COMP3421

Assignment 2: Data Migration

20 points

Problem Background:
A dataset has been generated containing information about video games that sold more than 100,000 copies. The dataset was obtained from Kaggle.com (a great site to find free interesting datasets) and modified slightly to make it a bit easier to use for this project. The data is in a single CSV format. Your task is to design a normalized database to store the data, migrate the data from the text file to your database, and then create a series of stored procedures to answer common queries on the dataset.

Step 1: Load the raw CSV data into MySQL

You will be given a large raw data CSV file. I do recommend loading it with Excel (you can just open any csv file and it will load into Excel). This will allow you to examine the data in bulk form.

In order to create the normalized tables we will eventually want for this assignment, it will be easiest if we first move all the CSV data into MySQL as one large table that matches the CSV data. That will not only allow us to examine the data more easily, but also make it easier to migrate the data in a future step.

There are several different ways we could load the raw CSV data into MySQL, including the Table Data Import Wizard from Workbench. However, I have been trying to avoid using Workbench Wizards for this class so you can learn how to do things across any database. There is a bulk file loading option that we will use to load from either Java or Python. However, this option requires several permissions to be set on the database server to accept that data. Thus, I am providing the bulk loading code (in both Java and Python) that sets these permissions for you. Please use one of these files to load in the original data. Note that this will create a new table called vg_csv in your database. Further note that I set all the field types to TEXT in this table no matter what the data is supposed to be (int, date, string, etc). This is to handle missing or wrong fields in the csv file. If we had tried to load things as ints or dates and they are missing or incorrect in the CSV file (which happens often), then those lines might crash the load or, worse, silently drop the offending line. We would rather load in the data, no matter what it is, as TEXT. This will load in anything, including missing or incorrect data. Then it will be up to us when we are looking at the data with SQL queries later to determine what to do with missing or incorrect data. You should make sure you use correct types of fields in all the actual tables you will be creating in step3.

Step 2: Create the Data Model

Examine the data both in Excel form and with selects on vg_csv. Derive the data relationships and create a crows-foot ER model using DrawIO for a database to manage this data. You will be turning in your crows-foot ER model as part of the submission. Name this file **vgCrow.** Your ER model should include all the primary keys and foreign keys just like in the lab.

While you do not need to formally normalize the data to 3rd normal form, if you examine the data in ER model form you should end up with something close to 3rd normal form. Make sure you create surrogate keys for entities that do not have unique primary key values. Do not use text fields such as names for primary keys. Use the entity and attribute naming conventions discussed in class. And pay special attention to your 1-M and M-M relationships. You might need to examine your data with selects to help you determine what type of relationship you are looking at.

Step 3: Create the DDLs and migrate data to the normalized table designs

Next create a .sql script file that does 3 things. It should first drop all tables created in the following steps so that you can start fresh with each running of the script.

Next, it should use DDL create table commands to create the tables necessary for the design. Please have all your table start with vg_. Make sure you include all the pk/fk constraints. Your normalized tables should have all the right types such as ints, doubles, dates and varchars, even though the csv file is all text. That is, dates should be sql dates, etc.

The last thing is the data migration. Note you can use bulk inserts with nested queries to help with this task. For example, I could do an insert like the following to generate a large table of data that gets inserted in bulk into table1. Note that you can reference the outer query table field values from a nested query, although sometimes you may need to reference it with the table name to avoid aliases.

```
INSERT INTO table1 (field1, field2, field3)
(SELECT x, y, z FROM table2)
```

Make sure to let MySQL pick the surrogate primary key values. This implies that you should never hardcode those surrogate keys, as either primary or foreign when updating. You will need to "reverse lookup" the primary keys when inserting foreign keys. Make sure the "reverse lookup" fields you are using select a unique row. If they don't it might be a sign you have the wrong cardinality on your relationships.

If the data has bad/missing fields, you will need to handle those during inserting. While standard SQL can't do loops, it does have a simple if/else construct which can be potentially useful:

```
(CASE WHEN booleanTest THEN
        trueReuslt
ELSE
        falseResult
END)
```

The file that contains your table creating and migration code should be called **vgMigration.sql**.

Step4: Stored Procedures

To make things easier for clients using your database, you will next create the following stored procedures that answer common queries/preform common tasks. You should put these in a file called **vgStoredProcedures.sql**.

gameProfitByRegion

This procedure should take 2 parameters: the minimal profit necessary to be in the returned results and the region. For example, gameProfitByRegion(30, 'WD') should return every game title along with its profit for the entire world (WD) where the profit exceeded 30 million in sales. The regions we define are World (WD), North America (NA), Europe (EU), and Japan (JP). Note that these regions do not need to be integrated in your tables but can be handled with procedure programming constructs. You may assume the region given is valid.

genreRankingByRegion

This procedure should take 2 parameters: a genre name and region. For example, genreRankingByRegion('Sport', 'WD') should return the profit ranking of all game sales of 'sports' games across the entire world (wd). The regions we define are the same as above. The ranking is single number that

tells where that genre ended up in profit for the region. Note that you may want to use the rank() over sql construct to help calculate this number. You may assume the genre and region given are valid.

publishedReleases
This procedure should take 2 parameters: a publisher name and a genre name. For example, publishedReleases('Electronic Arts', 'Sport') should return the total number of titles Electronic Arts has released in the Sports genre. You may assume the publisher and genre given are valid.

addNewRelease
This procedure should take 4 parameters: game title, platform name, genre name, and publisher name. For example, addNewRelease('Foo Attacks', 'X360', 'Strategy', 'Stevenson Studios') should add a new release entitled 'Foo Attacks' from a new publisher 'Stevenson Studios' with existing platform 'X360' and existing genre 'Strategy'. Note that any of the 4 parameters may be existing or new titles/platforms/genre/publishers. You may end up needing to add to multiple tables in your design to accomplish this task. This procedure should simply return a statement that data was added.

Finally, you should test your stored procedures with the following calls. Put these calls into a file called **vgCalls.sql**. You should also cut-paste the results obtained from each of these calls into the file as well under each call, so that it can be checked when grading. Here are the calls you need to check:

```
-- Calls to game profit by region
call gameProfitByRegion(35, 'WD');
call gameProfitByRegion(12, 'EU');
call gameProfitByRegion(10, 'JP');

-- Calls to genre ranking by region
call genreRankingByRegion('Sports', 'WD');
call genreRankingByRegion('Role-playing', 'NA');
call genreRankingByRegion('Role-playing', 'JP');

-- Calls to published releases
call publishedReleases('Electronic Arts', 'Sports');
call publishedReleases('Electronic Arts', 'Action');

-- Calls to add new release
call addNewRelease('Foo Attacks', 'X360', 'Strategy', 'Stevenson Studios');
-- Note that to show this works properly, you will need to perform some selects based on your table design to show that the data did in fact get inserted.
```

Step5: Writeup

Lastly, you should provide a writeup (**vgWriteup.pdf**) that explains the choices you made when creating your database design. In particular, I want you to talk about why you made the tables you made, selected the cardinality on tables you did (1-M, M-M), and selected which fields to go in which tables. You may also want to talk about any issues you ran into during the migration.

The files you should turn in for this assignment are:

vgCrow.png
vgMigration.sql
vgStoredProcedures.sql
vfCalls.sql
vgWriteup.pdf

Please zip these together, named ***yourname(s)*****Migration.zip** and upload to Canvas.

You may work with a partner on this assignment if you wish.

Crow's foot ER diagram: 6 points
Table creation and migration: 6 points
Stored Procedures: 6 points
Writeup: 2 points