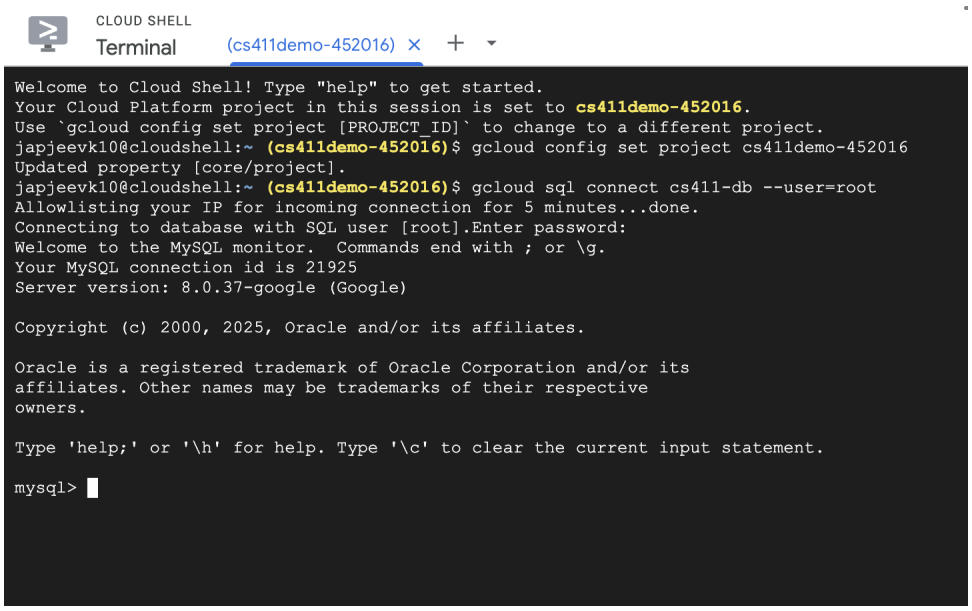
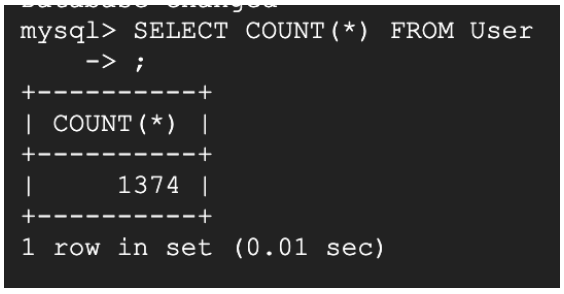


Here are the Data Definition Language (DDL) commands we used to create each of the tables in the database:

```
CREATE TABLE User (
        UserID PRIMARY KEY,
        Username VARCHAR(255) NOT NULL UNIQUE,
        Password VARCHAR(255) NOT NULL,
        Email VARCHAR(255) NOT NULL UNIQUE,
        HomeCurrency VARCHAR(50) NOT NULL
);
```




```
CREATE TABLE Trip (
        TripID PRIMARY KEY,
        TripName VARCHAR(255) NOT NULL,
        StartDate DATE NOT NULL,
        EndDate DATE NOT NULL,
        Description TEXT
```

);

```
mysql> SELECT COUNT(*) FROM Trip
    -> ;
+----------+
| COUNT(*) |
+----------+
|     1496 |
+----------+
1 row in set (0.00 sec)
```

```
CREATE TABLE TripUser (
        TripID INT NOT NULL,
        UserID INT NOT NULL,
        PermissionLevel VARCHAR(50) NOT NULL,
        FOREIGN KEY (TripID) REFERENCES Trip(TripID) ON DELETE CASCADE,
        FOREIGN KEY (UserID) REFERENCES User(UserID) ON DELETE CASCADE,
        PRIMARY KEY (TripID, UserID)
);

CREATE TABLE Booking (
        BookingID PRIMARY KEY,
        TripID INT NOT NULL,
        BookingType VARCHAR(100) NOT NULL,
        BookingDate DATE NOT NULL,
        Cost DECIMAL(10,2) NOT NULL,
        Currency VARCHAR(50) NOT NULL,
        FOREIGN KEY (TripID) REFERENCES Trip(TripID) ON DELETE CASCADE
);

CREATE TABLE Currency (
        Date DATE NOT NULL,
        BaseCurrency VARCHAR(50) NOT NULL,
        TargetCurrency VARCHAR(50) NOT NULL,
        Rate DECIMAL(10,4) NOT NULL,
        PRIMARY KEY (Date, BaseCurrency, TargetCurrency)
```

```
);

CREATE TABLE Expense (
        ExpenseID PRIMARY KEY,
        TripID INT NOT NULL,
        Date DATE NOT NULL,
        Amount DECIMAL(10,2) NOT NULL,
        Currency VARCHAR(50) NOT NULL,
        CategoryName VARCHAR(100) NOT NULL,
        Description TEXT,
        FOREIGN KEY (TripID) REFERENCES Trip(TripID) ON DELETE CASCADE
);
```

```
mysql> SELECT COUNT(*) FROM Expense;
+----------+
| COUNT(*) |
+----------+
|     2006 |
+----------+
1 row in set (0.01 sec)
```

1) Query: Get Total Expenses Per Trip with Currency Conversion
- JOIN (multiple tables) + Aggregation (GROUP BY) + Subquery
- Convert all expenses to the user's home currency and show the total trip cost.

```
SELECT
   t.TripName,
   u.Username,
   SUM(e.Amount * (
      SELECT c.Rate
      FROM Currency c
      WHERE c.BaseCurrency = e.Currency
      AND c.TargetCurrency = u.HomeCurrency
      AND c.Date <= e.Date
      ORDER BY c.Date DESC
      LIMIT 1
   )) AS TotalSpentHomeCurrency
FROM Trip t
JOIN Expense e ON t.TripID = e.TripID
JOIN TripUser tu ON t.TripID = tu.TripID
JOIN User u ON tu.UserID = u.UserID
GROUP BY t.TripName, u.Username;
```

```
mysql>
mysql> SELECT      t.TripName,      u.Username,      SUM(e.Amount * (         SELECT c.Rate          FROM Currency c          WHERE c.BaseCurrency = e.Currency          AND c.TargetCu
rrency = u.HomeCurrency          AND c.Date <= e.Date          ORDER BY c.Date DESC          LIMIT 1     )) AS TotalSpentHomeCurrency FROM Trip t JOIN Expense e ON t.TripID = e.TripID
 JOIN TripUser tu ON t.TripID = tu.TripID JOIN User u ON tu.UserID = u.UserID GROUP BY t.TripName, u.Username LIMIT 15;
+--------------------------+------------------+------------------------+
| TripName                 | Username         | TotalSpentHomeCurrency |
+--------------------------+------------------+------------------------+
| Mountain Trek Switzerland | Jaren_Block11    |                   NULL |
| Cultural Tour Athens     | Gwendolyn_Fadel38 |              95.506740 |
| Cultural Tour Kyoto      | Nakia33          |             743.713899 |
| Cultural Tour Kyoto      | Jess_Hauck       |                   NULL |
| Cultural Tour Athens     | Orie76           |                   NULL |
| Safari Adventure Kenya   | Shyanne_Emard40  |             134.998650 |
| Mountain Trek Switzerland | Maximus.Armstrong |              37.383560 |
| Cultural Tour Kyoto      | Eliza73          |                   NULL |
| Beach Getaway Bali       | Alyce_Gleason    |             648.482804 |
| Mountain Trek Norway     | Elfrieda.Kuvalis |             483.381968 |
| Mountain Trek Norway     | Elinore.Ruecker  |            1045.699537 |
| Cultural Tour Athens     | Precious.Bogan5  |                   NULL |
| Cultural Tour Kyoto      | Vanessa.Nicolas72 |             441.532081 |
| Mountain Trek Norway     | Aliza5           |             129.003264 |
| Beach Getaway Hawaii     | Vince_Hackett47  |             373.971464 |
+--------------------------+------------------+------------------------+
15 rows in set (0.25 sec)

mysql>
```

This query converts all expenses to the user's home currency, and uses a subquery to get the most recent exchange rate.

EXPLAIN ANALYSIS:

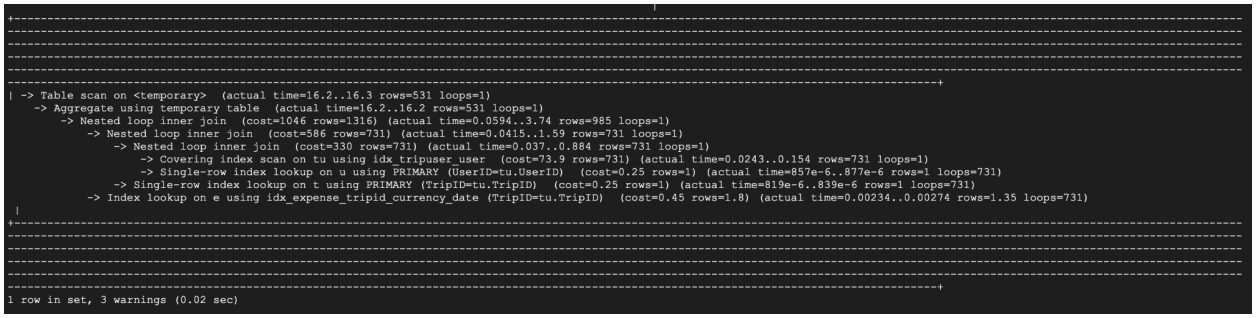
```
mysql> EXPLAIN ANALYZE SELECT      t.TripName,      u.Username,      SUM(e.Amount * (         SELECT c.Rate         FROM Currency c         WHERE c.BaseCurrency = e.Currency         A
ND c.TargetCurrency = u.HomeCurrency         AND c.Date <= e.Date         ORDER BY c.Date DESC         LIMIT 1     )) AS TotalSpentHomeCurrency FROM Trip t JOIN Expense e ON t.TripID
= e.TripID JOIN TripUser tu ON t.TripID = tu.TripID JOIN User u ON tu.UserID = u.UserID GROUP BY t.TripName, u.Username;
+----------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                |
+----------------------------------------------------------------------------------------------------+
| -> Table scan on <temporary>  (actual time=274..274 rows=531 loops=1)
    -> Aggregate using temporary table  (actual time=274..274 rows=531 loops=1)
        -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0637..3.95 rows=985 loops=1)
            -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0487..1.73 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0425..0.922 rows=731 loops=1)
                    -> Covering index scan on tu using UserID  (cost=73.9 rows=731) (actual time=0.0266..0.156 rows=731 loops=1)
                    -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=908e-6..928e-6 rows=1 loops=731)
                -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=961e-6..981e-6 rows=1 loops=731)
            -> Index lookup on e using TripID (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00245..0.00286 rows=1.35 loops=731)
 |
+----------------------------------------------------------------------------------------------------+
1 row in set, 3 warnings (0.28 sec)
```

Tried:
CREATE INDEX idx_currency_lookup ON Currency(BaseCurrency, TargetCurrency, Date DESC);

```
+----------------------------------------------------------------------------------------------------+
| -> Table scan on <temporary>  (actual time=20.2..20.3 rows=531 loops=1)
    -> Aggregate using temporary table  (actual time=20.2..20.2 rows=531 loops=1)
        -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0551..3.68 rows=985 loops=1)
            -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0419..1.58 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0372..0.86 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.025..0.147 rows=731 loops=1)
                    -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=839e-6..859e-6 rows=1 loops=731)
                -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=850e-6..870e-6 rows=1 loops=731)
            -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00226..0.00266 rows=1.35 loops=731)
 |
+----------------------------------------------------------------------------------------------------+
1 row in set, 3 warnings (0.02 sec)
```

Same cost, no improvement

Tried:
CREATE INDEX idx_currency_conversion ON Currency (BaseCurrency, TargetCurrency, Date, Rate DESC);
CREATE INDEX idx_expense_tripid_currency_date ON Expense (TripID, Currency, Date);
CREATE INDEX idx_user_homecurrency ON User (UserID, HomeCurrency);



```
| -> Table scan on <temporary>  (actual time=16.2..16.3 rows=531 loops=1)
    -> Aggregate using temporary table  (actual time=16.2..16.2 rows=531 loops=1)
        -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0594..3.74 rows=985 loops=1)
            -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0415..1.59 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.037..0.884 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0243..0.154 rows=731 loops=1)
                    -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=857e-6..877e-6 rows=1 loops=731)
                -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=819e-6..839e-6 rows=1 loops=731)
            -> Index lookup on e using idx_expense_tripid_currency_date (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00234..0.00274 rows=1.35 loops=731)
 |
1 row in set, 3 warnings (0.02 sec)
```

Same cost, no improvement

We tried implementing several indexes targeting the join operations and the currency exchange rate subquery, but the query cost remained unchanged. We think that indexing foreign keys in TripUser and Expense likely wouldn't improve performance because the database might have already been efficiently handling these joins using existing indexes. Indexes on the Currency table (idx_currency_lookup, idx_currency_conversion) and indexes combining Expense columns (idx_expense_tripid_currency_date) and User columns (idx_user_homecurrency) that we tried to use to optimize the subquery and joins, however, yielded no benefit. We think that one of the reasons for the lack of improvement is because this subquery executes repeatedly for each row, so its overhead might be overshadowing any potential gains from optimizing the individual joins or the lookups within the subquery itself. The final index design would use the database's default indexing based on primary and foreign key constraints, as the indexes we created did not demonstrably improve performance. Maybe a different query structure that avoids the subquery, such as pre-calculating or joining the exchange rates, would be more useful to reduce the cost.

2) Query: Find the Most Expensive Category Per Trip
- JOIN (multiple tables) + Aggregation (GROUP BY) + Subquery
- For each trip, determine the expense category where the user spent the most money.

```
SELECT TripName, CategoryName, total_spent
FROM (
   SELECT
      t.TripName,
      e.CategoryName,
      SUM(e.Amount) AS total_spent,
      RANK() OVER (PARTITION BY t.TripID ORDER BY SUM(e.Amount) DESC) AS `rank`
   FROM Trip t
   JOIN Expense e ON t.TripID = e.TripID
   GROUP BY t.TripID, e.CategoryName, t.TripName
) ranked_categories
WHERE `rank` = 1;
```

```
mysql> SELECT TripName, CategoryName, total_spent FROM (     SELECT          t.TripName,          e.CategoryName,          SUM(e.Amount) AS total_spent,         RANK() OVER (PARTITION
 BY t.TripID ORDER BY SUM(e.Amount) DESC) AS `rank`     FROM Trip t     JOIN Expense e ON t.TripID = e.TripID     GROUP BY t.TripID, e.CategoryName, t.TripName ) ranked_categories WHE
RE `rank` = 1 LIMIT 15;
+------------------------------+---------------+-------------+
| TripName                     | CategoryName  | total_spent |
+------------------------------+---------------+-------------+
| Cultural Tour Athens         | Transport     |      393.47 |
| Beach Getaway Hawaii         | Entertainment |      203.99 |
| Mountain Trek Norway         | Transport     |      699.37 |
| Safari Adventure Kenya       | Entertainment |      225.73 |
| Cultural Tour Athens         | Lodging       |      456.73 |
| Beach Getaway Bali           | Entertainment |      466.96 |
| Beach Getaway Hawaii         | Misc          |      474.17 |
| Safari Adventure Kenya       | Food          |      135.39 |
| Cultural Tour Athens         | Misc          |      276.95 |
| Mountain Trek Switzerland    | Misc          |      535.25 |
| Mountain Trek Switzerland    | Shopping      |      479.54 |
| Safari Adventure South Africa | Transport    |      314.27 |
| Beach Getaway Hawaii         | Misc          |       36.33 |
| Mountain Trek Norway         | Food          |      101.47 |
| Mountain Trek Switzerland    | Entertainment |      495.74 |
+------------------------------+---------------+-------------+
15 rows in set (0.01 sec)

mysql>
```

This query uses RANK() to get the highest spending category per trip, and helps users see where they spent the most money.

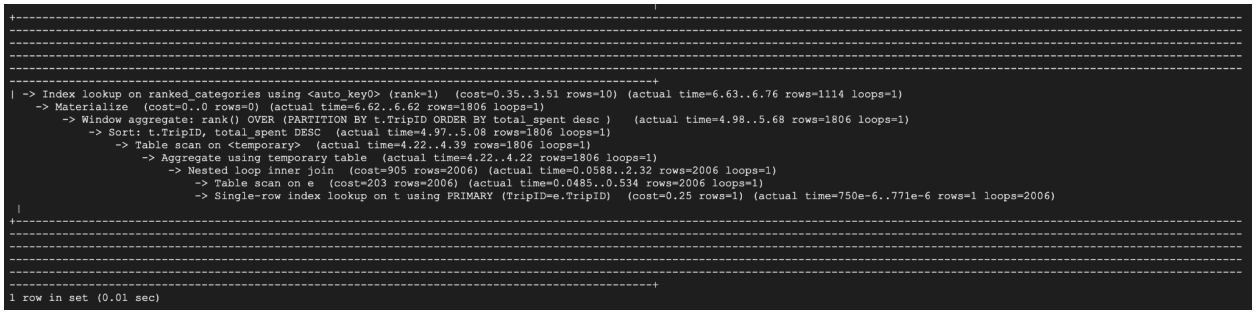
EXPLAIN ANALYSIS:

```
| -> Index lookup on ranked_categories using <auto_key0> (rank=1)  (cost=0.35..3.51 rows=10) (actual time=6.81..6.94 rows=1114 loops=1)
    -> Materialize  (cost=0..0 rows=0) (actual time=6.8..6.8 rows=1806 loops=1)
        -> Window aggregate: rank() OVER (PARTITION BY t.TripID ORDER BY total_spent desc )   (actual time=5.12..5.83 rows=1806 loops=1)
            -> Sort: t.TripID, total_spent DESC  (actual time=5.11..5.22 rows=1806 loops=1)
                -> Table scan on <temporary>  (actual time=4.32..4.54 rows=1806 loops=1)
                    -> Aggregate using temporary table  (actual time=4.32..4.32 rows=1806 loops=1)
                        -> Nested loop inner join  (cost=905 rows=2006) (actual time=0.0638..2.37 rows=2006 loops=1)
                            -> Table scan on e  (cost=203 rows=2006) (actual time=0.052..0.534 rows=2006 loops=1)
                            -> Single-row index lookup on t using PRIMARY (TripID=e.TripID)  (cost=0.25 rows=1) (actual time=753e-6..773e-6 rows=1 loops=2006)
 |
1 row in set (0.02 sec)
```

Tried:
CREATE INDEX idx_expense_tripid_category ON Expense (TripID, CategoryName);

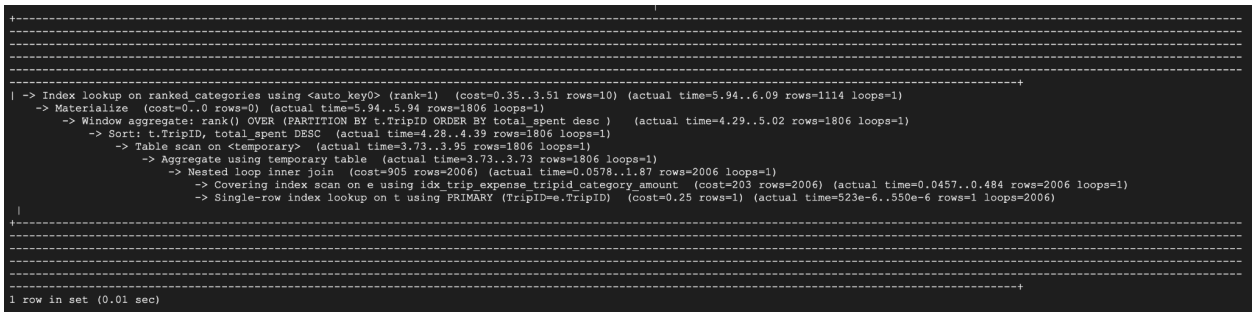
```
| -> Index lookup on ranked_categories using <auto_key0> (rank=1)  (cost=0.35..3.51 rows=10) (actual time=6.75..6.9 rows=1114 loops=1)
    -> Materialize  (cost=0..0 rows=0) (actual time=6.75..6.75 rows=1806 loops=1)
        -> Window aggregate: rank() OVER (PARTITION BY t.TripID ORDER BY total_spent desc )   (actual time=5.08..5.81 rows=1806 loops=1)
            -> Sort: t.TripID, total_spent DESC  (actual time=5.07..5.18 rows=1806 loops=1)
                -> Table scan on <temporary>  (actual time=4.3..4.49 rows=1806 loops=1)
                    -> Aggregate using temporary table  (actual time=4.3..4.3 rows=1806 loops=1)
                        -> Nested loop inner join  (cost=905 rows=2006) (actual time=0.0585..2.36 rows=2006 loops=1)
                            -> Table scan on e  (cost=203 rows=2006) (actual time=0.0481..0.549 rows=2006 loops=1)
                            -> Single-row index lookup on t using PRIMARY (TripID=e.TripID)  (cost=0.25 rows=1) (actual time=754e-6..774e-6 rows=1 loops=2006)
 |
1 row in set (0.01 sec)
```

Tried:
CREATE INDEX idx_expense_categoryname_amount ON Expense (CategoryName, Amount DESC);



```
| -> Index lookup on ranked_categories using <auto_key0> (rank=1)  (cost=0.35..3.51 rows=10) (actual time=6.63..6.76 rows=1114 loops=1)
    -> Materialize  (cost=0..0 rows=0) (actual time=6.62..6.62 rows=1806 loops=1)
        -> Window aggregate: rank() OVER (PARTITION BY t.TripID ORDER BY total_spent desc )  (actual time=4.98..5.68 rows=1806 loops=1)
            -> Sort: t.TripID, total_spent DESC  (actual time=4.97..5.08 rows=1806 loops=1)
                -> Table scan on <temporary>  (actual time=4.22..4.39 rows=1806 loops=1)
                    -> Aggregate using temporary table  (actual time=4.22..4.22 rows=1806 loops=1)
                        -> Nested loop inner join  (cost=905 rows=2006) (actual time=0.0588..2.32 rows=2006 loops=1)
                            -> Table scan on e  (cost=203 rows=2006) (actual time=0.0485..0.534 rows=2006 loops=1)
                            -> Single-row index lookup on t using PRIMARY (TripID=e.TripID)  (cost=0.25 rows=1) (actual time=750e-6..771e-6 rows=1 loops=2006)
 |
1 row in set (0.01 sec)
```

Tried:
CREATE INDEX idx_trip_expense_tripid_category_amount ON Expense (TripID, CategoryName, Amount DESC);
CREATE INDEX idx_expense_tripid_amount_categoryname ON Expense (TripID, Amount DESC, CategoryName);



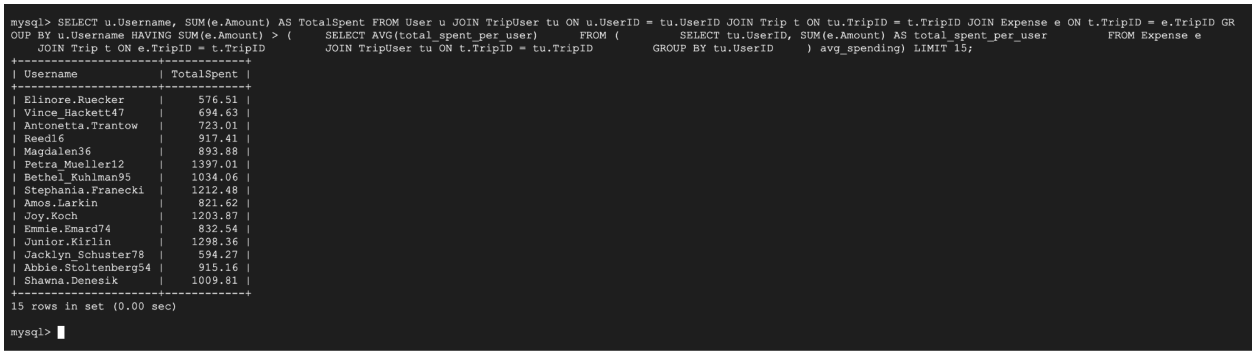
```
| -> Index lookup on ranked_categories using <auto_key0> (rank=1)  (cost=0.35..3.51 rows=10) (actual time=5.94..6.09 rows=1114 loops=1)
    -> Materialize  (cost=0..0 rows=0) (actual time=5.94..5.94 rows=1806 loops=1)
        -> Window aggregate: rank() OVER (PARTITION BY t.TripID ORDER BY total_spent desc )  (actual time=4.29..5.02 rows=1806 loops=1)
            -> Sort: t.TripID, total_spent DESC  (actual time=4.28..4.39 rows=1806 loops=1)
                -> Table scan on <temporary>  (actual time=3.73..3.95 rows=1806 loops=1)
                    -> Aggregate using temporary table  (actual time=3.73..3.73 rows=1806 loops=1)
                        -> Nested loop inner join  (cost=905 rows=2006) (actual time=0.0578..1.87 rows=2006 loops=1)
                            -> Covering index scan on e using idx_trip_expense_tripid_category_amount  (cost=203 rows=2006) (actual time=0.0457..0.484 rows=2006 loops=1)
                            -> Single-row index lookup on t using PRIMARY (TripID=e.TripID)  (cost=0.25 rows=1) (actual time=523e-6..550e-6 rows=1 loops=2006)
 |
1 row in set (0.01 sec)
```

We tried to optimize the query designed to find the most expensive category per trip by creating several indexes. Our initial approach involved indexing the Expense table on TripID and CategoryName (idx_expense_tripid_category), trying to improve the join operation with the Trip table and facilitate the subsequent grouping by category. Then, we focused on the aggregation and ranking aspects by indexing CategoryName and Amount in descending order (idx_expense_categoryname_amount). This was supposed to speed up the calculation of the total spent per category and the ordering required for the RANK() function. Finally, we experimented with composite indexes on the Expense table, combining TripID, CategoryName, and Amount in different orders (idx_trip_expense_tripid_category_amount and idx_expense_tripid_amount_categoryname). These composite indexes were designed to potentially cover both the join and the ranking operations more efficiently. However, despite these efforts, the EXPLAIN ANALYSIS indicated that none of these indexing strategies resulted in an improvement in query cost. This suggests that the bottleneck might lie in the window function RANK(), which requires processing the aggregated data for each trip to determine the rank. The database might already be performing the join and aggregation steps reasonably efficiently, and the cost associated with calculating the rank across partitions is dominating the overall execution time. Our final index design would likely rely on the default indexes provided by primary and foreign key constraints, as the explicitly created indexes did not reduce the query's cost. We might try exploring alternative approaches to identifying the maximum

spending category per trip that avoid the performance overhead of the RANK() function, if such alternatives exist while maintaining accuracy.

3) Query: Get Users Who Have Spent Above Average on Trips
- Aggregation (GROUP BY) + Subquery
- Find users who have spent more than the average total trip expense.

```
SELECT u.Username, SUM(e.Amount) AS TotalSpent
FROM User u
JOIN TripUser tu ON u.UserID = tu.UserID
JOIN Trip t ON tu.TripID = t.TripID
JOIN Expense e ON t.TripID = e.TripID
GROUP BY u.Username
HAVING SUM(e.Amount) > (
   SELECT AVG(total_spent_per_user)
   FROM (
      SELECT tu.UserID, SUM(e.Amount) AS total_spent_per_user
      FROM Expense e
      JOIN Trip t ON e.TripID = t.TripID
      JOIN TripUser tu ON t.TripID = tu.TripID
      GROUP BY tu.UserID
   ) avg_spending
);
```



```
mysql> SELECT u.Username, SUM(e.Amount) AS TotalSpent FROM User u JOIN TripUser tu ON u.UserID = tu.UserID JOIN Trip t ON tu.TripID = t.TripID JOIN Expense e ON t.TripID = e.TripID GR
OUP BY u.Username HAVING SUM(e.Amount) > (     SELECT AVG(total_spent_per_user)      FROM (         SELECT tu.UserID, SUM(e.Amount) AS total_spent_per_user         FROM Expense e
    JOIN Trip t ON e.TripID = t.TripID         JOIN TripUser tu ON t.TripID = tu.TripID         GROUP BY tu.UserID     ) avg_spending) LIMIT 15;
+---------------------+------------+
| Username            | TotalSpent |
+---------------------+------------+
| Elinore.Ruecker     |     576.51 |
| Vince_Hackett47     |     694.63 |
| Antonetta.Trantow   |     723.01 |
| Reed16              |     917.41 |
| Magdalen36          |     893.88 |
| Petra_Mueller12     |    1397.01 |
| Bethel_Kuhlman95    |    1034.06 |
| Stephania.Franecki  |    1212.48 |
| Amos.Larkin         |     821.62 |
| Joy.Koch            |    1203.87 |
| Emmie.Emard74       |     832.54 |
| Junior.Kirlin       |    1298.36 |
| Jacklyn_Schuster78  |     594.27 |
| Abbie.Stoltenberg54 |     915.16 |
| Shawna.Denesik      |    1009.81 |
+---------------------+------------+
15 rows in set (0.00 sec)

mysql>
```

This query identifies big spenders and uses a subquery to compare total spending against the average spending.

EXPLAIN ANALYSIS:

```
| -> Filter: (`sum(e.Amount)` > (select #2))  (actual tim[illegible].75..6.83 rows=161 loops=1)
    -> Table scan on <temporary>  (actual time=4.07..4.11 rows=428 loops=1)
        -> Aggregate using temporary table  (actual time=4.07..4.07 rows=428 loops=1)
            -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0592..3.4 rows=985 loops=1)
                -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0426..1.44 rows=731 loops=1)
                    -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0379..0.788 rows=731 loops=1)
                        -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0251..0.141 rows=731 loops=1)
                        -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=751e-6..771e-6 rows=1 loops=731)
                    -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=761e-6..781e-6 rows=1 loops=731)
                -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00213..0.00249 rows=1.35 loops=731)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(avg_spending.total_spent_per_user)  (cost=1037..1037 rows=1) (actual time=2.66..2.66 rows=1 loops=1)
            -> Table scan on avg_spending  (cost=975..984 rows=527) (actual time=2.6..2.63 rows=428 loops=1)
                -> Materialize  (cost=975..975 rows=527) (actual time=2.6..2.6 rows=428 loops=1)
                    -> Group aggregate: sum(e.Amount)  (cost=922 rows=527) (actual time=0.028..2.52 rows=428 loops=1)
                        -> Nested loop inner join  (cost=790 rows=1316) (actual time=0.0209..2.38 rows=985 loops=1)
                            -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0158..0.647 rows=731 loops=1)
                                -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0127..0.127 rows=731 loops=1)
                                -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=579e-6..599e-6 rows=1 loo
ps=731)
                            -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00188..0.0022 rows=1.35 loops=731)
 |
```

JOIN INDEXES:

CREATE INDEX idx_tripuser_userid ON TripUser(UserID);
CREATE INDEX idx_tripuser_tripid ON TripUser(TripID);

```
----------------+
| -> Filter: (`sum(e.Amount)` > (select #2))  (actual time=6.73..6.82 rows=161 loops=1)
    -> Table scan on <temporary>  (actual time=4.04..4.09 rows=428 loops=1)
        -> Aggregate using temporary table  (actual time=4.04..4.04 rows=428 loops=1)
            -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0555..3.38 rows=985 loops=1)
                -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0407..1.46 rows=731 loops=1)
                    -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0357..0.795 rows=731 loops=1)
                        -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0232..0.139 rows=731 loops=1)
                        -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=761e-6..781e-6 rows=1 loops=731)
                    -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=742e-6..762e-6 rows=1 loops=731)
                -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00208..0.00243 rows=1.35 loops=731)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(avg_spending.total_spent_per_user)  (cost=1037..1037 rows=1) (actual time=2.68..2.68 rows=1 loops=1)
            -> Table scan on avg_spending  (cost=975..984 rows=527) (actual time=2.61..2.64 rows=428 loops=1)
                -> Materialize  (cost=975..975 rows=527) (actual time=2.61..2.61 rows=428 loops=1)
                    -> Group aggregate: sum(e.Amount)  (cost=922 rows=527) (actual time=0.0281..2.54 rows=428 loops=1)
                        -> Nested loop inner join  (cost=790 rows=1316) (actual time=0.0212..2.39 rows=985 loops=1)
                            -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0159..0.662 rows=731 loops=1)
                                -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0125..0.129 rows=731 loops=1)
                                -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=595e-6..615e-6 rows=1 loops=731)
                            -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00187..0.00219 rows=1.35 loops=731)
 |
```

Old: CREATE INDEX idx_tripuser_userid_tripid ON TripUser(UserID, TripID);
New index: CREATE INDEX idx_expense_tripid ON Expense(TripID);

Old:

```
---------------+
| -> Filter: (`sum(e.Amount)` > (select #2))  (actual time=6.72..6.8 rows=161 loops=1)
    -> Table scan on <temporary>  (actual time=4.04..4.08 rows=428 loops=1)
        -> Aggregate using temporary table  (actual time=4.04..4.04 rows=428 loops=1)
            -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0642..3.37 rows=985 loops=1)
                -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0412..1.42 rows=731 loops=1)
                    -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0364..0.792 rows=731 loops=1)
                        -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0249..0.141 rows=731 loops=1)
                        -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=757e-6..777e-6 rows=1 loops=731)
                    -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=729e-6..749e-6 rows=1 loops=731)
                -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00212..0.00248 rows=1.35 loops=731)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(avg_spending.total_spent_per_user)  (cost=1037..1037 rows=1) (actual time=2.66..2.66 rows=1 loops=1)
            -> Table scan on avg_spending  (cost=975..984 rows=527) (actual time=2.59..2.62 rows=428 loops=1)
                -> Materialize  (cost=975..975 rows=527) (actual time=2.59..2.59 rows=428 loops=1)
                    -> Group aggregate: sum(e.Amount)  (cost=922 rows=527) (actual time=0.0253..2.52 rows=428 loops=1)
                        -> Nested loop inner join  (cost=790 rows=1316) (actual time=0.0179..2.37 rows=985 loops=1)
                            -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0134..0.651 rows=731 loops=1)
                                -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0109..0.127 rows=731 loops=1)
                                -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=584e-6..604e-6 rows=1 loops=731)
                            -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00186..0.00219 rows=1.35 loops=731)
 |
+--------------------------------------------------------------------------------------------------------------------------------------------------------
```

New:

```
-> Filter: (`sum(e.Amount)` > (select #2))  (actual time=6.84..6.92 rows=161 loops=1)
  -> Table scan on <temporary>  (actual time=4.19..4.23 rows=428 loops=1)
      -> Aggregate using temporary table  (actual time=4.18..4.18 rows=428 loops=1)
          -> Nested loop inner join  (cost=1049 rows=1316) (actual time=0.0545..3.51 rows=985 loops=1)
              -> Nested loop inner join  (cost=588 rows=731) (actual time=0.0386..1.4 rows=731 loops=1)
                  -> Nested loop inner join  (cost=332 rows=731) (actual time=0.0343..0.775 rows=731 loops=1)
                      -> Covering index scan on tu using idx_tripuser_user  (cost=76.1 rows=731) (actual time=0.0225..0.139 rows=731 loops=1)
                      -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=733e-6..753e-6 rows=1 loops=731)
                  -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=720e-6..741e-6 rows=1 loops=731)
              -> Index lookup on e using idx_expense_tripid (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00236..0.00271 rows=1.35 loops=731)
  -> Select #2 (subquery in condition; run only once)
      -> Aggregate: avg(avg_spending.total_spent_per_user)  (cost=1039..1039 rows=1) (actual time=2.63..2.63 rows=1 loops=1)
          -> Table scan on avg_spending  (cost=977..986 rows=527) (actual time=2.57..2.6 rows=428 loops=1)
              -> Materialize  (cost=977..977 rows=527) (actual time=2.57..2.57 rows=428 loops=1)
                  -> Group aggregate: sum(e.Amount)  (cost=924 rows=527) (actual time=0.0251..2.49 rows=428 loops=1)
                      -> Nested loop inner join  (cost=793 rows=1316) (actual time=0.0184..2.35 rows=985 loops=1)
                          -> Nested loop inner join  (cost=332 rows=731) (actual time=0.0135..0.636 rows=731 loops=1)
                              -> Covering index scan on tu using idx_tripuser_user  (cost=76.1 rows=731) (actual time=0.0108..0.126 rows=731 loops=1)
                              -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=566e-6..586e-6 rows=1 loops=731)
                          -> Index lookup on e using idx_expense_tripid (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00189..0.00219 rows=1.35 loops=731)
```

CREATE INDEX idx_expense_tripid_amount ON Expense(TripID, Amount);

```
-----------+
| -> Filter: (`sum(e.Amount)` > (select #2))  (actual time=6.82..6.9 rows=161 loops=1)
    -> Table scan on <temporary>  (actual time=4.13..4.18 rows=428 loops=1)
        -> Aggregate using temporary table  (actual time=4.13..4.13 rows=428 loops=1)
            -> Nested loop inner join  (cost=1046 rows=1316) (actual time=0.0575..3.46 rows=985 loops=1)
                -> Nested loop inner join  (cost=586 rows=731) (actual time=0.0417..1.47 rows=731 loops=1)
                    -> Nested loop inner join  (cost=330 rows=731) (actual time=0.037..0.829 rows=731 loops=1)
                        -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0243..0.141 rows=731 loops=1)
                        -> Single-row index lookup on u using PRIMARY (UserID=tu.UserID)  (cost=0.25 rows=1) (actual time=797e-6..818e-6 rows=1 loops=731)
                    -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=743e-6..764e-6 rows=1 loops=731)
                -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00216..0.00253 rows=1.35 loops=731)
    -> Select #2 (subquery in condition; run only once)
        -> Aggregate: avg(avg_spending.total_spent_per_user)  (cost=1037..1037 rows=1) (actual time=2.66..2.66 rows=1 loops=1)
            -> Table scan on avg_spending  (cost=975..984 rows=527) (actual time=2.6..2.63 rows=428 loops=1)
                -> Materialize  (cost=975..975 rows=527) (actual time=2.6..2.6 rows=428 loops=1)
                    -> Group aggregate: sum(e.Amount)  (cost=922 rows=527) (actual time=0.0267..2.53 rows=428 loops=1)
                        -> Nested loop inner join  (cost=790 rows=1316) (actual time=0.0198..2.38 rows=985 loops=1)
                            -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0146..0.659 rows=731 loops=1)
                                -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0122..0.132 rows=731 loops=1)
                                -> Single-row covering index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=579e-6..599e-6 rows=1 loops=731)
                            -> Index lookup on e using idx_expense_trip (TripID=tu.TripID)  (cost=0.45 rows=1.8) (actual time=0.00187..0.00218 rows=1.35 loops=731)
|
```

We experimented with several indexes to optimize the query that identifies users who spent above the average trip expense. We focused on improving the join performance by indexing the foreign key UserID in TripUser(idx_tripuser_userid), and the foreign key TripID in TripUser (idx_tripuser_tripid), because we thought that these would speed up the data retrieval across the joined tables. We also tried a composite index on TripUser combining UserID and TripID (idx_tripuser_userid_tripid). Finally, we created an index on Expense that included both TripID and Amount (idx_expense_tripid_amount), to try and improve the aggregation within the main and subquery. However, EXPLAIN ANALYSIS indicated no significant improvement in the cost. We believe this lack of impact might be because of the subquery that calculates the average spending. The database might need to process a large portion of the data to compute this average, and the indexes on the join columns might not reduce the overall workload to influence the query cost. Therefore, our final index design would likely rely on the default indexes created for primary and foreign keys. To achieve better performance, we could look at alternative queries that don't need a separate subquery to calculate the average.

Revised Explanation:

We attempted to index on the UserID and TripId in TripUser which was incorrect as these attributes together make up the composite primary key for TripUser. We simply indexed on TripID this time as a foreign key for the Expense table in order to extract the TripID in a more efficient manner before the join is executed. We think the cost of the query remains the same because we think the query planner is already able to do the join efficiently using the existing

primary key index. So, this additional index on TripID did not improve the performance. It could also be because we didn't have many unique TripIDs in our dataset.

4) Query: Find Trips with Shared Expenses Between Users
- JOIN (multiple tables) + GROUP BY + HAVING
- Identify trips where two or more users contributed to expenses, useful for group travel.

There were less than 15 rows that fulfilled the conditions for this query

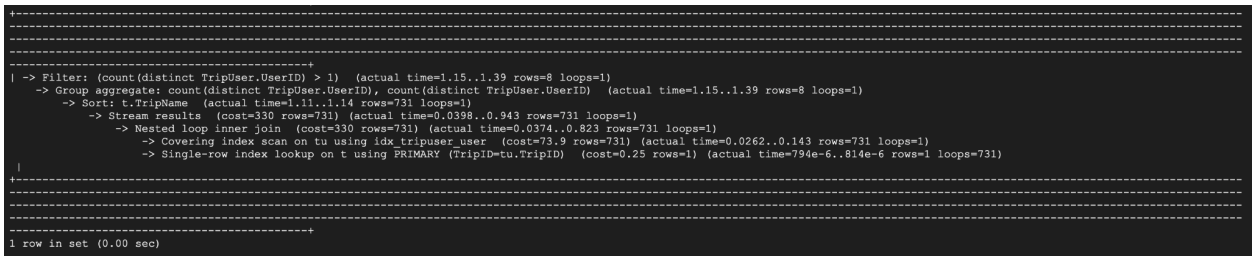
```
SELECT t.TripName, COUNT(DISTINCT tu.UserID) AS NumUsers
FROM Trip t
JOIN TripUser tu ON t.TripID = tu.TripID
GROUP BY t.TripName
HAVING COUNT(DISTINCT tu.UserID) > 1;
```

```
mysql> SELECT t.TripName, COUNT(DISTINCT tu.UserID) AS NumUsers
    -> FROM Trip t
    -> JOIN TripUser tu ON t.TripID = tu.TripID
    -> GROUP BY t.TripName
    -> HAVING COUNT(DISTINCT tu.UserID) > 1;
+------------------------------+----------+
| TripName                     | NumUsers |
+------------------------------+----------+
| Beach Getaway Bali           |      105 |
| Beach Getaway Hawaii         |       80 |
| Cultural Tour Athens         |      125 |
| Cultural Tour Kyoto          |      108 |
| Mountain Trek Norway         |      114 |
| Mountain Trek Switzerland    |       81 |
| Safari Adventure Kenya       |       58 |
| Safari Adventure South Africa |       26 |
+------------------------------+----------+
8 rows in set (0.01 sec)

mysql>
```

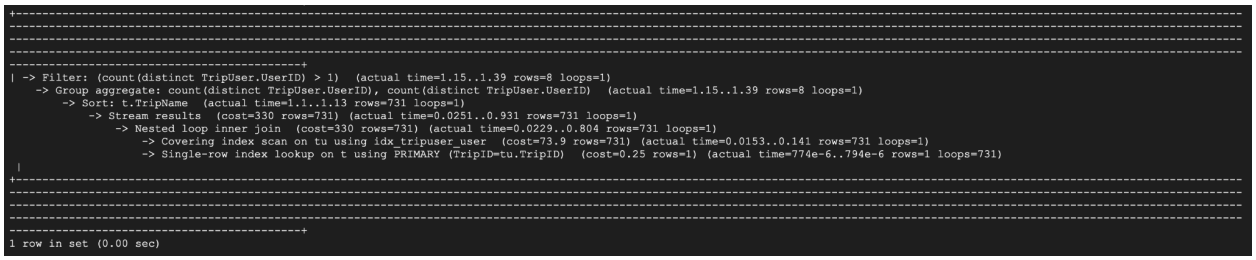
This query detects shared trips where multiple users have added expenses
Helps users see who contributed to group travel expenses

EXPLAIN ANALYSIS:



```
| -> Filter: (count(distinct TripUser.UserID) > 1)  (actual time=1.15..1.39 rows=8 loops=1)
    -> Group aggregate: count(distinct TripUser.UserID), count(distinct TripUser.UserID)  (actual time=1.15..1.39 rows=8 loops=1)
        -> Sort: t.TripName  (actual time=1.11..1.14 rows=731 loops=1)
            -> Stream results  (cost=330 rows=731) (actual time=0.0398..0.943 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0374..0.823 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0262..0.143 rows=731 loops=1)
                    -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=794e-6..814e-6 rows=1 loops=731)
 |
1 row in set (0.00 sec)
```

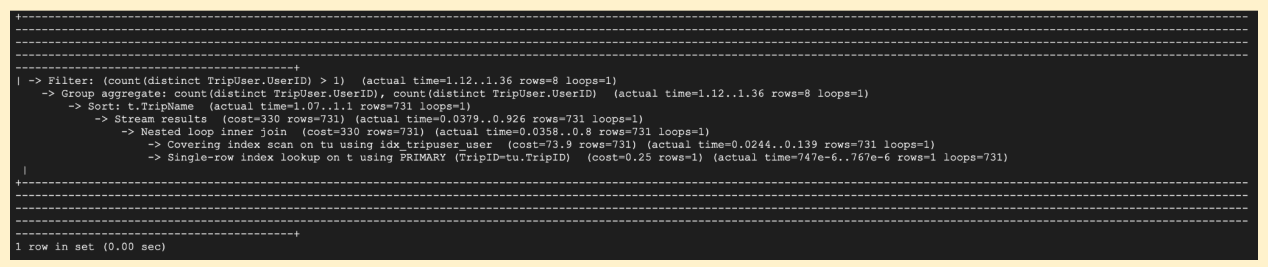
Tried:
CREATE INDEX idx_trip_tripname ON Trip (TripName);



```
| -> Filter: (count(distinct TripUser.UserID) > 1)  (actual time=1.15..1.39 rows=8 loops=1)
    -> Group aggregate: count(distinct TripUser.UserID), count(distinct TripUser.UserID)  (actual time=1.15..1.39 rows=8 loops=1)
        -> Sort: t.TripName  (actual time=1.1..1.13 rows=731 loops=1)
            -> Stream results  (cost=330 rows=731) (actual time=0.0251..0.931 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0229..0.804 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0153..0.141 rows=731 loops=1)
                    -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=774e-6..794e-6 rows=1 loops=731)
 |
1 row in set (0.00 sec)
```

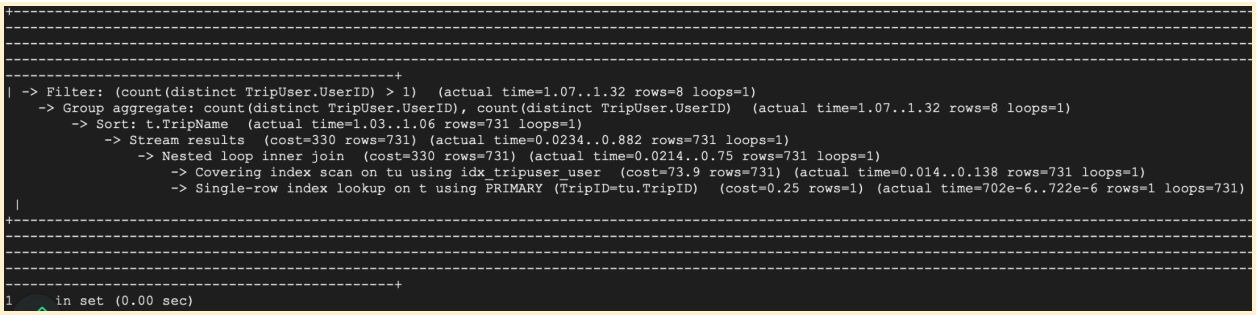
Tried:
Old: CREATE INDEX idx_tripuser_tripid_userid ON TripUser (TripID, UserID);
New: CREATE INDEX idx_trip_tripname_tripid ON Trip(TripName, TripID);

Old:



```
| -> Filter: (count(distinct TripUser.UserID) > 1)  (actual time=1.12..1.36 rows=8 loops=1)
    -> Group aggregate: count(distinct TripUser.UserID), count(distinct TripUser.UserID)  (actual time=1.12..1.36 rows=8 loops=1)
        -> Sort: t.TripName  (actual time=1.07..1.1 rows=731 loops=1)
            -> Stream results  (cost=330 rows=731) (actual time=0.0379..0.926 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0358..0.8 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0244..0.139 rows=731 loops=1)
                    -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=747e-6..767e-6 rows=1 loops=731)
 |
1 row in set (0.00 sec)
```

New:



```
| -> Filter: (count(distinct TripUser.UserID) > 1)  (actual time=1.07..1.32 rows=8 loops=1)
    -> Group aggregate: count(distinct TripUser.UserID), count(distinct TripUser.UserID)  (actual time=1.07..1.32 rows=8 loops=1)
        -> Sort: t.TripName  (actual time=1.03..1.06 rows=731 loops=1)
            -> Stream results  (cost=330 rows=731) (actual time=0.0234..0.882 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0214..0.75 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.014..0.138 rows=731 loops=1)
                    -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=702e-6..722e-6 rows=1 loops=731)
 |
1    in set (0.00 sec)
```

Tried:
CREATE INDEX idx_tripuser_userid ON TripUser (UserID);

```
+----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------+
| -> Filter: (count(distinct TripUser.UserID) > 1)  (actual time=1.15..1.4 rows=8 loops=1)
    -> Group aggregate: count(distinct TripUser.UserID), count(distinct TripUser.UserID)  (actual time=1.15..1.4 rows=8 loops=1)
        -> Sort: t.TripName  (actual time=1.11..1.14 rows=731 loops=1)
            -> Stream results  (cost=330 rows=731) (actual time=0.0353..0.946 rows=731 loops=1)
                -> Nested loop inner join  (cost=330 rows=731) (actual time=0.0332..0.829 rows=731 loops=1)
                    -> Covering index scan on tu using idx_tripuser_user  (cost=73.9 rows=731) (actual time=0.0227..0.158 rows=731 loops=1)
                    -> Single-row index lookup on t using PRIMARY (TripID=tu.TripID)  (cost=0.25 rows=1) (actual time=782e-6..802e-6 rows=1 loops=731)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------+
1 row in set (0.01 sec)
```

We explored several indexing strategies to potentially improve the performance of the query that identifies trips with shared expenses. We started by indexing the TripName column in the Trip table (idx_trip_tripname). The rationale behind this was to optimize the GROUP BY operation, since grouping by an indexed column can sometimes be more efficient. Finally, we tried indexing the UserID column in the TripUser table (idx_tripuser_userid) specifically to potentially speed up the COUNT(DISTINCT tu.UserID) aggregation. But the EXPLAIN ANALYSIS indicated that none of these indexing changes resulted in a noticeable improvement in the query cost. Given that the query returns a small number of rows (less than 15), it's possible that the overhead of the joins and aggregations is already minimal, and the database is efficiently handling the query even without these additional indexes. We think the final index design might rely on the default indexes that were already created for the primary and foreign key constraints, as the indexes we tried did not provide a performance benefit for this particular query. Maybe for such queries with small result sets, optimizations through indexing are not as impactful as other factors like query clarity and maintainability.

Revised Explanation:
We attempted to index on TripID and UserID but this was redundant as these two keys make up composite primary keys in the TripUser table. In order to fix this issue, we now indexed on TripID and UserID as in order to improve the performance of joins that filter by UserID first. We noticed that the performance of the query remained the same. This might be because the dataset is small, so there wasn't a big improvement in speed and this index might be more useful with more complicated queries or a larger dataset.

Database on GCP:

```
mysql> USE travelEase;
Database changed
mysql> 
```