Assignment 2 – SQL and Server-Side Application Logic

Submission

Your submission should include three items:

- 5 .sql files as outlined later in this assignment.
- 5 .txt files as outlined later in this assignment
- DO NOT submit a .zip file. 10 separate files is what is required.
- MAKE SURE in all files it is clear which question – and which part of which question – is being answered.

Reminders

This will form the basis of future assignments. We will continue to grow and enhance this database. Make sure you do a complete job of these tasks to help ensure a smooth process for future assignments.

Tasks

1 – (40%) – Referential Integrity - Lets make sure our music database contains consistent data with good data integrity.

- (a) ALTER your tables so that no columns can contain a NULL value. All fields must be mandatory.
- (b) ALTER the *bandmembers* table so that the End Year is set to 9999 if the band member is still currently in the band. You can use the DEFAULT value for that.
- (c) Define a Primary Key / Foreign Key relationship between Band Name in the *bandmembers* and *bandrecognition* table – and the *bandinfo* table - to ensure that it is impossible to have a Band Name in either table *bandmembers* or *bandrecognition* table which is not already in the *bandinfo* table.

- (d) Add a set of CONSTRAINTS on the *bandinfo* table to the following columns to ensure that their data values are meaningful: Formation Year, Current Status, Number of Releases, Number of Band Members, Genre
- (e) Add a set of CONSTRAINTS on the *bandrecognition* table to the following columns to ensure that their data values are meaningful: Nomination, Year
- (f) Add a set of CONSTRAINTS on the bandmembers table to the following columns to ensure that their data values are meaningful: Start Year, End Year, Role
- (g) Build a TRIGGER called *bandmemnum* which ensures that any time a Band Member is added or removed from the *bandmembers* table that the Number of Band Members in the *bandinfo* table is updated accordingly
- (h) Build a TRIGGER called *bandmemstart* which ensures that any time a Band Member is added or updated in the *bandmembers* table that the Band Member Start Year is not before the Band Formation Year in the *bandinfo* table.

2 – (10%) – SQL Queries

- (a) Write a query which lists all the bands, their formation year, current status and base city which have been nominated for at least 1 award (this includes actually receiving the award as well). Group the result set by base city and within those groups order the result set by number of awards the band has been nominated for or received.
- (b) Write a query that lists all of the current band members for each band in your database. The result set should include: band name, formation year, band member last name, band member

first name, band member start year, band member role. Order the bands by their formation year – then – for each member in the band order by last name.

3 – (20%) – Stored Procedure

- (a) Write a stored procedure called *bandhistory* which takes a band name as input and returns a list of all band members who were ever part of the band, including current members. The result set should be ordered by start year. The result set should include a title which says "Band Member History for band: <band name>" followed by the list of band members.
- (b) Write a stored procedure called *bandyear* which takes a band name and year as input and returns a list of all band members who were part of the band on Jan 1 of that year. (You will use an AS OF query for this). The output should be a list of the band members who were in the band at that time.
- NOTE: In order to accomplish these tasks you must also add in history data for at least 5 of the bands in your *bandmembers* table. Make sure there's at least 2 band member changes for each of the 5 bands in the history (for instance the band has gone through 3 drummers over the past 10 years).

4 – (10%) – User Defined Function

- Write a user defined function called *wordfun* which takes any character column as input and returns the pig latin version of the whatever was in that column.
- Pig Latin does the following to a word:

    - If the first letter is a not a vowel it moves the first letter to the end of the word and adds “ay” after. For instance “Pig -> igpay”, “King -> ingkay”, etc.
    - If the first letter is a vowel it simply adds “ay” to the end of the word. For instance “Anderson -> Andersonay”.
- For the purposes of this exercise “A”, “E”, “I”, “O” and “U” are vowels (upper and lower case count).

5 – (20%) – Working with the JSON (*releases*) column

- (a) Write a query – you can use SQL or NoSQL approaches depending on what database system you are using – which provide a result set summarizing the releases a band has had. I am leaving this very flexible in how you do this because different database systems provide different capabilities here – but – the output must have a list of release names, type of release and year of release. The band name is always the input.
    - For example if you use SELECT: SELECT …… FROM BANDINFO WHERE BAND = “*bandname*”
    - For example if you decide to create a stored procedure to accomplish this the bandname would be the input to the stored procedure
- (b) Write a stored procedure called *releaseupd* which goes through the entire *bandinfo* table and makes sure that the Number of Releases column is accurate for all bands. Again, you can use any approach you like to traverse the JSON document based on the capabilities of the database system you are using.

6 – BONUS – (10%) – If you can find a way – with your database system – to introduce some form of automated data integrity construct (CONSTRAINT, TRIGGER, etc) which ensures that the Number of Releases in the *bandinfo* table always matches exactly what the releases column JSON information says – you will get this bonus mark. This is a bonus only since some database systems may not provide any way for this to work.

What to hand in:

Testing -

You must test everything you write for every question. You must demonstrate that your code works – whether it is a SQL statement, a stored procedure, a trigger, a CONSTRAINT – whatever it is – you must demonstrate it functions correctly. This includes error handling where applicable.

For instance – in Q1 – test each of your data integrity rules to ensure they are working. Try and INSERT, UPDATE and/or DELETE data (as appropriate) which attempts to break the data integrity rules above and show that everything is working as it should.

Filenames –

For each question above you would hand in two files:

(i) An **A2Qn.sql** file which contains all of the above SQL code (ALTER TABLE commands, CREATE PROCEDURE commands, CREATE TRIGGER commands, etc). Make sure each section of code is clearly marked and commented and follows proper programming best practices as outlined in the lecture.

(ii) An **A2Qn.txt** output file which shows the results of all your tests.

In the end, you should hand in 5 .sql files called A2Q1.sql, A2Q2.sql, …. A2Q5.sql – and – 5 .txt (output) files called A2Q1.txt, A2Q2.txt, …. A2Q5.txt.

Marks will be deducted if you do not follow these naming conventions. It needs to be clear what question – and what part of what question – I am marking.

Marking Scheme –

Q1 - There are 8 categories of data integrity outlined above (a) – (h). Each is worth 5% for a total of 40%.

Q2 – There are 2 SQL queries (a) & (b) – each is worth 5% for a total of 10%

Q3 – There are 2 Stored Procedures (a) & (b) – each is worth 10% for a total of 20%

Q4 – There is 1 User Defined Function – worth 10%

Q5 – There are two parts to the JSON portion – worth 10% each for a total of 20%

Q6 – BONUS – 10%

To receive the marks you must have the appropriate code in the .sql file and tests in the .txt file showing the code works. Marks will be deducted if you fail to follow programming best practices (such as good comments) in your .sql file.