Final Report

1. Please list out changes in the directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).

In our stage 1 proposal, we aimed to create a platform that joins several online event ticket marketplaces' offerings into a unified shopping experience. However, based on the feedback from stage 1, we adjusted it to focus on certain events like sports and concerts and rather than searching online, tickets are posted to the platform. In addition, rather than implementing CRUD operations from a single kind of account, we split the CRUD operations between typical users and administrations, where administrators can perform more impactful operations like deleting users. We simplified our proposed recommendation system into stored procedures that recommend based on commonly wishlisted events or events in top cities. We also added a notification system to alert users when more tickets become available for wishlisted events.

2. What our application achieved or failed to achieve regarding its usefulness:

Regarding usefulness, our application has succeeded in our goal of allowing users to browse, compare, and wishlist event tickets using downloaded data from various online ticket vendors. We successfully implemented a combined database of event tickets, search and sort functionalities on events, a wishlist system including notifications when more tickets are available for wishlisted events, two user types to restrict certain functionalities to administrators, and a recommendation system based on other users' wishlist data. However, we were unable to implement the ticket cost heat map visualization nor the analytics dashboard creative components we had originally proposed.

3. Discuss if you changed the schema or source of the data for your application:

As proposed we gathered data from StubHub, ETickets, Sports Illustrated Tickets, Tickets.com, FeverUp, Vivid Seats, and MainLineTix as provided for free on the AWS marketplace then downloaded them as json files, which we then inserted into our

combined database. User accounts were artificially generated to simulate a large user base. We also added a “Notifications” table as part of an advanced feature. The table stores the username and event to be notified to a user based on their activity.

```
CREATE TABLE Notifications (
    notification_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(100),
    event_title VARCHAR(250),
    message TEXT,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    is_read BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (username) REFERENCES Users(username) ON DELETE CASCADE ON
UPDATE CASCADE
);
```

4. What changes were made to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?

In our stage 2 submission, we initially established a relationship between users and events as a Wishlist entity. However, a relation table was more appropriate as wishlist entries could be uniquely identified via their constituent foreign keys. This design was more suitable because it better adhered to ER/UML design principles and reduced the redundancy of a wishlist primary key. The most notable change with Stage 4 is the addition of a Notifications table for implementing user alerts about wishlisted events.

5. Discuss what functionalities you added or removed. Why?

Our original creative component of a heat map visualization of ticket prices as well as a proposed analytics dashboard were removed as other advanced functionalities, including our newly implemented notifications system, proved more applicable to our application’s purpose and core functionality of wish lists.

The notifications system was added to replace the heat map visualization creative component. The system alerts users when relevant changes are made to events they have wishlisted, such as if an event is canceled or if more tickets become available.

Although different user levels were always considered in the structure of our database, in stage 4 we added functionality to these levels by restricting higher impact operations, such as deleting a user, to users with admin privileges only. This functionality made our

application more realistic as regular users should not be able to access or delete other users' data, but application administrators should be able to make those sorts of changes through their login. For those administrators, we also added a user search bar feature to make finding specific accounts easier.

An alteration of our proposed recommendation system was implemented to recommend trending events based on user engagement as well as events from major cities both via stored procedures. To complement this, we also implemented advanced event search and filtering functionalities.

6. Explain how you think your advanced database programs complement your application: (SEE ADVANCED DATABASE FEATURES BELOW REPORT)

Our database programs complement our application in many ways. Our transactions are used to ensure that deleting an event both logs a notification and removes the event and to ensure that wishlisting an event both logs a notification and adds the proper wishlist entry, both with isolation from other processes and the ability to rollback all queries within simultaneously. Our stored procedures "GetPopularEvents" and "GetTopCitiesEvents" allow us to easily call our most common database queries throughout the application while also centralizing them such that any changes to one of these common functionalities need only be made once in redefining the stored procedure rather than many times throughout the code base. Our trigger "NotifyOnTicketQuantityIncrease" complements our application because it allows users who have wishlisted an event to be instantly notified (new entry in the Notifications table) when more tickets are available. For our constraints, we did not implement anything advanced and simply implemented our primary keys and foreign keys to ensure that all of our tables hold proper connections to one another throughout our many CRUD and advanced operations.

7. Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project:

Mathew: A technical problem our team encountered was poor code organization during the development of Stage 4, Part 1. When transitioning to Part 2, we had to spend time reorganizing and understanding the differences between the two parts. It also led to

errors on functions that had already been developed. This time could have been used more effectively if we had prioritized better organization from the start. I would recommend organizing and testing code as future groups develop their programs.

Nathan: One technical problem that the team encountered was that we started all of Stage 2: Conceptual and Logical Database Design before finalizing our ER/UML diagram, so we had to redo many later portions multiple times as the diagram changed. This caused a lot of unnecessarily wasted time and work, so I would definitely recommend first fully understanding ER/UML diagrams, second creating and finalizing your ER/UML diagram, and finally working on the rest of Stage 2.

Ajay: A technical problem that we encountered was when we were trying to come up with advanced queries for our application. During stage 3, we came up with such queries which efficiently reduced the cost and time to execute them. We had to modify our original advanced queries because it did not reduce the cost after indexing. But, when seen from application utility perspective, the new queries were less efficient than the older ones. Thus, I would recommend thinking about the advanced features along with the indexing strategies during initial stages of the project.

Kevin: One technical problem we encountered was choosing the appropriate dataset for our project. Initially, we planned to use an API, but we found that using an API was too ambitious given our project's scope and timeline. This forced us to pivot and select a static dataset instead, which led to a substantial change in our project's direction and overall idea. For future teams, I recommend thoroughly evaluating data sources early, asking questions about the datasets you are considering to the teaching assistants, and being flexible in planning. Always have backup data sources and document your decisions to help future projects.

8. Are there other things that changed comparing the final application with the original proposal?

As previously mentioned in response to question 1, the main difference between the final application and the original proposal is that we specialized from general marketplaces like Ebay and Amazon to event ticket marketplaces. As previously mentioned, we also restricted some more impactful CRUD operations, such as deleting a user, to users with admin privileges. We also added a Notifications table to the database to represent the notifications we would send to users if email notification functionality were added. Currently, notification only appears when the user logs in.

9. Describe future work that you think, other than the interface, that the application can improve on:

Beyond the interface, our code organization could have been improved. Efforts were made to address this between Stage 4 Part 1 and Part 2. For example, we reorganized the program by separating the code for events, index, login, tickets, and the wish list into distinct files within the project structure. While significant improvements have been made, there are some remaining parts of the program that could be more organized. However, due to time constrictions, the structure of our code should be sufficient.

10. Describe the final division of labor and how well you managed teamwork:

Our team division and collaboration were quite effective. Nathan and Ajay led the backend efforts by giving functions to different parts of the interface like the side panel, parts of pages, and buttons. Kevin and Mathew focused on the frontend by developing and designing several aspects of the interface such as the pages, side panel, colors on each page, and overall design of the platform. After our work, we all collaborated on the project report. Our teamwork was productive, with each individual putting in their best effort and maintaining good communication throughout the project.

Constraints:

```
CREATE TABLE Events (event_title VARCHAR(250) PRIMARY KEY, event_url VARCHAR(250),
datetime_local DATETIME, location_name VARCHAR(250), promoter_name VARCHAR(100),
username VARCHAR(100), FOREIGN KEY (location_name) REFERENCES Locations(location_name)
ON DELETE CASCADE ON UPDATE CASCADE, FOREIGN KEY (promoter_name) REFERENCES
Promoter(promoter_name) ON DELETE CASCADE ON UPDATE CASCADE);

CREATE TABLE Tickets (ticket_id VARCHAR(100) PRIMARY KEY, event_title VARCHAR(250),
ticket_price DECIMAL(10,2), total_price DECIMAL(10,2), fee DECIMAL(10,2), full_section
VARCHAR(100), section VARCHAR(100), row_num VARCHAR(100), quantity INT, FOREIGN KEY
(event_title) REFERENCES Events(event_title) ON DELETE CASCADE ON UPDATE CASCADE));

CREATE TABLE WishList (username VARCHAR(100), event_title VARCHAR(250), wishlist_date
DATETIME, PRIMARY KEY (username, event_title), FOREIGN KEY (username) REFERENCES
```

```
Users(username) ON DELETE CASCADE ON UPDATE CASCADE, FOREIGN KEY (event_title)
REFERENCES Events(event_title) ON DELETE CASCADE ON UPDATE CASCADE);
```

SPs:

```
CREATE PROCEDURE GetPopularEvents(IN top_count INT)
BEGIN
    -- Create a temporary table to store event popularity
    CREATE TEMPORARY TABLE IF NOT EXISTS EventPopularity (
        event_title VARCHAR(250),
        ticket_count INT,
        wishlist_count INT,
        total_score INT
    );

    -- Insert data into the temporary table
    INSERT INTO EventPopularity (event_title, ticket_count, wishlist_count)
    SELECT
        e.event_title,
        COALESCE(t.ticket_count, 0) AS ticket_count,
        COALESCE(w.wishlist_count, 0) AS wishlist_count
    FROM Events e
    LEFT JOIN (
        SELECT event_title, SUM(quantity) AS ticket_count
        FROM Tickets
        GROUP BY event_title
    ) t ON e.event_title = t.event_title
    LEFT JOIN (
        SELECT event_title, COUNT(*) AS wishlist_count
        FROM WishList
        GROUP BY event_title
    ) w ON e.event_title = w.event_title;

    -- Calculate total score
    UPDATE EventPopularity
    SET total_score = (ticket_count * 2) + wishlist_count;

    -- Select the top events
    SELECT
        ep.event_title,
        ep.ticket_count,
```

```
        ep.wishlist_count,
        ep.total_score,
        e.datetime_local,
        e.location_name,
        l.city,
        e.promoter_name
    FROM EventPopularity ep
    JOIN Events e ON ep.event_title = e.event_title
    JOIN Locations l on e.location_name = l.location_name
    ORDER BY ep.total_score DESC
    LIMIT top_count;

    -- Clean up
    DROP TEMPORARY TABLE IF EXISTS EventPopularity;
END //

DELIMITER ;
```

```
DELIMITER //
CREATE PROCEDURE GetTopCitiesEvents(IN city_count INT)
BEGIN
    SELECT DISTINCT event_title, datetime_local, location_name, promoter_name, city
    FROM Tickets
    NATURAL JOIN Events
    NATURAL JOIN Locations
    WHERE city IN (
        SELECT city
        FROM (
            SELECT city, COUNT(event_title) AS event_count
            FROM Locations
            NATURAL JOIN Events
            GROUP BY city
            ORDER BY event_count DESC
            LIMIT city_count
        ) AS top_cities
    );
END //
DELIMITER ;
```

Trigger:

```
CREATE TRIGGER NotifyOnTicketQuantityIncrease
AFTER UPDATE ON Tickets
FOR EACH ROW
BEGIN
    IF NEW.quantity > OLD.quantity THEN
        INSERT INTO Notifications (username, event_title, message, created_at,
is_read)
        SELECT
            u.username,
            NEW.event_title,
            CONCAT('More tickets are now available for ', NEW.event_title, '!'),
            NOW(),
            FALSE
        FROM
            Users u
        WHERE
            u.username IN (
                SELECT username
                FROM WishList
                WHERE event_title = NEW.event_title
            )
        ON DUPLICATE KEY UPDATE
            created_at = VALUES(created_at),
            is_read = FALSE,
            message = VALUES(message);
    END IF;
END;
```

Transactions:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;

           -- Insert the new wishlist entry
           INSERT INTO WishList (username, event_title, wishlist_date)
           VALUES (%s, %s, %s)
```

```
            ON DUPLICATE KEY UPDATE wishlist_date = VALUES(wishlist_date);

            INSERT INTO Notifications (username, event_title, message)
            SELECT
                %s,
                e.event_title,
                CONCAT('You have added "', e.event_title, '" to your wishlist. ',
                    ' at ', e.location_name, ' (', l.city, ', ', l.state, '). ',
                    'Promoted by ', e.promoter_name, '. ',
                    'Currently, ', COUNT(w.username), ' user(s) have wishlisted this
event, ',
                    'including you. ')
            FROM Events e
            JOIN Locations l ON e.location_name = l.location_name
            LEFT JOIN WishList w ON e.event_title = w.event_title
            WHERE e.event_title = %s
            GROUP BY e.event_title, e.location_name, l.city, l.state, e.promoter_name;

COMMIT;
```

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
INSERT INTO Notifications (username, event_title, message)
SELECT
    w.username,
    e.event_title,
    CONCAT('The event "', e.event_title,
           ' at ', l.location_name, ' (', l.city, ', ', l.state, ') ',
           'has been cancelled. ',
           'There are ', other_events.event_count, ' other events happening in ',
l.city, '. ',
           'Check them out!') AS message
FROM WishList w
JOIN Events e ON w.event_title = e.event_title
JOIN Locations l ON e.location_name = l.location_name
JOIN (
    SELECT city, COUNT(*) as event_count
    FROM Events natural join Locations
    WHERE event_title != %s
    GROUP BY city
) other_events ON l.city = other_events.city
```

```
WHERE e.event_title = %s;

DELETE FROM Events
WHERE event_title = %s;
commit;
```