EECS 484 F17 Project 1: “Fakebook” Database

EECS 484 F17 Staff

Due: Friday, September 29 at 11:55pm EST

Overview

In Project 1, you will be designing a relational database to store information for the fictional social network Fakebook. We will provide you with a description of the different kinds of data, fields, and requirements; from that, you will design an ER diagram using the concepts and skills from class. After designing your diagram, you will write a series of SQL scripts to create tables matching your schema, load data into those tables from a public repository, and generate views for displaying that data.

Due Date

Project 1 is due on **Friday, September 29, 2017** at **11:55pm EST**. If you do not turn in your project by this date, or you are unhappy with your work, you may resubmit until Tuesday, October 3, 2017 at 11:55pm EST (4 days after the due date). Each late day (or part thereof) on which you submit incurs a 1% deduction to your **overall course grade** at the end of the semester. These deductions are cumulative for all course projects, and the first two deductions (overall, not per project) will be reprieved.

Partners

This project can be done alone or in groups of no more than 2. Students who work in groups will receive the same score; as such, only a single submission per group is required (that is, both students in a group do not need to independently submit the project – but you can still submit multiple times). To join a partnership, create and join the same group as your partner on both the Autograder and through the Google Form. A tool for finding partners will be made available on Piazza.

Grading

Your ER diagrams will be hand-graded by course staff for adherence to the specifications, and your SQL scripts will be run through an Autograder. Directions for submitting your project to the Autograder can be found later in this document. We will accept your **last** submission, so do not submit unless you are willing to have the submission graded. This also means that any submissions made after the deadline but before the end of the late days will be counted and graded according to the course late policy. Please refer to the course syllabus and guidelines for the process by which you can ask for a regrade.

Part 1: ER Diagram

Your first task of Project 1 is to design an ER diagram that reflects the data stored by Fakebook with regards to its four major features: `Users`, `Messages`, `Photos`, and `Events`. Descriptions of these features are listed below. Note that these descriptions do not include *how* any of the data is maintained (i.e. datatype or nullity); this information can be found later in this specification and/or by investigating the schemas of the public dataset.

Users

Fakebook's `Users` feature is the most robust feature the social network offers. A Fakebook user is uniquely identified by their ID, and their profile consists of a first and last name; day, month, and year of birth; and non-binary gender. Additionally, users may (but are not required to) provide a single hometown and a single current location when they register with Fakebook, and those can be updated at any time. These locations consist of a city, state, and country. There is no limit on the number of users Fakebook can have.

In addition to its users' personal profile, Fakebook maintains a comprehensive educational history on its users. This history consists of a series of "programs" and graduation years from those program. A program is a trio of fields: the name of the university (i.e. "University of Michigan"), a field of study or concentration (i.e. "EECS"), and a degree earned (i.e. "M.S."); that trio must be unique for each program. Users can list any number of programs in their educational history, including multiple from the same university, and any program can be listed by an unlimited number of users. At this time, Fakebook does not prevent users from listing multiple different programs in their educational history with the same graduation year.

The last piece of the `Users` feature is friends. Two different users can be friends, but a user cannot be friends with themself. There is no limit to the number of friends a single user has, and users are not required to have any friends at all. Also, if *UserA* is friends with *UserB*, this necessarily means that *UserB* is friends with *UserA*; Fakebook friendship is reciprocal.

Messages

Fakebook's `Messaging` feature is tightly connected to the `Users` feature, but still independent enough to be considered its own feature. Each message sent over Fakebook is given a uniquely-identifying ID. Both the content of the message and the time at which the message was sent are recorded by Fakebook, as are both the user who sent the message and the user who received the message (both of which must be actual Fakebook users). Any user can send zero or more messages and can receive zero or more messages, but each message can only be sent once (and must actually be sent to be logged by Fakebook's system; Fakebook does not track messages that failed to send). At this time, Fakebook does not allow for group messages.

Photos

Like any good social media platform, Fakebook allows its users to upload photos and to store those photos in albums. Each photo must belong to exactly one album. Each photo is given a unique ID and the following metadata: the time at which the photo was uploaded, the time at which the photo was last modified, a link to the photo's individual Fakebook page, and a caption that accompanies the photo. Fakebook does not directly track the owner of a photo, but that information can be gleaned from the album to which the photo belongs.

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 a single album can contain, but it must contain at least one. The metadata associated with Fakebook albums are: 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 Fakebook page, and a visibility level for the album that defines who can see the photos it contains. Additionally, each album has a cover photo; however, that photo does not actually have to belong to the album itself (even though that would conceptually be a valid constraint) A single photo can be the cover photo of any number of albums, including 0.

In addition to creating albums and uploading photos, users can tag other users in photos. For tags, Fakebook tracks the user being tagged (but not the user doing the tagging), the photo in which that subject is tagged, the time at which the tag was applied, and the x- and y-coordinates of the tag within the photo. A user can be tagged in any number of photos, including 0, but cannot be tagged in the same photo more than once. However, multiple different users can be tagged in the same photo, including at the same coordinate location (even though this is theoretically illogical).

Events

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

Fakebook events can have an unlimited number of users, but they don't need to have any at all; it is perfectly valid for an event to have no participants. (Note that this means that the creator of an event *does not* need to be a participant in that event.) Each participant to an event has a confirmation statues (i.e. "will attend" or "might be going") that is also tracked by Fakebook. Users can participate in any number of events, but no user can be a participant in the same event twice.

Note on Design

Designing an ER diagram is not an exact science, so there are many different possible "correct" designs. When grading your diagram, we will look to see that all of the relationships, keys, key constraints, and participation constraints are accurately reflected even if your exact design does not match our ideal model.

Part 2: Create Tables

Your second task of Project 1 is to use SQL DDL statements to create data tables that reflect your ER diagram. You should write two SQL script files:

1. createTables.sql to *create* the data tables
2. dropTables.sql to *drop/destroy* the data tables

The DDL statements should also put in place any constraints, triggers, and sequences that you find are necessary to enforce the relationships described in the previous section. You should be sure to drop any constraints, triggers, and sequences as well.

When you have written your DDL statements, you should be able to run them using SQL*Plus with the following commands:

```
SQL< @dropTables.sql
SQL< @createTable.sql
```

You should be able to create and drop your content several times consecutively without error. If you cannot do this, you are liable to fail our Autograding scripts.

We will test your tables to ensure that the constraints expressed above are enforced. We will attempt to insert valid and invalid values into your tables, and we expect that the invalid inserts will be rejected. To facilitate this, your tables **must** conform to the following schema, even if it doesn't exactly match your ER diagram. You are allowed to add additional columns to your table, but if you do so, they cannot be required columns: we must be able to perform insertions into your tables knowing only that they have the following table names, column names, and column datatypes (all names are case-insensitive):

- `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 which order the two IDs are given in. That 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. This specification is covered more in the next section.

- CITIES
  1. `CITY_ID` (`INTEGER`)
  2. `CITY_NAME` (`VARCHAR2(100)`)
  3. `STATE_NAME` (`VARCHAR2(100)`)
  4. `COUNTRY_NAME` (`VARCHAR2(100)`)
- USER_CURRENT_CITY
  1. `USER_ID` (`NUMBER`)
  2. `CURRENT_CITY_ID` (`INTEGER`)
- USER_HOMETOWN_CITY
  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)`)
     allowed options are: `attending`, `unsure`, `declined`, and `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(2000)`)
  7. `ALBUM_VISIBILITY` (`VARCHAR2(100)`)
     allowed options are: `everyone`, `friends`, `friends_of_friends`, `myself`, and `custom`
  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 so long as all of the necessary constraints and other information are shown.

Don't forget to include things like primary keys, foreign keys, non-`NULL` requirements, and other constraint checking to your DDLs even though those things are not reflected in the schema listed above. Use your ER diagram to assist you in this.

Part 3: Populate Your Database

After you create your data tables, you will have to load the data from the public dataset into your schemas. In order to do this, you will have to write SQL DML statements that `select` the appropriate data from the public dataset and `insert` that date into your tables. The names of the tables, their fields, and a few business rules (input constraints) are listed in a section later in this specification, and they might give you some insight into how to design your ER diagram *and* your own database. Note: the public dataset is quite poorly designed, so you should not copy the schema verbatim or you will likely lose a significant number of points.

You should put all of your SQL DML statements into a single file, **loadData.sql**, that loads from **our public dataset** and *not* from a private copy of that dataset. You are free to copy the public dataset to a personal copy for ease of testing, but you will not have access to these copies when we run your scripts through the Autograder.

When loading data for Fakebook friends, you should only include one pair of users even though friendship is reciprocal. So if the public dataset has both $(2, 7)$ and $(7, 2)$, only one of them (either one) should be loaded into your data tables; loading both or neither will be considered incorrect. The best way to achieve this is to ensure that only one copy is loaded from the public dataset and that future insertions (or batch insertions) are rejected if they contain both such pairs.

The public dataset has not been cleaned, so it is very likely that there are duplicate rows of data. When you load the data from the public dataset into your data tables, you should make sure that you do not load duplicate rows. If you load duplicate rows, you will lose points.

Part 4: Create Views

The last part of Project 1 is to create a set of views for displaying the data you have loaded into your database. The views you create **must have the exact same schema as the public dataset**, even though the content of the views may be slightly different. This means that both the **table names** and the **column names** and **column datatypes** must exactly match those of the public dataset. The schema of the public dataset is covered in the next section.

You should create two SQL script files that contain the necessary statements for creating and dropping all five (5) of your views. These files should be:

1. createViews.sql to *create* the views

2. dropViews.sql to *drop/destroy* the views

Once you have created these files, you should be able to run the following sequence of commands several times in a row without encountering any errors in SQL*PLUS:

1. SQL< @createTable.sql
2. SQL< @loadData.sql
3. SQL< @createViews.sql
4. SQL< @dropView.sql
5. SQL< @dropTables.sql

Your views should exactly match the public dataset *except* for the `PUBLIC_ARE_FRIENDS` table, which should eliminate reciprocal duplicates where appropriate. To test your views against the public data set, you can run the following SQL two commands in SQL*Plus:

```
SQL< SELECT * FROM jsoren.PUBLIC_USER_INFORMATION
     MINUS
     SELECT * FROM VIEW_USER_INFORMATION

SQL< SELECT * FROM VIEW_USER_INFORMATION
     MINUS
     SELECT * FROM jsoren.PUBLIC_USER_INFORMATION
```

Both commands should return an empty result set and can be repeated for all of your views you wish to test.

The Public Dataset

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 option field

  6. `EVENT_HOST`

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

  7. `EVENT_TYPE`

     One of a predefined set of event types; this is a required 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 a required field, but 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 a required field

  14. `EVENT_END_TIME`

      The time at which the event ends; this is a required 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.

To view the full schema of any of these tables (such as to access the datatypes), you can use the `DESC` command in SQL*Plus like this:

SQL> DESC [userid].PUBLIC_ARE_FRIENDS

This will display the schema of the PUBLIC_ARE_FRIENDS table belonging to the user with the user ID `userid` (you should omit the brackets in real life). If you own the table whose schema you wish to view, you can omit the user ID altogether.

The public dataset is currently hosted by a member of the staff under the user ID `jsoren`. To access the five public dataset tables, use the following fully-qualified table names:

`jsoren.PUBLIC_USER_INFORMATION`

`jsoren.PUBLIC_ARE_FRIENDS`

`jsoren.PUBLIC_PHOTO_INFORMATION`

`jsoren.PUBLIC_TAG_INFORMATION`

`jsoren.PUBLIC_EVENT_INFORMATION`

Oracle and SQL*Plus

To access the public dataset for this project and to test your SQL scripts, you will be using a command line user interface from Oracle called SQL*Plus. An SQL*Plus has been created for you, so you should be all set to begin working on the project. If you are on the waitlist, it is possible that an account has not yet been created; if this is the case, or if you are unable to access your account for any reason, please contact eecs484f17@umich.edu.

To log into your SQL*Plus account, first log into a CAEN machine either through SSH, a VPN, or by sitting down at an engineering-enabled computer. Before doing anything else, run the following command:

```
module load eecs484
```

We suggest that you add the above line to the end of your `~/.bashrc` bash file, which will cause it to execute every time you log in. If you don't do this, you will have to run this command to execute the module manually each time you log into your CAEN account to do work for EECS484.

To start SQL*Plus, simply type in `sqlplus` into the command line. Your username will be your standard UMich log-in username, and the password will initially be **eecsclass** (this is case-sensitive). When you log in for the first time, the system will prompt you to change your password, which we recommend you do. The following characters are valid in SQL*Plus passwords:

- Alphabetic characters
- Numerals

- Underescore (_)
- Dollar Sign ($)
- Hash (#)

DO NOT USE QUOTATION MARKS OR THE "AT" SYMBOL (@) IN YOUR SQL*PLUS PASSWORD! If you do, you will likely be unable to access the system and will have to email eecs484f17@umich.edu for assistance.

Once you are in SQL*Plus, you can execute any arbitrary SQL commands. When you first use SQL*Plus, though, you might notice that the formatting is less than ideal. Here are some tricks to make the console output more readable and some general commands to access potentially-important information:

- To turn on your command history, which can be accessed with the "up" arrow and makes it so that you don't have to retype the same command over and over, execute the command `% rlwrap sqlplus` outisde of SQL*Plus.
- To view all of the existing tables, use `SELECT TABLE_NAME FROM USER_TABLES`
- Some columns might have really long output that jumbles the output. To truncate the text in a particular column and show only the first handful of characters, run `COLUMN` *`column_name`* `FORMAT` *`num_chars`*
- To clear an formatting that you've assigned to a column, run the command `cl` *`column_name`*
- SQL*Plus assumes that all lines are 80 characters wide, which is a problem for some fields that are defined to be 100-character-long strings. To change the number of characters displayed in a line/row, use `SET LINE` *`num_chars`*
- To execute SQL commands that are contained in an external file, run `START` *`filename`*. You might want to run `SET ECHO ON` before running the `START` command; this will print each command to the console as it is read in from the file, which will help with error identification.
- To view only the first several rows of a table, you can use the `ROWNUM` pseudovariable as such: `SELECT * FROM` *`table_name`* `WHERE ROWNUM <` *`num_rows`*
- To change the character that separates the columns of a table output, execute `SET COLSEP` *`'char'`*
- To quit SQL*Plus, simply type `QUIT`

Submitting

You should upload your ER diagram as a PDF file (name it whatever you'd like) to Gradescope. This file can be created on your computer as a PDF or hand-written and scanned. In either case, please make sure that your's and your partner's uniqnames are clearly visible on the document.

The remaining files should be placed into a ZIP file named **project1.zip** and uploaded to the Project 1 Autograder. We will make an announcement when the Autograder is open for submissions. This ZIP file should contain the following files and no others:

☐ createTables.sql

☐ dropTables.sql

☐ loadData.sql

☐ createViews.sql

☐ dropViews.sql

☐ An optional textfile named *readme.txt* that can contain any notes you'd like to submit alongside your project files

To create the ZIP file, transfer all of your files to a CAEN Linux machine and execute the following bash command:

```
% zip -r project1.zip createTables.sql dropTables.sql loadData.sql
             createViews.sql dropViews.sql readme.txt
```

To make sure that the file has been created appropriately, email it to yourself, open it from another machine, and make sure you can extract all of the files.

As a reminder, only one member of your partnership needs to submit the ZIP file. Be sure that you have reported your partnerships to the Google Form and that you take the proper steps to submit in groups on both the Autograder and through Gradescope. More information about how to do this will be made available when the Autograder and Gradescope are opened to accept submissions.

You will be allowed 3 submissions per day *per group* with feedback and an unlimited number of submissions after that for which no feedback will be provided. We will record your **last** submission for grading.

Appendix

Sequences

As you're loading data into your tables, you might find yourself in a situation where you need to assign ID numbers to rows of data. This might be because your personal table schema has an ID field that isn't present in the public dataset. This can be done by using a *Sequence*, which is an SQL construct for producing sets of numbers. To create a sequence, use the following syntax, replacing the italicized sections with the appropriate values for your use case:

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

To then use your sequence to tie unique ID numbers to rows being inserted into some table, replicate the following code exactly, replacing the italicized portions appropriately:

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

Do not forget the final slash; if you do, this code will not work correctly.

Triggers

Triggers are snippets of SQL code that execute before or after certain actions take place on a database. They can be helpful for enforcing constraints that can't be expressed in DDL statements. The structure for code that is to run before data is inserted or updated in a particular table is

```
CREATE TRIGGER trigger_name
    BEFORE INSERT OR UPDATE ON table_name
        FOR EACH ROW
            DECLARE var_name var_type
            BEGIN
                your code here
            END;
/
```

That code can be modified to run *after* insertion or update by changing "before" to "after" in the second line. The line declaring variables can be omitted or repeated as necessary.

Triggers, for the most part, are beyond the scope of this course. If you find yourself needing a trigger (which you probably will), we highly encourage you to post on Piazza or attend Office Hours; the course staff will be more than happy to walk you through writing a trigger provided you understand what you want the trigger to do and why.

Circular Dependencies for Foreign Keys

Consider the following situation: You have two tables, `TableA` and `TableB`. `TableA` needs to have a foreign key on `TableB` and `TableB` needs to have a foreign key on `TableA`. How would you implement these constraints?

The obvious answer is to simply put the foreign key constraints directly into your `CREATE TABLE` statements. This seems like it would work, but you'll run into issues when you try to create the tables because SQL will complain that `TableB` doesn't exist when you try to create `TableA`. The answer to this issue of circular dependencies is to defer the constraints.

To create a deferred constraint, do not place the constraint statements in the CREATE TABLE code. Instead, after all of the necessary tables have been created, add code analogous to the following:

```
ALTER TableA
ADD CONSTRAINT constraint
INITIALLY DEFERRED DEFERRABLE;
```

The *constraint* should match the code you would have put in the CREATE TABLE statement. If you then want to add data into either table that forms part of the circularly-dependent constraint, you must explicitly turn on/off the autocommit:

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

Enumeration-Style Constraints

Let's say that you have a table with a string field whose acceptable values are limited to a specific set of strings. For example, you might have a field COLOR whose acceptable values are in the set {red, orange, yellow, blue, green}. This constraint can be directly expressed in a DDL statement as follows:

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

The CHECK statement will cause any entries that don't pass the Boolean statement to be rejected.

FAQ From W17 (Last Semester)

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.