

Objective

The objective of this lab is to teach you how to develop and use basic stored procedures and triggers the procedural language of your chosen DBMS.

Prerequisites

Before attempting this lab, it is best to read the textbook and lecture material covering the objectives listed above. While this lab shows you how to create and use these constructs in SQL, the lab does not explain in full the theory behind the constructs, as does the lecture and textbook.

The second section in this lab builds on Lab 3. It is best to complete Lab 3 first before completing the second section in this lab.

Required Software

The examples in this lab will execute in modern versions of Oracle, Microsoft SQL Server, and PostgreSQL as is. Note that the sections in this lab have syntax specific to each DBMS, so you will need to complete the version for the DBMS you are using.

Saving Your Data

If you choose to perform portions of the assignment in different sittings, it is important to *commit* your data at the end of each session. This way, you will be sure to make permanent any data changes you have made in your current session, so that you can resume working without issue in your next session. To do so, simply issue this command:

```
COMMIT;
```

Data changes in one session will only be visible only in that session, unless they are committed, at which time the changes are made permanent in the database.

Lab Completion

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

Lab Overview

Modern relational DBMS natively support a procedural language in addition to the declarative SQL language. Standard programming constructs are supported in the procedural language, including if conditionals, loops, variables, and reusable logic. These constructs greatly enhance the native capabilities of the DBMS. The procedural languages also support the ability to embed and use the results of SQL queries. The combination of the programming constructs provided by the procedural language, and the data retrieval and manipulation capabilities provided by the SQL engine, is powerful and useful.

Database texts and DBMS documentation commonly refers to the fusion of the procedural language and the declarative SQL language as a whole within the DBMS. Oracle's implementation is named Procedural Language/Structured Query Language, and is more commonly referred to as PL/SQL, while SQL Server's implementation is named Transact-SQL, and is more commonly referred to as T-SQL. PostgreSQL supports multiple procedural languages including PL/pgSQL which is the one used in this lab. For more information on the languages supported reference the [postgresql.org](https://www.postgresql.org) documentation. SQL predates the procedural constructs in both Oracle and SQL Server, and therefore documentation for both DBMS refer to the procedural language as an extension to the SQL language. This idea can become confusing because database texts and documentation also refer to the entire unit, for example PL/SQL and T-SQL, as a vendor-specific extension to the SQL language.

It is important for us to avoid this confusion by recognizing that there are two distinct languages within a relational DBMS – declarative and procedural – and that both are treated very differently within a DBMS in concept and in implementation. In concept, we use the SQL declarative language to tell the database *what* data we want without accompanying instruction on *how* to obtain the data we want, but we use the procedural language to perform imperative logic that explicitly instructs the database on *how* to perform specific logic. The SQL declarative language is handled in part by a SQL query optimizer, which is a substantive component of the DBMS that determines how the database will perform the query, while the procedural language is not in any way handled by the query optimizer. In short, the execution of each of the two languages in a DBMS follows two separate paths within the DBMS.

Modern relational DBMS support the creation and use of persistent stored modules, namely, stored procedures and triggers, which are widely used to perform operations critical to modern information systems. A stored procedure contains logic that is executed when a transaction invokes the name of the stored procedure. A trigger contains logic that is automatically executed by the DBMS when the condition associated with the trigger occurs. Not surprisingly stored procedures and triggers can

be defined in both PL/SQL, T-SQL and PL/pgSQL. This lab helps teach you how to intelligently define and use both types of persistent stored modules.

This lab provides separate subsections for SQL Server, Oracle, and PostgreSQL, because there are some significant differences between the DBMS procedural language implementations. The syntax for the procedural language differs between Oracle, SQL Server, and PostgreSQL which unfortunately means that we cannot use the same procedural code across all DBMS. We must write procedural code in the syntax specific to the DBMS, unlike ANSI SQL which oftentimes can be executed in both with no modifications.

The procedural language in T-SQL is documented as a container for the declarative SQL language, which means that procedural code can be written with or without using the underlying SQL engine. It is just the opposite in PL/SQL, because the declarative SQL language is documented as a container for the procedural language in PL/SQL, which means that procedural code executes within a defined block in the context of the SQL engine. PL/pgSQL is similar to Oracle's PL/SQL in that the procedural code executes in blocks and these blocks are literal strings defined by the use of Dollar quotations (\$\$). Please be careful to complete only the subsections corresponding to your chosen DBMS.

Lab Schema Setup

In this lab you will be working with the database schema defined by the SQL commands listed below, which are DDL and DML that create and populate the tables in the schema. Copy, paste, and execute this SQL before continuing to the steps in this section. There is no need to provide screenshots of executing this DDL and DML.

```
-- To avoid errors, drop tables if they exist.
DROP TABLE line_item;
DROP TABLE customer_order;
DROP TABLE customer;
DROP TABLE item;

CREATE TABLE customer(
customer_ID      DECIMAL(10) NOT NULL,
customer_first  VARCHAR(30),
customer_last   VARCHAR(40),
customer_total  DECIMAL(12, 2),
PRIMARY KEY (customer_ID));

INSERT INTO customer VALUES(1, 'John', 'Smith', 0);
INSERT INTO customer VALUES(2, 'Mary', 'Berman', 0);
INSERT INTO customer VALUES(3, 'Elizabeth', 'Johnson', 0);
INSERT INTO customer VALUES(4, 'Peter', 'Quigley', 0);
INSERT INTO customer VALUES(5, 'Stanton', 'Hurley', 0);
INSERT INTO customer VALUES(6, 'Yvette', 'Presley', 0);
INSERT INTO customer VALUES(7, 'Hilary', 'Marsh', 0);

CREATE TABLE ITEM(
item_id         DECIMAL(10) NOT NULL,
description     VARCHAR(30),
```

```

price          DECIMAL(10),
PRIMARY KEY (item_id));

INSERT INTO item VALUES(1, 'Plate',10);
INSERT INTO item VALUES(2, 'Bowl',11);
INSERT INTO item VALUES(3, 'Knife',5);
INSERT INTO item VALUES(4, 'Fork',5);
INSERT INTO item VALUES(5, 'Spoon',5);
INSERT INTO item VALUES(6, 'Cup',12);

CREATE TABLE customer_order (
order_id       DECIMAL(10) NOT NULL,
customer_id    DECIMAL(10) NOT NULL,
order_total    DECIMAL(12,2),
order_date     DATE,
PRIMARY KEY (ORDER_ID),
FOREIGN KEY (CUSTOMER_ID) REFERENCES customer);

INSERT INTO customer_order VALUES(1,1,506,CAST('18-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(2,1,1000,CAST('17-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(3,3,15,CAST('19-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(4,3,15,CAST('20-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(5,2,1584,CAST('18-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(6,4,100,CAST('17-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(7,5,40,CAST('18-DEC-2005' AS DATE));
INSERT INTO customer_order VALUES(8,1,10,CAST('19-DEC-2005' AS DATE));

CREATE TABLE line_item(
order_id       DECIMAL(10) NOT NULL,
item_id        DECIMAL(10) NOT NULL,
item_quantity  DECIMAL(10) NOT NULL,
line_price     DECIMAL(12,2),
PRIMARY KEY (ORDER_ID, ITEM_ID),
FOREIGN KEY (ORDER_ID) REFERENCES customer_order,
FOREIGN KEY (ITEM_ID) REFERENCES item);

INSERT INTO line_item VALUES(1,1,10,100);
INSERT INTO line_item VALUES(1,5,2,10);
INSERT INTO line_item VALUES(1,2,36,396);
INSERT INTO line_item VALUES(2,1,95,950);
INSERT INTO line_item VALUES(2,3,10,50);
INSERT INTO line_item VALUES(3,4,3,15);
INSERT INTO line_item VALUES(4,4,3,15);
INSERT INTO line_item VALUES(5,6,132,1584);
INSERT INTO line_item VALUES(6,1,10,100);
INSERT INTO line_item VALUES(7,5,5,25);
INSERT INTO line_item VALUES(7,4,3,15);
INSERT INTO line_item VALUES(8,5,2,10);

COMMIT;

```

Section One – Stored Procedures

Overview

Stored procedures offer many significant advantages. Reusability is one significant advantage. The logic contained in a stored procedure can be executed repeatedly, so that each developer need not reinvent the same logic each time it is needed. Another significant advantage is division of responsibility. An expert in a particular area of the database can develop and thoroughly test reusable logic, so that others can execute what has been written without the need to understand the internals of that database area. Stored procedures can be used to support structural independence. Direct access to underlying tables can be entirely removed, requiring that all data access for the tables occur through the gateway of stored procedures. If the underlying tables change, the logic of the stored procedures can be rewritten without changing the way the stored procedures are invoked, thereby avoiding application rewrites. Enhanced security accompanies this type of structural independence, because all access can be carefully controlled through the stored procedures.

Follow the steps in this section to learn how to create and use stored procedures.

Steps for Oracle

Complete these steps for Oracle only if you are using Oracle. If you are using SQL Server or PostgreSQL, steps for those DBMS' are available in a subsequent section.

1. Before you begin creating stored procedures and triggers in the schema presented in this lab, you should create a logical ERD representing the schema. It is a best practice to reference an ERD while working with a database schema, so that you can quickly ascertain the overall design of the schema, including the entities, attributes, and relationships. Create a logical ERD of the database schema now, and make sure to include all attributes, primary keys, and foreign keys.

2. We now list out the code which defines a basic stored procedure.

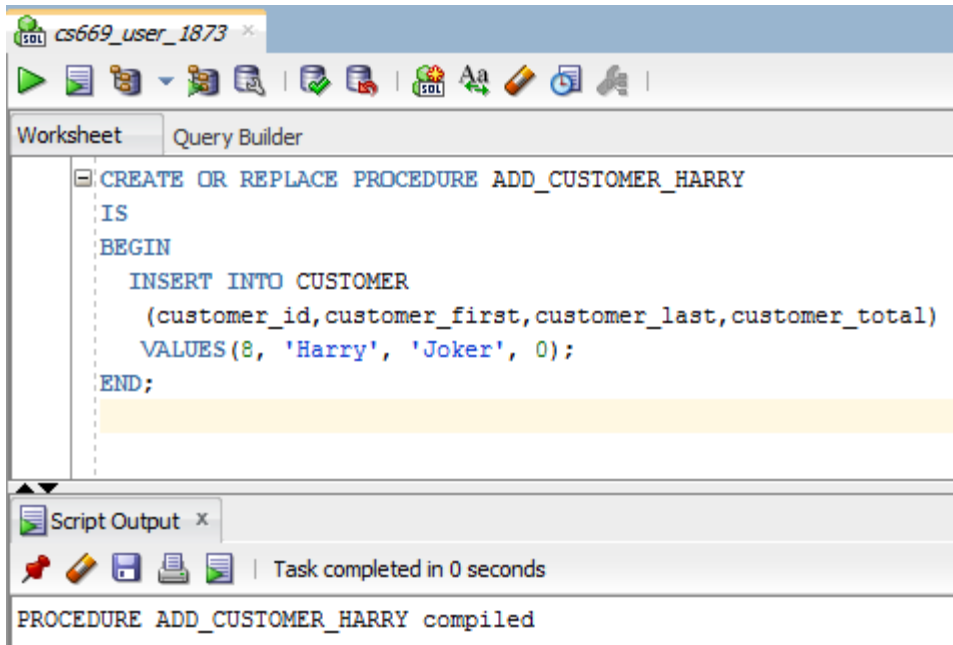
```
CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY
IS
BEGIN
    INSERT INTO CUSTOMER
        (customer_id,customer_first,customer_last,customer_total)
        VALUES(8, 'Harry', 'Joker', 0);
END;
```

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 Oracle that a stored procedure is to be created, and that the creation replaces any existing definition of a stored procedure with the same name. If we instead use the phrase CREATE PROCEDURE, then Oracle 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 stored procedure. This name is an <i>identifier</i>, which means that the language allows us to define our own name. Oracle does restrict the length of identifiers to be no longer than 30 characters, 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 stored procedure 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. Oracle 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 nonquoted identifier guidelines.</p>
Line 2: IS	
	<p>IS is a SQL keyword that is required by the language to define a stored procedure, 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 IS leads an English speaker to naturally think that the definition of the stored procedure follows the IS keyword.</p>
Line 3: BEGIN	
	<p>In Oracle, 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</p>

	<p>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 to as part of the beginning of the block for the ADD_CUSTOMER_HARRY stored procedure.</p>
	<p>Lines 4-6:</p> <pre>INSERT INTO CUSTOMER (customer_id,customer_first,customer_last,customer_total) VALUES(8, 'Harry', 'Joker', 0);</pre>
	<p>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 stored procedure, the stored procedure will insert the new customer on your behalf, without the need for you to type the SQL command yourself.</p>
	<p>Line 7: END;</p>
	<p>The END keyword tells Oracle that the procedural block, and therefore the stored procedure definition, ends.</p>

Just as creating a table and accessing that table are two distinct actions, creating a stored procedure and executing the stored procedure are two distinct actions. When we execute this code, we are instructing Oracle to create the stored procedure as defined, as illustrated in the screenshot below.



Notice the words “PROCEDURE ADD_CUSTOMER_HARRY compiled”. This tells us that Oracle created this stored procedure durably in the database.

3. Now you give it a try. Copy and paste the code of the stored procedure into your command window, then capture a screenshot illustrating its creation.
4. Now that we have created the stored procedure, we need to execute it for its code to take effect. We can do so by defining an *anonymous block* in which to execute procedural code, and then invoking the stored procedure within that block. First, let’s look at the code to do so.

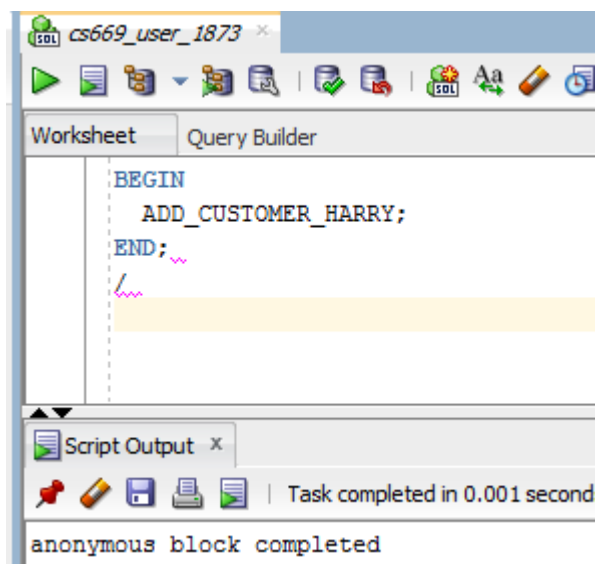
```
BEGIN
  ADD_CUSTOMER_HARRY;
END;
/
```

Again, let us visit the meaning of the code line by line.

Line 1: BEGIN
<p>Just as with a stored procedure, we use the BEGIN keyword as part of starting the block of procedural code. The difference is that we have not associated this block with any persistent stored module, and so we are creating an <i>anonymous</i> procedural block. Anonymous blocks are not durably saved to the database, and can be embedded into scripts to perform logic that does not need to be durably saved, commonly because it is a one-time action.</p>

Line 2:	ADD_CUSTOMER_HARRY;
	Here we have simply given the name of the stored procedure followed by a semicolon. In the context of a procedural block, this instructs Oracle to execute the code defined in the stored procedure.
Line 3:	END;
	The END keyword tells Oracle that the anonymous procedural block is ended. At this point, the anonymous procedural block has been defined, but has not yet been executed. This is similar to the way that a stored procedure is first defined and later executed. Why is it necessary to place a semicolon after the END keyword? By the nature of the language, procedural blocks are defined within the context of the SQL engine, but are actually executed by a separate procedural engine. So from the context of SQL, a block is just another command, and the block needs the semicolon as a statement separator, the same as any other SQL command.
Line 4:	/
	The slash (/) tells Oracle to execute the PL/SQL block most recently defined, which in our case is the block defined by the previous three lines of code. The first three lines defined the block, and this line instructs Oracle to execute the block.

Now when we execute this anonymous block, we see this in Oracle SQL Developer.



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

The screenshot shows a SQL query editor window with the title 'cs669_user_1873'. The 'Query Builder' tab is active, displaying the SQL statement: `SELECT * FROM Customer`. Below the query, the 'Query Result' tab is active, showing the results of the query. The status bar indicates 'All Rows Fetched: 8 in 0 seconds'. The results are displayed in a table with the following columns: `CUSTOMER_ID`, `CUSTOMER_...`, `CUSTOMER_LAST`, and `CUSTOMER_TOTAL`. The table contains 8 rows of data.

	CUSTOMER_ID	CUSTOMER_...	CUSTOMER_LAST	CUSTOMER_TOTAL
1	1	John	Smith	0
2	2	Mary	Berman	0
3	3	Elizabeth	Johnson	0
4	4	Peter	Quigley	0
5	5	Stanton	Hurley	0
6	6	Yvette	Presley	0
7	7	Hilary	Marsh	0
8	8	Harry	Joker	0

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

Note that by default stored procedures execute within the current transaction, so we would need to commit the transaction if we want to durably add it to the database.

5. Give it a try yourself. Execute the `ADD_CUSTOMER_HARRY` stored procedure, and once you are sure it executed, list the contents of the `Customer` table to ensure that “Harry Joker” made it into the table. Capture screenshots illustrating the execution and results of both commands.
6. Realistically the `ADD_CUSTOMER_HARRY` stored procedure can be executed only once. Inside the procedure, we placed the literal value “8” for the `customer_id` column, 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. So we conclude that this stored procedure is not reusable.

A second execution of the procedure would attempt to insert the same customer, resulting in a primary key violation. This is illustrated below.

```
BEGIN
  ADD_CUSTOMER_HARRY;
END;
```

Task completed in 0.016 seconds

Error starting at line : 1 in command -
BEGIN
 ADD_CUSTOMER_HARRY;
END;
Error report -
ORA-00001: unique constraint (CS669_USER_1873.SYS_C006998) violated
ORA-06512: at "CS669_USER_1873.ADD_CUSTOMER_HARRY", line 4
ORA-06512: at line 2
00001. 00000 - "unique constraint (%s.%s) violated"
*Cause: An UPDATE or INSERT statement attempted to insert a duplicate key.
 For Trusted Oracle configured in DBMS MAC mode, you may see
 this message if a duplicate entry exists at a different level.
*Action: Either remove the unique restriction or do not insert the key.

7. Try to execute the stored procedure a second time yourself. Capture a screenshot of the command and the results of its execution.
8. 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 the `ADD_CUSTOMER_HARRY` stored procedure cannot 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 Oracle 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, if instead of hardcoding the value “8” for the `customer_id` column, we defined a parameter named “`cus_id_arg`” with a datatype of “`DECIMAL`”, the particular value can be specified when the stored procedure is executed. Below is an `ADD_CUSTOMER` stored procedure that makes use of several parameters and is therefore reusable. Comments next to the parameters help explain their purpose.

```

CREATE OR REPLACE PROCEDURE ADD_CUSTOMER( -- Create a new customer
  cus_id_arg IN DECIMAL, -- The new customer ID, must be unused
  first_name_arg IN VARCHAR, -- The new customer's first name
  last_name_arg IN VARCHAR) -- The new customer's last name
IS -- Required by the syntax, but it doesn't do anything in particular
BEGIN
  INSERT INTO CUSTOMER
    (customer_id,customer_first,customer_last,customer_total)
  VALUES(cus_id_arg,first_name_arg,last_name_arg,0);
  -- We start the customer with zero balance.
END;

```

All that you need to do is type or copy and paste the text into your SQL client, then run it to create the stored procedure. Capture a screenshot illustrating the creation of this stored procedure in your SQL client.

9. Your next step is to invoke the procedure that you created to add a new customer. We do this with an anonymous PL/SQL block.

```

BEGIN
  ADD_CUSTOMER(9, 'Mary', 'Smith');
END;
/

```

Notice that you specify the values when you run the store procedure, in this example, “9”, “Mary”, and “Smith”. This means that we could add many more customers using this stored procedure just by changing the values given to the stored procedure.

Execute this command and capture a screenshot illustration its execution.

10. Next, we want to verify that Mary Smith was added to the Customer table with a simple select command.

```

SELECT *
FROM CUSTOMER;

```

Execute the command. Capture a screenshot illustrating its execution and results.

11. Up to this point, the precise commands have been given to you. Now you have a chance to modify the existing code without being given the precise code. Modify the ADD_CUSTOMER procedure so that it takes a fourth CUST_BALANCE IN argument which will be used to set the Customer’s balance. Make sure to also modify the

INSERT statement so that it inserts the value passed in instead of the hard-coded zero in the original definition of the stored procedure.

Capture a screenshot of the creation of your new stored procedure.

12. We will now add a new customer, making sure to take advantage of the additional balance parameter that you have provided.

```
BEGIN
  ADD_CUSTOMER(10, 'Gabriela', 'Jury', 10.99);
END;
/
```

Execute this command to insert Gabriela Jury into the Customer table with a balance of 10.99. Capture a screenshot illustrating the execution and results of this command.

13. Next, execute a SQL query that verifies that Gabriela Jury was added to the Customer table with the correct balance. Capture a screenshot illustrating the execution and results of the command.
14. In addition to reusability through the use of parameters, stored procedures can also be made more useful by executing multiple SQL commands that make up one logical unit of work. For example, what if we want to delete the Customer John Smith, as well as all of his Order and Line_item information? We would need to execute several delete statements in the correct order in order to do so. We could do so manually, but better yet we could create a parameterized stored procedure that would delete any customer's information of our choosing. We would just need to specify the customer_id when executing the stored procedure.

Do so now. Create such a stored procedure, then execute it to delete John Smith and all Order and Line_item information associated with John Smith. Capture a screenshot of the commands to create and execute the stored procedure, as well as the results of their execution. Also, be sure to capture screenshots of the SELECT statements that list out the Customer, Order, and Line_item table after the stored procedure has been executed. This will show that your stored procedure behaves as expected.

Depending upon the logic you use to solve this step, you may need to make use of a simple subquery construct to delete the line_items. Since subqueries are not covered fully in Module 4, below is the basic form of such a delete statement that makes use of a subquery:

```
DELETE FROM Line_item
WHERE order_id IN (SELECT order_id
                   FROM Customer_order
                   WHERE customer_id = 5);
```

This command instructs the database to delete any row in the Line_item table that has an order_id placed by a Customer with customer_id 5. The part in parentheses, (SELECT order_id...), is the subquery which retrieves the order ids associated to Customer 5. You will learn more details about subqueries in lab 5, but this knowledge of subqueries is sufficient for this step. Here we hardcode 5, but you will of course make your results more dynamic to solve this step.

Steps for SQL Server

Complete these steps if you are using SQL Server. If you are using Oracle or PostgreSQL, complete the corresponding steps in a different section.

1. Before you begin creating stored procedures and triggers in the schema presented in this lab, you should create a logical ERD representing the schema. It is a best practice to reference an ERD while working with a database schema, so that you can quickly ascertain the overall design of the schema, including the entities, attributes, and relationships. Create a logical ERD of the database schema now, and make sure to include all attributes, primary keys, and foreign keys.
2. We now list out the code which defines a basic stored procedure.

```
CREATE PROCEDURE ADD_CUSTOMER_HARRY
AS
BEGIN
    INSERT INTO CUSTOMER (CUSTOMER_ID,CUSTOMER_FIRST,CUSTOMER_LAST,CUSTOMER_TOTAL)
    VALUES(8, 'Harry', 'Joker', 0);
END;
```

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

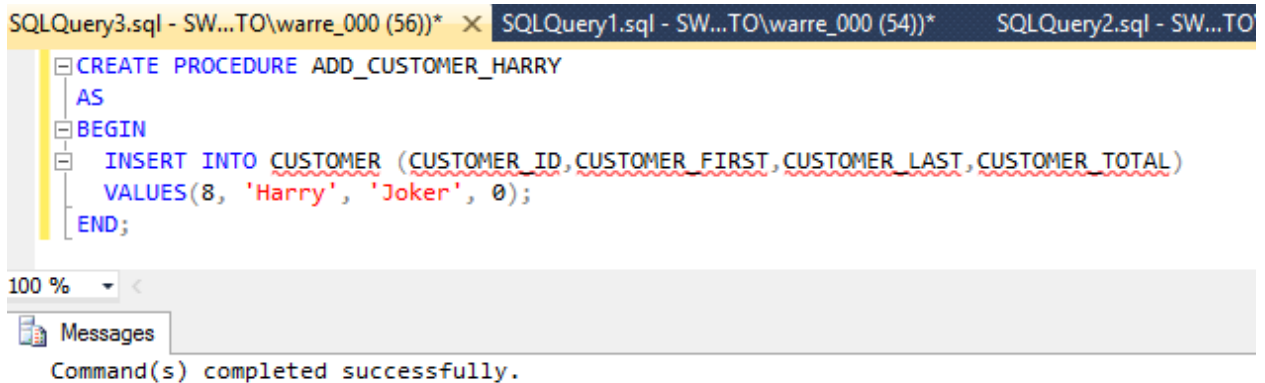
Line 1: CREATE PROCEDURE ADD_CUSTOMER_HARRY

The **CREATE PROCEDURE** phrase indicates to SQL Server that a stored procedure is to be created. All of the words in this phrase are SQL *keywords*, 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.

The **ADD_CUSTOMER_HARRY** word is the name of the stored procedure. This name is an *identifier*, which means that the language allows us to define our

	own name. SQL Server does restrict the length of identifiers to be no longer than 128 characters, 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 stored procedure 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. SQL Server relaxes many of its restrictions on an identifier if the identifier is quoted or enclosed in brackets; however, it is best not to use an identifier that must be quoted each time it is used, so we stay within the regular identifier guidelines.
Line 2: AS	
	AS is a SQL keyword that is required by the language to define a stored procedure, but otherwise has no significant meaning. SQL is defined to be natural for English speakers, so the full phrase CREATE PROCEDURE ADD_CUSTOMER_HARRY AS leads an English speaker to naturally think that the definition of the stored procedure follows the AS keyword.
Line 3: BEGIN	
	This word is optional when creating stored procedures in SQL Server. Its use helps in the readability of the stored procedure, so that one can determine at a glance where the content of the stored procedure begins. If the BEGIN word is used, it must be coupled with the END keyword described below.
Lines 4-5: INSERT INTO CUSTOMER (customer_id,customer_first,customer_last,customer_total) VALUES(8, '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 a stored procedure that uses the procedural language? 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 stored procedure, the stored procedure will insert the new customer on your behalf, without the need for you to type the SQL command yourself.
Line 6: END;	
	The END keyword is optional and is only required if the BEGIN keyword is used. This word helps in the readability of the stored procedure, so that one can determine at a glance where the content of the stored procedure ends. Likewise, the semicolon after the END keyword is optional.

Just as creating a table and accessing that table are two distinct actions, creating a stored procedure and executing the stored procedure are two distinct actions. When we execute this code, we are instructing SQL Server to create the stored procedure as defined, as illustrated in the screenshot below.



The screenshot shows a SQL Server Enterprise Manager window with three tabs: 'SQLQuery3.sql - SW...TO\warre_000 (56))*', 'SQLQuery1.sql - SW...TO\warre_000 (54))*', and 'SQLQuery2.sql - SW...TO...'. The active tab is 'SQLQuery3.sql'. The code editor displays the following SQL code:

```
CREATE PROCEDURE ADD_CUSTOMER_HARRY
AS
BEGIN
    INSERT INTO CUSTOMER (CUSTOMER_ID, CUSTOMER_FIRST, CUSTOMER_LAST, CUSTOMER_TOTAL)
    VALUES (8, 'Harry', 'Joker', 0);
END;
```

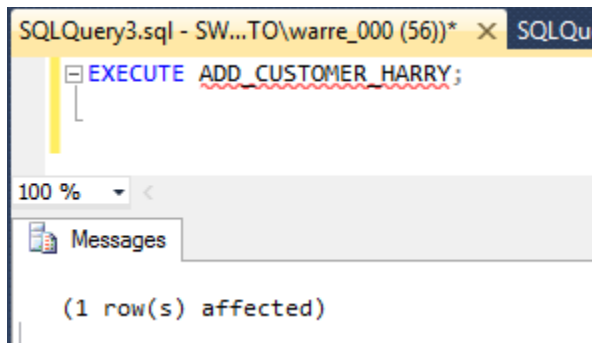
Below the code editor, a 'Messages' pane shows the status: 'Command(s) completed successfully.'

3. Now you give it a try. Copy and paste the code of the stored procedure into your command window, then capture a screenshot illustrating its creation.

4. 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.

```
EXECUTE ADD_CUSTOMER_HARRY;
```

The **EXECUTE** keyword can be used to execute many different kinds of objects in SQL Server, and in this context we use 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 screenshot of the execution is below.



The screenshot shows a SQL Server Enterprise Manager window with two tabs: 'SQLQuery3.sql - SW...TO\warre_000 (56))*' and 'SQLQuery1.sql - SW...TO\warre_000 (54))*'. The active tab is 'SQLQuery3.sql'. The code editor displays the following SQL code:

```
EXECUTE ADD_CUSTOMER_HARRY;
```

Below the code editor, a 'Messages' pane shows the status: '(1 row(s) affected)'

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

SQLQuery3.sql - SW...TO\warre_000 (56))* X SQLQuery1.sql - SW...TOV

```
SELECT *
FROM Customer;
```

100 %

Results Messages

	customer_ID	customer_first	customer_last	customer_total
1	1	John	Smith	0.00
2	2	Mary	Beman	0.00
3	3	Elizabeth	Johnson	0.00
4	4	Peter	Quigley	0.00
5	5	Stanton	Hurley	0.00
6	6	Yvette	Presley	0.00
7	7	Hilary	Marsh	0.00
8	8	Harry	Joker	0.00

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

5. Give it a try yourself. Execute the ADD_CUSTOMER_HARRY stored procedure, and once you are sure it executed, list the contents of the Customer table to ensure that “Harry Joker” made it into the table. Capture screenshots illustrating the execution and results of both commands.

6. Realistically the ADD_CUSTOMER_HARRY stored procedure can be executed only once. Inside the procedure, we placed the literal value “8” for the customer_id column, 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. So we conclude that this stored procedure is not reusable.

A second execution of the procedure would attempt to insert the same customer, resulting in a primary key violation. This is illustrated below.

SQLQuery1.sql - ZO...arren Mansur (54))* X

```
EXECUTE ADD_CUSTOMER_HARRY;
```

100 %

Messages

Msg 2627, Level 14, State 1, Procedure ADD_CUSTOMER_HARRY, Line 4
Violation of PRIMARY KEY constraint 'PK__customer__CD64CFDD53BF33E2'. Cannot insert duplicate key in object 'dbo.customer'. The duplicate key value is (8).
The statement has been terminated.

Notice that the error message states that there was an attempt to insert the same value, 8, into the primary key of Customer.

7. Try to execute the stored procedure a second time yourself. Capture a screenshot of the command and the results of its execution and the error message.

8. 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 the ADD_CUSTOMER_HARRY stored procedure cannot 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 SQL Server 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, if instead of hardcoding the value “8” for the customer_id column, we defined a parameter named “cus_id_arg” with a datatype of “DECIMAL”, the particular value can be specified when the stored procedure is executed. Below is an ADD_CUSTOMER stored procedure that makes use of several parameters and is therefore reusable. Comments next to the parameters help explain their purpose.

```
CREATE PROCEDURE ADD_CUSTOMER    -- Create a new customer
    @cus_id_arg DECIMAL,         -- The new customer's ID.
    @first_name_arg VARCHAR(30), -- The new customer's first name.
    @last_name_arg VARCHAR(40)   -- The new customer's last name.
AS -- This "AS" is required by the syntax of stored procedures.
BEGIN
    -- Insert the new customer with a 0 balance.
    INSERT INTO CUSTOMER
    (CUSTOMER_ID, CUSTOMER_FIRST, CUSTOMER_LAST, CUSTOMER_TOTAL)
    VALUES (@cus_id_arg, @first_name_arg, @last_name_arg, 0);
END;
```

All that you need to do is type or copy and paste the text into your SQL client, then run it to create the stored procedure. Capture a screenshot illustrating the creation of this stored procedure in your SQL client.

9. Your next step is to invoke the procedure that you created to add a new customer using the execute command.

```
EXECUTE ADD_CUSTOMER 9, 'Mary', 'Smith'
```

Notice that you specify the values when you run the store procedure, in this example, “9”, “Mary”, and “Smith”. This means that we could add many more customers using this stored procedure just by changing the values given to the stored procedure.

Execute this command and capture a screenshot illustration its execution.

10. Next, we want to verify that Mary Smith was added to the Customer table with a simple select command.

```
SELECT *  
FROM CUSTOMER;
```

Execute the command. Capture a screenshot illustrating its execution and results.

11. Up to this point, the precise commands have been given to you. Now you have a chance to modify the existing code without being given the precise text. Modify the ADD_CUSTOMER procedure so that it takes a fourth CUST_BALANCE IN argument which will be used to set the Customer’s balance. Make sure to also modify the INSERT statement so that it inserts the value passed in instead of the hard-coded zero in the original definition of the stored procedure.

Capture a screenshot of the creation of your new stored procedure.

12. We will now add a new customer, making sure to take advantage of the additional balance parameter that you have provided.

```
EXEC ADD_CUSTOMER 10,'Gabriela','Jury',10.99
```

The command EXEC is the short form of EXECUTE.

Execute this command to insert Gabriela Jury into the Customer table with a balance of 10.99. Capture a screenshot illustrating the execution and results of this command.

13. Next, execute a SQL query that verifies that Gabriela Jury was added to the Customer table with the correct balance. Capture a screenshot illustrating the execution and results of the command.

14. In addition to reusability through the use of parameters, stored procedures can also be made more useful by executing multiple SQL commands that make up one logical unit of work. For example, what if we want to delete the Customer John Smith, as well as all of his Order and Line_item information? We would need to execute several delete

statements in the correct order in order to do so. We could do so manually, but better yet we could create a parameterized stored procedure that would delete any customer's information of our choosing. We would just need to specify the `customer_id` when executing the stored procedure.

Do so now. Create such a stored procedure, then execute it to delete John Smith and all Order and Line_item information associated with John Smith. Capture a screenshot of the commands to create and execute the stored procedure, as well as the results of their execution. Also, be sure to capture screenshots of the SELECT statements that list out the Customer, Order, and Line_item table after the stored procedure has been executed. This will show that your stored procedure behaves as expected.

Depending upon the logic you use to solve this step, you may need to make use of a simple subquery construct to delete the line_items. Since subqueries are not covered fully in Module 4, below is the basic form of such a delete statement that makes use of a subquery:

```
DELETE FROM Line_item
WHERE order_id IN (SELECT order_id
                   FROM Customer_order
                   WHERE customer_id = 5);
```

This command instructs the database to delete any row in the Line_item table that has an order_id placed by a Customer with customer_id 5. The part in parentheses, (SELECT order_id...), is the subquery which retrieves the order ids associated to Customer 5. You will learn more details about subqueries in lab 5, but this knowledge of subqueries is sufficient for this step. Here we hardcode 5, but you will of course make your results more dynamic to solve this step.

Steps for PostgreSQL

Complete these steps if you are using PostgreSQL. If you are using Oracle or SQL Server, complete the corresponding steps in a different section.

1. Before you begin creating stored procedures and triggers in the schema presented in this lab, you should create a logical ERD representing the schema. It is a best practice to reference an ERD while working with a database schema, so that you can quickly ascertain the overall design of the schema, including the entities, attributes, and relationships. Create a logical ERD of the database schema now, and make sure to include all attributes, primary keys, and foreign keys.

2. We now list out the code which defines a basic stored procedure. Note that while PostgreSQL technically does not support stored procedures directly, it provides the same functionality with functions that return no value.

```
CREATE OR REPLACE FUNCTION ADD_CUSTOMER_HARRY() RETURNS VOID
AS
$proc$
    BEGIN
        INSERT INTO Customer
            (customer_id,customer_first,customer_last, customer_total)
        VALUES (8, 'Harry', 'Joker', 0);
    END;
$proc$ LANGUAGE plpgsql
```

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

Line 1: CREATE OR REPLACE FUNCTION ADD_CUSTOMER_HARRY() RETURN VOID	
	<p>The CREATE OR REPLACE FUNCTION phrase indicates to PostgreSQL that a function 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 FUNCTION, then PostgreSQL would create the function 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 function. 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 stored procedure 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> <p>The RETURNS VOID words specify the return data type of the function. Unlike Oracle and SQL Server, we use functions that return no value in PostgreSQL instead of stored procedures. In this example using a function that does not return any data we specify a VOID return type.</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 FUNCTION ADD_CUSTOMER_HARRY() RETURNS VOID AS leads an English speaker to naturally think that the definition of the function follows the AS keyword.</p>
Line 3: \$proc\$	
	<p>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 FUNCTION command is concerned. Therefore we use dollar quoting, denoted by the characters “\$\$”, to write the function 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 complex function bodies that require quotations. The “proc” in between the dollar signs is a label that adds to readability and is optional.</p>
Line 4: BEGIN	
	<p>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() function.</p>
Lines 5-7: INSERT INTO CUSTOMER (customer_id,customer_first,customer_last,customer_total) VALUES(8, 'Harry', 'Joker', 0);	
	<p>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.</p>
Line 8: END;	
	<p>The END keyword tells PostgreSQL that the procedural block, and therefore the function definition, ends.</p>

Line 9:	<code>\$proc\$ LANGUAGE plpgsql</code>
	In this final line you see the \$\$ again, because the function 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.

Just as creating a table and accessing that table are two distinct actions, creating a function and executing the function are two distinct actions. When we execute this code, we are instructing PostgreSQL to create the function as defined, as illustrated in the screenshot below.

```

CS669 on cs669@CS669_Connect
1 CREATE OR REPLACE FUNCTION ADD_CUSTOMER_HARRY() RETURNS VOID
2 AS
3 $proc$
4 BEGIN
5     INSERT INTO Customer (customer_id, customer_first, customer_last, customer_total)
6     VALUES (8, 'Harry', 'Joker', 0);
7 END;
8 $proc$ LANGUAGE plpgsql

```

Data Output Explain Messages Query History

CREATE FUNCTION

Query returned successfully in 126 msec.

- Now you give it a try. Copy and paste the code of the function into your command window, then capture a screenshot illustrating its creation.
- Now that we have created the function, we need to execute it for its code to take effect. We can do so by defining an *anonymous block* in which to execute procedural code, and then invoking the function within that block. First, let’s look at the code to do so.

```

DO
$$
BEGIN
EXECUTE ADD_CUSTOMER_HARRY();
END;
$$

```

Again, let us visit the meaning of the code line by line.

Line 1: DO	
	This command in PostgreSQL is used to begin an anonymous block of code.
Line 2: \$\$	
	Se need to use the Dollar quoting to specify the literal that will be run by the DO command.
Line 3: BEGIN	
	Just as with a function, we use the BEGIN keyword as part of starting the block of procedural code. The difference is that we have not associated this block with any persistent stored module, and so we are creating an <i>anonymous</i> procedural block. Anonymous blocks are not durably saved to the database, and can be embedded into scripts to perform logic that does not need to be durably saved, commonly because it is a one-time action.
Line 4: EXECUTE ADD_CUSTOMER_HARRY();	
	Here we use the EXECUTE command along with the name of the function followed by a semicolon. In the context of a procedural block, this instructs PostgreSQL to execute the code defined in the stored procedure.
Line 5: END;	
	The END keyword tells PostgreSQL that the anonymous procedural block is ended. At this point, the anonymous procedural block has been defined, but has not yet been executed. This is similar to the way that a function is first defined and later executed. Why is it necessary to place a semicolon after the END keyword? By the nature of the language, procedural blocks are defined within the context of the SQL engine, but are actually executed by a separate procedural engine. So from the context of SQL, a block is just another command, and the block needs the semicolon as a statement separator, the same as any other SQL command.
Line 6: \$\$	
	Here again we see the terminating Dollar quoting

Now when we execute this anonymous block, we see this in pgAdmin.


```
CS669 on cs669@CS669_Connect
1 DO
2 $$
3 BEGIN
4 EXECUTE ADD_CUSTOMER_HARRY();
5 END;
6 $$
```

Data Output Explain Messages Query History

D0

Query returned successfully in 117 msec.

Note that PostgreSQL provides another option for executing a function. Since a function can be used in a SQL script, calling the function in a SELECT statement will also fire the code in the function. This particular function can be invoked with the command `SELECT ADD_CUSTOMER_HARRY()`. Both Oracle and SQL Server support invoking functions in SQL as well (with some restrictions), but not stored procedures.

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

```
CS669 on cs669@CS669_Connect
1 SELECT * FROM Customer;
```

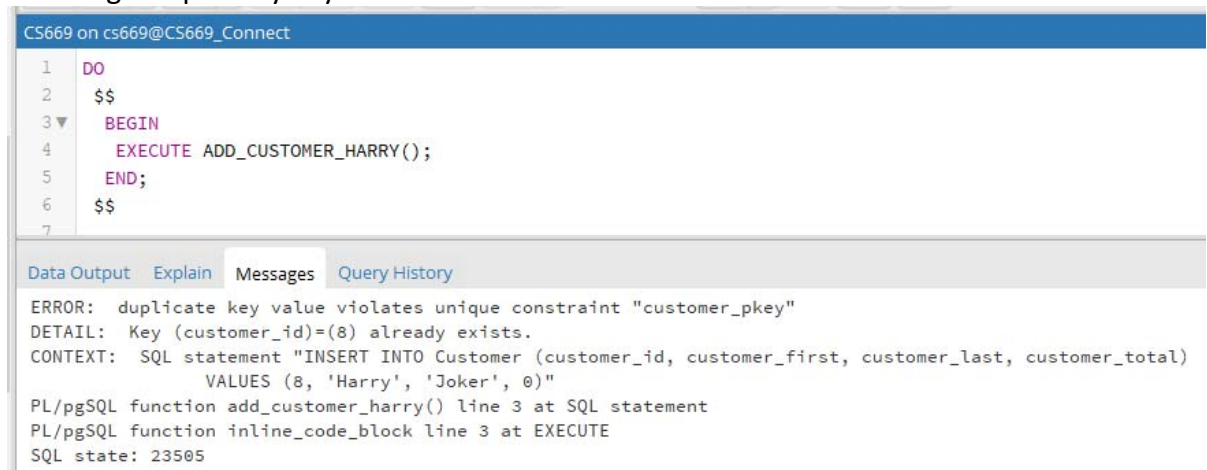
	customer_id numeric (10)	customer_first character varying (30)	customer_last character varying (40)	customer_total numeric (12,2)
1	1	John	Smith	0.00
2	2	Mary	Berman	0.00
3	3	Elizabeth	Johnson	0.00
4	4	Peter	Quigley	0.00
5	5	Stanton	Hurley	0.00
6	6	Yvette	Presley	0.00
7	7	Hilary	Marsh	0.00
8	8	Harry	Joker	0.00

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

Note that by default functions execute within the current transaction, so we would need to commit the transaction if we want to durably add it to the database. This can be accomplished by running the COMMIT command, or by having Auto Commit enabled.

5. Give it a try yourself. Execute the ADD_CUSTOMER_HARRY stored procedure, and once you are sure it executed, list the contents of the Customer table to ensure that “Harry Joker” made it into the table. Capture screenshots illustrating the execution and results of both commands.
6. Realistically the ADD_CUSTOMER_HARRY function can be executed only once. Inside the function, we placed the literal value “8” for the customer_id column, 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. So we conclude that this function is not reusable.

A second execution of the function would attempt to insert the same customer, resulting in a primary key violation. This is illustrated below.



```
CS669 on cs669@CS669_Connect
1 DO
2 $$
3 BEGIN
4 EXECUTE ADD_CUSTOMER_HARRY();
5 END;
6 $$
7

Data Output Explain Messages Query History
ERROR: duplicate key value violates unique constraint "customer_pkey"
DETAIL: Key (customer_id)=(8) already exists.
CONTEXT: SQL statement "INSERT INTO Customer (customer_id, customer_first, customer_last, customer_total)
VALUES (8, 'Harry', 'Joker', 0)"
PL/pgSQL function add_customer_harry() line 3 at SQL statement
PL/pgSQL function inline_code_block line 3 at EXECUTE
SQL state: 23505
```

7. Try to execute the function a second time yourself. Capture a screenshot of the command and the results of its execution.
8. It is best to make our functions reusable, so that they can be executed wherever the logic contained in them is needed. The fact that the ADD_CUSTOMER_HARRY() function cannot 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 PostgreSQL to use whatever value is given to the function 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, if instead of hardcoding the value “8” for the customer_id column, we defined a parameter named “cus_id_arg” with a datatype of “DECIMAL”, the particular value can be specified when the stored procedure is executed. Below is an ADD_CUSTOMER() function that makes use of several parameters and is therefore reusable. Comments next to the parameters help explain their purpose.

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

All that you need to do is type or copy and paste the text into your SQL client, then run it to create the function. Capture a screenshot illustrating the creation of this function in your SQL client.

9. Your next step is to invoke the function that you created to add a new customer. We do this with an anonymous PL/pgSQL block.

```
DO
$$
BEGIN
    EXECUTE ADD_CUSTOMER(9, 'Mary', 'Smith');
END;
$$
```

Notice that you specify the values when you run the function, in this example, “9”, “Mary”, and “Smith”. This means that we could add many more customers using this function just by changing the values given to the function parameters.

Execute this command and capture a screenshot illustrating its execution.

10. Next, we want to verify that Mary Smith was added to the Customer table with a simple select command.

```
SELECT *  
FROM CUSTOMER;
```

Execute the command. Capture a screenshot illustrating its execution and results.

11. Up to this point, the precise commands have been given to you. Now you have a chance to modify the existing code without being given the precise code. Modify the ADD_CUSTOMER function so that it takes a fourth CUST_BALANCE IN argument which will be used to set the Customer's balance. Make sure to also modify the INSERT statement so that it inserts the value passed in instead of the hard-coded zero in the original definition of the function.

Capture a screenshot of the creation of your new function.

12. We will now add a new customer, making sure to take advantage of the additional balance parameter that you have provided.

```
DO  
$$  
BEGIN  
EXECUTE ADD_CUSTOMER(10, 'Gabriela', 'Jury', 10.99);  
END;  
$$
```

Execute this command to insert Gabriela Jury into the Customer table with a balance of 10.99. Capture a screenshot illustrating the execution and results of this command.

13. Next, execute a SQL query that verifies that Gabriela Jury was added to the Customer table with the correct balance. Capture a screenshot illustrating the execution and results of the command.
14. In addition to reusability through the use of parameters, functions can also be made more useful by executing multiple SQL commands that make up one logical unit of work. For example, what if we want to delete the Customer John Smith, as well as all of his Order and Line_item information? We would need to execute several delete statements in the correct order in order to do so. We could do so manually, but better yet we could create a parameterized stored procedure that

would delete any customer's information of our choosing. We would just need to specify the `customer_id` when executing the function.

Do so now. Create such a function, then execute it to delete John Smith and all Order and Line_item information associated with John Smith. Capture a screenshot of the commands to create and execute the function, as well as the results of their execution. Also, be sure to capture screenshots of the SELECT statements that list out the Customer, Order, and Line_item table after the function has been executed. This will show that your function behaves as expected.

Depending upon the logic you use to solve this step, you may need to make use of a simple subquery construct to delete the line_items. Since subqueries are not covered fully in Module 4, below is the basic form of such a delete statement that makes use of a subquery:

```
DELETE FROM Line_item
WHERE order_id IN (SELECT order_id
                  FROM Customer_order
                  WHERE customer_id = 5);
```

This command instructs the database to delete any row in the Line_item table that has an order_id placed by a Customer with customer_id of 5. The part in parentheses, (SELECT order_id...), is the subquery which retrieves the order ids associated to Customer 5. You will learn more details about subqueries in lab 5, but this knowledge of subqueries is sufficient for this step. Here we hardcode 5, but you will of course make your results more dynamic to solve this step.

Section Two – Triggers

Overview

Triggers are another form of a persistent stored module. Just as with stored procedures, we define procedural and declarative SQL code in the body of the trigger that performs a logical unit of work. One key difference between a trigger and a stored procedure is that all triggers are associated to an *event* that determines when its code is executed. The specific event is defined as part of the overall definition of the trigger when it is created. The database then automatically invokes the trigger when the defined event occurs. We cannot directly execute a trigger.

Triggers can be powerful and useful. For example, what if we desire to keep a history of changes that occur to a particular table? We could define a trigger on one table that logs any changes to another table. What if, in an ordering system, we want to reject duplicate charges that occur from the same customer in quick succession as a safeguard? We could define a trigger to look for such an event and reject the offending transaction. These are just two examples. There are a virtually unlimited number of use cases where the use of triggers can be of benefit.

Triggers also have significant drawbacks. By default triggers execute within the same transaction as the event that caused the trigger to execute, and so any failure of the trigger results in the abortion of the overall transaction. Triggers execute additional code beyond the regular processing of the database, and as such can increase the time a transaction needs to complete, and can cause the transaction to use more database resources. Triggers operate automatically when the associated event occurs, so can cause unexpected side effects when a transaction executes, especially if the author of the transaction was not aware of the trigger's logic when authoring the transaction's code. Triggers silently perform logic, perhaps in an unexpected way.

Although triggers are powerful, because of the associated drawbacks, it is a best practice to reserve the use of triggers to situations where there is no other practical alternative. For example, perhaps we want to add functionality to a two-decade-old application's database access logic, but are unable to do so because the organization has no developer capable of updating the old application. We may then opt to use a trigger to execute on key database events, avoiding the impracticality of updating the old application. Perhaps the same database schema is updated from several different applications, and we cannot practically add the same business logic to all of them. We may then opt to use a trigger to keep the business logic consolidated into a single place that is executed automatically. Perhaps an application that accesses our database is proprietary, but we want to perform some logic when the application access the database. Again, we may opt to add a trigger to effectively add logic to an otherwise

proprietary application. There are many examples, but the key point is that triggers should be used sparingly, only when there is no other practical alternative.

Follow the steps in this section to learn how to create and use triggers.

Steps for Oracle

Complete these steps for Oracle only if you are using Oracle. If you are using SQL Server or PostgreSQL, steps for those DBMS' are available in a subsequent section.

1. We will now create a trigger that prevents the customer balance from being negative in the schema for this lab. Such a business rule could be implemented as a CHECK constraint. The advantage of doing it in a trigger is that we have control of what happens when the constraint is violated. In this case we call the library routine `RAISE_APPLICATION_ERROR`, which provides a human-readable error message, provides an ORA error code, throws an exception and rolls back the transaction. Triggers are more complex than they seem.

Below is the code for such a trigger.

```
CREATE OR REPLACE TRIGGER no_neg_cust_bal_trg
BEFORE UPDATE OR INSERT ON customer
FOR EACH ROW
BEGIN
    IF :NEW.customer_total < 0 THEN
        RAISE_APPLICATION_ERROR(-20001,'Customer balance cannot be negative.');
```

The logic for the trigger is as follows. The line:

```
IF :NEW.customer_total < 0
```

succeeded tests to determine if the `customer_total` parameter is negative. If it is, then then executes the `RAISE_APPLICATION_ERROR` function built into Oracle, which would fail the transaction with the error text supplied in the trigger, “Customer balance cannot be negative”.

The identifier “:NEW” refers to the value of the row that is being modified, after the modification, but before it is written to the database. `:NEW.customer_total` refers to the value of the balance column in that modified record. Note that the error message begins with “ORA-20001.” We provided the -20001 as the first argument to

RAISE_APPLICATION_ERROR. Execute the code to create the trigger in your database, and capture a screenshot of its creation.

2. Now test the trigger by attempting to insert a new customer with a negative balance. If your trigger was defined properly in the previous step, you received the error message “Customer balance cannot be negative.” when you attempted the insert, which was the text supplied in the trigger. You also received the Oracle error ORA-06512, which tells us that RAISE_APPLICATION_ERROR was invoked from line 3 of our trigger.

Capture a screenshot of the INSERT command along with the associated error message that occurs from the attempt.

3. Just to verify that the trigger blocked the transaction in the previous step, execute a SQL query which checks whether the customer with the negative balance was created, and capture a screenshot of the command and its results.

4. Create a trigger on the Customer table that blocks the insertion of any customers with the last name of “Glass”, and capture a screenshot of its creation.

5. Attempt to insert a customer with a last name of “Glass”, and verify that the trigger does indeed block the insert. Capture a screenshot of the INSERT command, along with the error indicating that the trigger blocked the insert.

6. Now let us do something more complex with a trigger. Imagine that we would like to store a history of price changes for each item. To do so, we would first need to create an Item_price_history table that stores at a minimum a reference to the item, its old price, its new price, and the date of the change. We could then define a trigger on the Item table that inserts a row into the Item_price_history table whenever an item price is updated.

For this step, go ahead and create the Item_price_history table as described. Paste in an updated ERD which includes that new table. Also capture a screenshot of the table creation in SQL.

Note that it is at your discretion whether you add a primary key for this history table. In this scenario, a primary key is not strictly necessary; however, 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.

7. Define the trigger as described in the previous step. Capture a screenshot of its creation, and briefly explain the logic of the trigger.

8. Last, change the price of plates to 11, and the price of spoons to 4. Capture a screenshot of a listing of the contents of the Item_price_history table, which verifies that your trigger did indeed store the price change history as expected. Briefly explain the contents of the table.

Congratulations! You have successfully worked with both stored procedures and triggers. There is a lot more we could go into for this procedural code, but what we have done thus far is enough for now.

Steps for SQL Server

Complete these steps if you are using SQL Server. If you are using Oracle or PostgreSQL, complete the corresponding steps in a different section.

1. We will now create a trigger that prevents the customer balance from being negative. Such a business rule could be implemented as a CHECK constraint. The advantage of doing it in a trigger is that we have control of what happens when the constraint is violated. In this case we call the library routine ROLLBACK to roll back the transaction, and the library routine RAISERROR, which provides a human-readable error message. Triggers are more complex than they seem.

Below is the code for such a trigger.

```
CREATE TRIGGER no_neg_cust_bal_trg
ON customer
AFTER INSERT, UPDATE
AS
BEGIN
DECLARE @CUSTOMER_TOTAL DECIMAL;
SET @CUSTOMER_TOTAL= (SELECT INSERTED.customer_total FROM INSERTED);

IF @CUSTOMER_TOTAL < 0
BEGIN
ROLLBACK
RAISERROR('Customer balance cannot be negative',14,1);
END;
END;
```

Paste the code into your database, and capture a screenshot of the creation of trigger in your database.

2. Now test that the trigger works as expected by trying to insert a new customer with an initial negative balance.

Note that you receive the error message “Customer balance cannot be negative.”

Further note the following:

- You received the error message “Customer balance cannot be negative” that you supplied as the second argument to the built-in procedure RAISERROR.
- The exception was raised when the test `IF @CUSTOMER_TOTAL < 0` succeeded, because the balance column that we were trying to insert was negative 1, which is less than zero.
- The error message begins with “Msg 5000 Level 14.” We provided level 14 as the second argument to `RAISERROR`.
- The error message indicates that it was called from line 13 of our trigger.
- You will also notice error Msg 3609 Level 16 “The transaction ended in the trigger. The batch has been aborted.” This results from our ROLLBACK command.

Capture a screenshot of the INSERT command executed in this step, as well as the error the trigger generated from the attempt.

3. Just to verify that the trigger blocked the transaction in the previous step, execute a SQL query which checks whether the customer with the negative balance was created, and capture a screenshot of the command and its results.
4. Create a trigger on the Customer table that blocks the insertion of any customers with the last name of “Glass”, and capture a screenshot of its creation.
5. Attempt to insert a customer with a last name of “Glass”, and verify that the trigger does indeed block the insert. Capture a screenshot of the INSERT command, along with the error indicating that the trigger blocked the insert.
6. Now let us do something more complex with a trigger. Imagine that we would like to store a history of price changes for each item. To do so, we would first need to create an Item_price_history table that stores at a minimum a reference to the item, its old price, its new price, and the date of the change. We could then define a trigger on the Item table that inserts a row into the Item_price_history table whenever an item price is updated.

For this step, go ahead and create the Item_price_history table as described. Paste in an updated ERD which includes that new table. Also capture a screenshot of the table creation in SQL.

Note that it is at your discretion whether you add a primary key for this history table. In this scenario, a primary key is not strictly necessary; however, 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.

7. Define the trigger as described in the previous step. Capture a screenshot of its creation, and briefly explain the logic of the trigger.
8. Last, change the price of plates to 11, and the price of spoons to 4. Capture a screenshot of a listing of the contents of the Item_price_history table, which verifies that your trigger did indeed store the price change history as expected. Briefly explain the contents of the table.

Congratulations! You have successfully worked with both stored procedures and triggers. There is a lot more we could go into for this procedural code, but what we have done thus far is enough for now.

Steps for PostgreSQL

Complete these steps if you are using PostgreSQL. If you are using Oracle or SQL Server, complete the corresponding steps in a different section.

1. We will now create a trigger that prevents the customer balance from being negative in the schema for this lab. Such a business rule could be implemented as a CHECK constraint. The advantage of doing it in a trigger is that we have control of what happens when the constraint is violated.

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 function.

```
CREATE OR REPLACE FUNCTION no_neg_cust_bal_func()  
RETURNS TRIGGER LANGUAGE plpgsql  
AS $trigfunc$  
BEGIN  
    RAISE EXCEPTION USING MESSAGE = 'Customer balance cannot be  
negative.',  
                                ERRCODE = 22000;  
END;  
$trigfunc$;
```

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 nothing by using RETURNS TRIGGER instead of RETURNS VOID (we use RETURN

VOID when creating stored procedure type functions). The remainder of the function is defined in a similar fashion as with stored procedures. The `$trigfunc$` keywords delineate the function body. 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, then it will raise this exception.

To complete our trigger, we must now define it and link it to the function, like so.

```
CREATE TRIGGER no_neg_cust_bal_trg
  BEFORE UPDATE OR INSERT ON customer
  FOR EACH ROW WHEN(new.customer_total < 0)
  EXECUTE PROCEDURE no_neg_cust_bal_func();
```

The `CREATE TRIGGER` words indicate that we are defining a trigger. The `no_neg_cust_bal_trg` word is an identifier, the name of the trigger. The “`BEFORE UPDATE OR INSERT ON customer`” words indicates that the trigger will fire just before an update or insert occurs on the table. The `FOR EACH ROW` keywords indicate that the trigger should be executed for every row that is updated or inserted (as opposed to only once per update or insert command). The “`WHEN (new.customer_total < 0)`” words give a condition for which the trigger will execute, that is, only when the `customer_total` parameter is negative. If the condition matches, the `no_neg_cust_bal_func` function that we previously defined will be executed, as indicated by “`EXECUTE PROCEDURE no_neg_cust_bal_func`”. The function will fail the transaction with the error text supplied in the function, “Customer balance cannot be negative”.

Note that the identifier “new” in “`new.customer_total`” refers to the value of the row that is being modified, after the modification, but before it is written to the database. Further note that the error message begins with the message we want displayed and then the built in Error Code 22000. PostgreSQL provides a list of built in error codes that can be used. Check the latest documentation for the most current list.

Execute the code to create the trigger in your database, and capture a screenshot of its creation.

2. Now test the trigger by attempting to insert a new customer with a negative balance. If your trigger was defined properly in the previous step, you received the error message “Customer balance cannot be negative.” when you attempted the insert, which was the text supplied in the function and used in the trigger.

Capture a screenshot of the `INSERT` command along with the associated error message that occurs from the attempt.

3. Just to verify that the trigger blocked the transaction in the previous step, execute a SQL query which checks whether the customer with the negative balance was created, and capture a screenshot of the command and its results.

4. Create a trigger on the Customer table that blocks the insertion of any customers with the last name of “Glass”, and capture a screenshot of its creation.

5. Attempt to insert a customer with a last name of “Glass”, and verify that the trigger does indeed block the insert. Capture a screenshot of the INSERT command, along with the error indicating that the trigger blocked the insert.

6. Now let us do something more complex with a trigger. Imagine that we would like to store a history of price changes for each item. To do so, we would first need to create an Item_price_history table that stores at a minimum a reference to the item, its old price, its new price, and the date of the change. We could then define a trigger on the Item table that inserts a row into the Item_price_history table whenever an item price is updated.

For this step, go ahead and create the Item_price_history table as described. Paste in an updated ERD which includes that new table. Also capture a screenshot of the table creation in SQL.

Note that it is at your discretion whether you add a primary key for this history table. In this scenario, a primary key is not strictly necessary; however, 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.

7. Define the trigger as described in the previous step. Capture a screenshot of its creation, and briefly explain the logic of the trigger.

8. Last, change the price of plates to 11, and the price of spoons to 4. Capture a screenshot of a listing of the contents of the Item_price_history table, which verifies that your trigger did indeed store the price change history as expected. Briefly explain the contents of the table.

Congratulations! You have successfully worked with both stored procedures and triggers. There is a lot more we could go into for this procedural code, but what we have done thus far is enough for now.