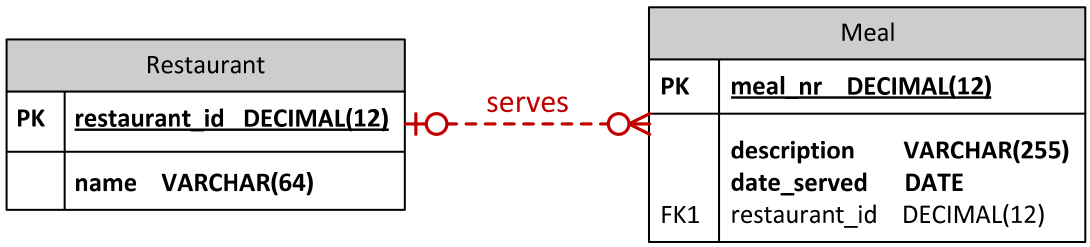
Lab 2 Explanations

This explanation document illustrates how to correctly execute each SQL construct step-by-step for Lab 2, and explains important theoretical and practical details. Before completing a step, read its explanation here first.

Table of Contents

Step 1 – Creating the Table Structure

In order to show you examples similar to the steps you need to complete, we will work with the Restaurant and Meal tables illustrated below in this section.



The Restaurant table stores the primary key and name of restaurants. The Meal table stores the primary key, a description of a meal, the date the meal was served, and a foreign key that references the Restaurant table which records the restaurant the meal was served in. Note that the bolded columns represent those with a NOT NULL constraint.The foreign key enforces the relationship between Meal and Restaurant. This schema is intentionally simplified to best aid you in your learning.

We execute the following commands to create the Restaurant and Meal tables.

```
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 the prior lab. 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.

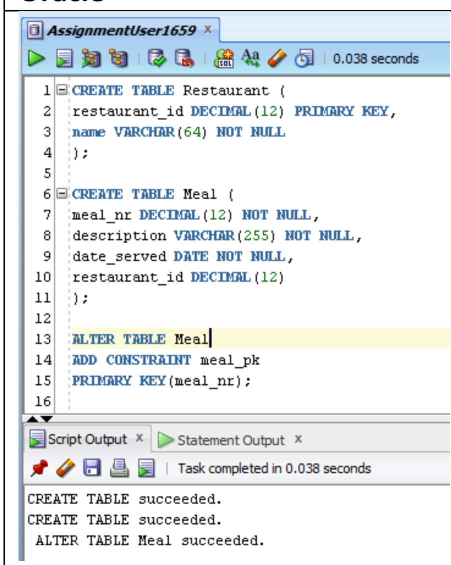
Just as we name tables with an identifier, we can name constraints. If we use the shorthand method of adding constraints by including them as part of a column definition in a CREATE TABLE statement, such as with the CREATE TABLE statement that creates the Restaurant table, the RDBMS generates a constraint 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.

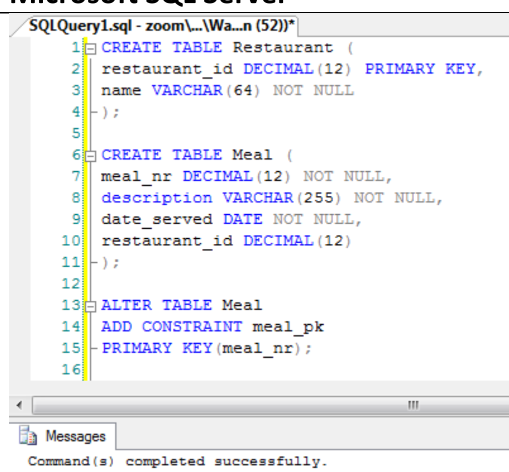
The PRIMARY KEY keywords further indicate to the RDBMS that we are adding a PRIMARY KEY constraint. The syntax requires us to then enclose a comma-separated list of column names within parentheses. The columns specified in this list will all be covered by the new primary key constraint. In our example,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

Below are screenshots of the command execution in each RDBMS.

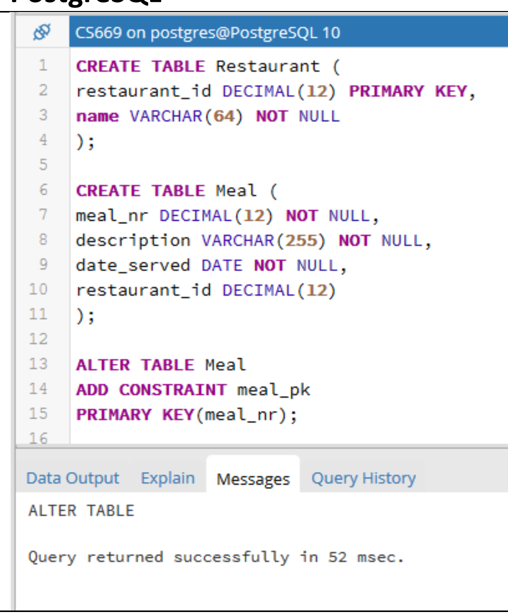
Oracle



Microsoft SQL Server



PostgreSQL



You may have noticed that the foreign key has not yet been defined, and we will do so now. The foreign key constraint can be defined with another ALTER statement as follows.

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

The first two lines have the same format as the ALTER TABLE command we used previously, 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.

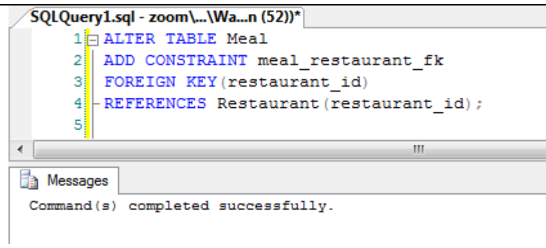
If you work with databases long enough, you will run into a situation where a composite foreign key covering more than one column is in use. When defining the constraint for composite foreign keys, the first column identified in the first list references the first column in the second list, the second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

Using foreign keys allows us to harness one of the most useful and powerful features 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. One of the first steps in relating data is setting up the structure of the tables to enforce the relationships we expect through foreign key constraints.

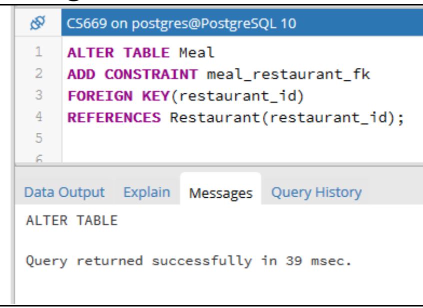
Below are screenshots of the command execution in each RDBMS.

Oracle

Microsoft SQL Server



PostgreSQL



Step 2 – Populating the Tables

Since you need to insert related data for this step, let's show you example inserts for the Restaurant and Meal schema, below.

```
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.

Below are sample screenshots of the command execution in each RDBMS.

Oracle

AssignmentUser1659

```sql
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);
```

Script Output | Statement Output | Autotrace

Task completed in 0.201 seconds

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

Microsoft SQL Server

SQLQuery1.sql - zoom\...\Wa...n (52))*

```sql
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);
```

Messages

```
(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)

(1 row(s) affected)
```

PostgreSQL

CS669 on postgres@PostgreSQL 10

```sql
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);
```

Data Output | Explain | Messages | Query History

```
INSERT 0 1

Query returned successfully in 38 msec.
```

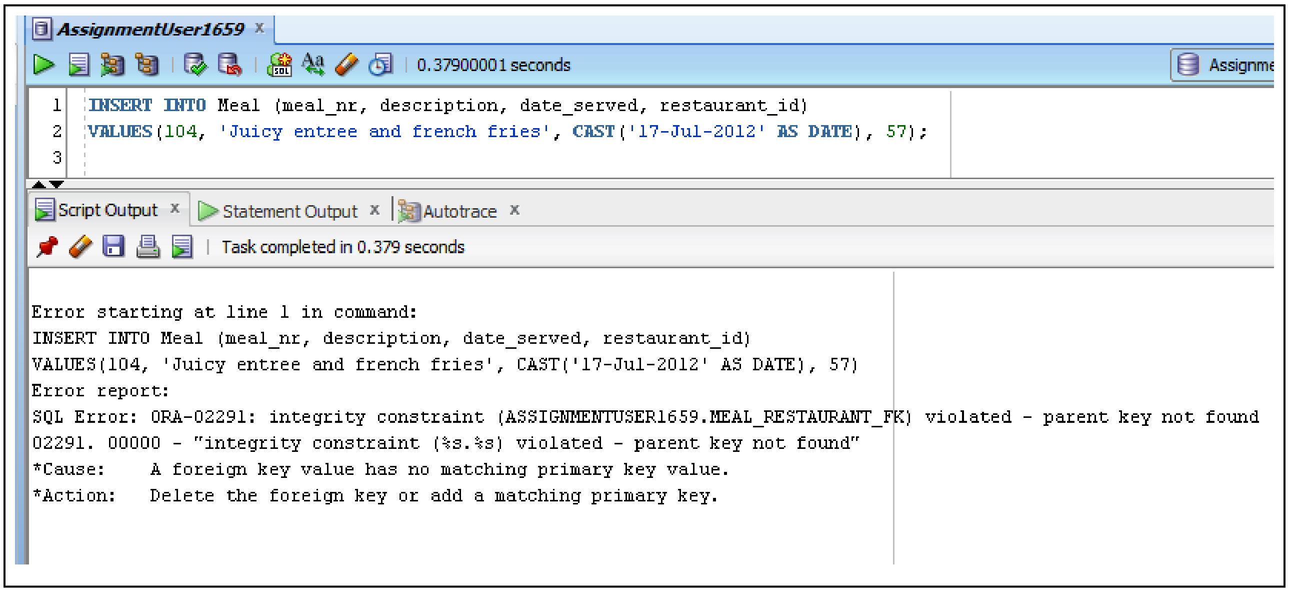
For your inserts, you can follow the same pattern. Create primary key values for each row, then reference the appropriate row by giving the same value in the foreign key.

Step 3 – Invalid Reference Attempt

Let’s try something similar by attempting an invalid insert into the Meal table, that is, by attempting to insert a Meal that references a non-existent Restaurant. We do this by using an invalid restaurant_id. The RDBMS will immediately reject the statement because it violates the foreign key constraint. Let us try it with the following command (note that restaurant_id 57 is invalid).

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

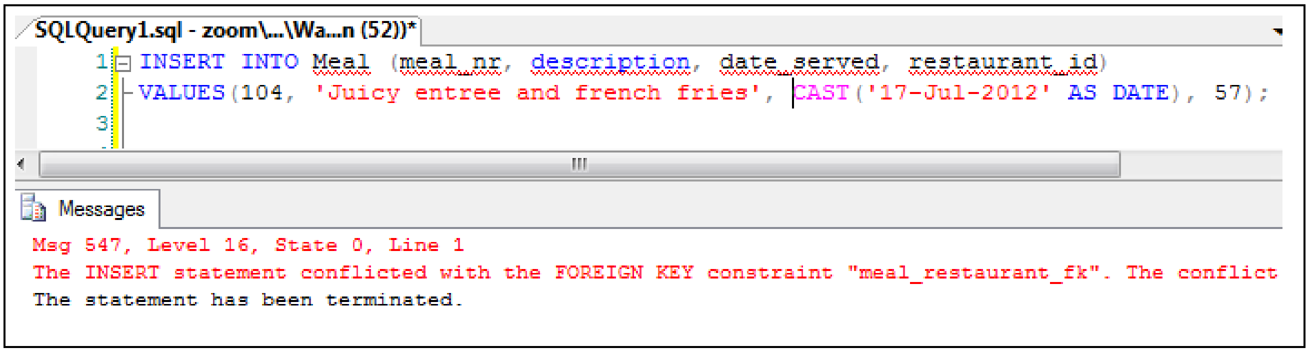
Below is a sample screenshot of the command execution in Oracle.



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.

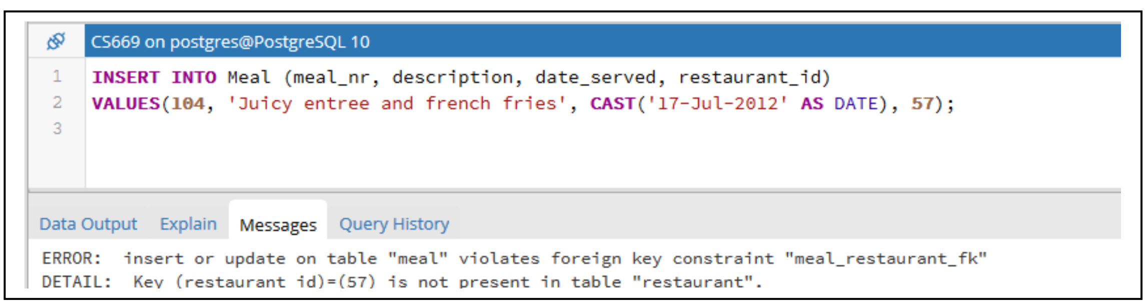


Notice that just as Oracle rejected the statement, so did SQL Server, indicating that the foreign key constraint “meal_restaurant_fk” would be violated. Some of the text of the error message is truncated, so it is reproduced below:

```
--
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.

Below is a sample screenshot of the command execution in Postgres.



Notice that PostgreSQL also rejects the insert. Not only does error message reference the table and constraint name “meal_restaurant_fk” but also gives detail such as the column name (restaurant_id) and value (57), plus the name of the parent table (restaurant), causing the foreign key violation.

Step 4 – Listing Matches

In prior steps you created a related structure and inserted related data; the next logical step is to learn how to make use of related data. A *join* is both a fundamental concept detailing how data from related tables can be retrieved, as well as a critical SQL operation. This step is explained conceptually first using two simplistic tables, then later applied to the example Restaurant and Meal schema we've been using throughout this lab.

The two simplistic, related tables used to teach you the join concepts are listed below.

Words and FirstLetters

FirstLetters

FirstLetterId	FirstLetter
1	A
2	B
3	C
4	D
5	E

Words

Word	FirstLetterId
Apple	1
Cherry	3
Elderberry	5
Kiwi	

The FirstLetters tables contains letters A through E, representing a letter words can start with, along with the primary key FirstLetterId. The Words table has various words in it, along with a reference to the FirstLetters table to indicate the first letter of the word. The FirstLetterId column in the Words table is the foreign key to the FirstLetters table, and is how the tables are related.

Before we dive into the technical mechanics of joins, let's look at what we use a join for. Simply put, *the purpose of a join is to enable us to answer questions about related data*. Without a join, we could only answer questions from a single table and not multiple, related tables. For example, with a single table in the Words schema, we could answer the following questions.

- How many first letters are available?
- What words are available?
- Is the letter X available as a first letter?

However, from a single table, we could *not* answer these questions.

- What letters have no words?
- What words start with letters not available as a first letter?
- Which words start with the letter C?

These questions require data from both tables, so we need a join to answer them.
If you're familiar with more advanced SQL, you may have noticed that some of the questions could be answered by using string analysis. This is a byproduct of using such a simple schema to teach you about joins. It is true that some questions could be answered with string analysis, but also inefficient when the datasets

become large, so generally in production environments we would still rely on using joins rather than attempting string analysis across many records. More importantly, however, is that many questions across many use cases can only be answered with joins and not a shortcut method such as string analysis.

Now that you know the purpose of a join, you need to understand its mechanics from a technical perspective. The mechanics are non-trivial, but systematic; a careful review of how joins work will enable to you to perform joins for a large variety of use cases. The relational model has a basis in set theory, so the operations that are applied to relational databases are mechanical and even somewhat mathematical in nature. When a join between two tables occurs, conceptually what is happening is a cartesian product between the two tables, then a selection of rows from that cartesian product.

For example, if we join the FirstLetters and Words tables on the FirstLetterId column, conceptually the cartesian product is created, then the matching rows are selected from that product. This is illustrated below.

FirstLetters and Words Join Mechanics

Cartesian Product of FirstLetters/Words

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
2	B	Apple	1
3	C	Apple	1
4	D	Apple	1
5	E	Apple	1
1	A	Cherry	3
2	B	Cherry	3
3	C	Cherry	3
4	D	Cherry	3
5	E	Cherry	3
1	A	Elderberry	5
2	B	Elderberry	5
3	C	Elderberry	5
4	D	Elderberry	5
5	E	Elderberry	5
1	A	Kiwi	
2	B	Kiwi	
3	C	Kiwi	
4	D	Kiwi	
5	E	Kiwi	

Result of Joining FirstLetters/Words on FirstLetterId

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5

Notice the cartesian product's column structure is the combination of both tables' columns, that is, a combination of FirstLetterId and FirstLetter from the FirstLetters table, and Word and FirstLetterId from the Words table. Further notice that the cartesian product's row structure is the combination of every possible row, such as "1 A Apple 1", "2 B Apple 1", and so on. Since there are 5 rows in the FirstLetters table, and 4 rows in the Words table, this results in 20 rows in the cartesian product.

While the cartesian product is the first conceptual step in this join, the second step is the selection of matching rows, resulting in the elimination of unmatching rows. Since we join on the FirstLetterId column, the matching rows are the ones where the FirstLetterId column has the same values. In this example, only three rows match, when FirstLetterId is 1, 3, and 5 on both sides. Thus, the result is a table that has the combination of both tables' columns, with the three matching rows, as illustrated in the figure.

To summarize what we've learned thus far, the purpose of a join is to ask questions about related data, and the mechanics conceptually start with the cartesian product of both tables, then matching rows are selected from that cartesian product (eliminating the unmatching rows). A join thus results in a table that has the combination of columns from both tables, and the matching rows.

You know both the purpose and the mechanics of joins in concept, but how do you use it in SQL? Basic joins are not too difficult. There is a JOIN keyword in SQL that is combined with an ON clause. This is illustrated in the figure below.

SQL Join Between FirstLetters and Words

```
SELECT *
FROM   FirstLetter
JOIN   Words ON Words.FirstLetterId = FirstLetters.FirstLetterId
```

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5

You are already familiar with the first two lines of this query, a basic SELECT/FROM statement. The third line is interesting. To join to the Words table, the JOIN keyword is used, followed by the name of the table, in this case, "Words". The ON clause is used to give a Boolean expression indicating which rows in the cartesian product should match in the join. Again, recall that the first conceptual step in a join is the creation of the cartesian product, and the second conceptual step is the selection of matching rows. The JOIN keyword indicates to the SQL processor that we are performing a join, and the ON keyword indicates what Boolean expression will be used to determine which rows match. In this example, since we used the simple equality condition of "Words.FirstLetterId = FirstLetters.FirstLetterId", we are performing a basic join using the FirstLetterId column.

Some questions require more than just matching rows, so there is more than one kind of join to handle such situations. For example, there is a subtle difference between these two use cases:

1. List the starting letters and the words they start with.
2. List all starting letters, and indicate which words each letter starts with (if available).

The first use case can be addressed with a simple join, as already demonstrated above. The second use case, however, is asking for all starting letters, whether or not they have a corresponding word in the database. In the aforementioned example, the "B" and "D" rows were not listed with a standard join because they have no corresponding rows in the database, so using the same approach would not solve use case #2.

We could list all letters simply by selecting from the FirstLetters table. However, this will not tell us the words that go along with the letters. We could list all words and their starting letters by using a join as demonstrated previously, but the mechanics of the way a join works would eliminate any words that do not reference a starting letter (in our example, the “B” and “D” letters have no words). What do we do? The answer is, use a join that allows us to both list all starting letters and words if available, but to also list starting letters that do not have words.

There are four broad join types – inner, left, right, and outer. All four joins have the conceptual first step of creating the cartesian product and all four keep the rows that match the join condition. The left, right, and outer join also keep additional rows. An inner join is what you have just learned about, a join between two tables where all rows that match the join condition are kept and all others are removed. The terms basic join, standard join, and inner join are synonymous. A left join keeps the rows that match the join condition as well a distinct list of rows from the first (leftmost) table that did not match the join condition. A right join keeps the rows that match the join condition as well a distinct list of rows from the second (rightmost) table that did not match the join condition. An outer join is a combination of the left and right join, where rows that match the join condition are included, as well as a distinct list of rows from both the first and second tables that did not match the join condition. These are somewhat complex concepts, so left, right, and outer joins are explained in more detail in the subsequent paragraphs.

Let’s look at the left join. First, what is meant by the “first” or “leftmost” table? Simple, it’s the table that comes first in the SQL statement. The first table listed is thought to be to the “left” of the second table. Next, what are the mechanics of a left join? Just as with the inner join, conceptually the cartesian product is the first step, and likewise matching rows are indeed added to the result. What makes a left join unique is that it also adds a distinct list of rows from the first table to the result. This is illustrated below.

Left Join Between FirstLetters and Words

Cartesian Product of FirstLetters/Words

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
2	B	Apple	1
3	C	Apple	1
4	D	Apple	1
5	E	Apple	1
1	A	Cherry	3
2	B	Cherry	3
3	C	Cherry	3
4	D	Cherry	3
5	E	Cherry	3
1	A	Elderberry	5
2	B	Elderberry	5
3	C	Elderberry	5
4	D	Elderberry	5
5	E	Elderberry	5
1	A	Kiwi	
2	B	Kiwi	
3	C	Kiwi	
4	D	Kiwi	
5	E	Kiwi	

Rows Matching Join Condition

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5

Rows from FirstLetters Not Matching the Join Condition

FirstLetterId	FirstLetter
1	A
2	B
3	C
4	D
5	E

Results

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5
2	B		
4	D		

Notice that the matching rows from the cartesian product are included in the left join, just as they would be with an inner join. What makes the left join unique is that any rows from the FirstLetters table that are not included in the matching rows are added to the result set. Since rows “2 B” and “4 D” were not matched in the cartesian product, they are added to the results. The columns that come from the Words table are left as null for such rows. Thus, if FirstLetters has 5 rows, you know that the results are guaranteed to have at least 5 rows, since every row in the FirstLetters table is guaranteed to appear at least once in the result.

This left join can successfully address the use case mentioned earlier, “List all starting letters, and indicate which words each letter starts with (if available).” The left join works because all FirstLetters are guaranteed to make it into the results whether or not they have corresponding words. An inner join would not suffice.

To do this in SQL, we use the keywords “LEFT JOIN” rather then just “JOIN”, as illustrated below.

SQL Left Join Between FirstLetters and Words

```
SELECT    *
FROM      FirstLetter
LEFT JOIN Words ON Words.FirstLetterId = FirstLetters.FirstLetterId
```

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5
2	B		
4	D		

The left join produces the results described, with the “2 B” and “4 D” rows added into the result set, with the Words columns left null for those rows. This happens simply by using the “LEFT JOIN” keywords rather than simply the “JOIN” keywords.

Let’s look at another use case, “List all words, and the letters they start with (if available)”. Of course, using an inner join would *not* satisfy this use case, because any words that do not reference starting letters would be excluded, yet the use case is asking for all words, not only some. We can however use a right join to get the results we expect. As you might have guessed, a right join uses similar logic as the left join, except that rows not used in the second (rightmost) table are included instead of the rows from the leftmost table. Just as with the inner and left joins, first conceptually the cartesian product is created and matching rows are selected from there. The right join then includes any rows not already included from the second (rightmost) table. This is illustrated below.

Right Join Between FirstLetters and Words

Cartesian Product of FirstLetters/Words

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
2	B	Apple	1
3	C	Apple	1
4	D	Apple	1
5	E	Apple	1
1	A	Cherry	3
2	B	Cherry	3
3	C	Cherry	3
4	D	Cherry	3
5	E	Cherry	3
1	A	Elderberry	5
2	B	Elderberry	5
3	C	Elderberry	5
4	D	Elderberry	5
5	E	Elderberry	5
1	A	Kiwi	
2	B	Kiwi	
3	C	Kiwi	
4	D	Kiwi	
5	E	Kiwi	

Rows Matching Join Condition

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5

Rows from Words Not Matching the Join Condition

Word	FirstLetterId
Apple	1
Cherry	3
Elderberry	5
Kiwi	

Results

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5
		Kiwi	

Notice that just as with the left and inner joins, the cartesian product's matching rows are included, and then the "Kiwi" row from Words, which was not included in the cartesian product's matches, is added to the results. Further notice that the columns from FirstLetters are null for that row. The results of this right join indeed address the use case, "List all words, and the letters they start with (if available)", since all words are listed whether or not they have a matching starting letter.

To do a right join in SQL, as you might have guessed, we use the words "RIGHT JOIN", as illustrated below.

SQL Right Join Between FirstLetters and Words

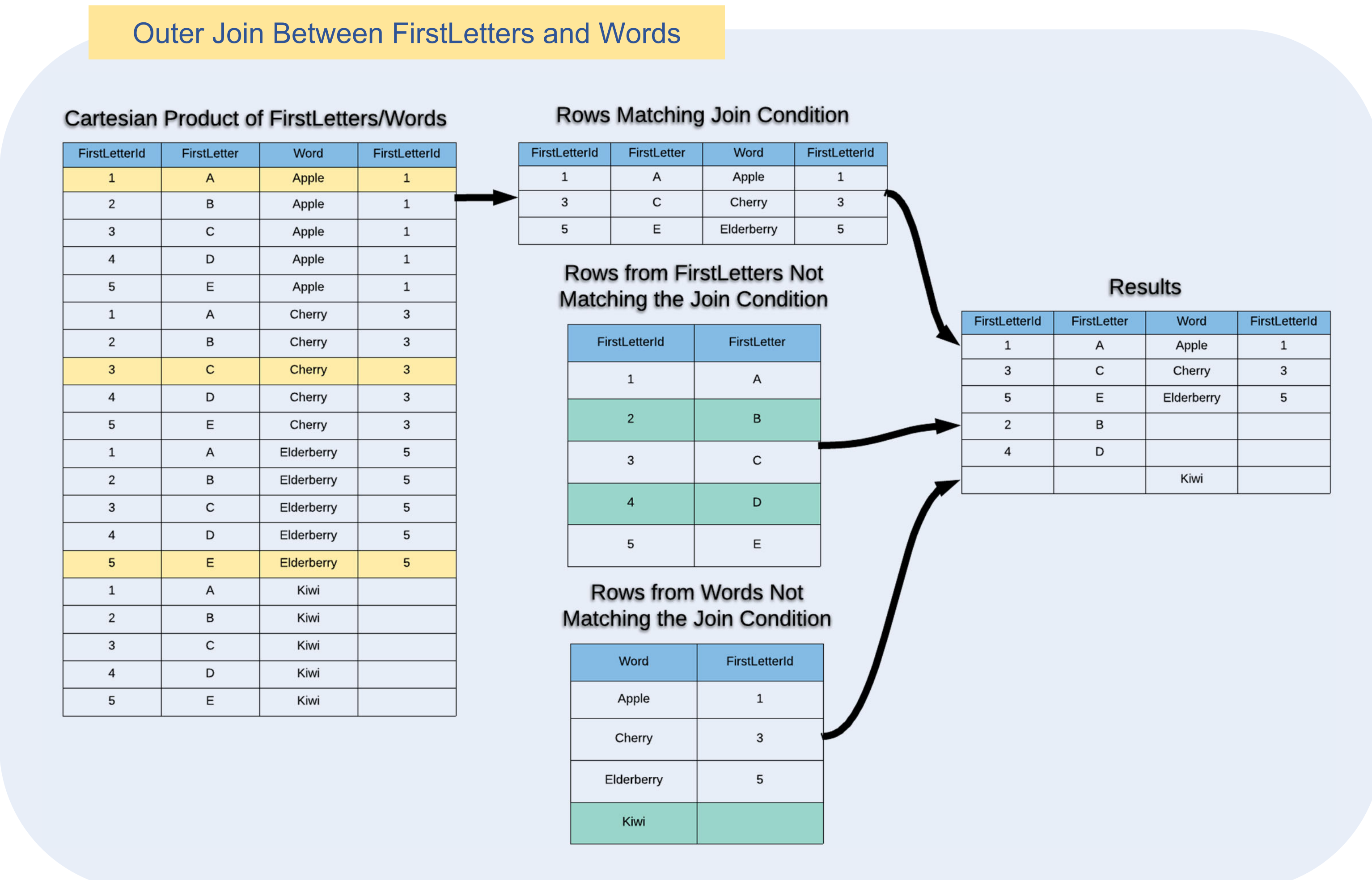
```
SELECT     *
FROM       FirstLetter
RIGHT JOIN Words ON Words.FirstLetterId = FirstLetters.FirstLetterId
```

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5
		Kiwi	

The right join produces the results described, with the “Kiwi” row added in.

So what does an outer join do? Simply put, it combines the extra rows from both the left and the right join. Let’s demonstrate the outer join by coming both use cases we used to demonstrate the left and right join, “List all words and all starting letters, and indicate which words start with which letters (if available)”. Of course, an inner join would not satisfy this use case because it would exclude words without starting letters, and starting letters without words. A left join or right join would not satisfy the use case because it would leave one set rows out (we would either get all words, or all letters, but not both).

The mechanics of the outer join are illustrated below.



The matching rows from the cartesian product are included, as are the missing rows from FirstLetters and Words. Where a missing row from FirstLetters is included, the Words columns are null in the result. Likewise, where a missing row from Words is included, the FirstLetters columns are null. So you see the results contain the three matching rows, then the rows “2 B” and “4 D” from the FirstLetters table, then the “Kiwi” row from the Words table. This outer join example thus satisifies the use case, “List all words and all starting letters, and indicate which words start with which letters (if available)”. All words and all starting letters are included, and the matches are indicated where they exist.

To perform an outer join in SQL, use the words “FULL JOIN”, as illustrated below.

SQL Outer Join Between FirstLetters and Words

```
SELECT    *
FROM      FirstLetter
FULL JOIN Words ON Words.FirstLetterId = FirstLetters.FirstLetterId
```

FirstLetterId	FirstLetter	Word	FirstLetterId
1	A	Apple	1
3	C	Cherry	3
5	E	Elderberry	5
2	B		
4	D		
		Kiwi	

Using the words FULL JOIN gives us the results we expect from an outer join.

Part of the exercise of this lab is understanding the different types of joins, and how to use them to address use cases correctly. Examples of using the four join types to solve use cases for our Restaurant and Meal schema are given below. Review the examples, and then you'll be able to make your own choice as to which join satisfies which use case for this and several steps. As part of the examples, ordering the rows in the result is also explained.

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.

```
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.

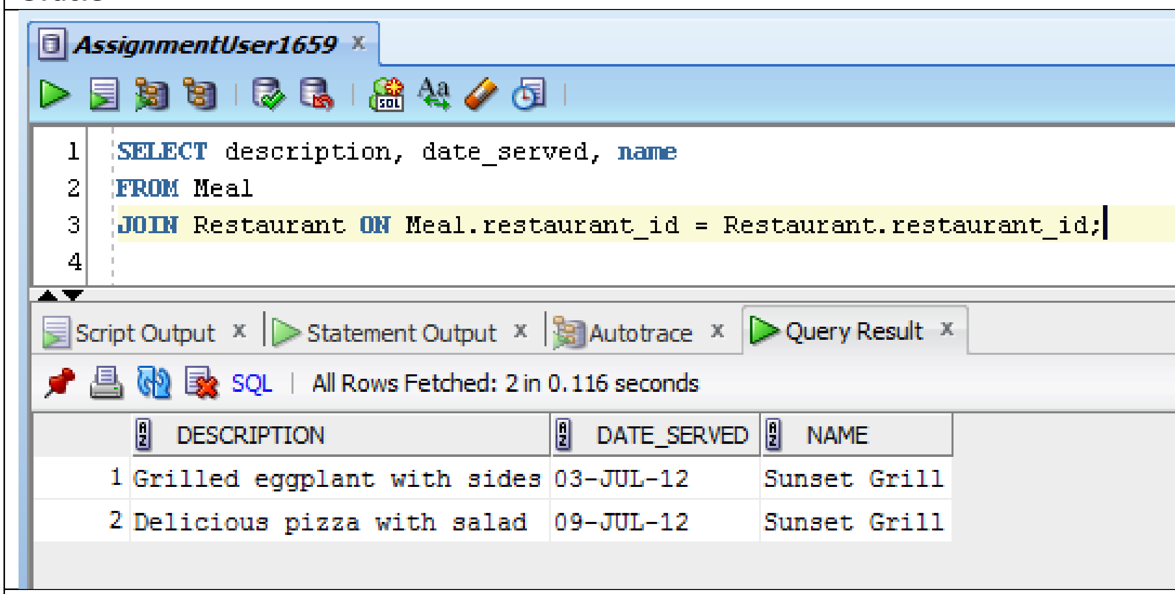
Recall that a join with no join condition between two tables is a Cartesian product. If we add a join condition, then we are selecting specific rows from the results of the Cartesian product. In other words, the join condition is evaluated for every row in the Cartesian product to determine which rows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

Below is a sample screenshot of the command execution in each RDBMS.

Oracle



Microsoft SQL Server

SQLQuery1.sql - zoom\...\Wa...n (52))*

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

Results | Messages

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

PostgreSQL

CS669 on postgres@PostgreSQL 10

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

Data Output | Explain | Messages | Query History

	description character varying (255)	date_served date	name character varying (64)
1	Delicious pizza with salad	2012-07-09	Sunset Grill
2	Grilled eggplant with sides	2012-07-03	Sunset Grill

The result set for each execution is exactly what we expect. 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 **Error! Reference source not found.** in full. This is exciting! We have learned to retrieve related data using a single SQL command!

The kind of join illustrated above, inner join, has several different syntaxes, 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.

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.

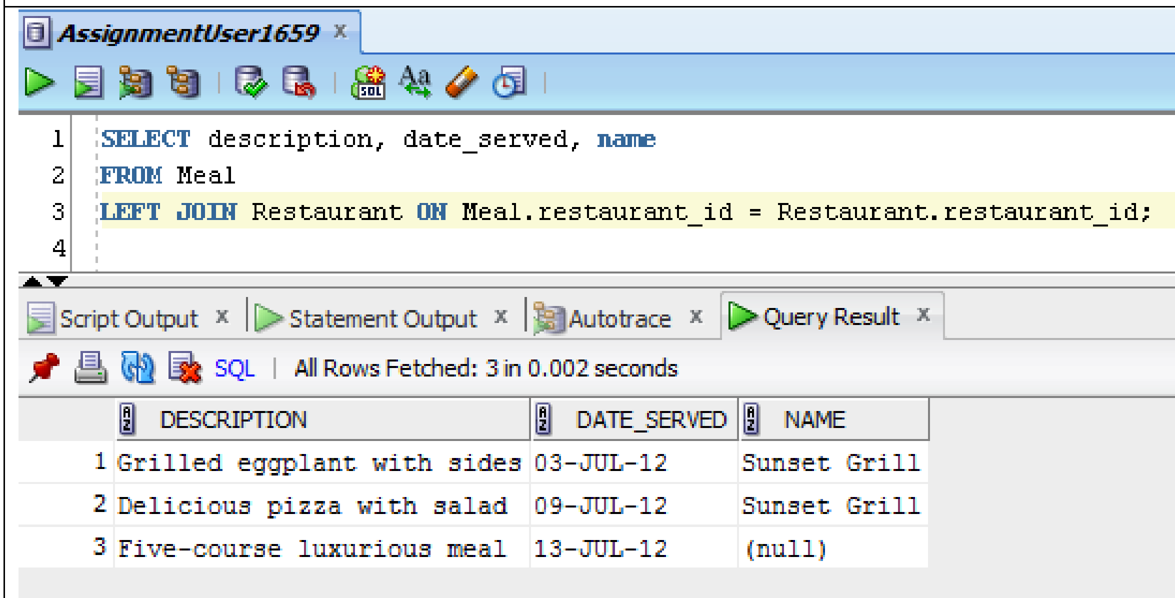
Do you see the difference between this request and the prior request? The prior request only asks for meals that were served at restaurants. The request in this step however is asking for all meals, whether or not they were served at a restaurant. We will fulfill this request by using a left join. We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

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

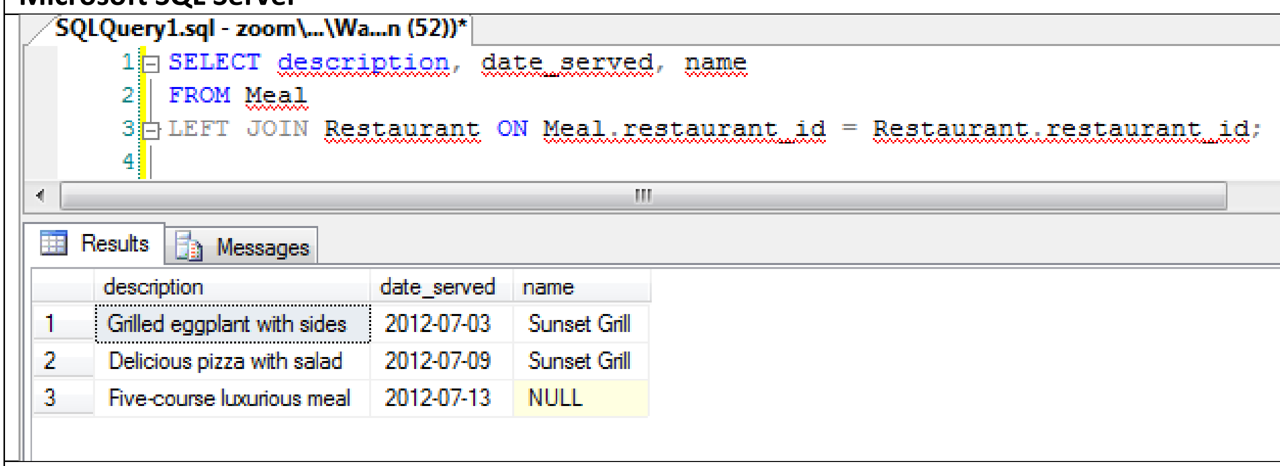
What does this LEFT keyword do for us? It instructs the RDBMS to retrieve all rows that match the join condition, *and also* retrieve rows from the first table listed that do not match the join condition. In our example, the Meal table is the first table listed, and so any meals that do not have restaurants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

Below is a sample screenshot of the command execution in each RDBMS.

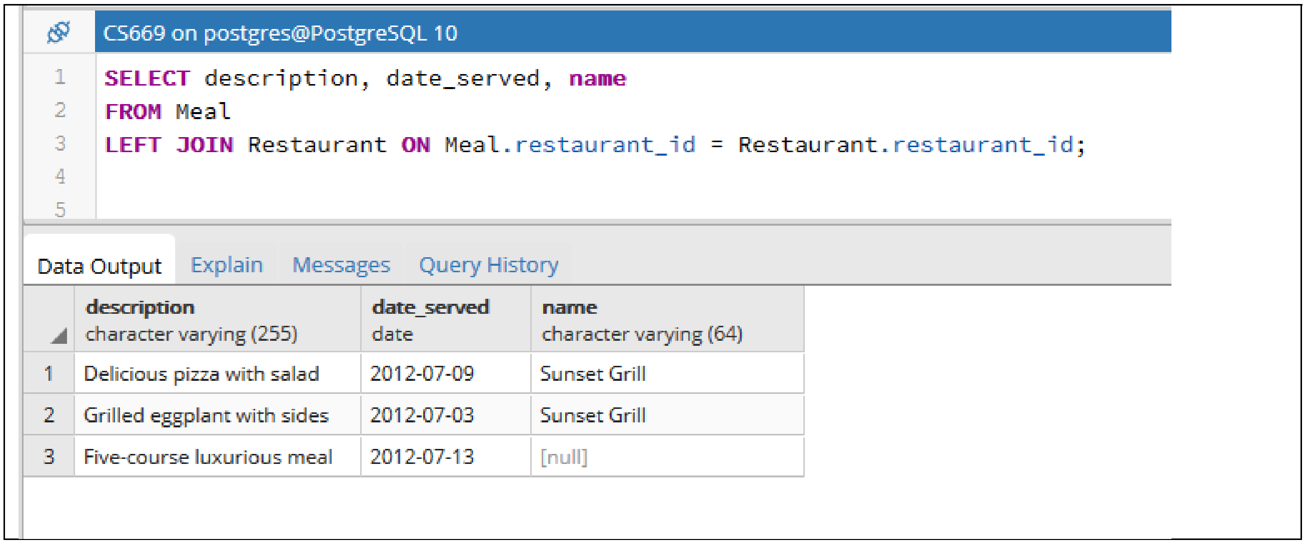
Oracle



Microsoft SQL Server



PostgreSQL



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.

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.

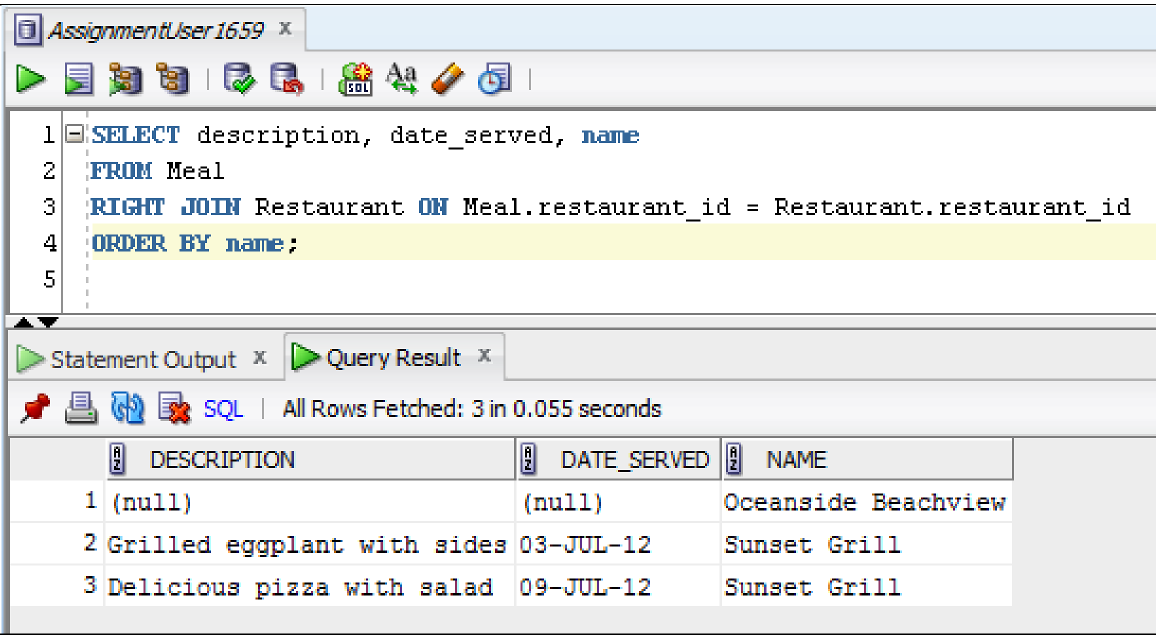
```
SELECT description, date_served, name
FROM Meal
RIGHT JOIN Restaurant ON Meal.restaurant_id = Restaurant.restaurant_id
ORDER BY name;
```

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

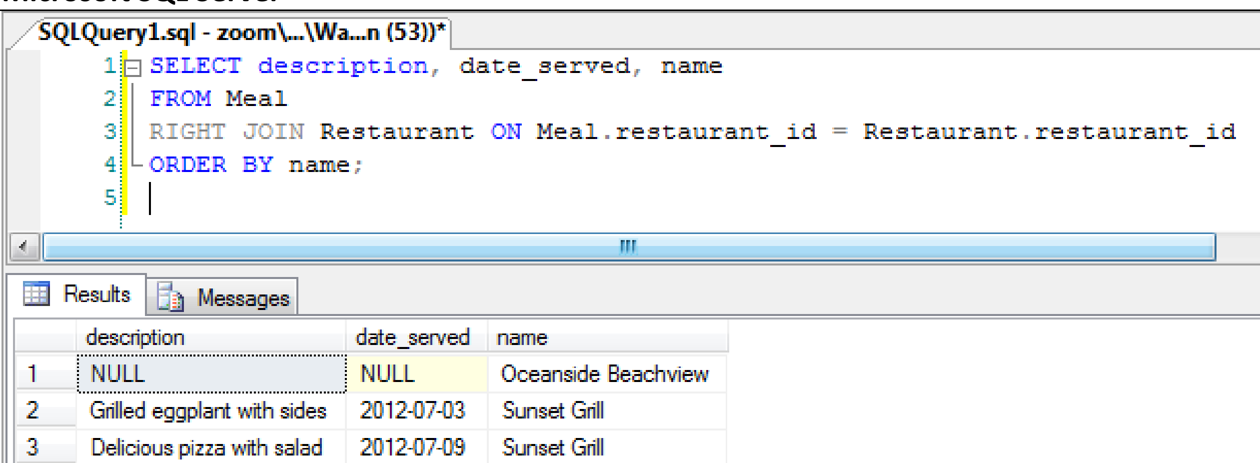
The ORDER BY construct tells the RDBMS to sort the results based upon the values in a column or group of columns. A comma-separated list of column names follows the ORDER BY keywords. In our example, we only wanted to sort by the restaurant name, and so we have listed only one column.

Below is a sample screenshot of the command execution in each RDBMS.

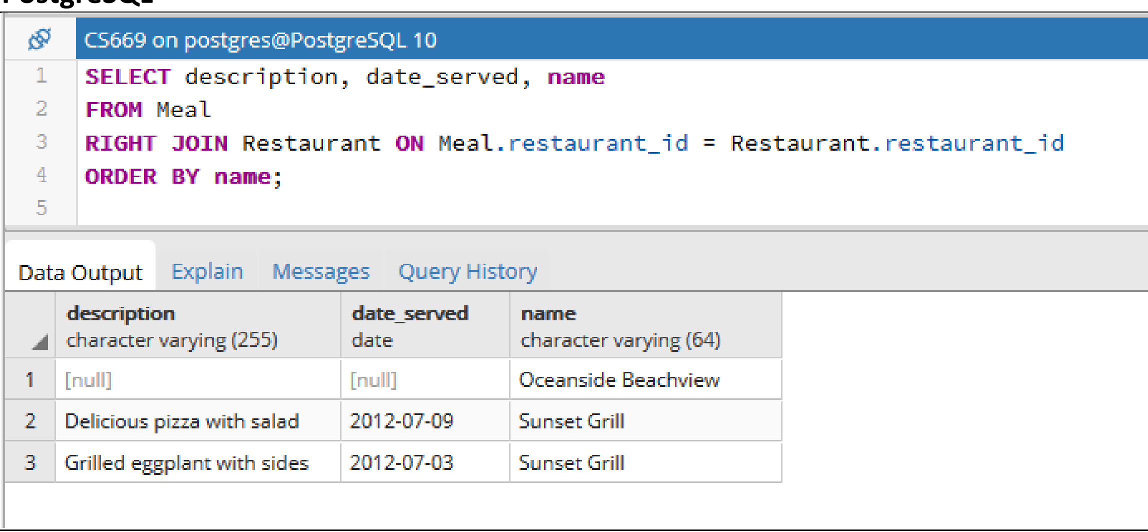
Oracle



Microsoft SQL Server



PostgreSQL



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.

We can also retrieve matching rows and rows that do not match in both tables, which is a combination of what we did in prior SQL commands. 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 an 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 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.

Below is a sample screenshot of the command execution in each RDBMS.

Oracle

AssignmentUser1659

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

Statement Output | Query...

SQL | All Rows Fetched: 4 in 0.015 seconds

	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

Microsoft SQL Server

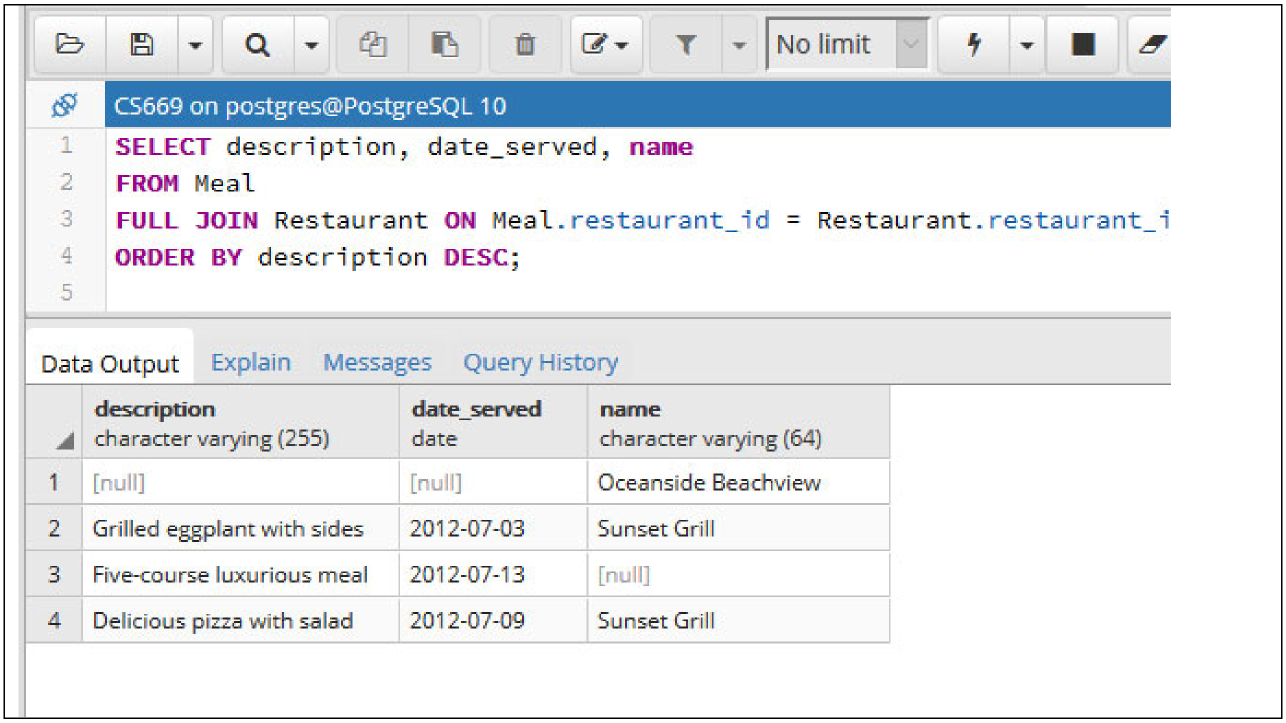
SQLQuery1.sql - zoom\...\Wa...n (53))*

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

Results | Messages

	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

PostgreSQL




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

	description character varying (255)	date_served date	name character varying (64)
1	[null]	[null]	Oceanside Beachview
2	Grilled eggplant with sides	2012-07-03	Sunset Grill
3	Five-course luxurious meal	2012-07-13	[null]
4	Delicious pizza with salad	2012-07-09	Sunset Grill

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 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 PostgreSQL, 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.

Step 5 – Listing All from One Table

The explanation for Step 4 lists examples for all four types of joins. Consider which of those join types satisfy this use case. Also, do not forget the row ordering specification.

Step 6 – Listing All from Another Table

You may review the explanation for Step 4 to know how to complete this step.

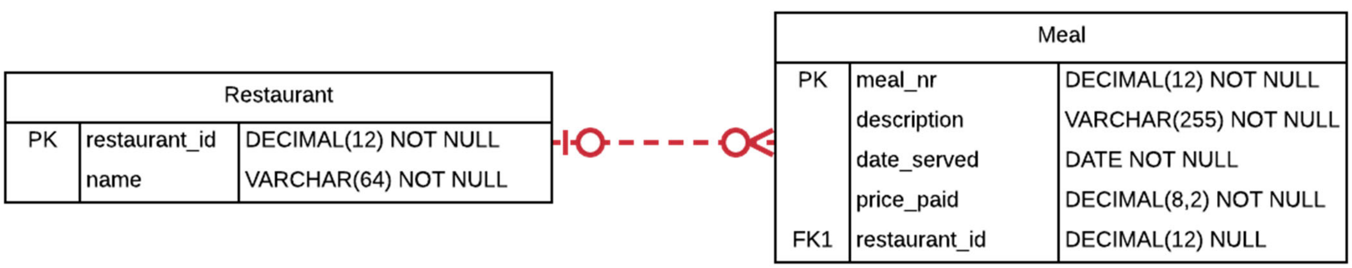
Step 7 – Listing All From Both Tables

Review the explanation for Step 4 to know how to complete this step.

Step 8 – Formatting as Money

If a simple question is asked of our data, such as "What are the names of the restaurants?", it may be tempting to list the names out from memory, eyeball the names in the Restaurant table, or even maintain our own spreadsheet with the names included. After all, why go through the trouble of using SQL to do so? Perhaps the most significant reason is that when we are developing a database and surrounding I.T. system, *we are not actually asking for a single answer, but are asking for logic that is capable of answering that same question repeatedly for the entire life of the database,* whether it is years or decades. Answering the question just once is not useful given our goal. We want to know the names of all restaurants today, tomorrow, next year, and ten years down the road, even if more restaurants are established or others close. A second reason is that *our goal is to give an I.T. system the ability to answer this question, not a human being.* I.T. systems and surrounding databases are all about automation, performing repeated tasks much more quickly than human beings, freeing us from the responsibility of doing tedious tasks ourselves. Obviously, I.T. systems are not capable of eyeballing, and need formal SQL logic in order to access relational databases. Lastly, *many real-world relational database schemas contain too many tables, relationships, and values for us to practically keep track them ourselves,* so even if we want to answer a question ourselves, we still need to use SQL to obtain the values we need from the database. We develop SQL queries to answer our data questions in today's world.

In order to show you examples similar to the steps you need to complete this step, we augment the previously used Restaurant and Meal schema by adding a price_paid column to Meal. The column is a record of how much each meal cost the customer. The updated schema looks as listed below.



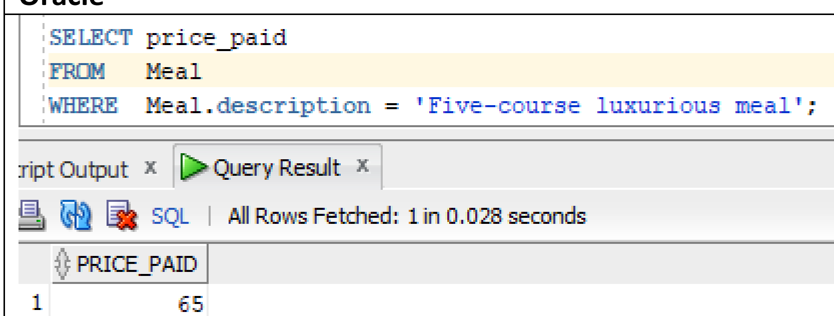
We re-create the schema as listed below to include the price_paid table.

```
DROP TABLE Meal;
DROP TABLE Restaurant;

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,
price_paid DECIMAL(8,2) NOT NULL,
restaurant_id DECIMAL(12)
);

ALTER TABLE Meal
ADD CONSTRAINT meal_pk
PRIMARY KEY(meal_nr);

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

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, price_paid, restaurant_id)
VALUES (101, 'Grilled eggplant with sides', CAST('03-Jul-2012' AS DATE), 20.99, 31);
INSERT INTO Meal (meal_nr, description, date_served, price_paid, restaurant_id)
VALUES(102, 'Delicious pizza with salad', CAST('09-Jul-2012' AS DATE), 14.50, 31);
INSERT INTO Meal (meal_nr, description, date_served, price_paid, restaurant_id)
VALUES(103, 'Five-course luxurious meal', CAST('13-Jul-2012' AS DATE), 65.00, NULL);
```

This SQL, with the exception of the price_paid column, is identical to the SQL in section 1. The same restaurants and meals are used in this section.

If we, for example, select the price_paid for the five-course luxurious meal, it looks as follows in the three SQL clients.

Oracle



Microsoft SQL Server

```sql
SELECT price_paid
FROM   Meal
WHERE  Meal.description = 'Five-course luxurious meal';
```

00 %

Results | Messages

	price_paid
1	65.00

PostgreSQL

```sql
SELECT price_paid
FROM   Meal
WHERE  Meal.description = 'Five-course luxurious meal';
```

Data Output | Explain | Messages | Notifications | Query History

	price_paid numeric (8,2)
1	65.00

You may observe two things here. First, Oracle displays the price as "65" with no decimal points, while SQL Server and Postgres display the price as "65.00" with two decimal points. Second, even though this is a monetary value, none of the SQL clients display the value with the monetary symbol, $, assuming we're using U.S. dollars.

SQL clients are sophisticated applications, but are not so sophisticated that they always display values in the format we expect. Listing out 2 decimal points is nothing more than the default for the SQL Server Management Studio (SSMS) and pgAdmin clients when displaying numbers that have 2 decimal points in the datatype. The default for the Oracle SQL Developer client is to remove trailing 0 digits that occur to the right of the decimal point, notwithstanding the fact that the datatype supports two decimal points. Other SQL clients may have different defaults for SQL Server and other databases. SQL clients oftentimes display a value from a basic SQL query in a nonconventional format.

The discrepancy between the value displayed and the value we conventionally expect shows us that there is something more involved. There are actually four significant components that determine how a value is displayed -- the raw value stored in a database table, manipulations on the value performed by the SQL query, formatting constructs applied in the SQL query, and how the particular SQL client displays the value. These components all collectively determine how a value will be displayed to us when we execute SQL in a SQL client. There is a tremendous amount of depth for each of these components, and while we will not be able to cover every detail, it is important that we explore each in more depth. Doing so will help give you the ability to craft queries that display values in whatever format you deem appropriate. Controlling how a value is displayed is an intricate subject.

Component 1: Raw Values

Different kinds of data have different limits that present a challenge for database designers as they consider how to store the data. Some kinds of values have no theoretical limit, for example, fractions that result in infinitely repeating decimals. How do we store these infinitely long values? Some kinds of values have theoretical limits, but we cannot determine them. For example, how would we determine how much storage we need for the text of the lengthiest book in the world? Even if we determine the lengthiest book known, we could always discover a new, lengthier book, or someone could write a lengthier one in the future. Some kinds of values have known limits, but their limits are too big for practical storage. For example, a business may know all websites visited by its employees while at work in the prior year, but would it practical for the business to store the full content of every website every time it is visited? We need to think about these kinds of limits before we store the data in our database.

All values stored in a relational database column have size limits, and interestingly datatypes, which we learned in prior labs determine the set of legal values for database columns, also establish size limits. All exact numeric datatypes have a precision, which is the maximum number of digits allowed in the number, and a scale, which is the maximum number of digits allowed to the right of the decimal point. For example, if we want to store the number 12.34, we need a precision of at least 4 since there are 4 digits in total, and a scale of at least 2 since there are 2 digits to the right of the decimal point. All inexact numeric datatypes used for storing fractional numbers are constrained by a maximum number of bytes. All text datatypes have a maximum limit of either characters or bytes. Date and time datatypes have limits on the number of digits used to store fractions of a second. Data stored in a relational database is limited in size in various ways as specified in the datatype for each table column.

The type and limitations of the datatype for a value is one component that determines how it will be displayed in a SQL client. For example, we observed the SQL Server and Postgres list out all of the decimal points given in the dastatype by default. If we use a DECIMAL(8,3), they will both list out three decimal points; if we use a DECIMAL(8,4), they will both list out four decimal points, assuming no other manipulations occur.

Component 2: Manipulations on the Value

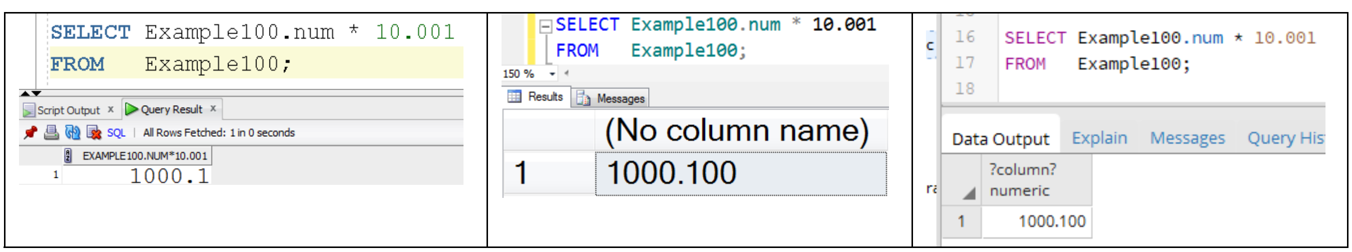
A SQL query always gives us a value in the same form as it is stored in the database, right? Wrong! Many SQL queries manipulate values, and the results have changes in the datatypes or size limits. For example, imagine we start with the number “100” in a column like so:

```
CREATE TABLE Example100 (num DECIMAL(3));
INSERT INTO Example100 (num) VALUES (100);
```

Further imagine that we multiply the value times 10 and 1 thousandth like so:

```
SELECT Example100.num * 10.001
FROM   Example100;
```

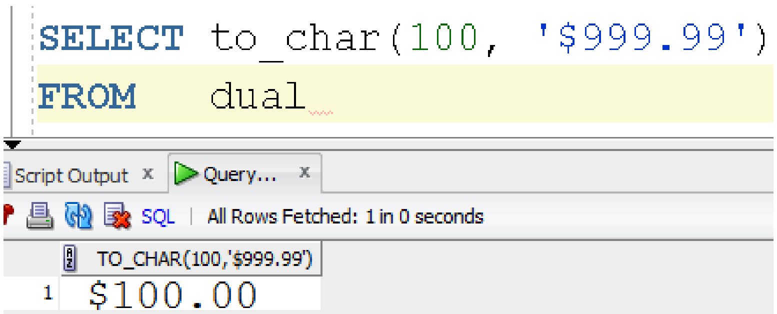
The number “100” is stored as a DECIMAL(3) datatype, which means that it supports 3 digits total with no digits allowed to the right of the decimal point. So should we expect that the result is also of that same form? Let’s find out! The screenshots below show the query executed in Oracle, SQL Server and PostgreSQL, respectively.



Whoa! The result in Oracle has 5 digits total, with four the left of the decimal point and one to the right, and the result in SQL Server and PostgreSQL both have seven digits total, with four to the left of the decimal point and three to the right of the decimal point. This looks more like a DECIMAL(5,1) or DECIMAL(7,3) than a DECIMAL(4). As demonstrated, a SQL query can yield a result in a form different from that of the raw stored value when manipulations are applied to that value.

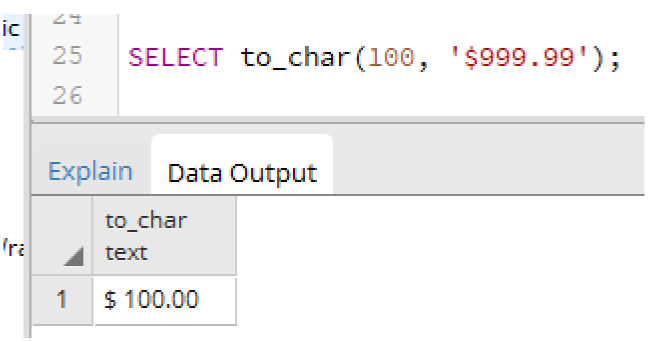
Component 3: Formatting Constructs

We can use formatting constructs to explicitly form the structure and appearance of the results. These constructs typically come in the form of formatting functions, which tend to be vendor specific; formatting functions for Oracle tend not to be available in SQL Server and vice versa. For example, suppose we want to display the raw number "100" as a currency, indicating it is 100 dollars. In Oracle, we can use the **`to_char`** function, as shown in the screenshot below.



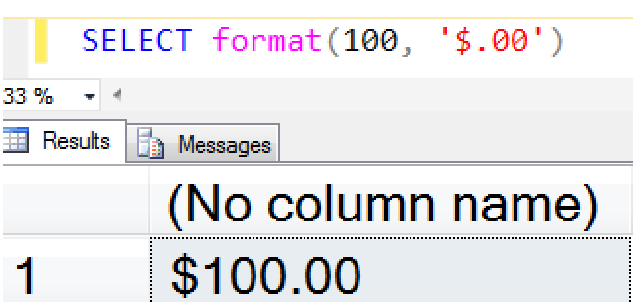
The first argument to **`to_char`** is the number we want to format, in this case, 100, and the second argument is a series of characters that describes exactly how we want to format the number, which Oracle documentation refers to as a *format model* (Oracle, 2016). In this case, the "$" symbol indicates we want to prefix the number with our currency symbol, the "." symbol indicates we want to make use of a decimal point, the initial three "9" digits indicate we want to display up to three digits to the left of the decimal point, and last two "9" digits indicate we want to display two digits to the right of the decimal point.

The formatting for PostgreSQL is similar to Oracle as seen below.



In this case like with Oracle, the "$" symbol indicates we want to prefix the number with our currency symbol, the "." symbol indicates we want to make use of a decimal point, the initial three "9" digits indicate we want to display up to three digits to the left of the decimal point, and last two "9" digits indicate we want to display two digits to the right of the decimal point.

In SQL Server, we can use the **`format`** function, as shown in the screenshot below.



Similar to Oracle's **`to_char`** function, the first argument to **`format`** is the number and the second argument is a series of characters that describes how we want to format the number (Microsoft, 2016b). The "$" symbol indicates the dollar sign symbol should prefix the result, the "." symbol indicates we want to make use of a decimal point, and the two "0" digits indicate we want to display two digits to the right of the decimal point. The results for Oracle, SQL server, and Postgres are conventionally what we expect to see for a currency – $100.00. Note that your database's region settings determine what currency symbol, digit group separator symbol, and decimal-point indicator symbol your database will use when formatting a currency, so your results may vary if your region is not the United States. Formatting functions are useful for displaying results in patterns human beings conventionally expect.

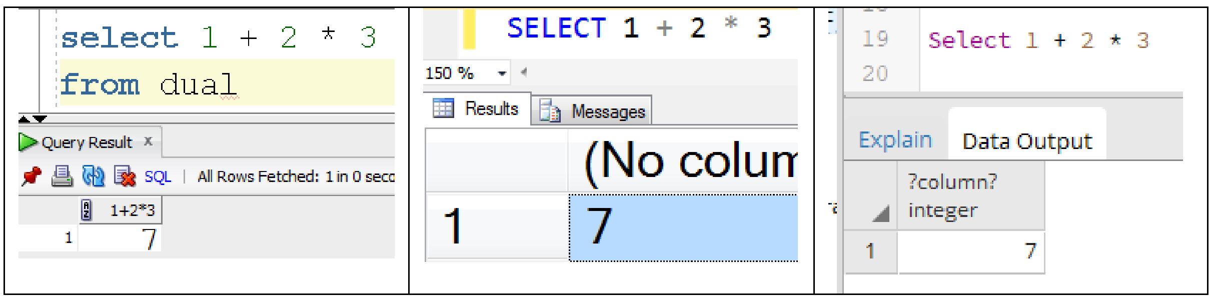
Component 4: SQL Client Choices
Different SQL clients may display the same results differently. Primarily, these discrepancies occur because each team or vendor creating its SQL client decides independently how results should be displayed, and different teams make different choices. There is no requirement or enforcement that different teams must make the same choices. This holds true even for SQL clients for the *same* database. For example, Oracle SQL Developer may display a result one way, the Toad client another, the PL/SQL Developer client another, all for the same Oracle instance! PL/SQL Developer, for example, gives you the option to "zoom in" on any particular value, where you can view the value as text, in binary as hexadecimal, in XML, and even as a picture (assuming the result is a picture), while other clients may not give that option. The same goes for SQL Server or Postgres, where Toad, SSMS, pgAdmin, DataGrip, for example, may display results differently. The key observation here is that *each SQL client is an application*, and is not bound to mechanically display a value in its raw form, but the client displays it in a way thought to be most useful to us. If we were to author our own application, such as a series of web pages that display values from our database, we would need to make our own choices as to how these values would be displayed. SQL clients are merely applications developed by application teams, and these teams decide how results are displayed.

Armed with this knowledge, you know enough to complete this step.

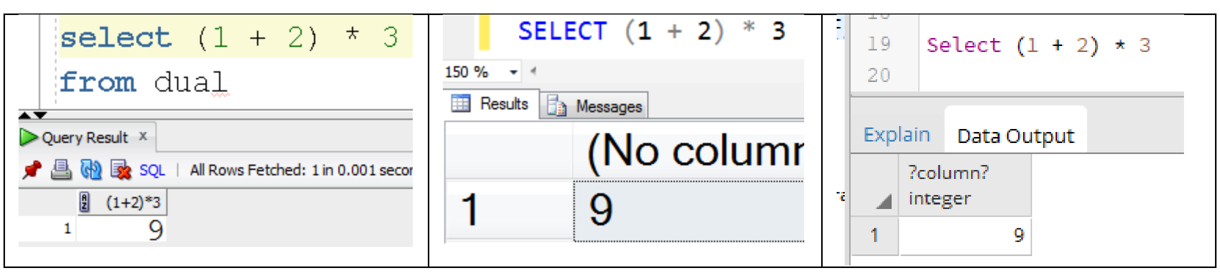
Step 9 – Using Expressions

Manipulations on a value in a SQL query affect how the results are displayed, and these manipulations are defined more technically as *expressions* in a SQL query. Expressions consist of *operands*, which are values from the database or hard-coded values, and *operators*, which are SQL keywords or symbols that derive results from one or more operands in a predefined way. For example, the simple expression “Example100.num * 10.001” we used earlier has two operands – Example100.num and 10.001 – and makes use of the “*” operator, which derives a new result by multiplying the values of two operands. The act of deriving the new result is termed an *operation,* so we say that operators perform operations on operands. Expressions give us the ability to transform raw database values in a variety of ways

We expect the expression “Example100.num * 10.001” to multiply the values together, but we may not expect that the expression “1 + 2 * 3” gives us a result of 7, instead of a result of 9. Look at the side-by-side screenshots from Oracle, SQL Server and PostgreSQL, respectively, below.



On the surface, we would think that 1 + 2 = 3, then 3 * 3 = 9, so why is the result 7? It is important to note the order in which the operations occur is strictly defined by the DBMS’ *operator precedence*, which is set of rules that indicates which operations will be evaluated before other operations. Each DBMS evaluate multiplication and division operators before addition and subtraction operators (Microsoft, 2016a; Oracle, 2015; The PostgreSQL Global Development Group, 2018), so the expression “1 + 2 * 3” yields 7 in each DBMS, rather than 9, because the multiplication operation occurs before the addition operation. In this example, 2 * 3 = 6, then 6 + 1 = 7. Parentheses can be used in an expression to override operator precedence. If we use parentheses judiciously in the expression, changing it to (1 + 2) * 3, we can change the result to 9, as shown in the results below:

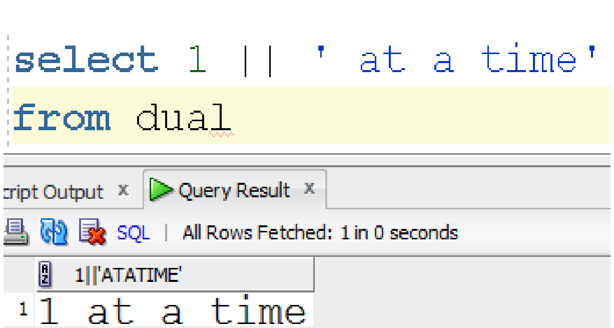


Operator precedence is applied *to all expressions in all SQL queries without variation*, so within the context of a particular DBMS and version, we know that the same set of rules applies everywhere. Each DBMS follows a strict set of rules to determine the results of an expression, and we can use these rules judiciously to ensure we obtain the results we expect.

You learned in Step 8 how to format a price to appear as the U.S. dollar currency. To complete this step, you can combine that knowledge with the knowledge you learned in this step.

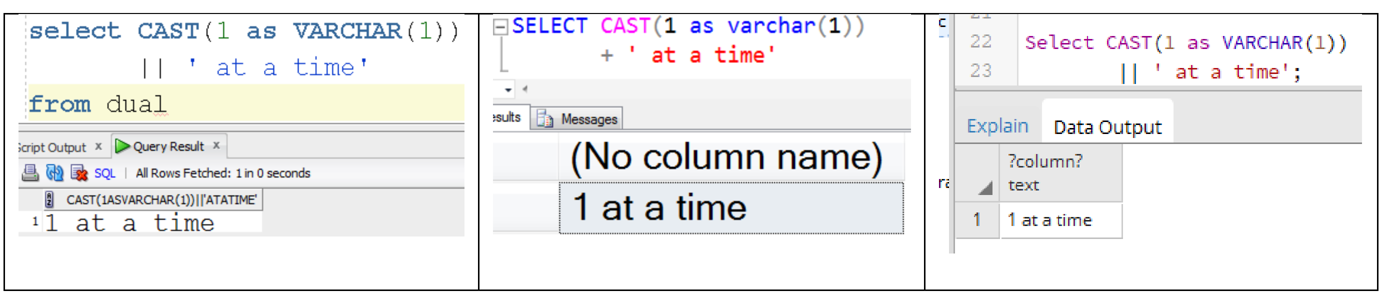
Step 10 – Advanced Formatting

While operator precedence determines the order of operations in an expression, *datatype precedence* determines the datatype that results from an expression. The result of each expression has a datatype, such as a VARCHAR, DATE, and so on, and this helps determine how the SQL client will display the result. For example, if we add an integer value and a floating-point value, such as "1 + 2.33", the datatype precedence of many modern relational DBMS will cause the datatype of that expression to be a floating point number rather than an integer. This makes sense, because if the result were an integer, the result could only be 3 if rounded down, or 4 if rounded up. But a floating point number datatype supports the expected result of 3.33. Many modern relational DBMS also support conversion to character values. For example, the expression "1 || ' at a time'" in oracle will produce a result of "1 at a time", as shown below.



Even though 1 is an integer, ' at a time' is a character string, and Oracle's datatype precedence rules dictate the result to be a character string. Modern relational DBMS have strict sets of rules that determine both the result and the datatype of the result from expressions.

Carefully combining operands with different datatypes in an expression will get you the resulting datatype you want, but using *explicit* datatype conversions is more production-worthy and maintainable. When we simply use operands and operators, we are relying on *implicit* datatype conversion. Even when that works, it is less clear to the reader how it is working, and implicit conversions are subject to change when the database is upgraded. Thankfully we can also use specific constructs to explicitly convert the datatypes. For example, if we want to concatenate an integer with a character sequence, we can explicitly cast the integer to a character sequence using the CAST function. Look at the equivalent examples below in Oracle, SQL Server and PostgreSQL respectively.



Notice that the integer 1 was explicitly cast to a Varchar datatype, and that Varchar is then concatenated to the character sequence " at a time". Oracle and PostgreSQL use the characters || as a concatenation operator, while SQL Server uses + as a concatenation operator. With this expression, we do not rely not on implicit datatype conversion. There is much detail involved with datatype conversion, so much so that we could go on exploring it with this entire lab, or even an entire book, but so that we do not lose the focus of this lab, let us move on. Suffice it to say, expressions in modern relational DBMS support operands that have different datatypes, and we explicitly or implicitly convert between datatypes to arrive at a final datatype for the expression's result.

Carefully combining the concepts you have learned thus far in the lab with your new knowledge of string concatenation and operator precedence will enable you to get the results you want.

Step 11 – Evaluating Boolean Expressions

Boolean expressions are used in many programming languages, including SQL, as a powerful and flexible tool to specify what matches and what does not match certain conditions. The result of a Boolean expression can only be two values – true or false. If true, it matches and condition, and otherwise is does not. All Boolean operators in a Boolean expression expect values to be either true or false, and always reduce to a true or false value. There are three primary Boolean operators that we work with in such expressions – AND, OR, and NOT.

AND Boolean Operator
The AND Boolean operator is a way to ask if two values are true, and can be summarized as "Is condition 1 true AND condition 2 true?" There are four possibilities:

- true AND true = true
- true AND false = false
- false AND true = false
- false and false = false

The above list shows that the results of an AND can only be true if both operands are true. If any of the operands are false, the result is false.

Let's take a casual example. Imagine we have two true comparisons, like so:
Does 5 = 5? ➔ true
Does 10 = 10? ➔ true

We can combine them with an AND to achieve a true result:
Does 5 = 5 AND does 10 = 10? ➔ true

If we had one or more false conditions, then that would no longer be the case:
Does 5 = 5? ➔ true
Does 17 = 13? ➔ false

Combining them would result in:
Does 5 = 5 AND does 17 = 13? ➔ false

The last statement is false because although one part of the statement is true, the other part is false, and the AND operator only accepts both as true.

In summary, we use AND when we want to require that both conditions be true.

OR Boolean Operator
The OR Boolean operator is a way to ask if at least one of two values is true, and can be summarized as "Is condition 1 true OR is condition 2 true?" There are four possibilities:

- true OR true = true
- true OR false = true
- false OR true = true
- false OR false = false

The above list shows that the results of OR are true if any or both of the conditions are true. The only way the result is false is if both conditions are false.

Let's take a casual example. Imagine we have the following conditions.
Does 5 = 5? ➔ true
Does 17 = 13? ➔ false

The OR operator would indicate true with the two statements combined:
Does 5 = 5 OR does 17 = 13? ➔ true

The reason is, the combined statement means that either condition can be true. Since 5 equals 5, it does not matter that 17 does not equal 13. However, if we give two false statements, such as:
Does 5 = 7? ➔ false
Does 17 = 13? ➔ false

Then the OR of them would also be false:
Does 5 = 7 OR does 17 = 13? ➔ false

The reason is, if both statements are false, the OR indicate false because none of them were true.

In summary, we use OR when satisfying either condition is acceptable. As long as at least one of them is satisfied, the OR operator is satisfied.

NOT Boolean Operator
The NOT operator quite simply returns true if a statement is false (and likewise, false if the statement is true). It can be summarized as "Is this NOT true?" The NOT operator only works on a single operand, so there are only two possibilities:

- NOT true = false
- NOT false = true

The above list shows that NOT simply switches the result so that false becomes true and true becomes false.

Using a casual example, the statement "Does 5 = 5?" is true. So if we use NOT, such as "NOT does 5 = 5?", the answer is false. Likewise, if we use a false statement, such as "Does 17 = 13?", then add a NOT, then "NOT does 17 = 13?" is true.

In summary, we use NOT when we want a condition false rather than true.

Nesting Operations
Just as with other operations, Boolean operations can be nested using parentheses. For example, we could do something like this:

NOT (condition1 AND condition2)

In plain English, that statement means "Condition1 and Condition2 must not both be true". For example, imagine the conditions were as follows:

NOT (first_name = 'Bill' AND last_name = 'Glass')

This would exclude any people whose first name is Bill and the last name is Glass. Because there are several operators, NOT, AND, and =, we can work through it in several steps:

Step	Reduction	Explanation
Beginning	NOT (first_name = ‘Bill’ AND last_name = ‘Glass’)	We’re starting with base expression, and the person named Bill Glass.
Step 1	NOT (true AND true)	Because first_name = ‘Bill’ and last_name = ‘Glass’, each of them individually evaluates to true.
Step 2	NOT (true)	Because true AND true is true, we reduce the inner expression to true.
Step 3	false	Because NOT(true) is false, the final result is false. In other words, the person named Bill Glass is excluded based upon this expression.

On the other hand, the expression would *not* exclude Jane Glass, Bill Joker, or Elaina Gunther, because none of these satisfy the logic. We can work through the logic with “Jane Glass” step-by-step.

Step	Reduction	Explanation
Beginning	NOT (first_name = ‘Bill’ AND last_name = ‘Glass’)	We’re starting with base expression, and the person named Jane Glass.
Step 1	NOT (false AND true)	first_name = ‘Jane’ evaluates to false, and last_name = ‘Glass’ evaluates to true.
Step 2	NOT (false)	Because false AND true is false, we reduce the inner expression to false.
Step 3	true	Because NOT(false) is true, the final result is true. In other words, the person named Jane Glass is included based upon this expression.

As another example, we could use:

(condition1 AND condition2) OR (condition3 AND condition4)

Imagine that condition1 is false, and the rest of the conditions are true. Working through the logic would be like this:

Step	Reduction	Explanation
Beginning	(condition1 AND condition2) OR (condition3 AND condition4)	We’re starting with base expression, with condition1 as false, and the rest as true.
Step 1	(false AND true) OR (true AND true)	We first replace the conditions with their results.
Step 2	(false) OR (true)	Because false AND true is false, and true AND true is true, we reduce the inner expressions to their results.
Step 3	true	Because false OR true is true, the final result is true.

Using Boolean expressions, AND, OR, and NOT can be combined and nested in various ways, as needed, to specify conditions that must be satisfied.

Step 12 – Using Boolean Expressions in Queries

Booleans expressions are not complete without comparison operators, and you'll need to make use of these to address this step. Comparison operators allow values to be compared with other values. Each comparison operator either gives a match by yielding a true result, or indicates it does not match by yielding a false result. Several important comparison operators are described below.

Symbol	**Meaning**	**Example**
=	The = operator yields true if the two values match exactly, and false otherwise.	The expression *last_name = 'Glass'* would match any row where the last name is "Glass".
>	The > operator yields true if the left-hand value is greater than the right-hand value, and false otherwise.	The expression *birth_date > '2010-04-01'* would match any row where the birthdate is greater than April 1st, 2010. The expression *number_people > 50* would match any row where the number of people is greater than 50.
<	The < operator yields true if the left-hand value is less than the right-hand value, and false otherwise.	The expression *birth_date < '2010-04-01'* would match any row where the birthdate is less than April 1st, 2010. The expression *number_people < 50* would match any row where the number of people is less than 50.
>=	The >= operator yields true if the left-hand value is greater than or equal to the right-hand value, and false otherwise.	The expression *birth_date >= '2010-04-01'* would match any row where the birthdate is greater than April 1st, 2010, or exactly April 1st, 2010. The expression *number_people >= 50* would match any row where the number of people is greater than 50, or exactly 50.
<=	The <= operator yields true if the left-hand value is less than or equal to the right-hand value, and false otherwise.	The expression *birth_date <= '2010-04-01'* would match any row where the birthdate is less than April 1st, 2010, or exactly April 1st, 2010. The expression *number_people <= 50* would match any row where

		the number of people is less than 50, or exactly 50.
<>	The <> operator yields true if the left-hand value is not equal to the right-hand value, and false otherwise.	The expression *last_name* <> *'Glass'* would match any row where the last name is not "Glass".

The comparison operators above allow us to compare values in our database with other values. These comparisons are combined with the Boolean operators AND, OR, and NOT to support potentially complex conditions that must be satisfied.

Let us demonstrate with an example by using a condition with the Product table for a hardware store below.

Product		
PK	product_id	DECIMAL(12) NOT NULL
	name	VARCHAR(32) NOT NULL
	length_inches	DECIMAL(4,1) NOT NULL
	height_inches	DECIMAL(4,1) NOT NULL
	width_inches	DECIMAL(4,1) NOT NULL
	launch_date	DATE NOT NULL

Besides a primary key, this table stores each product's name, length, width, and height in inches, and the date it was launched. Four products exist in the table as listed below.

product_id	name	length_inches	height_inches	width_inches	launch_date
1	Lightbulb	2	4	2	4/4/2021
2	Lawn Mower	70	50	41	3/1/2021
3	Shovel	45	11	15	6/1/2020
4	Wrench	6	1	1	5/15/2021

The SQL to create this table, along with the SQL to insert a few rows, is below.

Code: Creating Products

```
CREATE TABLE Product (
product_id DECIMAL(12) NOT NULL PRIMARY KEY,
name VARCHAR(32) NOT NULL,
length_inches DECIMAL(4,1) NOT NULL,
height_inches DECIMAL(4,1) NOT NULL,
width_inches DECIMAL(4,1) NOT NULL,
launch_date DATE NOT NULL);

INSERT INTO Product(product_id, name, length_inches, height_inches, width_inches, launch_date)
VALUES(1, 'Lightbulb', 2, 4, 2, CAST('04-APR-2021' AS DATE));
INSERT INTO Product(product_id, name, length_inches, height_inches, width_inches, launch_date)
VALUES(2, 'Lawn Mower', 70, 50, 41, CAST('01-MAR-2021' AS DATE));
INSERT INTO Product(product_id, name, length_inches, height_inches, width_inches, launch_date)
VALUES(3, 'Shovel', 45, 11, 15, CAST('01-JUN-2020' AS DATE));
INSERT INTO Product(product_id, name, length_inches, height_inches, width_inches, launch_date)
VALUES(4, 'Wrench', 6, 1, 1, CAST('15-MAY-2021' AS DATE));
```

Imagine that the hardware store needs to adjust its shipping charges. Shipping larger items to customers has become more expensive, yet the store does not want to discourage repeat purchases on longstanding products. To accommodate this, the store will charge more for shipping products that have a size of at least 4 cubic feet and were launched in the year 2021 or after. Products launched before 2021 are excluded. Shipping lightbulbs has also become more expensive because they must be packed in special protective containers, so the store will charge more for those as well.

To apply the extra shipping charges, a reasonable question the store manager could ask is, "Find all products eligible for extra shipping charges – all oversized products that were launched in the year 2021 or after, as well as lightbulbs." Armed with the knowledge of advanced Boolean expressions, we can answer this question with SQL! First, let's look at the query itself, below.

Code: Listing Eligible Products

```
SELECT Product.*,
       (length_inches*height_inches*width_inches)/1728 AS cubic_feet
FROM   Product
WHERE  name = 'Lightbulb' OR
       ((length_inches*height_inches*width_inches)/1728 >= 4 AND
         launch_date >= CAST('01-JAN-2021' AS DATE))
```

Let's examine the WHERE clause part by part. The first part:

```
name = 'Lightbulb'
```

is straightforward. Any lightbulb matches regardless of size. Next, `length_inches*height_inches*width_inches` gives the size in cubic inches of the product. One cubic foot is equivalent to 1,728 cubic inches (12 inches*12 inches*12 inches). Therefore, the expression:

```
(length_inches*height_inches*width_inches)/1728
```

gives us the size of the product in cubic feet. Then we compare that to 4 by using `>= 4`. The full expression:

```
(length_inches*height_inches*width_inches)/1728 >= 4
```

matches only products where the size is greater than or equal to 4 cubic feet. This part:

```
launch_date >= CAST('01-JAN-2021' AS DATE)
```

ensures that only the products launched on or after January 1, 2021, are eligible for extra charges.

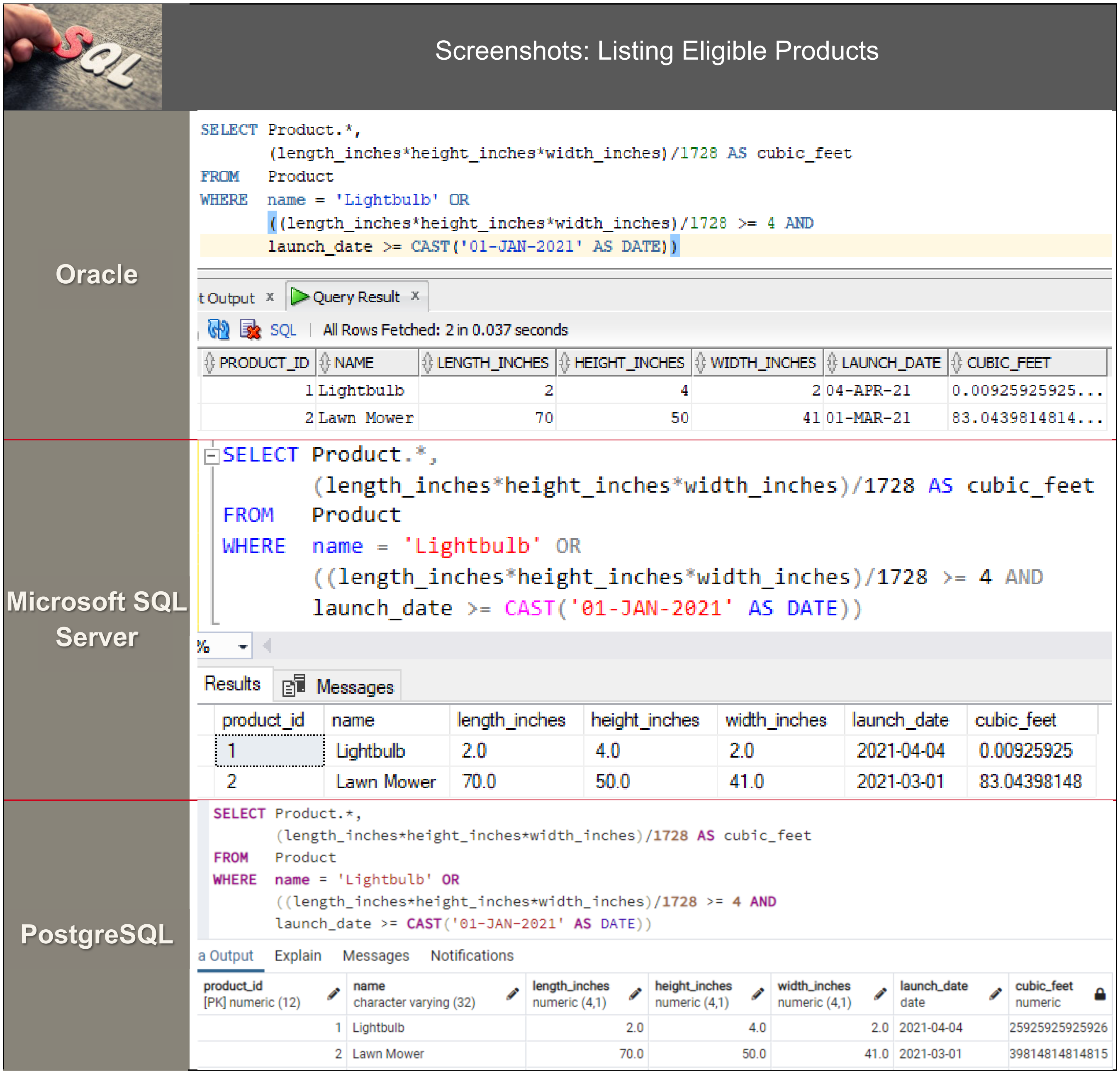
The full expression makes use of the OR and AND Boolean expressions, as well as parentheses:

```
name = 'Lightbulb' OR
((length_inches*height_inches*width_inches)/1728 >= 4 AND
launch_date >= CAST('01-JAN-2021' AS DATE))
```

The statement in plain English is, "Match any products that are lightbulbs, or are greater than 4 cubic feet and launched on or after January 1, 2021." This combination of a variety of operators supports the very specific condition that the hardware store is using to determine which products are eligible for extra shipping charges.

Take note that the entire AND clause was enclosed in parentheses, to make this logic work correctly. That is, being over 4 cubic feet and launching on/after 2021 go together with an AND; a product being a lightbulb is matched to that condition with an OR. *Review the statement and its English counterpart carefully to ensure you understand how it is defined.*

Below are the screenshots of running the query.



Notice that two products are returned in the results, the lightbulb and the lawn mower, out of the full set of products below.

product_id	name	length_inches	height_inches	width_inches	launch_date
1	Lightbulb	2	4	2	4/4/2021
2	Lawn Mower	70	50	41	3/1/2021
3	Shovel	45	11	15	6/1/2020
4	Wrench	6	1	1	5/15/2021

The lawn mower is well over 4 cubic feet and is launched in 2021, and so matches the condition. The lightbulb matches because it is a lightbulb. The wrench does not match, because it is not over 4 cubic feet. Although the shovel is over 4 cubic feet, it wasn't launched prior to 2021, so it is also excluded.

You can see now that Boolean expressions can be used to enforce a wide variety of complex conditions. The right combination of comparison operators, Boolean operators, and parentheses can create just the right logic.

Step 13 – Using Generated Columns

Modern relational databases have the ability to calculate the value in a column automatically based upon values in other columns. These are useful when the calculation is used over and over again in different queries, that is, when many different queries need the value in the column, and we do not want to calculate the same value again and again. These are also useful when the calculation is complex, and the database designer wants to define the calculation as part of the table definition, so that developers do not need to define it themselves. Developers may make mistakes in its definition, or at the very least, spread the logic across many queries, making it more difficult to change later.

Typically from a design perspective, a calculated column naturally fits into the table as if it were just another column. For example, if a Person table has a first_name and last_name column, we could define a calculated full_name column that contains the person's full name, derived from the first_name and last_name columns. When developers use the calculated full_name column in their queries, they think of it as just another column that a Person table naturally has; it doesn't concern them that the column happens to be calculated.

It is easiest to show the syntax for defining a calculated column by example, so we'll start with a simple example. Note that the same syntax can be used for Oracle and SQL Server, and a slightly different syntax must be used for Postgres. We'll start with the Oracle and SQL Server syntax.

Code: Multiply Table in Oracle and SQL Server

```
CREATE TABLE Multiply (
x decimal(4) NOT NULL,
y decimal(4) NOT NULL,
result AS (x * y));
```

Here we've defined a table name Multiply which has two standard columns named *x* and *y* which allow for small numbers to be inserted. The calculated column is named *result*. The syntax *AS (x * y)* instructs the database that the *result* column is not a usual column; rather, it is a calculated column that contains the results of multiplying *x* and *y*.

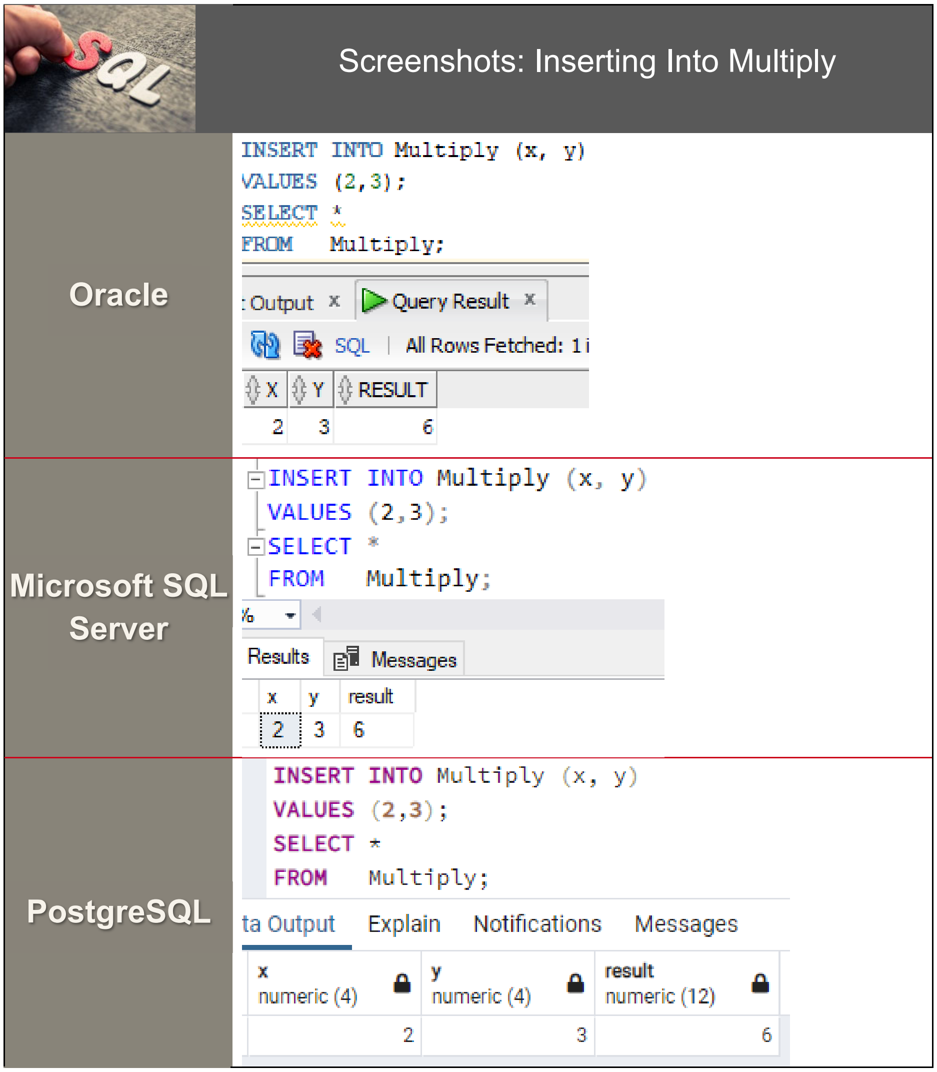
Postgres has a similar syntax, but with a few more necessary keywords, as shown below.

Code: Multiply Table in Postgres

```
CREATE TABLE Multiply (
x decimal(4) NOT NULL,
y decimal(4) NOT NULL,
result decimal(12) GENERATED ALWAYS AS (x * y) STORED);
```

You may first notice that unlike Oracle and SQL Server, Postgres requires that the calculated column have a defined datatype. In this case, we select *decimal(12)* as its datatype to support a larger number. Since *x* and *y* both support up to 4 digits, we chose 12 digits for *result* so it's sufficiently large to store any multiplication between *x* and *y*. The *AS (x * y)* fragment is the same as with Oracle and SQL Server, though it is surrounded with the *GENERATED ALWAYS … STORED* keywords. Postgres allows you to define calculated columns with expressions; it just requires additional syntax.

The below screenshots show the results of inserting a row into the Multiply table.

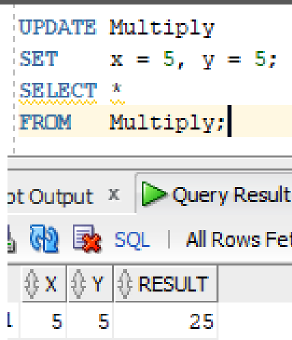


The screenshots illustrate that the values 2 and 3 are inserted into the Multiply table, and the database automatically calculates the result column as 6. The insert statement leaves out the *result* column because the database is responsible for calculating it.

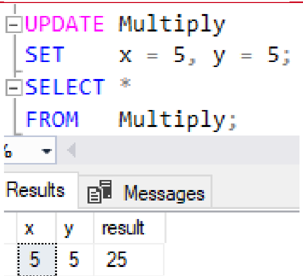
In a similar fashion, updating *x* or *y* will also cause the database to recalculate *result*, as illustrated below.

Screenshots: Updating Multiply

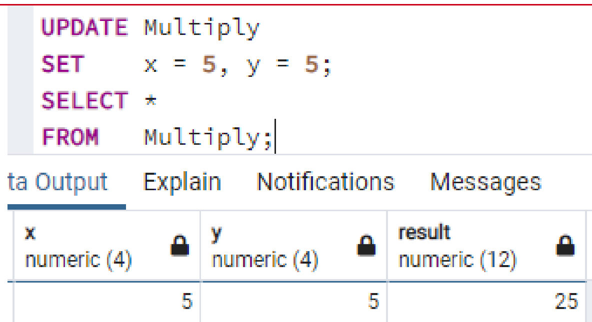
Oracle



Microsoft SQL Server



PostgreSQL



Notice that since *x* and *y* have both been changed to 5, *result* has been automatically changed to 25 by the database.

In this simple example, we used the expression *(x * y)*. Note however that a calculated column may define any expression supported by the database, no matter how complex. Just as with expressions in SQL queries, these may be a combination of Boolean, comparison, math, and other operators, nested in parentheses.

Recall that in #12, the hardware store has somewhat complex logic to determine which products are eligible for extra shipping charges. We dynamically calculate it in a query for #12; however, this is not optimal. It's reasonable that the store would have many queries that would involve use of this calculated column. Surely it would be included in some management reports, in its order fulfillment application, and possibly other areas. Every query would need to redefine this calculation, making it harder to change later, and making it more likely that some queries have it wrong. This is especially true because of the hardcoded number and complex logic. Instead, we can define a calculated column once and use it in as many queries as we need.

A *flag* in a database table is Boolean – true or false. A flag represents a condition, and rows that match that condition have the flag set to true, and rows that do not match the condition have the flag set to false. In this way, rows can be queried simply to determine if they match the condition. We define our calculated column as a flag. Unfortunately, as of this writing, only Postgres supports a Boolean datatype directly. SQL Server supports a Bit datatype which is either 0 or 1, so we can use 1 for true and 0 for false. Oracle has no direct support for Boolean, so it is common to use a decimal(1) datatype and insert a 0 for false and a 1 for true. This use of decimal(1) mirrors that of the SQL Server bit datatype, so the same code can be used for Oracle and SQL Server, and slightly different syntax must be used for Postgres.

To start, let's take a look at our query from #12.

Code: Listing Eligible Products

```
SELECT Product.*,
       (length_inches*height_inches*width_inches)/1728 AS cubic_feet
FROM   Product
WHERE  name = 'Lightbulb' OR
       ((length_inches*height_inches*width_inches)/1728 >= 4 AND
         launch_date >= CAST('01-JAN-2021' AS DATE))
```

We can make the entire condition in the WHERE clause into a flag in the Product table. We'll name it *extra_charge_flag* since it involves extra shipping charges. First, we need to alter the Product table to add this calculated column. The code for Oracle and SQL Server is shown below.

Code: Adding Flag to Product Table in Oracle/SQL Server

```
ALTER TABLE Product
ADD extra_charge_flag AS
(CASE
   WHEN name = 'Lightbulb' OR
        ((length_inches*height_inches*width_inches)/1728 >= 4 AND
          launch_date >= CAST('01-JAN-2021' AS DATE)) THEN 1
   ELSE 0
 END);
```

The *ALTER TABLE Product ADD extra_charge_flag* fragment is the syntax for adding another column to a table. The *AS* keyword tells Oracle and SQL Server that it's a calculated column. The *CASE … END* clause causes a 1 to be returned with a match, and 0 otherwise. In its basic form, *CASE … END* works like this:

```
CASE
   WHEN matching_condition THEN matching_result
   ELSE unmatching_result
 END
```

That is, the matching condition is placed after *WHEN*, and the matching result is placed after *THEN*. The unmatching result is placed after *ELSE*. This is equivalent to an *IF … ELSE* statement in many other languages. If a condition is true, the matching result is returned; otherwise, the unmatching result is returned. In our scenario, the extra shipping charges condition is the matching condition, 1 is the matching result, and 0 is the unmatching result.

The syntax for Postgres is similar, shown below.

Code: Adding Flag to Product Table in Postgres

```
ALTER TABLE Product
ADD extra_charge_flag Boolean GENERATED ALWAYS AS
(CASE
   WHEN name = 'Lightbulb' OR
        ((length_inches*height_inches*width_inches)/1728 >= 4 AND
          launch_date >= CAST('01-JAN-2021' AS DATE)) THEN true
   ELSE false
 END) STORED;
```

The same *CASE … END* clause is used, except that it returns *true* or *false* rather than *1* or *0*. The datatype is declared as *Boolean*, and the extra *GENERATED ALWAYS … STORED* keywords are used as required by Postgres.

Below are screenshots showing the product table with the new calculated column added.

Screenshots: Product Table with Flag

Oracle

```
SELECT *
FROM   Product;
```

ot Output × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 4 in 0.005 seconds

	PRODUCT_ID	NAME	LENGTH_INCHES	HEIGHT_INCHES	WIDTH_INCHES	LAUNCH_DATE	EXTRA_CHARGE_FLAG
1	1	Lightbulb	2	4	2	04-APR-21	1
2	2	Lawn Mower	70	50	41	01-MAR-21	1
3	3	Shovel	45	11	15	01-JUN-20	0
4	4	Wrench	6	1	1	15-MAY-21	0

Microsoft SQL Server

```
SELECT *
FROM   Product;
```

Results | Messages

product_id	name	length_inches	height_inches	width_inches	launch_date	extra_charge_flag
1	Lightbulb	2.0	4.0	2.0	2021-04-04	1
2	Lawn Mower	70.0	50.0	41.0	2021-03-01	1
3	Shovel	45.0	11.0	15.0	2020-06-01	0
4	Wrench	6.0	1.0	1.0	2021-05-15	0

PostgreSQL

```
SELECT *
FROM   Product;
```

Output | Explain | Notifications | Messages

product_id [PK] numeric (12)	name character varying (32)	length_inches numeric (4,1)	height_inches numeric (4,1)	width_inches numeric (4,1)	launch_date date	extra_charge_flag boolean
1	Lightbulb	2.0	4.0	2.0	2021-04-04	true
2	Lawn Mower	70.0	50.0	41.0	2021-03-01	true
3	Shovel	45.0	11.0	15.0	2020-06-01	false
4	Wrench	6.0	1.0	1.0	2021-05-15	false

Notice that Oracle and SQL Server indicate a 1 for both the Lightbulb and Lawn Mower, since they qualify for extra shipping charges, and a 0 for the other products, since they do not qualify. In a similar fashion, Postgres indicates *true* for both the Lightbulb and Lawn Mower, and *false* for the rest.

Now, instead of calculating the condition in the query as in #16, we can simply query the flag. This is shown below.

Screenshots: Query Extra Charges with Flag

Oracle

```
SELECT Product.*,
       (length_inches*height_inches*width_inches)/1728 AS cubic_feet
FROM   Product
WHERE  extra_charge_flag = 1;
```

ript Output × | Query Result × | Query Result 1 ×

SQL | All Rows Fetched: 2 in 0.006 seconds

	PRODUCT_ID	NAME	LENGTH_INCHES	HEIGHT_INCHES	WIDTH_INCHES	LAUNCH_DATE	EXTRA_CHARGE_FLAG	CUBIC_FEET
1	1	Lightbulb	2	4	2	04-APR-21	1	0.009259259
2	2	Lawn Mower	70	50	41	01-MAR-21	1	83.04398

Microsoft SQL Server

```
SELECT Product.*,
       (length_inches*height_inches*width_inches)/1728 AS cubic_feet
FROM   Product
WHERE  extra_charge_flag = 1;
```

Results | Messages

	product_id	name	length_inches	height_inches	width_inches	launch_date	extra_charge_flag	cubic_feet
	1	Lightbulb	2.0	4.0	2.0	2021-04-04	1	0.00925925
	2	Lawn Mower	70.0	50.0	41.0	2021-03-01	1	83.04398148

PostgreSQL

```
SELECT Product.*,
       CAST((length_inches*height_inches*width_inches)/1728 AS DECIMAL(4,2)) AS cubic_feet
FROM   Product
WHERE  extra_charge_flag = true;
```

a Output | Explain | Notifications | Messages

product_id [PK] numeric (12)	name character varying (32)	length_inches numeric (4,1)	height_inches numeric (4,1)	width_inches numeric (4,1)	launch_date date	extra_charge_flag boolean	cubic_feet numeric (4,2)
1	Lightbulb	2.0	4.0	2.0	2021-04-04	true	0.01
2	Lawn Mower	70.0	50.0	41.0	2021-03-01	true	83.04

We use *extra_charge_flag = 1* with Oracle and SQL Server, and *extra_charge_flag = true* with Postgres. Notice that only the products qualifying for extra shipping charges show up in the result, and we did not need to include the logic in the query. We simply queried the flag.

There is much more to learn about calculated columns; however, the explanation for this step gives you enough information to address this step.