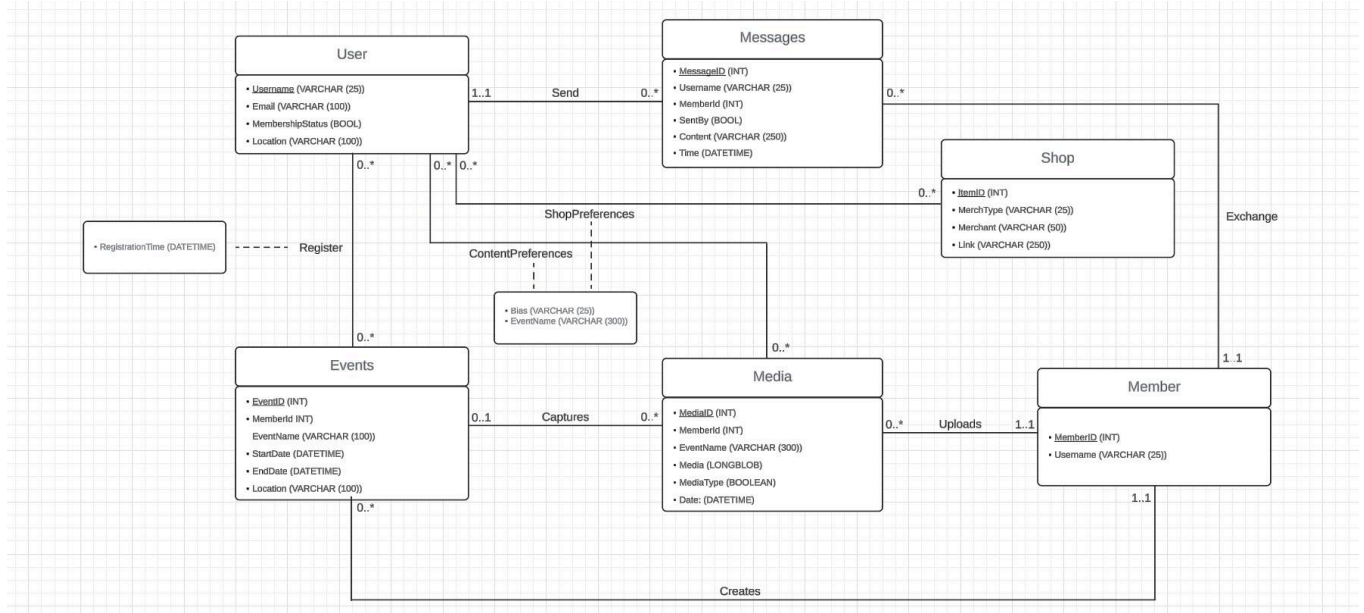


# LePhoning

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



## 2 Part 1: DDL Commands

```

CREATE TABLE User (
    Username VARCHAR(25) PRIMARY KEY,
    Email VARCHAR(100),
    MembershipStatus BOOLEAN,
    Location VARCHAR(100)
);

```

```

CREATE TABLE Messages (
    MessageID INT PRIMARY KEY,
    Username VARCHAR(25),
    MemberID INT,
    SentBy BOOLEAN,
    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 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 SET CASCADE  
  
);
```

```
CREATE TABLE 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 SET CASCADE,
    );

CREATE TABLE Shop (
    ItemID INT PRIMARY KEY,
    MerchType VARCHAR(25),
    Merchant VARCHAR(50),
    Link VARCHAR(250)
);

CREATE TABLE Member (
    MemberId INT PRIMARY KEY,
    Username VARCHAR(25)
);

CREATE TABLE 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 ContentPreferences (
    Username VARCHAR(25),
    MediaId INT,
    Bias VARCHAR(25),
    EventName VARCHAR(300),
    PRIMARY KEY (Username, MediaId),
    FOREIGN KEY (Username) REFERENCES User(Username),

```

```

        FOREIGN KEY (EventName) REFERENCES Events(EventName)
    );

CREATE TABLE 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: Find Users with Cross-Location Preferences for Shop Items

```

SELECT  u.Username,    u.Location  AS   UserLocation,    s.MerchType,    s.Merchant,
COUNT(sp.ItemID) AS CrossLocationPreferences
FROM User u
JOIN ShopPreferences sp ON u.Username = sp.Username
JOIN Shop s ON sp.ItemID = s.ItemID
WHERE s.Merchant IN (
    SELECT DISTINCT Merchant
    FROM Shop
    WHERE Location IS NOT NULL AND Location <> u.Location
)
GROUP BY u.Username, u.Location, s.MerchType, s.Merchant
HAVING COUNT(sp.ItemID) > 0
ORDER BY CrossLocationPreferences DESC
LIMIT 15;

```

Output:

RESULTS				
Username	UserLocation	MerchType	Merchant	CrossLocationPreferences
 No rows to display				

Note: The query yielded no results.

### SQL Query 2: Retrieve the Top 15 Users with the Most Messages Sent

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 determining user engagement with the app.

```
SELECT u.Username, COUNT(m.MessageID) AS MessageCount
FROM User u
JOIN Messages m ON u.Username = m.Username
GROUP BY u.Username
ORDER BY MessageCount DESC
LIMIT 15;
```

Output:

RESULTS			
Username	MessageCount		
adam64	1		
aedwards	1		
alanwilson	1		
alexandervasquez	1		
alice86	1		
aliciacain	1		
allisonjohnson	1		
amyphillips	1		
andrea61	1		
andrea Herrera	1		
andraperez	1		
angelagardner	1		
anthony01	1		
antoniomiller	1		
april92	1		

Rows per page: 20
1 – 15 of 15
|< < > >|

SQL Query 3: Find the Most Popular Merch Types by Event Preference

This query identifies the most popular merch types among users who attended specific events within the last year. It uses JOIN and GROUP BY to filter for the top merch types based on user preferences.

```
SELECT s.MerchType, COUNT(sp.ItemID) AS PreferenceCount
FROM Shop s
JOIN ShopPreferences sp ON s.ItemID = sp.ItemID
GROUP BY s.MerchType
ORDER BY PreferenceCount DESC
LIMIT 15;
```

RESULTS	
MerchType	PreferenceCount
T-shirt	222
Sticker	212
Poster	209
Keychain	183
Album	174

Without indexing:

-> Limit: 15 row(s) (actual time=1.294..1.294 rows=5 loops=1) -> Sort: PreferenceCount DESC, limit input to 15 row(s) per chunk (actual time=1.293..1.293 rows=5 loops=1) -> Table scan on <temporary> (actual time=1.283..1.283 rows=5 loops=1) -> Aggregate using temporary table (actual time=1.282..1.282 rows=5 loops=1) -> Nested loop inner join (cost=455.75 rows=1000) (actual time=0.041..0.918 rows=1000 loops=1) -> Covering index scan on sp using ItemID (cost=105.75 rows=1000) (actual time=0.029..0.212 rows=1000 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=1000)

Cost: 455.75

Indexing Bias in ShopPreferences:

-> Limit: 15 row(s) (actual time=1.289..1.289 rows=5 loops=1) -> Sort: PreferenceCount DESC, limit input to 15 row(s) per chunk (actual time=1.288..1.288 rows=5 loops=1) -> Table scan on <temporary> (actual time=1.278..1.278 rows=5 loops=1) -> Aggregate using temporary table (actual time=1.276..1.276 rows=5 loops=1) -> Nested loop inner join (cost=455.75 rows=1000) (actual time=0.042..0.898 rows=1000 loops=1) -> Covering index scan on sp using ItemID (cost=105.75 rows=1000) (actual time=0.031..0.211 rows=1000 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=1000)

Cost: 455.75

Indexing MerchType in Shop:

-> Limit: 15 row(s) (actual time=1.237..1.237 rows=5 loops=1) -> Sort: PreferenceCount DESC, limit input to 15 row(s) per chunk (actual time=1.236..1.237 rows=5 loops=1) -> Table scan on <temporary> (actual time=1.227..1.228 rows=5 loops=1) -> Aggregate using temporary table (actual time=1.226..1.226 rows=5 loops=1) -> Nested loop inner join (cost=455.75 rows=1000) (actual time=0.025..0.876 rows=1000 loops=1) -> Covering index scan on sp using ItemID (cost=105.75 rows=1000) (actual time=0.017..0.193 rows=1000 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=1000)

Cost: 455.75

Indexing had no effect in this case. This might be because the attributes in the query are primary attributes, so indexing on the other attributes will make no difference. We thus conclude that no indexing for this query is the best approach.

#### SQL Query 4: Find Users with High Engagement Across Events and Media, Grouped by Bias

This query retrieves the top users who have registered for the most events and have a preference for their favorite LE SSERAFIM member (bias) in their shop preferences. The query uses a subquery to calculate each user's event registration count, then filters for those who have set a particular member as their bias. We use a subquery and JOINS for this query.

```
SELECT u.Username, EventCount, sp.Bias
FROM (
    SELECT r.Username, COUNT(r.EventID) AS EventCount
    FROM Register r
    GROUP BY r.Username
) AS UserEventCounts
JOIN ShopPreferences sp ON UserEventCounts.Username = sp.Username
JOIN User u ON u.Username = sp.Username
WHERE sp.Bias = 'Chaewon'
ORDER BY EventCount DESC
LIMIT 15;
```



MerchType	ItemVariety
T-shirt	213
Keychain	208
Sticker	198
Poster	193
Album	188



## Query 1:

### Cost:

Without Indexing: 118.05

EXPLAIN
-> Limit: 15 row(s) (actual time=0.588..0.590 rows=15 loops=1) -> Sort: MessageCount DESC, limit input to 15 row(s) per chunk (actual time=0.588..0.588 rows=15 loops=1) -> Table scan on <temporary> (actual time=0.542..0.564 rows=259 loops=1) -> Aggregate using temporary table (actual time=0.540..0.540 rows=259 loops=1) -> Nested loop inner join (cost=118.05 rows=259) (actual time=0.042..0.412 rows=259 loops=1) -> Filter: (m.Username is not null) (cost=27.40 rows=259) (actual time=0.029..0.087 rows=259 loops=1) -> Covering index scan on m using Username (cost=27.40 rows=259) (actual time=0.029..0.072 rows=259 loops=1) -> Single-row covering index lookup on u using PRIMARY (Username=m.Username) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=259)

Indexing Schema 1: Index message.Username: 118.05

EXPLAIN
-> Limit: 15 row(s) (actual time=0.631..0.633 rows=15 loops=1) -> Sort: MessageCount DESC, limit input to 15 row(s) per chunk (actual time=0.631..0.631 rows=15 loops=1) -> Table scan on <temporary> (actual time=0.584..0.605 rows=259 loops=1) -> Aggregate using temporary table (actual time=0.583..0.583 rows=259 loops=1) -> Nested loop inner join (cost=118.05 rows=259) (actual time=0.064..0.445 rows=259 loops=1) -> Filter: (m.Username is not null) (cost=27.40 rows=259) (actual time=0.042..0.099 rows=259 loops=1) -> Covering index scan on m using idx_messages_username (cost=27.40 rows=259) (actual time=0.041..0.085 rows=259 loops=1) -> Single-row covering index lookup on u using PRIMARY (Username=m.Username) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=259)

Indexing Schema 2: Index user.Username: 118.05

RESULTS
EXPLAIN
-> Limit: 15 row(s) (actual time=0.574..0.589 rows=15 loops=1) -> Sort: MessageCount DESC, limit input to 15 row(s) per chunk (actual time=0.574..0.588 rows=15 loops=1) -> Table scan on <temporary> (actual time=0.529..0.551 rows=259 loops=1) -> Aggregate using temporary table (actual time=0.528..0.528 rows=259 loops=1) -> Nested loop inner join (cost=118.05 rows=259) (actual time=0.047..0.402 rows=259 loops=1) -> Filter: (m.Username is not null) (cost=27.40 rows=259) (actual time=0.033..0.089 rows=259 loops=1) -> Covering index scan on m using idx_messages_username (cost=27.40 rows=259) (actual time=0.032..0.075 rows=259 loops=1) -> Single-row covering index lookup on u using PRIMARY (Username=m.Username) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=259)

Indexing Schema 3: Index message.Username, message.MemberId:

EXPLAIN
-> Limit: 15 row(s) (actual time=0.580..0.582 rows=15 loops=1) -> Sort: MessageCount DESC, limit input to 15 row(s) per chunk (actual time=0.580..0.581 rows=15 loops=1) -> Table scan on <temporary> (actual time=0.535..0.557 rows=259 loops=1) -> Aggregate using temporary table (actual time=0.534..0.534 rows=259 loops=1) -> Nested loop inner join (cost=118.05 rows=259) (actual time=0.046..0.396 rows=259 loops=1) -> Filter: (m.Username is not null) (cost=27.40 rows=259) (actual time=0.032..0.089 rows=259 loops=1) -> Covering index scan on m using idx_messages_username (cost=27.40 rows=259) (actual time=0.031..0.074 rows=259 loops=1) -> Single-row covering index lookup on u using PRIMARY (Username=m.Username) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=259)

Because the query is not very complicated in terms of what it joins and groups on, the indexing has the same cost regardless of the indexing used. Even when indexing by message.Username and message.MemberId, the cost is the same, which means that we can run this query without indexing in order to get .