1 Preliminaries

In this lab, you will work with a Adventure database schema similar to the schema that you used in Lab2. We've provided a create_lab3.sql script for you to use (which is similar to, but not quite the same as the create.sql in our Lab2 solution), so that everyone can start from the same place. Please remember to DROP and CREATE the Lab3 schema before running that script (as you did in previous labs), and also execute:

ALTER ROLE yourlogin SET SEARCH_PATH TO Lab3;

so that you'll always be using the Lab3 schema without having to mention it whenever you refer to a table.

You will need to log out and log back in to the server for this default schema change to take effect. (Students often forget to do this.)

We've now also provided a load_lab3.sql script that will load data into your tables. You'll need to run that script before executing Lab3. The command to execute a script is: \i <filename>

In Lab3, you will be required to combine new data (as explained below) into one of the tables. You will need to add some new constraints to the database and do some unit testing to see that the constraints are followed. You will also create and query a view, and create an index.

New goals for Lab3:

1. Perform SQL to "combine data" from two tables
2. Add foreign key constraints
3. Add general constraints
4. Write unit tests for constraints
5. Create and query a view
6. Create an index

There are lots of parts in this assignment, but none of them should be difficult. Lab3 will be discussed during the Lab Sections before the due date, Tuesday, November 15. The due date of Lab3 is 3 weeks (plus a couple of days) after the due date of Lab2 because students want to study for the Midterm, and also because we didn't cover all of the Lab3 topics before the Midterm.

2. Description

2.1 Tables with Primary Keys for Lab3

The primary key for each table is underlined. The attributes are the same ones from Lab2, but there's one table that you haven't seen before.

Members(memberID, name, address, joinDate, expirationDate, isCurrent)

Rooms(roomID, roomDescription, northNext, eastNext, southNext, westNext)

Roles(role, battlePoints, initialMoney)

Characters(memberID, role, name, roomID, currentMoney, wasDefeated)

Things(thingID, thingKind, initialRoomID, ownerMemberID, ownerRole, cost, extraBattlePoints)

Monsters(monsterID, monsterKind, name, battlePoints, roomID, wasDefeated)

Battles(characterMemberID, characterRole, characterBattlePoints, monsterID, monsterBattlePoints)

ModifyMembers(memberID, name, address, expirationDate)

In the create_lab3.sql file that we've provided under Resources→Lab3, the first 7 tables are similar to the tables were in our Lab2 solution, except that the NULL and UNIQUE constraints from Lab2 are **not** included. Also, create_lab3.sql is missing one of the Foreign Key Constraints on Things (the constraint that says that a thing's owner is a character) and is missing both of the Foreign Key Constraints on Battles that were in our Lab2 solution. (You'll create new variations of those constraints in Lab3.)

In practice, primary keys, unique constraints and other constraints are almost always entered when tables are created, not added later. create_lab3.sql handles some constraints for you, but, you will be adding some additional constraints to these tables in Lab3, as described below.

Note also that there is an additional table, ModifyMembers, in the create_lab3.sql file that has most (but not all) of the attributes that are in the Members table. We'll say more about ModifyMembers below.

Under Resources→Lab3, you've also been given a load script named lab3_load_data that loads tuples into the tables of the schema. **You must run both create_lab3.sql and lab3_load_data.sql before you run the parts of Lab3 that are described below.**

2.2 Combine Data

Write a file, *combine.sql* (which should have multiple SQL statements that are in a Serializable transaction) that will do the following. For each tuple in ModifyMembers, there might already be a tuple in the Members table that has the same primary key (that is, the same value for memberID). If there **isn't** a tuple in Members with the same primary key, then this is a new member who should be inserted into Members. If there already **is** a tuple in Members with that primary key, then this is an update of information about that member. So here are the effects that your transaction should have:

- If there isn't already a tuple in the Members table which has that memberID, then you should insert a tuple into the Members table corresponding to all the attribute values in that ModifyMembers tuple. Your insert should also set joinDate to today's date (we explain how below), and isCurrent to NULL.

- If there already is a tuple in the Members table which has that memberID, then update the tuple in Members that has that memberID. Update name, address and expirationDate for that existing Members tuple to equal the corresponding attribute values in the ModifyMembers tuple. Also, make the value of the isCurrent to TRUE. Don't modify the joinDate value which is in the existing Members tuple.

Your transaction may have multiple statements in it. The SQL constructs that we've already discussed in class are sufficient for you to do this part (which is one of the hardest parts of Lab3).

[Today's date: There is a PostgreSQL function CURRENT_DATE that you can use as if it was a constant to set a value in a SQL statement. Don't use the DATE keyword before CURRENT_DATE, just write CURRENT_DATE.]

2.3 Add Foreign Key Constraints

Important: Before running Sections 2.3, 2.4 and 2.5, recreate the Lab3 schema using the *create_lab3.sql* script, and load the data using the script *load_lab3.sql*. That way, any database changes that you've done for Combine won't propagate to these other parts of Lab3.

Here's a description of the Foreign Keys that you need to add for this assignment. (Foreign Key Constraints are also referred to as Referential Integrity constraints.) The create_lab3.sql file that we've provided for Lab3 includes only some of the Referential Integrity constraints that were in the Lab2 solution, but you're asked to use ALTER to add additional constraints to the Lab3 schema.

The load data that you're provided with should not cause any errors when you add these constraint. Just add the constraints listed below, exactly as described, even if you think that additional Referential Integrity constraints should exist. Note that (for example) when we say that every monster (monsterID) in the Battles table must appear in the Monsters table, that means that the monsterID attribute of the Battles table is a Foreign Key referring to the Primary Key of the Monsters table (which also is monsterID).

- Each monsterID in the Battles table must appear in the Monsters table as a Primary Key (monsterID). If a tuple in the Monsters table is deleted, and there are Battles tuples that correspond to that monster, then that Monsters tuple deletion should be rejected . If the Primary Key monsterID of a Monsters tuple is updated, then all battles in Battles involving that monster should also be updated to the new monsterID value .

- Each character (characterMemberID, characterRole) that appears in the Battles table must also appear in the Characters table as a Primary Key (memberID, role). If a tuple in the Characters table is deleted, then all Battles tuples involving that character should also be deleted. If the Primary Key (memberID,

role) of a Characters tuple is updated and there are Battles tuples that correspond to that character, then the update of the Characters tuple should be rejected.

- Each character (ownerMemberID, ownerRole) that appears in the Things table as an owner of a thing must also appear in the Characters table as a Primary Key (memberID, role). If a tuple in the Characters table is deleted, then all Things tuples that that Character should have both ownerMemberID and ownerRole set to NULL. If the Primary Key (memberID, role) of a Characters tuple is updated, then all tuples in Things that are owned by that character should be updated to the new (memberID, role) owner value.

Write commands to add foreign key constraints in the same order that the foreign keys are described above. Your foreign key constraints should have names, but you may choose any names that you like. Save your commands to the file *foreign.sql*

2.4 Add General Constraints

General constraints for Lab3 are:

1. In Things, cost must be positive. (As I hope all of you know, zero is not positive.) This constraint should be named positiveCost.

2. In Monsters, if battlePoints is greater or equal to 40, then the monsterKind must represent a giant or a basilisk or be NULL. (A giant has monsterKind 'gi', and a basilisk has monsterKind 'ba'.) This must be handled by a single constraint. Please give a name to this constraint when you create it. This constraint should be named majorMonsters.

3. In Members, if expirationDate is NULL then isCurrent must also be NULL. This constraint should be named expirationCurrrent.

Write commands to add general constraints in the order the constraints are described above, and save your commands to the file *general.sq*l. Note that the values TRUE and UNKNOWN are okay for a Check constraint is okay, but FALSE isn't.

2.5 Write Unit Tests

Unit tests are important for verifying that your constraints are working as you expect. We will require tests for just a few common cases, but there are many more unit tests that are possible.

For each of the 3 foreign key constraints specified in section 2.3, write one unit test:

- An INSERT command that violates the foreign key constraint (and elicits an error). You must violate that specific foreign key constraint, not any other constraint.

Also, for each of the 3 general constraints, write 2 unit tests, with 2 tests for the first general constraint, followed by 2 tests for the second general constraint, followed by 2 tests for the third general constraint.

- An UPDATE command that meets the constraint.
- An UPDATE command that violates the constraint (and elicits an error).

Save these 3 + 6 = 9 unit tests in the order specified above in the file *unittests.sql*.

2.6 Working with a View

Important: Before starting Section 2.6, recreate the Lab3 schema once again using the *create_lab3.sql* script, and load the data using the script *load_lab3.sql*. That way, any changes that you've done for previous parts of Lab3 (e.g., Unit Test) won't affect the results of this query.

2.6.1 Create a view

There two ways that we could determine whether or not a character (or monster) has been defeated. One way is by looking at the wasDefeated attribute of a characters (or monster). Another way is by looking at the battles involving that character (or monster).

The number of points that a monster would have in a battle is the battlePoints for that monster. But the full battle points that a character would have in a battle equals the battlePoints for that character's role plus the number of extraBattlePoints for all the things which that character owns.

Create a view called **FullBattlePointsView** which for each character gives the full battle points which that character would have in a future battle. The attributes in your view should appear as memberID, role, name and fullBattlePoints. But only include a character in FullBattlesPoints View if that character owns at least one thing.

Save the script for creating that view in a file called *createview.sql*

2.6.2 Query a View

For this part of Lab3, you'll write a script called *queryview.sql* that contains a query which we'll informally refer to as the "WrongWasDefeated" query. (You don't actually name the query.) WrongWasDefeated uses **FullBattlePointsView** and (possibly) some tables. In addition to the WrongWasDefeated query, you must also include some comments in the queryview.sql script; we'll describe those necessary comments below.

A character in a battle lost that battle if the monsterBattlePoints for the monster in that battle is greater than the characterBattlePoints for that character. Write and run a SQL query which finds the characters that are in FullBattlePointsView who a) lost at least one battle to a monster , and b) whose wasDefeated value is FALSE. (Don't consider the characters who aren't in FullBattlePointsView.)

That attributes in your result should be the memberID, role, name and fullBattlePoints for the character, and the number of times that that character lost battles. Those attributes should appear as theMemberID, theRole, theName, theFullBattlePoints and numLosses. But only include a character in your result if theName isn't NULL and the value of numLosses is at least 3.

Then write the results of the "WrongWasDefeated" query in a comment. *The format of that comment is not important; it just has to have all the right information in it.*

Next, write commands that delete just the tuples that have the following Primary Keys from the Battles table:

- The Battles tuple whose Primary Key is (111, 'cleric', 925).
- The Battles tuple whose Primary Key is (101, 'knight', 944).

Run the "WrongWasDefeated" query once again after those deletions. Write the output of the query in a second comment. Do you get a different answer?

You need to submit a script named *queryview.sql* containing your query on the views. In that file you must also include:

- a comment with the output of the query on the load data before the deletions,
- the SQL statements that delete the tuples indicated above,
- and a second comment with the second output of the same query after the deletions.

You do not need to replicate the query twice in the *queryview.sql* file (but you won't be penalized if you do).

It probably was a lot easier to write this query using the view than it would have been if you hadn't had it!

2.7 Create an Index

Indexes are data structures used by the database to improve query performance. Locating the tuples in the Things table for a particular initialRoomID that have a particular thingKind might be slow if the database system has to search the entire Things table (if the number of Things was very large). To speed up that search, create an index named ThingFinder over the initialRoomID and thingKind columns (in that order) of the Things table. Save the command in the file *createindex.sql*.

Of course, you can run the same SQL statements whether or not this index exists; having indexes just changes the performance of SQL statements. But this index could make it faster to determine if there which things are in a particular room, or determine if there any things in a particular room that are of a particular thingKind

For this assignment, you need not do any searches that use the index, but if you're interested, you might want to do searches with and without the index, and look at query plans using EXPLAIN to see how queries are executed. Please refer to the documentation of PostgreSQL on EXPLAIN that's at https://www.postgresql.org/docs/14/sql-explain.html

3 Testing

Before you submit, login to your database via psql and execute the provided database creation and load scripts, and then test your seven scripts (combine.sql foreign.sql general.sql unittests.sql createview.sql queryview.sql createindex.sql). Note that there are two sections in this document (both labeled **Important**) where you are told to recreate the schema and reload the data before running that section, so that updates you performed earlier won't affect that section. Please be sure that you follow these directions, since your answers may be incorrect if you don't.

4 Submitting

1. Save your scripts indicated above as combine.sql foreign.sql general.sql unittests.sql createview.sql queryview.sql createindex.sql. You may add informative comments inside your scripts if you want (the server interprets lines that start with two hyphens as comment lines).

2. Zip the files to a single file with name Lab3_XXXXXXX.zip where XXXXXXX is your 7-digit student ID, for example, if a student's ID is 1234567, then the file that this student submits for Lab3 should be named Lab3_1234567.zip To create the zip file you can use the Unix command:

   zip Lab3_1234567 combine.sql foreign.sql general.sql unittests.sql createview.sql queryview.sql createindex.sql

   (Of course, you use your own student ID, not 1234567.)

3. You should already know how to transfer the files from the UNIX timeshare to your local machine before submitting to Canvas.

4. Lab3 is due on Canvas by 11:59pm on Tuesday, November 15. Late Things will not be accepted, and there will be no make-up Lab assignments.