This report explains the normalization and functional dependency calculation. The report also includes explanations on creating relationships by merging existing database tables.

Note: The program code is provided individually for each question in the root folder. The program can be executed by following the README.md instructions. The questions from the root directory contain method calls from the other files in the "code" subfolder to make the code more manageable and readable. The global settings are provided in "globals.py" inside the "code" subfolder.

Question 1: Create a new relation by joining User, Post, PostTags, and Tags.

The code to create the new relation is provided in "q1.py". If the database is not populated, the code first populates the relations. This can be done by setting the flag "db_exists" to "False", setting the data directory in the "code/globals.py", and providing the necessary files (users, posts, and tags) to populate the database. The query to join users, posts, tags, and post tags is provided in "code/queries.py" using "join_tables_q1". The query selects relevant attributes from users (ID, DisplayName, AboutMe, CreationDate, Reputation), posts (Id, Title, Body, Score, ViewCount, CreationDate), and tags (Id, TagName), by joining them on common keys between users, posts, and tags. To ensure that the selection only contains posts with one tag, results are filtered by counting post tag IDs equal to one. Grouping is applied to apply the filtering class correctly. I created the primary key as a serial key to uniquely identify each record. This ID will not be used during calculations regarding dependencies.

Question 2: Implement the naïve approach to discover functional dependencies on the relation from question 1. Check all possible dependencies against all pairs of rows excluding trivial dependencies and with one attribute on the right-hand side. Estimate the time the program should take. Explain your answer.

The code to implement the naïve approach to discover functional dependencies on the relation from question 1 is provided in "q2.py". The code checks all possible dependencies against all pairs of rows. A functional dependency $\alpha \rightarrow \beta$ states that if two rows have the same values for attributes in α (the left-hand side), they must also have the same value for attribute β (the right-hand side). I am excluding the trivial dependencies where the code checks for self-dependency for each functional dependency. The code also checks dependencies with one attribute on the right-hand side and a combination number of attributes (or one) on the left-hand side. To implement this, I read all the datasets in one panda's data frame and grouped them on unique values to reduce the number of comparisons, instead of checking each row.

I ran the program for around 15 iterations for a combination of attributes for the left-hand side with each attribute on the right-hand side. The code takes around 4.5 minutes to run, I assume the code would take around 1200 minutes to run for all possible combinations. The results of partial output are provided in "q2_fdss.txt". To run the program for all combinations, remove the "break" condition on the counter. The program first checks if the specified left hand functionally determines the right hand. It groups the data by the left hand and counts the unique values of the right hand. If every group has only one unique value of the right-hand side, then left hand functionally determines right-hand functionality. Therefore, non-trivial functional dependencies are identified by checking each possible combination of attributes and seeing if they uniquely determine other attributes in the dataset. Also, I exclude the artificially created primary key ("Join ID") for joining the table in the dependency identification. I could also exclude "User Creation" and "Post Creation" dates to speed up the computation, as it looks like those are system-generated timestamps to capture when the record is inserted. This also applies to the other questions.

Question 3: Implement the pruning approach to discover functional dependencies on the relation from question 1. (Partition input columns and a lattice of possible dependencies) Use combinations of no more than two attributes on the left-hand side (and one on the right). State the functional dependencies found and provide examples of pruning functional dependencies. Explain your answer.

The code to implement the pruning approach to discover functional dependencies on the relation from question 1 is provided in "q3.py" which takes around 1 hour to execute. Dependencies are stored in "q3_fd.txt" and pruned dependencies are stored in "q3_fd_prune.txt". A functional dependency $\alpha \rightarrow \beta$ states that if two rows have the same values for attributes in α (the left-hand side), they must also have the same value for attribute β

(the right-hand side). I exclude the artificially created primary key ("Join ID") for the joining table in the dependency identification. I am excluding the trivial dependencies where the code checks for self-dependency for each functional dependency. The code checks functional dependencies with combinations of no more than two attributes on the left-hand side (and one on the right). To implement this, I use the partitions based on the left-hand side attributes by grouping the data frame to identify unique values of the right-hand side attribute within each group. If all rows in a group share the same value for the right attribute, it indicates a functional dependency.

The implementation uses the lattice structure where keys are tuples of attributes (up to two attributes for the left-hand side). If a single attribute can determine multiple other attributes, those links are captured in the lattice. By grouping the data and checking for unique values, it creates partitions based on the left-hand side attributes. Subsequently, discovered functional dependencies are checked if any can be considered redundant. The implementation checks to find whether a dependency can still hold if one of the left attributes is removed. If it can, that dependency is considered redundant and pruned.

Question 4: Assuming that there are no more minimal functional dependencies than the ones computed in Question 3 (combinations of no more than two attributes on the left-hand side), explain the outcome if we do not restrict to posts with a single tag as in question 1.

If we do not restrict to posts with a single tag, each post could occur in the dataset multiple times with each tag present for that post. This would significantly increase the redundancy and number of rows, which would increase the computational complexity of finding dependencies. However, based on the output of minimal functional dependencies than the ones computed in question 1, there would not be any effect on the identified dependencies. For instance, the following dependencies would still hold if there are multiple tags for each post.

```
('tagid',) -> tagname
('tagname',) -> tagid
('postbody', 'tagid') -> reputation
('postbody', 'tagname') -> reputation
```

Question 5: Compute a 3NF decomposition of the relation from question 1 given the set of functional dependencies discovered in question 3. You may do this manually or write a program to do so. If you did not complete question 3, you may use dependencies you determined manually based on your knowledge of the data. Provide the results of the decomposition (candidate keys, canonical cover, and final decomposition).

Based on the relation from question 1 and given the set of functional dependencies discovered in question 3, I deduce the following base functional dependencies that are minimal representative of functional dependencies. These inferred from question 3 are inferred (minimal) as follows;

```
Minimum Dependencies
('userid',) -> userdisplayname
('userid',) -> aboutme
('userid',) -> usercreationdate
('userid',) -> reputation
('postid',) -> userid
('postid',) -> posttitle
('postid',) -> postbody
('postid',) -> postscore
('postid',) -> postviewcount
('postid',) -> postcreationdate
('tagid',) -> tagname
('posttitle', 'postviewcount') -> postid
```

As depicted in “q3_fd.txt” and “q3_fd_prune.txt”, all attributes are captured in the dependencies check. The attributes that constitute the functional dependencies set are “userdisplayname”, “aboutme”, “usercreationdate”, “reputation”, “userid”, “posttitle”, “postbody”, “postscore” “postviewcount”, “postcreationdate”, “tagname”, “tagid”, and “postid”. Based on these attributes constituting the functional dependencies and the knowledge of the data I have, I deduce the set of functional dependencies

F = {userid → userdisplayname, aboutme, usercreationdate, reputation},

{postid → posttitle, postbody, postscore, postviewcount, postcreationdate},

{postid → userid}

{tagid → tagname}

Here, each functional dependency is minimal, so they stay the same. I remove the redundant keys to have this composite set to identify and decompose the results. From this, postid, userid, and tagid can be part of candidate keys, but they don’t independently cover all attributes. Thus, the canonical cover is the same as the original set of functional dependencies. Using the canonical cover, following relations can be decomposed;

R1: (userid,userdisplayname, aboutme, usercreationdate, reputation)

R2: (postid, posttitle, postbody, postscore, postviewcount, postcreationdate, userid)

R3: (tagid, tagname)

Given that post → userid this can be a separate relation having (postid, userid), but it can be fitted with **R2** by adding userid into R2). Also, given that (posttitle, postviewcount) → postid does not add any new information and is therefore unnecessary to be added as another relation.

Note: Instructions to run the code are provided in the README.md file in the root folder.