

Lab 3 Explanations

This explanation document illustrates how to correctly execute each SQL construct step-by-step for Lab 3, and explains important theoretical and practical details. Before completing a step, read its explanation here first.

Table of Contents

Section One – Aggregating Data	3
STEP 1.....	3
Code: Creating Resort Schema	4
Screenshots: Creating the Resort Schema.....	5
Code: Inserting Resort Data	8
Screenshots: Inserting the Resort Values	9
STEP 2.....	11
Code: Counting Accommodations	11
Screenshots: Counting the Accommodations.....	12
Code: Counting cost_per_night	13
Screenshots: Counting the cost_per_night Column	13
Code: Distinct Counting	14
Screenshots: Distinct Counting	14
STEP 3.....	16
Code: Obtaining the Smallest Price	16
Screenshots: Obtaining the Smallest Price	16
STEP 4.....	18
Code: Obtaining the Smallest Price (No Group)	18
MIN with No Group.....	18
Code: Obtaining the Smallest Price Per Resort.....	19
MIN Grouped by resort_id.....	19
Screenshots: Least Expensive Accommodation Per Resort.....	20
Code: Obtaining Least Expensive Accommodation with Name	21
Screenshots: Least Expensive Accommodation With Name	22
STEP 5.....	23
Code: Cheapest Price Per Resort Less Than \$225.....	23
Screenshots: Cheapest Price Per Resort Less Than \$225	24
STEP 6.....	25
Code: Resorts Bringing at Least \$800 Revenue Per Night	25
Screenshots: Resorts Bringing at Least \$800 Revenue Per Night.....	26
STEP 7.....	27
Code: Intermediate Result with no Aggregation	28
Screenshots: Intermediate Result with no Aggregation.....	28
Code: All Resort Types along with Number Over 200	29

Section One – Aggregating Data

STEP 1

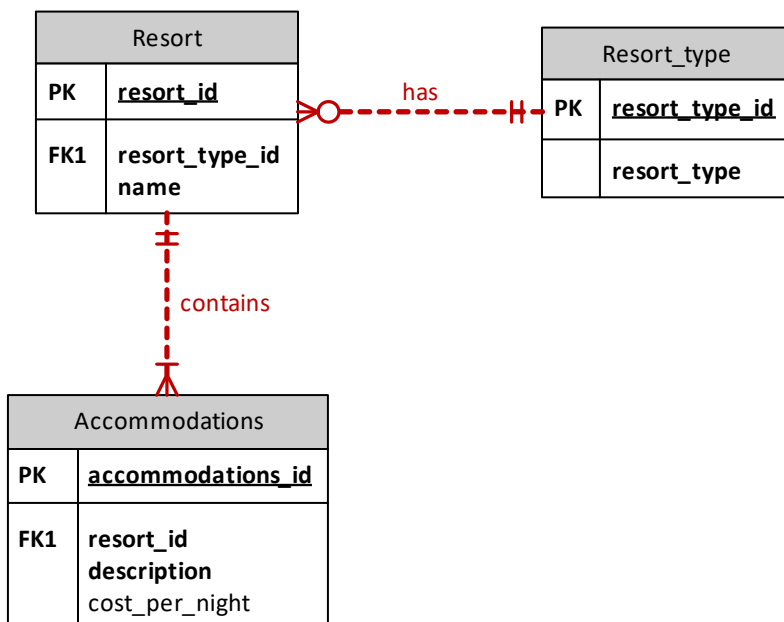
Create the tables in the schema, including all of their columns, datatypes, and constraints, and populate the tables with data. Most but not all of the data is given to you in the table below; *you should also insert information for one additional movie series of your choosing*. Although the data is in flattened representation below, you will of course insert the data relationally into the schema with foreign keys referencing the appropriate primary keys.

Genre	Creator	Series	Suggested Price	Movie	Length
Fantasy	George Lucas	Star Wars	\$129.99	Episode I: The Phantom Menace	136
Fantasy	George Lucas	Star Wars	\$129.99	Episode II: Attack of the Clones	142
...

Note that the suggested price for the Lord of the Rings series is null (has no value).

To help demonstrate how to complete the commands in this section, we 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.

We create the resort schema illustrated above in its entirety, including all primary and foreign key constraints, using the following guidelines:

- CREATE TABLE statements are used to create each table with the primary and foreign keys, and not null constraints.
- Datatypes are used that are appropriate for each kind of value, with reasonable discretion for the maximum number of digits in the primary key fields.

Here's the code we used for creating the schema.

Code: Creating Resort Schema

```
CREATE TABLE Resort_type (  
  resort_type_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_type VARCHAR(64) NOT NULL);  
  
CREATE TABLE Resort (  
  resort_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_type_id DECIMAL(12) NOT NULL,  
  name VARCHAR(255) NOT NULL,  
  FOREIGN KEY (resort_type_id) REFERENCES Resort_type(resort_type_id));  
  
CREATE TABLE Accommodations (  
  accommodations_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_id DECIMAL(12) NOT NULL,  
  description VARCHAR(255) NOT NULL,  
  cost_per_night DECIMAL(8,2) NULL,  
  FOREIGN KEY (resort_id) REFERENCES Resort(resort_id));
```

Below are screenshots of creating the schema in all three DBMS.



Screenshots: Creating the Resort Schema

Oracle

```
CREATE TABLE Resort_type (  
  resort_type_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_type VARCHAR(64) NOT NULL);  
  
CREATE TABLE Resort (  
  resort_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_type_id DECIMAL(12) NOT NULL,  
  name VARCHAR(255) NOT NULL,  
  FOREIGN KEY (resort_type_id) REFERENCES Resort_type(resort_type_id);  
  
CREATE TABLE Accommodations (  
  accommodations_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_id DECIMAL(12) NOT NULL,  
  description VARCHAR(255) NOT NULL,  
  cost_per_night DECIMAL(8,2) NULL,  
  FOREIGN KEY (resort_id) REFERENCES Resort(resort_id));
```

Script Output x Query Result x

Task completed in 0.084 seconds

Table RESORT_TYPE created.

Table RESORT created.

Table ACCOMMODATIONS created.

Microsoft
SQL Server

```
CREATE TABLE Resort_type (  
  resort_type_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_type VARCHAR(64) NOT NULL);  
  
CREATE TABLE Resort (  
  resort_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_type_id DECIMAL(12) NOT NULL,  
  name VARCHAR(255) NOT NULL,  
  FOREIGN KEY (resort_type_id) REFERENCES Resort_type(resort_type_id);  
  
CREATE TABLE Accommodations (  
  accommodations_id DECIMAL(12) NOT NULL PRIMARY KEY,  
  resort_id DECIMAL(12) NOT NULL,  
  description VARCHAR(255) NOT NULL,  
  cost_per_night DECIMAL(8,2) NULL,  
  FOREIGN KEY (resort_id) REFERENCES Resort(resort_id));
```

100 %

Messages

Command(s) completed successfully.

PostgreSQL

```
5 CREATE TABLE Resort_type (  
6 resort_type_id DECIMAL(12) NOT NULL PRIMARY KEY,  
7 resort_type VARCHAR(64) NOT NULL);  
8  
9 CREATE TABLE Resort (  
10 resort_id DECIMAL(12) NOT NULL PRIMARY KEY,  
11 resort_type_id DECIMAL(12) NOT NULL,  
12 name VARCHAR(255) NOT NULL,  
13 FOREIGN KEY (resort_type_id) REFERENCES Resort_type  
14  
15 CREATE TABLE Accommodations (  
16 accommodations_id DECIMAL(12) NOT NULL PRIMARY KEY,  
17 resort_id DECIMAL(12) NOT NULL,  
18 description VARCHAR(255) NOT NULL,  
19 cost_per_night DECIMAL(8,2) NULL,  
20 FOREIGN KEY (resort_id) REFERENCES Resort(resort_id)
```

Data Output Explain Messages Notifications Query History

CREATE TABLE

Query returned successfully in 65 msec.

We then insert the following data into the tables, making sure to relate them properly with 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

We assign our own values to the primary and foreign keys. We use the following guidelines for the insertions.

- INSERT INTO commands are used to insert each row.
- The column list and the VALUES clause is used for each INSERT INTO command.
- The values are inserted in the correct order so that the referenced values are inserted *before* the referencing values.

Here are the insert commands for inserting the data.

Code: Inserting Resort Data

```
--Resort types.
INSERT INTO Resort_type(resort_type_id, resort_type)
VALUES(1, 'Ocean');
INSERT INTO Resort_type(resort_type_id, resort_type)
VALUES(2, 'Lakeside');
INSERT INTO Resort_type(resort_type_id, resort_type)
VALUES(3, 'Mountaintop');
INSERT INTO Resort_type(resort_type_id, resort_type)
VALUES(4, 'Country');

--Resorts.
INSERT INTO Resort(resort_id, name, resort_type_id)
VALUES(101, 'Light of the Ocean', 1); --Ocean resort type
INSERT INTO Resort(resort_id, name, resort_type_id)
VALUES(102, 'Breathtaking Bahamas', 1); --Ocean resort type
INSERT INTO Resort(resort_id, name, resort_type_id)
VALUES(103, 'Mountainous Mexico', 3); --Mountaintop resort type
INSERT INTO Resort(resort_id, name, resort_type_id)
VALUES(104, 'Greater Lakes', 2); --Lakeside resort type

--Accommodations
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1001, 'Bungalow 1', 289, 101); --Accommodation for Light of the Ocean resort.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1002, 'Bungalow 2', 289, 101); --Accommodation for Light of the Ocean resort.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1003, 'Bungalow 3', 325, 101); --Accommodation for Light of the Ocean resort.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1004, 'Suite 101', 199, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1005, 'Suite 102', 199, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1006, 'Suite 201', 250, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1007, 'Suite 202', 250, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1008, 'Room 10', 150, 103); --Accommodation for Mountainous Mexico.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1009, 'Room 20', null, 103); --Accommodation for Mountainous Mexico.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1010, 'Cabin A', 300, 104); --Accommodation for Greater Lakes.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1011, 'Cabin B', null, 104); --Accommodation for Greater Lakes.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1012, 'Cabin C', 350, 104); --Accommodation for Greater Lakes.
```

Notice that we use the keyword “null” for the accommodations that have no price_per_night listed.

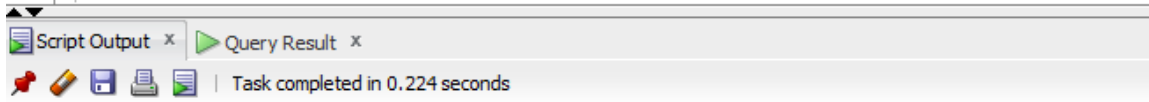
Below are the screenshots for running the inserts in each DBMS. Note that there are too many insert statements and results to fit on the screen, so we just capture what we can on one screen.



Screenshots: Inserting the Resort Values

Oracle

```
VALUES(1003, 'Bungalow 3', 325, 101); --Accommodation for Light of the Ocean resort.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1004, 'Suite 101', 199, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1005, 'Suite 102', 199, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1006, 'Suite 201', 250, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1007, 'Suite 202', 250, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1008, 'Room 10', 150, 103); --Accommodation for Mountainous Mexico.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1009, 'Room 20', null, 103); --Accommodation for Mountainous Mexico.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1010, 'Cabin A', 300, 104); --Accommodation for Greater Lakes.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1011, 'Cabin B', null, 104); --Accommodation for Greater Lakes.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1012, 'Cabin C', 350, 104); --Accommodation for Greater Lakes.
```



1 row inserted.

1 row inserted.

1 row inserted.

Microsoft SQL Server

```
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1004, 'Suite 101', 199, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1005, 'Suite 102', 199, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1006, 'Suite 201', 250, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1007, 'Suite 202', 250, 102); --Accommodation for Breathtaking Bahamas.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1008, 'Room 10', 150, 103); --Accommodation for Mountainous Mexico.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1009, 'Room 20', null, 103); --Accommodation for Mountainous Mexico.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1010, 'Cabin A', 300, 104); --Accommodation for Greater Lakes.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1011, 'Cabin B', null, 104); --Accommodation for Greater Lakes.
INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
VALUES(1012, 'Cabin C', 350, 104); --Accommodation for Greater Lakes.
```

0 %

Messages

(1 row(s) affected)
(1 row(s) affected)
(1 row(s) affected)
(1 row(s) affected)
(1 row(s) affected)
(1 row(s) affected)

PostgreSQL

```
51 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
52 VALUES(1005, 'Suite 102', 199, 102); --Accommodation for Breathtaking Bahamas.
53 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
54 VALUES(1006, 'Suite 201', 250, 102); --Accommodation for Breathtaking Bahamas.
55 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
56 VALUES(1007, 'Suite 202', 250, 102); --Accommodation for Breathtaking Bahamas.
57 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
58 VALUES(1008, 'Room 10', 150, 103); --Accommodation for Mountainous Mexico.
59 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
60 VALUES(1009, 'Room 20', null, 103); --Accommodation for Mountainous Mexico.
61 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
62 VALUES(1010, 'Cabin A', 300, 104); --Accommodation for Greater Lakes.
63 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
64 VALUES(1011, 'Cabin B', null, 104); --Accommodation for Greater Lakes.
65 INSERT INTO Accommodations(accommodations_id, description, cost_per_night, resort_id)
66 VALUES(1012, 'Cabin C', 350, 104); --Accommodation for Greater Lakes.
```

Data Output Explain Messages Notifications Query History

INSERT 0 1

Query returned successfully in 47 msec.

When completing this step, make sure to follow these same guidelines, to insert the data correctly and relationally using primary and foreign keys, and to add a movie series of your choosing to the content.

STEP 2

A video reseller needs to know how many movies are available. Write a single query to fulfill this request.

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, and we can use the command below to count the number of accommodations.

Code: Counting Accommodations

```
--Count the number of accommodations in the table.  
SELECT COUNT(*) AS nr_accommodations  
FROM Accommodations;
```

The "COUNT" construct is used in the column-list to instruct the DBMS to use the COUNT aggregate function. This is used in place of a column name, which is what you have seen put in that place in prior labs. The DBMS knows that COUNT is an aggregate function, so it knows to perform and return the single count, instead of returning every row along with the values in particular columns. The part inside the parentheses, "(*)", is a special syntax that instructs the DBMS to count the number of rows in the result set. Since all rows of the Accommodations table is in the result set, this results in the DBMS counting the number of rows in the Accommodations table. Last, the "AS nr_accommodations" syntax tells the DBMS to name the column "nr_accommodations" in the result set. Since "COUNT(*)" is not the name of a column, the DBMS would not know what to name the column in the result set otherwise. This is termed a *column alias*.

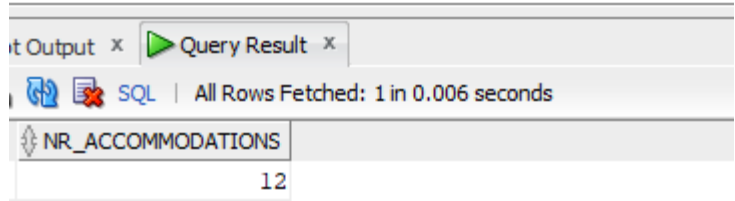
Here are the results from this command in each DBMS.



Screenshots: Counting the Accommodations


Oracle

```
--Count the number of accommodations in the table.
SELECT COUNT(*) AS nr_accommodations
FROM Accommodations;
```



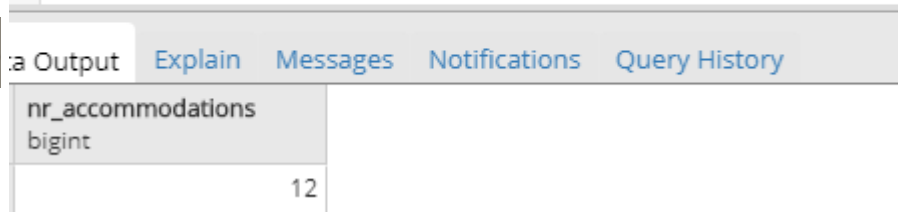
Microsoft SQL Server

```
--Count the number of accommodations in the table.
SELECT COUNT(*) AS nr_accommodations
FROM Accommodations;
```



PostgreSQL

```
--Count the number of accommodations in the table.
SELECT COUNT(*) AS nr_accommodations
FROM Accommodations;
```



Notice that the result in each DBMS is a single row, single column result with a value of 12. Even though there are 12 rows in the Accommodations table, the DBMS only returns one row since the aggregate function summarizes the result into one row.

In order to understand how this aggregate function is applied to this or any 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.

There is a nuance of the COUNT function you should be aware of, which is that you can use the name of a column instead of “*” when instructing COUNT what to count. When the name of a column is used, COUNT counts the number of rows where the value is not null for that column. For example, if we were count the “cost_per_night” column, the result would no longer be 12, because there are some null values in that column. The command would look like this.

Code: Counting cost_per_night

```
--Count the number of non-null costs in the table.  
SELECT COUNT(cost_per_night) AS nr_accommodations  
FROM Accommodations;
```

Each DBMS indicates that there are 10 rows where cost_per_night is not null, as illustrated below.

Screenshots: Counting the cost_per_night Column

Oracle

```
--Count the number of non-null costs in the table.  
SELECT COUNT(cost_per_night) AS nr_accommodations  
FROM Accommodations;
```

NR_ACCOMMODATIONS
10

Microsoft SQL Server

```
--Count the number of non-null costs in the table.  
SELECT COUNT(cost_per_night) AS nr_accommodations  
FROM Accommodations;
```

nr_accommodations
10

PostgreSQL

```
--Count the number of non-null costs in the table.  
SELECT COUNT(cost_per_night) AS nr_accommodations  
FROM Accommodations;
```


nr_accommodations
10

One other nuance you should be aware of is that you can also instruct COUNT to count distinct values instead of all values. For example, what if we want to know how many resorts the accommodations refer to? We could instruct the DBMS to count the distinct number of foreign keys to the resort table. Here is a command that does so.

Code: Distinct Counting

```
--Count the number of non-distinct and distinct foreign key references.
SELECT COUNT(resort_id) AS nr_non_distinct,
       COUNT(DISTINCT resort_id) AS nr_distinct
FROM Accommodations;
```

This command counts the resort_id column twice, the first counting all values, and the second only counting distinct values. Let's look at what each DBMS gives us.



Screenshots: Distinct Counting

Oracle

```
--Count the number of non-distinct and distinct foreign key references.
SELECT COUNT(resort_id) AS nr_non_distinct,
       COUNT(DISTINCT resort_id) AS nr_distinct
FROM Accommodations;
```

NR_NON_DISTINCT	NR_DISTINCT
12	4

Microsoft SQL Server

```
--Count the number of non-distinct and distinct foreign key references.
SELECT COUNT(resort_id) AS nr_non_distinct,
       COUNT(DISTINCT resort_id) AS nr_distinct
FROM Accommodations;
```

nr_non_distinct	nr_distinct
12	4

PostgreSQL

```
--Count the number of non-distinct and distinct foreign key references.
SELECT COUNT(resort_id) AS nr_non_distinct,
       COUNT(DISTINCT resort_id) AS nr_distinct
FROM Accommodations;
```

nr_non_distinct	nr_distinct
12	4

Notice that the counting the number of resort_id values without the DISTINCT keyword gets us 12, since there are 12 rows and no resort_id values are null. Counting the column with the DISTINCT keyword gets us 4, since there are only four unique values for this column in the Accommodations table (101, 102, 103, and 104).

As you have seen, COUNT allows you to:

- count the number of rows in a result set by using * in the parentheses.
- count the number of non-null values in a result set by using a column name in the parentheses.
- count the number of distinct non-null values in a result set by using the DISTINCT keyword and a column name in the parentheses.

Which of these you need depends upon the particular use case. Armed with this knowledge, you can now complete Step 2.

STEP 3

The same video reseller needs to know the price of the most expensive and least expensive series. Write two queries that fulfill this request, and also explain how and why the SQL processor treated the suggested price for the Lord of the Rings series differently than the other suggested price values.


To learn how to address this step, imagine something similar with the resort schema. A potential resort customer is budgeting for an upcoming vacation, and so telephones a request for the lowest price of all of the accommodations. To obtain this information for the customer we will need to use the MIN aggregate function, as follows.

Code: Obtaining the Smallest Price

```
--Obtain the least expensive price.  
SELECT MIN(cost_per_night) AS least_expensive  
FROM Accommodations;
```

The MIN function returns the value that is the lowest for the column indicated in parentheses. In this case, we used the function on a number (cost_per_night), which is a typical use of MIN. However, we can also use the MIN function on date columns and even on text columns.

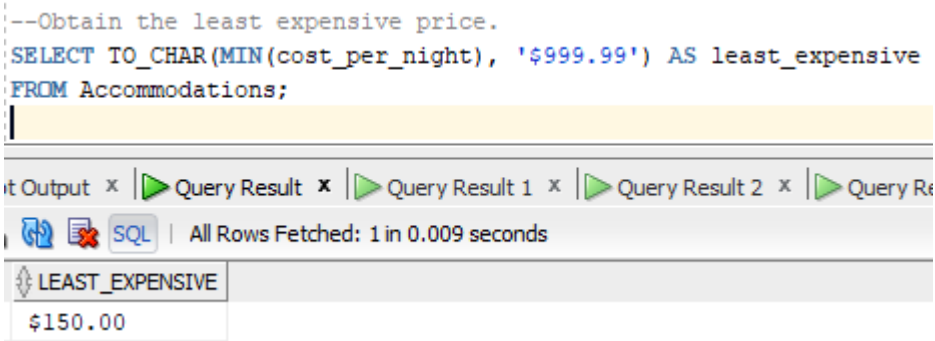
Below are screenshots of doing this in each DBMS. Note that in the screenshots, we additionally format the value as a currency for optimal human-readable representation.



Screenshots: Obtaining the Smallest Price

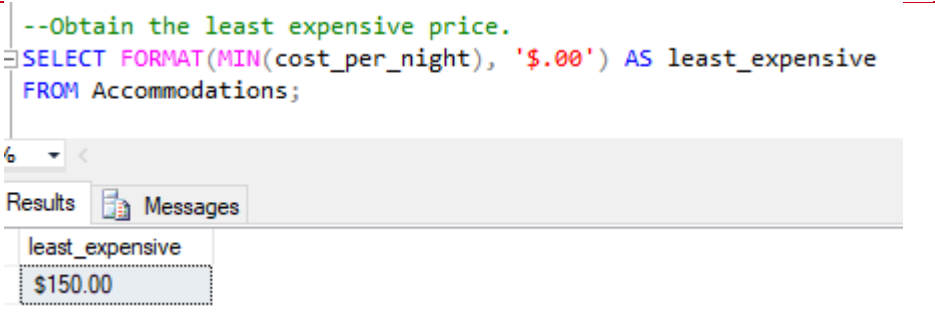
Oracle

```
--Obtain the least expensive price.  
SELECT TO_CHAR(MIN(cost_per_night), '$999.99') AS least_expensive  
FROM Accommodations;
```



Microsoft SQL Server

```
--Obtain the least expensive price.  
SELECT FORMAT(MIN(cost_per_night), '$.00') AS least_expensive  
FROM Accommodations;
```



PostgreSQL

--Obtain the least expensive price.
SELECT TO_CHAR(MIN(cost_per_night), '\$999.99') AS least_expensive
FROM Accommodations;

Query Output

Explain

Messages

Notifications

Query History

least_expensive
text
\$ 150.00

All three DBMS indicate that \$150.00 is the lowest price available.

Accommodations that have no cost per night are not taken into account, because the MIN function thus ignores null values. One major reason why many are concerned with the presence of null values in a table is because there is no one-size-fits-all solution for how they are treated with aggregate functions. Some aggregate functions ignore null values; some order null values at the beginning or end of the result set; some allow the query author to decide how nulls are treated; and all of this can vary between different DBMS. Make sure to use aggregate functions correctly and appropriately when you are using them on a nullable column.

There is another function that would allow us to obtain the most expensive accommodation rather than the least expensive. You can quickly find that with your own research, to address both queries needed for Step 3.

STEP 4

A film production company is considering purchasing the rights to extend a series, and needs to know the names and prices of all movie series, along with the number of movies in each series. Write a single query to fulfill this request.

This use case is more complex than the prior ones, but don't worry, you will be able address it with the proper knowledge. You have learned how to use some aggregate functions to retrieve a single value, but this use case is asking for one value per movie series (that is, the number of movies per series). How is this done? With grouping! When grouping is used, rows in the result set are grouped according to distinct values, and then the aggregate function is executed over each group of rows.

An example will help clarify the preceding paragraph. In step #3, our base query was:

Code: Obtaining the Smallest Price (No Group)

```
--Obtain the least expensive price.  
SELECT MIN(cost_per_night) AS least_expensive  
FROM Accommodations;
```

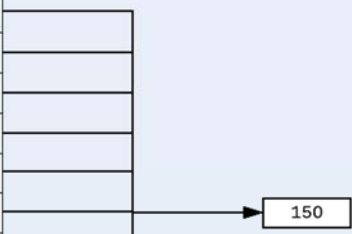
Conceptually what is happening in the SQL engine with this aggregate function is this:

MIN with No Group

Accommodations

accommodations_id	resort_id	description	cost_per_night
1001	101	Bungalow 1	289
1002	101	Bungalow 2	289
1003	101	Bungalow 3	325
1004	102	Suite 101	199
1005	102	Suite 102	199
1006	102	Suite 201	250
1007	102	Suite 202	250
1008	103	Room 10	150
1009	103	Room 20	
1010	104	Cabin A	300
1011	104	Cabin B	
1012	104	Cabin C	350

SELECT MIN(cost_per_night) AS least_expensive
FROM Accommodations;



As illustrated, the MIN function considers every row in the Accommodations table, then yields a single result of 150, since that is the lowest value.

However, what if we want to know the least expensive accommodation per resort? A single value will not suffice. We want one value per resort, and this requires grouping. We can fulfill this request with the following SQL.

Code: Obtaining the Smallest Price Per Resort

```
--Obtain the least expensive price per resort.  
SELECT resort_id, MIN(cost_per_night) AS least_expensive  
FROM Accommodations  
GROUP BY resort_id;
```

Notice the GROUP BY keywords followed by the “resort_id” column, which indicates to the SQL engine that any aggregate function present (in this case, MIN), is to run *once per unique resort_id*, rather than once for the entire result set. Although not absolutely required, we also add the resort_id column into the column-list so that we can tell which resort_id is associated with which minimum price.

Conceptually, the execution now looks as follows.

MIN Grouped by resort_id

Full SQL

```
SELECT resort_id, MIN(cost_per_night) AS least_expensive  
FROM Accommodations  
GROUP BY resort_id;
```

Step 1
(Grouping)

Accommodations

accomodations_id	resort_id	description	cost_per_night
1001	101	Bungalow 1	289
1002	101	Bungalow 2	289
1003	101	Bungalow 3	325
1004	102	Suite 101	199
1005	102	Suite 102	199
1006	102	Suite 201	250
1007	102	Suite 202	250
1008	103	Room 10	150
1009	103	Room 20	
1010	104	Cabin A	300
1011	104	Cabin B	
1012	104	Cabin C	350

accomodations_id	resort_id	description	cost_per_night
1001	101	Bungalow 1	289
1002	101	Bungalow 2	289
1003	101	Bungalow 3	325

accomodations_id	resort_id	description	cost_per_night
1004	102	Suite 101	199
1005	102	Suite 102	199
1006	102	Suite 201	250
1007	102	Suite 202	250

accomodations_id	resort_id	description	cost_per_night
1008	103	Room 10	150
1009	103	Room 20	

accomodations_id	resort_id	description	cost_per_night
1010	104	Cabin A	300
1011	104	Cabin B	
1012	104	Cabin C	350

accomodations_id	resort_id	description	cost_per_night
1001	101	Bungalow 1	289
1002	101	Bungalow 2	289
1003	101	Bungalow 3	325

289

accomodations_id	resort_id	description	cost_per_night
1004	102	Suite 101	199
1005	102	Suite 102	199
1006	102	Suite 201	250
1007	102	Suite 202	250

199

accomodations_id	resort_id	description	cost_per_night
1008	103	Room 10	150
1009	103	Room 20	


150

accomodations_id	resort_id	description	cost_per_night
1010	104	Cabin A	300
1011	104	Cabin B	
1012	104	Cabin C	350

300

Step 2
(Min Per Group)

In the first step, the Accommodations table is divided into groups based upon the resort_id. In the second step, the minimum cost_per_night is obtained for each of the groups. In this case, we obtain four minimums, one for each resort, instead of just one. To prove that it works this way, let's look at the screenshots of executing this command below.



Screenshots: Least Expensive Accommodation Per Resort

Oracle

```
--Obtain the least expensive price per resort.
SELECT resort_id, MIN(cost_per_night) AS least_expensive
FROM Accommodations
GROUP BY resort_id;
```

RESORT_ID	LEAST_EXPENSIVE
1	102
2	101
3	104
4	103

Microsoft SQL Server

```
--Obtain the least expensive price per resort.
SELECT resort_id, MIN(cost_per_night) AS least_expensive
FROM Accommodations
GROUP BY resort_id;
```

resort_id	least_expensive
101	289.00
102	199.00
103	150.00
104	300.00

PostgreSQL

```
--Obtain the least expensive price per resort.
SELECT resort_id, MIN(cost_per_night) AS least_expensive
FROM Accommodations
GROUP BY resort_id;
```

resort_id	least_expensive
104	300.00
103	150.00
102	199.00
101	289.00

As expected, each database returns the same rows – (101,289), (102,199), (103,150), and (104,300) – albeit in a different order (ordering aggregate results is a topic for a different step). Now that you see the result sets explicitly, you can see why we need the resort_id in addition to the aggregate result. Without it, there would be no way for us to determine which value goes with which resort. It is common practice (but not required) to

put all of the columns in the GROUP BY clause into the column-list for the SELECT clause, for this reason, so that it can be determined which aggregate value goes with which group.

Technically speaking, an aggregate function always runs on a group, even if no group is specified, because the default group is the entire result set. When no GROUP BY clause is used, the default group of the entire result set is used. We use the GROUP BY construct to explicitly control the groups over which aggregate functions execute. An alternative way to think about this concept is that an aggregate function always executes over groups, and that the omission of the GROUP BY statement means that there is only one group, which is the entire result set.

You may have noticed something missing from the last query we executed. The query gets us the least expensive prices per resort, but only gives us the resort_id. The resort_id is not helpful to a customer or employee viewing this information; what would 103 mean to them for example? Rather, they want to see the name of the resort, such as “Mountainous Mexico”. How do we do this in a query? As you may have guessed, we need to combine aggregating by group with a join across tables, requiring us to combine Lab 2 and Lab 3 concepts. Take a look at the below query which does so.

Code: Obtaining Least Expensive Accommodation with Name

```
--Obtain the least expensive price per resort, with resort name.  
SELECT name, MIN(cost_per_night) AS least_expensive  
FROM   Accommodations  
JOIN    Resort ON Resort.resort_id = Accommodations.resort_id  
GROUP BY Resort.resort_id, name;
```

In this query, the MIN aggregate function is still present, but you’ll notice a join in the FROM clause so that the resort name can be obtained. We select the name instead of the id for reasons described above. Further, you’ll notice that the grouping happens both by the resort_id column as well as the name column. Why? Even though there is one resort name per id, the SQL engine has the requirement that any column that appears in the column-list (that is not an aggregate function) must be part of the grouping. The SQL engine is not designed to understand that the name goes along with the resort (which would require a significant heuristic, statistical, and possibly even A.I. component), so it relies on the more rudimentary rule of requiring all selected columns to be part of the grouping.

Below are the screenshots of executing this query.



Screenshots: Least Expensive Accommodation With Name

Oracle

```
SELECT name, MIN(cost_per_night) AS least_expensive
FROM Accommodations
JOIN Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, name;
```

Query Result x | Script Output x | Query Result 1 x | Query Result 2 x

SQL | All Rows Fetched: 4 in 0.022 seconds

NAME	LEAST_EXPENSIVE
1 Breathtaking Bahamas	199
2 Mountainous Mexico	150
3 Light of the Ocean	289
4 Greater Lakes	300

Microsoft
SQL Server

```
--Obtain the least expensive price per resort, with resort name.
SELECT name, MIN(cost_per_night) AS least_expensive
FROM Accommodations
JOIN Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, name;
```

Results Messages

name	least_expensive
Light of the Ocean	289.00
Breathtaking Bahamas	199.00
Mountainous Mexico	150.00
Greater Lakes	300.00

PostgreSQL

```
--Obtain the least expensive price per resort, with resort name.
SELECT name, MIN(cost_per_night) AS least_expensive
FROM Accommodations
JOIN Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, name;
```

Script Output Explain Messages Notifications Query History

name	least_expensive
Greater Lakes	300.00
Mountainous Mexico	150.00
Breathtaking Bahamas	199.00
Light of the Ocean	289.00

The results of this query are more useful to a customer or employee when compared to the query that returns only the resort_id.

You have now seen how to execute an aggregate function over groups of rows rather than the entire result set, and how to combine aggregation with joins to handle more complex use cases. You can now address Step #4.

STEP 5

The same film production company wants to limit its search to genres that have at least 7 associated movies. Write a single query to fulfill this request, making sure to list only genres that have at least 7 associated movies, along with the number of movies for the genre.

You have already learned how to do much of what this step requires – aggregating across groups, and combining that with joins. Using those skills, you can obtain the number of movies per genre, and list any relevant fields. However, since the request is asking you to limit by the number of movies, which requires that you use aggregation, you can probably perceive that one more skill is needed, which is the ability to limit the result set by aggregated values (rather than values coming directly from a table).

Thankfully, the HAVING clause in SQL is provided just for the purpose. While the WHERE clause limits the result set based upon values in individual rows, the HAVING clause limits the result set based upon aggregate results. We use an aggregate function in place of a column name, but otherwise use Boolean expressions just as in the WHERE clause.

For an example in our Resort schema, what if instead of asking for only the least expensive accommodation per resort, we added the condition to only show resorts that had an accommodation less than \$225? The request would then look as follows:

List the least expensive price of accommodations per resort, but only for resorts with accommodations costing \$225 or less.

We can fulfill this request with the following SQL:

Code: Cheapest Price Per Resort Less Than \$225

```
--Obtain the least expensive price per resort, less than $225.  
SELECT name, MIN(cost_per_night) AS least_expensive  
FROM   Accommodations  
JOIN    Resort ON Resort.resort_id = Accommodations.resort_id  
GROUP BY Resort.resort_id, name  
HAVING MIN(cost_per_night) < 225;
```

We added the HAVING clause to limit the results by the results of the MIN function. By using "< 225" we are limiting the results to those having a minimum which is less than \$225 per night.

The screenshots are recorded below.



Screenshots: Cheapest Price Per Resort Less Than \$225

Oracle

```
--Obtain the least expensive price per resort, less then $225.  
SELECT name, MIN(cost_per_night) AS least_expensive  
FROM Accommodations  
JOIN Resort ON Resort.resort_id = Accommodations.resort_id  
GROUP BY Resort.resort_id, name  
HAVING MIN(cost_per_night) < 225;
```

	NAME	LEAST_EXPENSIVE
1	Breathtaking Bahamas	199
2	Mountainous Mexico	150

Microsoft
SQL Server

```
--Obtain the least expensive price per resort, less then $225.  
SELECT name, MIN(cost_per_night) AS least_expensive  
FROM Accommodations  
JOIN Resort ON Resort.resort_id = Accommodations.resort_id  
GROUP BY Resort.resort_id, name  
HAVING MIN(cost_per_night) < 225;
```

name	least_expensive
Breathtaking Bahamas	199.00
Mountainous Mexico	150.00

PostgreSQL

```
--Obtain the least expensive price per resort, less then $225.  
SELECT name, MIN(cost_per_night) AS least_expensive  
FROM Accommodations  
JOIN Resort ON Resort.resort_id = Accommodations.resort_id  
GROUP BY Resort.resort_id, name  
HAVING MIN(cost_per_night) < 225;
```

name	least_expensive
Mountainous Mexico	150.00
Breathtaking Bahamas	199.00

Notice that only the resorts with an accommodation less than \$225 are listed, and the rest are excluded.

You have now seen all of the constructs you need to address Step #5.

STEP 6

Boston University wants to offer its students a movie-binge weekend by playing every movie in a series. To make sure the series is as bingeable as possible, BU wants to be sure the series will run for at least 10 hours. Write a single query that gives this information, with useful columns.

You've learned how to count up values in SQL, as well as how to get the largest and smallest values, but you have not yet learned how to add up values. SQL offers a SUM aggregate function that does this, adding up values to arrive at a total. To demonstrate how to use the function, here is another use case.

A travel corporation would like to purchase a resort that can offer them a lot of revenue. To this end, they would like to see the names of resorts where the total rental revenue is at least \$800 per night, along with the revenue it offers.

For this use case, we need to add up the `cost_per_night` values for all accommodations a resort has, to determine what kind of revenue the resort would bring. Counting the values, or getting the most expensive or least expensive accommodations, will not get us what we need, but the following SQL will.

Code: Resorts Bringing at Least \$800 Revenue Per Night

```
--Show resorts that bring in at least $800 revenue per night.
SELECT name, SUM(cost_per_night) AS total_revenue
FROM   Accommodations
JOIN   Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, Resort.name
HAVING SUM(cost_per_night) >= 800;
```

We obtain the name of the resort, along with the sum of all accommodations the resort offers by using the SUM aggregate function. We further restrict the results to the resorts that total at least \$800 by using a HAVING clause. Each DBMS returns results are shown below.



Screenshots: Resorts Bringing at Least \$800 Revenue Per Night

Oracle

```
--Show resorts that bring in at least $800 revenue per night.
SELECT name, SUM(cost_per_night) AS total_revenue
FROM Accommodations
JOIN Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, Resort.name
HAVING SUM(cost_per_night) >= 800;
```

cs669: SELECT name, SUM(cost_per_night) AS total_revenue FROM Ac

Script Output x Query Result x Query Result 1 x

SQL | All Rows Fetched: 2 in 0.074 seconds

	NAME	TOTAL_REVENUE
1	Breathtaking Bahamas	898
2	Light of the Ocean	903

Microsoft
SQL Server

```
--Show resorts that bring in at least $800 revenue per night.
SELECT name, SUM(cost_per_night) AS total_revenue
FROM Accommodations
JOIN Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, Resort.name
HAVING Sum(cost_per_night) >= 800;
```

results Messages

	name	total_revenue
	Light of the Ocean	903.00
	Breathtaking Bahamas	898.00

PostgreSQL

```
--Show resorts that bring in at least $800 revenue per night.
SELECT name, SUM(cost_per_night) AS total_revenue
FROM Accommodations
JOIN Resort ON Resort.resort_id = Accommodations.resort_id
GROUP BY Resort.resort_id, Resort.name
HAVING SUM(cost_per_night) >= 800;
```

ta Output Explain Messages Notifications Query History

	name	total_revenue
	Breathtaking Bahamas	898.00
	Light of the Ocean	903.00

The Light of the Ocean resort brings a total revenue of \$903, while Breathtaking Bahamas brings \$898. The other two resorts do not bring in at least \$800 in revenue and so are excluded from the results.

In this use case, we had to discern what columns would be the most useful to the travel corporation. They did not specifically ask for the resort name, resort id, or the SUM. We needed to discern what they would want based upon their request. This kind of interpretation is common for database developers, whereby they need to turn a non-technical request into a technical specification where the precise logic and columns are known. Note that although not reflected in the results, formatting the revenue as a currency would be useful.

You now have enough knowledge to complete Step 6.

STEP 7

A research institution requests the names of all movie series' creators, as well as the number of "Family Film" movies they have created (even if they created none). The institution wants the list to be ordered from most to least; the creator who created the most family films will be at the top of the list, and the one with the least will be at the bottom. Write a single query that gives this information, with useful columns.

Here you may recognize aspects of the request that you have already learned to address. You know how to get a count of the number of movies per series creator. You know how to order result sets. You know how to limit the result set based upon some criteria. You know how to list out all movie creators whether or not they are related to other tables with the given criteria. The focus here is integrating all of the different components to correctly address this use case.

Let's look at an example with our resort schema that requires this kind of integration.

Management wants to see every type of resort, and for each type, the number of accommodations that cost at least \$200, ordered by the types with the least number of accommodations to the most.

Let's break down some of the components we need to use to satisfy this request.


Request Fragment	SQL Construct
Management wants to see every type of resort...	This suggests use of a left or right join, since they want to see every type of resort whether or not it matches the other criteria.
...and for each type, the number of accommodations...	This suggests use of a join to obtain the accommodations themselves, combined with the COUNT aggregate function to count them. Given our analysis of the previous fragment, we would use a left or right join so we don't inadvertently eliminate resort types that have no accommodations matching the criteria.
...that cost at least \$200...	Because this deals with restricting individual rows by the cost of the accommodation, this suggests adding a Boolean expression to limit the accommodations to those costing more than \$200. Since we're using a left or right join, we will likely add that restriction to the join so that we don't inadvertently eliminate resort types that have no accommodations costing at least \$200.
...ordered by the types with the least number of accommodations to the most.	This suggests use of an ORDER BY on the column that has the count in it. Since it is from the least to the most we can use the default ordering (ascending). You have not yet seen use of an ORDER BY on an aggregate column, but this will be explained below.

First, we give some SQL code that gives us an intermediate result that doesn't aggregate, but does yield all resort types along with all associated accommodations costing at least \$200. Doing this first allows us to see what rows will be aggregated.

Code: Intermediate Result with no Aggregation

```
--Show all accommodations along with their resorts and types, that cost more than $200.
SELECT      *
FROM        Resort
JOIN        Accommodations ON  Accommodations.resort_id = Resort.resort_id
                                AND Accommodations.cost_per_night >= 200
RIGHT JOIN  Resort_type ON  Resort_type.resort_type_id = Resort.resort_type_id;
```

Running this in each DBMS gives results as follows.



Screenshots: Intermediate Result with no Aggregation

Oracle

```
--Show all accommodations along with their resorts and types, that cost more than $200.
SELECT      *
FROM        Resort
JOIN        Accommodations ON  Accommodations.resort_id = Resort.resort_id
                                AND Accommodations.cost_per_night >= 200
RIGHT JOIN  Resort_type ON  Resort_type.resort_type_id = Resort.resort_type_id;
```

Query Result 1 x

Script Output x

Query Result 2 x

SQL

All Rows Fetched: 9 in 0.017 seconds

	RESORT_ID	RESORT_TYPE_ID	NAME	ACCOMMODATIONS_ID	RESORT_ID_1	DESCRIPTION	COST_PER_NIGHT	RESORT_TYPE_ID_1	RESORT_TYPE
1	101	1	Light of the Ocean	1001	101	Bungalow 1	289	1	Ocean
2	101	1	Light of the Ocean	1002	101	Bungalow 2	289	1	Ocean
3	101	1	Light of the Ocean	1003	101	Bungalow 3	325	1	Ocean
4	102	1	Breathtaking Bahamas	1006	102	Suite 201	250	1	Ocean
5	102	1	Breathtaking Bahamas	1007	102	Suite 202	250	1	Ocean
6	104	2	Greater Lakes	1010	104	Cabin A	300	2	Lakeside
7	104	2	Greater Lakes	1012	104	Cabin C	350	2	Lakeside
8	(null)	(null)	(null)	(null)	(null)	(null)	(null)	4	Country
9	(null)	(null)	(null)	(null)	(null)	(null)	(null)	3	Mountaintop

Microsoft SQL Server

```
--Show all accommodations along with their resorts and types, that cost more than $200.
SELECT      *
FROM        Resort
JOIN        Accommodations ON  Accommodations.resort_id = Resort.resort_id
                                AND Accommodations.cost_per_night >= 200
RIGHT JOIN  Resort_type ON  Resort_type.resort_type_id = Resort.resort_type_id
```

Results

Messages

	resort_id	resort_type_id	name	accommodations_id	resort_id	description	cost_per_night	resort_type_id	resort_type
1	101	1	Light of the Ocean	1001	101	Bungalow 1	289.00	1	Ocean
2	101	1	Light of the Ocean	1002	101	Bungalow 2	289.00	1	Ocean
3	101	1	Light of the Ocean	1003	101	Bungalow 3	325.00	1	Ocean
4	102	1	Breathtaking Bahamas	1006	102	Suite 201	250.00	1	Ocean
5	102	1	Breathtaking Bahamas	1007	102	Suite 202	250.00	1	Ocean
6	104	2	Greater Lakes	1010	104	Cabin A	300.00	2	Lakeside
7	104	2	Greater Lakes	1012	104	Cabin C	350.00	2	Lakeside
8	NULL	NULL	NULL	NULL	NULL	NULL	NULL	3	Mountaintop
9	NULL	NULL	NULL	NULL	NULL	NULL	NULL	4	Country

PostgreSQL

--Show all accommodations along with their resorts and types, that cost more than \$200.

```
SELECT *
FROM Resort
JOIN Accommodations ON Accommodations.resort_id = Resort.resort_id
AND Accommodations.cost_per_night >= 200
RIGHT JOIN Resort_type ON Resort_type.resort_type_id = Resort.resort_type_id;
```

a Output Explain Messages Notifications Query History

resort_id numeric (12)	resort_type_id numeric (12)	name character varying (255)	accommodations_id numeric (12)	resort_id numeric (12)	description character varying (255)	cost_per_night numeric (8,2)	resort_type_id numeric (12)	resort_type character varying (64)
102	1	Breathtaking Bahamas	1006	102	Suite 201	250.00	1	Ocean
102	1	Breathtaking Bahamas	1007	102	Suite 202	250.00	1	Ocean
101	1	Light of the Ocean	1001	101	Bungalow 1	289.00	1	Ocean
101	1	Light of the Ocean	1002	101	Bungalow 2	289.00	1	Ocean
101	1	Light of the Ocean	1003	101	Bungalow 3	325.00	1	Ocean
104	2	Greater Lakes	1010	104	Cabin A	300.00	2	Lakeside
104	2	Greater Lakes	1012	104	Cabin C	350.00	2	Lakeside
[null]	[null]	[null]	[null]	[null]	[null]	[null]	3	Mountaintop
[null]	[null]	[null]	[null]	[null]	[null]	[null]	4	Country

Notice that with these results, we see multiple accommodations associated with Ocean and Lakeside that cost more than \$200; however, neither the Mountaintop or Country types have any accommodations that cost over \$200. For Mountaintop, it does have some accommodations, just none costing more than \$200. For Country, it actually has no accommodations at all. Either way, the right join ensures that these resort types are still retrieved, and the rest of the columns are null, since there are no accommodations matching the criteria.

Now that you witnessed intermediate results without aggregation, let us go ahead and look at a final version of the SQL to fully address this use case.

Code: All Resort Types along with Number Over 200

```
--Show all resort types and the number of accommodations costing over $200, ordered least to greatest.
SELECT Resort_type.resort_type, COUNT(Accommodations.accommodations_id) AS nr_over_200
FROM Resort
JOIN Accommodations ON Accommodations.resort_id = Resort.resort_id
AND Accommodations.cost_per_night >= 200
RIGHT JOIN Resort_type ON Resort_type.resort_type_id = Resort.resort_type_id
GROUP BY Resort_type.resort_type_id, Resort_type.resort_type
ORDER BY nr_over_200;
```

Below is an explanation of how this final version of the SQL ties into the use case.

Request Fragment	SQL Code	Explanation
Management wants to see every type of resort...	RIGHT JOIN Resort_type ON Resort_type.resort_type_id = Resort.resort_type_id	Right joining to the Resort_type table ensures that all types are listed whether or not they have any matching accommodations.
...and for each type, the number of accommodations...	SELECT ... COUNT(Accommodations.accommodations_id) AS nr_over_200 ... FROM Resort JOIN Accommodations ON Accommodations.resort_id = Resort.resort_id ... GROUP BY Resort_type.resort_type_id, Resort_type.resort_type	The results are grouped by resort type, and then counted, to indicate the number of accommodations matching the criteria.

		Since COUNT only counts non-null accommodation_id values, it ignores any rows where it is null. The resort and accommodations tables are joined in so that the accommodations over \$200 can be counted.
...that cost at least \$200...	<code>AND Accommodations.cost_per_night >= 200</code>	This Boolean expression limits accommodations those costing at least \$200. This condition is included as part of the join condition, rather than the WHERE clause, so that it does not inadvertently override the intent of the right join by eliminating types with no accommodations.
...ordered by the types with the least number of accommodations to the most.	<code>ORDER BY nr_over_200;</code>	The ORDER BY orders from least to greatest by default. Notice that we simply use the aliased named of the aggregated column as the target of ordering.

The ORDER BY deserves further explanation, in terms of conceptually when it runs. Conceptually, this SQL statement is executing in this order:

1. Execute the FROM and JOIN statements to retrieve the individual rows in the result set (resort types, resorts, and accommodations costing over \$200).
2. Execute the aggregation by grouping by resort type and counting the accommodations.
3. Execute the ordering by ordering by the aggregated column (number of accommodations).

As you can see, conceptually the ORDER BY executes last, after individual rows have been retrieved, and after aggregation has occurred. As always, modern DBMS guarantee the results are as if the above step ordering occurs, but may parallelize, interleave and re-order them to optimize performance.

You may find it helpful to tackle Step #7 in the same way as demonstrated, to first create an intermediate query that retrieves the non-aggregated rows, then create the final version that performs aggregation. This allows you to see what rows you are actually dealing with before they are aggregated, so you can debug any issues with the individual rows before proceeding to aggregation and ordering.