|  |
| --- |
| **Overview of the Lab** |
| One purpose of this lab it to teach you how to meaningfully relate data and to answer questions using related data, using SQL. In the prior lab you learned the fundamentals creating and using tables to store data, which is a good introduction to SQL. To be effective, however, you need to know how to work with relationships that naturally occur in the data you work with.  We don’t need to look far to recognize examples of relationships. For just some examples, people have addresses, pets are owned by owners, products are sold by stores, and cases are heard in a court. Relationships are both plentiful and inevitable in virtually any database. In many ways, the richness and complexity of the relationships in a database determines its usefulness in answering important questions about the data.  It is likewise inevitable with virtually any database data items need be formatted into human readable form, or manipulated to derive different results. Directly extracting values exactly as they are stored in a database works for some but not all queries. For example, a customer table may store a first and a last name, such as “Smith” and “Bob”, but emails or letters to the customer would use the full name, “Bob Smith”. Another objective of this lab is for you to learn how to format and manipulate data using functions and expressions.  From a technical perspective, together, we will learn:   * how to enforce relationships between two tables using a FOREIGN KEY constraint. * how to add related data to related tables. * how to ask questions and answer them using SQL queries that relate data. * details about the significant components that determine how a value is displayed in your SQL client. * to understand and effectively use expressions to manipulate data values. * how to use formatting functions to format data values into to human readable or other formats. * how to create and use more advanced Boolean expressions to solve more complex use cases. * how to create and use calculated columns. |

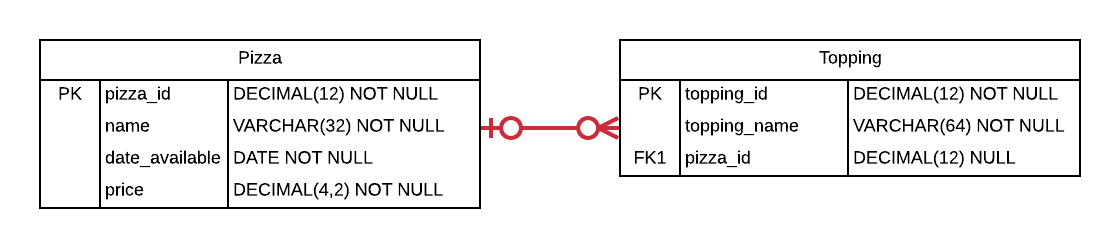
|  |
| --- |
| **Lab 2 Explanations Reminder** |
| As a reminder, it is important to read through the Lab 2 Explanation document to successfully complete this lab, available in the assignment inbox alongside this lab. The explanation document illustrates how to correctly execute each SQL construct step-by-step, and explains important theoretical and practical details. |

|  |
| --- |
| **Other Reminders** |
| * The examples in this lab will execute in modern versions of Oracle, Microsoft SQL Server, and PostgreSQL as is. * The screenshots in this lab display execution of SQL in the default SQL clients supported in the course – Oracle SQL Developer, SQL Server Management Studio, and pgAdmin – but your screenshots may vary somewhat as different version of these clients are released. * Don’t forget to commit your changes if you work on the lab in different sittings, using the “COMMIT” command, so that you do not lose your work. |

**Section One – Relating Data**

**Section Background**

To practice relating data, you will be working with the following simplified Pizza and Toppings schema.



In this schema, the Pizza table contains a primary key, the name of the pizza (for example, “Veggie” or “Meat Lovers”), the date when the pizza became available to order, and the price of the pizza. The Topping table contains a primary key, the name of the topping (such as “Sausage” or “Peppers”), and a foreign key that references the Pizza the topping is put on. The foreign key enforces the relationship between Pizza and Topping so that many toppings can be a part of a pizza. The foreign key is nullable since a particular topping may not have been assigned to a pizza (for example, perhaps a topping is available as an add-on but not part of the standard ingredients). There can also exist a pizza that has no toppings, namely, a plain pizza that only has tomato sauce and spice.

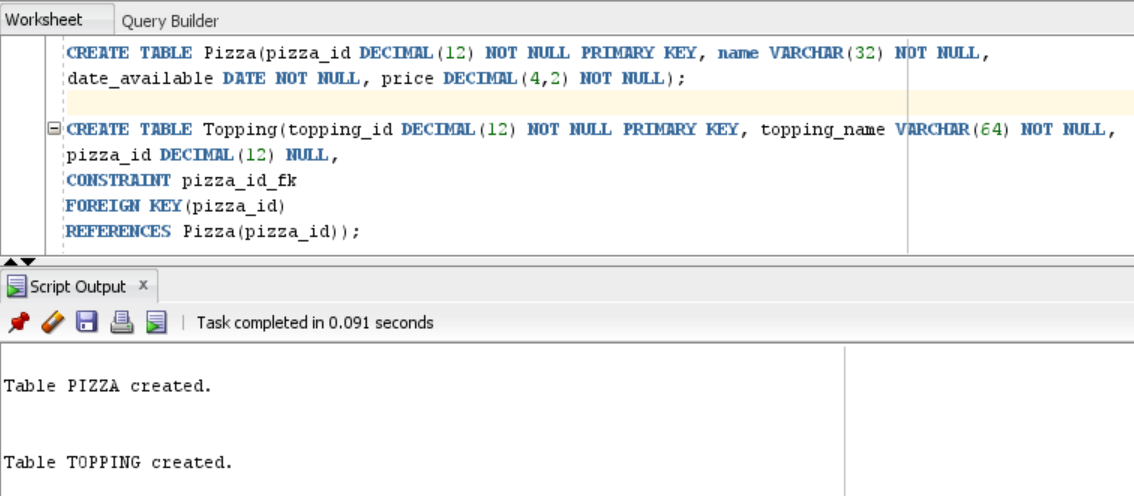
The schema is intentionally simplified when compared to what you might see in a real-world production schema. The schema only allows a particular topping to be a standard ingredient for one Pizza. The schema does not record a history of price changes as the price changes, nor does it support special pricing during special events. Many other attributes that would exist in a production database are not present. The current complexity is sufficient; additional complexity in the schema would not aid your learning at this point.

Do not worry if you don’t yet fully understand foreign keys and relationships. The Lab 2 explanations document gives you the information you need to complete the steps in this lab.

As a reminder, for each step that requires SQL, make sure to capture a screenshot of the command and the results of its execution.

**Section Steps**

1. *Creating the Table Structure –* Create the Pizza and Toppings tables, including all of their columns, datatypes, and constraints, including the foreign key constraint.



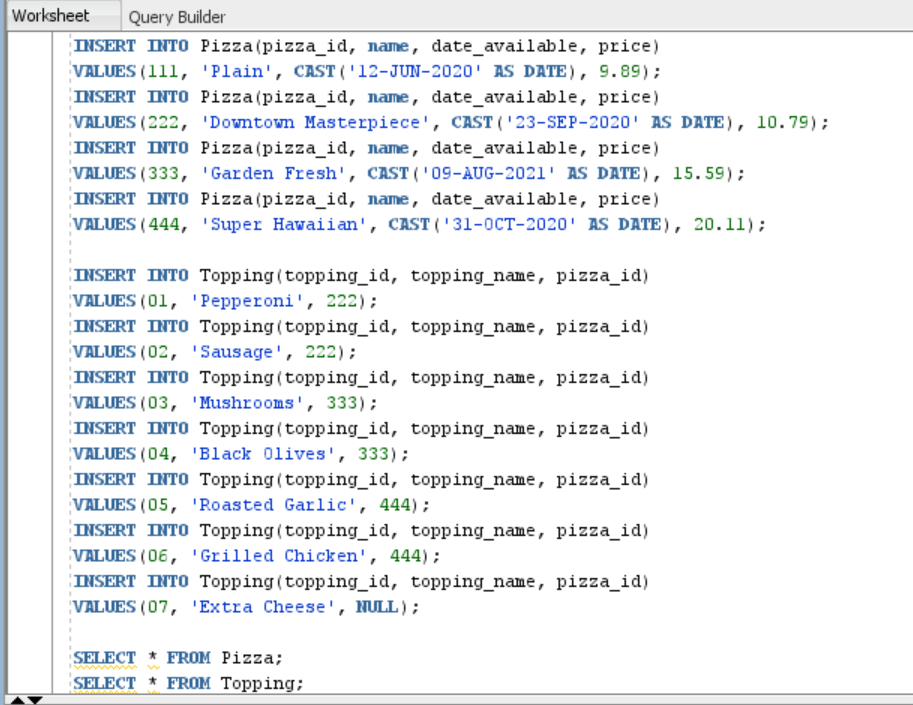
1. *Populating the Tables –* Insert at least four rows into the Pizza table. Two of the rows should have the values given below.

**Pizza 1**

name = Plain  
date\_available = 6/13/2020  
price = $9.89  
toppings: This pizza has no toppings.

**Pizza 2**  
name = Downtown Masterpiece  
date\_available = 9/23/2020  
price = $10.79  
toppings: You choose at least two veggie toppings for this pizza.  
  
For the other rows, you insert the ids, names, dates, prices, and toppings of your choosing (maybe you have some favorite pizzas?). Ensure that these other pizzas you create have at least two toppings.  
  
Lastly, insert an extra topping that is not associated with any pizza, that is, the topping should be an “add-on” which is not included in any pizza’s standard toppings.

Select all rows in both tables to view what you inserted.

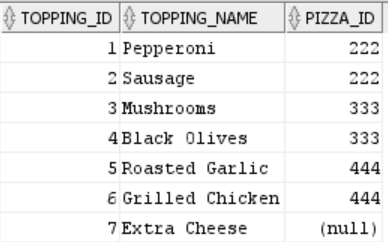


Pizza table:

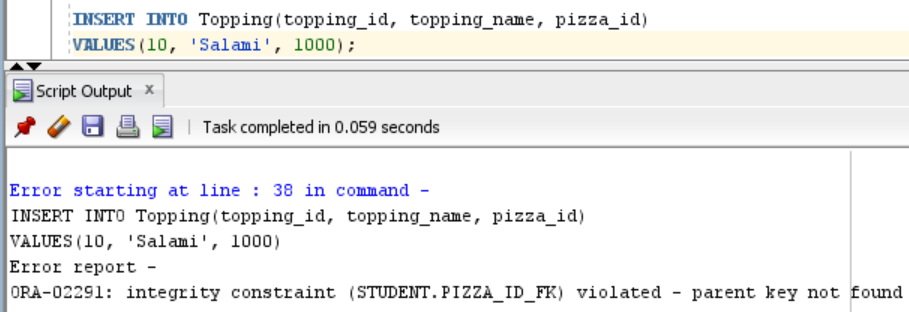
Table

Description automatically generated

Topping table:



1. *Invalid Reference Attempt –* As an exercise, attempt to insert a topping that references a pizza that doesn’t exist. Summarize:



* 1. why the insertion failed, and

The insertion failed as there was no pizza in the *Pizza* table that was referenced by the *Topping* table. The insert statement attempts to insert the topping “Salami” to a pizza with *pizza\_id* as 1000. From the *Pizza* table, 1000 is not present in the *pizza\_id* column. This is means there is no pizza for this ID. All references from the *Topping* table to the *Pizza* table will be valid due to the existence of the foreign key constraint. Any attempt to insert an invalid reference will be rejected by the RDBMS. Invalid reference in this case is the pizza\_id 1000 which does not exist in the parent (*Pizza*) table.

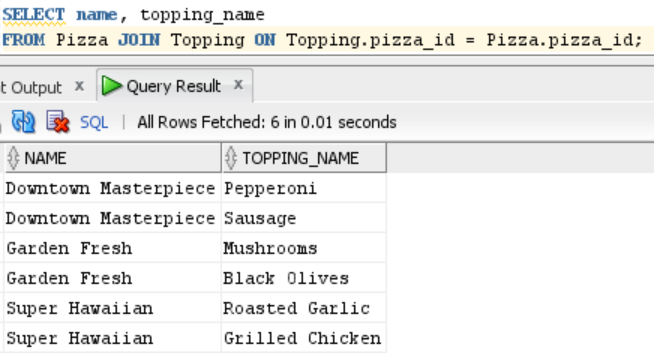
* 1. how you would interpret the error message from your RDBMS so that you know that the error indicates the Pizza reference is invalid.

The error report provides the following data on the error-

ORA-02291: integrity constraint (STUDENT.PIZZA\_ID\_FK) violated - parent key not found

STUDENT is the schema name on that was used when creating this lab, PIZZA\_ID\_FK is the identifier for the foreign key constraint. This constraint identifier helps us trace the error to the foreign key field *pizza\_id* that is present in the *Topping* table. The error is starting at line 38 in this command, so the *pizza\_id* present in the insert statement on line 38 is the reason the error occurred. The insert statement has an invalid reference hence it is rejected by the RDBMS. Invalid reference, in this case, is the pizza\_id 1000 which does not exist in the parent (*Pizza*) table.

1. *Listing Pizzas with Toppings –* With a single SQL query, fulfill the following request:   
     
   List the names of the pizzas that have toppings, and the names of all of the toppings that go with each pizza.



From a technical SQL perspective, explain why some rows in the Pizza table and some rows in the Toppings table were not listed.

The JOIN operation eliminates a row from each table: the row containing “Plain” in the Pizza table under the *name* column and the row “Extra Cheese” from the *topping-name* from the Topping table. The JOIN operation performs a cartesian product on the columns of both the tables, selects the rows that have matching data, and eliminates unmatched rows.

The cartesian product on the Pizza table and Topping table looks like the following:

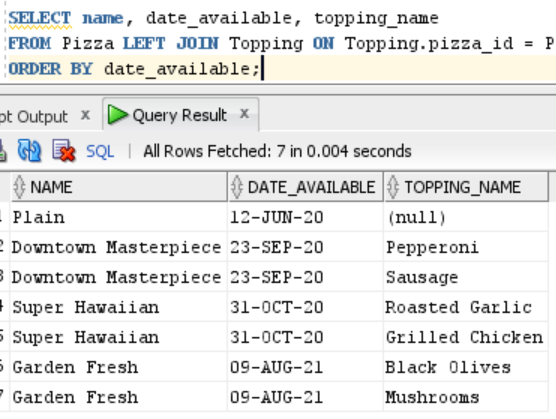
|  |  |  |  |
| --- | --- | --- | --- |
| **pizza\_id** | **name** | **topping\_name** | **pizza\_id** |
| 111 | Plain | Pepperoni | 222 |
| 111 | Plain | Sausage | 222 |
| 111 | Plain | Mushrooms | 333 |
| 111 | Plain | Black Olives | 333 |
| 111 | Plain | Roasted Garlic | 444 |
| 111 | Plain | Grilled Chicken | 444 |
| 111 | Plain | Extra Cheese |  |
| 222 | Downtown Masterpiece | Pepperoni | 222 |
| 222 | Downtown Masterpiece | Sausage | 222 |
| 222 | Downtown Masterpiece | Mushrooms | 333 |
| 222 | Downtown Masterpiece | Black Olives | 333 |
| 222 | Downtown Masterpiece | Roasted Garlic | 444 |
| 222 | Downtown Masterpiece | Grilled Chicken | 444 |
| 222 | Downtown Masterpiece | Extra Cheese |  |
| 333 | Garden Fresh | Pepperoni | 222 |
| 333 | Garden Fresh | Sausage | 222 |
| 333 | Garden Fresh | Mushrooms | 333 |
| 333 | Garden Fresh | Black Olives | 333 |
| 333 | Garden Fresh | Roasted Garlic | 444 |
| 333 | Garden Fresh | Grilled Chicken | 444 |
| 333 | Garden Fresh | Extra Cheese |  |
| 444 | Super Hawaiian | Pepperoni | 222 |
| 444 | Super Hawaiian | Sausage | 222 |
| 444 | Super Hawaiian | Mushrooms | 333 |
| 444 | Super Hawaiian | Black Olives | 333 |
| 444 | Super Hawaiian | Roasted Garlic | 444 |
| 444 | Super Hawaiian | Grilled Chicken | 444 |
| 444 | Super Hawaiian | Extra Cheese |  |

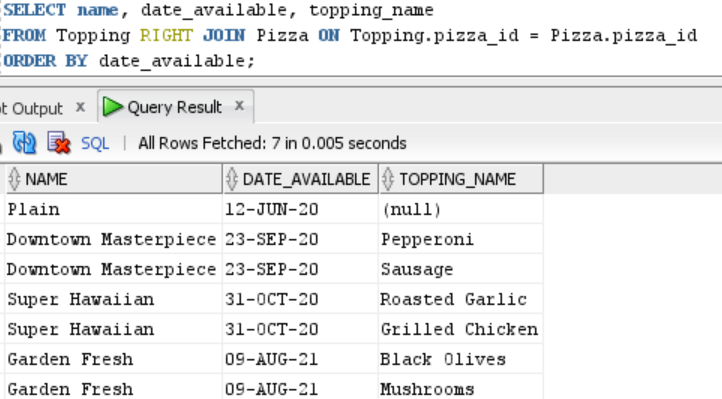
The highlighted rows are the matching rows that will be picked during the JOIN operation and the other rows are eliminated as they do not have a match.

1. *Listing All Pizzas –* Fulfill the following request:  
     
   List the names and availability date of all pizzas whether or not they have toppings. For the pizzas that have toppings, list the names of the toppings that go with each of those pizzas. Order the list by the availability date, oldest to newest.

There are two kinds of joins that can be used to satisfy this request. Write two queries using each type of join to satisfy this request.

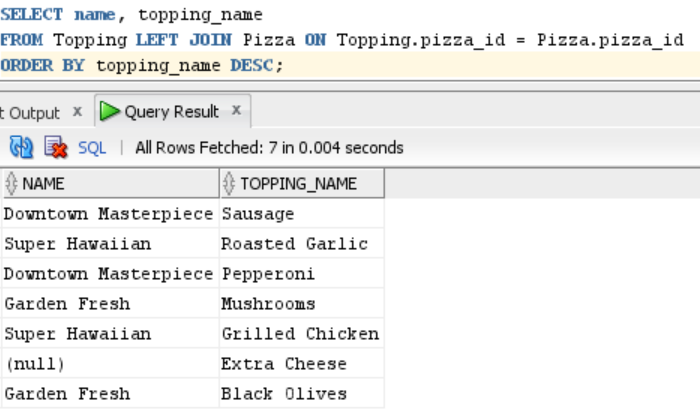
LEFT JOIN:



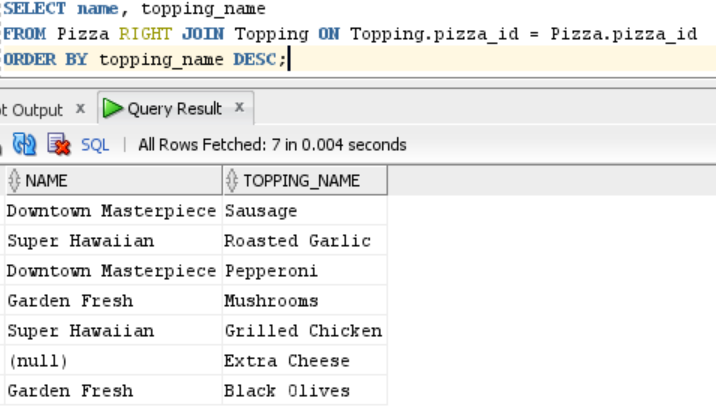
RIGHT JOIN:  
  


1. *Listing All Toppings –* Fulfill the following request:  
   List the names of all toppings whether or not they go with a pizza, and the names of the pizzas the toppings go with. Order the list by topping name in reverse alphabetical order.  
     
   Just as with step #5, there are two kinds of joins that can be used to satisfy this request. Write two queries using each type of join to satisfy this request.

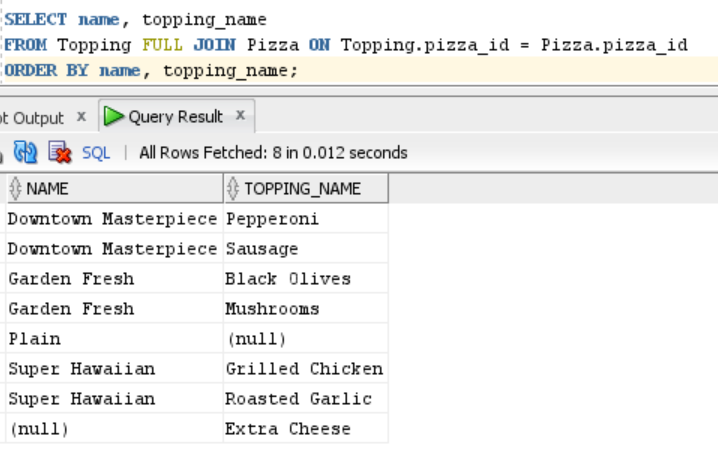
LEFT JOIN:



RIGHT JOIN:



1. *Listing All Pizzas and All Toppings –* Fulfill the following request with a single SQL query:   
   List the names of all pizzas and all toppings, as well as which pizzas go with which toppings. Order the list alphabetically by pizza name then by topping name. (default: ascending order; desc for descending order)



**Section Two – Expressing Data**

**Section Background**

While it is certainly useful to directly extract values as they are stored in a database, it is more useful in some contexts to manipulate these values to derive a different result. In this section we practice using value manipulation techniques to transform data values in useful ways. For example, what if we want to tell a customer exactly how much money they need to give for a purchase? We could extract a price and sales tax from the database, but it would be more useful to compute a price with tax as a single value by multiplying the two together and rounding appropriately, and formatting it as a currency, as illustrated in the figure below.

|  |  |
| --- | --- |
| *Less Useful to Customer* | |
| **price** | **tax\_percent** |
| 7.99 | 8.5 |

|  |
| --- |
| *More Useful to Customer* |
| **price\_with\_tax** |
| $8.67 |

We do not need to store the price with tax, because we can derive it when we need it.

As another example, what if we need to send an email communication to a customer by name? We could extract the prefix, first name, and last name of the customer, but it would be more useful to properly format the name by combining them in proper order, as illustrated below.

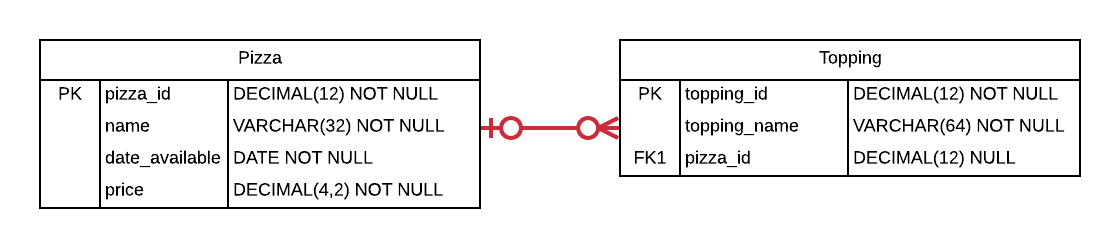
|  |  |  |
| --- | --- | --- |
| *Less Useful to Customer* | | |
| **prefix** | **first\_name** | **last\_name** |
| Mr. | Seth | Nemes |

|  |
| --- |
| *More Useful to Customer* |
| **name** |
| Mr. Seth Nemes |

Again, we do not need to store the formatted name, because we can derive it when we need it from its constituent parts. Manipulating raw data values stored in database tables can yield a variety of useful results we need without adding the burden of storing every such result.

In this section, you use expressions to manipulate and format data values. The first several steps in this section teach you several important concepts needed to correctly use expressions, including attributes of SQL clients, operator precedence, datatype precedence, and formatting functions. The later steps have you use this knowledge to manipulate and format data values.

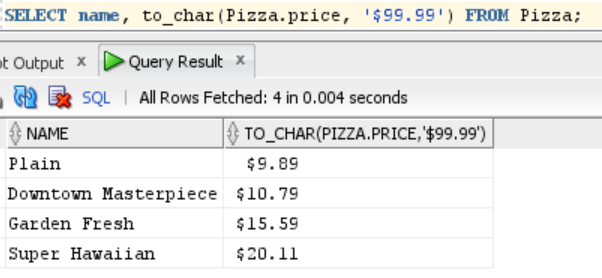
You work with the same Pizza and Toppings schema from Section One. The schema is illustrated below again for your review.



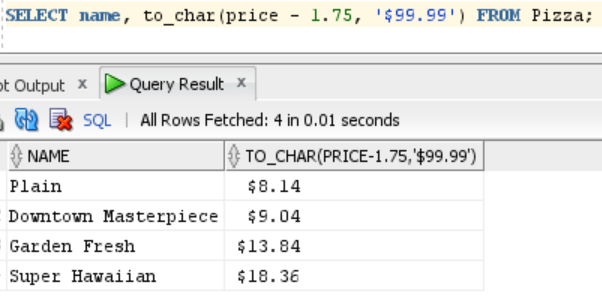
**Section Steps**

1. *Formatting as Money –* Fulfill the following request with a single query:

Management of the pizza shop wants to review its pizza pricing. List the names and prices of all pizzas, making sure to format the price monetarily in U.S. dollars (for example, “$11.99”).



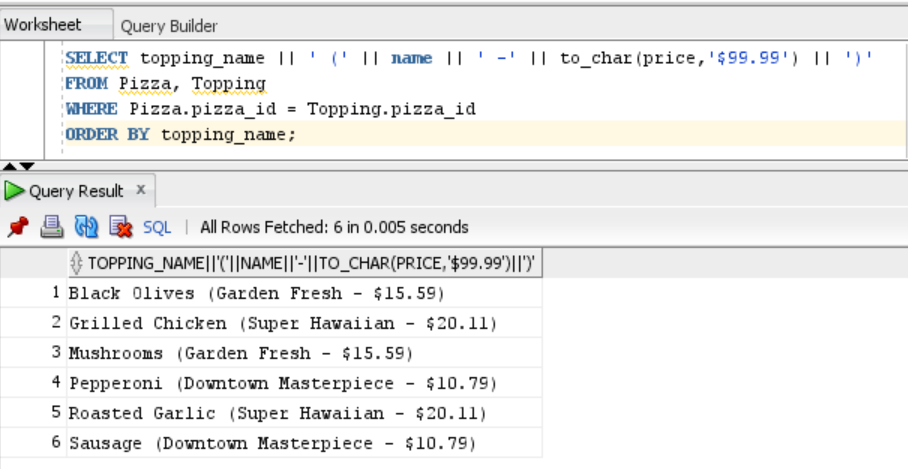
1. *Using Expressions –* Fulfill the following request with a single query:  
     
   The pizza shop is running a special where every pizza is discounted by a $1.75. List the names and discounted prices of all pizzas, making sure to format the price monetarily in U.S. dollars.



1. *Advanced Formatting –* Fulfill the following request with a single query:

The pizza shop wants to mail out mailers that promotes the toppings it offers, tied into the pizzas it sells. The shop wants each line in the mailer formatted like “ToppingName (PizzaName - Price)”, and wants the lines ordered alphabetically by topping name. For example, if a “Meat Lover’s” pizza costs $10.00 and has two toppings – Sausage and Pepperoni – the results would have two lines for this pizza:  
  
Pepperoni (Meat Lover’s - $10.00)

Sausage (Meat Lover’s - $10.00)



**Section Three – Advanced Data Expression**

**Section Background**

Boolean expressions can become complex, yet are essential to filtering results in SQL. Boolean expressions can determine if a set of columns meet a possibly complex set of conditions. In this section, you learn to work with more advanced Boolean expressions.

Modern relational databases have the ability to calculate a column automatically. Such a column is identified by many terms – *generated, computed, calculated, derived, and virtual*. If one column can be calculated from the values in other columns, it is best practice to avoid storing the extra value, because it can become out of sync with the other columns. That is, if one of the columns change value, but the derived column is not updated, the data becomes inconsistent. In fact, storing derived columns is a form of data redundancy.

As described in Section Two, one option to avoid storage is to dynamically calculate derived values in a SQL query using an expression. Another option, the topic of this section, is to create a generated column, and have the database calculate it automatically.

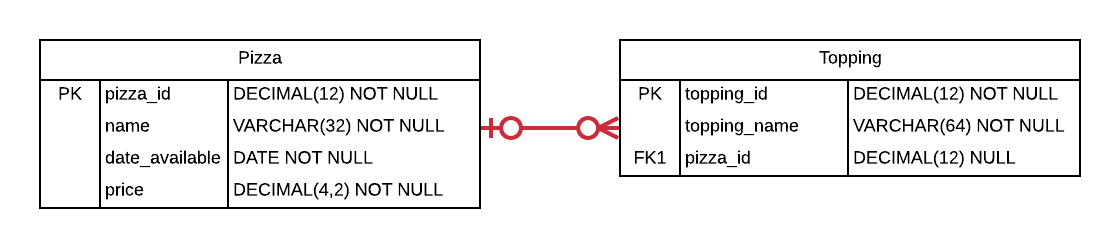
Using an example from Section Two, if one column contains a price, and another column contains a tax percentage, a third column could contain the price after tax.

|  |  |
| --- | --- |
|  | |
| **price** | **tax\_percent** |
| 7.99 | 8.5 |

|  |
| --- |
|  |
| **price\_with\_tax** |
| $8.67 |

This third value can be calculated by multiplying the price by the tax percentage, and performing proper rounding to two decimal points. We would not want to store the price with tax, because we can derive it. In this section, you learn how to create generated columns to contain values that can be derived from other columns.

You work with the same Pizza and Toppings schema from prior sections. The schema is illustrated below again for your review.



**Section Steps**

1. *Evaluating Boolean Expressions* – Indicate the final values for each of the Boolean expressions below. You must show your work for full credit, by showing the value of each operation step-by-step.
   1. (true AND false) OR (true AND true)

false OR true

true

* 1. (true OR false) AND NOT(false OR false) AND (false AND true)

true AND NOT(false) AND false

true AND true AND false

true AND false

false

* 1. NOT((false OR true) AND NOT(true AND true) AND (false OR true))

NOT((true) AND NOT(true) AND (true))

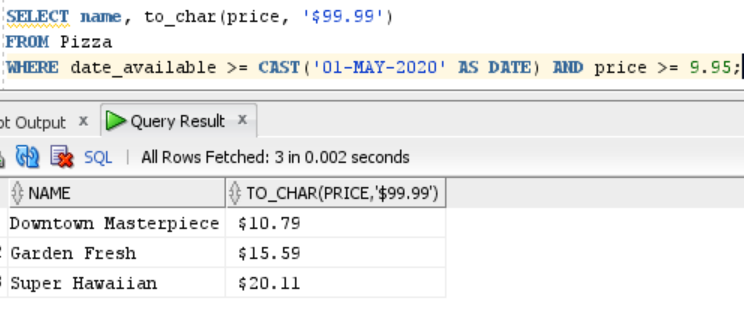
NOT(true AND false AND true)

NOT(false AND true)

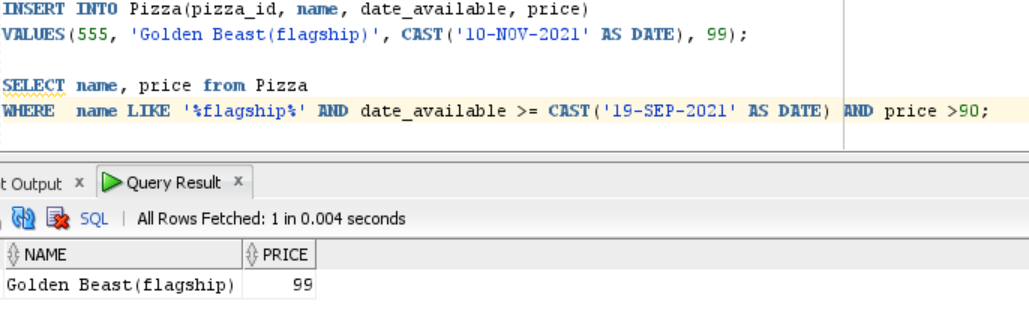
NOT(false)

true

1. *Using Boolean Expressions in Queries* – Address the following scenarios.
   1. Any pizza matching the following condition is considered a *signature pizza* for the pizza shop: Any pizza, except for the “Plain” pizza, that is available on or after 5/1/2020, with a price of $9.55 or higher, is a signature pizza. Write a query that shows the name and price of all signature pizzas.



* 1. The pizza shop also has one *flagship pizza* that sets the shop apart from other pizza shops. First, define your own conditions for this flagship pizza, making sure the conditions include the name, date, and price. Then write a query that shows the name and price of the flagship pizza. It’s fine if you’d like to insert another row of pizzas to become your flagship pizza.



A new row has been inserted to include the flagship pizza in the Pizza table. This new pizza is named “Golden Beast (flagship)” that was introduced on Nov 10th, 2021 and is the most expensive pizza on the menu, costing $99.

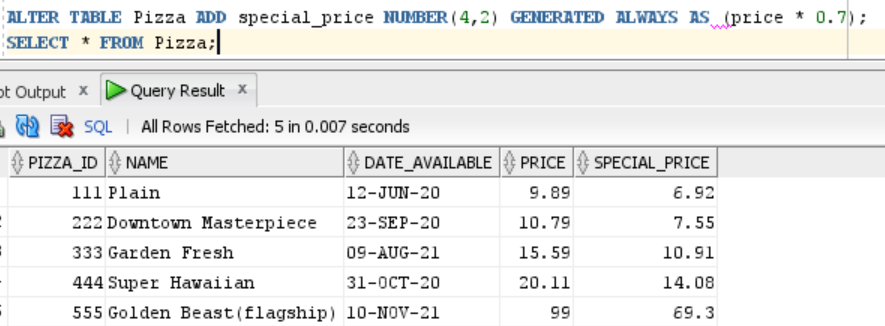
To retrieve this row, the condition uses the LIKE operator to search for a substring in the given column. In this example, the LIKE operator will search for a string that has “flagship” as a part of it under the *name* column in the *Pizza* table. The % sign in %flagship% means there can be any number of unknown characters before and after the word “flagship”. The expression is used in conjunction with *date\_available* and *price* attributes.

name LIKE '%flagship%' AND date\_available >= CAST('19-SEP-2021' AS DATE) AND price > 90

The Boolean AND operator is used to execute logical combinations between name AND date\_available and date\_available AND price. When all of these evaluate to true, the desired result is obtained.

1. *Using Generated Columns –* Address the following.
   1. Define a new generated column named *special\_price*, which gives a lower price for the pizza for when the pizza shop offers specials (such as on holidays or during weekly specials). You determine the percentage or fixed value discount for the special price. Then write a query that lists out the name of all pizzas, along with their regular and special prices.

The column named *special\_price* is added by altering the existing Pizza table. The *special\_price* column lists the discounted prices of pizzas. The discount offered is 30% off on the original price of each pizza.



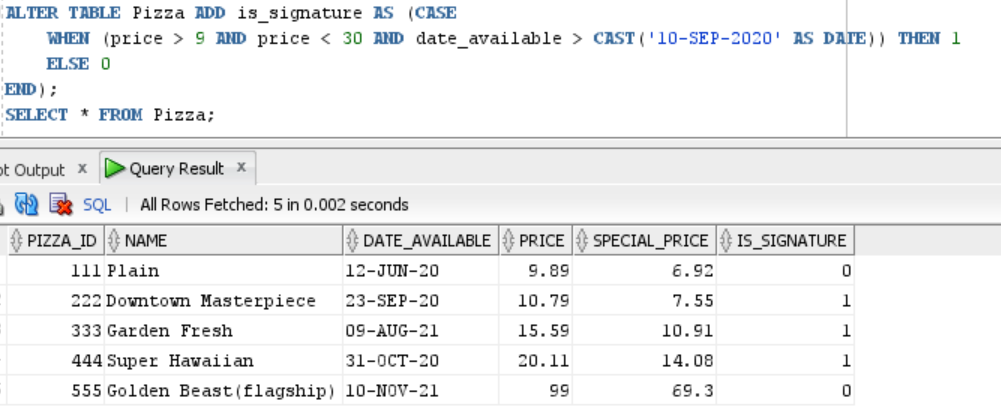
* 1. Address #12a again in a different way. First, define a generated column named *is\_signature* on the Pizza table, which indicates whether it’s a signature pizza or not. Then write a query that lists only the signature pizzas. Include relevant columns in the result.

The *is\_signature* column is added into the Pizza table and the values in the column are based on the conditions provided in the CASE…END part of the query.

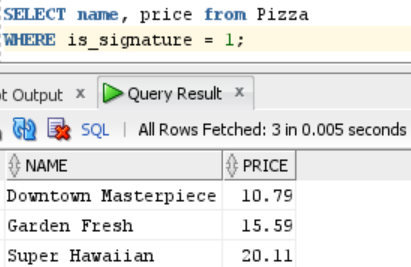
“In its basic form, CASE … END works like this:

CASE WHEN matching\_condition THEN matching\_result ELSE unmatching\_result END”

The matching condition is upon the price and the date as shown in the query below. The matching result returns 1, meaning the pizza is a signature pizza and 0 for results that don’t match. In this example, *Downtown Masterpiece*, *Garden Fresh*, and *Super Hawaiian* are the signature pizzas.



Query that lists only the signature pizzas:



# Evaluation

Your lab will be reviewed by your facilitator or instructor with the criteria outlined in the table below. Note that the grading process:

* involves the grader assigning an appropriate letter grade to each criterion.
* uses the following letter-to-number grade mapping – A+=100,A=96,A-=92,B+=88,B=85,B-=82,C+=88,C=85,C-=82,D=67,F=0.
* provides an overall grade for the submission based upon the grade and weight assigned to each criterion.
* allows the grader to apply additional deductions or adjustments as appropriate for the submission.
* applies equally to every student in the course.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criterion** | **A** | **B** | **C** | **D** | **F** |
| **Section 1: Quality (30%)** | The results for all steps in Section 1 are complete and correct. Appropriate SQL constructs have been used for all steps, and supporting explanations are present and accurate. All screenshots in Section 1 are legible. The section is well organized. All supporting explanations are clear and understandable. | The results for most steps in Section 1 are complete and correct. The appropriate SQL constructs have been used for most steps, and supporting explanations are mostly present and accurate. Most screenshots in Section 1 are legible. The section is organized. Most supporting explanations are clear and understandable. | The results for some steps in Section 1 are complete and correct. Appropriate SQL constructs have been used for some steps, and some supporting explanations are present and accurate. Some screenshots in Section 1 are legible. Some supporting explanations are clear and understandable. | The results for most steps in Section 1 are incomplete or incorrect. Appropriate SQL constructs have not been used for most steps. The screenshots in Section 1 are mostly illegible or missing. Most supporting explanations are unclear or missing. The section is disorganized. | The results for virtually all steps in Section 1 are incomplete or incorrect. Appropriate SQL constructs have not been used. Virtually all screenshots in Section 1 are illegible or missing. Virtually all supporting explanations are unclear or missing. The section is disorganized. |
| **Section 2: Quality (30%)** | The results for all steps in Section 2 are complete and correct. Appropriate SQL constructs have been used for all steps, and supporting explanations are present and accurate. All screenshots in Section 2 are legible. The section is well organized. All supporting explanations are clear and understandable. | The results for most steps in Section 2 are complete and correct. The appropriate SQL constructs have been used for most steps, and supporting explanations are mostly present and accurate. Most screenshots in Section 2 are legible. The section is organized. Most supporting explanations are clear and understandable. | The results for some steps in Section 2 are complete and correct. Appropriate SQL constructs have been used for some steps, and some supporting explanations are present and accurate. Some screenshots in Section 2 are legible. Some supporting explanations are clear and understandable. | The results for most steps in Section 2 are incomplete or incorrect. Appropriate SQL constructs have not been used for most steps. The screenshots in Section 2 are mostly illegible or missing. Most supporting explanations are unclear or missing. The section is disorganized. | The results for virtually all steps in Section 2 are incomplete or incorrect. Appropriate SQL constructs have not been used. Virtually all screenshots in Section 2 are illegible or missing. Virtually all supporting explanations are unclear or missing. The section is disorganized. |
| **Section 3: #11 Soundness (10%)** | The results of all Boolean expressions are correct. Step-by-step work for each expression is present and correct. | The results of most Boolean expressions are correct. Step-by-step work for most expressions is present and correct. | The results of some Boolean expressions are correct. Step-by-step work for some expressions is present and somewhat correct. | The results of all or almost all expressions are incorrect. Step-by-step work for some expressions is minimally present and partially correct. | The answer is missing, or all results are incorrect. Work has not been shown, or is entirely incorrect. |
| **Section 3: #12a Soundness (10%)** | The signature pizzas are listed correctly, and the logic used in the query is entirely accurate. All non-matching pizzas are excluded. | The signature pizzas are mostly listed correctly, and the logic used in the query is mostly accurate. Most non-matching pizzas are excluded. | The signature pizzas are listed somewhat correctly, and the logic used in the query is somewhat accurate. Some non-matching pizzas are excluded. | The signature pizzas listed are mostly incorrect, and the logic used in the query is mostly inaccurate. Non-matching pizzas may be included. | The answer for #16a is missing, or the query's results are entirely incorrect. |
| **Section 3: #12b Soundness (10%)** | The condition for the flagship pizza includes the name, date, and price, and is reasonable in complexity. The flagship pizza has been listed correctly, and the logic used in the query is entirely accurate. All non-matching pizzas are excluded. | The condition for the flagship pizza includes the name, date, and price. The flagship pizza has been listed correctly, and the logic used in the query is mostly accurate. Most non-matching pizzas are excluded. | The condition for the flagship pizza includes at least two of these -- name, date, and price. The flagship pizza has been listed. The logic used in the query is somewhat inaccurate, and non-matching pizzas may be included. | The condition for the flagship pizza includes at least one of these -- name, date, and price. The flagship pizza may not have been listed. The logic used in the query is mostly inaccurate, and non-matching pizzas may be included. | The answer for #16b is missing, or the query's results are entirely incorrect. The condition for the flagship pizza does not contain name, date, or price. |
| **Section 3: #13 Soundness (10%)** | Both generated columns give accurate values for all rows. The logic for both is entirely accurate for current and any future pizzas. Only meaningful columns are included. | Both generated columns give accurate values for most rows. The logic for both is mostly accurate for current and any future pizzas. Mostly meaningful columns are included. | Both generated columns give accurate values for some rows. The logic for both is somewhat accurate for current and any future pizzas. Some meaningful columns are included. | Both generated columns give mostly inaccurate values. The logic for both is mostly inaccurate for current and any future pizzas. Meaningful columns may not be included. | The answer for #17 is missing, or both generated columns give entirely inaccurate results. The logic for both columns is entirely unsound. Meaningful columns may not be included. |

Use the **Ask the Teaching Team Forum** if you have any questions regarding how to approach this lab. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.