**CISC637 Spring 2014, Project Final Submission**

4. (a). **Relational schema diagram:**

user\_id

first\_name

last\_name

gender

join\_date

user\_name

users

user\_id

follower\_user\_id

followlist

comment\_id

review\_id

user\_id

comment\_text

comment

user\_id

friend\_user\_id

friendlist

user\_id

checkin\_date

checkin\_time

friend\_user\_id

checkin\_friends

user\_id

location\_id

checkin\_date

checkin \_time

checkin\_lat

checkin\_long

dist\_lat\_long

postal\_code

place\_name

checkins

review\_id

rating

location\_id

user\_id

review\_text

review

postal\_code

place\_name

nearest\_lat

nearest\_long

dname1

dcode1

place\_level1

location\_id

location\_name

add\_date

locations

dname1

dname1

dcode1

dname2

dcode2

country

place\_level2

(b). **(i)** **Database design decisions:**

**# users:**

A table users was created from the unmodified users.txt file since **it follows BCNF**. Since it was mentioned that, a user should be able to create a new account and also log into an existing account, a **new field username was included**. Since a password authentication was not required, it was not included. **user\_id was set as the primary key** because user\_id is unique and to satisfy the functional dependency user\_id 🡪first\_name, last\_name, gender, join\_date, user\_name.

**# locations:**

The table locations was not modified since **it follows BCNF** and thus there is no redundancy. **location\_id** was assigned as the **primary key** since it is a unique value that cannot be duplicated.

**# followlist and friendlist:**

Both these files **were not following 1NF or BCNF**. It is important to have them in 1NF to avoid update anomalies, maintain referential integrity and also to minimize redundant data. A custom python script was written to normalize them into 1NF. Both **user\_id and follower\_user\_id** were made **primary keys** for **followlist** table; and **user\_id** and **friend\_user\_id** were made **primary keys** for the **friendlist** table. All the keys of both these tables are **references to the user\_id** in users table.

# **comment and review:**

Since it is required that users should be able to review locations, comment on friends’ reviews and also rate these reviews, a **comment and review table was created**. For review table (containing review\_id, rating, location\_id, user\_id, review\_text), **review\_id was set as the primary key** since it is supposed to be unique. **review\_id, location\_id and user\_id are foreign keys**. The comment table (containing comment\_id, review\_id, user\_id, comment\_text) has **comment\_id as the primary key**. **user\_id** is **referencing** to the user table. **review\_id** in comment table was set as **unique value** since it is a foreign key for review\_id in review table.

# **checkins:**

The checkins.txt file that was provided **does not follow BCNF**. So, the original checkins.txt file was decomposed into three different tables such as checkins, place\_level1 and place\_level2. Even though all the three tables do not follow BCNF because of foreign key constraints, we could still minimize data redundancy. The checkins table contains the fields user\_id, location\_id, checkin\_date, checkin \_time, checkin\_lat, checkin\_long, dist\_lat\_long, postal\_code and place\_name. The **primary keys are user\_id, location\_id, checkin\_date and checkin \_time** since they all together can uniquely identify the checkin events of users at a particular location and at a specific time. place\_level1 table contains the fields postal\_code, place\_name, nearest\_lat, nearest\_long, dname1 and dcode1. Since the combination of **postal\_code and place\_name** can uniquely identify the other fields in the table, both of them were set as the **primary keys**. place\_level2 table contains the fields dname1, dcode1, dname2, dcode2 and country. Since the combination of **dname1 and dcode1** can uniquely identify the other fields in the table, both of them were set as the **primary keys**.

# **checkin\_friends:**

Another requirement that was specified was that users should be able to check in at locations with friends. Considering this, we created a **checkin\_friends table** with **fields** **user\_id, checkin\_date, checkin\_time, friend\_user\_id**. **user\_id and friend\_user\_id** are foreign keys **referencing** to user\_id in the checkins table; and **checkin\_date and checkin\_time** are foreign keys **referencing** to checkin\_date and checkin\_time in the checkins table.

**The F.D.s that may hold in this setting are:**

* + 1. user\_id **->** first\_name, last\_name, gender,join\_date,user\_name
    2. location\_id **->** location\_name, add\_date
    3. follow\_user\_id, user\_id **->** follow\_user\_id, user\_id
    4. freind\_user\_id, user\_id **->** freind\_user\_id, user\_id
    5. user\_id, location\_id, checkin\_date, checkin \_time **->** checkin\_lat, checkin\_long, dist\_lat\_long, postal\_code, place\_name, nearest\_lat, nearest\_lat, nearest\_long, dname1, dcode1, dname2, dcode2, country

The (v)th functional dependency violates BSNF. Solution is to decompose this relation into 3 new relations to have the following functional dependencies:

1. user\_id, location\_id, checkin\_date, checkin \_time **->** checkin\_lat, checkin\_long, dist\_lat\_long, postal\_code, place\_name
2. postal\_code, place\_name **->** nearest\_lat, nearest\_lat, nearest\_long, dname1, dcode1
3. dname1, dcode1 **->** dname2, dcode2, country

**(ii) Data Import Decisions:**

1. Did you delete any of the original data? i.e., did any of the original data not end up in your database?

We did not delete any data from the original files. All of the data from the original files are present in the database.

1. Did you modify any of the original data?

Yes, some modifications were made to avoid redundancy and prevent any future update anomalies in the database.

1. Which files did you need to change the most, & roughly how many records did you delete or modify from those files?
2. Why did you make the modifications to the data that you did?

We **decompose friendlist and followlist** because they were not in 1NF. We **decomposed original checkins table into checkins, locations, place\_level1, and place\_level2** because the original checkins table had lot of redundancy and was also not in BCNF. The following table depicts all changes we made to data files.

|  |  |  |
| --- | --- | --- |
| **Decomposed Tables** | **Attributes** | **Number of Records** |
| followlist | user\_id, follower\_user\_id | 54214 |
| friendlist | user\_id, friend\_user\_id | 26452 |
| checkins | user\_id,location\_id,checkin\_date,checkin\_time  checkin\_lat, checkin\_long, dist\_lat\_long  postal\_code, place\_name | 563737 |
| locations | location\_id, location\_name, add\_date | 220811 |
| place\_level1 | postal\_code, place\_name, nearest\_lat  nearest\_long, dname1, dcode1 | 563737 |
| place\_level2 | dname1, dcode1, dname2, dcode2, country | 1127473 |

**(iii.) Index design decision:**

2. **Easy Queries:** (referring to the numbering in the query file)

i. We created secondary index on location\_name in locations table. Before we created the index, it was doing full table scan. However, it used index look up to find matching record of location\_name=’zoo’. After index look up, this query finished executing in 0.02 sec compared to 0.10 sec (without using any index).

ii. A) We created secondary index on country in place\_level2 table. Before we created the index, it was doing full table scan. However, it used index look up to find matching record of country=’US’. After index look up, this query finished executing in 0.00 sec compared to 1.58 sec (without using any index).

iii. A), B) AND C) We created secondary index on user’s first name in users table. Before we created the index, it was doing full table scan. However, it used index look up to find matching record of user. After index look up, on average, this query got little bit faster (finished executing in 0.07-0.10 sec compared to 0.20-0.25 sec (without using any index on first\_name)). It is still questionable whether such an improvement in terms of query time is significant enough or not. It could also be an artifact that machine (server) just processed these queries faster because of less workload on it at that time.

2. **Hard Queries:**

i. We created secondary index on location\_name in locations table. Before we created the index, it was doing full table scan of locations. After index look up, this query finished executing in 0.40 sec compared to 2.64 sec (without using any index).

ii. We created secondary indices on location\_name in locations table as well as on location\_id in review table. After creating secondary index on location\_name in locations and location\_id in review, the querying got significantly faster. When using a location\_id on all three relations (locations, review and checkins) and sorting them on it, mysql might be utilizing merge join between checkins, location and review. The query processing time before we made secondary indices was 15.79 sec and 0.26 sec after the secondary indices. This is a significant improvement in query processing time and shows great use of b+tree index.

iii We created secondary index on place\_name in checkins table. After creating secondary index on place\_name in checkins MySQL used index look up to find matching record of place\_name=’New York City’. After index look up, this query finished executing in 0.07 sec compared to 1.24 sec (without using any index). For rest of the easy and hard queries, MySQL had the optimal evaluation plans. Mostly, MySQL chose primary indices for index look up to find matching records.

The information regarding query processing time is summarized in the below table.

|  |  |  |  |
| --- | --- | --- | --- |
| Easy Queries # | Secondary Index on | Query Processing Time | |
|  |  | Before | After |
| 1 | location\_name | 0.10 sec | 0.02 sec |
| 2 | country | 0.02 sec | 0.00 sec |
| 3. A) B) C) | user’s first name | 0.20-0.25 sec | 0.07-0.10 sec |
| Hard Queries # |  | | |
| 1 | location\_name | 2.64 sec | 0.40 sec |
| 2 | location\_name, location\_id | 15.79 sec | 0.26 sec |
| 3 | place\_name | 1.24 sec | 0.07 sec |

**(iv.) Query processing decisions:**

By following the best practices for query designing and optimizing queries, we could improve the execution time of the queries that we initially came up with.

1) Wherever possible select \* was replaced with select specific fields. This helped improved query processing by eliminating the selection of unwanted records. Also, while evaluating the query processing, selections and projections were pushed down the tree so that they are executed earlier, thereby reducing the number of blocks to be carried over for the next stage of query.

2) Full joins were avoided as much as possible.

3) Nested sub queries were avoided as much as possible.

4) Secondary indexes were inserted on frequently accessed fields minimize the execution time.

(c). **Conclusion:**

Scaling the database to a large user base require less redundancy in the database tables, faster query processing time, efficient memory as well as storage, concurrency and recovery. We could reduce the redundancy by making sure that all of our existing as well as new tables (if we create any new tables) are in BCNF/3NF/1NF. We need to have sufficient storage and memory to hold the database and to process user requests (queries). For faster querying times, we could utilize index structures such as hash index as well as B+-tree index. Additionally, for a large user base systems, we have many users accessing the database simultaneously. Therefore, we have to make sure that we follow ACID rules which will give us concurrency, consistency, isolation, durability and atomicity. Recovery is also an important aspect of a large scale database in case of system or hardware failure. We could utilize recovery manager and proper locking protocols to recover from a crash and avoid dead locks scenarios, respectively.

Additionally, if we were to write an actual application and scale up to a large user base, some of the considerations would be:

1) Develop a NoSQL database if this has to be scaled to handle significantly larger data.

2) Add more components to the database like including a password for users to login (in the users table), option for users to message each other.

3) For additional accuracy and consistency in the database, “auto\_increment” and “primary” unique key field could be used for each user. This would ensure that every user\_id is unique. Also in other tables that use user\_id as a foreign key, keeping it as the primary key will improve the query processing efficiency.