

Lab 4 Explanations For Postgres

This explanation document illustrates how to correctly execute each SQL construct step-by-step for Lab 4, and explains important theoretical and practical details. Before completing a step, read its explanation here first.

Use this explanations document only if you are using Postgres. If you are using Oracle or SQL Server, explanations for those DBMS' are available in a different document in the assignments section of the course.

Table of Contents

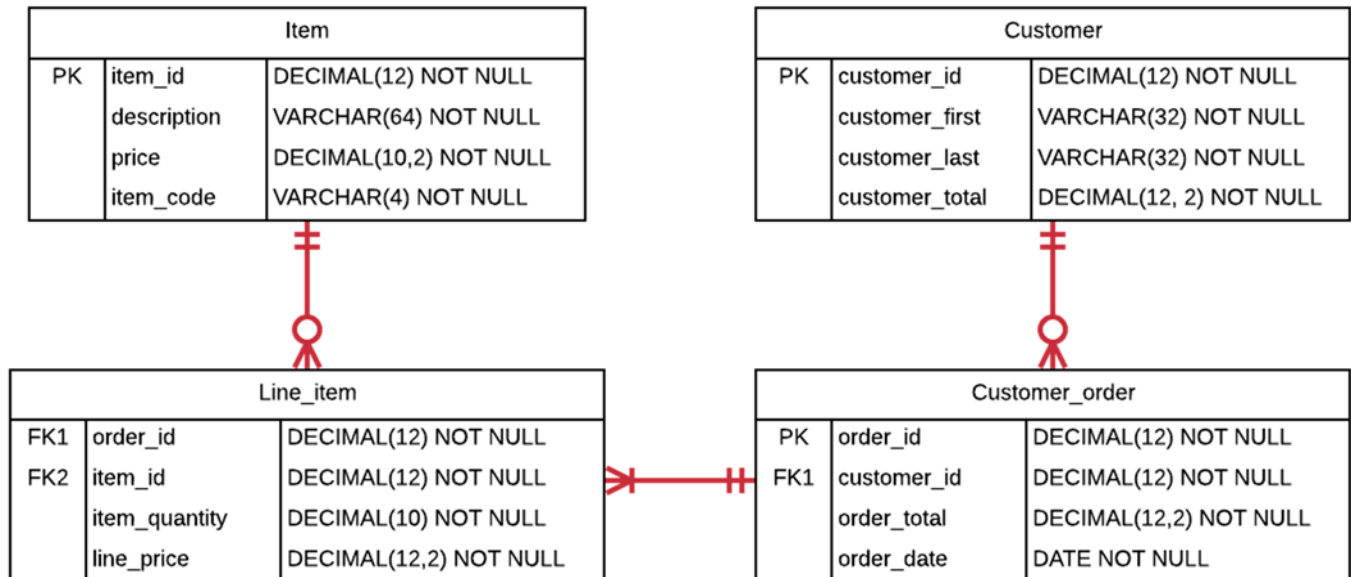
Section One – Stored Procedures	4
Step 1 – Create Table Structure	4
Code: Creating Customer Order Schema	5
Screenshot: Creating the Customer Order Schema.....	6
Code: Creating Customer Order Sequences	7
Screenshot: Creating the Customer Order Sequences	7
Step 2 – Populate Tables.....	8
Code: Inserting First Customer and Order	8
Screenshots: Tables After Firsts Inserted	9
Code: Inserting All Items	9
Code: Inserting Line Items for First Order	10
Code: Viewing First Order	10
Screenshot: Viewing First Order	11
Code: Inserting Remainder of Customer Data.....	12
Code: Viewing All Orders	12
Screenshot: Viewing All Orders.....	13
Step 3 – Create Hardcoded Procedure.....	14
Code: Creating add_customer_harry Procedure.....	14
Screenshot: Creating add_customer_harry Procedure	16
Code: Executing add_customer_harry Procedure.....	16
Screenshot: Executing add_customer_harry Procedure	16
Screenshot: Customer Table After Execution.....	17
Step 4 – Create Reusable Procedure.....	18
Screenshot: Second Execution of add_customer_harry.....	18
Code: Creating add_customer Procedure	19
Screenshot: Creating add_customer Procedure	19
Code: Executing add_customer Procedure	19
Screenshot: Executing Parameterized add_customer Procedure	20
Screenshot: Listing Customer Table After Add.....	20
Step 5 – Create Deriving Procedure	22
Pseudocode for Basic Variable Use	22
Code: Creating add_item Procedure	23
Screenshot: Compiling and Executing add_item.....	24
Screenshot: Listing Item Table	24
Step 6 – Create Lookup Procedure.....	26
Code: ADD_LINE_ITEM procedure	26

Screenshot: Compiling and Executing add_line_item	27
Screenshot: Listing Line_item After Execution	28
Section Two – Triggers	29
Step 7 – Single Table Validation Trigger	29
Code: No Negative Balance Validation Trigger	29
Screenshot: Compile the No Negative Balance Trigger	31
Screenshot: Test the No Negative Balance Trigger	31
Step 8 – Cross-Table Validation Trigger	32
Code: Correct Line Price Validation Trigger	32
Screenshot: Test the Price Validation Trigger	33
Step 9 – History Trigger	34
Code: Create the Item_price_history Table	34
Code: The Item_price_history Trigger	35
Screenshot: Compilation and Update for Item_price_history	36
Screenshot: Listing Item_price_history	36
Section Three – Normalization	37
Step 10 – Creating Normalized Table Structure	37

Section One – Stored Procedures

Step 1 – Create Table Structure

To help demonstrate how to complete the commands in this section, we work with simplified customer order schema which tracks customers and their orders of items. The schema itself is illustrated below.



The Item table contains items that can be purchased, with a primary key, a description of the item, a price, and an item_code which is an identifier by which the item can be referenced. The Customer table contains basic information on customers such as their first and last name, and a total balance which they owe. The Customer_order table contains basic information about an order itself, including a reference to the customer who placed the order, the sum total for the order, and the date the order was placed. The Line_item table contains information on individual lines in the order, including a reference to the item that was purchased, the quantity that was purchased, and the total amount for that line (for example, if an item costs \$10 and 2 of them were purchased, the total amount for the line would be \$20).

We create the customer order schema illustrated above in its entirety, including all primary and foreign key constraints, with the SQL below.

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

Code: Creating Customer Order Schema

```
CREATE TABLE Customer(  
customer_id    DECIMAL(12) NOT NULL,  
customer_first VARCHAR(32),  
customer_last  VARCHAR(32),  
customer_total DECIMAL(12, 2),  
PRIMARY KEY (customer_ID));  
  
CREATE TABLE Item(  
item_id    DECIMAL(12) NOT NULL,  
description VARCHAR(64) NOT NULL,  
price      DECIMAL(10, 2) NOT NULL,  
item_code  VARCHAR(4) NOT NULL,  
PRIMARY KEY (item_id));  
  
CREATE TABLE Customer_order (  
order_id    DECIMAL(12) NOT NULL,  
customer_id DECIMAL(12) NOT NULL,  
order_total DECIMAL(12,2) NOT NULL,  
order_date  DATE NOT NULL,  
PRIMARY KEY (ORDER_ID),  
FOREIGN KEY (CUSTOMER_ID) REFERENCES customer);  
  
CREATE TABLE Line_item(  
order_id    DECIMAL(12) NOT NULL,  
item_id      DECIMAL(12) NOT NULL,  
item_quantity DECIMAL(10) NOT NULL,  
line_price   DECIMAL(12,2) NOT NULL,  
PRIMARY KEY (ORDER_ID, ITEM_ID),  
FOREIGN KEY (ORDER_ID) REFERENCES customer_order,  
FOREIGN KEY (ITEM_ID) REFERENCES item);
```

Below are screenshots of creating the schema.



Screenshot: Creating the Customer Order Schema

Postgres

```
1 CREATE TABLE Customer(  
2   customer_id    DECIMAL(12) NOT NULL,  
3   customer_first VARCHAR(32),  
4   customer_last  VARCHAR(32),  
5   customer_total DECIMAL(12, 2),  
6   PRIMARY KEY (customer_ID));  
7  
8 CREATE TABLE Item(  
9   item_id    DECIMAL(12) NOT NULL,  
10  description VARCHAR(64) NOT NULL,  
11  price      DECIMAL(10, 2) NOT NULL,
```

Data Output Explain Notifications Messages

CREATE TABLE

Query returned successfully in 67 msec.

Our next step is to create a sequence for each table. A *sequence* is a database object capable of generating unique primary key values, and is the preferred mechanism for doing so. Sequences generate unique whole numbers, starting with the first, and incrementing to the next number each time a new value is needed. The database guarantees a sequence will not generate the same number twice, thus making the values unique and suitable for primary keys.

Why use a sequence when we can hardcode values like 1 and 2 in our insert statements? There are many reasons. First, inserts often happen over many years as applications live on and on, so it is impractical to hardcode every value. Additionally, real-world databases oftentimes have millions of rows in some of the more used tables. Again, it is not practical to hardcode millions of values in such instances. We are looking for automation, so we do not want a human being to be asked for the primary and foreign key values every time an application needs to insert more rows. In short, dynamically generating primary key values is critical to modern database operation, and sequences are the preferred means of doing so.

There are a few advanced options with sequences which are available but not typically used for primary key configurations. Many sequences are configured to start at 1, and increment upward by 1 each time a new value is needed; however, the starting value and incrementing value can be changed if needed (such as starting at 10 and incrementing by 10, for example). Sequences can be set to cycle, which means once they reach their maximum value, they reset back to the minimum value again. This means once a cycle occurs, the sequence will be regenerating numbers already generated, making them non-unique. Typically for primary keys, the sequence is not configured to cycle. Instead, we make sure the primary key is large enough to hold all values, current and future.

The SQL syntax for creating a basic sequence is straightforward:

```
CREATE SEQUENCE <sequencename> START WITH 1
```

Because we need one sequence per table, a convention I recommend for sequence names is to name it like this:

tablename_seq

For example, if we had a Person table and that table needed a sequence, we would name the sequence "person_seq".

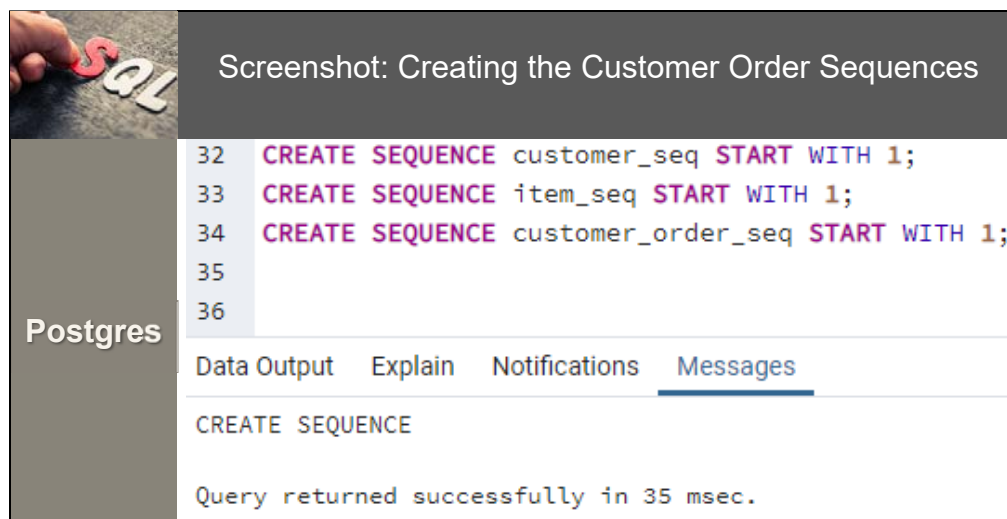
Below are the sequences I create for the customer order schema.

Code: Creating Customer Order Sequences

```
CREATE SEQUENCE customer_seq START WITH 1;  
CREATE SEQUENCE item_seq START WITH 1;  
CREATE SEQUENCE customer_order_seq START WITH 1;
```

The three sequences are for the Customer, Item, and Customer_order tables, respectively. Line_item does not need a sequence, because it uses a composite primary key consisting of other tables' keys. That is, we do not need to generate unique values for the Line_item table since it does not use a synthetic primary key of its own.

Creating them looks as follows.



Screenshot: Creating the Customer Order Sequences

```
32 CREATE SEQUENCE customer_seq START WITH 1;  
33 CREATE SEQUENCE item_seq START WITH 1;  
34 CREATE SEQUENCE customer_order_seq START WITH 1;  
35  
36
```

Postgres

Data Output Explain Notifications Messages

CREATE SEQUENCE

Query returned successfully in 35 msec.

As you can see, creating sequences for tables is straightforward.

Step 2 – Populate Tables

You can interact with a sequence in one of two ways – obtaining the next value of the sequence, and obtaining the current value. Obtaining the next value generates the next unique number. The next value is typically requested when a new primary key value is needed for a table. Sequences are safe to use concurrently. This means even if hundreds or thousands of transactions are running simultaneously that use the same sequence, the sequence generates unique numbers for them all. Obtaining the current value does not generate the next unique number, but instead returns the last number requested. The current value is typically used when a primary key value has already been obtained, and that same value is needed again so it can be inserted into a different table as a foreign key.

When sequences are involved, the order of inserts becomes quite important. The reason is, if we use a sequence to insert a new row, and then a referencing table must have a foreign key to that row. We would want to use the sequence's current value to create the foreign key, so wouldn't want to change the value.

For example, in the customer order schema, `Customer_order` references `Customer`. So rather than inserting all customers followed by all customer orders, we would insert one customer, followed by all of that customer's orders. Then we would insert the second customer, followed by the second customer's orders, and so on. This way we can make use of the sequence's *current value* to insert the foreign keys. Let's look at an example of inserting the first customer and order.

Code: Inserting First Customer and Order


```
--Insert first customer and order.  
INSERT INTO customer VALUES(nextval('customer_seq'),'John','Smith',0);  
INSERT INTO customer_order VALUES(nextval('customer_order_seq'),currval('customer_seq'),  
506,CAST('18-DEC-2005' AS DATE));
```

Since this is the first time we are using a sequence in this lab, let's examine this code closely. The first line inserts the first customer, "John Smith". Instead of hardcoding a primary key value such as "1" in the insert, the function call `nextval('customer_seq')` is used to retrieve the next unique value from `customer_seq`, which is the sequence we created for the customer table. The next unique value is "1" since the sequence was just created.

The second insert follows the pattern of the first to create the first order for John Smith, but with a twist. It uses the function call `nextval('customer_order_seq')` to retrieve the next unique value for the primary key (which will be "1" since that sequence was also just created). It uses `currval('customer_seq')` to retrieve the current value of `customer_seq`. Getting the current value does not advance the sequence; rather, it retrieves the last unique value retrieved. This allows the customer order to reference back to the customer without hardcoding any values. So the `customer_order_id` primary key value is retrieved with one sequence, and the `customer_id` foreign key is retrieved with another sequence, using a combination of `nextval` and `currval`.

The use of the foreign key shows why it is important to only insert one customer at-a-time. We need to retrieve the next unique value for the customer, then use it to create the new customer as well as all of that customer's orders. If we had inserted all customers first instead of just one, then we'd need some other means of looking up the foreign key values when inserting the orders.

Let's examine the two tables after the inserts above have been executed.



Screenshots: Tables After Firsts Inserted

```
41 SELECT * FROM Customer;
42
```

Data Output

Explain

Notifications

Messages

customer_id [PK] numeric (12)	customer_first character varying (32)	customer_last character varying (32)	customer_total numeric (12,2)
1	John	Smith	0.00

```
SELECT * FROM Customer_order;
```

a Output

Explain

Notifications

Messages

order_id [PK] numeric (12)	customer_id numeric (12)	order_total numeric (12,2)	order_date date
1	1	506.00	2005-12-18

Because we had just created the sequence and it starts at 1, John Smith's `customer_id` value was generated as 1 when using `nextval` on `customer_seq`. Similarly, the first order has a primary key value of 1 for the same reason. And, the order successfully references back to John Smith's primary key value of 1, by using `currval` on `customer_seq` for the foreign key value. So, we have seen an example where, with careful ordering of our inserts and judicious use of a variable, we can use sequences for our inserts, instead of hardcoding values.

Before proceeding further, we need to think about how we are going to insert line items and items. Line items reference back to the orders they belong to, in addition to the item they are representing. While we may be able to use `currval` reference back to the order, we can't do the same for the item foreign key, because many different line items may reference the same item. That is, different customers may purchase the same item, so we can't possibly order everything in a way such that using a variable is sufficient.

We need another method, which is a *subquery lookup*. In short, in lieu of hardcoding an item's primary key value, we'll look it up with a small subquery instead. Continuing on after inserting our first customer and order, we'll go ahead and insert all items with the following code.

Code: Inserting All Items

```
INSERT INTO item VALUES(nextval('item_seq'),'Plate',10, 'P001');
INSERT INTO item VALUES(nextval('item_seq'),'Bowl',11, 'B002');
INSERT INTO item VALUES(nextval('item_seq'),'Knife',5, 'K003');
INSERT INTO item VALUES(nextval('item_seq'),'Fork',5, 'F004');
INSERT INTO item VALUES(nextval('item_seq'),'Spoon',5, 'S005');
INSERT INTO item VALUES(nextval('item_seq'),'Cup',12, 'C006');
```

We insert all items, generating unique primary keys with `nextval` for `item_seq`. Later, when we need to reference back to them with foreign keys, we'll look them up as needed. With those inserted, we can now create the line items for the first customer's first order, shown below.

Code: Inserting Line Items for First Order

```
--Create the line items for the first order.
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Plate'),10,100);
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Spoon'),2,10);
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Bowl'),36,396);
```

Let's review these inserts. For the foreign key to `customer_order`, we use *currval* on *customer_order_seq* to refer to the first inserted customer order. We are already familiar with this technique, so this part of the insert is just another example of doing so.

For retrieving the primary key of each item, however, we use the subquery (`SELECT item_id FROM item WHERE description=<item_description>`). Although subqueries in general are the topic of a future lab, for this lab it is enough to know that instead of hardcoding a single value, we can substitute a query that retrieves exactly one value in its place. In this case, each subquery retrieves one item id corresponding to the item that matches the description in the subquery's WHERE clause. For example, the first insert retrieves the `item_id` for "Plate", meaning that the first line will reference "Plate" in the Item table. The second line item is for spoons, and the third line item is for bowls.

With the first order completely inserted, we can now use the query shown below to view the items in the order.

Code: Viewing First Order

```
--Get the first order details.
SELECT customer_first, customer_last, description, item_quantity
FROM Customer
JOIN Customer_order ON customer_order.customer_id = Customer.customer_id
JOIN Line_item ON line_item.order_id = customer_order.order_id
JOIN Item ON item.item_id = line_item.item_id;
```



Screenshot: Viewing First Order

Postgres

```
--Get the first order details.  
SELECT customer_first, customer_last, description, item_quantity  
FROM Customer  
JOIN Customer_order ON customer_order.customer_id = Customer.customer_id  
JOIN Line_item ON line_item.order_id = customer_order.order_id  
JOIN Item ON item.item_id = line_item.item_id;
```

[a Output](#) [Explain](#) [Notifications](#) [Messages](#)

customer_first character varying (32)	customer_last character varying (32)	description character varying (64)	item_quantity numeric (10)
John	Smith	Plate	10
John	Smith	Spoon	2
John	Smith	Bowl	36

The query results show us that the first order is for a plate, bowl, and spoon, with varying quantities of each.

Below are the remainder of the inserts. These use both methods.

Code: Inserting Remainder of Customer Data

```
--Insert the rest of the customers, orders, and line items.
--John's remaining orders.
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),1000,CAST('17-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Plate'),95,950);
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Knife'),10,50);
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),10,CAST('19-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Fork'),3,15);
--Mary's orders.
INSERT INTO customer VALUES(nextval('customer_seq'),'Mary','Berman',0);
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),1584,CAST('18-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Fork'),3,15);
--Elizabeth's orders.
INSERT INTO customer VALUES(nextval('customer_seq'),'Elizabeth','Johnson',0);
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),15,CAST('19-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Cup'),132,1584);
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),15,CAST('20-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Fork'),3,15);
--Peter's orders.
INSERT INTO customer VALUES(nextval('customer_seq'),'Peter','Quigley',0);
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),100,CAST('17-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Spoon'),5,25);
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Bowl'),2,10);
INSERT INTO customer_order
VALUES(nextval('customer_order_seq'),currval('customer_seq'),40,CAST('18-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(currval('customer_order_seq'),(SELECT item_id FROM item WHERE description='Knife'),3,15);
```

Below the code and screenshot captures the important order information, showing all orders.

Code: Viewing All Orders

```
--Get all order details.
SELECT customer_first, customer_last, order_date, description, item_quantity,
line_price
FROM Customer
JOIN Customer_order ON customer_order.customer_id = Customer.customer_id
JOIN Line_item ON line_item.order_id = customer_order.order_id
JOIN Item ON item.item_id = line_item.item_id
ORDER BY customer_first, customer_last, order_date, description;
```



Screenshot: Viewing All Orders

Postgres

```
--Get all order details.  
SELECT customer_first, customer_last, order_date, description, item_quantity, line_price  
FROM Customer  
JOIN Customer_order ON customer_order.customer_id = Customer.customer_id  
JOIN Line_item ON line_item.order_id = customer_order.order_id  
JOIN Item ON item.item_id = line_item.item_id  
ORDER BY customer_first, customer_last, order_date, description;
```

Messages Data Output

customer_first character varying (32)	customer_last character varying (32)	order_date date	description character varying (64)	item_quantity numeric (10)	line_price numeric (12,2)
Elizabeth	Johnson	2005-12-19	Cup	132	1584.00
Elizabeth	Johnson	2005-12-20	Fork	3	15.00
John	Smith	2005-12-17	Knife	10	50.00
John	Smith	2005-12-17	Plate	95	950.00
John	Smith	2005-12-18	Bowl	36	396.00
John	Smith	2005-12-18	Plate	10	100.00
John	Smith	2005-12-18	Spoon	2	10.00
John	Smith	2005-12-19	Fork	3	15.00
Mary	Berman	2005-12-18	Fork	3	15.00
Peter	Quigley	2005-12-17	Bowl	2	10.00
Peter	Quigley	2005-12-17	Spoon	5	25.00
Peter	Quigley	2005-12-18	Knife	3	15.00

In summary, you have seen two methods that use sequences to retrieve the correct foreign key values. The first method is using a variable to store the value so it can be used later when the foreign key values are needed. This method works when we can control the order of inserts, and we insert the referencing values immediately after the referenced values (such as inserting a customer's orders immediately after inserting the customer). The second method is using a subquery lookup. With the second method, there is less concern about the order of inserts and referencing values don't need to be inserted immediately after referenced values. Instead, the foreign key value can be retrieved with a subquery.

Armed with both of these methods, you can now complete this step.

Step 3 – Create Hardcoded Procedure

To demonstrate something similar, we'll create a stored procedure named "add_customer_harry" that has no parameters and adds a customer named Harry Joker to the customer order schema. Below is code for this procedure.

Code: Creating add_customer_harry Procedure

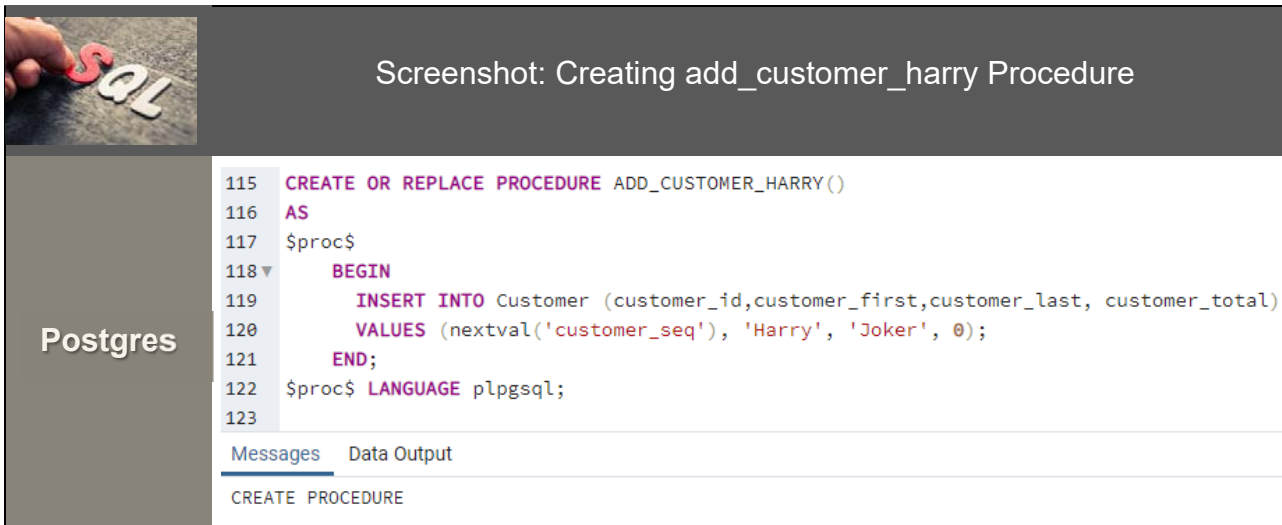
```
CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY()  
AS  
$proc$  
    BEGIN  
        INSERT INTO Customer (customer_id,customer_first,customer_last, customer_total)  
        VALUES (nextval('customer_seq'), 'Harry', 'Joker', 0);  
    END;  
$proc$ LANGUAGE plpgsql;
```

Let us go through code line by line and discuss the meaning.

Line 1: CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY()	
	<p>The CREATE OR REPLACE PROCEDURE phrase indicates to PostgreSQL that a stored procedure is to be created, and that the creation replaces any existing definition of a function with the same name. If we instead use the phrase CREATE PROCEDURE, then PostgreSQL would create the procedure only if one with the same name does not exist. All of the words in this phrase are SQL <i>keywords</i>, meaning that they are words predefined in the SQL language to have a specific meaning. We must use keywords exactly as they are defined by the language in order to tell the language what we want to do.</p> <p>The ADD_CUSTOMER_HARRY() word is the name of the procedure. This name is an <i>identifier</i>, which means that the language allows us to define our own name. PostgreSQL does restrict the length of identifiers to be no longer than 63 characters by default, and has some character restrictions, for example, that identifiers should not contain the "%" character. Within these restrictions we can specify any name we like. Of course, it is best to name a function reasonably based upon the function it performs. For this procedure, I chose the name ADD_CUSTOMER_HARRY because the logic of the procedure inserts a customer named "Harry" into the Customer table. PostgreSQL relaxes many of its restrictions on an identifier if the identifier is quoted; however, it is best not to use an identifier that must be quoted each time it is used, so we stay within the unquoted identifier guidelines.</p>
Line 2: AS	
	<p>AS is a SQL keyword that is required by the language to define a function, but otherwise has no significant meaning. SQL is defined to be natural for English speakers, so the full phrase CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY() AS leads an English speaker to naturally think that the definition of the function follows the AS keyword.</p>
Line 3: \$proc\$	

	In PostgreSQL, the procedural language always exists within a <i>block</i> within SQL. This block is simply a string literal from the prospective of the CREATE OR REPLACE PROCEDURE command is concerned. Therefore we use dollar quoting, denoted by the characters “\$\$”, to write the procedure body instead of using single quotes. Single quote syntax will also work; however, it is recommended to use the \$\$ syntax for readability, to mitigate confusion, and to mitigate conflicts with more procedure function bodies that require quotations. The “proc” in between the dollar signs is a label that adds to readability and is optional.
Line 4: BEGIN	
	In PostgreSQL, the procedural language always exists within a <i>block</i> within SQL. The BEGIN keyword is part of the opening definition of a block, and is always accompanied with an END keyword which closes the block. We can also embed some nonprocedural (declarative) SQL commands within a block, so that procedural constructs coexist with nonprocedural SQL commands. However, it is important to note that embedding SQL inside of the procedural language is different than using SQL outside of a procedural block. Inside of a procedural block, only certain SQL commands can be embedded, and the commands are expected to fit within the overall flow of the procedural language. Outside of a procedural block, SQL commands are simply executed to produce results, and the commands are not intertwined with procedural constructs. In this case, I use the BEGIN keyword as part of the beginning of the block for the ADD_CUSTOMER_HARRY() procedure.
Lines 5-6: INSERT INTO CUSTOMER (customer_id,customer_first,customer_last,customer_total) VALUES(nextval('customer_seq'), 'Harry', 'Joker', 0);	
	You might think this command looks familiar, because it does! It is just a standard SQL statement. Wait a minute. The procedural language and the SQL language are two different languages, right? So why is this SQL statement inside of the procedural language block? Simple! Because certain SQL commands can be embedded inside of the procedural language in the right context. In this case, I have embedded an insert statement that inserts a new customer “Harry” into the Customer table. This way, when you execute this function, the function will insert the new customer on your behalf, without the need for you to type the SQL command yourself.
Line 7: END;	
	The END keyword tells PostgreSQL that the procedural block, and therefore the procedure definition, ends.
Line 8: \$\$proc\$ LANGUAGE plpgsql	
	In this final line you see the \$\$ again, because the procedure body is actually a quoted literal, and this second set of “\$\$” characters ends the quoted block. Unlike Oracle and MS SQL, PostgreSQL has built in support for a few different procedural languages, and when using procedural code, you must specify to the compiler which language is being used. Here we are declaring the language to be plpgsql; which is one of the pre-loaded procedural languages supported by PostgreSQL.

Now to be clear, the above code, when executed, creates the stored procedure so that it’s available for use. Below is a screenshot of creating this stored procedure.



Notice that the output indicates that the procedure was compiled. This is a technical way for the DBMS to state that the procedure has been processed and is available for use.

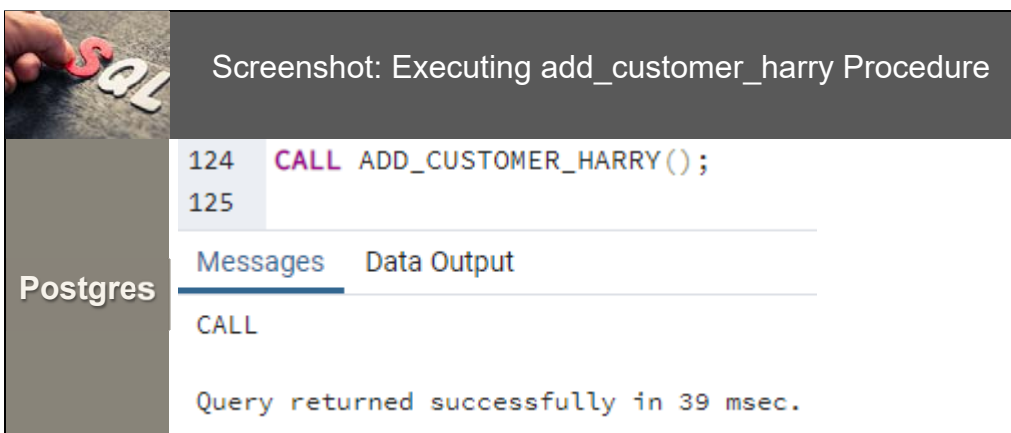
Now that we have created the stored procedure, we need to execute it for its code to take effect. First, let's look at the code to do so.

Code: Executing add_customer_harry Procedure

```
CALL ADD_CUSTOMER_HARRY();
```

The **CALL** keyword is used to execute the stored procedure we have created. We used the name of our stored procedure, **ADD_CUSTOMER_HARRY**, to specify which stored procedure to execute. The invocation is expecting any parameters to be passed to the procedure; since we have no parameters, we simply use **()** with no parameters.

Below is a screenshot of executing this code.



We can now select all rows from the Customer table to make sure that our stored procedure inserted a row as we would expect.



Screenshot: Customer Table After Execution

Postgres

126 SELECT *
127 FROM Customer;

Messages Data Output

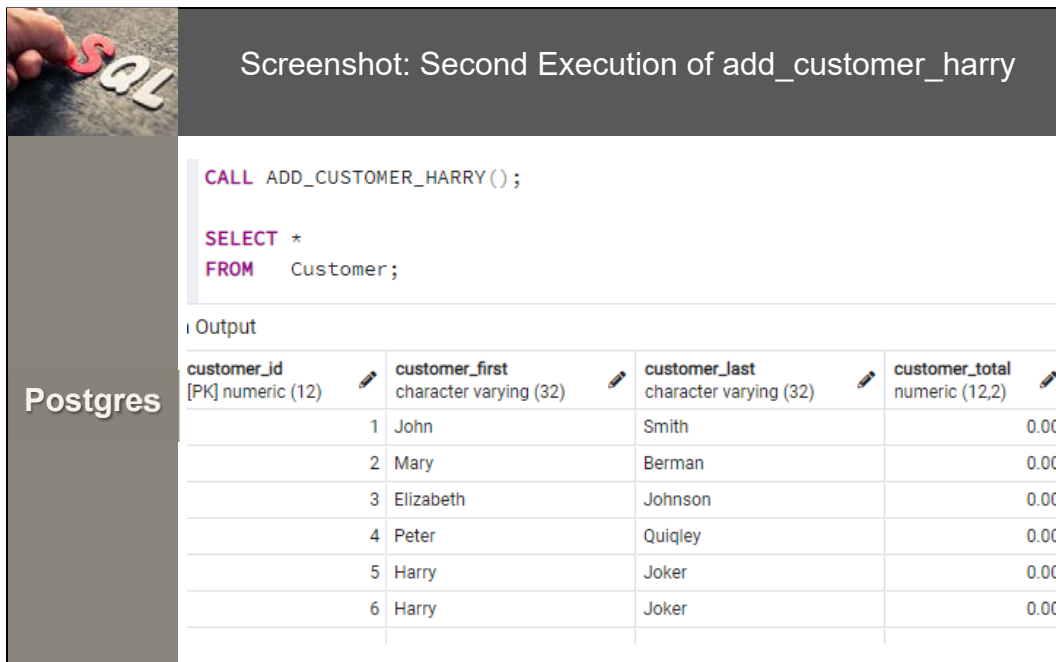
	customer_id [PK] numeric (12)	customer_first character varying (32)	customer_last character varying (32)	customer_total numeric (12,2)
1	1	John	Smith	0.00
2	2	Mary	Berman	0.00
3	3	Elizabeth	Johnson	0.00
4	4	Peter	Quiqley	0.00
5	5	Harry	Joker	0.00

Sure enough, we see that the customer “Harry Joker” is listed in the table as the last row listed. We have now successfully created and executed a stored procedure!

You can now create similar code to complete this step.

Step 4 – Create Reusable Procedure

Before we create a reusable procedure, let us look at what happens if we attempt to execute the `add_customer_harry` procedure a second time. Realistically the `ADD_CUSTOMER_HARRY` stored procedure can be meaningfully executed only once. Inside the procedure, we placed the literal value “Harry” for the `customer_first` column, the literal value “Joker” for the `customer_last` column, and the literal value “0” for the `customer_total` column. This placement is termed “hardcoding” by computer programmers, a term which means that a value is embedded directly into the source of a program, instead of obtaining the value dynamically. Executing it a second time would result in inserting the same person again, just with a different primary key. This is illustrated below.



Screenshot: Second Execution of `add_customer_harry`

```
CALL ADD_CUSTOMER_HARRY();
```

```
SELECT *
```

```
FROM Customer;
```

Output

customer_id [PK] numeric (12)	customer_first character varying (32)	customer_last character varying (32)	customer_total numeric (12,2)
1	John	Smith	0.00
2	Mary	Berman	0.00
3	Elizabeth	Johnson	0.00
4	Peter	Quigley	0.00
5	Harry	Joker	0.00
6	Harry	Joker	0.00

Notice that Harry Joker has been inserted a second time with the next primary key value. This obviously causes many issues, anomalies resulting from data redundancy perhaps being the most significant. Which record should Harry’s orders be tied to? Will there be orders tied to both records? What if Harry has a name change? Should we delete one of these records, and if so, which one? These are valid questions!

The root problem is that the stored procedure is not reusable. Every time it is invoked, it will add Harry Joker and only Harry Joker. The stored procedure is only useful for one execution, which essentially defeats the point of creating the logic in a stored procedure. One significant purpose of a stored procedure is to encapsulate logic so it can be invoked again and again by name. This stored procedure has not achieved that purpose. You should get similar results when you execute your stored procedure a second time.


It is best to make our stored procedures reusable, so that they can be executed wherever the logic contained in them is needed. The fact that our `ADD_CUSTOMER_HARRY` stored procedure cannot meaningfully be executed multiple times makes it less valuable as a resource. To achieve reusability, instead of hardcoding literal values in a procedure, we use placeholders which instruct the DBMS to use whatever value is given to the stored procedure when it is executed. These placeholders are termed *parameters*. At a minimum, a parameter has a name, which is an identifier by which it is referred, and a datatype, which determines the set of legal values that can be assigned to the parameter.

For example, instead of hardcoding the value “Harry” for the *first_name* column, we can define a *first_name_arg* parameter with a datatype of “VARCHAR”. The parameter the particular value can be specified when the stored procedure is executed. Below is an ADD_CUSTOMER stored procedure that makes use of parameters and is therefore reusable, allowing us to add any customer rather than just one specific customer. Comments next to the parameters help explain their purpose.

Code: Creating add_customer Procedure

```
CREATE OR REPLACE PROCEDURE ADD_CUSTOMER( -- Create a new customer
  first_name_arg IN VARCHAR, -- The new customer's first name
  last_name_arg IN VARCHAR) -- The new customer's last name
LANGUAGE plpgsql
AS -- Required by the syntax, but it doesn't do anything in particular
$resuableproc$ --opens the $$ quotation for the block
BEGIN
  INSERT INTO CUSTOMER(customer_id,customer_first,customer_last,customer_total)
  VALUES(nextval('customer_seq'),first_name_arg,last_name_arg,0);
  -- We start the customer with zero balance.
END;
$resuableproc$; -- closes the $$ quotation for the block
```

Notice that instead of hardcoding particular values in the insert statement, the parameter names are referenced instead, particularly in the `VALUES(nextval('customer_seq'), @first_name_arg, @last_name_arg,0)` part of the insert statement. *first_name_arg* and *last_name_arg* parameters are used in place of hardcoded “Harry” and “Joker” values. Essentially, this is instructing the SQL engine to insert whatever values are passed into the stored procedure when it is executed. Creating the stored procedure looks as follows.



Screenshot: Creating add_customer Procedure

```

129 CREATE OR REPLACE PROCEDURE ADD_CUSTOMER( -- Create a new customer
130   first_name_arg IN VARCHAR, -- The new customer's first name
131   last_name_arg IN VARCHAR) -- The new customer's last name
132   LANGUAGE plpgsql
133 AS -- Required by the syntax, but it doesn't do anything in particular
134 $resuableproc$ --opens the $$ quotation for the block
135 BEGIN
136   INSERT INTO CUSTOMER(customer_id,customer_first,customer_last,customer_total)
137   VALUES(nextval('customer_seq'),first_name_arg,last_name_arg,0);
138   -- We start the customer with zero balance.
139 END;

```

Postgres

Messages

CREATE PROCEDURE


Query returned successfully in 34 msec.

When the stored procedure is executed, the parameter values are specified by the executor. Example code for executing this stored procedure is below. Notice that the parameters are specified within parentheses, and separated by a comma.

Code: Executing add_customer Procedure

```
CALL ADD_CUSTOMER('Mary', 'Smith');
```

Notice that 'Mary', 'Smith' part of the stored procedure call which specifies what parameters to use. The order in which the parameters appear matters, as this ordering is correlated with the ordering the parameters are declared in the stored procedure. Since "Mary" comes first, it's matched to *first_name_arg*, and "Smith" is matched to *last_name_arg*. Using this approach, you need to know the order in which the parameters are declared in the stored procedure in order to execute the stored procedure. Executing the stored procedure gives us the same confirmation we saw with the prior execution of the "add_customer_harry" procedure, as shown below.



Screenshot: Executing Parameterized add_customer Procedure

Postgres

```

144 CALL ADD_CUSTOMER('Mary', 'Smith');
145
146


```

Messages

CALL

Query returned successfully in 32 msec.

Just to make sure Mary Smith made it in, we'll list out our Customer table again, illustrated below.



Screenshot: Listing Customer Table After Add

Postgres

```

126 SELECT *
127 FROM Customer;
128

```

Data Output

	customer_id [PK] numeric (12)	customer_first character varying (32)	customer_last character varying (32)	customer_total numeric (12,2)
1	1	John	Smith	0.00
2	2	Mary	Berman	0.00
3	3	Elizabeth	Johnson	0.00
4	4	Peter	Quigley	0.00
5	5	Harry	Joker	0.00
6	6	Harry	Joker	0.00
7	7	Mary	Smith	0.00

Notice that Mary Smith is now listed in the table. More importantly, we could add many more customers using this stored procedure just by changing the parameter values given to the stored procedure!

Hopefully this gives you an idea of the usefulness of parameterized stored procedures. You can code the logic once, then execute the stored procedure whenever you need it. For example, the logic we put into this procedure is:

- to use customer_seq to generate a unique primary key value for the new customer.
- to use the parameters given for the first and last name.
- to initialize the new customer's balance to 0.

We could do the above manually again and again each time we insert a new customer. But doing so is error prone and less convenient. Of course, the logic above is fairly simple so only saves us minimal work; however, you will end up putting much more logic than this into more complex procedures, including parameter validations.

You can now use a similar approach to address this step.

Step 5 – Create Deriving Procedure

You learned in the prior step how to create reusable stored procedures by using parameters, so using a variable is the new skill for this step. The basic concepts of variables are not too complex. A variable is a named placeholder that can store a value, and can later be referenced by name to retrieve the stored value.

Let's take an example from the customer order schema we have been using throughout this section. What if, instead of hardcoding the item code for an item, we wanted the database to assign it a unique value? What we could do is, create a variable, calculate the item code and store it in the variable, then reference the variable when inserting into the item table. Let's first illustrate this in pseudo-code so that you understand the concepts.

Pseudocode for Basic Variable Use

```
1: Decl are vari able v_i tem_code as a character string
2: Calcul ate a unique value and store it in the v_i tem_code vari able
3: Insert whatever value is in the v_i tem_code vari able into the i tem
   tabl e
```

In line 1 in the pseudocode, the `v_item_code` variable is declared. A *variable declaration* identifies the existence, name, and datatype of the variable. In programmatic SQL (and also in many programming languages), a variable cannot be used unless it is first declared. Its name identifies how the variable will be later referenced, and its datatype indicates what kind of value it can store (such as character string, number, date, etc ...)

In line 2 in the pseudocode, the variable is assigned a value. A *variable assignment* places a value into the variable. Referencing the variable later will use the value assigned.

In line 3, the variable is used by referencing it by name. A *variable reference* uses whatever value is in the variable. Of course, the references to the variable are what makes a variable useful, since simply declaring one and assigning a value to it would not be useful alone.

Now that you understand the pseudocode, let's look at the stored procedure code then analyze the lines.

Code: Creating add_item Procedure

```
CREATE OR REPLACE PROCEDURE ADD_ITEM(  
    p_description IN VARCHAR, -- The item's description  
    p_price IN DECIMAL)      -- The item's price  
    LANGUAGE plpgsql  
AS  
$$  
DECLARE  
    v_item_code VARCHAR(4); --Declare a variable to hold an item_code value.  
BEGIN  
    --Calculate the item_code value and put it into the variable.  
    v_item_code := SUBSTRING(p_description FROM 1 FOR 1) || FLOOR(RANDOM() * 1000);  
  
    --Insert a row with the combined values of the parameters and the variable.  
    INSERT INTO ITEM (item_id, description, price, item_code)  
    VALUES(nextval('item_seq'), p_description, p_price, v_item_code);  
END;  
$$; (NEXT VALUE FOR item_seq, @p_description, @p_price, @v_item_code);  
END;
```

First, you'll notice that the procedure is named "add_item" since it allows for adding an item to the database. Next, you'll notice that two of the four values needed in the Item table – description and price – all have corresponding parameters. The executor will decide what these values are whenever the stored procedure is invoked (you have already witnessed this strategy in the prior step).

Notice the `v_item_code VARCHAR(4);` code that sits between the `DECLARE` and the `BEGIN` statements. This line is the variable declaration, where we indicate the variable exists (by the existence of the declaration), give the variable its name (`v_item_code`), and its datatype (`VARCHAR(4)`). We give it that datatype since that is the same datatype as found in the table. This variable declaration sets up the variable so it can be assigned values and its values can be retrieved.

Next, you'll notice the `v_item_code := SUBSTRING(p_description FROM 1 FOR 1) || FLOOR(RANDOM() * 1000);` line. There are several pieces of code here you may not recognize, but don't let that keep you from understanding the basic fact that this is the line that sets the value for the variable. What we are setting it to is the first character of the description, followed by a random 3-digit number. For example, if the item description is "Napkin", then the item code would start with "N" since that is the first letter, followed by a random 3-digit number. If the database randomly selects 867 for example, then the item code would be "N867".

The `SUBSTRING` function in Postgres returns a portion of a character string. The `SUBSTR(p_description FROM 1 FOR 1)` code indicates to start at the first character and to grab 1 character from there, thereby retrieving the first character. The `RANDOM()` obtains a random number between 0 and 1, so we multiply it times 1000 to get a number between 0 and 999. The `FLOOR` function chops off the decimal point and leaves the nearest whole number. When the results of these functions are combined through concatenation, it results in a 4-character item code as described above.

Last, you'll notice the insert line, which inserts the values into the Item table, with a reference to "v_item_code" for the item_code value. Referencing the variable instructs the SQL engine to pull the value stored in the variable.

Let's try out compiling and executing the stored procedure to add a "napkin" item.



Screenshot: Compiling and Executing add_item

```
148 CREATE OR REPLACE PROCEDURE ADD_ITEM(  
149     p_description IN VARCHAR, -- The item's description  
150     p_price IN DECIMAL)      -- The item's price  
151     LANGUAGE plpgsql  
152 AS  
153 $$  
154 DECLARE  
155     v_item_code VARCHAR(4); --Declare a variable to hold an item_code value.  
156 BEGIN  
157     --Calculate the item_code value and put it into the variable.  
158     v_item_code := SUBSTRING(p_description FROM 1 FOR 1) || FLOOR(RANDOM() * 1000);  
159  
160     --Insert a row with the combined values of the parameters and the variable.  
161     INSERT INTO ITEM (item_id, description, price, item_code)  
162     VALUES(nextval('item_seq'), p_description, p_price, v_item_code);  
163 END;  
164 $$;  
165  
166 CALL ADD_ITEM('Napkin', 1);  
167
```


Messages

CALL

Query returned successfully in 36 msec.

Postgres

We execute the add_item stored procedure with parameter, similar to how we executed the add_customer procedure in the prior step. Let's look at the item table now to see if our item was added.



Screenshot: Listing Item Table

```
168 SELECT *  
169 FROM Item;
```

Data Output

	item_id [PK] numeric (12)	description character varying (64)	price numeric (10,2)	item_code character varying (4)
1		1 Plate	10.00	P001
2		2 Bowl	11.00	B002
3		3 Knife	5.00	K003
4		4 Fork	5.00	F004
5		5 Spoon	5.00	S005
6		6 Cup	12.00	C006
7		7 Napkin	1.00	N567

Postgres

Sure enough, the Napkin item was added, and the item_code value was automatically calculated rather than being passed in as a parameter by the executor. We achieved something new!

There is one more important concept you need to understand about variables to make effective use of them. *Variable scope* is the region in which a variable is accessible. In the examples in this step, we declare the

variable within the stored procedure, which means that the variable is only accessible within that same stored procedure. Another stored procedure, or the SQL engine itself, cannot access the variable. When the procedure is invoked, the variable becomes accessible by code in the procedure. Unless a value is given in its declaration, the variable is initialized to null, and another line of code can explicitly set its value to something else.

What about multiple executions of the same procedure? Does the variable's value remain across executions? The simple answer is no. Every time the procedure is invoked, the variable's value is initialized and available only to that particular execution. Different executions of the stored procedure have access to different values of the variable. That is, even though it appears multiple executions are accessing the same variable since it carries the same and declaration, *each execution has its own copy of the variable* so that each execution can use the variable independent of another execution.

Hopefully the examples in this step help illustrate one purpose of using variables. A variable provides a place to store values, which can be calculated by using expressions, and then the variable can later be referenced to retrieve its value.

You now have enough knowledge to create the stored procedure for this step.

Step 6 – Create Lookup Procedure

What you're being asked to do is certainly becoming more complex, but don't worry, you already have most of the skills you need. You already know how to create parameterized stored procedures, and declare and use variables. The one skill you have not learned yet is setting the value of a variable based upon the results of a query. Since your procedure will be given a username and not a person_id, you will need to look this up by executing a query. There is a way to do this and store it into a variable.

We'll demonstrate how to do this by creating an "add_line_item" stored procedure that supports adding line items to the database. Rather than the executor specifying the item_id, the stored procedure will take the item_code as a parameter, then lookup the item_id. The other parameters will be specified as usual. Such a procedure can look like this.

Code: ADD_LINE_ITEM procedure

```
CREATE OR REPLACE PROCEDURE ADD_LINE_ITEM(  
  p_item_code IN VARCHAR,    -- The code of the item.  
  p_order_id IN DECIMAL,     -- The ID of the order for the line item.  
  p_quantity IN DECIMAL)     -- The quantity of the item.  
  LANGUAGE plpgsql  
AS $$  
DECLARE  
  v_item_id DECIMAL(12);      --Declare a variable to hold the ID of the item code.  
  v_line_price DECIMAL(12,2); --Declare a variable to calculate line price.  
BEGIN  
  --Get the item_id based upon the item_code, as well as the line total.  
  SELECT item_id, price * p_quantity  
  INTO   v_item_id, v_line_price  
  FROM   Item  
  WHERE  item_code = p_item_code;  
  
  --Insert the new line item.  
  INSERT INTO LINE_ITEM(item_id, order_id, item_quantity, line_price)  
  VALUES(v_item_id, p_order_id, p_quantity, v_line_price);  
END;  
$$;
```

There are four columns in the line_item table – item_id, order_id, item_quantity, and line_price. Order_id and item_quantity are specified explicitly as parameters, so the executor decides what values to pass in explicitly. However, the other two are not specified explicitly, but are looked up by using the item_code. Notice that there are two variables declared, v_item_id and v_line_price; these will be used to store these values. It's this select statement that is interesting for this step.

```
SELECT item_id, price * p_quantity  
INTO   v_item_id, v_line_price  
FROM   Item  
WHERE  item_code = p_item_code;
```

The standard SELECT, FROM, WHERE clauses combined are retrieving the item corresponding to the item code passed in as a parameter, and pulling back the item_id column, and calculating what the line price would be by multiplying the price times the quantity specified. You might have guessed that the clause we have not dealt with before, INTO (on the second line of the query), instructs the SQL engine to put the values retrieved into variables. The list of variables is correlated to the list of columns. That is, the item_id column corresponds to


the v_item_id variable since they come first in both lists, and the price * quantity column corresponds to the v_line_price variable since they come second in both lists.

We refer to this statement as a whole as a SELECT INTO statement. After it is executed, both variables are populated with the values retrieved in the select. Do you see the power of the SELECT INTO statement? You can use it to lookup values in other tables, store them in variables, then later use those variables as needed. In our case, we are using the SELECT INTO to determine what the item_id and line_price values should be, then using those variables in our insert statement.

Next, let's use our function to add a line item for an order, where three "fork" items are added. As a reminder, the "fork" item has these values:

item_id	description	price	item_code
4	Fork	5	F004

It's item_code is "F004", its price is \$5, and its ID is 4. Here is a screenshot of the code used to add three fork items to an order (order with id 8).



Screenshot: Compiling and Executing add_line_item

```
171 CREATE OR REPLACE PROCEDURE ADD_LINE_ITEM(  
172     p_item_code IN VARCHAR, -- The code of the item.  
173     p_order_id IN DECIMAL,  -- The ID of the order for the line item.  
174     p_quantity IN DECIMAL)  -- The quantity of the item.  
175     LANGUAGE plpgsql  
176 AS $$  
177 DECLARE  
178     v_item_id DECIMAL(12);    --Declare a variable to hold the ID of the item code.  
179     v_line_price DECIMAL(12,2); --Declare a variable to calculate line price.  
180 BEGIN  
181     --Get the item_id based upon the item_code, as well as the line total.  
182     SELECT item_id, price * p_quantity  
183     INTO   v_item_id, v_line_price  
184     FROM   Item  
185     WHERE  item_code = p_item_code;  
186  
187     --Insert the new line item.  
188     INSERT INTO LINE_ITEM(item_id, order_id, item_quantity, line_price)  
189     VALUES(v_item_id, p_order_id, p_quantity, v_line_price);  
190 END;  
191 $$;  
192  
193 CALL ADD_LINE_ITEM('F004', 8, 3);  
194
```

Postgres

Messages

CALL

Query returned successfully in 29 msec.

The ADD_LINE_ITEM procedure was invoked, referencing the "F004" item code, order id 8, and a quantity of 3. Now let's see if our results made it into the table by selecting all line items for order 8.



Screenshot: Listing Line_item After Execution

Postgres

```
195 SELECT *
196 FROM Line_item
197 WHERE order_id = 8;
```

Data Output

	order_id [PK] numeric (12)	item_id [PK] numeric (12)	item_quantity numeric (10)	line_price numeric (12,2)
1	8	3	3	15.00
2	8	4	3	15.00

Notice that last line in the results is the one we just inserted using the procedure! Order with ID 8 now has an additional line item for forks (with ID 4), quantity 3, and a price of \$15 (since $\$5 * 3 = \15).

We are able to use this procedure to automatically calculate the line price, and to retrieve the correct item, all with its item code. Do you see now the power of lookups and storing the values in? It gives a new dimension to your stored procedures, as your procedures can now take parameters, lookup values in various tables, and use them as needed. You're on your way to becoming an expert!

Now you can use a similar strategy to create your procedure.

Step 7 – Single Table Validation Trigger

To demonstrate how to do this, we will create a trigger that prevents the customer balance from being negative. This validation only needs information from the table being modified and so qualifies as an intra-table validation. In Postgres, we cannot directly define a trigger as in Oracle or SQL Server. Rather, we must first define a function that contains the logic, then add a trigger that uses the function. Below is the code for such a trigger.

Code: No Negative Balance Validation Trigger


```
CREATE OR REPLACE FUNCTION no_neg_bal_func()  
RETURNS TRIGGER LANGUAGE plpgsql  
AS $trigfunc$  
BEGIN  
    RAISE EXCEPTION USING MESSAGE = 'Customer balance cannot be negative.',  
    ERRCODE = 22000;  
END;  
$trigfunc$;  
  
CREATE TRIGGER no_neg_bal_trg  
BEFORE UPDATE OR INSERT ON Customer  
FOR EACH ROW WHEN(NEW.customer_total < 0)  
EXECUTE PROCEDURE no_neg_bal_func();
```

Since we have not yet reviewed triggers, let's examine this line by line.

Lines 1-4: CREATE OR REPLACE FUNCTION no_neg_bal_func() RETURNS TRIGGER LANGUAGE plpgsql AS \$trigfunc\$ BEGIN	
	Notice that we use the familiar CREATE OR REPLACE FUNCTION syntax to create the function. One difference is that we must indicate that a trigger is returned rather than VOID by using RETURNS TRIGGER instead of RETURNS VOID (RETURN VOID is used for creating “stored procedure” functions). The \$trigfunc\$ and BEGIN keywords start the function body.
Lines 5-6: RAISE EXCEPTION USING MESSAGE = 'Customer balance cannot be negative.', ERRCODE = 22000;	
	In this case we invoke the RAISE EXCEPTION command, which provides a human-readable error message along with an error code, throws an exception, and rolls back the transaction. When the function is invoked by the trigger, then it will raise this exception.
Lines 7-8: END; \$trigfunc\$;	
	This ends the block and definition of the function.
Line 9: CREATE TRIGGER no_neg_bal_trg	
	The CREATE OR REPLACE TRIGGER phrase indicates to Postgres that a trigger is to be created, and that the creation replaces any existing definition of the

	<p>trigger. If we instead use the phrase CREATE TRIGGER, then Oracle would create the trigger only if one with the same name does not exist.</p> <p>The <code>no_neg_bal_trg</code> word is the name of the trigger, an identifier of our own choosing. We put “trg” at the end of the identifier as a convention, so it’s recognizable as the name of a trigger. The rest of the name helps describe what the trigger does, which is validate that there is no negative balance.</p>
Line 10:	BEFORE UPDATE OR INSERT ON Customer
	<p>BEFORE is a SQL keyword that instructs Postgres the trigger is to be executed before the insert or update occurs in the database. UPDATE OR INSERT indicates that the trigger is to be fired when either an insert or an update statement happens on the table. If we had omitted “INSERT” for example, then the trigger would only fire when an update occurs. We want to block all negative balances so we have the trigger fire on both updates and inserts. ON Customer ties to trigger to the Customer table, that is, indicates to Postgres that the aforementioned actions (update and insert) will fire this trigger only if they happen on the Customer table. Triggers are inexorably linked to one table by definition. If the table is dropped, the trigger is also automatically dropped.</p>
Line 11:	FOR EACH ROW WHEN(NEW.customer_total < 0)
	<p>This tells Postgres that the trigger is a <i>row-level trigger</i>, which means it is executed <i>once for each row</i> that is affected by the triggering SQL statement. For example, an UPDATE statement can affect many rows in a table, and the FOR EACH ROW words ensure the trigger is executed once for each affected row. If these words were omitted, the trigger would be a statement-level trigger, which would mean it would execute only once regardless of the number of rows affected.</p> <p>The WHEN(NEW.customer_total < 0) words create a condition for the trigger. The trigger will only fire when the associated Boolean expression is true. NEW is a pseudo-table provided by Postgres in triggers and triggering functions that contains all of the same columns as the table the trigger is attached to (in this case, the Customer table). NEW has the set of rows affected by the triggering event (in this case, the update). Since this is a row-level trigger, NEW only ever contains a single row for the row that was affected. The trigger will thus only fire when the value set by the update statement is less than 0, which is what we want.</p>
Line 12:	EXECUTE PROCEDURE no_neg_bal_func();
	<p>This line indicates that the <code>no_neg_bal_func</code> triggering function will be invoked when the trigger fires.</p>

First, we compile the trigger as illustrated in the screenshot below.

Screenshot: Compile the No Negative Balance Trigger

Postgres


```
116
117 CREATE OR REPLACE FUNCTION no_neg_bal_func()
118 RETURNS TRIGGER LANGUAGE plpgsql
119 AS $trigfunc$
120 BEGIN
121     RAISE EXCEPTION USING MESSAGE = 'Customer balance cannot be negative.',
122     ERRCODE = 22000;
123 END;
124 $trigfunc$;
125
126 CREATE TRIGGER no_neg_bal_trg
127 BEFORE UPDATE OR INSERT ON Customer
128 FOR EACH ROW WHEN(NEW.customer_total < 0)
129 EXECUTE PROCEDURE no_neg_bal_func();
130
```

Data Output Explain Messages Notifications Query History

CREATE TRIGGER

Query returned successfully in 57 msec.

As soon as the trigger is compiled successfully, it is active in the database and will execute when the triggering event happens from that point forward (until, of course, the trigger is disabled or dropped). If we attempt to insert a customer with a negative balance, the trigger will fire and reject it, shown below.

Screenshot: Test the No Negative Balance Trigger

Postgres

```
230
231 INSERT INTO Customer(customer_id, customer_first, customer_last, customer_total)
232 VALUES(11, 'Negative', 'Nelly', -10);
233
```

Data Output Explain Messages Notifications Query History

ERROR: Customer balance cannot be negative.
CONTEXT: PL/pgSQL function no_neg_bal_func() line 3 at RAISE
SQL state: 22000

Notice that the insert was immediately rejected by the trigger, and the “Customer balance cannot be negative” is reported back. We cannot execute the trigger directly, but we can see the effects of the trigger when we execute a triggering statement such as an insert.

You can use similar logic to create your validation trigger.

Step 8 – Cross-Table Validation Trigger

To demonstrate cross-table validation, imagine we want to validate the fact that the line price for a line item actually equals the quantity times the item price. The quantity is stored in the `Line_item` table while the item price is stored in the `Item` table. We can setup a trigger on the `Line_item` table to perform this validation using constructs we've already used in prior steps in this lab. Here is the code for such a trigger.

Code: Correct Line Price Validation Trigger

```
CREATE OR REPLACE FUNCTION line_price_func()
RETURNS TRIGGER LANGUAGE plpgsql
AS $$
DECLARE
    v_correct_line_price DECIMAL(12,2);
BEGIN
    SELECT NEW.item_quantity * Item.price
    INTO   v_correct_line_price
    FROM   Item
    WHERE  item.item_id = NEW.item_id;

    IF NEW.line_price <> v_correct_line_price THEN
        RAISE EXCEPTION USING MESSAGE = 'The line price is incorrect.',
        ERRCODE = 22000;
    END IF;
    RETURN NEW;
END;
$$;


CREATE TRIGGER line_price_trg
BEFORE UPDATE OR INSERT ON Line_item
FOR EACH ROW
EXECUTE PROCEDURE line_price_func();
```

You've most of the constructs here individually, but some of them, as well as the integration of them, needs more explanation. We put a `DECLARE` block so that we could declare a variable that will store the correct line price. Then the first line of the body of the function uses a query to multiply the item quantity times the item price, and stores it into the `v_correct_line_price` variable. It then uses an `if` statement to determine if the line price of the new or updated row is correct. If it's not, it raises an application error that indicates what the line price is supposed to be.

The `IF/THEN` keywords tell Postgres that the block following is only to be executed if the Boolean expression evaluates to true. `IF` statements allow us to conditionally execute code. For this trigger, we only want to reject SQL statements that attempt to create a negative balance, so we use an `if` statement. The `NEW.line_price <> v_correct_line_price` component is the Boolean expression that the `if` statement will evaluate. The `END IF` line ends the `IF` block. Any code within the block is executed conditionally.

The `RETURN` statement indicates what is to be returned back out of the function. Since this is a row-level trigger function, we must return the `NEW` pseudo-table out of the function, so that the row is actually inserted or updated as is. If we desired to modify the row being inserted or updated, we could return different values out of the function.

Let's try it out. We'll try to insert a line item with an invalid line price, as follows.



Screenshot: Test the Price Validation Trigger

Postgres

250

257

258

259

INSERT INTO

Line_item(item_id,order_id,item_quantity,line_price)

VALUES (3,7,5,100);

Data Output

Explain

Messages

Notifications

Query History

ERROR: The line price is incorrect.

CONTEXT: PL/pgSQL function line_price_func() line 11 at RAISE

SQL state: 22000

We attempted to insert a line item with a line price of 100, and the trigger rejected it because the line price should be 25. Why? Item with ID 3 has a price of 5, and there is a quantity of 5, so the line price would be 25 and not 100. We successfully used a trigger to perform a cross-table validation, simply by combining constructs we were already familiar with.

With all of these skills, you may begin feel very powerful. Just make sure to use your powers for good and not evil. You can use similar logic to create your cross-validation trigger.

Step 9 – History Trigger

To demonstrate capturing history, imagine that we would like to store a history of price changes for each item, so that we know the price of an item at any point in time despite any price updates. Abstractly, these fields should be included in a history table – a reference (foreign key) to the table being changed, the old value of the column, the new value of the column, and the date of change. You can think of this set of fields as a design pattern for history tables. For our example, we would create an `Item_price_history` table that stores a reference to the item, its old price, its new price, and the date of the change.

Before showing you code, let's make sure to differentiate history and auditing, which have two different purposes. A history table records changes over time but remains active in the schema, with proper foreign keys for references. The fields are setup so that SQL queries and transactions can use the table along with the other tables in the schema, that is, so that the table can be used regularly like any other table in the schema. The purpose of a history table is to make prior values available to the people and applications that use the database in a way that coexists with the current values.

An audit table also records changes over time, but it does not remain active in the schema. The purpose of an audit table is to simply record that a change happened in a way that people can manually review the changes later in case of any concern or dispute. An audit table has no foreign keys, and acts more like a log which contains the full value of each field. Since an audit table does not make use of foreign keys, and flattens out the needed fields, it survives schema changes over time well. For example, as already mentioned a history table for price changes would have a foreign key to the item, but an audit table would instead have the description of the item along with any other information needed to identify the item. Someone could manually review the audit table to see which prices changes over time for which items.

It's a best practice to be aware of which kind of table you're creating – history or audit – and follow the design patterns for that table. Some organizations inadvertently create hybrid tables that perhaps start out strictly for auditing, but then later add in foreign keys, and this can cause problems as changes are made to the database. In this step, we are creating a history table that remains active in the schema.

First, let's look at the code for creating the `Item_price_history` table, below.

Code: Create the `Item_price_history` Table

```
CREATE TABLE Item_price_history (  
  item_id DECIMAL(12) NOT NULL,  
  old_price DECIMAL(10,2) NOT NULL,  
  new_price DECIMAL(10,2) NOT NULL,  
  change_date DATE NOT NULL,  
  FOREIGN KEY (item_id) REFERENCES Item(item_id));
```

You're very familiar with table creation syntax at this point, so there's no need to belabor the basics of this SQL. Instead, we'll focus on what's important here. We opted to avoid giving this table a primary key; a primary key is not necessary to record the history. In some real-world schemas, the standards employed by the organization may mandate a primary key, especially if the schema will be mapped by an object-relational mapping tool within an application. There is a foreign key to the `Item` which has been changed. The `old_price` and `new_price` columns have the same datatype as the original `Item.price` column, since it will be a record of the change.

Once the table is created, we then define a trigger on the Item table that inserts a row into the Item_price_history table whenever an item price is updated. The code for the trigger is below.

Code: The Item_price_history Trigger


```
CREATE OR REPLACE FUNCTION item_history_func()
RETURNS TRIGGER LANGUAGE plpgsql
AS $$
BEGIN
    IF OLD.price <> NEW.price THEN
        INSERT INTO Item_price_history(item_id, old_price, new_price, change_date)
        VALUES(NEW.item_id, OLD.price, NEW.price, CURRENT_DATE);
    END IF;
    RETURN NEW;
END;
$$;

CREATE TRIGGER item_history_trg
BEFORE UPDATE ON Item
FOR EACH ROW
EXECUTE PROCEDURE item_history_func();
```

This trigger only fires on updates, because we only want to record changes in price. We've opted not record the initial price so we don't have the trigger fire on insert; however, one variation of the history table would record every price including the initial one. In that case, the trigger would be modified to also trigger on insert, and the old_price column would be nullable so that when the first price is created, the old_price is null (since there is no price).

The trigger only records an update if the old price is different than the new price, so an if statement is used. The OLD pseudo-table is used to retrieve the old price, and the NEW pseudo-table is used to retrieve the new price and the item ID. The keyword CURRENT_DATE is used to retrieve the current date. In this trigger, we return the NEW pseudo-table as is, because we are not looking to modify the values of the inserted row, but simply record a history.

Let's test that our trigger works by modifying the price of an item in the Item table. The compilation and update are illustrated first, below.



Screenshot: Compilation and Update for Item_price_history

Postgres

```

269 CREATE OR REPLACE FUNCTION item_history_func()
270 RETURNS TRIGGER LANGUAGE plpgsql
271 AS $$
272 BEGIN
273     IF OLD.price <> NEW.price THEN
274         INSERT INTO Item_price_history(item_id, old_price, new_price, change_date)
275         VALUES(NEW.item_id, OLD.price, NEW.price, CURRENT_DATE);
276     END IF;
277     RETURN NEW;
278 END;
279 $$;
280
281 CREATE TRIGGER item_history_trg
282 BEFORE UPDATE ON Item
283 FOR EACH ROW
284 EXECUTE PROCEDURE item_history_func();
285
286 UPDATE Item
287 SET Price=35
288 WHERE description='Plate';

```

Data Output

Explain

Messages


Notifications

Query History

UPDATE 1

Query returned successfully in 46 msec.

Next, the listing of the table itself is included, to show that the trigger recorded the update.



Screenshot: Listing Item_price_history

Postgres

```

290 SELECT *
291 FROM Item_price_history;

```

Data Output

Explain

Messages

Notifications

Query History

	item_id numeric (12)	old_price numeric (10,2)	new_price numeric (10,2)	change_date date
1	1	10.00	35.00	2019-01-25

We now see a row in the history table indicating the item with ID 1 (Plate) had an old price of 10, a new price of 35, and the change happened on the specific date. With this structure, all such price changes will be recorded over time. And following this pattern, we could record a history for whatever column we like for whatever table we need. Amazing!

You can follow this pattern to create and populate the history table for your lab.

Step 10 – Creating Normalized Table Structure

The scope and technical complexity on how to perform normalization from scratch is too broad and deep to be a part of this document. Please use the textbook, online lectures, and live classroom sessions to learn about normalization, how to identify and represent functional dependencies, and the steps to follow to normalize a table. Keep in mind that it is best practice to normalize tables to BCNF when possible. If a table is not normalized to BCNF for specific reasons, we should be aware of those reasons and make a conscious choice to do so.