EECS 484 Project #1: Fakebook Database

Due Thursday, Feb 3rd at **11:55** PM Eastern time

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.

We recommend completing this project in teams of 2 students; individual work is permitted, but not recommended. 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 on the autograder, follow these steps **before making your first submission**:

1. One team member creates the team by clicking the “Send group invitation” button on the project’s assignment page.
2. The second team member will receive an email with instructions on how to join the team.
3. **Do not make any submissions before joining your team! Once you click on “I’m working alone", the autograder will not let you change team members.** If you do need to make a correction, the teaching staff has the ability to modify teams.

Project #1 is due on Thursday, Feb 3rd at 11:55 PM 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 Monday, Feb 7th at 11:55 PM EST (4 days after the regular deadline). Please refer to the official course policies for more information on *penalties for late submissions*.

The University of Michigan College of Engineering Honor Code strictly applies to this assignment, and we will be 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 (except with your partner if you decide to have one), 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 data types and nullability are explicitly omitted. You may find later sections of this spec and/or the public data set helpful in determining these specifics -- practicing such design decisions is valuable for your growth as an engineer. 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 user ID. A Fakebook profile consists of first name, last name, day, month, year of birth, and a non-binary gender. Additionally, users may (but are not required to) list a hometown city and/or a current city on their profile, and these cities can be switched at any time, though they can only have 1 hometown and 1 current city at any given time. Each city has a unique city ID, a city name, a state name, and a country name. The combination of city name, state name and country name is unique (you may not need to reflect this property in your ER Diagram).

In addition to its users' personal information, Fakebook maintains educational history on each user, which consists of "programs" and graduation year. Besides a unique program ID, each program also has a trio of fields: the institution (e.g. "University of Michigan"), the concentration (e.g. "Computer Science") and the degree (e.g. "B.S."); this trio must be unique for every such program. Users may list any number of programs (including 0) in their educational history; and a program may or may not be listed in the educational history of any number of users. Fakebook does not prevent users from listing different programs with the same graduation year; however, a user cannot list the same program multiple times with different graduation years.

The last piece of the Users feature is friends. Two different Fakebook users can become friends through the platform, but a user cannot befriend him/herself (you may not need to reflect this property in your ER Diagram). Fakebook tracks the members of a friendship as "Requester" and "Requestee" (refer to the concept of different roles in the same entity in lecture 2). There is no limit to the number of friends a Fakebook user can have. Also, a Fakebook user can have zero friends.

Messages

Fakebook allows messages to be sent between two users. Each message is given a unique message ID. Fakebook records the message content and the message sent time. It also tracks the user who sends the message as "Sender" and the user who receives the message as

“Receiver”. A Fakebook user can send or receive 0 or more messages, but a message can only be sent exactly once. Group messages are not currently supported by Fakebook.

Photos & Albums

Like any good social media platform, Fakebook allows its users to post photos. Once uploaded, photos are placed into albums and each photo must belong to exactly one album. Each photo is given a unique photo ID, and the metadata for photos consists of the photo uploaded time, last modified time of the photo, the photo’s link, and a caption. Fakebook does not directly track the owner/uploader of a photo, but this information can be retrieved via the album in which the photo is contained.

Each Fakebook album has a unique album 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. However, each album must contain at least one photo. Fakebook tracks metadata for albums: the album name, the time the album was created, the last modified time of the album, the album's link, and a visibility level (e.g. ‘Myself’, ‘Everyone’). In addition, each album must have a cover photo; however, that photo does not have to be one of the photos in the album. An album can have only one cover photo. 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 be tagged in the photos. Fakebook tracks the tagged user (but not the user doing the tagging), the tagged time, and the x- and y-coordinate within the photo. A user can be tagged in any number of photos, but cannot be tagged in the same photo more than once. A single photo can contain tags of 0 or more users. The tagged time and x-,y- coordinates may be the same or different (e.g., two tags on one photo could share the same x,y coordinates).

Events

The final feature of Fakebook is Events. An event itself is uniquely identified by an event ID and also contains a name, a tagline, and a description. Each 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 street address, the event’s type and subtype, the start and end time. Each event must be located in exactly one city; each city may have 0 or more events being held in.

The creator of an event does not have to participate in the event, which means that Fakebook events can have an unlimited number (including 0) of users participating. Each participant in an event has a confirmation status (e.g. ‘Attending’, ‘Declines’). Users can participate in any number of events, but no user can participate in the same event more than once, even with a different confirmation status.

Note on ER Diagram 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 within SQL*PLUS on your CAEN Linux machine. For accessing your Oracle account please refer to the section **Oracle and SQL*Plus** in a later part of this spec.

```
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 data types **must** match exactly. Deviating from this schema will cause you to fail tests on the Autograder.

`USERS`

- `USER_ID (NUMBER) [Required field]`

- FIRST_NAME (VARCHAR2(100)) [Required field]
- LAST_NAME (VARCHAR2(100)) [Required field]
- YEAR_OF_BIRTH (INTEGER)
- MONTH_OF_BIRTH (INTEGER)
- DAY_OF_BIRTH (INTEGER)
- GENDER (VARCHAR2(100))

FRIENDS
- USER1_ID (NUMBER) [Required field]
- USER2_ID (NUMBER) [Required field]

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 -- attempting to insert one while the other is already in the table should result in the insertion being rejected. The means of implementing this constraint is given later in the spec (Look in the appendix for the “Friends Trigger” section).

CITIES
- CITY_ID (INTEGER) [Required field]
- CITY_NAME (VARCHAR2(100)) [Required field]
- STATE_NAME (VARCHAR2(100)) [Required field]
- COUNTRY_NAME (VARCHAR2(100)) [Required field]

USER_CURRENT_CITIES
- USER_ID (NUMBER) [Required field]
- CURRENT_CITY_ID (INTEGER) [Required field]

USER_HOMETOWN_CITIES
- USER_ID (NUMBER) [Required field]
- HOMETOWN_CITY_ID (INTEGER) [Required field]

MESSAGES
- MESSAGE_ID (NUMBER) [Required field]
- SENDER_ID (NUMBER) [Required field]
- RECEIVER_ID (NUMBER) [Required field]
- MESSAGE_CONTENT (VARCHAR2(2000)) [Required field]
- SENT_TIME (TIMESTAMP) [Required field]

PROGRAMS
- PROGRAM_ID (INTEGER) [Required field]
- INSTITUTION (VARCHAR2(100)) [Required field]
- CONCENTRATION (VARCHAR2(100)) [Required field]
- DEGREE (VARCHAR2(100)) [Required field]

EDUCATION

- USER_ID (NUMBER) [Required field]
- PROGRAM_ID (INTEGER) [Required field]
- PROGRAM_YEAR (INTEGER) [Required field]

USER_EVENTS

- EVENT_ID (NUMBER) [Required field]
- EVENT_CREATOR_ID (NUMBER) [Required field]
- EVENT_NAME (VARCHAR2(100)) [Required field]
- EVENT_TAGLINE (VARCHAR2(100))
- EVENT_DESCRIPTION (VARCHAR2(100))
- EVENT_HOST (VARCHAR2(100))
- EVENT_TYPE (VARCHAR2(100))
- EVENT_SUBTYPE (VARCHAR2(100))
- EVENT_ADDRESS (VARCHAR2(2000))
- EVENT_CITY_ID (INTEGER) [Required field]
- EVENT_START_TIME (TIMESTAMP)
- EVENT_END_TIME (TIMESTAMP)

PARTICIPANTS

- EVENT_ID (NUMBER) [Required field]
- USER_ID (NUMBER) [Required field]
- CONFIRMATION (VARCHAR2(100)) [Required field]

Confirmation must be one of these options (case-sensitive): ATTENDING, UNSURE, DECLINES, or NOT_REPLIED

ALBUMS

- ALBUM_ID (NUMBER) [Required field]
- ALBUM_OWNER_ID (NUMBER) [Required field]
- ALBUM_NAME (VARCHAR2(100)) [Required field]
- ALBUM_CREATED_TIME (TIMESTAMP) [Required field]
- ALBUM_MODIFIED_TIME (TIMESTAMP)
- ALBUM_LINK (VARCHAR2(100)) [Required field]
- ALBUM_VISIBILITY (VARCHAR2(100)) [Required field]

Album_visibility must be one of these options (case-sensitive): EVERYONE, FRIENDS, FRIENDS_OF_FRIENDS, or MYSELF

- COVER_PHOTO_ID (NUMBER) [Required field]

PHOTOS

- PHOTO_ID (NUMBER) [Required field]
- ALBUM_ID (NUMBER) [Required field]
- PHOTO_CAPTION (VARCHAR2(2000))
- PHOTO_CREATED_TIME (TIMESTAMP) [Required field]

- PHOTO_MODIFIED_TIME (TIMESTAMP)
- PHOTO_LINK (VARCHAR2(2000)) [Required field]

TAGS
- TAG_PHOTO_ID (NUMBER) [Required field]
- TAG_SUBJECT_ID (NUMBER) [Required field]
- TAG_CREATED_TIME (TIMESTAMP) [Required field]
- TAG_X (NUMBER) [Required field]
- TAG_Y (NUMBER) [Required field]

Feel free to use this schema to better inform the design of your ER Diagram, but do not feel like your diagram 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 though those things are not reflected in the schema list above. We recommend using your ER Diagram to assist in this.

Important Note: Using very long constraint names can cause some of your Autograder test cases to fail. Keep your constraint names short or don't use the CONSTRAINT keyword at all unless necessary. For example, inside of your CREATE TABLE statements, instead of writing
`CONSTRAINT a_very_long_constraint_name CHECK (Column_A = 'A')`
you can write
`CHECK (Column_A = 'A')`

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 exactly 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

Any use of the keyword "`project1`" in code or comment in your `createViews.sql` will cause your submission to automatically fail on the Autograder. This is to prevent any potential cheating. Please be cautious of this as you develop your solutions.

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 SQLplus, 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 project1.PUBLIC_USER_INFORMATION
  2 MINUS
  3 SELECT * FROM VIEW_USER_INFORMATION;

SQL> SELECT * FROM VIEW_USER_INFORMATION
  2 MINUS
  3 SELECT * FROM project1.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 project1.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 project1.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. When referring to any of these tables in your SQL scripts, you will need to use the fully-qualified table name by prepending `project1.` (including the ".") to the table name (as seen in the testing examples above).

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 optional 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 optional 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 optional 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 optional 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 menu.

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 menu.
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. However, you should assume that MESSAGE_ID would have been provided by the public dataset and does not need to be created using sequences and triggers.

Again, when referring to any of these tables in your SQL scripts, you will need to use the fully-qualified table name by prepending `project1.` (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 do not have an account please email eecs484staff@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. In order to use SQL*PLUS, you will need to load the class module by running `module load eecs484` in the command line. We suggest that you add this line to your ~/.bashrc or ~/.bash_profile 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` instead (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. Only use alphabetic characters, numerals, the underscore, the dollar sign, and the hash in your SQL*PLUS password. **Never use quotation marks or the "at" symbol (@) in your SQL*PLUS password.** If you do, it is likely that you will not be able to log into your account, and you will need to contact course staff to reset it.

Once in SQL*PLUS, you can execute arbitrary SQL commands. You will notice that the formatting of output from SQL*PLUS can be 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:
  ```
  COLUMN [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];
  ```
  Another command that helps with formatting is:
  ```
  set markup csv on;
  ```

- 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 load commands in SQL*PLUS from a file, say createTables.sql:
  ```
  START createTables.sql
  ```
  (The name of the file is relative to the current directory from which sqlplus was launched).

- To allow command-history to be used using up-arrow/down-arrow keys within a sqlplus session, launch the sqlplus shell command as usual, but prefaced with the Linux rlwrap utility, e.g.:
  ```
  rlwrap sqlplus
  ```

- To quit SQL*PLUS, run:
  ```
  QUIT;
  ```
  or press ctrl+D

We have made an Autograder project named 'SQL*Plus Reset' that would allow you to reset your sqlplus password or kill your sqlplus sessions.

To reset your password to `eecsclass`, submit any text file named `password.txt` to this project.

To kill all your sqlplus sessions and fix the “ORA-00054: resource busy and acquire with NOWAIT specified or timeout expired.” error, submit any text file named `sessions.txt` to this project.

If you still have issues accessing your SQL*PLUS account after trying the solutions above, please email us and we can take a look. 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 your 5 SQL scripts. These will be submitted separately, the former to Gradescope for hand-grading and the latter to the Autograder for automated testing. The self-enrollment code for Gradescope is **X3PJJY** if you do not have access. Part 1 is worth 62 points and the rest are worth 50 points each, for a total of 212 points (150 on the Autograder, 62 from ER Diagram).

The PDF of your ER Diagram can be named whatever you would like. Your diagram can either be fully computer-generated or a scan of something hand-drawn. One team member should submit on Gradescope, but ***make sure to submit as a team, specifying your partner on Gradescope at submission time*. If you do not do this, we will not be able to assign points to your partner.**

Your five SQL scripts (`createTables.sql`, `dropTables.sql`, `loadData.sql`, `createViews.sql`, and `dropViews.sql`) should be submitted through the online Autograder. An email notification will be sent once it is open to accept submissions.

Each group will be allowed 3 submissions per day with feedback; any submissions made in excess of those 3 will be graded, but the results of those submissions will be hidden from the group. Your highest scoring submission will be used for grading, with ties favoring your latest submission.

Appendix

Working with CAEN and SQLPlus

To connect to CAEN remotely and work using SQLPlus, you will need to use SSH. Linux and Mac Users have an advantage here since they will have SSH built into their terminals. Windows users will need to download a tool such as PuTTY to SSH in. This will allow you to connect to CAEN and establish a terminal session where you will be able to run SQLPlus and other CAEN tools. Additionally, to transfer files to CAEN you will need to use FTP. Again Linux and Mac users will have FTP built into their terminals, but Windows users will need a utility such as WinSCP or Cyberduck to perform this transfer. Please see Discussion 1 for more details on this.

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 one 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;
/
```

Don't forget the trailing backslash!

This SQL should be included in your `createTables.sql` file, and you should drop it in your `dropTables.sql` file. All this trigger is doing is making sure that any incoming pair of friend IDs is sorted, which preserves uniqueness. If you're having any difficulty understanding what this is doing, 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?

One tempting solution is to directly include the foreign key constraints in your `CREATE TABLE` statements, but this unfortunately does not work. 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 will not yet exist.

Instead, we can add foreign key constraints to a table *after* it and the table it references have been created using an `ALTER TABLE` statement. The syntax for adding a foreign-key constraint to a previously created table is:

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

where “constraint syntax” should be the foreign key syntax that you would have put in a `CREATE TABLE` statement. (Note: the above `ALTER TABLE` syntax works for other kinds of SQL constraints as well.)

For simplicity and safety, you can write an `ALTER TABLE` statement using the above syntax for *both* tables in a circular dependency. (Note: you don’t *have* to implement constraints this way to get full credit; other syntax can work and can be more concise.)

Adding `INITIALLY DEFERRED DEFERRABLE` makes it so that when you run your `loadData.sql` script to populate your tables, Oracle will defer the constraint check until later. Why would we need to defer the check? Imagine your **Table A** and **Table B** have a circular dependency and are currently empty, but we are about to insert data into both tables. As soon as we insert data into **Table A** that is supposed to reference rows in **Table B**, Oracle will claim that **Table A**’s foreign key constraint has been violated, since all references to **Table B** in **Table A** don’t currently point to anything valid (remember, **Table B** is currently empty!)

By *deferring* the check on **Table A**’s foreign key references to **Table B**, we can give Oracle the chance to insert data into both **Table A** and **Table B** before it performs the foreign key check. You can ensure that this happens by grouping the `INSERT` statements for **Table A** and **Table B** into a single transaction. A transaction is a single unit of work that the database performs. When you defer a constraint check, Oracle will wait until the end of a transaction to check for constraint violations. We will learn more about transactions at the end of the semester. By default, Oracle treats each individual SQL statement as its own transaction. This feature is called autocommiting, which is not desirable if you would like to insert into **Table A** and **Table B** in the same transaction. Instead, you should manually define a transaction by turning off autocommit with the statement `SET AUTOCOMMIT OFF` in your script. After this statement, you can include your `INSERT` statements for **Table A** and **Table B**. After your `INSERT` statements, you should include the statement `COMMIT`. You will want to turn autocommit back on once this is done by including the statement `SET AUTOCOMMIT ON`.

Debugging and Dependencies between Tables

Be mindful of the dependencies between tables when debugging your code. For example, in Part 2, if your Users table fails to be created, other subsequent tables that depend on the Users table will also fail to be created. As another example, if you are failing Test_User_Current_Cities in Part 3 on the Autograder, check that you have first gotten Test_Cities correct.

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 data types 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: Nope.

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: 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, Oracle has no equivalent to the auto increment specifier. You will have to use sequences to achieve an equivalent effect (see Appendix).

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. You do, however, have to show this constraint on your ER diagram.

Q: There are many ways to declare a foreign key, how should I do it/is there an advised way?

A: For this project, it is advised that you use
"FOREIGN KEY (Column_ID) REFERENCES Table_name (Other_ID)"
Students have encountered autograder errors when attempting other declarations. If you have encountered errors while using a different declaration, it may be helpful to try this!

Things to Be Careful when using SQL*Plus

1. Blank lines inside of a command will cancel the command

This works

```
SELECT *
FROM table_name;
```

But this will break your code

```
SELECT *

FROM table_name;
```

2. Properly formatting comments

SQL*Plus (at least the version we use for this project) is quite strict about how comments should be formatted.

- Begin multiline comment symbol /* must have a space right afterwards

This works

```
SELECT * FROM table_name;
/* this is a multiline
comment block*/
```

But this will break your code

```
SELECT * FROM table_name;
/*this is a multiline
comment block*/
```

- Comments should not come right after the semicolon

  These work

```
SELECT * FROM table_name;
-- this is a single line comment
```

```
SELECT * FROM table_name;
/* this is a single line comment*/
```

  But these will break your code

```
SELECT * FROM table_name; -- this is a single line comment
```

```
SELECT * FROM table_name; /* this is a single line comment*/
```