Databases: Task 1

This website application contains the functionality to create users, log in as a user, create forums, create topics within those forums, and then create posts within those topics. Therefore, the schema required 4 tables; one each for users (Person - already provided), forums, topics, and posts.

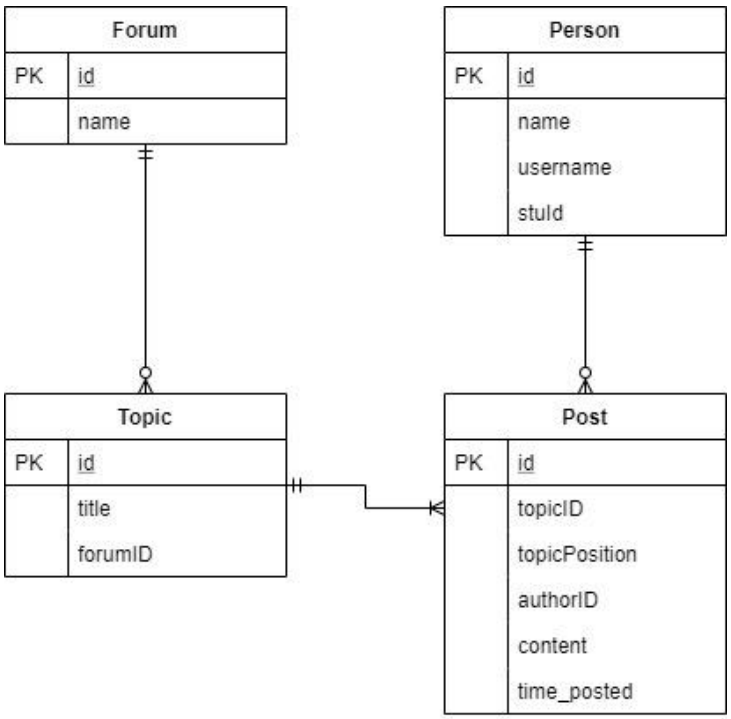

Figure 1 – Entity relationship diagram for the task 1 schema

Figure 1 shows these four tables and the relationships between them. For example, a person can create multiple posts, but each post can only have one author, and a topic can have one or more posts within it, but each post can only exist within one topic.

Table Designs

In terms of the design of each table, Forum could be identified by *name* since these have to be unique, although if functionality were to be added allowing users to change forum names then the references to these forums in other tables would all need updating. For this reason, each forum has a unique id as its primary key, which can then be referenced by other tables. Forum is currently in BCNF because it is in 3NF (no transitive dependencies) and does not contain any composite keys. This schema design didn't really increase the complexity of the API implementation because of how simplistic this table is.

Topic could use (*title*, *forumID*) as it's composite candidate key to uniquely identify it, since topic titles are unique within each forum, but this would introduce the same issues as if the Forum table used name as its primary key. Therefore, a uniquely identifying *id* variable was used as the primary key, also allowing each topic to easily be referenced by other tables. Topic is also normalised to BCNF because title and *forumID* are both dependent on *id*. For the API implementation, because the

combination of title and *forumID* must be unique this introduced an extra part of the SQL query to check that they don't already exist, but otherwise this was also a simple table to implement.

Post was the most complex table to implement, with the most variability. Each post is uniquely identified by its auto-incrementing id, and contains the id of the topic in which it exists, its position relative to the other posts in that topic, the id of the user that wrote it, the content within the post and the time at which it was posted. *topicID* allows the Post table to be joined to the Topic table so that the name of the current topic can be retrieved and published to the web page. The original schema design stored the author's username instead of *authorID,* because usernames are unique and this made it easier to publish posts. However, it was never specified whether the website allows users to change their usernames, and if this were to be added then every row in the database would have to be updated with the user's new username (which is bad database design). Therefore, the schema and the API were updated, with the SQL query to retrieve the data now including a join on to the Person table to retrieve the post author's username using their id. This made the API a little more complex, but allows for future added functionality.

Because posts cannot be deleted, the variable *topicPosition* is used to store the order in which posts should appear on the web page. When creating a new post, this is calculated by counting all the posts currently in a topic, adding one to the result, and then it as the new post's *topicPosition*. Currently, this works because the website doesn't allow for posts to be deleted – if a post were to be deleted, every post created after the deleted post would have to be updated to subtract one from its *topicPosition*, which is a bad database design. The schema could be simplified by removing *topicPosition* and then assigning the posts a position by ordering by when they were created (*time_posted)*, which would also allow for posts to be deleted in future because the order is assigned when the available posts are published. However, this functionality isn't needed and complicates the API design, also making the *countPostsInTopic* function redundant which is a part of the interface and should be implemented.

Each row in the Post table could be uniquely identified by *(topicID, topicPosition)*, but having a surrogate *id* key means that the table is normalised to BCNF, with all 4 other variables uniquely identifiable by *id*.

The schema for the Person table couldn't be modified, and for the most part had already been implemented in the API, however it already satisfies BCNF and so is sufficiently normalised.

Databases: Task 3

Question 1

The updated schema, which incorporates the functionality to like topics and posts, required two new relational tables; LikesTopic and LikesPost. Since a topic/post can be liked by many people, and people can like many different topics/posts, both required many-to-many relationships between Person and Topic, and Person and Post. These tables can be seen in Figure 2 below:

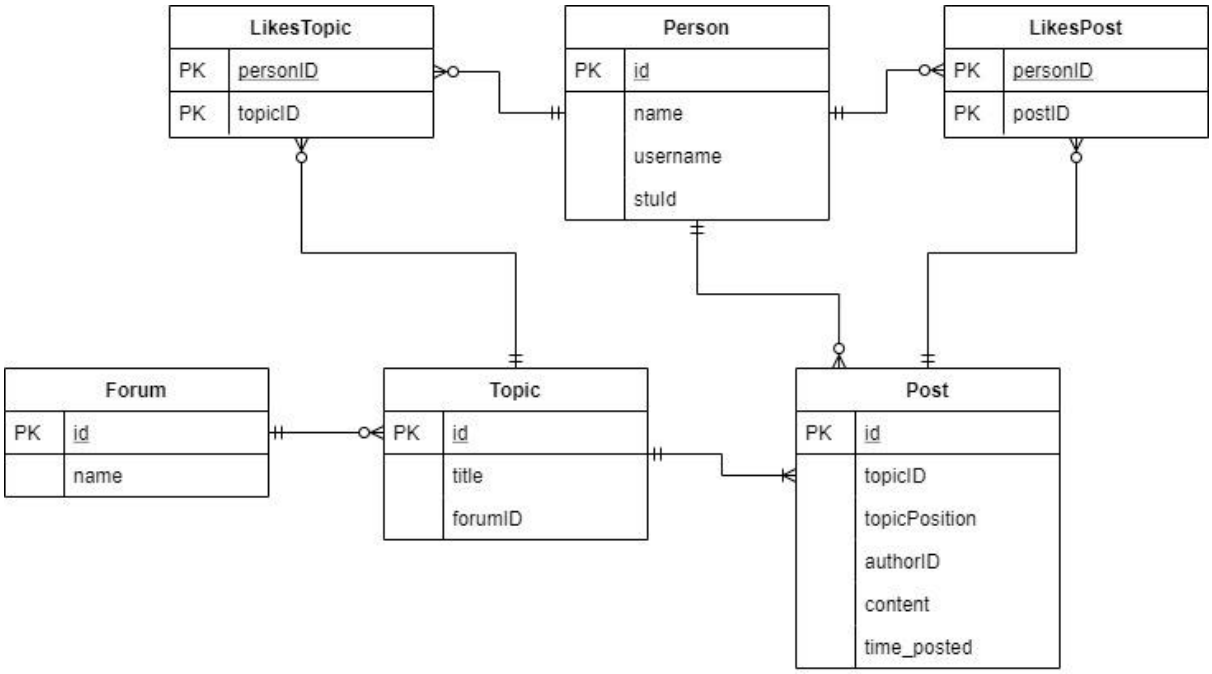

Figure 2 – Entity relationship diagram for the task 3 schema

The primary key in each relational table is a composite key formed of both variables and can't be simplified anymore, so it can be assumed that they are both fully normalised. Having this as the primary key also prevents duplicates of this entry – i.e. people aren't able to like topics/posts more than once. If they did want this functionality, an *id* field as the primary key would solve this but given how most websites implement 'like' functionality, this is unlikely.

Adding in these tables doesn't impact the current schema design because they are both associative tables referencing the primary keys of pre-existing tables. Therefore, no variables need to be added or removed to maintain the new functionality.

Question 2

To record a user liking a post, an INSERT query will need to be executed that adds the *personID* and the *postID* to the LikesPost Table:

INSERT INTO LikesPost (personID, postID) VALUES (?, ?);

If the *personID* wasn't know, a SELECT query would have to be done beforehand to find out the id from whatever information was known (and the same for *postID*).

To retrieve the names of all the people who have liked a specific topic (in alphabetical order), the following statement would need to be used:

SELECT Person.name FROM LikesTopic

JOIN Person ON Person.id = LikesTopic.personID

WHERE topicID = 3

ORDER BY Person.name ASC;

This query assumes the topicID is already know, however if the user only knows the name of the topic and the forum it is in, a modification could be made to get all the information needed with one query.

Question 3

Adding functionality to an existing schema can introduce complications which require the current schema to be restructured. For example, as mentioned earlier, if this website wanted to allow users to delete their posts, the *postPosition* variable would become redundant and the API would have to number the posts as they were published to the web page. However, if the website were to allow users to change their usernames, the current schema allows for this because the author variable in the Post table is tied to the user id, not the username. Therefore, if a username is changed, the user's posts are updated with it.

This is also the case regarding forum and topic names – if functionality were to be added allowing these to be renamed, the current schema can accommodate this because of the surrogate *id* primary keys in each table. If *name* in Forum were to be used as the primary key, topics would refer to forum names instead of ids, and if a forum were to be renamed a lot of rows referencing that forum would need to be updated. Having *id* as the primary key therefore allows for the functionality to be expanded.

The danger associated with restructuring the current schema lies in the potential loss of data. For example, if an extra column were to be added to an existing table which is specified with 'NOT NULL', every entry in that table now needs to include a value for that column – if not, the database could throw an error and the data would be lost. The API would also need refactoring/testing because adding this new column would break the current methods – every SQL query which references the updated table would need to be modified to include the extra column. This is also the case if a column were to be removed, although there isn't a risk of losing data because the entries in the removed column will simply be deleted.