Grade 48/A

Exam 1DV513 / 2DV513

Name: Sarpreet Singh Buttar
Personal ID: 950131-0271
Lnu email: sb223ce@student.lnu.se

January 10, 2018

A. E/R to schemas

1. X(A1, A2)
   Y(C1,C2, C3)
   R(B, A1, C1, C2)

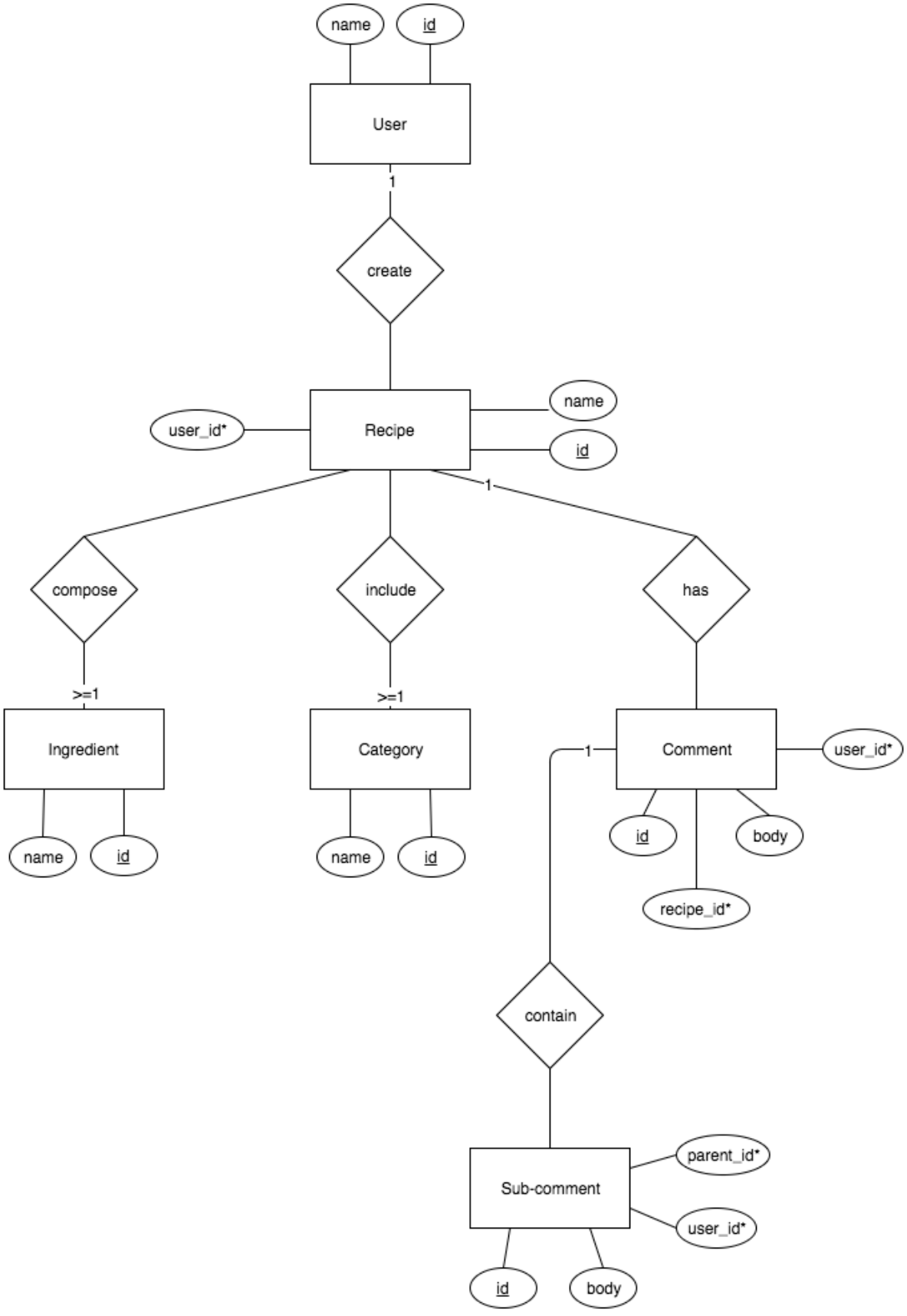
2. X(A1)
   Y(C1)
   Z(D1)
   W(E1)
   R1*(C1, A1)
   R2*(D1, A1)
   R3*(D1, E1)

3. X(A1, A2, C1*)
   Y(C1)

4. X(A1, A2, C1*)
   Y(C1, C2, A1*)

B. Logical Design

1.



2.

```
-- File generated with SQLiteStudio v3.1.1 on Sat Jan 6 18:20:36 2018
--
-- Text encoding used: UTF-8
--
PRAGMA foreign_keys = off;
BEGIN TRANSACTION;

-- Table: category
DROP TABLE IF EXISTS category;

CREATE TABLE category (
    id   INT (11)     PRIMARY KEY
                      UNIQUE
                      NOT NULL,
    name VARCHAR (45) NOT NULL
);


-- Table: comment
DROP TABLE IF EXISTS comment;

CREATE TABLE comment (
    id        INT (11)      PRIMARY KEY
                            UNIQUE
                            NOT NULL,
    body      VARCHAR (200) NOT NULL,
    recipe_id INT (11)      REFERENCES recipe (id) ON DELETE CASCADE
                                                   ON UPDATE RESTRICT
                            NOT NULL,
    user_id   INT (11)      NOT NULL
                            REFERENCES user (id) ON DELETE CASCADE
                                                 ON UPDATE RESTRICT
);


-- Table: favourites
DROP TABLE IF EXISTS favourites;

CREATE TABLE favourites (
    recipe_id INT (11) NOT NULL
                       REFERENCES recipe (id) ON DELETE CASCADE
                                              ON UPDATE RESTRICT,
    user_id   INT (11) NOT NULL
                       REFERENCES user (id) ON DELETE CASCADE
                                            ON UPDATE RESTRICT
);
```

```
-- Table: ingredient
DROP TABLE IF EXISTS ingredient;

CREATE TABLE ingredient (
    id   INT (11)     PRIMARY KEY
                      UNIQUE
                      NOT NULL,
    name VARCHAR (45) NOT NULL
);


-- Table: recipe
DROP TABLE IF EXISTS recipe;

CREATE TABLE recipe (
    id      INT (11)     PRIMARY KEY
                         UNIQUE
                         NOT NULL,
    name    VARCHAR (45) NOT NULL,
    user_id INT (11)     REFERENCES user (id) ON DELETE CASCADE
                                              ON UPDATE RESTRICT
                         NOT NULL
);


-- Table: recipe_category
DROP TABLE IF EXISTS recipe_category;

CREATE TABLE recipe_category (
    recipe_id   INT (11) REFERENCES recipe (id) ON DELETE CASCADE
                                                ON UPDATE RESTRICT
                         NOT NULL,
    category_id INT (11) NOT NULL
                         REFERENCES category (id) ON DELETE RESTRICT
                                                  ON UPDATE RESTRICT
);


-- Table: recipe_ingredient
DROP TABLE IF EXISTS recipe_ingredient;

CREATE TABLE recipe_ingredient (
    recipe_id     INT (11) NOT NULL
                           REFERENCES recipe (id) ON DELETE CASCADE
                                                  ON UPDATE RESTRICT,
    ingredient_id INT (11) NOT NULL
                           REFERENCES ingredient (id) ON DELETE RESTRICT
                                                      ON UPDATE RESTRICT
);
```

```
-- Table: sub_comment
DROP TABLE IF EXISTS sub_comment;

CREATE TABLE sub_comment (
    id        INT (11)      NOT NULL
                            PRIMARY KEY
                            UNIQUE,
    body      VARCHAR (200) NOT NULL,
    user_id   INT (11)      NOT NULL
                            REFERENCES user (id) ON DELETE CASCADE
                                                 ON UPDATE RESTRICT,
    parent_id INT (11)      NOT NULL
                            REFERENCES comment (id) ON DELETE CASCADE
                                                    ON UPDATE RESTRICT
);


-- Table: user
DROP TABLE IF EXISTS user;

CREATE TABLE user (
    id   INT (11)     PRIMARY KEY
                      UNIQUE
                      NOT NULL,
    name VARCHAR (45) NOT NULL
);


COMMIT TRANSACTION;
PRAGMA foreign_keys = on;
```

3.

a.
```
SELECT * FROM user WHERE id IN (
    SELECT user_id FROM recipe GROUP BY user_id HAVING count(*) > 10
);
```

b.
```
SELECT recipe_id AS id, recipe.name AS name, count(*) AS score
    FROM favourites
JOIN recipe ON recipe.id = favourites.recipe_id
    GROUP BY id ORDER BY score DESC LIMIT 1;
```

C. Generic questions

1. Indexing is important because it helps to find the data fastly. It creates an index file where it saves search key or anchor attribute along with the block pointer. So, when we perform a search operation, rather then searching it in every block; we will go to index file where we will match the searck key and go to that block pointer where we can look at the data sequentially and find the suitable one. We must know that finding data is not hard but finding correct block is hard and once we know that block; finding data is very fast. There are some things which we must consider before creating an indexing such as whether to create sparse or dense indexing. If the file is sorted then sparse indexing(only writing few search keys) is good however if file is unsorted then we must go for dense indexing. It is very important to have small index file because if it is too big then we will also spend time to find the block pointer. Index file is usually big when we create dense indexing; in this case we can create a new index file of the index file in order to speed up the process because that is the goal of indexing. In addition, if database is small or we do not care about performance then we can leave the indexing. So indexing is important if we have big data and we want to access it fastly.

2. ACID stands for *Atomicity, Consistency, Isolation and Durability*. These are known as the properties of the *transactions* and if we remove one of these properties then transactions will not work properly because:

    - *Atomicity* is responsible for executing everything as a atomic unit. For instance, if we have 5 operations in the transaction then atomicity will either give us the effect of all of them or none of them.
    - *Consistency* is responsible for the moving the database from a consitence state to a new consistence state. For instance, if a bank transfer money within their customer account, the total money of the bank should be same. Consistency helps the database to achieve this state by making sure that all the constrainst that were before the transcation must holds after the transcation is committed.
    - *Isolation* is responsible for not showing the effects of concurrent transcactions to each other. For instance, if one message has been sent to many users and one of the user delete it from his account, then it should not be deleted from everybody's account. Isolation helps to handle these types of situations in order to maintain the consistency of the database.
    - *Durability* is responsible for saving the effects of committed transcations. For instance, if the transcation is committed then its effect will be permanently saved in the database. In addition, it also help to keep the DBMS data durable and consistent during the time of crashes, system shutdown etc.

3. Redundancy means storing the same data in different places. This can occur by mistake or design decision. No doubt that we need some redundancy in order to connect tables but we must try to minimize it. Redundancy has both advantages and disadvantages. Let's start with advantages; For instance, in the *food recipe service* model on page 1, if we know that comment and sub-comment tables will contain millions of rows as well as fetched many times in a day then perhaps storing name of the user into these tables will increase the performance because name will be needed to display the comment and we do not have to fetch it in a separate query. In this case, redundancy will be by design not by mistake. However, having redundancy can also lead to many problem. For instance, if user update its name then we have to update it in user, comment and sub-comment tables. In my case, deleting user will not make trouble because I have foreign key in all of these tables and I am using CASCADE action however, if I do not have it then I have to also delete user name from all the tables. Let's take one more example, assume we have a table

called student(id, name, course_name, course_id, course_instructor) where course_name, course_id, course_instructor attributes will be same for every student. Now if we perform an insert operation we are saving the same data in diffrent rows except id and name. If we perform update operation on any of the repeated data then we must update it everywhere otherwise we introduce inconsistency in the data. Similarly, if we delete all the students then we automatically delete the course information too. In addition, for small budget developers redundancy can be costly also because it increases the size of database as we are storing same data multiple times.

4. DDL(*Data Defination Language*) and DML(*Data Manipulation Language*) are both part of SQL. DML is responsible for manipulating the schema such as creating, removing and updating the relations. It has 3 main clasues named as *CREATE* (used for creation), *DROP*(used for deletion) and *ALTER*(used for updating). On the other hand, DML is reponsible for manipulating the data stored in database but not in schema. It has 4 main clauses named as *SELECT* (used for selecting data), *INSERT INTO* (used for adding data), *UPATE* (used for updating data) and *DELETE FROM* (used for deleting data).

5. Referential integrity is responsible to ensure the consistency of the database. It is also known as a subset of *data integrity*. It helps to maintain the relationships between the tables by assigning the constraints in order to reject the data that is incorrect or does not exist. For instance, according to the *food recipe service* model on page 1; a user must exist before the recipe; a recipe must exist before the comment; and a comment must exist before the sub-comment. In order to achieve this referential integrity, we define *user_id* as a foreign key in recipe table which references a row in the user table. Now, due to the foreign key reference constraint we cannot add a recipe if the *user_id* does not exist in the user table. Similarly, we can also add foreign key constraint in the other tables. In addition, referential integrity also allow us to configure the possible referential actions such as CASCADE etc that will be executed when foreign key will face the ONDELETE and/or ONUPDATE event. Each referential action behaves differently during the events such as:

   - CASCADE will apply the same change to the row i.e. if foreign key is deleted; the row that referencing it will also be deleted and if the foreign key is updated; the referencing row will also be updated.
   - RESTRICT will prevent the foreign key from updating and deleting.
   - NO ACTION will do nothing on update event. However, it may rollback if the foreign key will face delete event.
   - SET NULL will upate the foreign key value to NULL on both events.

   Refrential integrity brings complexity and performance penalities in the database however it changes the vague relations(one to many, many to many) of our model into exactly or at least one.

Normalization

In general, normalization is a process to eliminate the redundancy and unwanted relations in the table which saves some space in the database; however normalization can effect on performance. Normalization has various normal forms such 1NF, 2NF, 3NF and BCNF etc. Below are detail information about these forms:

- 1NF or First Normal Form only allow that data which is stored in two-dimensional table and includes no repeating groups. For instance: the below data is sored in two-dimensional table but it has repeating groups(courses).

student#	**name**	**courses**
1	John	1dv600, 2dv513

  In order to qualify 1NF, we have to split this data into two tables so that we can eliminate the repeating groups. Below is the data representation after spliting it.

student#	**name**
1	John

student#	**courses**
1	1dv600
1	2dv513

  Now, both tables are in 1NF. On the other hand, we must know that the partial dependency is the major issue in 1NF because it does not care about the primary key.

- 2NF or Second Normal Form helps to solve the partial dependency problem by only allowing the tables which fulfills the 1NF conditions and whose non-key attributes are functionally dependent on the entire primary key. Having partial dependency in a table means that there is risk that we might not able to find all the non-key attributes from primary key. For instance: Lets assume a table R(ABCD) where AB is primary key, $AB \rightarrow C$ and $B \rightarrow C$. The table R fulfills the 1NF conditions but fails the 2NF conditions because C is partially dependent on primary key rather than entirely. The combination of AB gives us four total possible outcomes: A, B not NULL, only A NULL, only B NULL and A,B are NULL(not valid because primary key cannot be NULL). We can see that there is one case when B becomes NULL and in that case it is not possible to find C. So, in order to remove the partial dependency, we have to decompose R into R_1(ABC) and R_2(BC). Now we know that B cannot be NULL because it is a primary key in table R_2 and hence the data is in 2NF. On the other hand, we must know that transitivity dependency such a non-key attribute finds other non-key attribute is the major issue in 2NF because apart from partial dependency it allows all the other cases.

- 3NF or Third Normal Form helps to solve the transitive dependency problem by only allowing the tables which fulfills the 2NF conditions and whose non-key attributes does not have any functional dependency on other non-key attributes. Similar to partial dependency, having transitive dependency in a table means that there is risk that we might not able to find all the non-key attributes from primary key. Lets assume a table R(ABCD) where AB is primary key, $AB \rightarrow C$ and $C \rightarrow D$. The table R fulfills the 2NF conditions but fails the 3NF conditions because of functional dependency($C \rightarrow D$) between non-key attribues. We know that C is non-key attribute which means it can be NULL and if it is; then it is not possible to find D. So, in order to remove the functional dependency between non-key attributes, we have to decompose R into R_1(ABC) and R_2(CD). Now we know that C cannot be NULL because it is a primary key in table R_2 and hence the data is in 3NF. On the

other hand, we must know that having functional dependency of a key attribute with a key or non-key attribute is the major issue in 3NF because apart from functional dependency between non-key attributes it allows all the other cases.

- BCNF or Boyce-Codd Normal Form helps to solve the functional dependency of a key attribute with a key or non-key attribute by only allowing the tables which fullfills the 3NF conditions and whose all determinants are candidate keys. Lets assume a table R(ABCDE) where ABE is a candidate key, $AB \rightarrow C$ and $DE \rightarrow C$ and $B \rightarrow D$. The table R fulfills only 1NF conditions. Now, we will directly decompose into BCNF and according to the defination if it fulfill BCNF then it automatically fulfills 3NF and 2NF. The decomposition of R will give us R_1(ABC), R_2(BD) and R_3(DEC). We manage to keep all the functional dependencies and now the data is in BCNF which means it is also in 3NF, 2NF and 1NF.

E. SQL

1.
   - ```
     INSERT INTO Star (`Name`) VALUES ('sun');

     INSERT INTO Planet (`Name`, `EquatorialCircumference`, `StarName`)
         VALUES
         ('earth', '40075', 'sun'),
         ('mercury', '15329', 'sun'),
         ('venus', '38025', 'sun');
     ```

   - In order to change from KM to Miles, we need to divide the current value with 1.609344 and I assume that the type of 'EquatorialCircumference' column is Float or Double otherwise the result of the conversion will be only saved before the point(.). For instance, 1.609344 will be saved as 1. In addition, I also assume that you want me to update all the rows.

     ```
     # If needed, to update the column type.
     ALTER TABLE Planet MODIFY EquatorialCircumference FLOAT;

     # For updating all the rows.
     UPDATE Planet
     set EquatorialCircumference = EquatorialCircumference / 1.609344;
     ```

   - ```
     DELETE FROM Planet
         WHERE Name IN (
         SELECT PlanetName FROM Moon
         GROUP BY PlanetName
         HAVING count(*) < 20
     );
     ```

   - ```
     SELECT * FROM Moon WHERE Name LIKE '%tan%';
     ```

2. Follwing are the the join operators in SQL:
   - Joins can be divided into two types:
     - *Inner Join:* The outcome of the inner join will contains only matching tuples. For instance, we have table A with 10 tuples and B with 8 tuples and based on some join condition only 4 tuples match, so in this case inner join will give us those 4 tuples. It can be further divided into three different joins:
       - *Theta Join:* includes comparitive operator such as >, <, =, != , $\geq$, $\leq$.
       - *Equi Join:* only include '=' operator. It is also a theta join.
       - *Natural Join:* includes common attributes in both relation.
     - *Outer Join:* The outcome of the outer join will contains all tuples from either one or both relations. It can be further divided into three different joins:
       - *Left Outer Join:* always includes tuples of left relation whether the condition is matched or not.
       - *Right Join:* always includes tuples of right relation whether the condition is matched or not.
       - *Full Outer Join:* includes tuples from both relations..
   - Transactions are used to perform one or more database operations as an atomic unit which either executed completely or not at all. For instance, in the real world when
     ```

we transfer money from one account to another; it either transferred or not. Similarly, transactions are either *committed* or *rollbacked*. Transcations are explicitly initiated using START TRANSACTION command and its changes will only be permanent in the database if it is committed using COMMIT command. If something goes wrong we can abort the transaction using ROLLBACK command and restore the database to the state it had before the transaction started. Transaction have four properties known as *Atomicity, Consistency, Isolation and Durability* which I have already discussed in section C. Every transaction must fulfill these properties in order to work in a way we want it to work.