MET CS 669 Database Design and Implementation for Business
SQL Lab 3 Instructions: Aggregating Data

OBJECTIVE

Sometimes we are interested in specific data item that resides in our database, and sometimes we are interested in the result of aggregating multiple data items. For example, a store may be interested in the monetary amount of a single sale, but may be equally or more interested in the sum the monetary amount of all sales that occurred on a specific day. SQL provides many useful ways to aggregate data, and the objective of this lab is to provide you with a practical exercise of using SQL queries to aggregate data.

PREREQUISITES

Before attempting this lab, it is best to read the textbook and lecture material covering the objectives listed above. While this lab shows you how to create and use these constructs in SQL, the lab does not explain in full the theory behind the constructs, as does the lecture and textbook.

REQUIRED SOFTWARE

The examples in this lab will execute in modern versions of Oracle and Microsoft SQL Server as is. If you have been approved to use a different RDBMS, you may need to modify the SQL for successful execution, though the SQL should execute as is if your RDBMS is ANSI compliant.

The screenshots in this lab display execution of SQL in the Oracle SQL Developer client and in Microsoft SQL Server Management Studio. Note that if you are using Oracle, you are not required to use Oracle SQL Developer; there are many capable SQL clients that connect to Oracle.

SAVING YOUR DATA

If you choose to perform portions of the lab in different sittings, it is important to *commit* your data at the end of each session. This way, you will be sure to make permanent any data changes you have made in your curent session, so that you can resume working without issue in your next session. To do so, simply issue this command:

```
COMMIT;
```

We will learn more about committing data in future weeks. For now, it is sufficient to know that data changes in one session will only be visible only in that session, unless they are committed, at which time the changes are made permanent in the database.

LAB COMPLETION

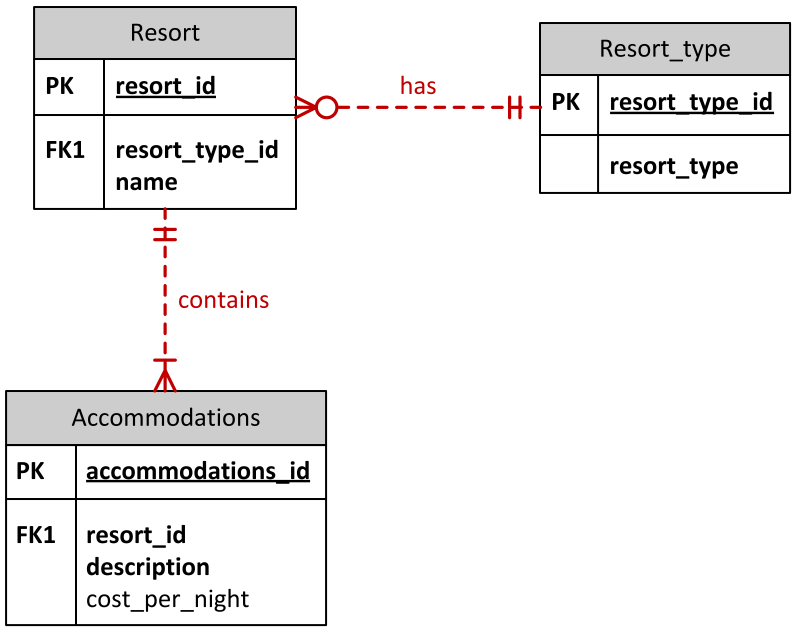
Use the submission template provided in the assignment inbox to complete this lab.

SECTION ONE

OVERVIEW

We will work with the resort schema illustrated below, which tracks resorts, their type (such as “ocean”or “lakeside”), and the rooms offered by the resorts. The relationships are described with the following business rules:

- Each resort has a type.
- Each resort type may be assigned to multiple resorts.
- Each resort contains multiple rooms.
- Each room is contained by a single resort.



Note that the bolded columns represent those with a NOT NULL constraint. The only nullable column in this schema is *cost_per_night* table in the Accommodations table.

STEPS

1. Create the resort schema illustrated above in its entirety, including all primary and foreign key constraints.

   Hints
   - Use CREATE TABLE statements to create each table with the primary key and not null constraints.
   - Use ALTER TABLE statements to create the foreign constraints after all tables have been created.
   - Use datatypes that are appropriate for each kind of value. You can use your discretion for the maximum number of digits in the primary key fields.

2. Due to the large number of commands, you do not need to provide individual screenshots for the commands in #1. Rather, provide a one or two screenshots containing all of your commands in #1.

3. Insert the following data into the tables. *Note that the primary and foreign key values, such as those for resort_id and others, have not been given to you, but you still need primary and foreign keys to represent the schema.* You will need to assign your own values to the primary and foreign keys.

   Resort_type

resort_type
Ocean
Lakeside
Mountaintop
Country

   Resort

name	resort_type
Light of the Ocean	Ocean
Breathtaking Bahamas	Ocean
Mountainous Mexico	Mountaintop
Greater Lakes	Lakeside

Accommodations

description	resort	cost_per_night
Bungalow 1	Light of the Ocean	$289
Bungalow 2	Light of the Ocean	$289
Bungalow 3	Light of the Ocean	$325
Suite 101	Breathtaking Bahamas	$199
Suite 102	Breathtaking Bahamas	$199
Suite 201	Breathtaking Bahamas	$250
Suite 202	Breathtaking Bahamas	$250
Room 10	Mountainous Mexico	$150
Room 20	Mountainous Mexico	
Cabin A	Greater Lakes	$300
Cabin B	Greater Lakes	
Cabin C	Greater Lakes	$350

Hints
- Use an INSERT INTO commands to insert each row.
- Make sure to specify the column list, and the VALUES clause, for each INSERT INTO command.
- Make sure to insert the values in the correct order, so that the referenced values are inserted *before* the referencing values.
- Make sure to assign your own primary and foreign key values.

4. Capture a screenshot of the commands and the results of their execution. It may be helpful if you capture 3 screenshots, where each screenshot has all of the INSERT INTO commands for a single table, and its execution results.

5. While we could write a SQL query to retrieve specific data items of interest, such as the name of a resort, or the type of a resort, sometimes we are interested in the result of aggregating multiple data items. For instance, perhaps we are interested in the total number of accommodations available in our database. This result can be obtained by writing a query that counts the number of rows in the Accommodations table. Such a query uses a SQL construct termed an *aggregate function* in conjunction with the already familiar SELECT clause.

   An aggregate function has key differences from a standard function. One difference is how many results an aggregate function yields. A standard function always yields one value per row in a result set, while an aggregate function always yields a single value, regardless of the number of rows in the result set. It follows that another difference is the behavior of an aggregate function. Standard functions use the value(s) from a single row to generate a result, while aggregate functions use values from multiple rows to generate a result.

While there are a virtually unlimited number of aggregations that can be performed by an aggregate function, there are some common kinds of aggregate functions. One kind applies a mathematical operation to a column's value across all rows in a result set. For example, the SUM function adds up a column's value across multiple rows. The COUNT function counts all rows that have a value for a given column. Another kind locates one value, from a column's value across multiple rows, that has a desired property. For example, the MAX function returns the highest numerical value across multiple rows, and the MIN function returns the smallest numerical value across multiple rows.

What aggregate function can we use to count the number of rows in the Accommodations table? The COUNT function will suffice.

```
SELECT COUNT(accommodations_id)
FROM Accommodations;
```

In the command just executed, you may have noticed that the "COUNT(accommodations_id)" construct was used instead of the name of a column from the Accommodations table. This is how we instruct the DBMS to use the COUNT function to count the number of accommodations_id values, instead of simply listing the accommodations_id values or other columns' values.

In order to understand how this aggregate function is applied to the result set, you should be made aware of the responsibility of each SQL construct. In any SQL statement similar to the above, the FROM clause determines which tables in the relational database are accessed. Join conditions, if present, in combination with expressions in the WHERE clause, if present, determine what rows from those tables are listed in the result set. Column names in the SELECT clause determine what columns from those tables are listed in the result set. Thus the SELECT/FROM/WHERE clause determines what rows and columns from what pre-existing tables in the relational database are included in the result set.

Aggregate functions *do not* determine the pre-existing database tables, or their rows and columns, that will be included in the result set. Rather, aggregate functions *derive results* from the rows and columns specified in the SELECT/FROM/WHERE clause. This being the case, you can think of the ordering as occurring in two steps:

1. Retrieve the rows and columns specified in the SELECT/FROM/WHERE clause.
2. Apply the aggregate function to these rows and columns.

The results of these SQL statements are always guaranteed to be the same as if the above ordering is followed; however, any particular DBMS may optimize, parallelize, and reorder operations so that the result is retrieved more quickly.

6. Capture a screenshot of the command and the result of its execution.

7. Imagine that a potential resort customer is budgeting for an upcoming vacation, and so telephones a request for the lowest cost per night of all of the accommodations. To obtain this information for the customer we will need to use the MIN aggregate function, as follows.

```
SELECT MIN(cost_per_night)
FROM Accommodations;
```

   Notice that the lowest cost per night is listed, ignoring the accommodations that have no cost per night listed. The MIN function thus ignores null values.

8. Capture a screenshot of the command and the result of its execution.

9. Imagine that management is considering building a new resort, and wants more information to help them determine what type of resort to build. They come to you asking you for the number of existing resorts of each type, and indicate that they want the list to be ordered from the highest number of resorts to the lowest number of resorts.
   In order for you to obtain this information, you will need to combine an inner join with the COUNT aggregate function. You will also need to use the GROUP BY construct in your query, as well as an ORDER BY query, as follows.

```
SELECT Resort_type.resort_type_id, Resort_type.resort_type,
       COUNT(Resort.resort_id) AS nr_resorts
FROM Resort_type
JOIN Resort ON Resort_type.resort_type_id = Resort.resort_type_id
GROUP BY Resort_type.resort_type_id, Resort_type.resort_type
ORDER BY nr_resorts DESC
```

   The purpose of the GROUP BY construct in this query is to instruct the COUNT aggregate what results should be counted. *Without* an accompanying GROUP BY construct, the COUNT aggregate function simply counts all specified items in the result set. *With* an accompanying GROUP BY construct, the results are first grouped by the specified columns, and then the COUNT function counts the items within that group, rather than everything in the result set.

   An alternative way to think about this concept is that an aggregate function always counts in groups, and that the omission of the GROUP BY statement means that there is only one group, which is the entire result set.

The ORDER BY clause instructs the DBMS to order the entire result set by the specified column or columns. The DESC keyword means to order the result set in a descending fashion (from the highest to the lowest). The ASC keyword, which is the default, would mean to order the result set in an ascending fashion (from the lowest to the highest).

You may have also noticed the "AS nr_resorts" clause in the query above. This is the way we specify a column alias. The COUNT column in the result set will be titled "nr_resorts".

10. Capture a screenshot of the command and the result of its execution.

11. Imagine that after receiving the information you provided them in Step #9, management requests more detailed information to help them determine what type of resort to build. They ask for a listing of all resort types that have fewer than three accommodations available, along with the number of accommodations for the matching resort types. That is, out of the available resort types -- Ocean, Lakeside, Mountaintop, and Country – they would like to know which of these have fewer than three accommodations available across all resorts, as well as a listing of the actual number of accommodations available for that resort type.

    When we consider this request, we can see that it is a bit more technically complex than the request in step 9. We will need to join all three tables in our schema, use the COUNT aggregate function, and use the GROUP BY clause. We will also need to use another construct – the HAVING clause – to limit the aggregate results. The WHERE clause *limits the result set based upon values in individual rows,* while the HAVING clause *limits the result set based upon aggregate results*.

```
SELECT Resort_type.resort_type_id, Resort_type.resort_type,
       COUNT(Accommodations.accommodations_id) AS nr_accommodations
FROM Resort_type
JOIN Resort ON Resort_type.resort_type_id = Resort.resort_type_id
JOIN Accommodations ON Accommodations.resort_id = Resort.resort_id
GROUP BY Resort_type.resort_type_id, Resort_type.resort_type
HAVING COUNT(Accommodations.accommodations_id) < 3
```

    Notice that to fulfill this request, we using the `HAVING COUNT(Accommodations.accommodations_id) < 3` clause so that only the results that have fewer than three accommodations are returned.

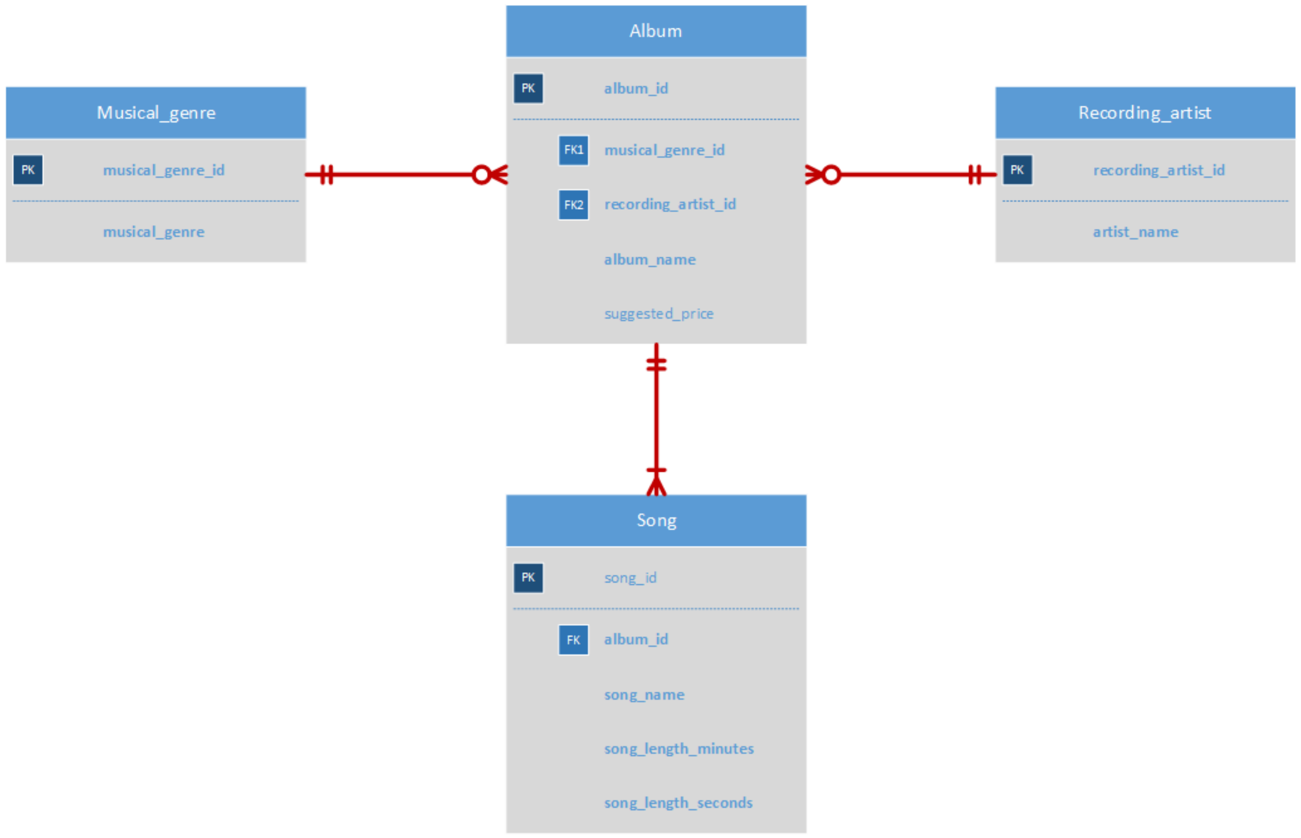
12. Capture a screenshot of the command and the result of its execution.

SECTION TWO

OVERVIEW

We will work with the musical schema illustrated below, which contains the Album, Song, Recording_artist, and Musical_genre tables and their relationships. The relationships are described with the following business rules:

- Each album is categorized by one musical genre.
- Each musical genre may categorize multiple albums.
- Each album must consist of one or more songs.
- Each song is included in exactly one album.
- Each album is recorded by a recording artist.
- Each recording artist may record one or more albums.



Note that the only nullable column in this schema is *suggested_price* in the Album table.

STEPS

1. Create the musical schema illustrated above in its entirety, including all primary and foreign key constraints.

   Hints
   - Use CREATE TABLE statements to create each table with the primary key and not null constraints.
   - Use ALTER TABLE statements to create the foreign constraints after all tables have been created.
   - Use datatypes that are appropriate for each kind of value. You can use your discretion for the maximum number of digits in the primary key fields.

2. Due to the large number of commands, you do not need to provide individual screenshots for the commands in #1. Rather, provide a one or two screenshots containing all of your commands in #1.

3. Insert the following data into the tables. *The primary key values have not been given to you, such as those for album_id and others, so you should assign your own values.*

Artist_name

artist_name
April Wine
Enya
Elvis Costello

Musical_genre

musical_genre
Big Band
Rock
World
Country
New Age
Heavy Metal

Album

album_name	suggested_price	Musical_genre reference	Recording_artist reference
Power Play	$15.99	Rock	April Wine
A Day Without Rain	$13.75	New Age	Enya
Armed Forces	$12.99	Rock	Elvis Costello
Nature of the Beast		Rock	April Wine

Song

song_name	length	Album reference
Doin' It Right	3:42	Power Play
Ain't Got Your Love	4:33	Power Play
If You See Kay	3:59	Power Play
Tell Me Why	3:39	Power Play
Fairytail	3:03	A Day Without Rain
Sun Stream	2:55	A Day Without Rain
Deireadh An Tuath	1:43	A Day Without Rain
Oliver's Army	2:58	Armed Forces
Peace, Love, and Understanding	3:31	Armed Forces
Moods For Moderns	2:48	Armed Forces
All Over Town	3:00	Nature of the Beast
Tellin' Me Lies	2:59	Nature of the Beast

Hints

- Use an INSERT INTO commands to insert each row.
- Make sure to specify the column list, and the VALUES clause, for each INSERT INTO command.
- Make sure to insert the values in the correct order, so that the referenced values are inserted *before* the referencing values.

4. Capture a screenshot of the commands and the results of their execution. It may be helpful if you capture 4 screenshots, where each screenshot has all of the INSERT INTO commands for a single table, and its execution results.

5. A music producer requests the total number of songs available in the database. Write a single query that fulfills the producer’s request.

6. Capture a screenshot of the command and the result of its execution.

7. A music reseller needs to know the highest and lowest suggested prices for all available albums. Write two queries that obtain the requested information.

8. Capture a screenshot of the two commands and the result of their execution.

9. Explain how and why the suggested price for album “Nature of the Beast” was treated differently than the other suggested price values in your queries for step 7.

10. A record company requests the names of all albums, along with the number of songs per album. Write a single query that obtains these results. Make sure to eliminate unneeded columns from the result set, and to name your columns something user-friendly and human readable.

11. Capture a screenshot of the command and the result of its execution.

12. A record company requests the names of all available genres that have at least four associated songs, and for these genres, the number of songs written in each genre. Write a single query that obtains these results. Make sure to eliminate unneeded columns from the result set, and to name your columns something user-friendly and human readable.

13. Capture a screenshot of the command and the result of its execution.

14. A record company requests the names of all recording artists, as well as the number of rock songs each artist sings. Note that some artists may not sing any rock songs, but should still be included in the list. The record company would like this list to be ordered so that the artist with the least number of rock songs appears first in the list, and the artist with the most appears last in the last. Write a single query that obtains these results. Make sure to eliminate unneeded columns from the result set, and to name your columns something user-friendly and human readable.

15. Capture a screenshot of the command and the result of its execution.