INT NOT NULL,
    StopId2 INT NOT NULL,
    PRIMARY KEY (`StopId1`, `StopId2`, `Email`),
    FOREIGN KEY (`Email`) REFERENCES `User` (`Email`) ON DELETE CASCADE,
    FOREIGN KEY (`StopId1`) REFERENCES `Stop` (`StopId`) ON DELETE CASCADE
    );

CREATE TABLE `411-project` . `Trip` AS (
    SELECT t.trip_id AS `TripId`, t.route_id AS `RouteId`, t.trip_headsign AS `Headsign`, t.service_id AS `ServiceId`, c.weekdays AS `Weekdays`, c.saturday AS `Saturday`, c.sunday AS `Sunday`
```

```
    FROM `411-project` . `input_trips` AS  t JOIN `411-project` . `input_calendar` AS c ON t.service_id = c.service_id
);

CREATE TABLE  `411-project` . `Trip` (
    TripId VARCHAR(32) NOT NULL,
    RouteId VARCHAR(32) NOT NULL,
    Headsign VARCHAR(100),
    ServiceId VARCHAR(32) NOT NULL,
    Weekdays BIT,
    Saturday BIT,
    Sunday BIT,
    PRIMARY KEY(`TripId`),
    FOREIGN KEY (`RouteId`) REFERENCES `Route` (`RouteId`)
    );


CREATE TABLE `411-project` . `Point` AS (
    SELECT t.trip_id AS `TripId`, s.shape_pt_sequence AS `Sequence`, s.shape_pt_lat AS `Latitude`, s.shape_pt_lon AS `Longitude`
    FROM `411-project` . `input_shapes` AS s JOIN `411-project` . `input_trips` as t ON s.shape_id = t.shape_id
);

CREATE TABLE `411-project` . `Point` (
    TripId VARCHAR(32) NOT NULL,
    Sequence INT NOT NULL,
    Latitude REAL,
    Longitude REAL,
    PRIMARY KEY ( `TripId`, `Sequence`),
    FOREIGN KEY (`TripId`) REFERENCES `Trip` (`TripId`) ON DELETE CASCADE
```

```
    );

CREATE TABLE `411-project` . `StopsToTrips` AS (
    SELECT s.stop_id AS `StopId`, s.trip_id AS `TripId`, s.stop_sequence AS `Sequence`,
s.arrival_time AS `Time`
    FROM `411-project` . `input_stop_times` AS s
);



CREATE TABLE  `411-project` . `StopsToTrips` (
    StopId VARCHAR(32) NOT NULL,
    TripId VARCHAR(32) NOT NULL,
    Sequence INT NOT NULL,
    Time TIME,
    PRIMARY KEY (`StopId`, `TripId`, `Sequence`),
    FOREIGN KEY (`StopId`) REFERENCES `Stop` (`StopId`) ON DELETE CASCADE,
    FOREIGN KEY (`TripId`) REFERENCES `Trip` (`TripId`) ON DELETE CASCADE
    );

CREATE TABLE `411-project` . `Route` AS (
    SELECT r.route_id AS `RouteId`, fr.fare_id AS `FareId`, fa.price AS `Price`
    FROM (`411-project` . `input_routes` as r LEFT OUTER JOIN `411-project` .
`input_fare_rules` as fr ON r.route_id = fr.route_id) JOIN `411-project` . `input_fare_attr` as
fa ON fr.fare_id = fa.fare_id
);

CREATE TABLE  `411-project` . `Route`(
    RouteId VARCHAR(32) NOT NULL,
    FareId VARCHAR(32),
    Price REAL,
    PRIMARY KEY (`RouteId`, `FareId`)
```

);

```
mysql> select Count(*)  from `411-project` . `Point`;
+----------+
| Count(*) |
+----------+
|  1128783 |
+----------+
1 row in set (0.19 sec)

mysql> select Count(*)  from `411-project` . `Stop`;
+----------+
| Count(*) |
+----------+
|    20902 |
+----------+
1 row in set (0.01 sec)

mysql> select Count(*)  from `411-project` . `Trip`;
+----------+
| Count(*) |
+----------+
|     2227 |
+----------+
1 row in set (0.00 sec)

mysql> select Count(*)  from `411-project` . `Route`;
+----------+
| Count(*) |
+----------+
|     5427 |
+----------+
1 row in set (0.00 sec)
```

1. Return: For each Trip Headsign, the number of distinct TripIDs, Stop.Name and the Price, ordered from cheapest to most expensive. Will utilize join and aggregation (Group By Headsign).

This is relevant to our app because let's say the user has a hotel near point A. If their trip starts there, they can find all the different trips that start at A and their average prices. The user can decide which trip they want to take and see the average amount of money they need to pay.

```
SELECT Trip.Headsign, COUNT(DISTINCT Trip.TripId), AVG(Route.Price) AS avgRoutePrice
FROM `411-project` . `Trip`
JOIN `411-project` . `StopsToTrips` ON StopsToTrips.TripId = Trip.TripId
JOIN `411-project` . `Stop` ON StopsToTrips.StopId = Stop.StopId
JOIN `411-project` . `Route` ON Trip.RouteId = Route.RouteId
GROUP BY Trip.Headsign
ORDER BY avgRoutePrice ASC
LIMIT 15
```

```
mysql> SELECT Trip.Headsign, COUNT(DISTINCT Trip.TripId), AVG(Route.Price) as avgRoutePrice
    -> FROM `411-project` . `Trip`
    -> Join `411-project` . `StopsToTrips` on StopsToTrips.TripId = Trip.TripId
    -> join `411-project` . `Stop` on StopsToTrips.StopId = Stop.StopId
    -> Join `411-project` . `Route` on Trip.RouteId = Route.RouteId
    -> GROUP BY Trip.Headsign
    -> ORDER BY avgRoutePrice ASC
    -> LIMIT 15
    -> ;
+-------------------------+----------------------------+--------------------+
| Headsign                | COUNT(DISTINCT Trip.TripId) | avgRoutePrice      |
+-------------------------+----------------------------+--------------------+
| Term. Pq. D. Pedro Ii   |                         78 |  7.937499999998954 |
| MetrÃ´ Ana Rosa         |                         11 |  7.937499999999721 |
| PÃ§a. Da SÃ©           |                          9 |  7.937499999999725 |
| MetrÃ´ Vl. Mariana      |                         11 |  7.937499999999728 |
| Term. Sapopemba         |                         10 |  7.937499999999729 |
| Pq. Ibirapuera          |                          6 |  7.937499999999736 |
| Term. Cid. Tiradentes   |                          9 |  7.937499999999743 |
| MetrÃ´ ConceiÃ§Ã£o      |                         11 |  7.937499999999744 |
| Cptm Guaianazes         |                         18 |  7.937499999999754 |
| Jd. Selma               |                          8 |  7.937499999999754 |
| Pq. Edu Chaves          |                          9 |  7.937499999999757 |
| SÃ£o Miguel             |                          9 | 7.9374999999997655 |
| Jd. Ã¢Ngela             |                         11 |  7.937499999999767 |
| Shop. Center Norte      |                          8 | 7.9374999999997815 |
| Itaim Bibi              |                          7 | 7.9374999999997895 |
+-------------------------+----------------------------+--------------------+
```

2.
Return: The query outputs TripId, ServiceId, and the count of distinct visit times (NumTimes) for trips that include the User specified stops (in this example, StopId 18848 and StopId 18847) within the specified time range.

This is relevant to our app because it empowers users to better understand their trips by providing information on the frequency of stops during their journeys. This knowledge is essential for route planning and optimizing travel experiences. While the current query uses fixed stop values for illustration, our future plan involves populating the "StopsToTrips" table based on user interactions to offer more personalized and dynamic results.

```
(SELECT stt.TripId, t.ServiceId, COUNT( distinct stt.Time) as NumTimes
FROM `Stop` s JOIN
        (select Stot.TripId as TripId, Stot.Time as Time, Stot.StopId as StopId, Stot.Sequence as Sequence from  `StopsToTrips` Stot where Stot.StopId = 18848 and (TIME_TO_SEC(Stot.Time) > TIME_TO_SEC('0:00:00') and TIME_TO_SEC(Stot.Time) < TIME_TO_SEC('23:59:59'))) as stt
        ON s.StopId = stt.StopId JOIN `Trip` t
        ON t.TripId = stt.TripId
GROUP BY stt.TripId, t.ServiceId
ORDER BY stt.Sequence ASC)
UNION
(SELECT stt.TripId, t.ServiceId, COUNT( distinct stt.Time) as NumTimes
FROM `Stop` s JOIN
        (select Stot.TripId as TripId,  Stot.Time as Time, Stot.StopId as StopId, Stot.Sequence as Sequence from  `StopsToTrips` Stot where Stot.StopId = 18847 and (TIME_TO_SEC(Stot.Time) > TIME_TO_SEC('0:00:00') and TIME_TO_SEC(Stot.Time) < TIME_TO_SEC('23:59:59'))) as stt
        ON s.StopId = stt.StopId JOIN `Trip` t
        ON t.TripId = stt.TripId
GROUP BY stt.TripId, t.ServiceId
ORDER BY stt.Sequence ASC)
LIMIT 15;
```

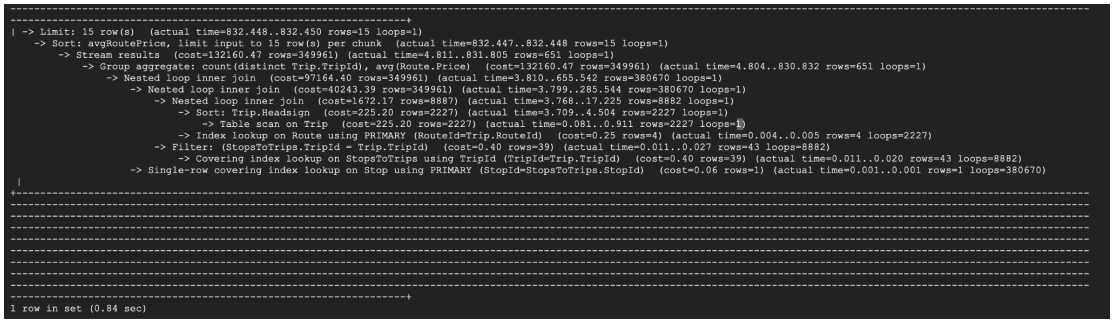
```
+-----------------+-----------+----------+
| TripId          | ServiceId | NumTimes |
+-----------------+-----------+----------+
| METRÃ” L2-0     | USD       |        1 |
| METRÃ” L2-1     | USD       |        1 |
| 4735-10-0       | US_       |        2 |
+-----------------+-----------+----------+
3 rows in set (0.00 sec)
```

5. Execute your advanced SQL queries and provide a screenshot of the top 15 rows of each query result (you can use the LIMIT clause to select the top 15 rows). If your output is less than 15 rows, say that in your output.

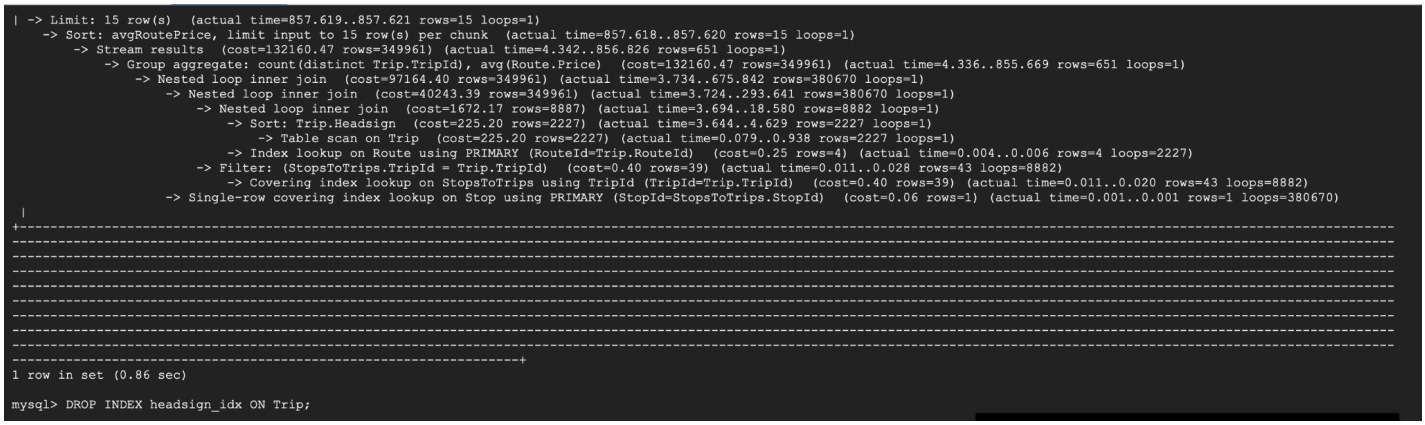
Part 2 Indexing: As a team, for each advanced query:

1. Use the EXPLAIN ANALYZE command to measure your advanced query performance before adding indexes.

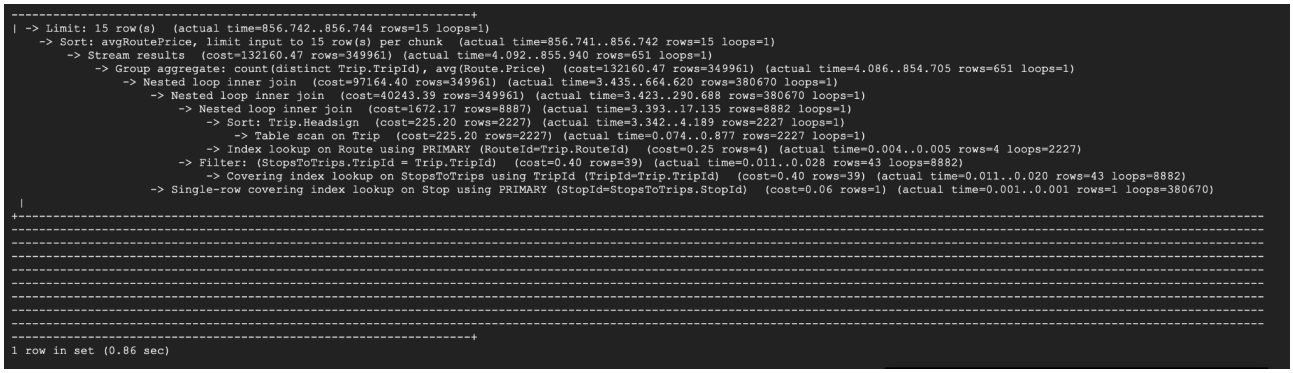
1st Query



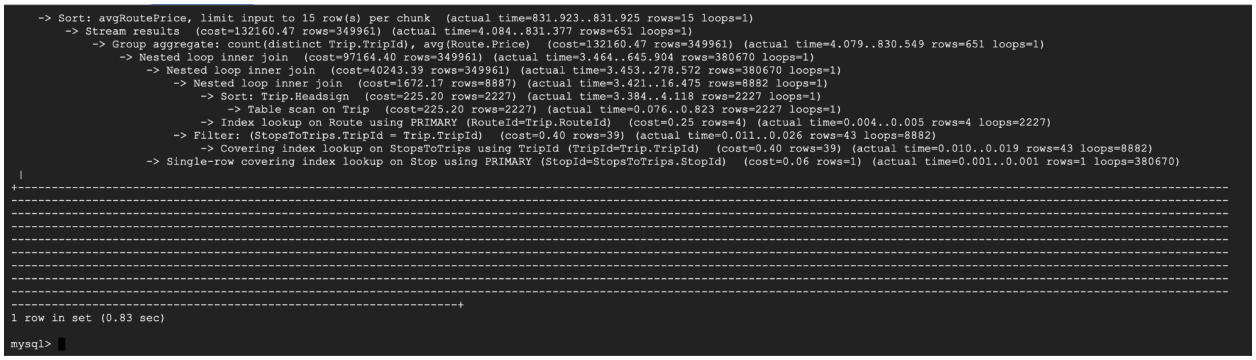
- The cost of running the first SQL query is 225.20. There are a total of 2227 rows that were examined by the operation. It took about 0.081 seconds to read the first row and took 0.911 seconds to read the entire 2227 rows. We've gone over the data 1 time to perform this operation. It takes about 0.84 seconds to perform the entire query.



- Here we are inserting an index on Headsign Attribute which is in the Trip table. There wasn't any significant improvement. The cost to run the query was the same. It took about 0.049 seconds to read the first row and took 0.938 seconds to read the entire 2227 rows. The reason for why there isn't an improvement is because Group By still needs to scan the entire table which is why cost doesn't change.



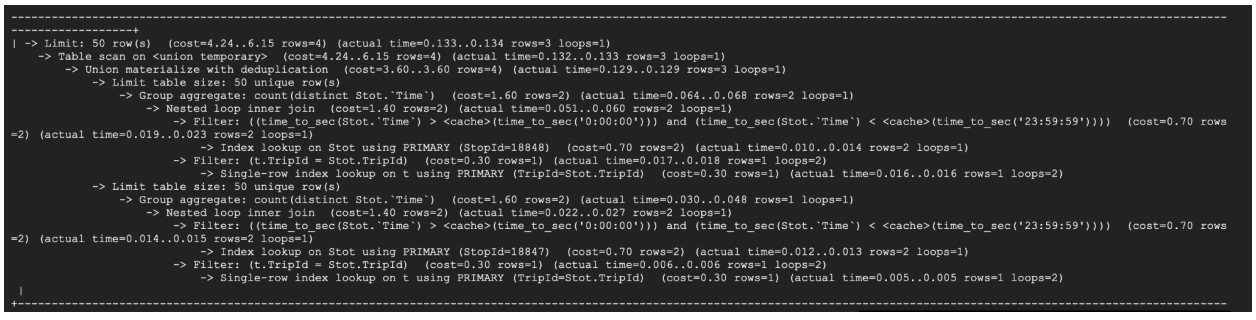
- Here we inserted an index on the Price attribute There wasn't any significant improvement either. The cost to run the query was the same. It took about 0.074 seconds to read the first row and took 0.877 seconds to read the entire 2227 rows. The reason for why there isn't an improvement is because running Average on price is similar to running sum on price where every entry in the table needs to be checked, then dividing by number of rows.



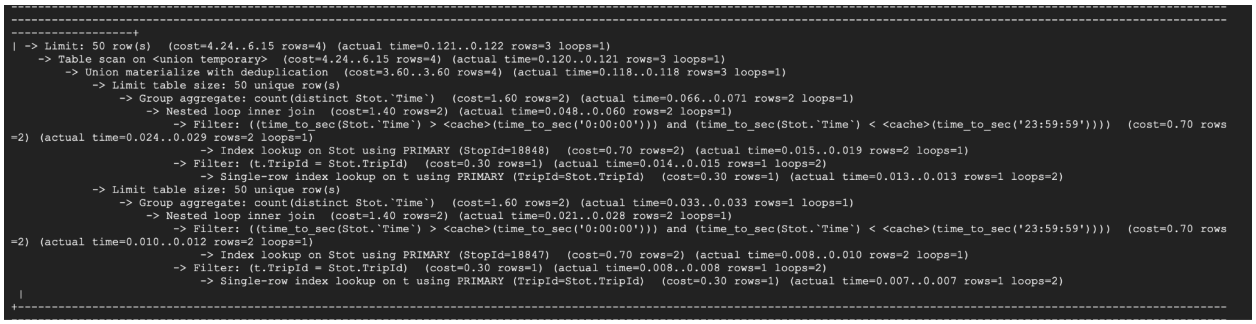
- This time we added indices for both Headsign and Price. There wasn't any significant improvement either. The cost to run the query was the same as well. It took about 0.076 seconds to read the first row and took 0.877 seconds to read the entire 2227 rows. The reason for why there isn't an improvement is because of the same reasons for the previous 2 index analysis.

Query#2

Before

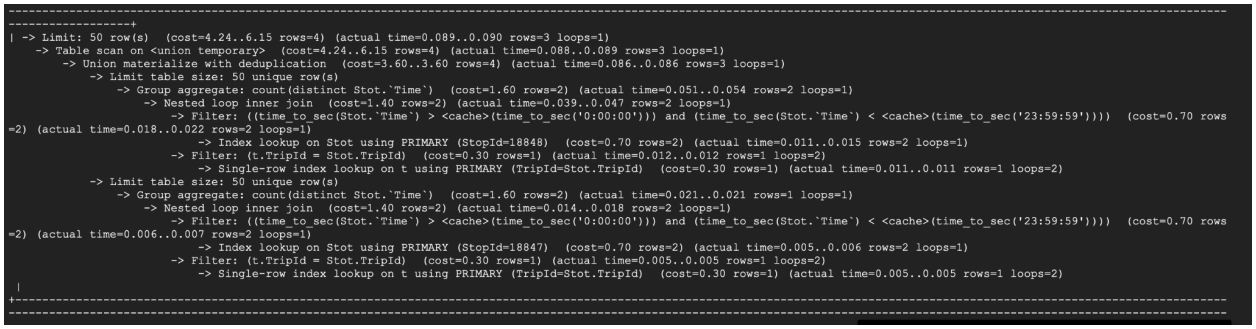


After adding index on Time



- There doesn't seem to be an improvement after placing an index on time since in one query the time it took to read a single row and both rows seem to have gone up. (0.019 -> 0.024, 0.023 -> 0.029), while for the other query the time it took to read a single row and both rows seem to have gone down (0.014 -> 0.01, 0.015 -> 0.012). Thus this indicates that this index yields no improvement. Additionally, the cost to perform this advanced query remains the same at 0.7.

After adding index on Service ID



- There doesn't seem to be any improvement in the query performance. Since we're grouping by both ServiceID and the primary key for the trip which is TripId. Trip ID has precedence over ServiceID.

After adding index on Time and ServiceID

```
-----------------+
| -> Limit: 50 row(s)  (cost=4.24..6.15 rows=4) (actual time=0.095..0.096 rows=3 loops=1)
    -> Table scan on <union temporary>  (cost=4.24..6.15 rows=4) (actual time=0.094..0.095 rows=3 loops=1)
        -> Union materialize with deduplication  (cost=3.60..3.60 rows=4) (actual time=0.093..0.093 rows=3 loops=1)
            -> Limit table size: 50 unique row(s)
                -> Group aggregate: count(distinct Stot.`Time`)  (cost=1.60 rows=2) (actual time=0.058..0.060 rows=2 loops=1)
                    -> Nested loop inner join  (cost=1.40 rows=2) (actual time=0.045..0.053 rows=2 loops=1)
                        -> Filter: ((time_to_sec(Stot.`Time`) > <cache>(time_to_sec('0:00:00'))) and (time_to_sec(Stot.`Time`) < <cache>(time_to_sec('23:59:59'))))  (cost=0.70 rows
=2) (actual time=0.020..0.023 rows=2 loops=1)
                            -> Index lookup on Stot using PRIMARY (StopId=18848)  (cost=0.70 rows=2) (actual time=0.012..0.015 rows=2 loops=1)
                        -> Filter: (t.TripId = Stot.TripId)  (cost=0.30 rows=1) (actual time=0.014..0.014 rows=1 loops=2)
                            -> Single-row index lookup on t using PRIMARY (TripId=Stot.TripId)  (cost=0.30 rows=1) (actual time=0.010..0.010 rows=1 loops=2)
            -> Limit table size: 50 unique row(s)
                -> Group aggregate: count(distinct Stot.`Time`)  (cost=1.60 rows=2) (actual time=0.021..0.021 rows=1 loops=1)
                    -> Nested loop inner join  (cost=1.40 rows=2) (actual time=0.013..0.018 rows=2 loops=1)
                        -> Filter: ((time_to_sec(Stot.`Time`) > <cache>(time_to_sec('0:00:00'))) and (time_to_sec(Stot.`Time`) < <cache>(time_to_sec('23:59:59'))))  (cost=0.70 rows
=2) (actual time=0.006..0.007 rows=2 loops=1)
                            -> Index lookup on Stot using PRIMARY (StopId=18847)  (cost=0.70 rows=2) (actual time=0.005..0.006 rows=2 loops=1)
                        -> Filter: (t.TripId = Stot.TripId)  (cost=0.30 rows=1) (actual time=0.005..0.005 rows=1 loops=2)
                            -> Single-row index lookup on t using PRIMARY (TripId=Stot.TripId)  (cost=0.30 rows=1) (actual time=0.005..0.005 rows=1 loops=2)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

- Since there isn’t an improvement after adding index on Time nor for ServiceID, there isn’t an overall improvement by putting indices on both keys.