Project DB1: TDT4145 - Data Modelling, Databases and Database Management Systems

alexaoh, jimoskar, molderskog

March 12, 2021

a)

The ER-model showing the complete data model is shown in figure 1.

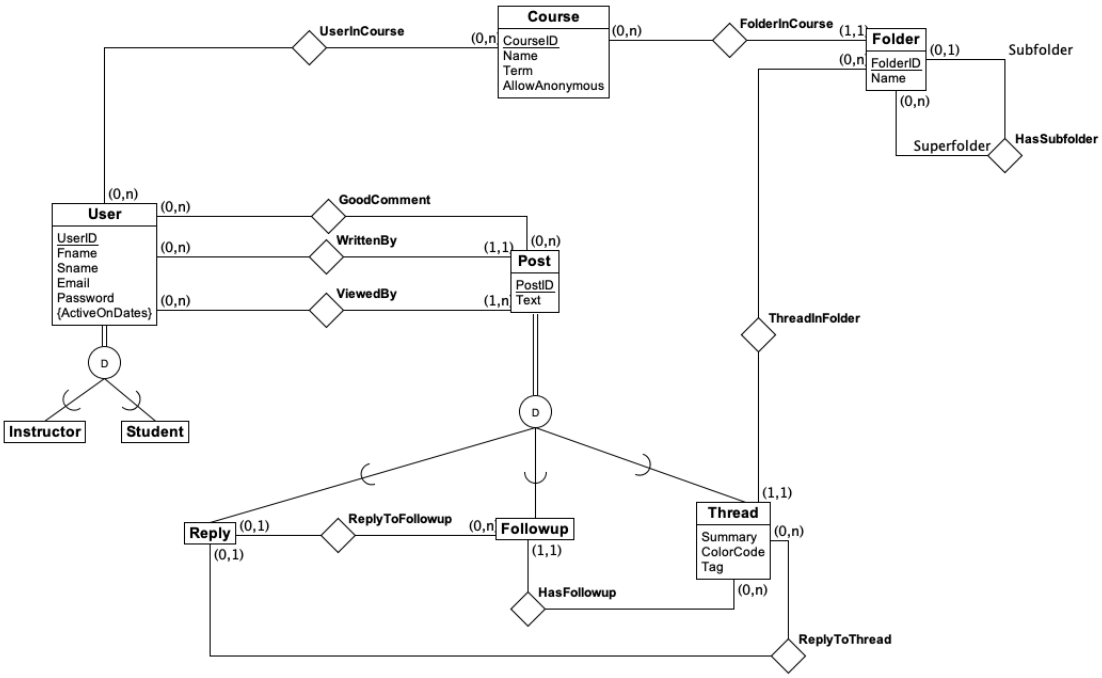

Figure 1: ER-diagram showing the complete data model. For clarity, we have kept the specialization in 'Student' and 'Instructor' in the 'User'-entity class. Martin: Bør det ikke være (1,n) fra Course til User? Et Course må ha en Instructor som starter det og denne blir automatisk første medlem
Da må du opprette en user i databasen før du kan lage et course, men det kan jo funke greit? Er det ikke mest naturlig at brukeren kommer før Course og at det er en User som oppretter et Course?

Assumptions in the ER-model

Assumptions directly related to the ER-model which go beyond the problem text are listed here.

- Since replies, follow-ups and the first post/thread share the same relationships with the users (can be written, liked, viewed) and all contain text, we have modeled these as sub-classes of the entity class 'Post'.
- Each post belongs to only one thread, i.e. the first post of a thread always belongs to the thread (which is modeled as the 'Thread' subclass in the ER-model) and each follow-up always belongs to only one thread. A reply is either directly connected to one thread or one follow-up. Moreover, we assume that threads and follow-ups can have an arbitrary amount of replies.
- The task description states that "Posts may be tagged as questions, announcements, homework, homework solutions, lectures notes and general announcements." We assume that this functionality is only relevant for the first post in each thread. Hence, only the 'Thread' subclass in our ER-diagram has the 'Tag'-attribute. Also, we assume that each thread only has one tag, just as 'Post Type' in Piazza.
- The task description states that "A post does also get a color code depending on if it has a reply or not, and it has a different color code depending on if it is an instructor which has replied, or if it is a student." Similarly to the earlier case with the 'Tag'-attribute, we also assume that this color code is only relevant for the 'Thread' subclass. Also we assume that each post can only be written by one user. This can be deduced from the cardinality from 'Post' to 'User' in 'WrittenBy', which is $(1, 1)$. This is done to avoid ambiguity when deciding the color code of each 'Thread', e.g. in a case where both an instructor and a student are authors to the same post, what color code should be added?
- The task description states that "Each post may have "likes" (named "good comment"). This may be given by each user." Based on functionality in Piazza, we have decided to assume that all types of posts should be able to be liked. This is modelled via the 'GoodComment'-relationship in the ER-model.
- The task description states that "A thread (i.e., the first post) must be categorized to which folder it belongs, e.g., like exam, exercise 1, etc". Based on this, we assume that it is sufficient that each thread is categorized into one and only one folder. This can be deduced from the cardinality from 'Thread' to 'Folder' in 'ThreadInFolder', which is $(1, 1)$.
- The task description states that "An instructor may decide if discussions allow anonymous posts, or if posts should be identified publicly by the user's name." To solve this, we have added a BOOLEAN attribute to each 'Course'-entity called 'AllowAnonymous', which has the value 0 (false) if users are not allowed to be anonymous and a nonzero value (true) if they are allowed to (which logically follows the standard in MySQL 8.0). Hypothetically, the restriction can be enforced for each course in the application program by querying the database, extracting the value of the 'AllowAnonymous' attribute and acting based on the retrieved value.
- When it comes to the courses, the task description states that each course has a sequence of threads. This does not explicitly appear in the ER-model, but it can be found from the model by joining some of the tables 'FolderInCourse', 'Folder', 'ThreadInFolder' and/or 'Thread', depending on exactly what information is in demand. Therefore, we did not find it necessary to have a specific relationship between 'Course' and 'Thread'.

Modelling choices

In the modeling of the problem we have made some choices concerning what to include in the ER-model and what to consider application based functionalities. These choices are documented here.

- The task description states that "It is possible to create links between posts, such that a previous reply may be referenced when the same question or a similar question is posed". We imagine a solution where users write '@ThreadID' in their posts, in order to link it to the post with the specified identifier. Hence, the linking-aspect is not included in the ER-diagram.
- The Instructors' ability to manage folders and invite students is not represented in the ER-model, because we do not see the utility in storing these data in the database. We assume that these features and restrictions are implemented in the application.
- An important functionality which is available for instructors is statistics. The statistics are gathered by querying against the data stored in the database. Since the compiled set of statistics depends on the database state, they should not be stored in the database. Statistics are therefore not represented in the ER-model, but rather implemented in the application program using aggregation operators in MySQL. We also assume that only instructors are allowed to use this functionality.
- The task description states that statistics should be able to show how many users have been active each day. In order to see how many students have been active, i.e. logged into the system, each day, the relation 'ActiveOnDate' stores dates and user-id's each day a user logs in.
- Hva med ViewedBy og GoodComment (og muligens UserInCourse?) som kilder til statistikk? Statistics are also gathered from the relationship classes 'ViewedBy', 'GoodComment', 'WrittenBy' and 'UserInCourse'. The statistics include both instructors and students, though either one can be filtered out in the query.

b)

The ER model translated to relational schemas is shown below. For simplicity, we use the abbreviations FK (foreign key) and PK (primary key) in the description.

Entity classes and attributes from the ER-model:

- User(UserID, Fname, Sname, Email, Password, Type)

  Email attributes must be unique.

  Type states if the user is a student or an instructor.

- ActiveOnDate(UserID, Dated)

  (UserID, Dated) is PK in order to let several users be active each date.

  UserID is FK to User(UserID) and cannot take NULL value.

- Course(CourseID, Name, Term, AllowAnonymous)

- Folder(FolderID, Name, CourseID, SuperFolder, CreatedBy)

  Hvis Superfolder → CourseID gjelder, så bryter tabellen over med 3NF. Martin: Hva da med å splitte det opp slik: Superfolder(SuperfolderID, Name, CourseID, Subfolder, CreatedBy) og Subfolder(SubfolderID, Name, SuperfolderID, CreatedBy)? På den måten slipper måten får man ikke en SubfolderID som peker opp på en CourseID. Ser nå at det forslaget i rødt også fører til en del problemer... Alternativt kan man splitte ut CourseID i en egen FolderInCourse(CourseID, FolderID), som fører til at vi får Folder(FolderID, Name, SuperFolder, CreatedBy) SuperFolder is FK to Folder(FolderID) and can take NULL value.

  Evt. Folder(FolderID, FolderName, CourseID) og SubFolder(SubFolderID, SuperFolderID), så slipper du NULL-verdier

- Post(PostID, Text, Type, UserID)

  UserID is FK to User(UserID) and cannot take NULL value.

The Type-attribute states if the Post is a Reply, Followup or Thread, in order for the post-text to be joined with the rest of the relations to the correct type of post in each respective subclass-table. Each post must have a type, since the specialization is total. Hence, Type cannot take NULL value.

- Thread(ThreadID, Summary, ColorCode, Tag, FolderID)
  ThreadID is FK to Post(PostID) and cannot take NULL value.
  FolderID is FK to Folder(FolderID) and cannot take NULL value.
- Followup(FollowupID, ThreadID)
  FollowupID is FK to Post(PostID) and cannot take NULL value.
  ThreadID is FK to Thread(ThreadID) and cannot take NULL value.
  ReplyID is FK to Post(PostID) and cannot take NULL value.
- ReplyFollowup(ReplyID, FollowupID)
  ReplyID is FK to Post(PostID) and cannot take NULL value.
  FollowupID is FK to Followup(FollowupID) and can take NULL value.
- ReplyThread(ReplyID, ThreadID)
  ReplyID is FK to Post(PostID) and cannot take NULL value.
  ThreadID is FK to Thread(ThreadID) and can take NULL value.

Relationship classes from the ER-model:

- UserInCourse(UserID, CourseID)
  UserID is FK to User(UserID) and cannot take NULL value.
  CourseID is FK to Course(CourseID) and cannot take NULL value.
- GoodComment(UserID, PostID)
  UserID is FK to User(UserID) and cannot take NULL value.
  PostID is FK to Post(PostID) and cannot take NULL value.
- ViewedBy(UserID, PostID)
  UserID is FK to User(UserID) and cannot take NULL value.
  PostID is FK to Post(PostID) and cannot take NULL value.

Normal Form

We start by noting that all the relations are in 1NF since we do not allow any attributes in a tuple to have a set of values. The relations ActiveOnDate, Course, Folder, Post, ReplyFollowup, ReplyThread, UserInCourse, GoodComment and ViewedBy only have the functional dependency PK $\rightarrow$ R, where PK is the primary key and R is the full relation. These relations thus satisfy BCNF (and hence 3NF and 2NF). The User relation has the functional dependencies {UserID $\rightarrow$ R, Email $\rightarrow$ R}. Since both UserID and Email are candidate keys, User is also on BCNF. The relations where nontrivial MVDs can occur have more than two attributes, i.e. possible candidates are Course, User The relational schema is on 4NF because all the relations fulfill the necessary and sufficient criteria, i.e. for each non-trivial multivalued dependency $X \twoheadrightarrow Y$ (in this case, for all functional dependencies), in each relation, X is a superkey.

c)

This section presents a description of how the proposed model satisfies the 5 usecases listed in the problem description. We have discussed some issues not directly related to the usecases, which follow the word '**Additionally**'.

1. When a user logs into the system, the e-mail and password that is entered will be checked with the e-mail and password in the 'User'-table in the database. In this case, a student wants to log in, which means that we need to find the row that contains the given e-mail among the tuples in the 'User'-table with the 'Type' = "student"-attribute value. When the correct tuple matching the e-mail is found, the password belonging to that specific user may be checked. If the passwords match, the student may log in. Here we assume that the user already has an account with the correct e-mail in the database. **Additionally:** In order to keep the statistics over the active users, we need to add the login-date together with the 'UserID' of the student to the 'ActiveOnDate'-table. This way, we register that the student has logged on the specific date, and we can compile the wanted statistics later.

2. When a student makes a post, the input to the system should be the post itself, together with the folder-name and the tag the student wants to attach. In this case, the folder-name "Exam" and the tag "Question" is given, together with the post. The post will then be added to the 'Post'-table, with the post-text received from the student, a type-attribute depending on the type of the post and the UserID of the student. In this case, we assume that the student wants to make a first post in a thread, which means that 'Type' = "thread" should be inserted into the tuple in the 'Post'-table. Furthermore, there should be added a tuple in the 'Thread'-table with 'ThreadID' = 'PostID' (the same ID as the post added to the 'Post'-table), in order to fulfill the connection between the entities in these tables. In this tuple, the summary of the post is added, the color code is set to "red", which means that the post is unanswered, and the "Question"-tag is added to the 'Tag'-column. Moreover, the 'FolderID' of the "Exam"-folder is inserted into the last column of the tuple, which has been retrieved from the 'Folder'-table, by searching through the folder names and returning the 'FolderID' that matches the name "Exam". All in all, the tables needed to retrieve all information to complete this usecase are 'Post', 'Thread' and 'Folder', assuming that the student is logged in, i.e. we already have the 'UserID' of the student. Also, we have assumed that an instructor has already made the "Exam"-folder. Legg inn info i 'ActiveOnDate' og 'ViewedBy' også, slik at statistikken kan føres! Se usecase 3 for hvordan det kan skrives. Ev. kan det skrives her, for deretter å vise til teksten her oppe lenger nede, slik at vi ikke trenger å skrive (nesten) det samme flere steder.

3. When an instructor wants to reply to a post, the input is the unique identifier of the post he wants to reply to. As in usecase 2, we assume that the instructor is already logged in, i.e. we already have the 'UserID' of the instructor. Let us say that an instructor wants to reply to the post made in usecase 2. This means that we can find out what type of post it is in the 'Post'-table, by extracting the 'Type'-attribute from the tuple with the given id. In this case, we know that 'Type' = "thread". Then, the post can be located in the 'Thread'-table, where it is shown that the post belongs to the "Exam"-folder (which we already knew in this case). This shows that it is easy to locate the folder a post belongs to in a case where the folder is unknown to begin with. Since the instructor is writing a reply, a new tuple is inserted into the 'Post'-table, with the reply-text, 'Type' = "reply" and 'UserID' matching the id of the author. Moreover, a tuple is added to the 'ReplyThread'-table, with the same unique id as the new reply-post added to the 'Post'-table. The 'ThreadID' in the ReplyThread tuple is set to match the 'PostID' of the original post that is being replied to. **Additionally:** the 'ColorCode'-attribute of the matching tuple in the 'Thread'-table is set to "green", which signals that the post has been answered by an instructor. Furthermore, as usual, the statistics need to be kept, which means that we need to insert a new row in the 'ActiveOnDate'-table, which contains the 'UserID' and the date the reply was added. Further, we potentially need to insert two rows in 'ViewedBy', one that connects the instructor to the post that is being replied to (if seen for the first time) and another that connects the 'UserID' of the instructor to the newly added reply-post.

4. When a student wants to find posts that contain specific keywords, the 'Post'-table may be searched through. In this specific case, the student wants to locate all posts that contain the keyword "WAL". In our relational model, this constitutes a simple SQL-query in the 'Post'-table, with where clause 'WHERE Text LIKE "%WAL%"'. In addition, for all posts that are threads, we should search for the keyword in the 'Summary'-column as well. This query can be made to return a list of post identifiers that contain the specified keyword. Hence, the only two tables that are needed for this usecase is 'Post' and potentially 'Thread'. **Additionally:** AS usual, the statistics need to be kept. This means that, in the following, for each post from the returned list the student views, this needs to be registered in the 'ViewedBy'-table.

5. The amount of posts each user has read can be found in the 'ViewedBy'-relation, by simply grouping on the 'UserID' and counting the amount of posts each user has viewed. Furthermore, the 'Post'-table has a column containing a 'UserID' for each row, which stores the author of each post. Hence, this relation can be queried in order to count the amount of posts created by each user. This data can be sorted on highest read posting numbers and output to the instructor in the format "user name, number of posts read, number of posts created", e.g. by specifying it in the SQL-queries. Legg inn setning om assumption. **Additionally:** if an instructor specifies it, a ranked/sorted list of most active threads can be calculated based on an aggregation of information in 'GoodComment', 'ViewedBy', 'FollowUp', 'ReplyFollowup', 'ReplyThread', 'Thread' and 'Post'. Hence, more specifically, statistics like how many "likes" all posts in each thread has, how many total views all posts in each thread has and how many followup-posts each thread has, can be compiled and displayed to an instructor.

d)

```
-- The two lines are added below (in the script) in order to create a new schema after dropping.
CREATE SCHEMA `DB1Project`;
USE DB1Project;

CREATE TABLE User(
    UserID VARCHAR(30) NOT NULL,
    Fname VARCHAR(30),
    Sname VARCHAR(30),
    Email VARCHAR(50),
    Password VARCHAR(50),
    Type VARCHAR(10),
    CONSTRAINT User_PK PRIMARY KEY (UserID)
);

CREATE TABLE ActiveOnDate(
    UserID VARCHAR(30) NOT NULL,
    Dated DATE NOT NULL,
    CONSTRAINT ActiveOnDate_PK PRIMARY KEY (UserID, Dated),
    CONSTRAINT ActiveOnDate_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE Course(
    CourseID VARCHAR(30) NOT NULL,
    Name VARCHAR(30),
    Term VARCHAR(6),
    AllowAnonymous BOOL,
    CONSTRAINT Course_PK PRIMARY KEY (CourseID)
);

CREATE TABLE Folder(
    FolderID INT NOT NULL,
```

```
35      Name VARCHAR(30),
36      CourseID VARCHAR(30),
37      SuperFolder INT,
38      CreatedBy VARCHAR(30),
39      CONSTRAINT Folder_PK PRIMARY KEY (FolderID),
40      CONSTRAINT Folder_FK_Course FOREIGN KEY (CourseID)
41          REFERENCES Course(CourseID)
42              ON DELETE SET NULL
43              ON UPDATE CASCADE,
44      CONSTRAINT Folder_FK_Super FOREIGN KEY (SuperFolder)
45          REFERENCES Folder(FolderID)
46              ON DELETE SET NULL
47              ON UPDATE CASCADE,
48      CONSTRAINT Folder_FK_User FOREIGN KEY (CreatedBy)
49          REFERENCES User(UserID)
50              ON DELETE SET NULL
51              ON UPDATE CASCADE
52  );
53
54  CREATE TABLE Post(
55      PostID INT NOT NULL,
56      Text VARCHAR(500),
57      Type VARCHAR(8) NOT NULL,
58      UserID VARCHAR(30) NOT NULL,
59      CONSTRAINT Post_PK PRIMARY KEY (PostID),
60      CONSTRAINT Post_FK_User FOREIGN KEY (UserID)
61          REFERENCES User(UserID)
62              ON DELETE CASCADE
63              ON UPDATE CASCADE
64  );
65
66  CREATE TABLE Thread(
67      ThreadID INT NOT NULL,
68      Summary VARCHAR(100),
69      ColorCode VARCHAR(10),
70      Tag VARCHAR(20),
71      FolderID INT NOT NULL,
72      CONSTRAINT Thread_PK PRIMARY KEY (ThreadID),
73      CONSTRAINT Thread_FK_Post FOREIGN KEY (ThreadID)
74          REFERENCES Post(PostID)
75              ON DELETE CASCADE
76              ON UPDATE CASCADE,
77      CONSTRAINT Thread_FK_Folder FOREIGN KEY (FolderID)
78          REFERENCES Folder(FolderID)
79              ON DELETE CASCADE
80              ON UPDATE CASCADE
81  );
82
83  CREATE TABLE Followup(
84      FollowupID INT NOT NULL,
85      ThreadID INT NOT NULL,
86      CONSTRAINT Followup_PK PRIMARY KEY (FollowupID),
87      CONSTRAINT Followup_FK_Post FOREIGN KEY (FollowupID)
88          REFERENCES Post(PostID)
89              ON DELETE CASCADE
90              ON UPDATE CASCADE,
91      CONSTRAINT Followup_FK_Folder FOREIGN KEY (ThreadID)
92          REFERENCES Thread(ThreadID)
93              ON DELETE CASCADE
94              ON UPDATE CASCADE
95  );
96
97  CREATE TABLE ReplyFollowup(
98      ReplyID INT NOT NULL,
99      FollowupID INT,
100     CONSTRAINT ReplyFollowup_PK PRIMARY KEY (ReplyID),
```

```
    CONSTRAINT ReplyFollowup_FK_Post FOREIGN KEY (ReplyID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT ReplyFollowup_FK_Followup FOREIGN KEY (FollowupID)
        REFERENCES Followup(FollowupID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE ReplyThread(
    ReplyID INT NOT NULL,
    ThreadID INT,
    CONSTRAINT ReplyThread_PK PRIMARY KEY (ReplyID),
    CONSTRAINT ReplyThread_FK_Post FOREIGN KEY (ReplyID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT ReplyThread_FK_Thread FOREIGN KEY (ThreadID)
        REFERENCES Thread(ThreadID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE UserInCourse(
    UserID VARCHAR(30) NOT NULL,
    CourseID VARCHAR(30) NOT NULL,
    CONSTRAINT UserInCourse_PK PRIMARY KEY (UserID, CourseID),
    CONSTRAINT UserInCourse_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT UserInCourse_FK_Course FOREIGN KEY (CourseID)
        REFERENCES Course(CourseID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE GoodComment(
    UserID VARCHAR(30) NOT NULL,
    PostID INT NOT NULL,
    CONSTRAINT GoodComment_PK PRIMARY KEY (UserID, PostID),
    CONSTRAINT GoodComment_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT GoodComment_FK_Post FOREIGN KEY (PostID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE ViewedBy(
    UserID VARCHAR(30) NOT NULL,
    PostID INT NOT NULL,
    CONSTRAINT ViewedBy_PK PRIMARY KEY (UserID, PostID),
    CONSTRAINT ViewedBy_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT ViewedBy_FK_Post FOREIGN KEY (PostID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);
```