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

alexaoh, jimoskar, molderskog

March 12, 2021

a) ER-Model

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

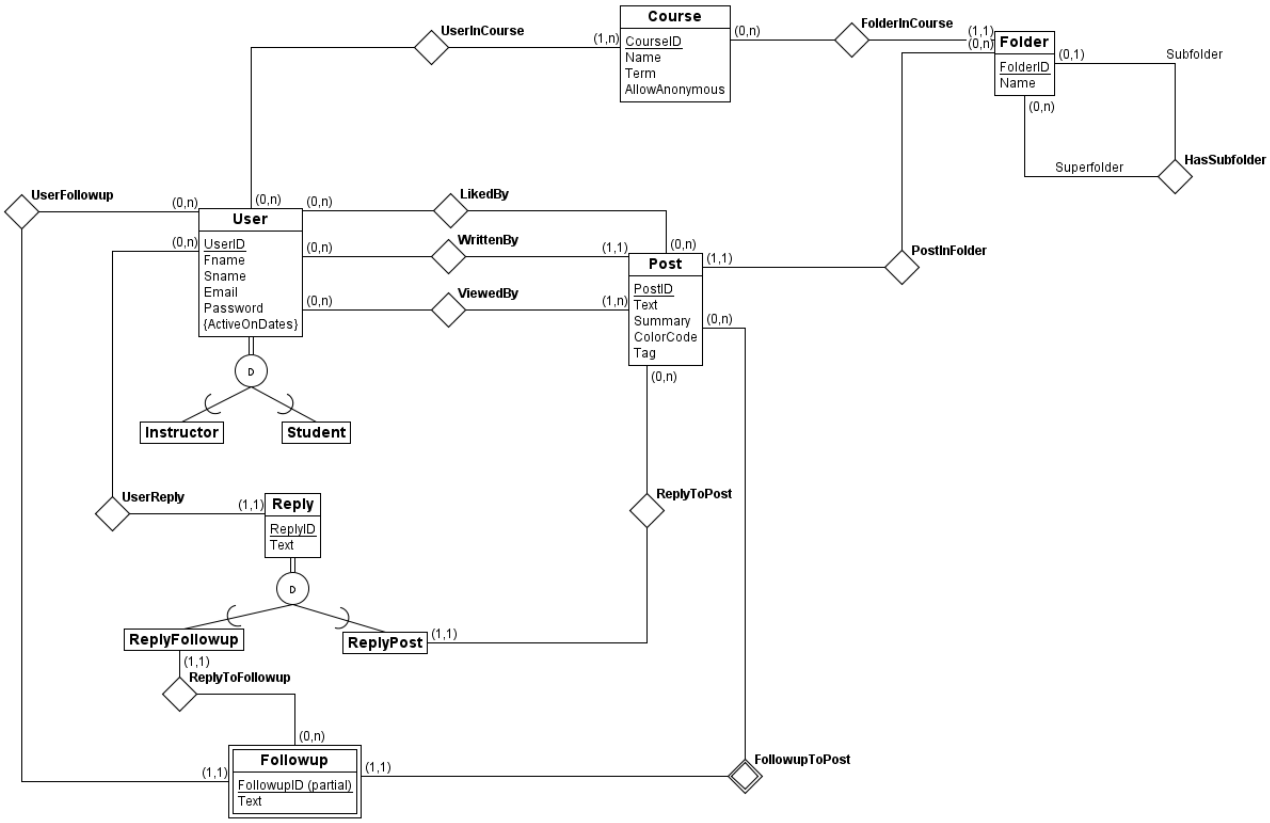

Figure 1: The ER-Model.

Assumptions in the ER-model

- We organize the textual aspect of the forum in Posts, Replies and Followups. A Post must be connected to exactly one Folder and represents a thread in that it can have an arbitrary amount of Followups in addition to an arbitrary amount of Replies. The Followups can also have an arbitrary amount of Replies.
- We have assumed that the ColorCode and Tag attributes are only relevant for the Post entities. Also, we assume that each Post only has one Tag, just as 'Post Type' in Piazza. Furthermore, we assume that only Posts can be Liked or Viewed.
- Posts, Followups and Replies may only have one author. This is done to avoid ambiguity when deciding the color code of each Post, e.g. in a case where both an instructor and a student are authors to the same post, what color code should be added?
- We assume that a Course must have at least one User. This is reasonable, because we imagine that an Instructor-user creates a Course and is automatically enrolled as an Instructor in that course.
- 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, 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.
- The task description states that each Course has a sequence of threads. This does not explicitly appear in the ER-model, but the relationship between the Post and Folder entity classes represents this sequence of threads associated with a Course.

Modelling choices

In the modelling of the problem we have made some choices concerning what to include in the ER-model and what to consider as 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. Even though this makes the specialization of User into Student and Instructor in the ER-diagram functionally unnecessary (in the ER-Diagram), we have chosen to keep it for illustrative purposes.
- 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 changes with 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 and queries in SQL. We also assume that only instructors are allowed to use this functionality, which can be controlled in the application.
- 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.

b) Relational Schema

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.

- User(UserID, Fname, Sname, Email, Password, Type)
  Email attributes must be unique. In other words, Email and UserID constitute the candidate keys.
  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, SuperFolder)
  SuperFolder is FK to Folder(FolderID) and can take NULL value.
- Post(PostID, Text, Summary, ColorCode, Tag, FolderID, UserID)
  FolderID is FK to Folder(FolderID) and cannot take NULL value.
  UserID is FK to User(UserID) and cannot take NULL value.
- Followup(PostID, FollowupID, Text, UserID)
  PostID is FK to Post(PostID) and cannot take NULL value.
  UserID is FK to User(UserID) and cannot take NULL value.
- ReplyPost(ReplyID, Text, UserID, PostID)
  UserID is FK to User(UserID) and cannot take NULL value.
  PostID is FK to Post(PostID) and cannot take NULL value.
- ReplyFollowup(ReplyID, Text, UserID, PostID, FollowupID)
  UserID is FK to User(UserID) and cannot take NULL value.
  (PostID, FollowupID) is FK to Followup(PostID, FollowupID) and cannot 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.
- LikedBy(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.
- FolderInCourse(FolderID, CourseID)
  FolderID is FK to Folder(FolderID) and cannot take NULL value
  CourseID is FK to Course(CourseID) 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, FolderInCourse, Post, ReplyPost, ReplyFollowup, UserInCourse, LikedBy 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 User, Course, Folder, Post, Followup, ReplyPost and ReplyFollowup. These relations are on 4NF because they 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. For further visualization of the functional dependencies, we refer to Figure 2 at the end of the document.

c) Use cases

This section presents a description of how the proposed model satisfies the 5 use cases listed in the problem description. We have discussed some issues not directly related to the use cases, 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. We assume that the student wants to create a Post in the sense that we have interpreted Post, which entails that the student makes a new thread. First the FolderID of the folder named 'Exam' is found by searching through the Folder-table. Here we assume that only one folder is named 'Exam' since a Post can only belong to one Folder. Then the Post tuple is created with Summary and Text written by the student, ColorCode is set to 'red' (signifying that the post is unanswered), Tag is set to 'Question', UserID is set to the ID of the user and FolderID is set to the retrieved FolderID. **Additionally:** We need to insert the id of the Instructor and Post into ViewedBy.

3. To find the posts in the 'Exam'-folder, we can do an inner join with the tables Post and Folder on Post.FolderID = Folder.FolderID. The relevant PostID can then be selected from the resulting tuples. Since the instructor is writing a reply to a Post, a new tuple is inserted into the ReplyPost-table, with the reply-text, UserID matching the id of the author, and the correct PostID. **Additionally:** The ColorCode-attribute of the matching tuple in the Post-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 need to insert a row in ViewedBy that connects the instructor to the post that is being replied to.

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%"'. This query can easily be extended to search the

different Followup- and Reply-tables as well, and then return the associated PostID with a join. Another natural extension would be to also search through the Summary-attribute in the Post-table. **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. To include users who have not written any Posts, we can perform an outer join: 'ViewedBy RIGHT OUTER JOIN User USING(UserID)' before counting. 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. **Additionally:** if an instructor specifies it, a ranked/sorted list of most active threads can be calculated based on an aggregation of information in ActiveOnDate, LikedBy, ViewedBy, FollowUp, ReplyFollowup, ReplyPost and Post. More specifically, statistics like how many "likes", Followups and Replies all Posts have can be compiled and displayed to an Instructor. Like noted earlier, we can also compile statistics for how many Users have been active each day.

d) SQL

The SQL for creating the relational schema in b) is found below, as well as in the txt-file.

```
-- Authors: Alexander Ohrt, Martin Olderskog, Jim Totland - DB1.

-- Three top lines added in order to make our testing a bit more effective.
DROP DATABASE DB1Project;
CREATE SCHEMA DB1Project;
USE DB1Project;

CREATE TABLE User(
    UserID INT,
    Fname VARCHAR(30),
    Sname VARCHAR(30),
    Email VARCHAR(50) UNIQUE,
    UserPassword VARCHAR(50),
    UserType VARCHAR(10),
    CONSTRAINT User_PK PRIMARY KEY (UserID)
);

CREATE TABLE ActiveOnDate(
    UserID INT,
    Dated DATE,
    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 CHAR(7),
    Name VARCHAR(30),
    Term VARCHAR(6),
    AllowAnonymous BOOL,
    CONSTRAINT Course_PK PRIMARY KEY (CourseID)
);

CREATE TABLE Folder(
    FolderID INT AUTO_INCREMENT,
    Name VARCHAR(30),
```

```
    SuperFolder INT,
    CONSTRAINT Folder_PK PRIMARY KEY (FolderID),
    CONSTRAINT Folder_FK_Super FOREIGN KEY (SuperFolder)
        REFERENCES Folder(FolderID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE Post(
    PostID INT AUTO_INCREMENT,
    Text VARCHAR(500),
    Summary VARCHAR(50),
    ColorCode VARCHAR(10),
    Tag VARCHAR(20),
    FolderID INT NOT NULL,
    UserID INT NOT NULL,
    CONSTRAINT Post_PK PRIMARY KEY (PostID),
    CONSTRAINT Post_FK_Folder FOREIGN KEY (FolderID)
        REFERENCES Folder(FolderID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT Post_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE Followup(
    PostID INT,
    FollowupID INT,
    Text VARCHAR(500),
    UserID INT NOT NULL,
    CONSTRAINT Followup_PK PRIMARY KEY (PostID, FollowupID),
    CONSTRAINT Followup_FK_Post FOREIGN KEY (PostID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT Followup_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE ReplyPost(
    ReplyID INT NOT NULL AUTO_INCREMENT,
    Text VARCHAR(500),
    UserID INT NOT NULL,
    PostID INT NOT NULL,
    CONSTRAINT ReplyPost_PK PRIMARY KEY (ReplyID),
    CONSTRAINT ReplyPost_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT ReplyPost_FK_Post FOREIGN KEY (PostID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE ReplyFollowup(
    ReplyID INT AUTO_INCREMENT,
    Text VARCHAR(500),
    UserID INT NOT NULL,
    PostID INT NOT NULL,
    FollowupID INT NOT NULL,
    CONSTRAINT ReplyFollowup_PK PRIMARY KEY (ReplyID),
```

```
    CONSTRAINT ReplyFollowup_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT ReplyFollowup_FK_Followup FOREIGN KEY (PostID, FollowupID)
        REFERENCES Followup(PostID, FollowupID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE UserInCourse(
    UserID INT,
    CourseID CHAR(7),
    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 LikedBy(
    UserID INT,
    PostID INT,
    CONSTRAINT LikedBy_PK PRIMARY KEY (UserID, PostID),
    CONSTRAINT LikedBy_FK_User FOREIGN KEY (UserID)
        REFERENCES User(UserID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT LikedBy_FK_Post FOREIGN KEY (PostID)
        REFERENCES Post(PostID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);

CREATE TABLE ViewedBy(
    UserID INT,
    PostID INT,
    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
);

CREATE TABLE FolderInCourse(
    FolderID INT,
    CourseID CHAR(7),
    CONSTRAINT FolderInCourse_PK PRIMARY KEY (FolderID, CourseID),
    CONSTRAINT FolderInCourse_FK_Folder FOREIGN KEY (FolderID)
        REFERENCES Folder(FolderID)
            ON DELETE CASCADE
            ON UPDATE CASCADE,
    CONSTRAINT FolderInCourse_FK_Course FOREIGN KEY (CourseID)
        REFERENCES Course(CourseID)
            ON DELETE CASCADE
            ON UPDATE CASCADE
);
```

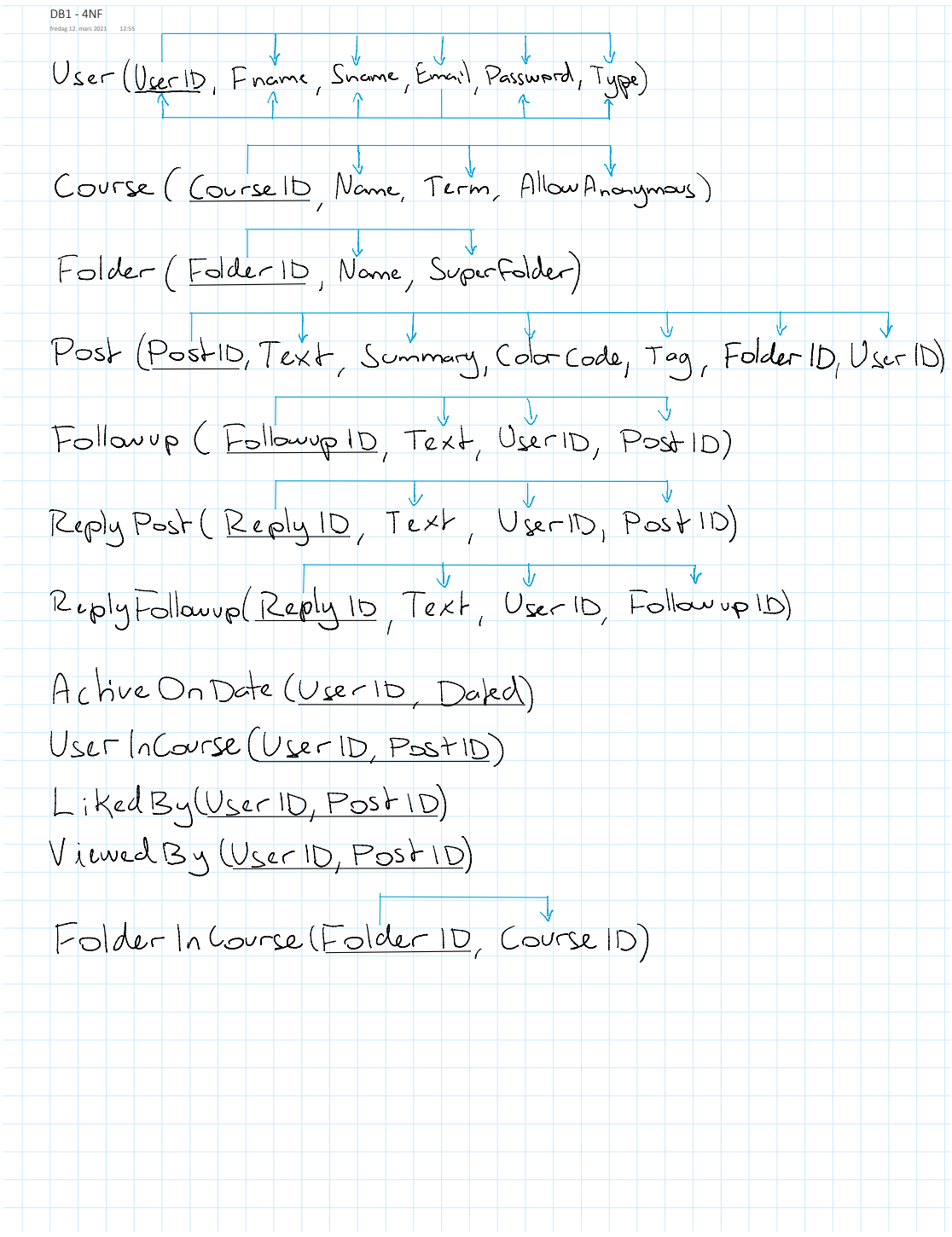

Figure 2: Visualization of functional dependencies