DATABASES COURSEWORK TWO
TASK 1

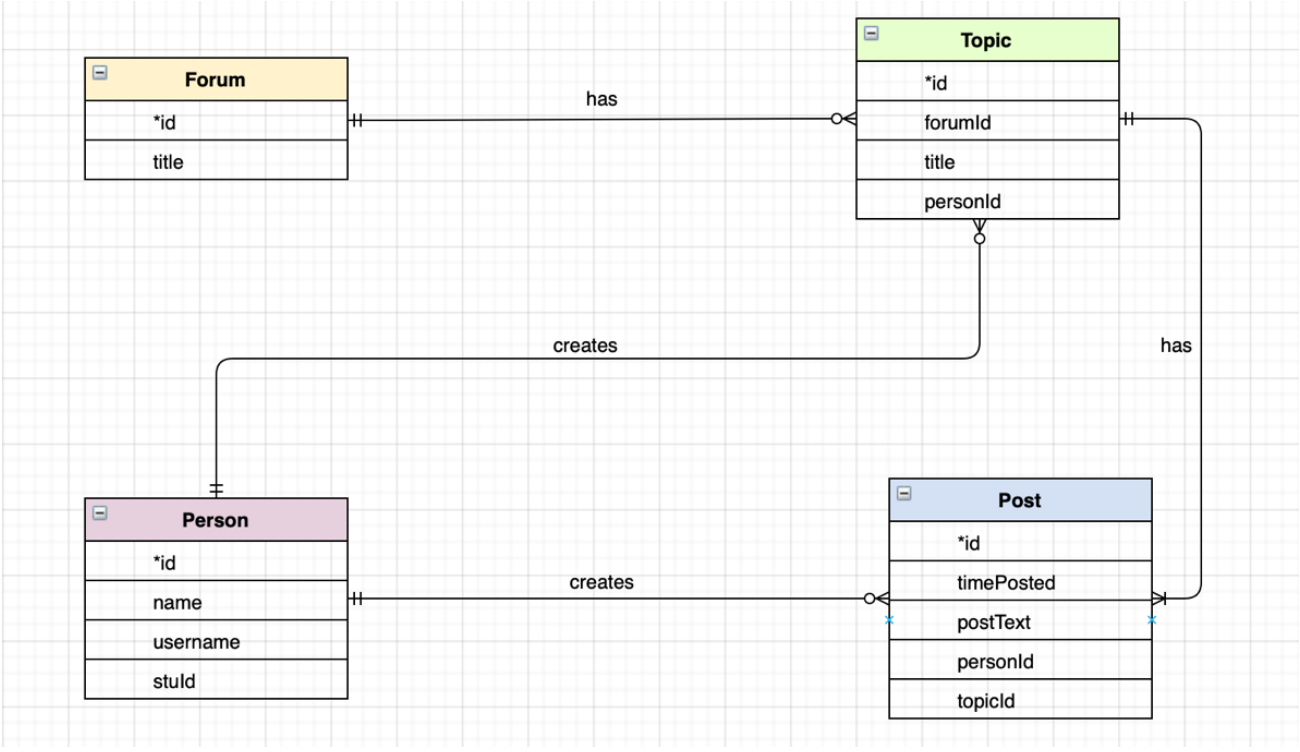
Schema Description

The schema designs the database for a forum application. Each forum can have topics, and each topic can have posts. The person is the user of the forum application. Design decisions as to the relationship, chosen primary keys, and other considerations will be detailed.

Any of the attributes that have a limit for characters will be explained. A generic checkLength( ) method was created however, and returns a Boolean to ensure that the value is less than the specified characters. This will than flag a Result.failure to tell the user to insert an attribute with the required characters. Implementing the requirements in this form ensured that there were no database exceptions thrown.

Each table has an id that is autoincremented as the primary key. This number if for internal use only. The general rationale for this is that this technical primary key, also known as a surrogate key,[1] is generated by the database management system when the forum is entered in the database. The decision to generate an id number for each table was made as if the primary key was made on the contextual data, it may not have the same meaning later on and may need to be altered.[2] Thus, the id number would be read-only as the primary key to ensure user values are not written over.

One final general note for the design of this schema, is that due to the architecture many redundant queries can be avoided. Duplicated security validation functionalities are avoided as many of the 'id' attributes are foreign keys in other tables. Using this foreign key, if data is entered that does not exist (ie. the creation of a Topic using a forumId that is not real) an SQLExcpetion will be returned. The use of foreign keys is an extra layer of security protection for the database.



[1] http://www.agiledata.org/essays/keys.html
[2] https://www.sisense.com/en-gb/blog/when-and-how-to-use-surrogate-keys/

The schema has been designed as per the above ER diagram, and the tables, attributes, and relationships are as follows:

Tables

- **Forum**: the Forum table has an id and title stored in it, the title is provided by the user.
  - **id**: the id of the Forum. The title was not chosen as the primary key as if the functionality was added to allow the user to rename the forum have creation, then the title could not be the primary key despite the fact that it is unique.
  - **title:** the title of the forum is created by the user and cannot be generated duplicated. although the title of the forum is unique (there cannot be the same two forums with the same name) it is not the primary key for this table. This allows for scalability of the database if, in the future, there was the option of editing the Forum title, as mentioned above. The title has a varchar limit of 100 characters as the title should be short and not exceed this length, the length is checked in the code prior to inserting to ensure that there is not a database exception. The title can also not be null, as a forum must have a title to be accessed, this is also checked in the code (the code also checks that the username cannot be empty).
- **Topic:** the Topic table contains the id number, the forumId, the title of the Topic, and the personId of the author.
  - **id:** the id of the Topic. For reasons listed above, this is the primary key for the Topic table.
  - **forumId:** this is the corresponding forumId number that the topic is created in. This number is passed in as a parameter. The forumId number cannot be null, thus the code checks that the forum exists prior to inserting the data in to the database. Including this attribute also allows for all requirements of the getForum( ) method to be executed in one query:

```
SELECT Forum.title AS forumTitle, Topic.id AS topicId,
Topic.title AS topicTile
FROM Topic
RIGHT JOIN Forum ON Forum.id = Topic.forumId
WHERE Forum.id = ?
```

    This is a foreign key on Forum.id. Which means that when creating a topic, it is not necessary to check that it is an existing forum due to the constraint, thus removing redundant queries.
  - **title:** the title of the topic is generated by the user. It does not have to be unique. The title has a varchar limit of 100 characters, similar to the forum this number was chosen to keep the titles short. The code ensures that it does not exceed 100 characters. The title cannot be null to ensure that each topic has a title.
  - **personId:** the personId is the author of the topic. This was not originally included in the database schema, but was added at a later date to ensure that each topic knows who posted it. This can allow for future scalability of the database. For example, if the initial post is deleted (which would contain the author of the topic, as the first post is authored by the creator of the topic) it would be possible to use a simple query to access who posted the topic should that information be needed:

```
SELECT Person.name FROM Topic
INNER JOIN Person ON Topic.personId = Person.id
WHERE Person.id = ?
```

    This would therefore allow access to who posted the topic. This could be useful in cases such as where the Topic posted was controversial and sparked a debate, and it was necessary to find out who posted it. PersonId is accessed to create the first post, so therefore it was not more difficult to implement. This is a foreign key on Person.id. This ensures that when topics are created the person creating it must be a user in the database, or else it is invalid.

- **Post:** the Post table contains the id number as the primary key, the time it was posted, the text (body) of the post, the personId of the author, and the corresponding topicId.
    - **id:** the id of the Post. For the same reasons as the other tables, this is the primary key.
    - **timePosted:** the timePosted attribute is a DATETIME value to record the time that the user posted. This cannot be null, and is generated when the data is inserted in to the database using the now ( ) function call.
    - **postText:** this attribute contains the body of the text. This cannot be null, or empty and this is checked prior to inserting. This attribute has a varchar limit of 8000 characters. This decision was made over the use of VARCHAR(MAX) for a few reasons. Primarily, as this is a forum application it seemed sensible to limit the length of posts or else they could be exceedingly long. Secondly, it allows for simpler design, including the prevention of any UI issues. Finally, using VARCHAR(n) instead of VARCHAR(MAX) was sensible for the performance benefits of the storage, as well as the fact that VARCHAR(MAX) does not compress.[3] The length of the text is checked prior to inserting in to the database to avoid any database errors being returned.
    - **personId:** the personId is stored to ensure that the author can be attributed to the post when getTopic( ) is called. This is a foreign key on Person.id, which again removes redundant queries due to the fact that as a back-up check an SQLException is returned if the Person.id does not exist. Including the personId attribute allowed for all requirements of getTopic( ) to be executed in one SQL query:

```
SELECT title, Person.name AS name,
Post.postText AS text, Post.timePosted AS date
FROM Topic
INNER JOIN Post ON Post.topicId = Topic.id
INNER JOIN Person ON Person.id = Post.personId
WHERE Topic.id = ?
```

    - **topicId:** difficulties did arise when creating a Topic to access the topicId and insert it in to the Post.topicId as the first post is created when creating a topic. This is perhaps a limitation in the design of the schema, and was solved by ordering all ids from Topic.id in descending order and placing a limit of 1 result to select the most recent id number after the topic was created in the database. However, as the Post has to know which topic it is in, it made the most sense to include the topicId to keep track of this. Thus, this seemed like the best solution. This is a foreign key on Topic.id.

  NB. the decision was made to not include a post number attribute in the Post table. This was decided as the post numbers are generated per topic, however the posts are stored in the Post table regardless of which topic they are under. Therefore, to select the post number when getTopic( ) is called an iterator is used to generate post numbers depending on topic. This was decided after an issue arose where under the second topic, the only two posts were numbered #3 and #4, as they were the 3d and 4th posts in the forum.
- **Person:** the Person table is the table detailing each user of the system. The schema for the Person table was provided, however the implementation of each aspect will be detailed.
    - **id:** the database dependant primary key, which is auto-incremented upon insertion to the database.
    - **name:** the name of the user. This has a varchar limit of 100 characters and cannot be null. This is checked prior to inserting the data in the database.

[3] https://www.red-gate.com/simple-talk/sql/database-administration/whats-the-point-of-using-varcharn-anymore/

- **username:** the username which has a varchar limit of 10, cannot be null, and must be unique. Each of these requirements are checked prior to inserting the data in the database.
- **stuId:** the student Id can be null, however it does have varchar limit of 10 as well. This is also checked prior to inserting the data in the database.

Relationships

- **Forum – Topic:** there is a one to many relationship between Forum and Topic. Each forum has any number of topics, but each topic only has one forum.
- **Person - Topic:** there is one to many relationship between Person and Topic. Each person can create any number of topics, but each topic has only been created by one person.
- **Person – Post:** there is a one to many relationship between Person and Post. Each person can create any number of posts, but each post has only been created by one person. A person can have 0 posts, as they can register prior to the creation of a post.
- **Post – Topic:** there is a one to at least one relationship between Post and Topic. The topic has at least one post because a post is created upon creation of the topic, and the post is only contained in one topic.

Normal Forms

As normal forms stack, each normal form will be considered in turn. The candidate keys in this schema are the {id} attributes in each table, as they are the primary key. Moreover, they cannot be null and it is necessary for each entry in each table to have an Id number. Other candidate keys include the {title} attribute in the Forum application, and the {username} attribute in the People table as both are required to be unique and can therefore uniquely identify the row of data. All the rest of the attributes are non-key attributes, as they are not required to be unique and can be null. The schema satisfies 1NF form as there are no collection-values attributes in the schema, each attribute has single (automatic) values.

The schema also satisfies 2NF (as it satisfies 1NF and) there are no partial functional dependencies (where the primary key is composed of two fields) from any candidate keys to any non-key attributes. Each attribute depends on the primary key (the id).

The schema satisfies 3NF (as it satisfies 2NF and) there are no transitive dependencies between non-key attributes. In each table, each field is only dependent on the id (the candidate key) to be accessed. Moreover, the title in the Forum table is the only attribute given the UNIQUE property. As only {id} and {title} are in the table and they are both candidate keys, this does not constitute as a transitive dependency as those must be between non-key attributes. All non-key attributes therefore only depend on the candidate keys. Additionally, although there is a dependency between {id} -> {username} -> {name}, as both id and username are candidate keys (key attributes), this also does not constitute a transitive dependency.

The schema satisfies BCNF (as it satisfies 3NF and), as for the same reasons as to why it satisfies 3NF, each attribute in the tables only depend on the {id} candidate key. This is with the exception of the Forum and Person tables, which are still in BCNF. This is because in the Person table, there are no non-trivial functional dependencies that are created that do not include the username or the id in the determinant. Additionally, the Forum table is in BCNF as each column is a candidate key {id}->{title}, and as a result the determinant of the nontrivial functional dependencies is a super key.

The schema does not satisfy 4NF however, as there is a multivalued dependency in Post. This is due to the fact that there can be two 'post' entries made that contain the same {topicId} and {userId,} however with different {timePosted} and {postText} data. Therefore, this schema is normalised to BCNF.

This database schema has been normalised in order to avoid duplicated data, to minimize data modification, and to allow for simplified queries. This ensures that we avoid that 'Update Anomaly' and that the same information is not duplicated in several rows, and that if all rows are not updated accordingly then there could be inconsistencies. This could occur if an edit feature was added to the forum, and for instance the user just wanted to edit the post. By separating the post and topic data, this ensures simpler queries as just the data in the post table is updated. Moreover, the data is separated between each table which means there will not be any inconsistencies (if for example the name of the forum were to change, this would not affect the topic and post). Additionally, normalising the database ensures that the 'Deletion Anomaly' is avoided. For example, if there were to be a row deleted then there would be a loss of more information that necessary. This means that if, for example, the ability to delete posts were to be implemented – data would also not be lost about the Topic (if Topic and Post were not normalised). Finally, this avoids the 'Insert Anomaly' where there are facts that cannot be records until all information is acquired. This ensures that the user, for instance, does not have to post something in order for their details to be added. A user can be added separate from the posts in the forum. Additionally, you can create a forum without inserting any posts. This ensures separation of data, and more simplified SQL queries.

There was one difficulty that was noted in the implementation of this schema, however. As the schema has been normalised to include a Post and Topic table, when the topic is created there is an extra step of accessing the newly created topicId in order to use this data to create the post. The work around to this was described above. The final implementation was the best solution, as it ensured that the topic row was added prior to the creation of the post. The drawback is the fact that this is done in two SQL queries, albeit simple queries. On the whole, the implementation of the database in this form allowed for simplified queries and the avoidance of the aforementioned anomalies.

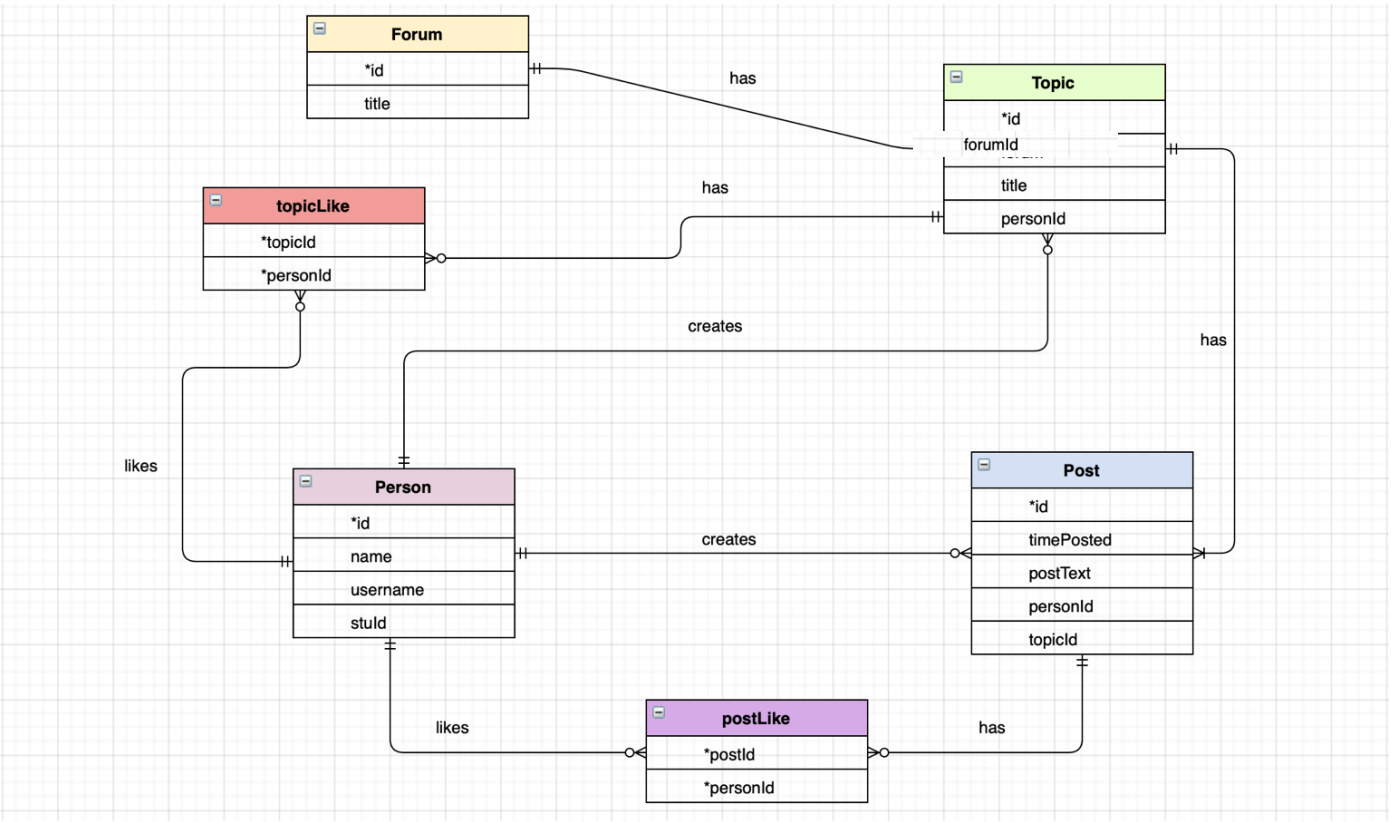
API Design

One final note on the design of the code for this assignment, for simplicity and design a second 'Queries' class was made. Only certain data features can be added through this second class, including any 'insert' changes.[4] This decision was made with security and maintainability to allow for future scaling of the database. Moreover, all SQL queries are executed using prepared statement to prevent against SQL injections. Additionally, as mentioned above, to avoid redundant queries and to also avoid duplicating the security validation functionality (as is done in a separate piece of code), the code does not check for a valid user when creating new Topics or Posts. As the Person.id is stored as a foreign key in both tables, a SQLException will be thrown if the other security layers fail. This is similarly handled in other sections such as the creation of a Topic with checking that a Forum exists, as Forum.id is stored as a foreign key in the Topic table. Finally, it is of note that stuId in the Person table must be NULL, however for PersonView stuId must be empty and NOT NULL. Thus, if it is null it is set as an empty string before the PersonView is created. Future work would involve editing the schema to ensure consistency.

[4] https://www.red-gate.com/simple-talk/sql/database-administration/ten-common-database-design-mistakes/

TASK 3

Changes for Implementation of "like" Functionality (max. 2 pages)



Two new tables were created to implement the "like" feature for the Forum. As users are now able to "like" a post or topic, the two tables created were postLike and topicLike.

Tables

- **postLike:** this table details the likes on a post. Each entry counts as one like on a post. The primary key for this table is a composite candidate key of {postId, personId}. The decision to implement the table like so was because each person can only like a post once. Thus, in order to select total likes on a post to display them to the user, a simple query:

```
SELECT COUNT(*) AS totalLikes FROM postLike WHERE postId = ?;
```

  can be used.
  - **postId** references the post that has been liked by a user. This is a foreign key on Post.id. Each post has a unique id to allow for an indexing of posts by postId. While there are likely to be multiple entries of the same postId, as multiple users can like a single post, this was still the best implementation for creating a table of each like. This value cannot be null.
  - **personId** references the id number of the user. This is a foreign key on Person.id. As the id is a surrogate key, this ensures that even if the user were be able to change the username in the future, the likes would stay the same as they are indexed by the id number. This value cannot be null.

  Implementing the like functionality for a post in this form ensured that the Update Anomaly, as described in Task 1, is avoided. If the user were to update their username, or if the post text were to change for example, the like would still be documented in the table.

- **topicLike:** topicLike has a similar implementation as postLike, creating a table where each entry documents a like on a Topic. The primary key is a composite candidate key of {topicId, personId}.
    - **topicId** similar to postLike.postId, topicId is a foreign key that refers to Topic.id. This value cannot be null.
    - **personId** is the same as postLike.personId, it is a foreign key that refers to Person.id. This value cannot be null.

Relationships

All other tables and relationships were not updated for the implementation of the like feature. However, the following new relationships exist:

- **Topic – TopicLike:** there is a one to many relationship between Topic and TopicLike. Each like on a Topic (TopicLike) can only be on a single Topic, however each Topic can have any number of likes.
- **Person – TopicLike:** there is a one to many relationship between Person and TopicLike. Each Person can like any number of Topics, however each TopicLike can only have been liked (or created) by one Person (TopicLike).
- **Post – PostLike:** there is a one to many relationship between Post and PostLike. Each like on a Post (PostLike) only has one Post that it has liked, however each Post can have any number of likes.
- **Person – PostLike:** there is a one to many relationship between Person and PostLike. Each individual like on a Post (PostLike) only comes from one Person. However, each Person can like any number of Posts (PostLike).

Normalisation

With the addition of the the PostLike and TopicLike tables, the schema remains in BCNF. As none of the table are altered, the dependencies remain the same. The addition of the tables keeps the schema in 1NF as teach cell contains atomic values, and each value is a single element. The schema is still in 2NF as there are no additions of non-key attributes, and hence no partial functional dependencies from a candidate key to a non-key attributes. The schema is also in 3NF as, again there are no non-key attributes added, and hence there are no transitive dependencies. Finally, the schema remains in BCNF, this is because as each single column is part of the composite primary key, there can be no dependencies between non-key attributes. For the same reasons as Task1, the schema is not in 4NF.

Future Updates

Implementing the like functionality in this way allows for future updates on the Forum application to be executed that would not affect the creation of these tables, or the storing of the likes. This also allows for likes per Post and Topic to be displayed cumulatively, and by user, depending on the UI of the forum. If in the future a user were to wish to delete their account, the removal of the topicLike and postLike data attributed to that Person.id would not affect any of the other like data, nor the data in any of the other tables. Moreover, if the user were to change their username this would not affect how the like data is stored, however as the Person.id is utilised to store the like data the username can be displayed if the Forum application were to include a "view users who liked this" functionality.

Provide functional SQL queries in your report (max. 2 pages)

The following two queries explain use case, the query, and the parameters before explaining more about the query.

Recording a "like" for a post:

```
INSERT INTO postLike VALUES (?, ?);
```

- **Parameters for query:** (personId, postId)

In order to update the data in the postLike table the personId (foreign key for Person.id) and postId (foreign key for Post.id) must be passed in as arguments for the method. Ideally the use of the query in the API would be as follows: Person "likes" the post, this triggers a call of the method recordLike ( ). This method would take the personId and the postId as the arguments which are then used in this query as seen above. If the post is deleted, the "likes" can be removed by passing in the postLike.postId value. If the users account is deleted, similarly the "likes" can be easily removed by passing in the postLike.personId to do so.

By designing the postLike feature like this, it means that inserting the data is simple. Moreover, the removal of any of this data does not affect any of the other tables. This is because the database schema is normalised, thus ensuring none of the anomalies are encountered. Also, due to the design of this feature very little code would have to be modified on the database end to implement it due to the simplicity of the queries, and the few parameters that are necessary.

Getting names of all people who have liked a specific topic, ordered alphabetically:

```
SELECT Person.name AS name FROM topicLike
INNER JOIN Person ON LikeTopic.personId = Person.id
WHERE topicLike.topicId = ?
ORDER BY Person.name ASC;
```

- **Parameters for query:** (topicId)

This query allows for the names of the users to queried based on the likes on the topic, they are ordered alphabetically. The use of this query in the API is as follows: getTopicLikes( ) method implemented, which takes the topicId (foreign key for Topic.id) as the argument for the method. Due to the fact that personId is a foreign key for the Person table and topicId is a foreign key for the Topic table, this task can be implemented with one simple query.

This query uses an INNER JOIN due to the fact that if a Topic has no likes, it would return an empty set instead of a single null result. Moreover, similarly to the query above, in order to implement this functionality in the API very little would have to be changed. This query does not require accessing any of the other methods or SQL queries, and only requires for the topicId for the requested Topic to be passed as a parameter. Due to the same reasons as for the query above, as the database is normalised none of the anomalies are encountered.

Deploying new schema functionality to an existing application can be dangerous. Why? How does your schema design mitigate these issue(s)? (max. 1 page)

The potentially dangerous aspects of deploying new schema functionality to an existing application will be considered first, before outlining how the schema design mitigates these issues. While adding brand new columns and tables tends to be safe, primarily due to the fact that it does not affect any existing data, one potential issue is that previous tables may be dropped, and data could be deleted when adding the new tables. Likely this could occur if the create/drop script drops the previous existing tables. To mitigate this, task3.sql (the create / drop script for adding the "like" functionality) does not drop any of the existing tables. This ensures that there is no existing data lost when adding the two new tables. Moreover, another potential issue is that the data is updated in one table but not another.[5] This occurs when the data is replicated across tables. For example, if the username of a Person is updated, and this were to be included in a second table but not updated there, this would create inconsistencies across the database. However, as the primary keys for all tables (and the primary keys for the new PostLike and TopicLike tables) are generated using surrogate keys rather than candidate data, this ensures that this will not be an issue as the id number

[5] https://www.martinfowler.com/articles/evodb.html

are never accessed and do not rely on data that may change. Moreover, as this database is normalised there are no functional dependencies and none of the attributes are required to change together. Thus, the insertion of the two new like tables does not affect any existing data, nor would it be affected by the removal of existing data in the future. Another potential issues with deploying new schema functionality to an existing application is that the renaming or removing of columns is dangerous.[6] This is because if there were to be existing functional dependencies, it would take care to ensure that the other data is not adversely affected by the addition. However, as this database is normalised to BCNF and this has not changed with the addition of the like tables, if any tables were to be renamed or removed then existing attributes will not be affected. Regardless, the addition of the two like tables does not require the altering of any existing data. Finally, another potential issue with deploying new schema functionality to an already existing application is that the design of the new functionalities is implemented with little care or thoughtful planning. This can affect the normalisation of the database and means that certain issues are avoided by simply hacking away at them.[7] It is argued that the addition of the two tables is the best possible implementation of the "like" functionality. While it would be possible to include one like table to documents all likes, this could create functional dependencies between the like type and the id numbers. Thus, affecting the normalisation of the database. By editing the schema with careful attention to these issues, future problems can be avoided due to the fact that the schema was implemented with proper planning and attention to the aforementioned problems.

[6] https://medium.com/better-programming/evolving-a-database-schema-10b7f4094d14
[7] https://www.red-gate.com/simple-talk/sql/database-administration/ten-common-database-design-mistakes/