EECS 484 W18 Project #1: Fakebook Database

EECS 484 W18 Staff

Due: January 26th, 2018 at 11:55pm EST

In Project #1, you will be designing a relational database to store information for the fictional social media platform Fakebook. We will provide you with a description of the kinds of data you will need to store, complete with fields and requirements; from that, you will create an ER Diagram and a series of SQL scripts using the concepts and skills from class. This project will give you additional practice with ER Diagrams as well as hands-on experience translating a design specification into SQL.

This project is to be done in teams of 2 students; individual work will not be permitted for this project except in extreme circumstances with the written permission of the professor. Both members of each team will receive the same score; as such, it is not necessary for each team member to submit the assignment. A tool for finding teammates has been made available on Piazza. To create a team, follow these steps:

1. One team member creates the team by clicking the "Create Team' button on the project's assignment page
2. Follow the directions on the "Create Team" page to invite the second team member to join the team
3. The second team member will receive an e-mail with instructions on how to use the eight-letter join code to join the team
4. **Do not make any submissions until the second member joins the team, otherwise they will be prevented from joining!**

Project #1 is due on **Friday, January 26th** at **11:55pm EST**. If you do not turn in your project by that deadline, or if you are unhappy with your work, you may continue to submit up until Tuesday, January 30th at 11:55m EST (4 days after the regular deadline). Submitting late by any length of time will incur a net 15% penalty to your team's project grade. Please refer to the official course policies for more information on late days.

The University of Michigan College of Engineering Honor Code strictly applies to this assignment, and we will be thoroughly checking to ensure that all submissions adhere to the Honor Code guidelines. Students whose submissions are found to be in violation of the Honor Code will be reported directly to the Honor Council. You may not share answers with other students actively enrolled in the course, nor may you consult with students who took the course in previous semesters. You are, however, allowed to discuss general approaches and class concepts with other students, and you are also permitted (and encouraged!) to post questions on Piazza.

Part 1: Creating an ER Diagram

Your first task of Project #1 is to design an ER Diagram that reflects the business rules of the Fakebook platform as described by the company's CEO Clark Huckelburg. Fakebook has four major features: `Users`, `Messages`, `Photos`, and `Events`. Descriptions of these features are listed below, though specifics such as datatype and nullability are explicitly omitted. You may find later sections of this spec and/or the public data set helpful in determining these specifics. Do not make any additional assumptions, even if they would be reasonable in the "real world."

Users

Fakebook's `Users` feature is its most robust feature currently available to the public. When a Fakebook user signs up for the platform, they are assigned a unique ID. A complete Fakebook profile consists of a first name and a last name; a day, month, and year of birth; and a non-binary gender. Additionally, users may (but are not required to) list a hometown and a city of current residence on their profile, and these locations can be updated at any time. A location consists of a city, state, and country. There is no limit to the number of users that Fakebook can support.

In addition to its users' personal information, Fakebook maintains a comprehensive educational history on each user. This educational history consists of a list of "programs" and graduation years from those programs. A program is a trio of fields: the name of the university (i.e. "University of Michigan"), the field of study (i.e. "Computer Science") and the degree earned (i.e. "B.S."); this trio must be unique for every such program. Users can list any number of programs in their educational history, and a single program can be listed in the educational history of any number of users. Fakebook does not prevent users from listing multiple programs with the same graduation year in their educational history.

The last piece of the `Users` feature is friends. Two different Fakebook users can become friends through the platform, but a user cannot be friends with themself. Fakebook tracks the members of a friendship as "Requester" (the user who sent the friend request) and "Requestee" (the user who received the friend request), but no other metadate is stored (such as date of friendship). There is no limit to the number of friends a Fakebook user can have, and no Fakebook user is required to have any friends at all.

Messages

Fakebook allows instant messages to be sent both over its web platform, its mobile app, and in some countries through standard SMS (this feature is awaiting a wider rollout). Each message sent by any of these means is given a unique ID, but the actual method of transmission is not tracked. Fakebook records the content of the message, the time at which the message was sent, the user who sent the message, and the user to whom the message was sent. A Fakebook user can sent 0 or more messages and receive 0 or more messages, but a message can only be sent once; messages that fail to send are not kept in the Fakebook database. Fakebook does not support empty-body messages (i.e. messages with no content) or message attachments, but it does support the most recent version of Unicode and all emoji. Additionally, group messages are not currently supported by Fakebook.

Photos

Like any good social media platform, Fakebook allows its users to post photos (although Fakebook is not yet fully integrated with Instagram). Once uploaded, photos are placed into albums; every photo must belong to exactly one album. Each photo is given a unique ID when it is uploaded, and the metadata for photos consists of the time at which the photo was uploaded, the time at which the photo was last modified, a link to the photo's page on Fakebook, and a caption that accompanies the photo. Fakebook does not directly track the owner/uploader of a photo, but this information can be retrieved by interrogating the album in which the photo is contained.

Each Fakebook album has a unique ID and is owned by exactly one Fakebook user. There is no limit to the number of albums a single user can own, and there is no limit to the number of photos that an album can contain, but each album must contain at least one photo. Fakebook tracks a wealth of metadata for albums: the name of the album, the time at which the album was created, the time at which the album was last modified, a link to the album's page on Fakebook, and a visibility level that defines what group of Fakebook users is allowed to view the photos in the album. In addition, each album must have a cover photo; however, that photo does not actually have to be one of the photos in the album. A single photo can be the cover photo of 0 or more albums.

In addition to creating albums and uploading photos to those albums, Fakebook users can tag one another in uploaded photos. Fakebook tracks the user who is tagged in the photo (but not the user doing the tagging), the time at which the tag was applied, and the x- and y-coordinate of the tagged user in the photo. A user can be tagged in any number of photos but cannot be tagged in the same photo more than once. Any number of users can be tagged in a single photo, including more than one at the same (x, y) coordinate.

Events

The final feature of Fakebook is `Events`. There are two aspects to a Fakebook event: the event itself and the participants to that event. An event itself is uniquely identified by an ID and also has a name, a tagline, and a description. Every event is created by a single Fakebook user (the "creator"); a Fakebook user can create 0 or more events. Other metadata for an event includes the host (not a Fakebook user but a simple string), the address of the event, the event's type and subtype, the location of the event (city, state, and country), and the event's scheduled start and end time.

Fakebook events can have an unlimited number of participants, including 0: this means that the creator of an event does not actually have to participate in the event. Each participant to an event has a confirmation status (i.e. "will attend" or "might be going," although neither of these is actually a valid confirmation status) that is also tracked by Fakebook. Users can participate in any number of events, but no user can participate in the same event more than once, even with the same confirmation status.

Brief Note on Design

Creating ER Diagrams is not an exact science: for a given specification, there are often several valid ways to represent all the necessary information in an ER Diagram. When grading your ER Diagrams, we will look to make sure that all of the entities, attributes, relations, keys, key constraints, and participation constraints are accurately depicted even if your diagram does not exactly match our intended solution. Also note that there may be some constraints described above that are not possible to depict on an ER Diagram. As such, it is perfectly acceptable to ignore these constraints for `Part 1`; you'll implement them later in `Part 2` instead.

Part 2: Creating Data Tables

Your second task of Project #1 is to write SQL DDL statements to create data tables that reflect the Fakebook specifications. You will need to write 2 SQL scripts for this part: `createTables.sql` (to create the data tables) and `dropTables.sql` (to drop/destroy the data tables). These scripts should also create and drop/destroy any constraints, sequences, and/or triggers you find are necessary to enforce the rules of the Fakebook specification.

Once you have written these two files, you should be able to run them using SQL*PLUS from the command line of your CAEN Linux machine:

```
SQL> @createTables.sql;
SQL> @dropTables.sql;
```

You should be able to run the above commands several times sequentially without error. If you cannot do this (i.e. if SQL*PLUS reports errors), you are liable to fail tests on the Autograder.

We will test that your `createTables.sql` script properly creates the necessary data tables with all of the correct constraints. We will attempt to insert both valid and invalid data into your tables with the expectation that the valid inserts will be accepted and the invalid inserts will be rejected. To facilitate this, your tables **must** conform **exactly** to the schema below, even if it doesn't exactly match the schema you would have created following from your ER Diagram. You are not allowed to add any additional tables or columns to the schema, and both the column names and datatypes must match exactly. Deviating from this schema will cause you to fail tests on the Autograder.

- `USERS`
    1. `USER_ID` (`NUMBER`)
    2. `FIRST_NAME` (`VARCHAR2(100)`)
    3. `LAST_NAME` (`VARCHAR2(100)`)
    4. `YEAR_OF_BIRTH` (`INTEGER`)
    5. `MONTH_OF_BIRTH` (`INTEGER`)
    6. `DAY_OF_BIRTH` (`INTEGER`)
    7. `GENDER` (`VARCHAR2(100)`)

- FRIENDS
  1. USER1_ID (NUMBER)
  2. USER2_ID (NUMBER)
     Important Note: This table should not allow duplicate friendships, regardless of the order in which the two IDs are listed. This means that $(1, 9)$ and $(9, 1)$ should be considered *the same entry* in this table, and an insertion of the latter while the former is in the table should be rejected. The means of implementing this constraint is given later in the spec.
- CITIES
  1. CITY_ID (INTEGER)
  2. CITY_NAME (VARCHAR2(100))
  3. STATE_NAME (VARCHAR2(100))
  4. COUNTRY_NAME (VARCHAR2(100))
- USER_CURRENT_CITIES
  1. USER_ID (NUMBER)
  2. CURRENT_CITY_ID (INTEGER)
- USER_HOMETOWN_CITIES
  1. USER_ID (NUMBER)
  2. HOMETOWN_CITY_ID (INTEGER)
- MESSAGES
  1. MESSAGE_ID (NUMBER)
  2. SENDER_ID (NUMBER)
  3. RECEIVER_ID (NUMBER)
  4. MESSAGE_CONTENT (VARCHAR2(2000))
  5. SENT_TIME (TIMESTAMP)
- PROGRAMS
  1. PROGRAM_ID (INTEGER)
  2. INSTITUTION (VARCHAR2(100))
  3. CONCENTRATION (VARCHAR2(100))
  4. DEGREE (VARCHAR2(100))
- EDUCATION
  1. USER_ID (NUMBER)
  2. PROGRAM_ID (INTEGER)
  3. PROGRAM_YEAR (INTEGER)

- USER_EVENTS
  1. EVENT_ID (NUMBER)
  2. EVENT_CREATOR_ID (NUMBER)
  3. EVENT_NAME (VARCHAR2(100))
  4. EVENT_TAGLINE (VARCHAR2(100))
  5. EVENT_DESCRIPTION (VARCHAR2(100))
  6. EVENT_HOST (VARCHAR2(100))
  7. EVENT_TYPE (VARCHAR2(100))
  8. EVENT_SUBTYPE (VARCHAR2(100))
  9. EVENT_ADDRESS (VARCHAR2(2000))
  10. EVENT_CITY_ID (INTEGER)
  11. EVENT_START_TIME (TIMESTAMP)
  12. EVENT_END_TIME (TIMESTAMP)
- PARTICIPANTS
  1. EVENT_ID (NUMBER)
  2. USER_ID (NUMBER)
  3. CONFIRMATION (VARCHAR2(100))
     case-sensitive allowed options are: `attending`, `unsure`, `declined`, or `not_replied`
- ALBUMS
  1. ALBUM_ID (NUMBER)
  2. ALBUM_OWNER_ID (NUMBER)
  3. ALBUM_NAME (VARCHAR2(100))
  4. ALBUM_CREATED_TIME (TIMESTAMP)
  5. ALBUM_MODIFIED_TIME (TIMESTAMP)
  6. ALBUM_LINK (VARCHAR2(100))
  7. ALBUM_VISIBILITY (VARCHAR2(100))
     case-sensitive allowed options are: EVERYONE, FRIENDS, FRIENDS_OF_FRIENDS, or MYSELF
  8. COVER_PHOTO_ID (NUMBER)
- PHOTOS
  1. PHOTO_ID (NUMBER)
  2. ALBUM_ID (NUMBER)
  3. PHOTO_CAPTION (VARCHAR2(2000))
  4. PHOTO_CREATED_TIME (TIMESTAMP)
  5. PHOTO_MODIFIED_TIME (TIMESTAMP)
  6. PHOTO_LINK (VARCHAR2(2000))

- `TAGS`
    1. `TAG_PHOTO_ID` (`NUMBER`)
    2. `TAG_SUBJECT_ID` (`NUMBER`)
    3. `TAG_CREATED_TIME` (`TIMESTAMP`)
    4. `TAG_X` (`NUMBER`)
    5. `TAG_Y` (`NUMBER`)

Feel free to use this schema to better inform the design of your ER Diagram, but do not feel like you must represent this specific schema as long as all of the necessary constraints and other information are shown.

Don't forget to include things like primary keys, foreign keys, `NOT NULL` requirements, and other constraint checking to your DDLs even those those things are not reflected in the schema list above. We recommend using your ER Diagram to assist in this.

Part 3: Populate Your Database

After you create your data tables, you will have to load the data from the public data set into your personal tables. To do this, you will have to write SQL DML statements that `SELECT` the appropriate data from the public data set and `INSERT` that data into your tables. The names of the public tables, their fields, and a few business rules (input constraints) are listed later in the specification, and they might give you some insight into how to design your ER Diagram *and* your own data tables. The public data set is quite poorly designed, so you should not copy the public schema verbatim for your ER Diagram or you will lose a significant number of points.

You should put all of your DML statements into a single file named `loadData.sql` that loads data from the public data set and *not* from a private copy of that data set. You are free to copy the public data set to your own SQL*PLUS account for development and testing, but your scripts will not have access to this account when the Autograder runs them for testing.

When loading data for Fakebook friends, you should only include one directional pair of users even though Fakebook friendship is reciprocal. This means that if the public data set includes both $(2, 7)$ and $(7, 2)$, only one of them (it doesn't matter which one) should be loaded into your table. The friends trigger provided later in the specification will ensure that your data matches what is expected, but only if you properly select only one copy out of the public data set.

Part 4: Creating External Views

The final part of Project #1 is to create a set of external views for displaying the data you have loaded into your data tables. The views you create must have the **exact same** schema as the public data set. This means that the column names and data types must match exactly; this schema is covered later in the spec.

You will need to write 2 SQL scripts for this part: `createViews.sql` (to create the views and load data into them) and `dropViews.sql` (to drop/destroy the views). You should have a total of 5 views named as follows:

- `VIEW_USER_INFORMATION`
- `VIEW_ARE_FRIENDS`
- `VIEW_PHOTO_INFORMATION`
- `VIEW_EVENT_INFORMATION`
- `VIEW_TAG_INFORMATION`

Once you have written these two files, you should be able to run them using SQL*PLUS from the command line of your CAEN Linux machine:

```
SQL> @createTables.sql;
SQL> @loadData.sql;
SQL> @createViews.sql;
SQL> @dropViews.sql;
SQL> @dropTables.sql;
```

You should be able to run the above commands several times sequentially without error. If you cannot do this (i.e. if SQL*PLUS reports errors), you are liable to fail tests on the Autograder.

For each of the views other than `VIEW_ARE_FRIENDS`, your views should exactly match the corresponding table in the public data set. To test this, you can run the following queries in SQL plus, changing the name of the views as necessary. The output of both queries should be `no rows selected`; anything else indicates an error in your views.

```
SQL> SELECT * FROM jsoren.PUBLIC_USER_INFORMATION
  2  MINUS
  3  SELECT * FROM VIEW_USER_INFORMATION;

SQL> SELECT * FROM VIEW_USR_INFORMATION
  2  MINUS
  3  SELECT * FROM jsoren.PUBLIC_USER_INFORMATION;
```

To test `VIEW_ARE_FRIENDS`, use the following test scripts instead. The outputs should again be `no rows selected`.

```
SQL> SELECT LEAST(USER1_ID, USER2_ID), GREATEST(USER1_ID, USER2_ID)
  2  FROM jsoren.PUBLIC_ARE_FRIENDS
  3  MINUS
  4  SELECT LEAST(USER1_ID, USER2_ID), GREATEST(USER1_ID, USER2_ID)
  5  FROM VIEW_ARE_FRIENDS;

SQL> SELECT LEAST(USER1_ID, USER2_ID), GREATEST(USER1_ID, USER2_ID)
  2  FROM VIEW_ARE_FRIENDS
  3  MINUS
  4  SELECT LEAST(USER1_ID, USER2_ID), GREATEST(USER1_ID, USER2_ID)
  5  FROM jsoren.PUBLIC_ARE_FRIENDS;
```

The Public Data Set

The public dataset is divided into five tables, each of which has a series of data fields. Those data fields may or may not have additional business rules (constraints) that define the allowable values.

Here is an overview of the public dataset. All table names and field names are case-insensitive:

- `PUBLIC_USER_INFORMATION`

  1. `USER_ID`

     The unique Fakebook ID of a user
  2. `FIRST_NAME`

     The user's first name; this is a required field
  3. `LAST_NAME`

     The user's last name; this is a required field
  4. `YEAR_OF_BIRTH`

     The year in which the user was born; this is an optional field
  5. `MONTH_OF_BIRTH`

     The month (as an integer) in which the user was born; this is an optional field
  6. `DAY_OF_BIRTH`

     The day on which the user was born; this is an optional field
  7. `GENDER`

     The user's gender; this is an optional field
  8. `CURRENT_CITY`

     The user's current city; this is an optional field, but if it is provided, so too will `CURRENT_STATE` and `CURRENT_COUNTRY`
  9. `CURRENT_STATE`

     The user's current state; this is an optional field, but if it is provided, so too will `CURRENT_CITY` and `CURRENT_COUNTRY`
  10. `CURRENT_COUNTRY`

      The user's current country; this is an optional field, but if it is provided, so too will `CURRENT_CITY` and `CURRENT_STATE`
  11. `HOMETOWN_CITY`

      The user's hometown city; this is an optional field, but if it is provided, so too will `HOMETOWN_STATE` and `HOMETOWN_COUNTRY`
  12. `HOMETOWN_STATE`

      The user's hometown state; this is an optional field, but if it is provided, so too will `HOMETOWN_CITY` and `HOMETOWN_COUNTRY`
  13. `HOMETOWN_COUNTRY`

      The user's hometown country; this is an optional field, but if it is provided, so too will `HOMETOWN_CITY` and `HOMETOWN_STATE`

14. `INSTITUTION_NAME`

    The name of a college, university, or school that the user attended; this is an option field, but if it is provided, so too will `PROGRAM_YEAR`, `PROGRAM_CONCENTRATION`, and `PROGRAM_DEGREE`

15. `PROGRAM_YEAR`

    The year in which the user graduated from some college, university, or school; this is an option field, but if it is provided, so too will `INSTITUTION_NAME`, `PROGRAM_CONCENTRATION`, and `PROGRAM_DEGREE`

16. `PROGRAM_CONCENTRATION`

    The field in which the user studied at some college, university, or school; this is an option field, but if it is provided, so too will `INSTITUTION_NAME`, `PROGRAM_YEAR`, and `PROGRAM_DEGREE`

17. `PROGRAM_DEGREE`

    The degree the user earned from some college, university, or school; this is an option field, but if it is provided, so too will `INSTITUTION_ NAME`, `PROGRAM_YEAR`, and `PROGRAM_CONCENTRATION`

- `PUBLIC_ARE_FRIENDS`

  1. `USER1_ID`

     The ID of the first of two Fakebook users in a friendship

  2. `USER2_ID`

     The ID of the second of two Fakebook users in a friendship

- `PUBLIC_PHOTO_INFORMATION`

  1. `ALBUM_ID`

     The unique Fakebook ID of an album

  2. `OWNER_ID`

     The Fakebook ID of the user who owns the album

  3. `COVER_PHOTO_ID`

     The Fakebook ID of the album's cover photo

  4. `ALBUM_NAME`

     The name of the album; this is a required field

  5. `ALBUM_CREATED_TIME`

     The time at which the album was created; this is a required field

  6. `ALBUM_MODIFIED_TIME`

     The time at which the album was last modified; this is an optional field

  7. `ALBUM_LINK`

     The Fakebook URL of the album; this is a required field

  8. `ALBUM_VISIBILITY`

     The visibility/privacy level for the album; this is a required field

  9. `PHOTO_ID`

     The unique Fakebook ID of a photo in the album

10. `PHOTO_CAPTION`

    The caption associated with the photo; this is an optional field

11. `PHOTO_CREATED_TIME`

    The time at which the photo was created; this is a required field

12. `PHOTO_MODIFIED_TIME`

    The time at which the photo was last modified; this is an optional field

13. `PHOTO_LINK`

    The Fakebook URL of the photo; this is a required field

- `PUBLIC_TAG_INFORMATION`

  1. `PHOTO_ID`

     The ID of a Fakebook photo

  2. `TAG_SUBJECT_ID`

     The ID of the Fakebook user being tagged in the photo

  3. `TAG_CREATED_TIME`

     The time at which the tag was created; this is a required field

  4. `TAG_X_COORDINATE`

     The x-coordinate of the location at which the subject was tagged; this is a required field

  5. `TAG_Y_COORDINATE`

     The y-coordinate of the location at which the subject was tagged; this is a required field

- `PUBLIC_EVENT_INFORMATION`

  1. `EVENT_ID`

     The unique Fakebook ID of an event

  2. `EVENT_CREATOR_ID`

     The Fakebook ID of the user who created the event

  3. `EVENT_NAME`

     The name of the event; this is a required field

  4. `EVENT_TAGLINE`

     The tagline of the event; this is an optional field

  5. `EVENT_DESCRIPTION`

     A description of the event; this is an optional field

  6. `EVENT_HOST`

     The host of the event; this is an optional field, but it does not need to identify a Fakebook user

7. `EVENT_TYPE`

   One of a predefined set of event types; this is an optional field, but the Fakebook front-end takes care of ensuring that the value is actually one of that predefined set by using a dropdown

8. `EVENT_SUBTYPE`

   One of a predefined set of event subtypes based on the event's type; this is an optional field, but cannot be provided unless the `TYPE` field is provided. The Fakebook front-end takes care of ensuring that the value is actually one of that predefined set by using a dropdown

9. `EVENT_ADDRESS`

   The street address at which the event is to be held; this is an optional field

10. `EVENT_CITY`

    The city in which the event is to be held; this is a required field

11. `EVENT_STATE`

    The state in which the event is to be held; this is a required field

12. `EVENT_COUNTRY`

    The country in which the event is to be held; this is a required field

13. `EVENT_START_TIME`

    The time at which the event starts; this is an optional field

14. `EVENT_END_TIME`

    The time at which the event ends; this is an optional field

There is no data for event participants or messages in the public dataset, so you do not need to load anything into your table(s) corresponding to this information.

When referring to any of these tables in your SQL scripts, you will need to use the fully-qualified table name by prepending `jsoren.` (including the `.`) to the table name.

Oracle and SQL*PLUS

To access the public data set for this project and to test your SQL scripts, you will be using a command line interface (CLI) from Oracle called SQL*PLUS. An SQL*PLUS account has been set up for you by the staff, so you should be all set to begin working on the project. If you are still on the waitlist, it is possible that you do not yet have an account; if this is the case, please e-mail `484questions@umich.edu`.

To access your SQL*PLUS account, you must be on a CAEN Linux machine; you can either SSH to one of these machines or access it through a VPN if you cannot get to an actual CAEN computer. Immediately after logging in, before anything else, you will need to load the class module by running `module load eecs484`. We strongly suggest that you add this line to your `~/bashrc` file so that it automatically runs every time you log in to your CAEN account.

To start SQL*PLUS, type `sqlplus` at the command line and press enter; if you wish to have full access to your query history, type `rlwrap sqlplus` (we recommend you use SQL*PLUS in this way). Your username is your University of Michigan uniqname, and your password is `eecsclass` (this is case-sensitive). The first time you log in, the system will prompt you to change your password, which we recommend you do. You may only use alphabetic characters, numerals, the underscore, the dollar sign, and the hash in your SQL*PLUS password. **Under no circumstances should you use quotation marks or the "at" symbol `@` in your SQL*PLUS password.**

Once in SQL*PLUS, you can execute arbitrary SQL commands. You will notice that the formatting of output from SQL*PLUS is less than ideal. Here are some tricks to make output more readable and some SQL commands to access information that might be important. Anything shown below in brackets should be replaced by an actual value:

- To view all of your tables, run the SQL command `SELECT table_name FROM user_tables;`
- To view the full schema of any table, including the tables of the public data set, run the SQL command `DESC [table_name];`
- To truncate the text in a particular column to only show a certain number of characters, run the command `FORMAT [column_name] FORMAT a[num_chars]`
- To remove the formatting from a particular column, run the command `cl [column_name]`, and to remove the formatting from all columns, run the command `CLEAR COLUMNS`
- To change the number of characters displayed on a single line from the default of 100, run the command `SET LINE [num_chars]`
- To change the character that is used to separate the contents of adjacent columns of data, run the command `SET COLSEP '[char]'`
- To select on the first several rows from a table you can use the `ROWNUM` pseudovariable, such as `SELECT * FROM [table_name] WHERE ROWNUM < [num];`
- To change your SQL*PLUS password, run the command `PASSWORD` and follow the prompts
- To quit SQL*PLUS, run `QUIT` or press `ctrl+D`

If you ever forget your password or have other issues accessing your SQL*PLUS account, email `484questions@umich.edu` and we will reset your password to the default as soon as possible. Keep in mind that this may take several hours, during which you will be unable to use SQL*PLUS to work on the project.

Submitting

There will be two deliverables for Project #1: a PDF of your ER Diagram and a zipped tarball containing your SQL scripts. These will be submitted separately, the former to Gradescope for hand-grading and the latter to the Autograder for automated testing.

The PDF of your ER Diagram can be named whatever you would like. Ensure that both team members' uniqnames are clearly visible on the PDF, which can either be fully computer-generated or a scan of something hand-drawn.

Your five SQL scripts (`createTables.sql`, `dropTables.sql`, `loadData.sql`, `createViews.sql`, and `dropViews.sql`) should be zipped into a tarball and submitted through the online Autograder, which should already be open to accept submissions. To create the tarball, put the five SQL files in the same directory, navigate to that directory, and run the following bash command:

```
% tar -zcf project1.tar.gz createTables.sql dropTables.sql
        loadData.sql createViews.sql dropViews.sql
```

Each group will be allowed 4 submissions per day with feedback; any submissions made in excess of those 4 will be graded, but the results of those submissions will be hidden from the group.

Appendix

Sequences

As you're loading data into your tables from the public data set, you might find that you need ID numbers for entities where such ID numbers don't exist in the public data. The way to do this is to use a *Sequence*, which is an SQL construct for generating streams of numbers. To create a sequence and use it to populate IDs for a table, use the following syntax, replacing the bracketed sections with the names/fields specific to your use case:

```
CREATE SEQUENCE [sequence_name]
START WITH 1
INCREMENT BY 1;

CREATE TRIGGER [trigger_name]
  BEFORE INSERT ON [table_name]
    FOR EACH ROW
      BEGIN
        SELECT [sequence_name].NEXTVAL INTO :NEW.[id_field] FROM DUAL;
      END;
/
```

Don't forget the trailing backslash!

Friends Trigger

Triggers are an SQL construct that can be used to execute arbitrary code when certain events happen, such as inserts into a table or updates of the contents of a table. You have already seen on trigger above, which we used to populate the ID field of a table when data is inserted. In this project, you will also have to use a trigger to help enforce the more complicated constraint of the `FRIENDS` table. Because triggers are beyond the scope of this course, we have provided you with the entirety of the trigger syntax here:

```
CREATE TRIGGER order_friends_pairs
  BEFORE INSERT ON FRIENDS
    FOR EACH ROW
          DECLARE temp NUMBER;
          BEGIN
            IF :NEW.USER1_ID > :NEW.USER2_ID THEN
                  temp := :NEW.USER2_ID;
                  :NEW.USER2_ID := :NEW.USER1_ID;
                  :NEW.USER1_ID := temp;
               END IF;
      END;
/
```

This SQL should be included in your `createTables.sql` file, and you should drop it in your `dropTables.sql` file. You do not have to understand what this trigger is doing or what any of the syntax means, but if you are curious, come to Office Hours and the staff will be happy to explain it.

Circular Dependencies for Foreign Keys

Consider the following situation: you have two data tables `TableA` and `TableB`. `TableA` needs to have a foreign key constraint on a column of `TableB`, and `TableB` needs to have a foreign key constraint on a column of `TableA`. How would you implement this in SQL?

The obvious answer is to directly include the foreign key constraints in your `CREATE TABLE` statements, but this unfortunately doesn't work. The reason is that in order to create a foreign key, the table being referenced must already exist; no matter which order we attempt to write out `CREATE TABLE` statements, the first one is going to fail because the other table won't exist yet.

The solution to this conundrum is to add constraints *after* our `CREATE TABLE` statements and to then defer the constraints so that we can actually insert data into those tables. The syntax for adding a constraint after the fact is

```
ALTER TABLE [table_name]
ADD CONSTRAINT [constraint_name]
[constraint_syntax]
INITIALLY DEFERRED DEFERRABLE;
```

where "constraint_syntax" should match what you would have put in the `CREATE TABLE` statement had you done it that way.

Once you've altered both tables in this fashion to add the constraint and defer it, you will have to be careful when you go to insert data into these tables. For reasons you will understand later in the course, you have to fiddle with the SQL*PLUS autocommit. Don't worry too much about why this is, just following the below syntax to insert data into the tables involved in the circular dependency.

```
SET AUTOCOMMIT OFF;
[SQL statements]
COMMIT;
SET AUTOCOMMIT ON;
```

Enumeration-Style Constraints

Let's say that you have a table with a `VARCHAR2` field whose acceptable values are limited to a finite set of options. For example, you may have a `COLOR` field whose valid values are { `red`, `orange`, `yellow`, `blue`, `green` }. This constraint can be directly expressed as part of a DDL statement as follows:

```
CREATE TABLE MyTable(
   EntryID NUMBER PRIMARY KEY,
   Color VARCHAR2(10) NOT NULL,
   CHECK(Color = 'red' OR
         Color = 'orange' OR
         Color = 'yellow' OR
         Color = 'blue' OR
         Color = 'green')
);
```

The `CHECK` will cause any inserts that fail the Boolean to be rejected. This can be extended to any Boolean conditions, including numerical comparisons.

FAQ From Past Semesters

Q: The order of columns in my table and/or view schemas does not match the order of columns in the public dataset's schema. Is this a problem?

A: No, this is not a problem. As long as the table names, column names, and column datatypes match, your schema will be valid.

Q: Are the IDs in the public dataset all unique?

A: Kind of. Each user/event/etc. in the public dataset has a unique ID, but there may be multiple rows in a given table representing data for a single user/event/etc. In those cases, the IDs will be repeated.

Q: Do I need to include checking for the `Type` and `Subtype` fields in the `Events` table?

A: No.

Q: Can we trust all of the data in the public dataset?

A: All of the data in the public dataset conforms to all of the constraints laid out in this document. The only exception is the `PUBLIC_ARE_FRIENDS` table, which may contain impermissible duplicates.

Q: There's a constraint that I can't show on the ER diagram, what should I do?

A: Don't worry – there are some constraints that simply can't be conveyed through ER diagrams. If this is truly the case, you won't be penalized so long as the constraint is enforced in your SQL.

Q: I looked up the schema for one of the tables, and I saw `NUMBER(38)` where the spec says the datatype should be `INTEGER`. Which should I use?

A: Our database uses `INTEGER` as an alias for a specifically-sized `NUMBER` type, which is why you see `NUMBER(38)` in the `DESC` output. Stick to using `INTEGER` in your DDLs.

Q: Is there an automatically-incrementing numeric type that I can use?

A: No, there is not. For those of you familiar with MySQL, there is no equivalent to the `auto_increment` specifier in Oracle. You will have to use sequences to achieve an equivalent effect.

Q: How do I make sure that every `Album` contains at least one `Photo` in my SQL scripts?

A: You can do this with a couple of more complicated triggers, but that is beyond the scope of this course, so you do not need to have this constraint enforced by your SQL scripts. If you choose to accept the challenge of doing so, you will not be penalized so long as the rest of your scripts work appropriately. You do, however, have to show this constraint on your ER diagram.