TDT4145 - Project

Simen Omholt-Jensen, Anna Zhang, Vebjørn Steinsholt

March 12, 2021

Contents

1 ER Diagram **2**
1.1 Assumptions 3

2 Relational Schemas **5**

3 Use Case Process Description **12**
3.1 Use Case 1: 12
3.2 Use Case 2 12
3.3 Use Case 3 14
3.4 Use Case 4 15
3.5 Use Case 5 16

4 SQL **17**

1 ER Diagram

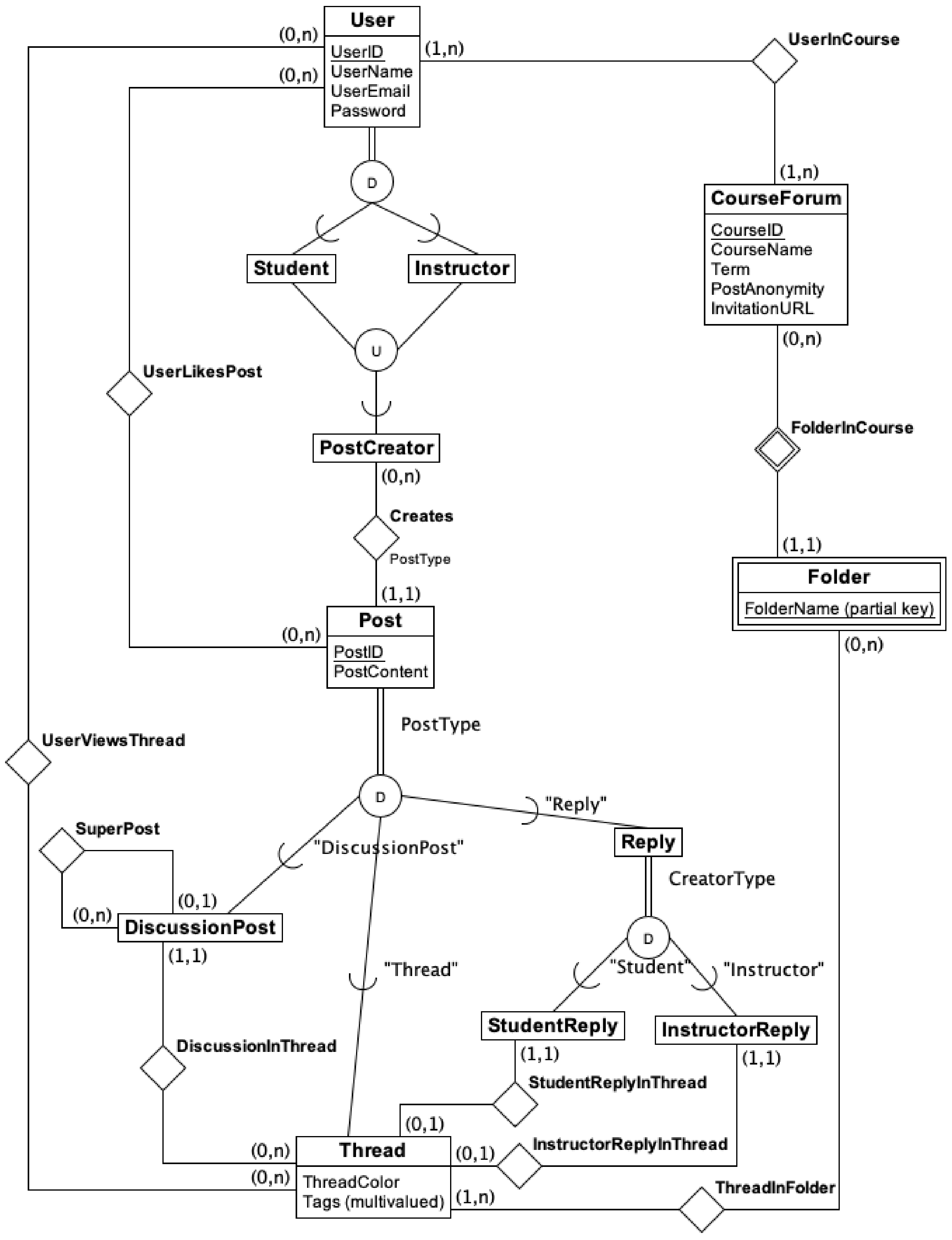

Figure 1: ER Diagram showing the complete data model.

1.1 Assumptions

Overall assumptions :

- A post, as referenced in the project description, can either be of the type thread, reply, or a discussion post. A thread can contain none, some or all of the other post types. Thus, when a user creates a post, he needs to specify which type of post he is about to create.
- Although one could imagine a simpler design for the different types of posts, the current post design was chosen as it most closely resembles the functionality of Piazza. Students can collaborate on a student reply. Instructors can collaborate on instructor replies, and there also exists additional followup discussions and comments.
- Keyword search will return the post ids of keyword matches in the post content, in the folder name of which the post belongs, in the post tags, and in the user name of the user who created the post. Keyword search will cover all types of posts.

User :

- All users (**User**) must have a unique ID (UserID).
- All users must also have a name (*UserName)*, an email (*UserEmail)* and a password (*Password*).
- Multiple users can have the same user name. Same goes for passwords. Emails are unique.
- A user must participate (`UserInCourse`) in at least one course forum (**CourseForum**), but can participate in many course forums.
- A user can like (`UserLikesPost`) zero or many posts (**Post**). Likes are recorded for every post.
- A user can view (`UserViewsThread`) zero or many threads (**Thread**). Views are not recorded for every post type, but only for the threads.
- All users are either a student (**Student**) or an instructor (**Instructor**).

Student :

- A student is a specialization of a user.

Instructor :

- An instructor is a specialization of a user.
- An instructor can create, edit and delete folders. These functionalities are not modeled in the ER diagram as they will be handled by the program implementation.
- An instructor can change the post anonymity for a course forums. This functionality will be handled by the program implementation.
- An instructor can invite people to participate in a course forums. It is assumed that an instructor can only invite people to courses he participates in. This functionality will be handled by the program implementation.

CourseForum :

- A course forum must have a unique id (CourseID), a name (*CourseName*), a term (*Term*), a post anonymity setting (*PostAnonymity*), and an invitation URL (*InvitationURL*).
- A course forum must have at least one user, but can have many users.
- A course forum can have (`FolderInCourse`) zero or many folders.

Folder :

- A folder is identified by its name (*FolderName*) as a partial key and has an identifying course forum entity type.
- Although the project description specifies the existence of sub folders, they were disregarded as there is no need for a file hierarchy in this application.
- A folder is related to exactly one course forum. A folder is only unique for a course

forum because of the weak identity.
- A folder can have zero or many threads (`ThreadInFolder`).

PostCreator :
- A post creator (**PostCreator**) is a category of students and instructors.
- A post creator can create zero or many posts (**Post**).
- A post creator specifies the type of post (*PostType*) when creating a post.

Post :
- A post (**Post**) must have a unique id (*PostID*). It also has post content (*PostContent*).
- A post can be liked by zero or many users.
- All posts are either a discussion (**DiscussionPost**), a thread (**Thread**), or a reply (**Reply**).

DiscussionPost :
- A discussion post is a specialization of a post with a post type of "Discussion".
- A discussion post can have zero or one super posts (`SuperPost`).
- A super post can have zero or many sub discussion posts.
- A discussion post can exist in (`DiscussionInThread`) exactly one thread.

Reply :
- A reply is a specialization of post with a post type of "Reply".
- All replies are either a student reply (**StudentReply**) or an instructor reply (**InstructorReply**).

StudentReply :
- A student reply is a specialization of a reply with a post creator type (*CreatorType*) of "Student". This creator type is derived using the post creator attribute in post and joining with post creator (see relational schemas).
- A student reply can exist in (`StudentReplyInThread`) exactly one thread. This is to model the Piazza behavior where students collaborate in a single post. How to deal with editing and updates of existing replies is handled by the programming implementation.

InstructorReply :
- An instructor reply is a specialization of a reply with a post creator type of "Instructor". This creator type is derived using the post creator attribute in post and joining with post creator (see relational schemas).
- An instructor reply can exist in (`InstructorReplyInThread`) exactly one thread. This is to model the Piazza behavior where instructors collaborate in a single post. How to deal with editing and updates of existing replies is handled by the programming implementation.

Thread :
- A thread is a specialization of post with a post type of "Thread".
- A thread gets a color (*ThreadColor*) depending on whether it has a reply or not. Instructor replies and student replies have different colors. The color code is only recorded for threads, and not for every post type.
- A thread also has a multivalued attribute for tags (*Tags*). A thread is the only post type that gets to have tags.
- A thread can be viewed by zero or many users.
- A thread can be categorized in (`ThreadInFolder`) at least one folder, but can be categorized in many folders.
- A thread can have zero or many discussion posts.
- A thread can have zero or one student reply.
- A thread can have zero or one instructor reply.

2 Relational Schemas

User

UserID	*UserName*	*UserEmail*

- UserID is the primary key.
- **Normalization**:
  - Nontrivial functional dependencies: $\{$ UserID $\rightarrow$ *UserName*, *UserEmail*$\}$
  - Nontrivial multivalued dependencies: $\{$ UserID $\twoheadrightarrow$ *UserName*, UserID $\twoheadrightarrow$ *UserEmail* $\}$
  - Primary key: $\{$ UserID $\}$
  - Nonprime attributes: $\{$*UserName*, *UserEmail*$\}$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **User** is also a superkey for **User**. Here, this left hand side attribute is UserID.
  - The table is on 4NF because it is on BCNF and because every left hand side attribute of every nontrivial multivalued dependency in **User** is also a superkey for **User**. Here, this left hand side attribute is UserID.

Login

UserEmail	*Password*

- UserEmail is the primary key.
- UserEmail is a foreign key to **User**.
- **Normalization**:
  - Nontrivial functional dependencies: $\{$ UserEmail $\rightarrow$ *Password*$\}$
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: $\{$ UserEmail $\}$
  - Nonprime attributes: $\{$*Password*$\}$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **Login** is also a superkey for **Login**. Here, this left hand side attribute is UserEmail.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

Note: The **User** entity was composed into two tables (**User** and **Login**) to ensure 4NF.

Student

StudentID	*PCID*

- StudentID is the primary key for the **Student** specialization of **User**.
- *PCID* is a surrogate key to **PostCreator**
- **Normalization**:
  - Nontrivial functional dependencies: { StudentID $\rightarrow$ *PCID*}
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { StudentID }
  - Nonprime attributes: {*PCID*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **Student** is also a superkey for **Student**. Here, this left hand side attribute is StudentID.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

Instructor

InstructorID	*PCID*

- InstructorID is the primary key for the **Instructor** specialization of **User**.
- *PCID* is a surrogate key to **PostCreator**
- **Normalization**:
  - Nontrivial functional dependencies: { InstructorID $\rightarrow$ *PCID*}
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { InstructorID }
  - Nonprime attributes: {*PCID*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **Instructor** is also a superkey for **Instructor**. Here, this left hand side attribute is InstructorID.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

PostCreator

PCID	*CreatorType*

- PCID is the primary key.
- *CreatorType* is the category type.
- **Normalization**:
  - Nontrivial functional dependencies: { PCID $\to$ *CreatorType*}
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { PCID }
  - Nonprime attributes: {*CreatorType*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **PostCreator** is also a superkey for **PostCreator**. Here, this left hand side attributes is PCID.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

Post

PostID	*PostContent*	*PCID*	*PostType*

- PostID is the primary key.
- *PCID* is a foreign key to **PostCreator**.
- **Normalization**:
  - Nontrivial functional dependencies: { PCID $\to$ *PostContent*, *PCID* , *PostType*}
  - Nontrivial multivalued dependencies: { PostID $\twoheadrightarrow$ *PostContent*, PostID $\twoheadrightarrow$ *PCID*, PostID $\twoheadrightarrow$ *PostType*}
  - Primary key: { PCID }
  - Nonprime attributes: {*PostContent*, *PCID*, *PostType*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **Post** is also a superkey for **Post**. Here, this left hand side attributes is PostID.
  - The table is on 4NF because it is on BCNF and because every left hand side attribute of every nontrivial multivalued dependency in **Post** is also a superkey for **Post**. Here, this left hand side attribute is PostID.

UserLikesPost

UserID	PostID

- UserID, PostID is the primary key.
- UserID is a foreign key to **User**.
- PostID is a foreign key to **Post**.
- **Normalization**:
  - Nontrivial functional dependencies: $\emptyset$
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { UserID, PostID }
  - Nonprime attributes: $\emptyset$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because as there are no nonprime attributes.
  - The table is on BCNF because it is on 2NF and because there are no functional dependencies between the attributes.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

DiscussionPost

DiscussionPostID	*SuperPostID*	*ThreadID*

- DiscussionPostID is the primary key for the **DiscussionPost** specialization of **Post**.
- *SuperPostID* is a foreign key to **DiscussionPost** to model the recursive relationship.
- *ThreadID* is a foreign key to the **Thread** specialization of **Post**.
- **Normalization**:
  - Nontrivial functional dependencies: { DiscussionPostID $\rightarrow$ *SuperPostID*, *ThreadID*}
  - Nontrivial multivalued dependencies: { DiscussionPostID $\twoheadrightarrow$ *SuperPostID*, DiscussionPostID $\twoheadrightarrow$ *ThreadID*}
  - Primary key: { DiscussionPostID }
  - Nonprime attributes: {*SuperPostID*, *ThreadID*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **DiscussionPost** is also a superkey for **DiscussionPost**. Here, this left hand side attributes is DiscussionPostID.
  - The table is on 4NF because it is on BCNF and because every left hand side attribute of every nontrivial multivalued dependency in **DiscussionPost** is also a superkey for **DiscussionPost**. Here, this left hand side attribute is DiscussionPostID.

Thread

ThreadID	*ThreadColor*	*StudentReplyID*	*InstructorReplyID*

- **ThreadID** is the primary key for the **Thread** specialization of **Post**.
- *StudentReplyID* is a foreign key to the **StudentReply** specialization of **Reply** which is a specialization of **Post**.
- *InstructorReplyID* is a foreign key to the **InstructorReply** specialization of **Reply** which is a specialization of **Post**.
- **Normalization**:
  - Nontrivial functional dependencies: { ThreadID $\rightarrow$ *ThreadColor*, *StudentReplyID*, *InstructorReplyID*}
  - Nontrivial multivalued dependencies: { ThreadID $\twoheadrightarrow$ *ThreadColor*, ThreadID $\twoheadrightarrow$ *StudentReplyID*, ThreadID $\twoheadrightarrow$ *InstructorReplyID*}
  - Primary key: { ThreadID }
  - Nonprime attributes: {*ThreadColor*, *StudentReplyID*, *InstructorReplyID*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **Thread** is also a superkey for **Thread**. Here, this left hand side attributes is ThreadID.
  - The table is on 4NF because it is on BCNF and because every left hand side attribute of every nontrivial multivalued dependency in **Thread** is also a superkey for **Thread**. Here, this left hand side attribute is ThreadID.

Tags

ThreadID	Tag

- **Tags** is a multivalued attribute for **Post**.
- ThreadID, Tag is the primary key.
- ThreadID is a foreign key to **Post**.
- **Normalization**:
  - Nontrivial functional dependencies: { ThreadID $\rightarrow$ Tag}
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { ThreadID, Tag }
  - Nonprime attributes: $\emptyset$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because as there are no nonprime attributes.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **Tags** is also a superkey for **Tags**. Here, this left hand side attribute is ThreadID.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

ThreadInFolder

ThreadID	CourseID	FolderName

- ThreadID, CourseID, FolderName is the primary key.
- ThreadID is a foreign key to the **Thread** specialization of **Post**.
- CourseID, FolderName is a foreign key to **Folder**.
- **Normalization**:
  - Nontrivial functional dependencies: $\emptyset$
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { ThreadID, CourseID, FolderName }
  - Nonprime attributes: $\emptyset$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because as there are no nonprime attributes.
  - The table is on BCNF because it is on 2NF and because there are no functional dependencies between the attributes.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies.

UserViewsThread

UserID	ThreadID

- UserID, ThreadID is the primary key.
- UserID is a foreign key to **User**.
- ThreadID is a foreign key to the **Thread** specialization of **Post**
- **Normalization**:
  - Nontrivial functional dependencies: $\emptyset$
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { UserID, ThreadID }
  - Nonprime attributes: $\emptyset$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because as there are no nonprime attributes.
  - The table is on BCNF because it is on 2NF and because there are no functional dependencies between the attributes.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

CourseForum

CourseID	*CourseName*	*Term*	*PostAnonymity*	*InvitationURL*

- CourseID is the primary key.
- **Normalization**:
  - Nontrivial functional dependencies: { CourseID $\rightarrow$ *CourseName*, *Term*, *PostAnonymity*, *InvitationURL*}
  - Nontrivial multivalued dependencies: { CourseID $\twoheadrightarrow$ *CourseName*, CourseID $\twoheadrightarrow$ *Term*, CourseID $\twoheadrightarrow$ *PostAnonymity*, CourseID $\twoheadrightarrow$ *InvitationURL*}
  - Primary key: { CourseID }
  - Nonprime attributes: {*CourseName*, *Term*, *PostAnonymity*, *InvitationURL*}
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because every nonprime attribute is fully functionally dependent on the primary key.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **CourseForum** is also a superkey for **CourseForum**. Here, this left hand side attribute is CourseID.
  - The table is on 4NF because it is on BCNF and because every left hand side attribute of every nontrivial multivalued dependency in **CourseForum** is also a superkey for **CourseForum**. Here, this left hand side attribute is CourseID.

UserInCourse

UserID	CourseID

- UserID, CourseID are together the primary key.
- UserID is a foreign key to **User**.
- CourseID is a foreign key to **CourseForum**.
- **Normalization**:
  - Nontrivial functional dependencies: { UserID $\rightarrow$ CourseID, CourseID $\rightarrow$ UserID }
  - Nontrivial multivalued dependencies: $\emptyset$
  - Primary key: { UserID, CourseID }
  - Nonprime attributes: $\emptyset$
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because as there are no nonprime attributes.
  - The table is on BCNF because it is on 2NF and because every left hand side attribute of every nontrivial functional dependency in **UserInCourse** is also a superkey for **UserInCourse**. Here, these left hand side attributes are UserID and CourseID.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies. At least three attributes are required for multivalued dependencies.

Folder

CourseID	FolderName

- CourseID, FolderName is the primary key.
- CourseID is a foreign key to **CourseForum**.
- **Normalization**:
  - Nontrivial functional dependencies: ∅
  - Nontrivial multivalued dependencies: ∅
  - Primary key: { CourseID, FolderName }
  - Nonprime attributes: ∅
  - The table is on 1NF because the table is defined only for atomic values.
  - The table is on 2NF because it is on 1NF and because as there are no nonprime attributes.
  - The table is on BCNF because it is on 2NF and because there are no functional dependencies between the attributes.
  - The table is on 4NF because it is on BCNF and because there are no multivalued dependencies.

3 Use Case Process Description

3.1 Use Case 1:

When a user, in this case a student, is about to log in, he is prompted for an email. When entered, the program queries the **Login** table to check whether the entered email exists in the *UserEmail* attribute column.

```
SELECT EXISTS(SELECT * FROM Login WHERE UserEmail=<user_entered_email>);
```

If the result is 0, meaning that it does not exist in the table, the user is shown an error message, and is asked to reenter the email. If the result is 1, meaning that it does exist in the table, the user is prompted for a password. When entered, the program queries the **Login** table where the *UserEmail* attribute column is equal to the entered email and checks if the corresponding *Password* attribute column value is equal to the entered password.

```
SELECT EXISTS(SELECT * FROM Login WHERE UserEmail=<user_entered_email>
                                  AND
                                  Password=<user_entered_password>);
```

If the result is 0, the user is shown an error message and is asked to reenter the password. If the result is 1, the user is logged in.

This completes the process description of the first use case.

3.2 Use Case 2

When a user, in this case a student, is about to create a post, he is prompted for which post type to create which can be one of the following values: ”Thread”, ”Reply”, ”DiscussionPost”. The default value is ”Thread”. For this use case, the user enters the default value. The user is then prompted for a post content text. Once entered, the user is prompted for a folder. The user enters ”Exam”, and is finally prompted for a post tag where the user enters ”Question”.

The following process description assumes that the entered *FolderName* attribute value ”Exam” exists in the database. If this is not the case, the program would need to handle this by either adding ”Exam” as a new folder, or give the user an error message. This will be implemented as part of the program. It is also assumed that information about which course the user wishes to create a post in is know, i.e. the relevant *CourseID* attribute value in **CourseForum** is known.

After the entered user input, the program creates a new post by inserting a new tuple into the **Post** table that looks like the following:

- (PostID, *PostContent*, *PCID*, *PostType*)

A new unique value is created for the PostID attribute. The value for the *PostContent* attribute is set to the user entered post content text. The program then checks whether the current user id (*UserID*) exist in the **Student** table

```
SELECT EXISTS(SELECT * FROM Student WHERE StudentID=<current_userid>);
```

or in the **Instructor** table.

```
SELECT EXISTS(SELECT * FROM Instructor WHERE InstructorID=<current_userid>);
```

The program finds a match in the **Student** table and selects the corresponding *PCID* attribute value.

```
SELECT PCID FROM Student WHERE StudentID=<current_userid>;
```

The resulting *PCID* attribute value is then inserted into the above **Post** tuple as the *PCID* attribute value. The *PostType* attribute value will be the post type entered by the user. In this case, the default ”Thread” value.

```
INSERT INTO Post VALUES(<generated_postid>, <user_entered_post_content>, <pcid>, "Thread");
```

Next, the program inserts a new tuple in the **Thread** table that looks like the following:

- (ThreadID, *StudentReplyID*, *InstructorReplyID*, *ThreadColor*)

Here, the ThreadID attribute value is the newly generated PostID attribute value from above. The *StudentReplyID* attribute value and the *InstructorReplyID* attribute value are both set to `NULL` as there has been no replies yet. Lastly, the *ThreadColor* attribute value is set to the default value of 0. The *ThreadColor* attribute has three values (0,1,2) depending on whether the thread has received any replies, and depending on if the reply was from a student or an instructor.

```
INSERT INTO Thread VALUES(<generated_postid>, NULL, NULL, 0);
```

Next, the program inserts a new tuple into the **Tags** table that looks like the following:

- (ThreadID, Tag)

Here, the ThreadID attribute value will be the unique value generated above for the new post. The Tag attribute value will be the user entered tag value ”Question”.

```
INSERT INTO Tag VALUES(<generated_postid>, "Question");
```

Finally, the program inserts a new tuple into the **ThreadInFolder** table that looks like the following:

- (ThreadID, CourseID, FolderName)

Here, the ThreadID attribute value is the same value as inserted in the **Thread** table. As information about the course was assumed to be known, the *CourseID* attribute value will be the known course id for the current user. The *FolderName* attribute value will be the user entered ”Exam” value.

```
INSERT INTO ThreadInFolder VALUES(<generated_postid>, <known_courdid>, "Exam");
```

This completes the process description of the second use case.

3.3 Use Case 3

An instructor replies to a post belonging to the folder “Exam”. The input to this is the id of the post replied to. This could be the post created in use case 2.

When the user, in this case an instructor, is about to create a post, in this case a reply, he is prompted for which post type to create which can be one of the following values: ”Thread”, ”Reply”, ”DiscussionPost”. For this use case, the user enters ”Reply”. The user is then prompted for a post content text which the user enters. Finally, the user is prompted for a post id. It is assumed that the user enters the post id of the post generated in the second use case.

The program then proceeds by creating a post by inserting a new tuple into the **Post** table that looks like the following:

- (PostID, *PostContent*, *PCID*, *PostType*)

A new unique value is created for the PostID attribute. The value for the *PostContent* attribute is set to the user entered post content text. The program then checks whether the current user id (*UserID*) exist in the **Student** table

```
SELECT EXISTS(SELECT * FROM Student WHERE StudentID=<current_userid>);
```

or in the **Instructor** table.

```
SELECT EXISTS(SELECT * FROM Instructor WHERE InstructorID=<current_userid>);
```

The program finds a match in the **Instructor** table and selects the corresponding *PCID* attribute value.

```
SELECT PCID FROM Instructor WHERE InstructorID=<current_userid>;
```

The resulting *PCID* attribute value is then inserted into the above **Post** tuble as the *PCID* attribute value. The *PostType* attribute value will be the post type entered by the user. In this case, the ”Reply” value.

```
INSERT INTO Post VALUES(<generated_postid>, <user_entered_post_content>, <pcid>, "Reply");
```

As the ”Reply” value was entered, the program then filters the **Post** table on where PostID equals the PostID attribute value created in the above **Post** and joins the result with the **PostCreator** table on the *PCID* attribute. It then selects the *CreatorType* from the result which will yield a value of ”Instructor”.

```
SELECT  CreatorType
FROM
(SELECT  *  FROM  Post  WHERE  PostID=<generated_postid>)  JOIN  PostCreator  USING  (PCID);
```

From this, the program knows that the created reply is an instructor reply (*InstructorReply*).

Next, the program filters the **Thread** table for where the **ThreadID** attribute value equals the user entered post id. Having found the tuple of interest, the program then updates the *InstructorReplyID* attribute value from `NULL` to the PostID attribute value generated in the above **Post**.

Finally, as the thread now has a reply, the program updates the *ThreadColor* attribute value in the tuple of interest found above from 0 to 2. This value signals that the thread has a reply, and that a instructor replied.

```
UPDATE  Thread
SET  InstructorReplyID=<generated_postid>,  ThreadColor=2
WHERE  ThreadID=<user_entered_id>;lp
```

This completes the process description of the third use case.

3.4 Use Case 4

As stated in the overall assumptions, keyword search will return the post ids of keyword matches in the post content, in the folder name of which the post belongs, in the post tags, and in the user name of the user who created the post. Keyword search will cover all types of posts.

When a user, in this case a student, is about to search for a post, he is prompted for a search keyword. In this case, the user enters the keyword ”WAL”.

The query consists of four sub queries:

1. **Search in post content:** The program queries the **Post** table for where the *PostContent* attribute values includes ”WAL” and selects the resulting **PostID** attribute values.

```
SELECT  PostID  FROM  Post  WHERE  PostContent  LIKE  "%WAL%";
```

2. **Search in tags:** The program queries the **Tag** table for where the *Tag* attribute values includes ”WAL” like before and selects the resulting **ThreadID** attribute values.

```
SELECT  ThreadID  AS  PostID  FROM  Tag  WHERE  Tag  LIKE  "%WAL%";
```

3. **Search for posts in folder:** The program then queries the **Folder** table for the known CourseID and for where the FolderName include ”WAL” like before. The resulting CourseID and the FolderName is then joined with the **ThreadInFolder** table on CourseID and FolderName and the resulting ThreadID attribute values are selected.

```
SELECT  ThreadID  AS  PostID
```

```
FROM
ThreadInFolder
JOIN
(SELECT CourseID, FolderName FROM Folder WHERE FolderName LIKE %WAL%)
USING (CourseID, FolderName);
```

4. **Search for posts by people:** The program queries the **User** table for where the *UserName* attribute values includes ”WAL” like before. The resulting table is then joined with the **Student** table where UserID equals StudentID. The program then takes the union of this resulting table with the **User** table joined with the **Instructor** table where UserID equals InstructorID. The resulting table consist of instructors and students whose name includes ”WAL” and their *PCID* attribute values. This table is then joined with the **Post** table on *PCID* and the resulting PostID attribute values are then selected.

```
SELECT PostID
FROM Post

JOIN

((SELECT PCID FROM
Student
JOIN
(SELECT UserID FROM User WHERE UserName LIKE "%WAL%")
ON (UserID=StudentID))

UNION

(SELECT PCID FROM
Instructor
JOIN
(SELECT UserID FROM User WHERE UserName LIKE "%WAL%")
ON (UserID=InstructorID)))

USING (PCID);
```

The program then takes the union of the resulting post ids to remove duplicate post ids and returns the result to the user.

```
SELECT DISTINCT PostID
FROM
<search_query_1>
UNION
<search_query_2>
UNION
<search_query_3>
UNION
<search_query_4>;
```

This completes the process description of the fourth use case.

3.5 Use Case 5

The program solves this query by first left outer joining **User** with **UserViewsThread** using the UserID and grouping the result by UserID while selecting *UserName*, UserID and the count of ThreadID. This gives the number of posts viewed by all users.

```
SELECT UserName, UserID, Count(ThreadID) AS NumberOfPostsRead
FROM User LEFT OUTER JOIN UserViewsThread USING (UserID) GROUP BY (UserID)
```

To find the number of posts created, the program joins the **Student** table with the **User** table on where UserID equals StudentID and selects the *PCID*. The program then takes the union of the resulting table with the join of **Instructor** and **User** on where InstructorID equals UserID while selecting the *PCID*. This table is then joined with the **Post** table on *PCID*. From this result, the program groups by the UserID while selecting the *UserName*, UserID, and the count of PostID.

```
SELECT UserName, UserID, Count(PostID) AS NumberOfPostsCreated
FROM Post
JOIN
((SELECT PCID FROM Student JOIN User ON (UserID=StudentID))
UNION
(SELECT PCID FROM Instructor JOIN User ON (UserID=InstructorID)))
USING (PCID)
GROUP BY (UserID)
```

The first query is then left outer joined with the second query using the UserID, and from this query the *UserName*, NumberOfPostsRead and NumberOfPostsCreated are selected in descending order of NumberOfPostsRead.

```
SELECT UserName, NumberOfPostsRead, NumberOfPostsCreated

FROM

(SELECT UserName, UserID, Count(ThreadID) AS NumberOfPostsRead
FROM User LEFT OUTER JOIN UserViewsThread USING (UserID) GROUP BY (UserID)) AS Temp1

LEFT OUTER JOIN

(SELECT UserName, UserID, Count(PostID) AS NumberOfPostsCreated
FROM Post
JOIN
((SELECT PCID FROM Student JOIN User ON (UserID=StudentID))
UNION
(SELECT PCID FROM Instructor JOIN User ON (UserID=InstructorID)))
USING (PCID)
GROUP BY (UserID)) AS Temp2

USING (UserID)

ORDER BY NumberOfPostsRead DESC;
```

4 SQL

See attached `TDT4145_Project_2021.txt` file.