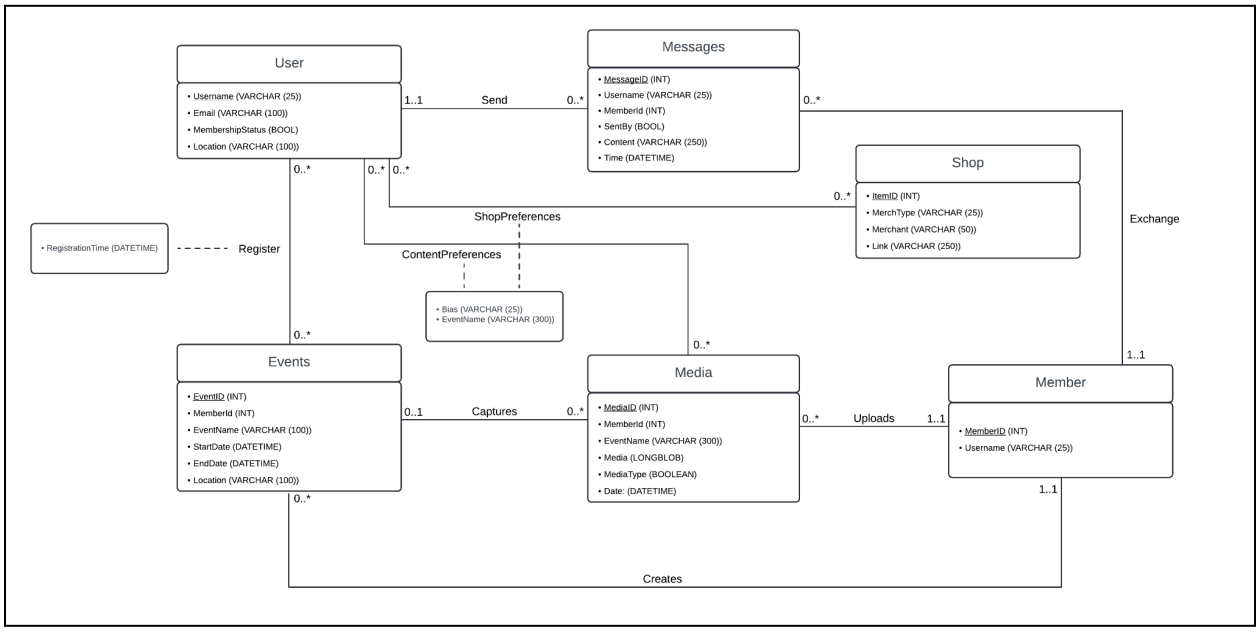
Nithin Parthasarathy (nithinp2), Neil Teje (nteje2), Abhishek Saigal (asaigal3), Ritvik Sood (ritviks3)

# LePhoning

## 1 UML Diagram (updated from Stage 2) - for stage 2 regrade




## 2 Part 1: DDL Commands

```
CREATE TABLE IF NOT EXISTS User (
   Username VARCHAR(25) PRIMARY KEY,
   Email VARCHAR(100),
   MembershipStatus BOOLEAN,
   Location VARCHAR(100)
);

CREATE TABLE IF NOT EXISTS Member (
   MemberId INT PRIMARY KEY,
   Username VARCHAR(25)
);

CREATE TABLE IF NOT EXISTS Events (
   EventID INT PRIMARY KEY,
   EventName VARCHAR(100),
   StartDate DATETIME,
```

```
    EndDate DATETIME,
    Location VARCHAR(100),
    MemberId INT,
    FOREIGN KEY (MemberId) REFERENCES Member(MemberId)
    ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Messages (
    MessageID INT PRIMARY KEY,
    Username VARCHAR(25),
    MemberId INT,
    SentBy BOOL,
    Content VARCHAR(250),
    Time DATETIME,
    FOREIGN KEY (Username) REFERENCES User(Username)
    ON DELETE CASCADE,
    FOREIGN KEY (MemberId) REFERENCES Member(MemberId)
    ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Media (
    MediaID INT PRIMARY KEY,
    EventName VARCHAR(300),
    Media LONGBLOB,
    MediaType BOOLEAN,
    Date DATETIME,
    EventId INT,
    MemberId INT,
    FOREIGN KEY (EventId) REFERENCES Events(EventId)
    ON DELETE SET NULL,
    FOREIGN KEY (MemberId) REFERENCES Member(MemberId)
    ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS Shop (
    ItemID INT PRIMARY KEY,
```

```
    MerchType VARCHAR(25),
    Merchant VARCHAR(50),
    Link VARCHAR(250)
);

CREATE TABLE IF NOT EXISTS Register (
    Username VARCHAR(25),
    EventID INT,
    RegistrationTime DATETIME,
    PRIMARY KEY (Username, EventID),
    FOREIGN KEY (Username) REFERENCES User(Username),
    FOREIGN KEY (EventID) REFERENCES Events(EventID)
);

CREATE TABLE IF NOT EXISTS ContentPreferences (
    Username VARCHAR(25),
    MediaID INT,
    Bias VARCHAR(25),
    EventName VARCHAR(300),
    PRIMARY KEY (Username, MediaID),
    FOREIGN KEY (Username) REFERENCES User(Username),
    FOREIGN KEY (MediaID) REFERENCES Media(MediaID)
);

CREATE TABLE IF NOT EXISTS ShopPreferences (
    Username VARCHAR(25),
    ItemID INT,
    Bias VARCHAR(25),
    EventName VARCHAR(300),
    PRIMARY KEY (Username, ItemID),
    FOREIGN KEY (Username) REFERENCES User(Username),
    FOREIGN KEY (ItemID) REFERENCES Shop(ItemID)
);
```

## 3 Part 1: Inserting Data

We auto-generated data in our User table because we do not have any users yet. For similar reasons we also auto-generated messages into our Messages, Shop, and ShopPreferences tables. Our Member table has the unique key and name of each of the five members of LE SSERAFIM.

## 4 Part 1: Advanced SQL Queries

**SQL Query 1: Select all the events that a user can attend based on the following criteria: The user's bias is attending the event, the event is happening in the user's location, and the event is within 6 months of the user's latest message to their bias.**

This query finds all the events that a user can attend based on certain criteria related to the user bias, user location and user message history. The query joins the User, ShopPreferences and Events relations and uses subqueries to find the member username and latest user message time to the member.

```
SELECT u.Username, e.EventName, e.Location
FROM User u
JOIN ShopPreferences s ON u.Username = s.Username
NATURAL JOIN Events e
WHERE u.Location = e.Location
AND s.Bias = (SELECT Username FROM Member m WHERE m.MemberId = e.MemberId)
AND TIMESTAMPDIFF(MONTH, e.StartDate, (SELECT MAX(Time) FROM Messages m WHERE m.Username
= u.Username AND s.Bias = (SELECT Username FROM Member m WHERE m.MemberId = e.MemberId)))
<= 6
ORDER BY u.Username, e.EventName DESC
LIMIT 15;
```

**Output:**

RESULTS

| Username | EventName | Location |
|---|---|---|
| aaronadams | Tour | Mexico City, Mexico |
| aaronadams | Tour | Mexico City, Mexico |
| aaronadams | Merch Sale Event | Mexico City, Mexico |
| abyrd | Merch Sale Event | Beijing, China |
| adamwilliams | Merch Sale Event | Manila, Philippines |
| adamwilliams | Merch Sale Event | Manila, Philippines |
| aharper | Concert | Kolkata, India |
| aliciagreen | Reality Show | Delhi, India |
| aliciapalmer | Fan Meet-Up | Lagos, Nigeria |
| aliciapalmer | Fan Meet-Up | Lagos, Nigeria |
| aliciapalmer | Fan Meet-Up | Lagos, Nigeria |
| amandaclark | Tour | Karachi, Pakistan |
| amber63 | Merch Sale Event | Karachi, Pakistan |
| amber79 | Reality Show | Mexico City, Mexico |
| amber79 | Reality Show | Mexico City, Mexico |

Rows per page: 20 1 – 15 of 15

**SQL Query 2: Find the Count of Users in each Location for Each Member that have sent above some threshold (35) of messages to that member.**

This query finds the top users based on the number of messages they have sent. It uses a JOIN and GROUP BY with a COUNT() aggregation. This query would be useful in finding which member people like the most in each location.

```
SELECT u.Location, mb.Username AS MemberName, COUNT(u.Username) AS NumFans
FROM User u
JOIN Messages m ON u.Username = m.Username
JOIN Member mb ON m.MemberId = mb.MemberId
GROUP BY u.Location, mb.MemberId
HAVING COUNT(m.MessageId) >= 35
ORDER BY Location
LIMIT 15;
```

**Output:**

RESULTS

| Location | MemberName | NumFans |
|---|---|---|
| Belo Horizonte, Brazil | Yunjin | 37 |
| Buenos Aires, Argentina | Eunchae | 35 |
| Ho Chi Minh City, Vietnam | Sakura | 35 |
| Kigali, Rwanda | Chaewon | 36 |
| Kigali, Rwanda | Yunjin | 37 |
| Kigali, Rwanda | Kazuha | 38 |
| Lahore, Pakistan | Sakura | 40 |
| Lahore, Pakistan | Chaewon | 39 |
| Lahore, Pakistan | Kazuha | 37 |
| Lahore, Pakistan | Eunchae | 43 |
| Mexico City, Mexico | Chaewon | 35 |
| Mexico City, Mexico | Yunjin | 36 |
| São Luís, Brazil | Chaewon | 35 |
| São Luís, Brazil | Eunchae | 35 |
| Sydney, Australia | Sakura | 37 |

Rows per page: 20 1 – 15 of 15

**SQL Query 3: Get the seller for each bias with the most amount of items for that bias for events within the last year.**

This query is quite useful, as it shows us who the most important sellers are for a given member. It first computes ItemCounts

```
WITH ItemCounts AS (
 SELECT sp.Bias,s.Merchant AS Seller, COUNT(sp.ItemID) AS ItemCount
 FROM ShopPreferences sp
 JOIN Shop s ON sp.ItemID = s.ItemID
 JOIN Events e ON sp.EventName = e.EventName
 WHERE YEAR(e.StartDate) >= YEAR(CURRENT_DATE()) - 1
 GROUP BY sp.Bias, s.Merchant
)
SELECT Bias, Seller, ItemCount
FROM ItemCounts ic
WHERE (Bias, ItemCount) IN (
 SELECT Bias, MAX(ItemCount) FROM ItemCounts GROUP BY Bias
)
ORDER BY Bias;
```

**Output:**

RESULTS

| Bias | Seller | ItemCount |
|---|---|---|
| Chaewon | Martinez-Hernandez | 5060 |
| Eunchae | Williams-Sims | 5062 |
| Kazuha | Tapia Inc | 5575 |
| Sakura | Cameron-Guerrero | 6255 |
| Yunjin | Edwards-Livingston | 5396 |

**SQL Query 4: Rank the 5 LE SSERAFIM members by a popularity score, which is calculated by multiplying the number of messages the member receives from users whose bias is that member and the average length of the messages.**

This query is essential to LePhoning, as it shows which members are the most popular. It joins the Messages, ShopPreferences and Member tables, groups by the MemberName and finally shows the resulting popularity score using the COUNT and AVG aggregators.

```
SELECT m.Username AS MemberName, COUNT(msg.MessageID) * AVG(LENGTH(msg.Content)) AS
PopularityScore
FROM Messages msg
JOIN ShopPreferences s ON msg.Username = s.Username
JOIN Member m ON msg.MemberId = m.MemberId
WHERE s.Bias = m.Username
GROUP BY MemberName
ORDER BY PopularityScore DESC;
```

**Output:**

RESULTS

| MemberName | PopularityScore |
|---|---|
| Sakura | 18328.0000 |
| Chaewon | 18165.0000 |
| Yunjin | 16863.0000 |
| Eunchae | 16765.0000 |
| Kazuha | 15604.0000 |

# 5 Part 2: Advanced SQL Query Costs and Indexing

## Query 1

Without Indexing: cost = 92402.00

EXPLAIN

-> Limit: 15 row(s) (actual time=1530.177..1530.180 rows=15 loops=1) -> Sort: u.Username, e.EventName DESC, limit input to 15 row(s) per chunk (actual time=1530.176..1530.179 rows=15 loops=1) -> Stream results (cost=92402.00 rows=6000) (actual time=3.085..1529.893 rows=461 loops=1) -> Nested loop inner join (cost=92402.00 rows=6000) (actual time=3.083..1529.653 rows=461 loops=1) -> Nested loop inner join (cost=71402.00 rows=60000) (actual time=0.309..1370.174 rows=119771 loops=1) -> Table scan on e (cost=152.00 rows=1500) (actual time=0.050..0.790 rows=1500 loops=1) -> Filter: ((s.Bias = (select #2)) and (s.EventName = e.EventName)) (cost=7.50 rows=40) (actual time=0.102..0.907 rows=80 loops=1500) -> Index lookup on s using idx_shoppreferences_bias (Bias=(select #2)) (cost=7.50 rows=400) (actual time=0.096..0.446 rows=399 loops=1500) -> Select #2 (subquery in condition; dependent) -> Single-row index lookup on m using PRIMARY (MemberId=e.MemberId) (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=602138) -> Filter: ((u.Location = e.Location) and (timestampdiff(MONTH,e.StartDate,(select #3)) <= 6)) (cost=0.25 rows=0.1) (actual time=0.001..0.001 rows=0 loops=119771) -> Single-row index lookup on u using PRIMARY (Username=s.Username) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=119771) -> Select #3 (subquery in condition; dependent) -> Aggregate: max(m.`Time`) (cost=2.28 rows=1) (actual time=0.016..0.016 rows=1 loops=1202) -> Filter: (s.Bias = (select #4)) (cost=1.77 rows=5) (actual time=0.009..0.015 rows=5 loops=1202) -> Index lookup on m using Username (Username=u.Username) (cost=1.77 rows=5) (actual time=0.008..0.009 rows=5 loops=1202) -> Select #4 (subquery in condition; dependent) -> Single-row index lookup on m using PRIMARY (MemberId=e.MemberId) (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=6057)

Indexing Schema 1: Index User(Location): 17711.31

EXPLAIN

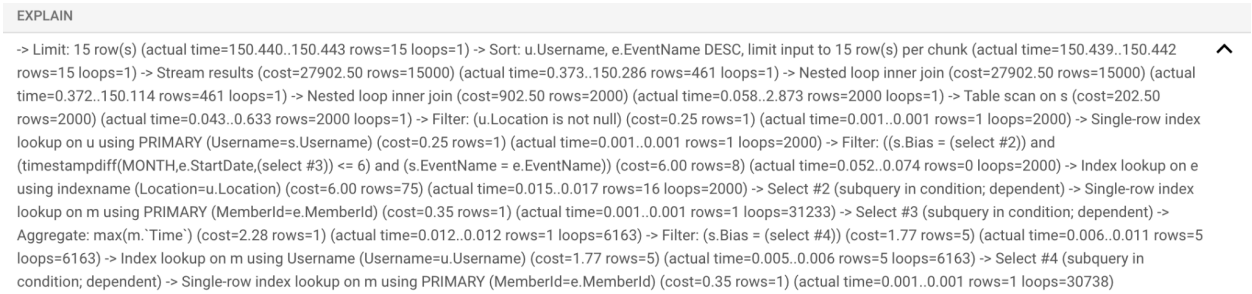
-> Limit: 15 row(s) (actual time=258.304..258.307 rows=15 loops=1) -> Sort: u.Username, e.EventName DESC, limit input to 15 row(s) per chunk (actual time=258.304..258.306 rows=15 loops=1) -> Stream results (cost=17711.31 rows=3362) (actual time=0.798..258.103 rows=461 loops=1) -> Nested loop inner join (cost=17711.31 rows=3362) (actual time=0.797..257.903 rows=461 loops=1) -> Nested loop inner join (cost=5943.84 rows=30612) (actual time=0.057..16.611 rows=31850 loops=1) -> Filter: (e.Location is not null) (cost=152.00 rows=1500) (actual time=0.035..0.637 rows=1500 loops=1) -> Table scan on e (cost=152.00 rows=1500) (actual time=0.034..0.520 rows=1500 loops=1) -> Covering index lookup on u using indexname2 (Location=e.Location) (cost=1.82 rows=20) (actual time=0.006..0.010 rows=21 loops=1500) -> Filter: ((s.Bias = (select #2)) and (timestampdiff(MONTH,e.StartDate,(select #3)) <= 6) and (s.EventName = e.EventName)) (cost=0.27 rows=0.1) (actual time=0.007..0.007 rows=0 loops=31850) -> Index lookup on s using indexname2 (Bias=(select #2), Username=u.Username) (cost=0.27 rows=1) (actual time=0.004..0.004 rows=0 loops=31850) -> Select #2 (subquery in condition; dependent) -> Single-row index lookup on m using PRIMARY (MemberId=e.MemberId) (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=69863) -> Select #3 (subquery in condition; dependent) -> Aggregate: max(m.`Time`) (cost=2.28 rows=1) (actual time=0.013..0.013 rows=1 loops=6163) -> Filter: (s.Bias = (select #4)) (cost=1.77 rows=5) (actual time=0.007..0.012 rows=5 loops=6163) -> Index lookup on m using Username (Username=u.Username) (cost=1.77 rows=5) (actual time=0.006..0.007 rows=5 loops=6163) -> Select #4 (subquery in condition; dependent) -> Single-row index lookup on m using PRIMARY (MemberId=e.MemberId) (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=30738)

Indexing Schema 2: Index Events(EventName): 72902.50

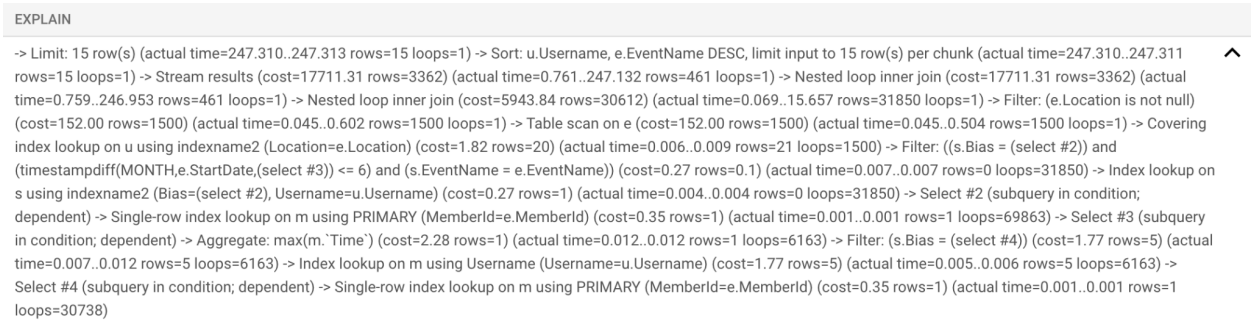
EXPLAIN

-> Limit: 15 row(s) (actual time=691.247..691.251 rows=15 loops=1) -> Sort: u.Username, e.EventName DESC, limit input to 15 row(s) per chunk (actual time=691.246..691.250 rows=15 loops=1) -> Stream results (cost=72902.50 rows=60000) (actual time=0.791..691.050 rows=461 loops=1) -> Nested loop inner join (cost=72902.50 rows=60000) (actual time=0.789..690.861 rows=461 loops=1) -> Nested loop inner join (cost=902.50 rows=2000) (actual time=0.078..3.083 rows=2000 loops=1) -> Filter: (s.EventName is not null) (cost=202.50 rows=2000) (actual time=0.063..0.907 rows=2000 loops=1) -> Table scan on s (cost=202.50 rows=2000) (actual time=0.063..0.753 rows=2000 loops=1) -> Single-row index lookup on u using PRIMARY (Username=s.Username) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=2000) -> Filter: ((e.Location = u.Location) and (s.Bias = (select #2)) and (timestampdiff(MONTH,e.StartDate,(select #3)) <= 6)) (cost=6.00 rows=30) (actual time=0.320..0.344 rows=0 loops=2000) -> Index lookup on e using indexname (EventName=s.EventName), with index condition: (s.EventName = e.EventName) (cost=6.00 rows=300) (actual time=0.097..0.311 rows=300 loops=2000) -> Select #2 (subquery in condition; dependent) -> Single-row index lookup on m using PRIMARY (MemberId=e.MemberId) (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=6170) -> Select #3 (subquery in condition; dependent) -> Aggregate: max(m.`Time`) (cost=2.28 rows=1) (actual time=0.013..0.013 rows=1 loops=1202) -> Filter: (s.Bias = (select #4)) (cost=1.77 rows=5) (actual time=0.007..0.012 rows=5 loops=1202) -> Index lookup on m using Username (Username=u.Username) (cost=1.77 rows=5) (actual time=0.006..0.007 rows=5 loops=1202) -> Select #4 (subquery in condition; dependent) -> Single-row index lookup on m using PRIMARY (MemberId=e.MemberId) (cost=0.35 rows=1) (actual time=0.001..0.001 rows=1 loops=6057)

Indexing Schema 3: Index Events(Location): 27902.50



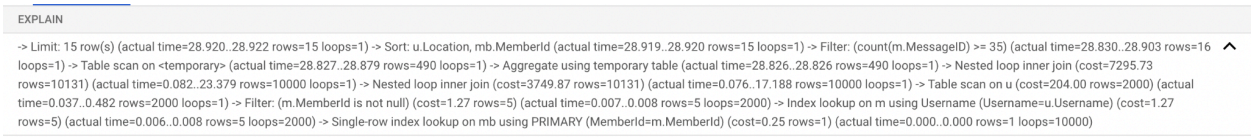

Indexing Schema 4: Index User(Location) and Events(EventName, Location): 17711.31



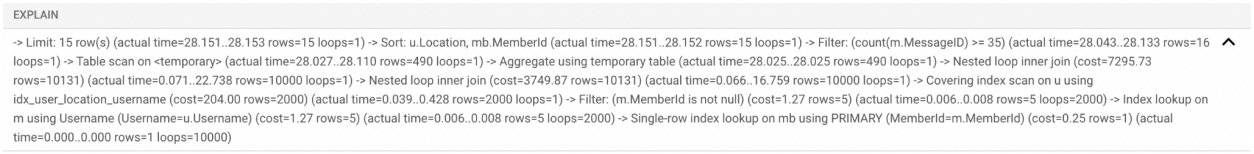

Since the JOIN clause only included primary keys, we focused on indexing non-key attributes that appeared in the WHERE and ORDER BY clauses. Indexing on User(Location) significantly brought the cost down from 92402.00 to 17711.31, while indexing on Events(EventName) and Events(Location) also brought down the cost to 72902.50 and 27902.50 respectively. These speedups could be because indexing improves the performance of the WHERE and ORDER BY clauses by taking advantage of locality. We conducted a final experiment including all three individual strategies and found a cost of 17711.31, which did not improve on the strategy of only indexing User(Location). Thus, the final index design we select is to index User(Location) (schema 4).

## Query 2

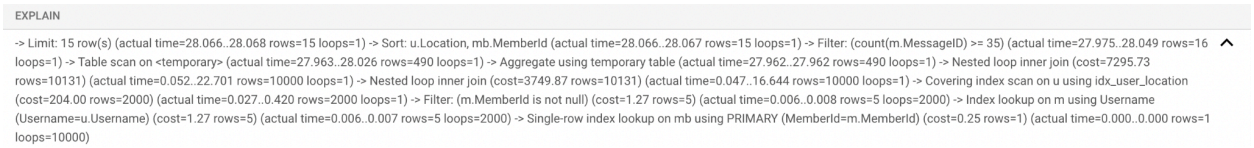
Without Indexing: cost = 7295.73



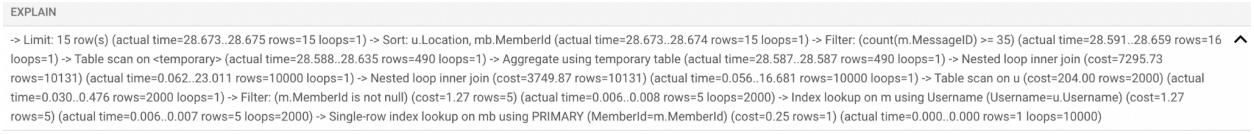

Indexing Schema 1: Index User(Location, Username): 7295.73




Indexing Schema 2: Index User(Location): 7295.73




Indexing Schema 3: Index User(Username): 7295.73




Because the query uses mainly primary keys of tables within the database, the indexing schemas do not seem to have an effect on the cost. Even indexing User.Location, which is one of our columns used in our GROUP BY that is not a primary key, the cost remains the same. Therefore, we think that no indexing for this query would be fine.

## Query 3

Without Indexing: cost = 330210.34

EXPLAIN

-> Sort: ic.Bias (cost=2.60..2.60 rows=0) (actual time=939.243..939.243 rows=5 loops=1) -> Filter: <in_optimizer>((ic.Bias,ic.ItemCount),(ic.Bias,ic.ItemCount) in (select #3)) (cost=2.50..2.50 rows=0) (actual time=939.150..939.232 rows=5 loops=1) -> Table scan on ic (cost=2.50..2.50 rows=0) (actual time=939.051..939.061 rows=100 loops=1) -> Materialize CTE ItemCountBySeller if needed (cost=0.00..0.00 rows=0) (actual time=939.050..939.050 rows=100 loops=1) -> Table scan on <temporary> (actual time=938.995..939.016 rows=100 loops=1) -> Aggregate using temporary table (actual time=938.990..938.990 rows=100 loops=1) -> Nested loop inner join (cost=330210.34 rows=300000) (actual time=0.595..179.680 rows=376728 loops=1) -> Filter: (sp.EventName = e.EventName) (cost=300160.34 rows=300000) (actual time=0.582..90.438 rows=376728 loops=1) -> Inner hash join (<hash>(sp.EventName)=<hash>(e.EventName)) (cost=300160.34 rows=300000) (actual time=0.580..25.352 rows=376728 loops=1) -> Table scan on sp (cost=0.02 rows=2000) (actual time=0.036..0.916 rows=2000 loops=1) -> Hash -> Filter: (year(e.StartDate) >= <cache>((year(curdate()) - 1))) (cost=152.00 rows=1500) (actual time=0.045..0.426 rows=943 loops=1) -> Table scan on e (cost=152.00 rows=1500) (actual time=0.041..0.323 rows=1500 loops=1) -> Single-row index lookup on s using PRIMARY (ItemID=sp.ItemID) (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=376728) -> Select #3 (subquery in condition; run only once) -> Filter: ((ic.Bias = `<materialized_subquery>`.Bias) and (ic.ItemCount = `<materialized_subquery>`.`MAX(ItemCount)`)) (cost=0.00..0.00 rows=0) (actual time=0.001..0.001 rows=0 loops=101) -> Limit: 1 row(s) (cost=0.00..0.00 rows=0) (actual time=0.001..0.001 rows=0 loops=101) -> Index lookup on <materialized_subquery> using <auto_distinct_key> (Bias=ic.Bias, MAX(ItemCount)=ic.ItemCount) (actual time=0.001..0.001 rows=0 loops=101) -> Materialize with deduplication (cost=0.00..0.00 rows=0) (actual time=0.070..0.070 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.062..0.063 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.062..0.062 rows=5 loops=1) -> Table scan on ItemCountBySeller (cost=2.50..2.50 rows=0) (actual time=0.004..0.014 rows=100 loops=1) -> Materialize CTE ItemCountBySeller if needed (query plan printed elsewhere) (cost=0.00..0.00 rows=0) (never executed)

Indexing Schema 1: Index Event(StartDate): 330210.34

EXPLAIN

-> Sort: ic.Bias (cost=2.60..2.60 rows=0) (actual time=939.476..939.477 rows=5 loops=1) -> Filter: <in_optimizer>((ic.Bias,ic.ItemCount),(ic.Bias,ic.ItemCount) in (select #3)) (cost=2.50..2.50 rows=0) (actual time=939.386..939.466 rows=5 loops=1) -> Table scan on ic (cost=2.50..2.50 rows=0) (actual time=939.291..939.301 rows=100 loops=1) -> Materialize CTE ItemCountBySeller if needed (cost=0.00..0.00 rows=0) (actual time=939.290..939.290 rows=100 loops=1) -> Table scan on <temporary> (actual time=939.234..939.255 rows=100 loops=1) -> Aggregate using temporary table (actual time=939.229..939.229 rows=100 loops=1) -> Nested loop inner join (cost=330210.34 rows=300000) (actual time=0.628..176.826 rows=376728 loops=1) -> Filter: (sp.EventName = e.EventName) (cost=300160.34 rows=300000) (actual time=0.617..90.326 rows=376728 loops=1) -> Inner hash join (<hash>(sp.EventName)=<hash>(e.EventName)) (cost=300160.34 rows=300000) (actual time=0.614..25.667 rows=376728 loops=1) -> Table scan on sp (cost=0.02 rows=2000) (actual time=0.037..0.891 rows=2000 loops=1) -> Hash -> Filter: (year(e.StartDate) >= <cache>((year(curdate()) - 1))) (cost=152.00 rows=1500) (actual time=0.053..0.456 rows=943 loops=1) -> Table scan on e (cost=152.00 rows=1500) (actual time=0.049..0.340 rows=1500 loops=1) -> Single-row index lookup on s using PRIMARY (ItemID=sp.ItemID) (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=376728) -> Select #3 (subquery in condition; run only once) -> Filter: ((ic.Bias = `<materialized_subquery>`.Bias) and (ic.ItemCount = `<materialized_subquery>`.`MAX(ItemCount)`)) (cost=0.00..0.00 rows=0) (actual time=0.001..0.001 rows=0 loops=101) -> Limit: 1 row(s) (cost=0.00..0.00 rows=0) (actual time=0.001..0.001 rows=0 loops=101) -> Index lookup on <materialized_subquery> using <auto_distinct_key> (Bias=ic.Bias, MAX(ItemCount)=ic.ItemCount) (actual time=0.001..0.001 rows=0 loops=101) -> Materialize with deduplication (cost=0.00..0.00 rows=0) (actual time=0.068..0.068 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.060..0.060 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.059..0.059 rows=5 loops=1) -> Table scan on ItemCountBySeller (cost=2.50..2.50 rows=0) (actual time=0.004..0.013 rows=100 loops=1) -> Materialize CTE ItemCountBySeller if needed (query plan printed elsewhere) (cost=0.00..0.00 rows=0) (never executed)

Indexing Schema 2: Index Event(StartDate) and Shop(Merchant): 330210.34

EXPLAIN

-> Limit: 15 row(s) (actual time=28.066..28.068 rows=15 loops=1) -> Sort: u.Location, mb.MemberId (actual time=28.066..28.067 rows=15 loops=1) -> Filter: (count(m.MessageID) >= 35) (actual time=27.975..28.049 rows=16 loops=1) -> Table scan on <temporary> (actual time=27.963..28.026 rows=490 loops=1) -> Aggregate using temporary table (actual time=27.962..27.962 rows=490 loops=1) -> Nested loop inner join (cost=7295.73 rows=10131) (actual time=0.052..22.701 rows=10000 loops=1) -> Nested loop inner join (cost=3749.87 rows=10131) (actual time=0.047..16.644 rows=10000 loops=1) -> Covering index scan on u using idx_user_location (cost=204.00 rows=2000) (actual time=0.027..0.420 rows=2000 loops=1) -> Filter: (m.MemberId is not null) (cost=1.27 rows=5) (actual time=0.006..0.008 rows=5 loops=2000) -> Index lookup on m using Username (Username=u.Username) (cost=1.27 rows=5) (actual time=0.006..0.007 rows=5 loops=2000) -> Single-row index lookup on mb using PRIMARY (MemberId=m.MemberId) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=10000)

Indexing Schema 3: Index Event(StartDate, EventName), Shop(Merchant), and ShopPreferences(Bias): 300927.60

EXPLAIN

-> Sort: ic.Bias (cost=2.60..2.60 rows=0) (actual time=341.252..341.252 rows=5 loops=1) -> Filter: <in_optimizer>((ic.Bias,ic.ItemCount),(ic.Bias,ic.ItemCount) in (select #3)) (cost=2.50..2.50 rows=0) (actual time=341.164..341.241 rows=5 loops=1) -> Table scan on ic (cost=2.50..2.50 rows=0) (actual time=341.060..341.070 rows=100 loops=1) -> Materialize CTE ItemCountBySeller if needed (cost=0.00..0.00 rows=0) (actual time=341.059..341.059 rows=100 loops=1) -> Table scan on <temporary> (actual time=341.018..341.029 rows=100 loops=1) -> Aggregate using temporary table (actual time=341.016..341.016 rows=100 loops=1) -> Filter: ((year(e.StartDate) >= <cache>((year(curdate()) - 1))) and (sp.EventName = e.EventName)) (cost=300927.60 rows=300000) (actual time=35.046..143.889 rows=376728 loops=1) -> Inner hash join (<hash>(sp.EventName)=<hash>(e.EventName)) (cost=300927.60 rows=300000) (actual time=2.741..54.186 rows=599261 loops=1) -> Covering index scan on e using idx_events_startdate_eventname (cost=0.02 rows=1500) (actual time=0.028..0.401 rows=1500 loops=1) -> Hash -> Nested loop inner join (cost=902.50 rows=2000) (actual time=0.058..2.350 rows=2000 loops=1) -> Table scan on sp (cost=202.50 rows=2000) (actual time=0.047..0.545 rows=2000 loops=1) -> Single-row index lookup on s using PRIMARY (ItemID=sp.ItemID) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=2000) -> Select #3 (subquery in condition; run only once) -> Filter: ((ic.Bias = `<materialized_subquery>`.Bias) and (ic.ItemCount = `<materialized_subquery>`.`MAX(ItemCount)`)) (cost=0.00..0.00 rows=0) (actual time=0.001..0.001 rows=0 loops=101) -> Limit: 1 row(s) (cost=0.00..0.00 rows=0) (actual time=0.001..0.001 rows=0 loops=101) -> Index lookup on <materialized_subquery> using <auto_distinct_key> (Bias=ic.Bias, MAX(ItemCount)=ic.ItemCount) (actual time=0.001..0.001 rows=0 loops=101) -> Materialize with deduplication (cost=0.00..0.00 rows=0) (actual time=0.074..0.074 rows=5 loops=1) -> Table scan on <temporary> (actual time=0.066..0.067 rows=5 loops=1) -> Aggregate using temporary table (actual time=0.066..0.066 rows=5 loops=1) -> Table scan on ItemCountBySeller (cost=2.50..2.50 rows=0) (actual time=0.005..0.016 rows=100 loops=1) -> Materialize CTE ItemCountBySeller if needed (query plan printed elsewhere) (cost=0.00..0.00 rows=0) (never executed)

We want to index columns that we use in JOINS, WHERE clauses, and GROUP BYs that are not primary keys. So, we tried different combinations of indices of the relevant columns in our query that were not primary keys in tables. While most of them did not yield any improvement over having no indices at all, including all of them (schema 3) as indices yielded a small improvement in the total cost of the query, so we will include those indices to improve the performance of this query. This might be because EventName is being used as part of the JOIN and

indexing StartDate might make the WHERE clause easier by getting values within a range (since the >= DATE()-1 means that it needs to be within a range), so a Btree format might be have been useful in improving performance.

## Query 4

Without Indexing: cost = 2776.19

EXPLAIN

-> Sort: PopularityScore DESC (actual time=16.867..16.867 rows=5 loops=1) -> Table scan on <temporary> (actual time=16.850..16.851 rows=5 loops=1) -> Aggregate using temporary table (actual time=16.849..16.849 rows=5 loops=1) -> Nested loop inner join (cost=2776.19 rows=1013) (actual time=0.079..15.855 rows=1952 loops=1) -> Inner hash join (cp.Bias = m.Username) (cost=1003.26 rows=1000) (actual time=0.051..0.953 rows=2000 loops=1) -> Table scan on cp (cost=4.50 rows=2000) (actual time=0.020..0.490 rows=2000 loops=1) -> Hash -> Table scan on m (cost=0.75 rows=5) (actual time=0.018..0.021 rows=5 loops=1) -> Filter: (msg.MemberId = m.MemberId) (cost=1.27 rows=1) (actual time=0.006..0.007 rows=1 loops=2000) -> Index lookup on msg using Username (Username=cp.Username) (cost=1.27 rows=5) (actual time=0.006..0.007 rows=5 loops=2000)

Indexing Schema 1: Index ShopPreferences(Bias): 3791.20

EXPLAIN

-> Sort: PopularityScore DESC (actual time=17.093..17.094 rows=5 loops=1) -> Table scan on <temporary> (actual time=17.076..17.077 rows=5 loops=1) -> Aggregate using temporary table (actual time=17.075..17.075 rows=5 loops=1) -> Nested loop inner join (cost=3791.20 rows=2026) (actual time=0.086..16.132 rows=1952 loops=1) -> Nested loop inner join (cost=245.34 rows=2000) (actual time=0.062..0.733 rows=2000 loops=1) -> Filter: (m.Username is not null) (cost=0.75 rows=5) (actual time=0.027..0.035 rows=5 loops=1) -> Table scan on m (cost=0.75 rows=5) (actual time=0.026..0.032 rows=5 loops=1) -> Covering index lookup on s using indexname (Bias=m.Username) (cost=16.92 rows=400) (actual time=0.025..0.122 rows=400 loops=5) -> Filter: (msg.MemberId = m.MemberId) (cost=1.27 rows=1) (actual time=0.007..0.008 rows=1 loops=2000) -> Index lookup on msg using Username (Username=s.Username) (cost=1.27 rows=5) (actual time=0.006..0.007 rows=5 loops=2000)

Indexing Schema 2: Index Member(Username): 2776.19

EXPLAIN

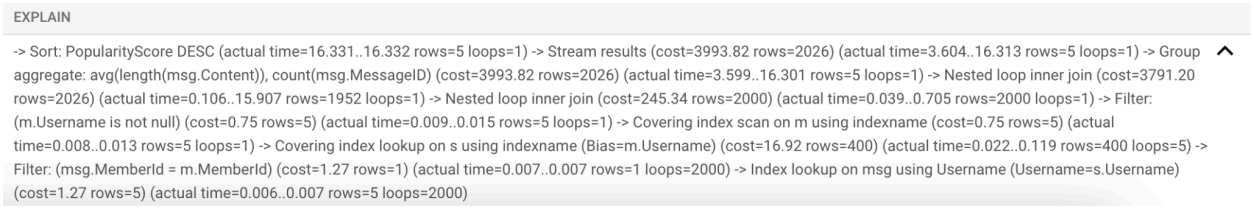
-> Sort: PopularityScore DESC (actual time=16.596..16.597 rows=5 loops=1) -> Table scan on <temporary> (actual time=16.580..16.581 rows=5 loops=1) -> Aggregate using temporary table (actual time=16.579..16.579 rows=5 loops=1) -> Nested loop inner join (cost=2776.19 rows=1013) (actual time=0.060..15.602 rows=1952 loops=1) -> Inner hash join (s.Bias = m.Username) (cost=1003.26 rows=1000) (actual time=0.038..0.918 rows=2000 loops=1) -> Table scan on s (cost=4.50 rows=2000) (actual time=0.019..0.473 rows=2000 loops=1) -> Hash -> Covering index scan on m using indexname (cost=0.75 rows=5) (actual time=0.009..0.010 rows=5 loops=1) -> Filter: (msg.MemberId = m.MemberId) (cost=1.27 rows=1) (actual time=0.006..0.007 rows=1 loops=2000) -> Index lookup on msg using Username (Username=s.Username) (cost=1.27 rows=5) (actual time=0.006..0.007 rows=5 loops=2000)

Indexing Schema 3: Index Messages(Content): 2776.19

EXPLAIN

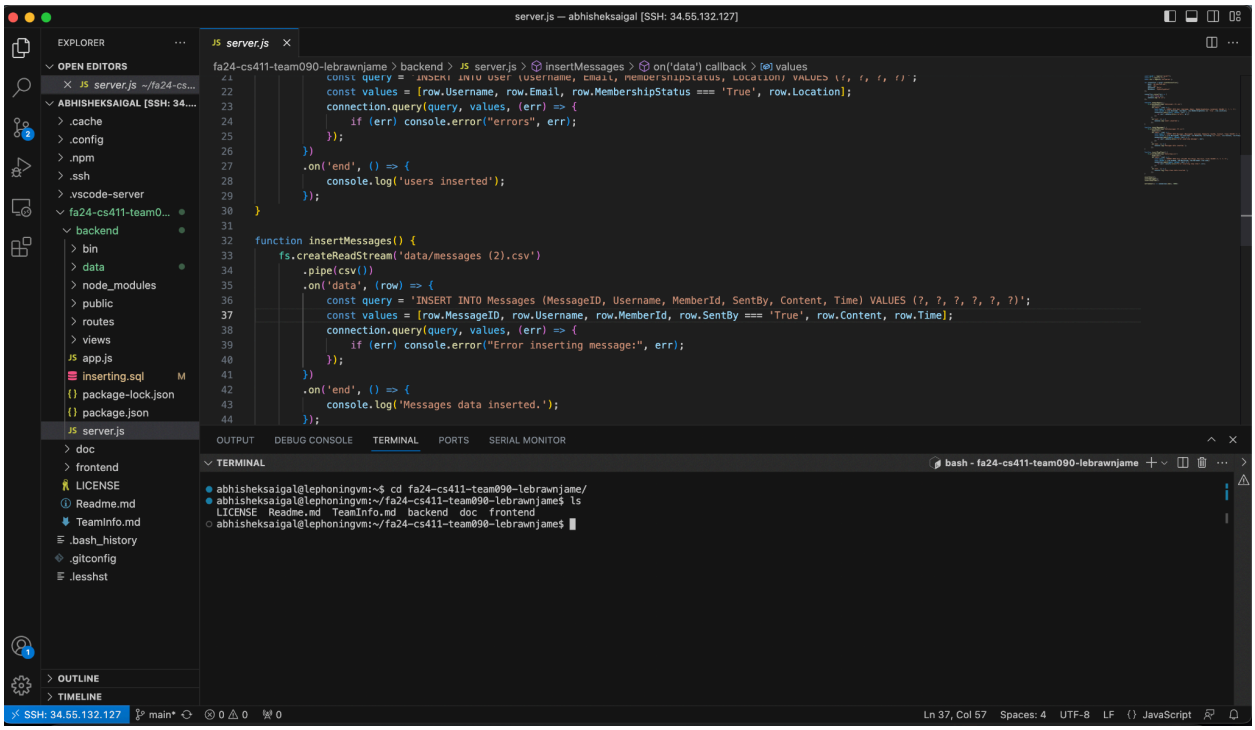
-> Sort: PopularityScore DESC (actual time=17.052..17.053 rows=5 loops=1) -> Table scan on <temporary> (actual time=17.034..17.035 rows=5 loops=1) -> Aggregate using temporary table (actual time=17.033..17.033 rows=5 loops=1) -> Nested loop inner join (cost=2776.19 rows=1013) (actual time=0.091..16.028 rows=1952 loops=1) -> Inner hash join (s.Bias = m.Username) (cost=1003.26 rows=1000) (actual time=0.050..0.947 rows=2000 loops=1) -> Table scan on s (cost=4.50 rows=2000) (actual time=0.020..0.490 rows=2000 loops=1) -> Hash -> Table scan on m (cost=0.75 rows=5) (actual time=0.018..0.021 rows=5 loops=1) -> Filter: (msg.MemberId = m.MemberId) (cost=1.27 rows=1) (actual time=0.007..0.007 rows=1 loops=2000) -> Index lookup on msg using Username (Username=s.Username) (cost=1.27 rows=5) (actual time=0.006..0.007 rows=5 loops=2000)

Indexing Schema 4: Index ShopPreferences(Bias), Member(Username) and Index Messages(Content): 3993.82




We tried indexing different non-key attributes appearing in the WHERE, GROUP BY and LENGTH clauses but found that none of the indexing strategies improved the query performance. Indexing Bias actually worsened the performance; this could be because the size of the relation was not big enough to justify the overhead of the indexing data structure. Furthermore, the Member relation only contains 5 lines, so indexing the Username did not improve performance. Finally, indexing Content in the LENGTH clause did not improve performance because the attribute was not in a WHERE or GROUP BY clause. Using all three indexing strategies at once yielded the highest cost as the overhead was highest. Thus, we conclude that not indexing for this query is the best strategy.

Screenshot showing our workspace (SSH'd into the VM on GCP):




Screenshot showing our GCP Cloud SQL instance with the cardinalities of our tables: