

Project 1
TDT 4145

Sara Elise Wøllo, Thomas Torkildsen, Markus Trætli

March 2021

1 Introduction

In this project, we are creating a Piazza-forum. This forum has the functionality that we are used to from our studies. Where we have made assumptions about the functionality of the forum, we have specified these. To build our forum, we have first made an ER-model, we have then presented a relational database for our Piazza forum which we have implemented using SQL. Lastly, we have looked at a number of different use-cases of our Piazza-forum, and shown how these problems are solved in the database. Later, we will implement the Piazza-forum.

2 Entity-Relationship model

2.1 The ER-model

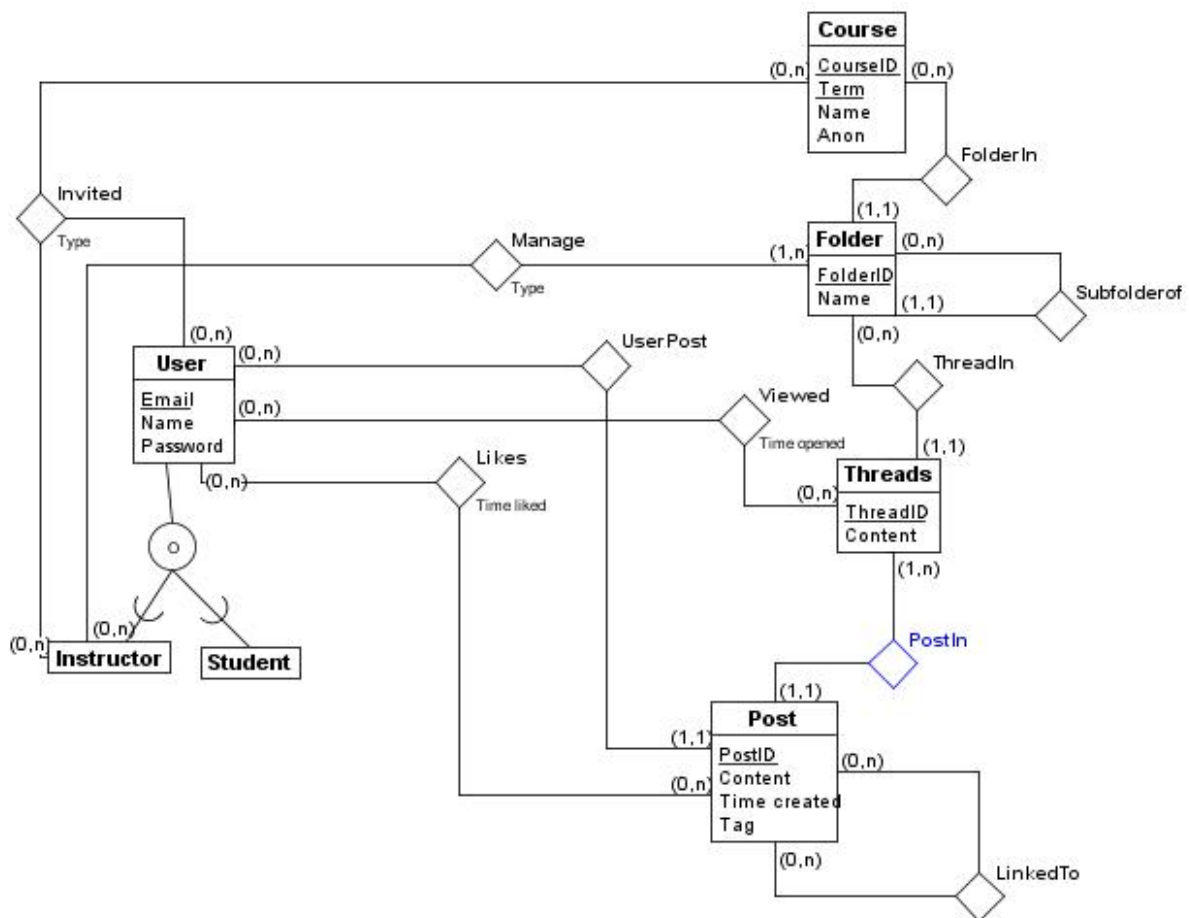


Figure 1: Entity-Relationship model for our Piazza-database

2.2 Assumptions

- A user can be an instructor in one course and a student in another. For instance, the teaching assistants are instructors in the courses they are TAs for, and students in their regular courses.
- A instructor can invite all users to all courses where he is instructor. *Type* defines the type of invitation (student or instructor).
- There might exist users that are not students nor instructors, for example Admin-people.
- Once a course is created the anonymity setting (*Anon*) can not be changed.
- Each course can have from zero to several folders, but a folder can only belong to one course.
- Each folder can have from zero to several subfolders, but each subfolder can only belong to one folder.
- Each folder can have from zero to several threads, but a thread can only belong to one folder.
- Each thread can have one to several posts, but a post can only belong to one thread.
- Each post can only have one tag.
- Every post can be liked by several users and a user can like several posts. However, a user can only like one post one time, but this is something that is handled in our database system.
- Every user can post several posts, but a post can only be posted by one user.
- Every folder must be managed once (every folder must have been created) or several times, but not every instructor must manage a folder.
- Once a thread is viewed every post in the thread is automatically viewed as well. This is because once you click on a thread in Piazza, all the posts in the thread are automatically shown on screen.
- A post can link to arbitrarily many posts and the other way around.

3 Mapping the ER-model to a relational database

When choosing the different tables we need in our database, we have focused on all our tables fulfilling the forth Normal Form (4NF).

We have ended up with a database model with 15 tables. The names of these tables are defined in table 1.

Table name
Course
Folder
Thread
Post
Likes
Viewed
Manage
EnrolledIn
Subfolder
UserPost
LinkedTo
UserProfile
FolderInCourse
ThreadInFolder
PostInThread

Table 1: Tables in our relational database

We have defined the different tables below, where the primary keys (or combinations of primary keys) are underlined in the table with a continuous line, and the foreign keys are underlined with a dashed line.

If an attribute is both a primary and a foreign key, it is underlined with both a continuous and dashed line.

The first table we define is called **Course**. This is shown in table 2

<u>CourseID</u>	<u>Term</u>	CourseName	Anon
-----------------	-------------	------------	------

Table 2: **Course**

Here, the primary key is (*CourseID*, *Term*). The reason why we need both of these to be the primary key, is because a course has the same ID from year to year, but the lecturer may decide to create a new Piazza forum each year. These two together define a unique primary key. *Anon* denotes whether the instructor has made it possible for the participants in the course to post anonymously in the forum. We have assumed this to be a setting that the instructor can switch on when they make the forum, and that it is not possible to change this setting once the Piazza forum is created. We assume that *CourseName* and *Anon* are independent of each other and functionally dependent on (*CourseID*, *Term*), thus we have 4NF.

We then define the table **Folder** in table 3,

<u>FolderID</u>	Name
-----------------	------

Table 3: **Folder**

The *FolderID* is the primary key for this table and *Name* is functionally dependent on *FolderID*. For the **Thread**-table, shown in table 4,

<u>ThreadID</u>	Title
-----------------	-------

Table 4: **Thread**

we have a primary key consisting of *ThreadID*. *Title* is functionally dependent on *ThreadID*. It is trivial that both table 3 and table 4 is on 4NF.

<u>PostID</u>	Content	PostCreatedAt	Tag
---------------	---------	---------------	-----

Table 5: **Post**

For the **Post**-table, we have a primary key consisting of one attribute *PostID*. You can only have one tag connected to each post, saved in *Tag*. This attribute can also have a NULL-value. *Content*, *PostCreatedAt* and *Tag* are all functionally dependent on *PostID* and independent of each other, thus we have 4NF in this table as well.

We need to specify the users of the Piazza forum, and information about them are found in **User-Profile**, in table 6.

<u>Email</u>	UserName	Password
--------------	----------	----------

Table 6: **UserProfile**

Here, *Email* is the primary key and *UserName* and *Password* are functionally dependent on *Email*. Since several users can have the same password, but different username and the other way around we have functional independence between *UserName* and *Password*. Thus we have 4NF.

The next table in our database is **Likes**, seen in table 7.

<u>PostID</u>	<u>Email</u>	TimeLiked
---------------	--------------	-----------

Table 7: **Likes**

This table keeps track of instances where a person "Likes" a post, which post they "Liked", and when they "Liked" the post. Here, (*PostID*, *Email*) all together is the primary key of the table. *Email* is a foreign key referencing the **UserProfile**-table, and *PostID* is a foreign key to the **Post**-table. *TimeLiked* is functionally dependent on *PostID* and *Email*

We then have a table called **Viewed**. This table keeps track of each instance where a user views a thread (and by opening the thread views all the posts in the thread), and logs this. This is used to generate statistics about user and post activity. The table is shown in table 8.

<u>ThreadID</u>	<u>Email</u>	TimeOpened
-----------------	--------------	------------

Table 8: **Viewed**

In the **Viewed**-table, all the attributes together make up the primary key, so that is (*ThreadID*, *Email*, *TimeOpened*). This is because one user can open a thread several times, and to register all these instances we include the *TimeOpened* in the primary key. The foreign keys found in the table are *Email*, referencing **UserProfile**, and *ThreadID* referencing the **Thread**-table. There are no functional dependencies in this table.

The next table is **Manage**. This table keeps track of all users that makes changes to folders, and which folder they change. Changes can include making a new folder, deleting a folder or editing the folder. **Manage** is seen in table 9.

<u>Email</u>	<u>FolderID</u>	ActionType
--------------	-----------------	------------

Table 9: **Manage**

In the **Manage**-table, the primary key is (*Email*, *FolderID*, *ActionType*). *Email* is a foreign key referencing the **UserProfile**-table, and *FolderID* is a foreign key referencing the **Folder**-table. There are no functional dependencies in this table.

<u>UserEmail</u>	<u>CourseID</u>	<u>Term</u>	InstructorEmail	AsInstructor
------------------	-----------------	-------------	-----------------	--------------

Table 10: **EnrolledIn**

For the **EnrolledIn**-table, We use (*UserEmail*, *CourseID*, *Term*) as the primary key. Both *UserEmail* and *InstructorEmail* are a foreign keys referencing the **UserProfile**-table. (*CourseID*, *Term*) is a foreign key referencing the **Course**-table. Here we assume that *InstructorEmail* is functionally dependent on the primary key, otherwise we would loose 4NF. This means that each user can only be invited to a course once each term. *AsInstructor* is also functionally dependent on the primary key. If a user is invited to a course as student, the user can not be re-invited as instructor.

For the **FolderInCourse**-table shown in table 11,

<u>FolderID</u>	<u>CourseID</u>	<u>Term</u>
-----------------	-----------------	-------------

Table 11: **FolderInCourse**

The primary key is *FolderID*. *CourseID* and *Term* is functionally dependent on the primary key. This table has two foreign keys. *FolderID* is a foreign key referencing the **Folder**-table, and (*CourseID*, *Term*) is a foreign key referencing the **Course**-table. We have 4NF because *CourseID* and *Term* can be assumed independent.

For the **SubFolder**, in table 12, we have a primary key consisting of both attributes in the table, (*FolderID*, *SubFolderID*). This is because we might have sub-subfolders such that the *SubFolder*-attribute fail to be unique. This table signalizes that one folder is a subfolder of another folder.

<u>FolderID</u>	<u>SubFolderID</u>
-----------------	--------------------

Table 12: **Subfolder**

Both *FolderID* and *SubFolderID* are foreign keys referencing the attribute *FolderID* in the **Folder**-table.

<u>PostID</u>	<u>Email</u>
---------------	--------------

Table 13: **UserPost**

For the **UserPost**-table seen in table 13, the primary key is *PostID*, and there are two foreign keys in the table. These are *PostID* which is referencing **Post**, and *Email* referencing **UserProfile**. *Email* is functionally dependent on *PostID*.

The next table in our relational database is **LinkedTo**, and it keeps track of posts that link to other posts. This is so that one for instance can link to a post that answered a question a user asked in a different post.

<u>PostID</u>	<u>ToPostID</u>
---------------	-----------------

Table 14: **LinkedTo**

The primary key for this table is then both the attributes in the table, namely (*PostID*, *ToPostID*). The table has two foreign keys, each linking to the **Post**-folder. These are *PostID* and *ToPostID*, and they both link to the *PostID*- attribute in the **Post**-table.

To keep track of the relationship between folders and threads, we need **ThreadInFolder**, shown in table 15. This denotes which folder each thread is located, and used *ThreadID* as the primary key. Both these attributes are also foreign keys, *FolderID* references **Folder**, and *ThreadID* references **Thread**. *FolderID* is functionally dependent on *ThreadID*.

<u>FolderID</u>	<u>ThreadID</u>
-----------------	-----------------

Table 15: **ThreadInFolder**

Lastly, we need to connect which post is connected with which thread. This is done in table 16, **ThreadInPost**. In this table, *PostID* is the primary key, and *PostID* is a foreign key to **Post**, whereas *ThreadID* is a foreign key to **Thread**. *ThreadID* is functionally dependent on *PostID*.

<u>PostID</u>	<u>ThreadID</u>
---------------	-----------------

Table 16: **PostInThread**

We can conclude that all the tables are on the fourth normal form, with the assumptions made.

4 Use-cases for the database

4.1 Use case 1

In order for a user to log in, they must enter an email and a password. This information can be obtained from the user in the java-program. The login details can then be used to make a query to the **UserProfile** table and check for any matches. One such query is shown in figure 2. One is considered logged in if the query returns a match, but if it returns an empty table then the login attempt failed. If the query succeeds one will be prompted to select which courses piazza one would like to see.

```
select * from UserProfile where Email="person@mail.no" and UserPassword="passord123";
```

Figure 2: Example query for logging in

4.2 Use case 2

An instructor has made the the folder "Exam" with FolderID=1 belonging to some course. Now, for a student to make a new post in this folder, they have to be logged in and enrolled in this course. Then one has to obtain the title, content and tag from user input. Since this is a post posing a question, one will first have to make a thread and then make a post attached to that thread. We also add the time it was created and add the relation to a different table to keep track of which users wrote which posts across different forums. Using the provided information, one can insert the new post using the code shown in figure 3.

```
insert into Thread values (1,"Title"); #ThreadID, Title
insert into ThreadInFolder values (1,1); #ThreadID, FolderID. FolderID = 1 is a folder named Exam
insert into Post values(1,"Hey...", "2021-11-03-10:00:00", "Question"); #PostID, Content, TimeDate, Tag
insert into PostInThread values(1,1); #PostID, ThreadID
insert into UserPost values(1,"person@mail.com"); #PostID, Email
```

Figure 3: Insert statements that inserts a new question.

4.3 Use case 3

In order to make a reply one has to make a post in the same thread as the post one wants to reply to. Thus when making a reply, all information about folder, course, thread etc is known. Adding a reply to a post will then simply just adding another post in the same thread.

```
insert into Post values(2,"My reply is...", "2021-11-03-11:00:00", null); #PostID, Content, TimeDate, Tag
insert into PostInThread values (2,1); #PostID, ThreadID
insert into UserPost values(2,"instructor@mail.com"); #PostID, Email
```

Figure 4: Insert statements that inserts a reply.

4.4 Use case 4

In order to search for posts that contains a certain keyword, one can make a query like in figure 5. One may also add conditions for a specific *Term* and *CourseID* if one is looking in a specific piazza forum.

```
select PostID from Post
natural join PostInThread
natural join ThreadInFolder
natural join Folder
natural join FolderInCourse
natural join Course
where Content like "%WAL%" and CourseID = "TDT-4145" and Term = "V21";
```

Figure 5: Query that finds the PostID for all posts in the spring 2021 TDT-4145 piazza that contains "WAL"

4.5 Use case 5

An instructor wants to find statistics for users and how many posts they have created and read in a certain course, and sort this by number of posts viewed. The database allows for this information to be found by using a series of subqueries. Firstly, we find information about all Piazza users and the courses they are enrolled in in the **EnrolledIn**-table. In our first subquery, we use natural join to join together the **Viewed**, **ThreadInFolder** and **FolderIn**, and aggregate the resulting table on *CourseID*, *ThreadID*, *Term*, *Email* to find the number of views each user has on posts in each course they are taking. We do a similar subquery to find the number of posts each user enrolled in a course has posted in the course. We left join the two subqueries to our **EnrolledIn**-table, so that we have information on all users enrolled in all courses, the number (if any) posts they have created in each course, and the number of threads they have viewed (if any). To get the statistics for one specific course, we use a where-clause on our SQL-query, and finally we order by the number of viewed threads.

```

select tab1.Email, Nviewed, Nposted
from (select Ei.UserEmail as Email, EI.CourseID, EI.Term, NVviewed from EnrolledIn as EI
left join
    (select temp.Email,temp.CourseID ,temp.Term, temp.ThreadID, count(ThreadID) as Nviewed
    from
        (select Email, CourseID, Term, ThreadID
        from FolderInCourse
        natural join ThreadInFolder
        natural join Viewed)
        as temp group by temp.CourseID,temp.Term, temp.ThreadID, temp.email) as tab1
    on EI.UserEmail = tab1.Email and EI.CourseID = tab1.CourseID and EI.Term = tab1.Term) as tab1
left join
(select temp.Email,temp.CourseID,temp.Term,count(PostID) as Nposted
from (select Email, CourseID, Term, PostID
    from FolderInCourse natural join ThreadInFolder
    natural join Thread natural join PostInThread
    natural join Post natural join UserPost) as temp
    group by temp.CourseID, temp.email, temp.term) as tab2
on tab1.CourseID = tab2.CourseID and tab1.Email=tab2.Email
where tab1.CourseID = "TDT-4145" and tab1.Term = "H21"
order by Nviewed desc;

```

Figure 6: Query that finds the number of posts created and viewed in a given course.

5 Discussion and Conclusion

In our database-model, we emphasised getting tables on the forth normal form. This is useful, because we are then less likely to encounter problems with redundancy.

In a large database system however, we will need to break down our information into many small tables to keep the forth normal form. This makes the process of compiling information from the database more tedious, as we will need to join many tables when we need information. In a large database, this may lead the fetch time of the queries to become very large.

It might me useful to keep some information that is often used together is the same tables when the database is very large, even though this sacrifices the tables forth normal form.

This, however, must be done with caution. Lossless join is guaranteed when the tables are on Boyce-Codd Normal Form, which all our tables are as they are on forth normal form. However, if we have tables not on BCNF, then we need to make sure our tables are secured with lossless join. If this is not the case, then we might end up accidentally creating false rows in our tables when joining them together.