James Zafiri
11/26/2023
CSC6302 Module 04

Quiz Two

1.) An inner join only returns the rows that have matching values in both tables, and the rows that do not have matching values are excluded (focuses only on intersection). A full outer join returns all the data from both tables, as well as any matching rows if they are available. If there is not a match in any row of a table, the results will still include that row, but with null values for the column that did not match.

2.) While the default delimiter for MySQL statements is a semicolon, functions and procedures in MySQL contain multiple statements within them. Because of this, we need to change the delimiter so that the database engine knows where this group of statements (the function/procedure) begins and ends. This way they can be logically interpreted as a single statement from the system.

3.)
```
SELECT PlaylistName, SongName
FROM playlist AS p JOIN playlistcontent AS pc ON p.PlaylistName = pc.PlaylistName
JOIN song AS s ON pc.SongId = s.Id
ORDER BY p.PlaylistName, s.SongName;
```

4.) The purpose of line 13 is that it's used as a constraint on what values can be entered into the Age column of the Artist table. It ensures that when you insert/update a record in the Artist table, the value of Age must be higher than 13, or else it will be rejected when trying to insert a value of 13 or less.

5.) A join is more efficient than a Cartesian product because you combine rows based on a condition (ON clause for example), and this is usually specified in the WHERE clause. For Cartesian products, you do not specify a join condition and it will pull all the data from the tables. We know from the order of operations, that FROM comes first, then next in the order WHERE is used (which is the case for Cartesian, but not a join). Because of this, Cartesian products pull of the data without a join condition, and this can be inefficient when working with a lot of data. However, when using a join condition, the database engine will apply that condition during the actual join operation. This saves us from unnecessary combinations, so it is more efficient.

6.)
```
CREATE TABLE IF NOT EXISTS playlistcontent (
PlaylistName varchar(100) not null,
SongId int not null,
PRIMARY KEY (PlaylistName, SongId),
```

```
FOREIGN KEY (PlaylistName) REFERENCES playlist(PlaylistName) on update cascade on
delete cascade,
FOREIGN KEY (SongId) REFERENCES song(Id) on update cascade on delete cascade
);
```

7.)
```
DROP FUNCTION IF EXISTS GetPlaylistSongCount;
delimiter $$
CREATE FUNCTION GetPlaylistSongCount (MyPlayListName varchar(100))
RETURNS INT deterministic
BEGIN
        DECLARE songCount INT;
        SELECT count(SongId) into songCount
        FROM playlistcontent
        WHERE PlaylistName = MyPlaylistName;
RETURN songCount;
END $$
delimiter;
```

8.)
```
SELECT UPPER(SongName) AS UpperSongName
FROM song;
```

9.) One reason why stored procedures are ideal is because of code reusability and logic being stored in the database. Business logic and database operations are separated, and all the application must do is interact with the stored procedure. Also, the same code for the stored procedure can be used multiple times throughout the application easily, which helps code design. Another reason why they are ideal is it improves the performance of the application. It only needs a single command to call a stored procedure from the database, instead of multiple SQL statements within the code. Overall stored procedures are a very effective interface that provides abstraction for database operations.

10.)

a.) The scope of numSongs is **local scope**. This is because it is declared within the function itself, and it is defined with a begin/end block.

b.) The scope of desiredLength is **session scope**. This is because the function is being called in a session, so the arguments to functions/stored procedures are user defined variables.

11.)

a.)
```
delimiter $$
DROP PROCEDURE IF EXISTS InsertArtist $$
CREATE PROCEDURE InsertArtist (newFirstName varchar(200), newLastName
varchar(200), newStageName varchar(200), newAge int)
BEGIN
```

```
        INSERT INTO artist (FirstName, LastName, StageName, Age)
        VALUES (newFirstName, newLastName, newStageName, newAge);
    END;
    $$
    delimiter ;
```

b.) CALL InsertArtist('Curtis', 'Jackson', '50 Cent', 48);

12.)

```
SELECT *
FROM song AS S
WHERE NOT EXISTS (
        SELECT *
        FROM playlistcount AS pc
        WHERE pc.SongId = s.Id
);
```