EECS 484 Projects | p1-fakebook-db

Project 1: Fakebook Database

Worth	Released	Due
212 points (62 on Gradescope, 150 on Autograder)	Aug 29th	**Sep 24th at 11:45PM EST**

Project 1 is due on **Sep 24th at 11:45 PM EST**. Please refer to the EECS 484 F24 Course Policies for more information on penalties for late submissions, and late day tokens.

Introduction

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 data we will need to store, complete with fields and requirements. Armed with the design specification, you will create an ER Diagram as well as a series of SQL scripts to create, load, and drop objects.

Submissions

This project is to be done in teams of 2 students. Both members of each team will receive the same score - the highest score among all submissions; as such, it is not necessary for each team member to submit the assignment.

Before making your first submission, follow these steps to create a team on the Autograder:

- One team member clicks the "Send group invitation" button on the Project 1 page.
- The other team member confirms the invitation on their Autograder assignment page.

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.

Honor Code

By submitting this project, you are agreeing to abide by the Honor Code: "I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code." You may not share answers with other students actively enrolled in the course outside of your teammate, 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.

If you are retaking the class, you must work alone for the projects (and homeworks) you worked on in previous semesters. You may reuse your code. For projects that you have not worked on in previous semesters, you are free to find a teammate and work together.

Part 1: Creating an ER Diagram

Your first task 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 Albums, 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 dataset 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. Their profile also consists of a 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 changed 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 allows different users to register for the same program with the same or different graduation years; 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" (recall the concept of different roles in the same entity from

lecture). 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 (including themselves). 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 (i.e. it has exactly one sender and exactly one receiver). Group messages are currently not supported by Fakebook.

Photos and 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/poster 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 exactly one cover photo; however, that photo does not 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 be tagged in the photos. Fakebook tracks the tagged user (but not the user doing the tagging), the tagged time, and the x-coordinate 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 the Data Tables

Your second task of Project 1 is to write SQL DDL statements to create/drop 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 any constraints, sequences, and triggers** you find are necessary to enforce the rules of the Fakebook specification.

Once you have written these two files, you should run them within SQL*Plus on your CAEN Linux machine. You should be able to run the commands below 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. To access CAEN and your Oracle account, follow the setup instructions at Tools.

```
SQL> @createTables
SQL> @dropTables
```

Desired Schema

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 from your ER Diagram. Column

names, types, ordering, and constraints must be the same. Deviating from this schema will cause you to fail tests on the Autograder.

You may find some of the commands listed in Helpful SQL*Plus Commands useful when viewing your tables.

Users

Column	Type	Required
user_id	INTEGER	yes
first_name	VARCHAR2(100)	yes
last_name	VARCHAR2(100)	yes
year_of_birth	INTEGER	
month_of_birth	INTEGER	
day_of_birth	INTEGER	
gender	VARCHAR2(100)	

Friends

Column	Type	Required
user1_id	INTEGER	yes
user2_id	INTEGER	yes

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. To implement this, see Friends Trigger.

Cities

Column	Type	Required
city_id	INTEGER	yes
city_name	VARCHAR2(100)	yes
state_name	VARCHAR2(100)	yes
country_name	VARCHAR2(100)	yes

User_Current_Cities

Column	Type	Required
user_id	INTEGER	yes
current_city_id	INTEGER	yes

User_Hometown_Cities

Column	Type	Required
user_id	INTEGER	yes
hometown_city_id	INTEGER	yes

Messages

Column	Type	Required
message_id	INTEGER	yes
sender_id	INTEGER	yes
receiver_id	INTEGER	yes
message_content	VARCHAR2(2000)	yes
sent_time	TIMESTAMP	yes

Programs

Column	Type	Required
program_id	INTEGER	yes
institution	VARCHAR2(100)	yes
concentration	VARCHAR2(100)	yes
degree	VARCHAR2(100)	yes

Education

Column	Type	Required
user_id	INTEGER	yes
program_id	INTEGER	yes
program_year	INTEGER	yes

User_Events

Column	Type	Required
event_id	INTEGER	yes
event_creator_id	INTEGER	yes
event_name	VARCHAR2(100)	yes
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	yes
event_start_time	TIMESTAMP	
event_end_time	TIMESTAMP	

Participants

Column	Type	Required
event_id	INTEGER	yes
user_id	INTEGER	yes
confirmation	VARCHAR2(100)	yes

The value of `confirmation` must be one of these options (case-sensitive): `Attending`, `Unsure`, `Declines`, or `Not_Replied`.

Albums

Column	Type	Required
album_id	INTEGER	yes
album_owner_id	INTEGER	yes
album_name	VARCHAR2(100)	yes
album_created_time	TIMESTAMP	yes

Column	Type	Required
album_modified_time	TIMESTAMP	
album_link	VARCHAR2(2000)	yes
album_visibility	VARCHAR2(100)	yes
cover_photo_id	INTEGER	yes

The value of `album_visibility` must be one of these options (case-sensitive): `Everyone`, `Friends`, `Friends_Of_Friends`, or `Myself`.

Photos

Column	Type	Required
photo_id	INTEGER	yes
album_id	INTEGER	yes
photo_caption	VARCHAR2(2000)	
photo_created_time	TIMESTAMP	yes
photo_modified_time	TIMESTAMP	
photo_link	VARCHAR2(2000)	yes

Tags

Column	Type	Required
tag_photo_id	INTEGER	yes
tag_subject_id	INTEGER	yes
tag_created_time	TIMESTAMP	yes
tag_x	NUMBER	yes
tag_y	NUMBER	yes

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 primary keys (each table should have one), foreign keys, `NOT NULL` requirements, and other constraints in your DDLs even though they 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')
```

Additionally, since we will be loading data into your tables to test them, **you may find it helpful to read through Part 3**. You may also find it helpful to read the section on Circular Dependencies.

Sequences

In Part 3, as you're loading data, you will `SELECT` data from columns (e.g. an ID) in the public dataset and `INSERT` it into your tables. However, you will 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 sequences, which are SQL constructs for generating streams of numbers.

When you're loading data, you do not need to reference these sequences. However, since we are executing DDL statements, these sequences need to be created beforehand in your `createTables.sql` script (one for each ID that does not exist in the public dataset). However, note that there is no Messages data in the public dataset to load, so you do not need a sequence for `MESSAGE_ID`.

To create a sequence and its corresponding trigger, 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. This trigger makes sure that any incoming pair of friend IDs is sorted, which preserves uniqueness. Like the above sequences, the DDL statement to create this trigger should be in your `createTables.sql` script. Because triggers are beyond the scope of this course, we have provided you with the entirety of the trigger syntax here:

```
CREATE TRIGGER Order_Friend_Pairs
    BEFORE INSERT ON Friends
    FOR EACH ROW
        DECLARE temp INTEGER;
        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!

Part 3: Populating Your Database

The third part of Project 1 is to load data from the public dataset (a poorly designed database) into the tables you just created (a well designed database). To do this, you will have to write SQL DML statements that `SELECT` the appropriate data from the public dataset and `INSERT` that data into your tables. You may also need additional SQL statements like `DISTINCT`, `JOIN`, and `WHERE`, as well as set operators like `INTERSECT`, `UNION`, and `MINUS`.

You should put all of your DML statements into a single file named `loadData.sql` that loads data from the public dataset (and not from a private copy of that dataset). You are free to copy the public dataset 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 dataset 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 will ensure that the direction of friendship matches what is expected, but you are still expected to properly select exactly one copy out of the public dataset.

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. 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 Part 4: Creating External Views).

Additionally, beware of common SQL*Plus Potholes when creating your queries.

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

Public_User_Information

Column	Required	Description
user_id	yes	The unique Fakebook ID of a user
first_name	yes	The user's first name
last_name	yes	The user's last name
year_of_birth	yes	The year in which the user was born
month_of_birth	yes	The month (as an integer) in which the user was born
day_of_birth	yes	The day on which the user was born
gender	yes	The user's gender
current_city	yes	The user's current city
current_state	yes	The user's current state
current_country	yes	The user's current country
hometown_city	yes	The user's hometown city
hometown_state	yes	The user's hometown state
hometown_country	yes	The user's hometown country
institution_name		The name of a college, university, or school that the user attended
program_year		The year in which the user graduated from some college, university, or school
program_concentration		The field in which the user studied at some college, university, or school

Column	Required	Description
program_degree		The degree the user earned from some college, university, or school

If one of `institution_name`, `program_year`, `program_concentration`, or `program_degree` are provided, then all four columns will be provided. If none are provided, then all four columns will be empty.

Public_Are_Friends

Column	Required	Description
user1_id	yes	The ID of the first of two Fakebook users in a friendship
user2_id	yes	The ID of the second of two Fakebook users in a friendship

Public_Photo_Information

Column	Required	Description
album_id	yes	The unique Fakebook ID of an album
owner_id	yes	The Fakebook ID of the user who owns the album
cover_photo_id	yes	The Fakebook ID of the album's cover photo
album_name	yes	The name of the album
album_created_time	yes	The time at which the album was created
album_modified_time	yes	The time at which the album was last modified
album_link	yes	The Fakebook URL of the album
album_visibility	yes	The visibility/privacy level for the album
photo_id	yes	The unique Fakebook ID of a photo in the album
photo_caption		The caption associated with the photo
photo_created_time	yes	The time at which the photo was created
photo_modified_time	yes	The time at which the photo was last modified
photo_link	yes	The Fakebook URL of the photo

Public_Tag_Information

Column	Required	Description
photo_id	yes	The ID of a Fakebook photo
tag_subject_id	yes	The ID of the Fakebook user being tagged in the photo
tag_created_time	yes	The time at which the tag was created
tag_x_coordinate	yes	The x-coordinate of the location at which the subject was tagged
tag_y_coordinate	yes	The y-coordinate of the location at which the subject was tagged

Public_Event_Information

Column	Required	Description
event_id	yes	The unique Fakebook ID of an event
event_creator_id	yes	The Fakebook ID of the user who created the event
event_name	yes	The name of the event
event_tagline		The tagline of the event
event_description		A description of the event
event_host	yes	The host of the event, which does not need to identify a Fakebook user
event_type	yes	One of a predefined set of event types
event_subtype	yes	One of a predefined set of event subtypes based on the event's type
event_address	yes	The street address at which the event is to be held
event_city	yes	The city in which the event is to be held
event_state	yes	The state in which the event is to be held
event_country	yes	The country in which the event is to be held
event_start_time	yes	The time at which the event starts
event_end_time	yes	The time at which the event ends

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. 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.

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 dataset. This means that the column names and data types must match exactly. 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 the commands below 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.

```
SQL> @createTables
SQL> @loadData
SQL> @createViews
SQL> @dropViews
SQL> @dropTables
```

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

```
SELECT * FROM project1.Public_User_Information
MINUS SELECT * FROM View_User_Information;
```

```
SELECT * FROM View_User_Information
MINUS 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`.

```
SELECT LEAST(user1_id, user2_id),  GREATEST(user1_id, user2_id)
FROM project1.Public_Are_Friends
MINUS SELECT LEAST(user1_id, user2_id),  GREATEST(user1_id, user2_id)
FROM View_Are_Friends;
```

```
SELECT LEAST(user1_id, user2_id),  GREATEST(user1_id, user2_id)
FROM View_Are_Friends
MINUS SELECT LEAST(user1_id, user2_id),  GREATEST(user1_id, user2_id)
FROM project1.Public_Are_Friends;
```

Submitting

There are two deliverables for Project 1:

- a PDF of your ER Diagram, and
- your 5 SQL scripts (`createTables.sql` , `dropTables.sql` , `loadData.sql` , `createViews.sql` , and `dropViews.sql`)

Part 1 (ER Diagram) is worth 62 points and Parts 2-4 (SQL scripts) are worth 50 points each, for a total of 212 points (62 on Gradescope, 150 on the Autograder). There are no private tests.

Gradescope

Your ER Diagram will be submitted to Gradescope for hand-grading.

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. You may submit any number of times before the deadline. We will grade your latest submission. **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.**

Autograder

Your SQL scripts will be submitted to the Autograder for automated testing. Reach out via Piazza if you do not have access to the Autograder. It is your responsibility to ensure you have access well before the deadline.

Each team 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 team. Your highest scoring submission will be used for grading, with ties favoring your latest submission.

Appendix

Circular Dependencies

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 <constaint_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. You don't have to implement constraints this way to get full credit; other syntax can work and can be more concise. **Remember that like sequences and triggers, `ALTER TABLE` statements help define your database schema, and thus, are SQL DDL commands.** Think about which file they should be included in.

Adding `INITIALLY DEFERRED DEFERRABLE` tells Oracle to defer the constraint check until later when you run your `loadData.sql` script to populate your tables. 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 an atomic 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 autocommitting, 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: Yes, the column order in your table and view schemas must match the order specified in this spec.

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 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 may see `NUMBER` or `NUMBER(38)` in the `DESC` output. Stick to using `INTEGER` in your

DDLs, with the exception of `TAG_X` and `TAG_Y`, which should be `NUMBER`.

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.

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!

Q: Why do we sometimes use the AS keyword for aliases, but sometimes we don't?

A: In Oracle, column aliases can be defined as `column_name AS column_alias` or `column_name column_alias`. Table aliases can only be defined as `table_name table_alias`.

Acknowledgements

This project was written and revised over the years by EECS 484 staff at the University of Michigan. The most recent version was updated and moved to Primer Spec by Owen Pang.