

OBJECTIVE

The objective of this lab is to provide you with a practical exercise of creating and using related data using the Structured Query Language (SQL). Together, we will learn how to:

- enforce relationships between two tables using a FOREIGN KEY constraint
- add related data to related tables
- ask questions and answer them using SQL queries that relate 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 current 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

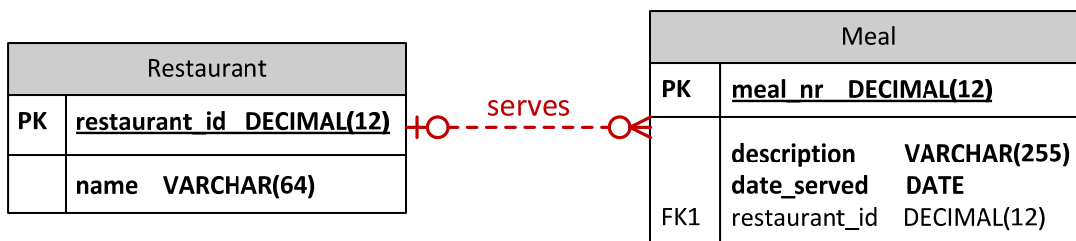
Use the submission template provided in the assignment inbox to complete this lab.

SECTION ONE

OVERVIEW

In Section One, we will work with the Restaurant and Meal tables illustrated below. The relationship between these two tables is described by the following business rules:

- A Restaurant may serve many Meals
- A Meal may be served at a Restaurant



Note that the bolded columns represent those with a NOT NULL constraint. Further note that restaurant_id in Meal is a foreign key referencing the Restaurant table. It is this foreign key that enforces the one-to-many relationship between Meal and Restaurant.

STEPS

1. Execute the following commands to create the Restaurant and Meal tables. Make sure to type the commands exactly as illustrated, including spaces, parentheses, and newlines.

```
CREATE TABLE Restaurant (  
  restaurant_id DECIMAL(12) PRIMARY KEY,  
  name VARCHAR(64) NOT NULL  
);  
  
CREATE TABLE Meal (  
  meal_nr DECIMAL(12) NOT NULL,  
  description VARCHAR(255) NOT NULL,  
  date_served DATE NOT NULL,  
  restaurant_id DECIMAL(12)  
);  
  
ALTER TABLE Meal  
ADD CONSTRAINT meal_pk  
PRIMARY KEY(meal_nr);
```

The structure of the first two commands are familiar to us from previous labs. We have not yet seen the ALTER TABLE command in previous labs, so let us explore what this command is accomplishing. An ALTER TABLE command lets us modify all aspects of the structure of a table. Any property we can specify in a CREATE TABLE statement, we can change using an ALTER TABLE statement. This command is quite useful because in production systems with live data, we cannot always drop and re-create an existing table in order to make changes to it. Instead, we use the ALTER TABLE statement.

The ALTER TABLE command adds a primary key constraint to the Meal table. Why is this needed? If you examine the CREATE TABLE for the Meal table, you'll notice that no primary key constraint is present. This was left out purposefully to show you another method of specifying primary key constraints, by using the ALTER TABLE command after the table is created. Because there are many kinds of changes we can make with an ALTER TABLE command, we used the words ADD CONSTRAINT keywords to indicate that we are specifically adding a constraint.

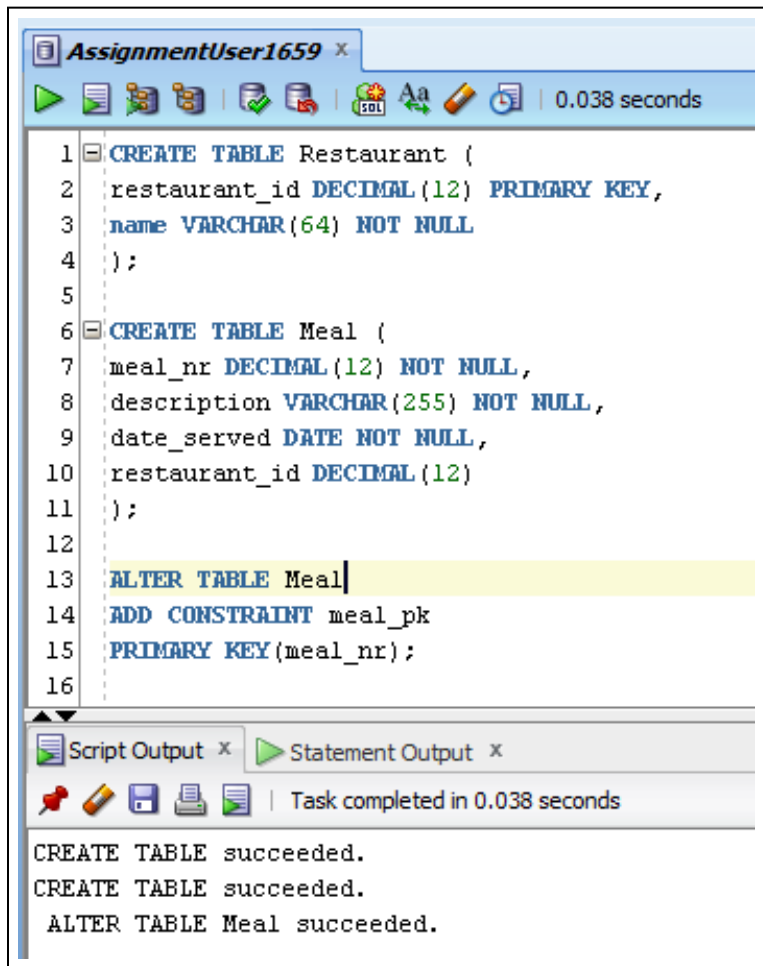
Just as we name tables with an identifier, we can name constraints. If we use the shorthand method of adding constraints by including them as part of a column definition in a CREATE TABLE statement, such as with the CREATE TABLE statement that creates the Restaurant table, the RDBMS generates a constraint name for us. This is known as a system-generated constraint name. So why would we want to name our constraints? System-generated names usually do not help us identify the table, column, or condition enforced by the constraint. If one of our SQL commands

violates a constraint that has a system-generated name, oftentimes we need to lookup the constraint to find the condition that has been violated. If we name our constraints, many times we will know the condition by the constraint's name, and can avoid the lookup.

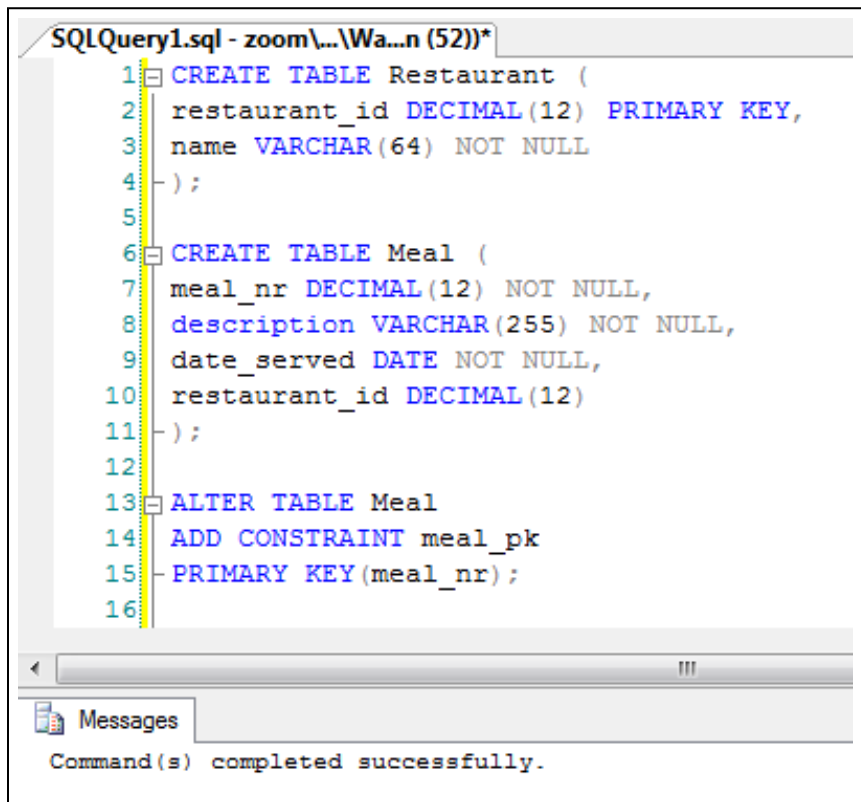
The word following the ADD CONSTRAINT keywords, "meal_pk" in this case, is the identifier that names our constraint. Though we could have used any legal identifier of our choosing, we used the name of the table, followed by "pk" as an acronym for "primary key", to indicate that the constraint is the primary key constraint for the Meal table. There are many conventions that can be used when naming constraints, and many organizations adopt their own conventions. One important aspect of any naming convention is consistency, so that the convention can be understood, and so that the reader is not required to guess at what the name of the constraint means.

The PRIMARY KEY keywords further indicate to the RDBMS that we are adding a PRIMARY KEY constraint. The syntax requires us to then enclose a comma-separated list of column names within parentheses. The columns specified in this list will all be covered by the new primary key constraint. In our example, we added only one column – meal_nr – to indicate that only the meal_nr column is covered by the primary key constraint.

2. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer.



Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.

A screenshot of the Microsoft SQL Server Management Studio (SSMS) interface. The main window displays a SQL query script in a text editor. The script consists of three statements: creating a 'Restaurant' table, creating a 'Meal' table, and adding a primary key constraint to the 'Meal' table. The 'Restaurant' table has columns 'restaurant_id' (DECIMAL(12), PRIMARY KEY) and 'name' (VARCHAR(64), NOT NULL). The 'Meal' table has columns 'meal_nr' (DECIMAL(12), NOT NULL), 'description' (VARCHAR(255), NOT NULL), 'date_served' (DATE, NOT NULL), and 'restaurant_id' (DECIMAL(12)). The third statement adds a primary key constraint named 'meal_pk' to the 'meal_nr' column. The bottom pane shows the 'Messages' tab with the text 'Command(s) completed successfully.'

```
SQLQuery1.sql - zoom\...\Wa...n (52))*
1 CREATE TABLE Restaurant (
2   restaurant_id DECIMAL(12) PRIMARY KEY,
3   name VARCHAR(64) NOT NULL
4 );
5
6 CREATE TABLE Meal (
7   meal_nr DECIMAL(12) NOT NULL,
8   description VARCHAR(255) NOT NULL,
9   date_served DATE NOT NULL,
10  restaurant_id DECIMAL(12)
11 );
12
13 ALTER TABLE Meal
14   ADD CONSTRAINT meal_pk
15   PRIMARY KEY(meal_nr);
16
```

Messages

Command(s) completed successfully.

3. We will now begin harnessing the most useful and powerful feature of a relational database – related data! Related data, and the ability to ask planned and unplanned questions about this data, are the in-demand features of RDBMSs today, and enable us to solve a wide variety of problems using an RDBMS.

The first step to relating data is setting up the structure of the tables to enforce the relationships we expect. We do this by creating a *foreign key* constraint. In our relational schema, the Meal table has a foreign key to the Restaurant table. Let us go ahead and create that foreign key constraint.

```
ALTER TABLE Meal
ADD CONSTRAINT meal_restaurant_fk
FOREIGN KEY(restaurant_id)
REFERENCES Restaurant(restaurant_id);
```

We have used another form of the ALTER TABLE command which we learned about in steps 1 and 2. The first two lines have the same format as the ALTER TABLE command in step 1, though we did use a different identifier, "meal_restaurant_fk", to indicate that the constraint defines a foreign key from the Meal table to the Restaurant table. The letters "fk" are an acronym to represent "foreign key".

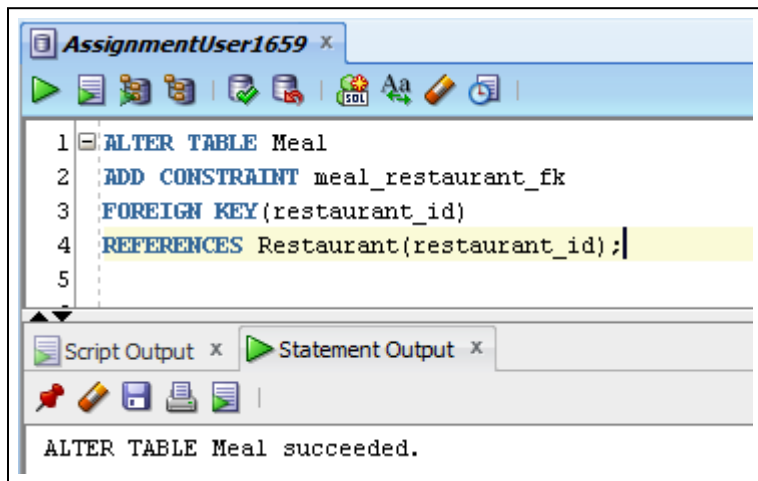
The next keywords, FOREIGN KEY, indicate to the RDBMS that the constraint we are adding is a foreign key constraint. The RDBMS expects a comma-separated list of column names enclosed in parentheses to follow the FOREIGN KEY keywords. In our case, only the restaurant_id column is covered by the foreign key constraint, and so we have identified only that column. Note that this list of columns identifies the columns within the table we are altering, in this case, the Meal table.

The next keyword, REFERENCES, indicates that what follows are the table and column identities that the foreign key will reference. The first following word, "Restaurant", indicates that the foreign key will reference the Restaurant table. Following this is a second comma-separated list of column names enclosed within parentheses, indicating that names of the columns that will be referenced in the referenced table, in this case, Restaurant.

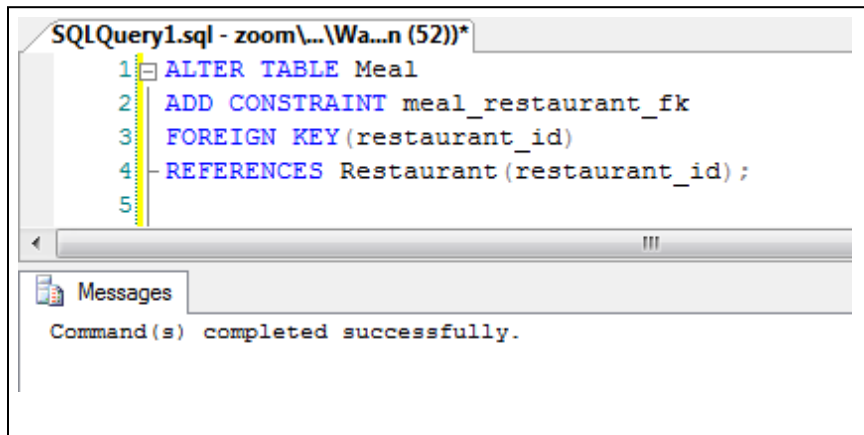
So in summary, the first comma-separated list identifies the columns within the table containing the foreign key constraint, while the second comma-separated list identifies the columns within the referenced table. . In our case, we only have one column identified in each table because our foreign key spans only one column. This is the common case.

If you work with databases long enough, you will run into a situation where a composite foreign key covering more than one column is in use. When defining the constraint for composite foreign keys, the first column identified in the first list references the first column in the second list, the second column identified in the first list references the second column in the second list, and so on.

4. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer.



Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



5. Now that we have enforced the relationship between Meal and Restaurant using a foreign key constraint, let us add in some related data using the already familiar INSERT INTO command.

```
INSERT INTO Restaurant (restaurant_id, name)
VALUES (31, 'Sunset Grill');
INSERT INTO Restaurant (restaurant_id, name)
VALUES (32, 'Oceanside Beachview');

INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
VALUES (101, 'Grilled eggplant with sides', CAST('03-Jul-2012' AS DATE), 31);
INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
VALUES (102, 'Delicious pizza with salad', CAST('09-Jul-2012' AS DATE), 31);
INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
VALUES (103, 'Five-course luxurious meal', CAST('13-Jul-2012' AS DATE), NULL);
```

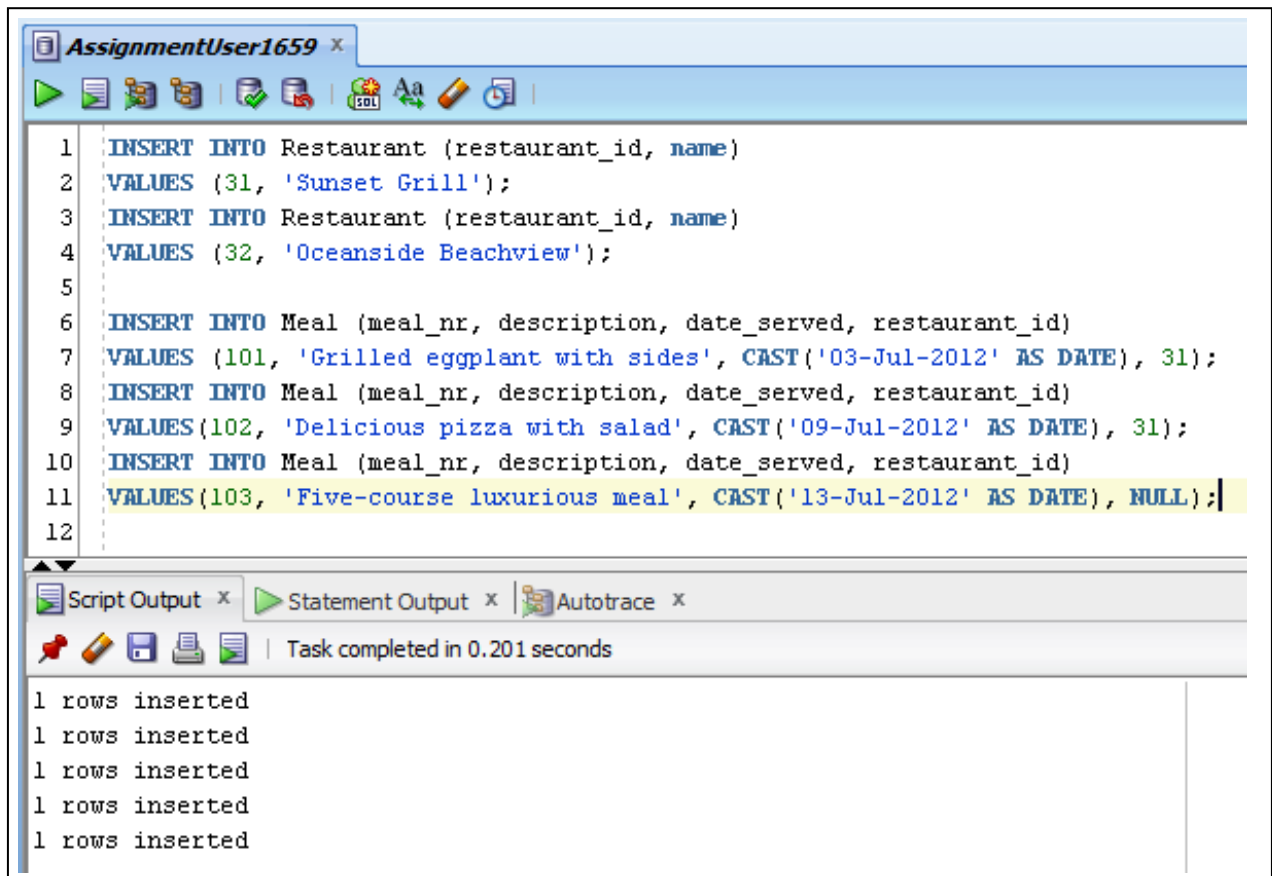
Let us examine these insertions to determine the relationships between the data. The first two insert commands simply insert two restaurants' IDs and names, and alone do not create any relationships; it is the insertions into the Meal table that create the relationships. How? The first three values in each Meal insertion are standard data, but the fourth value, the foreign key, is of interest. Notice that "31" is the ID for the first Sunset Grill Restaurant, and that the first two meals have "31" as their restaurant_id value. How should we interpret this? Simple! The first two meals were served at the Sunset Grill Restaurant.

Do you see how this works? To determine related rows, we are matching up the value in the foreign key column in the referencing table to the value in the referenced column in the referenced table, in our example, the restaurant_id values in the Meal table to the restaurant_id values in the Restaurant table. In short, the same value in two different rows indicates a relationship between those rows.

How should we interpret the NULL in the restaurant_id column for the third meal? Again, simple! The third meal has no indication of the restaurant that served it, so we know the meal was not served at the Sunset Grill Restaurant or the Oceanside Beachview Restaurant. Simply put, the restaurant that served the third meal is unknown.

We have determined the relationships between the Sunset Grill Restaurant and its meals, and have also determined that the third meal does not specify a restaurant, but what about the Oceanside Beachview Restaurant? Because no Meal rows have a foreign key value corresponding to Oceanside Beachview's primary key, 32, that restaurant has served no meals.

6. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer.



The screenshot displays the Oracle SQL Developer interface. The top toolbar includes icons for running, saving, and other database operations. The main editor window shows a SQL script with the following commands:

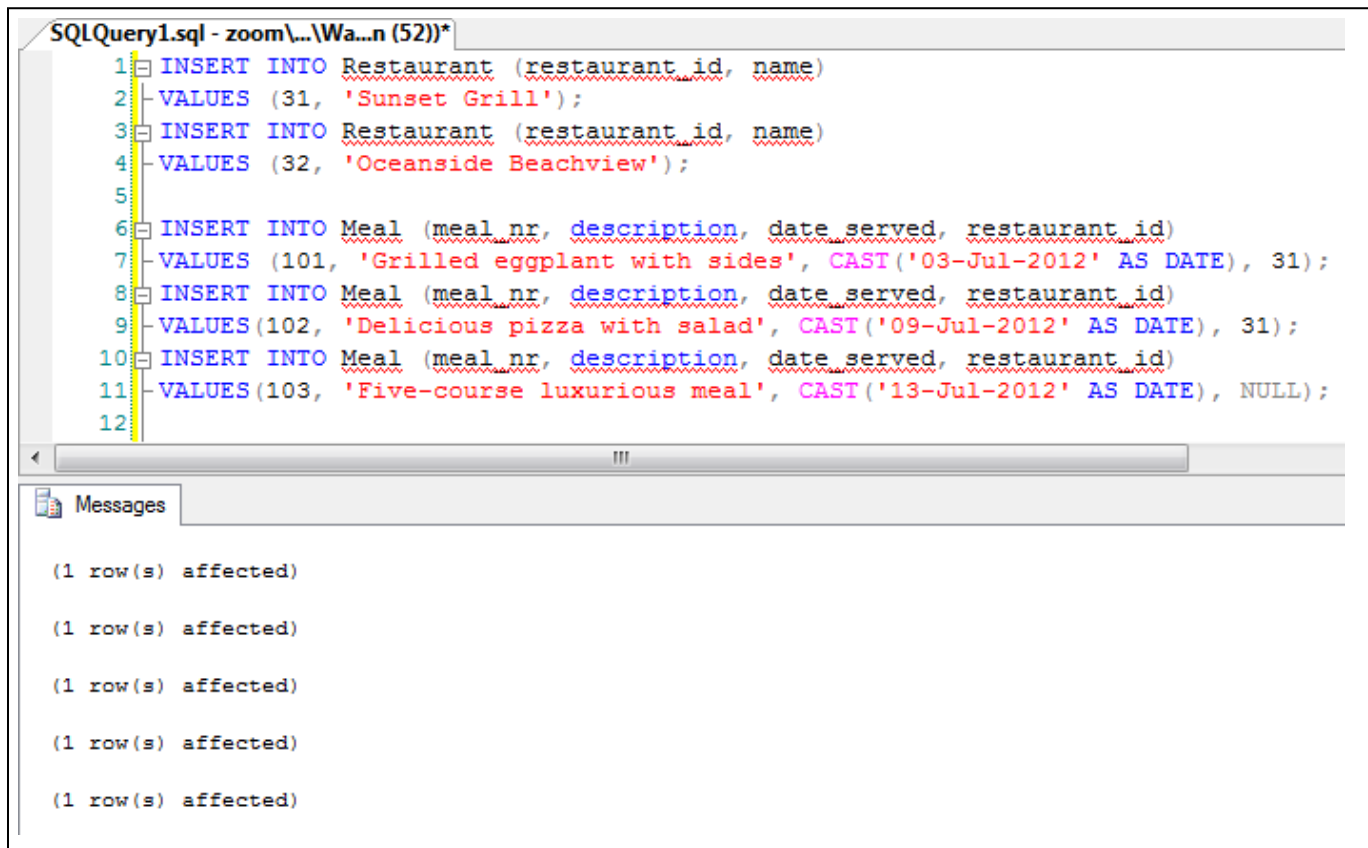
```
1 INSERT INTO Restaurant (restaurant_id, name)
2 VALUES (31, 'Sunset Grill');
3 INSERT INTO Restaurant (restaurant_id, name)
4 VALUES (32, 'Oceanside Beachview');
5
6 INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
7 VALUES (101, 'Grilled eggplant with sides', CAST('03-Jul-2012' AS DATE), 31);
8 INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
9 VALUES (102, 'Delicious pizza with salad', CAST('09-Jul-2012' AS DATE), 31);
10 INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
11 VALUES (103, 'Five-course luxurious meal', CAST('13-Jul-2012' AS DATE), NULL);
12
```

Below the editor, the 'Script Output' tab is active, showing the results of the execution:

```
1 rows inserted
1 rows inserted
1 rows inserted
1 rows inserted
1 rows inserted
```

The status bar at the bottom indicates 'Task completed in 0.201 seconds'.

Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



The screenshot displays the SQL Server Enterprise Edition interface. The top pane, titled 'SQLQuery1.sql - zoom\...\Wa...n (52))*', contains a SQL script with 12 lines of code. The script inserts data into the 'Restaurant' and 'Meal' tables. The bottom pane, titled 'Messages', shows the execution results for each statement.

```
1 INSERT INTO Restaurant (restaurant_id, name)
2 VALUES (31, 'Sunset Grill');
3 INSERT INTO Restaurant (restaurant_id, name)
4 VALUES (32, 'Oceanside Beachview');
5
6 INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
7 VALUES (101, 'Grilled eggplant with sides', CAST('03-Jul-2012' AS DATE), 31);
8 INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
9 VALUES (102, 'Delicious pizza with salad', CAST('09-Jul-2012' AS DATE), 31);
10 INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
11 VALUES (103, 'Five-course luxurious meal', CAST('13-Jul-2012' AS DATE), NULL);
12
```

Messages

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

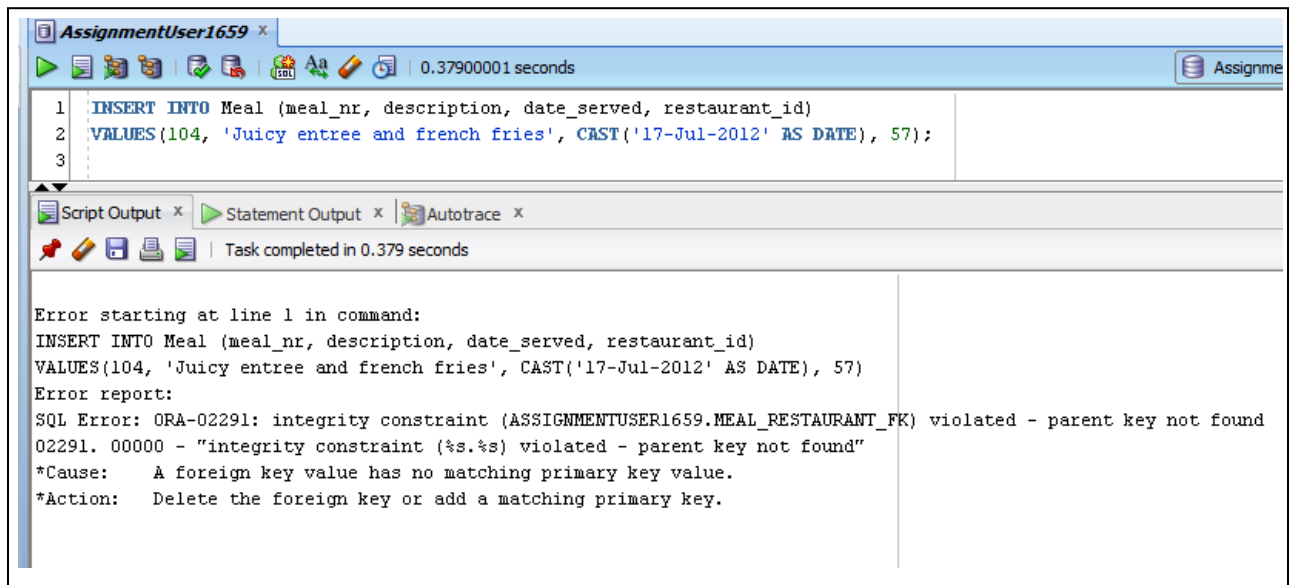
(1 row(s) affected)

(1 row(s) affected)

7. What if we attempt to insert a Meal that references a non-existent Restaurant, by using an invalid restaurant_id? The RDBMS will immediately reject the statement because it violates the foreign key constraint. Let us try it.

```
INSERT INTO Meal (meal_nr, description, date_served, restaurant_id)
VALUES(104, 'Juicy entree and french fries', CAST('17-Jul-2012' AS DATE), 57);
```

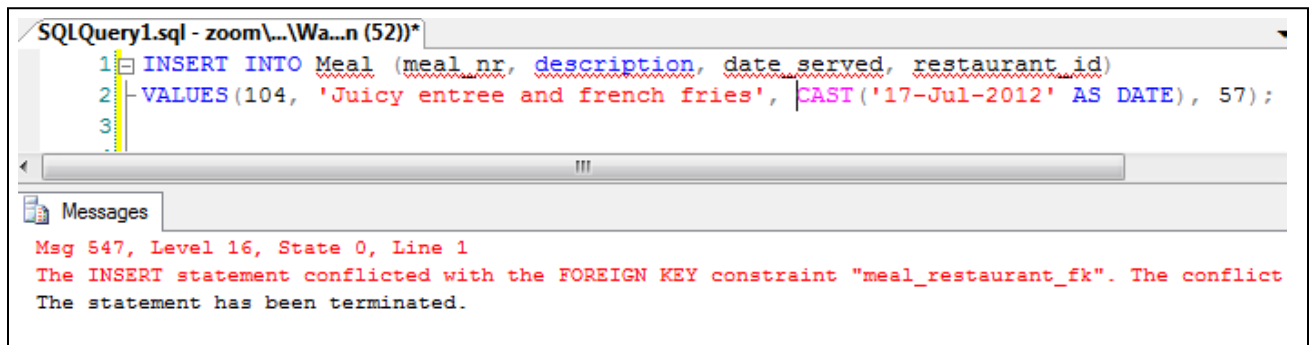
8. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer.



Notice that Oracle reports an error message indicating that the constraint `ASSIGNMENTUSER1659.MEAL_RESTAURANT_FK` has been violated. `ASSIGNMENTUSER1659` was the schema used when creating this lab, and that your schema will be different. Also recall that “meal_restaurant_fk” is the name we ascribed the foreign key constraint. Now you see the value of naming the constraint, since we can determine that the meal-to-restaurant foreign key has been violated without looking up the constraint.

Oracle provides this text, “parent key not found”, as an indication that we attempted to insert a value into a referencing column, that does not exist in a referenced column. In our example, we attempt to insert `restaurant_id 57`, which does not exist in the `Restaurant` table. Now you see how the foreign key constraint helps enforce the relationship between `Meal` and `Restaurant`. All references from `Meal` to `Restaurant` will be valid because of the presence of the foreign key constraint. Any attempt to insert an invalid reference is rejected by the RDBMS.

Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



Notice that just as Oracle rejected the statement, so did SQL Server, indicating that the foreign key constraint “meal_restaurant_fk” would be violated. Some of the text of the error message is truncated, so it is reproduced below:

```
--
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the FOREIGN KEY constraint
"meal_restaurant_fk". The conflict occurred in database "cs6692",
table "dbo.Restaurant", column 'restaurant_id'.
The statement has been terminated.
--
```

This error message may be easier to interpret than the corresponding Oracle error message, because it also indicates the table and column that participated in the attempted foreign-key violation. So we have the name of the constraint, “meal_restaurant_fk”, as well as an indication from the RDBMS of the table and column, removing all ambiguity. Note that database “cs6692” was the database used when creating this lab, and yours will likely be different.

9. We have created the table structure for our relationship, and also inserted related data. Now let us learn to retrieve our related data in a meaningful way.

When we retrieve data, generally we are not aimlessly or randomly retrieving rows and columns from tables; rather, we are answering a specific question for a specific purpose. For example, a manager may be gathering some general information on customers, or a web server may be generating a web page that lists all previous orders placed by a customer. Though there are widely varied questions to be answered and widely varied purposes for answering them, the same kinds of SQL constructs can be used for all of them.

The question to be answered can be simple or complex, but let us start with a simple question.

Which meals were served by which restaurants?

This general question gives us a direction, but we need more details, namely, precisely what properties of restaurants and meals we are looking for. Do we need the IDs? Do we need the names or descriptions? Do we need dates? We need to construct our request to be specific enough to be implemented. Let us then pose a more specific request:

List the description and date served of all of the meals served at a restaurant, and the name of the restaurant that served the meal.

This request requires data from two tables – Restaurant and Meal – and we can fulfill this request using a SELECT statement with a JOIN clause. Try the command below.

```
SELECT description, date_served, name
FROM Meal
JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id;
```

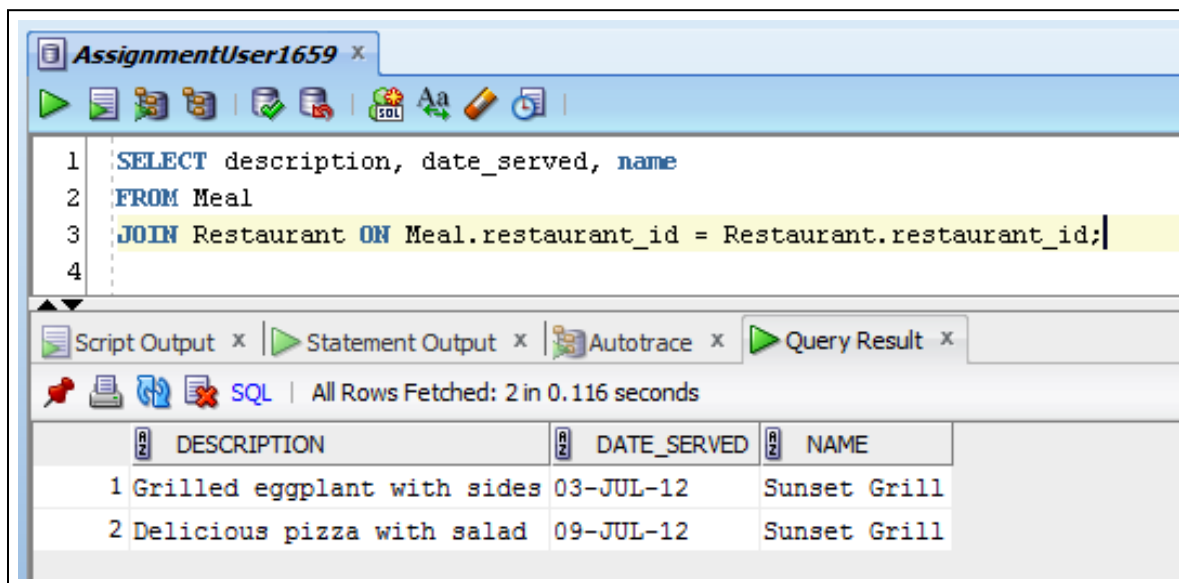
Let us examine this SQL command piece by piece. On the first line, you see the familiar SELECT keyword along with the column names to be retrieved. We retrieved specifically the description, date_served, and name columns. On the second line, you see the familiar FROM keyword along with the name of the table. It is the third line that introduces the JOIN keyword. The JOIN in this statement indicates that the Meal table will be joined with the Restaurant table. The ON keyword indicates the join condition, which is a Boolean expression that can use the columns in the Meal and Restaurant tables.

Recall that a join with no join condition between two tables is a Cartesian product. If we add a join condition, then we are selecting specific rows from the results of the Cartesian product. In other words, the join condition is evaluated for every row in the Cartesian product to determine which rows will be selected. In this example, our join condition:

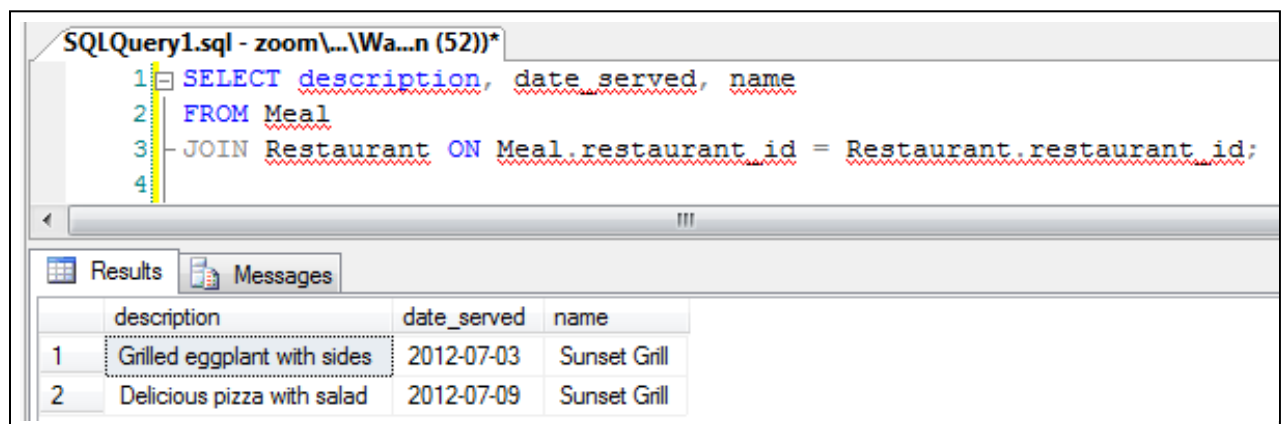
```
Meal.restaurant_id = Restaurant.restaurant_id
```

indicates that we only want the rows from the Cartesian product where the restaurant_id values are equal in the two tables. In plain English this means we want only want the meals that were served at restaurants, and only the restaurant that served the meal will be listed with a meal.

10. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer .



Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



The result set at the bottom is exactly what we expected. The first two meals' descriptions and dates served were listed, along with the Restaurant they were served at, the Sunset Grill. We have satisfied the request in Step 9 in full. This is exciting! We have learned to retrieve related data using a single SQL command!

The kind of join illustrated above is known as an inner join. There are several different ways in SQL to specify inner joins, and these are covered in the textbook and lectures. However, the style used above is defined in the ANSI standards and is the recommended style to use for maximum portability and ease of use.

11. Although inner joins can be used to fulfill many requests we have with our related data, some requests can only be answered with an outer join. For example, we could not use an inner join to fulfill this request:

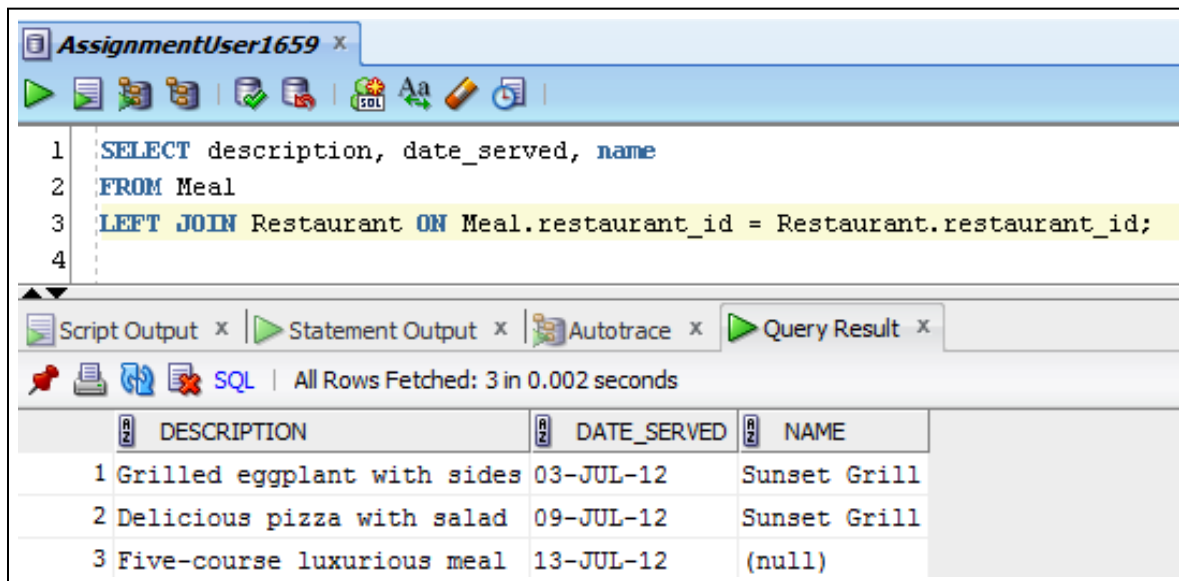
List the description and date served of all of the meals, and if the meal was served at a restaurant, list the name of the restaurant that served the meal.

Do you see the difference between this request and the request in Step 9? The request in Step 9 only asks for meals that were served at restaurants. The request in this step however is asking for all meals, whether or not they were served at a restaurant. We will fulfill this request by using a left outer join. We only need add one word to the SQL query listed in Step 9 to do so.

```
SELECT description, date_served, name
FROM Meal
LEFT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id;
```

What does this LEFT keyword do for us? It instructs the RDBMS to retrieve all rows that match the join condition, *and also* retrieve rows from the first table listed that do not match the join condition. In our example, the Meal table is the first table listed, and so any meals that do not have restaurants will also be listed. Let us try it.

12. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer .



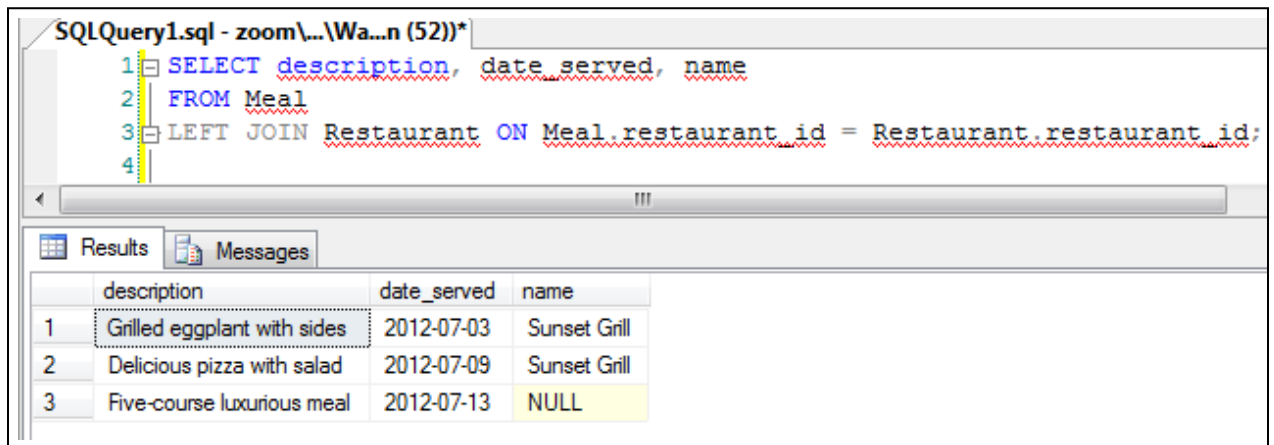
The screenshot shows the Oracle SQL Developer interface. The top toolbar includes icons for running, saving, and other database operations. The main window displays the following SQL query:

```
1 SELECT description, date_served, name
2 FROM Meal
3 LEFT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id;
4
```

Below the query editor, the 'Query Result' tab is active, showing the results of the query. The status bar indicates 'All Rows Fetched: 3 in 0.002 seconds'. The results are displayed in a table with three columns: DESCRIPTION, DATE_SERVED, and NAME.

	DESCRIPTION	DATE_SERVED	NAME
1	Grilled eggplant with sides	03-JUL-12	Sunset Grill
2	Delicious pizza with salad	09-JUL-12	Sunset Grill
3	Five-course luxurious meal	13-JUL-12	(null)

Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



The screenshot shows the SQL Query Editor with the following query:

```
1 SELECT description, date_served, name
2 FROM Meal
3 LEFT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id;
4
```

The Results pane shows the following data:

	description	date_served	name
1	Grilled eggplant with sides	2012-07-03	Sunset Grill
2	Delicious pizza with salad	2012-07-09	Sunset Grill
3	Five-course luxurious meal	2012-07-13	NULL

Notice that in this result set, the two matching rows are listed, just as with the inner join, but a third row is also listed. The “five course luxurious meal” is listed, and NULL is indicated for the restaurant name. This is because when we inserted our data, the “five course luxurious meal” was not given a restaurant_id because it was not served at a restaurant we had in our system.

13. The inverse request is straightforward.; however, let us enhance the request by introducing another requirement, which is that of sorting the results based upon the name of the restaurant.

List the name of all restaurants, ordered alphabetically. If the restaurant served any meals, list the description and date served of each meal.

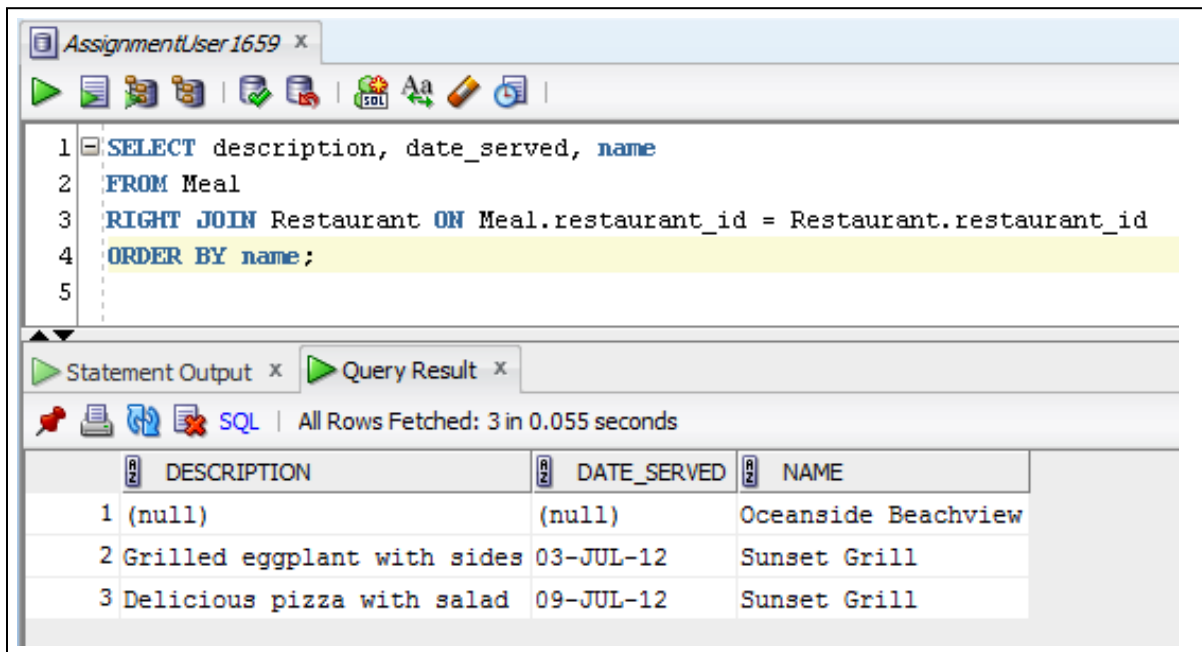
We fulfill this request by using a right outer join and an ORDER BY construct.

```
SELECT description, date_served, name
FROM Meal
RIGHT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id
ORDER BY name;
```

The difference between a right outer join and a left outer join is that a right outer join retrieves rows that do not match from the second table listed, while a left outer join retrieves rows that do not match from the first table listed.

The ORDER BY construct tells the RDBMS to sort the results based upon the values in a column or group of columns. A comma-separated list of column names follows the ORDER BY keywords. In our example, we only wanted to sort by the restaurant name, and so we have listed only one column.

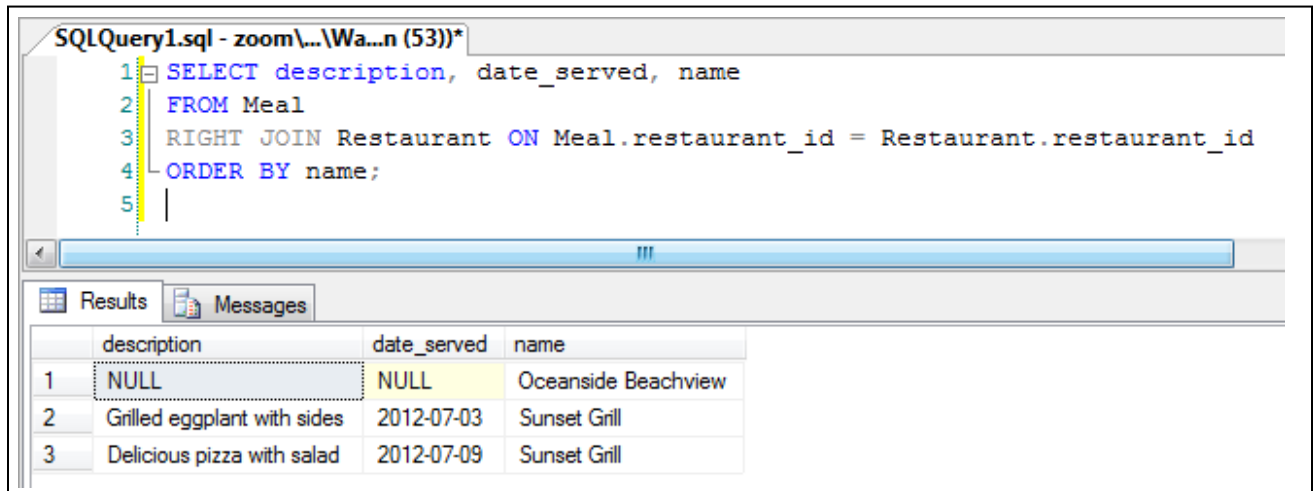
14. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer .



The screenshot shows the Oracle SQL Developer interface. The top pane contains a SQL query: `SELECT description, date_served, name FROM Meal RIGHT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id ORDER BY name;`. The bottom pane shows the query results in a table format. The table has three columns: DESCRIPTION, DATE_SERVED, and NAME. The results are as follows:

	DESCRIPTION	DATE_SERVED	NAME
1	(null)	(null)	Oceanside Beachview
2	Grilled eggplant with sides	03-JUL-12	Sunset Grill
3	Delicious pizza with salad	09-JUL-12	Sunset Grill

Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



The screenshot shows the Microsoft SQL Server Management Studio interface. The top pane contains a SQL query: `SELECT description, date_served, name FROM Meal RIGHT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id ORDER BY name;`. The bottom pane shows the query results in a table format. The table has three columns: description, date_served, and name. The results are as follows:

	description	date_served	name
1	NULL	NULL	Oceanside Beachview
2	Grilled eggplant with sides	2012-07-03	Sunset Grill
3	Delicious pizza with salad	2012-07-09	Sunset Grill

Notice that in this result set, the second and third rows are the matching rows, and are the same rows returned with by the equivalent inner join. The first row is the row that does not match in the Restaurant table, resulting from the right outer join we performed. "Oceanside Beachview" is the restaurant name, and NULL is indicated for the Meal columns. Recall that when we inserted the data, we did not indicate that the Oceanside Beachview Restaurant served any meals.

Further notice that the results have been ordered just as we specified. Since “O” comes before “S”, the Oceanside Beachview restaurant is ordered before the Sunset Grill restaurant. If we had not used the ORDER BY construct, the Oceanside Beachview row would have come last, since it was inserted after the Sunset Grill rows.

15. We can also retrieve matching rows and rows that do not match in both tables, which is a combination of what we did in Steps 11 and 13. Below is a request that asks us to do so.

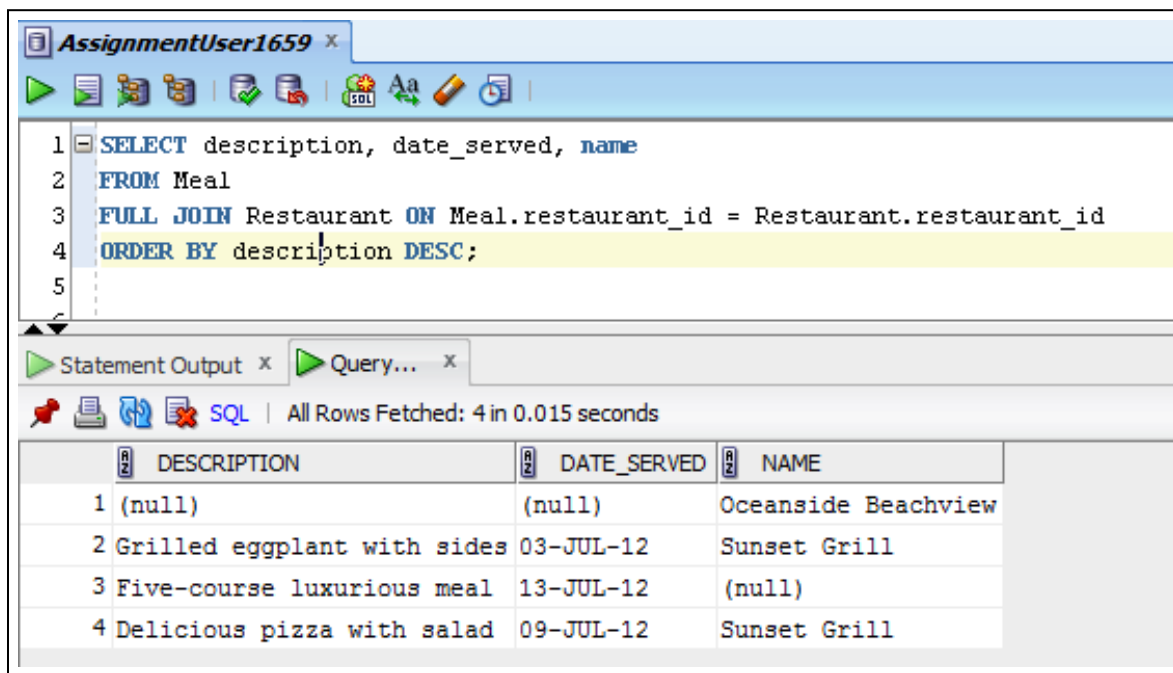
List the descriptions and dates served of all meals, and the names of all restaurants. Show which restaurants served which meals. The list should be sorted by the descriptions of the meals in reverse alphabetical order.

This request can be fulfilled with a full outer join, which returns the matching rows, and also rows that do not match in both the first and second tables. In short, a full outer join is a combination of left and right outer joins. We again need to use the ORDER BY construct to fulfill the sorting requirement.

```
SELECT description, date_served, name
FROM Meal
FULL JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id
ORDER BY description DESC;
```

We added a qualification to the ORDER BY construct by placing the DESC keyword after the list of column names. The request asked us to sort by the meal descriptions in reverse alphabetical order. The keyword DESC is short for “descending”, meaning that the ordering will be reversed. When we want to sort in order, we can either use the keyword ASC, which is short for “ascending”, or we can omit these words altogether, because ascending is the default when ORDER BY is used.

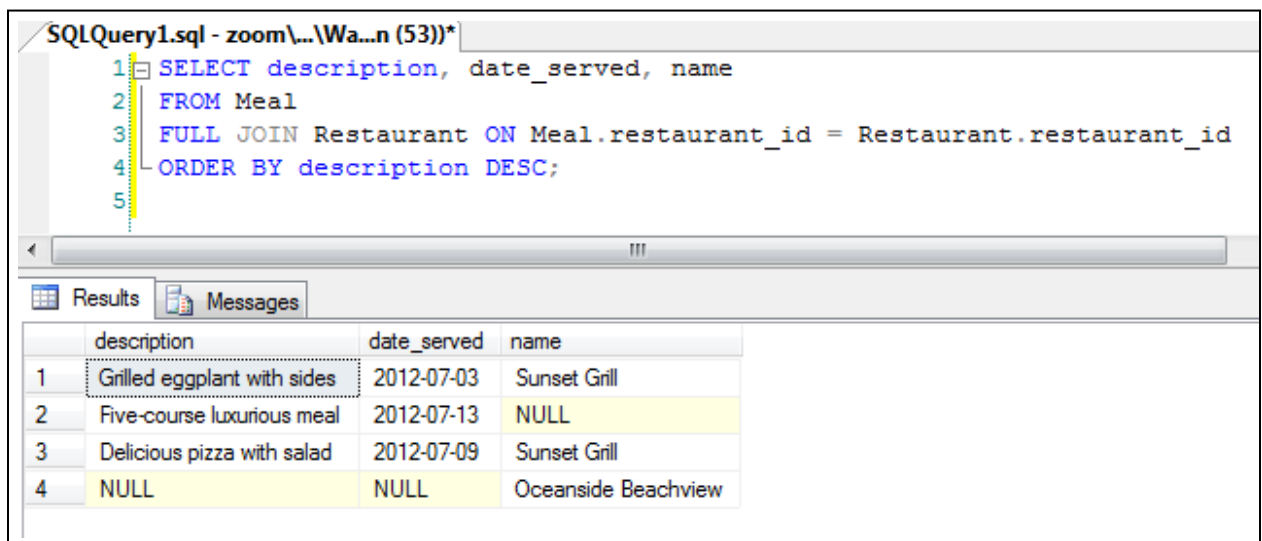
16. Capture a screenshot of the command and the result of its execution. Below is a sample screenshot of the command execution in Oracle SQL Developer .



The screenshot shows the Oracle SQL Developer interface. The top pane displays a SQL query: `SELECT description, date_served, name FROM Meal FULL JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id ORDER BY description DESC;`. The bottom pane, titled 'Statement Output', shows the results of the query. It indicates 'All Rows Fetched: 4 in 0.015 seconds'. The results are displayed in a table with three columns: DESCRIPTION, DATE_SERVED, and NAME.

	DESCRIPTION	DATE_SERVED	NAME
1	(null)	(null)	Oceanside Beachview
2	Grilled eggplant with sides	03-JUL-12	Sunset Grill
3	Five-course luxurious meal	13-JUL-12	(null)
4	Delicious pizza with salad	09-JUL-12	Sunset Grill

Below is a sample screenshot of the command execution in Microsoft SQL Server Management Studio.



The screenshot shows the Microsoft SQL Server Management Studio interface. The top pane displays a SQL query: `SELECT description, date_served, name FROM Meal FULL JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id ORDER BY description DESC;`. The bottom pane, titled 'Results', shows the results of the query. It indicates 'All Rows Fetched: 4 in 0.015 seconds'. The results are displayed in a table with three columns: description, date_served, and name.

	description	date_served	name
1	Grilled eggplant with sides	2012-07-03	Sunset Grill
2	Five-course luxurious meal	2012-07-13	NULL
3	Delicious pizza with salad	2012-07-09	Sunset Grill
4	NULL	NULL	Oceanside Beachview

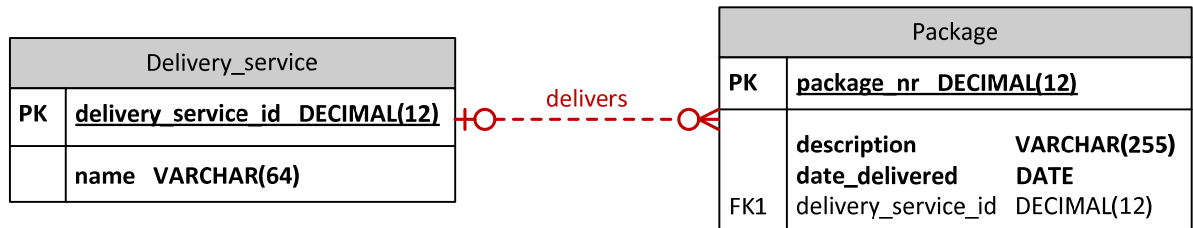
Notice that in this result set, the two matching rows are listed, just as with the inner join. In addition, the row that does not match from the Meal table is listed, as is the row that does not match from the Restaurant table. These results were as expected from the full outer join we used above. We retrieved which meals were served at which restaurants, retrieved the meals that were not served at restaurants, and retrieved the restaurants that did not serve any meals.

In this result set, the meal descriptions were sorted in reverse alphabetical order just as we specified in our command. You may have noticed that the row with the NULL description is first in the result set with Oracle, and last with SQL Server. Because NULL represents no value at all, one RDBMS will treat it as greater than all values, while another RDBMS will treat it as less than all values, as demonstrated above. This small example has hinted at the world of nuances and complexities that exist with NULL.

SECTION TWO

OVERVIEW

In Section Two, you will apply what you learned in Section One to new tables, without being given the commands and screenshots. You will work with `Delivery_service` and `Package` tables that are related via a “delivers” relationship, all of which are diagrammed below. Note that the bolded columns represent those with a NOT NULL constraint.



STEPS

1. Create the `Delivery_service` and `Package` tables, including all of their columns, datatypes, and constraints. Make sure to create the foreign key constraint.
2. Capture a screenshot of the commands and the results of their execution.
3. Insert the following delivery services into the `Delivery_service` using `delivery_service_ids` of your choosing.
 - United Package Delivery
 - Federal Delivery
 - Dynamic Duo Delivery
4. Capture a screenshot of the insert commands and the results of their execution.
5. Insert the following packages into the `Package` table using `package_nr` values of your choosing.

Delivered by United Package Delivery:

description = Perfect diamonds
date_delivered = 29-Apr-2012

Delivered by Federal Delivery:

description = Care package
date_delivered = 14-Jun-2012

No delivery service:

description = French wine

date_delivered = 19-Jul-2012

6. Capture a screenshot of the insert commands and the results of their execution.
7. Attempt to insert a Package that references a non-existent delivery service.
8. Capture a screenshot of the command and the result of its execution.
9. Explain:
 - a. why the insertion in Step 7 failed, and
 - b. how you would interpret the error message from your RDBMS so that you know that the error indicates the delivery service ID was invalid
10. Explain the similarities and differences between an inner join, a left join, a right join, and a full outer join.
11. With a single SQL query, fulfill the following request:

List the description and date delivered of all packages delivered by a delivery service, as well as the name of the delivery service that delivered the package.

12. Capture a screenshot of the command and the result of its execution.
13. Explain why some rows in Delivery_service and some rows in Package were not listed in Step 11.
14. With a single SQL query, fulfill the following request:

List the names of all delivery services, and if the delivery service delivered any packages, list the descriptions and delivery dates for those packages.

15. Capture a screenshot of the command and the result of its execution.
16. With a single SQL query, fulfill the following request:

List the description and date delivered of all packages, and if the package was delivered by a delivery service, list the name of the delivery service. The list should be ordered alphabetically by the name of the delivery service.

17. Capture a screenshot of the command and the result of its execution.
18. With a single SQL query, fulfill the following request:

List the descriptions and dates delivered of all packages, and also list the names of all delivery services. Indicate which packages were delivered by which delivery services. The package descriptions should be listed in reverse alphabetical order.

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