

Overview of the Lab

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, lists, and other extended datatypes. The procedural languages also support the ability to embed and use the results of SQL queries. Combining the procedural language with SQL is powerful and allows you to solve problems that cannot be addressed with SQL alone.

From a technical perspective, together, we will learn:

- how to create and use sequences.
- how to create reusable stored procedures.
- how to save calculated and database values into variables, and make use of the variables.
- how to implement full transactions in stored procedures.
- how to create triggers to perform intra-table and cross-table validations.
- how to store a history of a column using a table and a trigger.
- to normalize a schema's tables to BCNF.

Lab 4 Explanations Reminder

As a reminder, it is important to read through the Lab 4 Explanation document to successfully complete this lab, available in the assignment inbox alongside this lab. The explanation document illustrates how to correctly execute each SQL construct step-by-step, and explains important theoretical and practical details.

Other Reminders

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

Section One – Stored Procedures

Section Background

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.

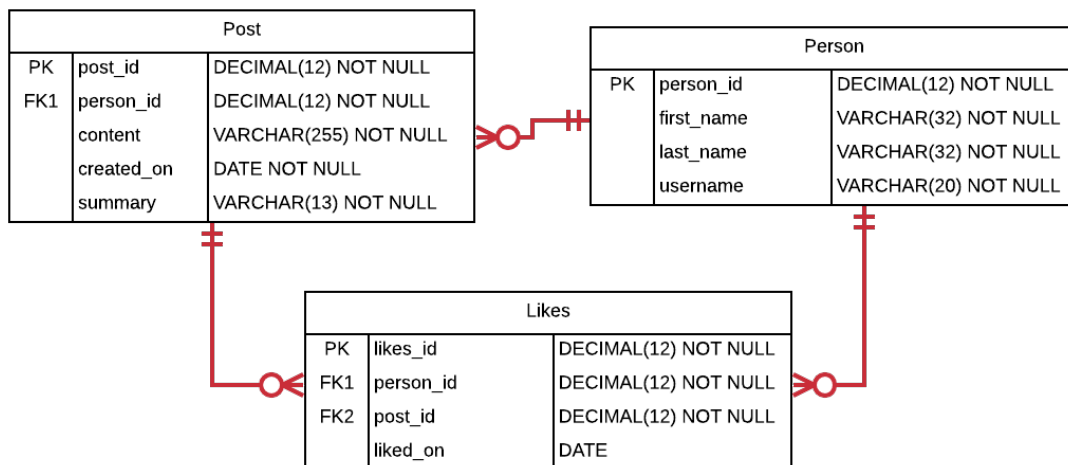
Modern relational DBMS support the creation and use of persistent stored modules, namely, stored procedures and triggers, which are widely used to perform operations critical to modern information systems. A stored procedure contains logic that is executed when a transaction invokes the name of the stored 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 many DBMS 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.

You will be working with the following schema in this section, which is a greatly simplified social networking schema. It tracks the people who join the social network, as well as their posts and the "likes" on their posts.



The Person table contains a primary key, the person's first and last name, and the person's username that they use to login to the social networking website. The Post table contains a primary key, a foreign key to the Person that made the post, a

shortened content field containing the text of the post, a created_on date, and a summary of the content which is the first 10 characters (including spaces) followed by "...". For example, if the content is "Check out my new pictures.", then the summary would be "Check out ...". The Likes table contains a primary key, a foreign key to the Person that likes the Post, a foreign key to the Post, and a date on which the Post was liked.

In this first section, you will work with stored procedures on this schema, which offer many significant benefits. Reusability is one significant benefit. 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 benefit 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.

You will also learn to work with sequences in this section, which are the preferred means of generating synthetic primary keys for each of your tables.

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

Section Steps

- 1. *Create Table Structure*** – Create the tables in the social networking schema, including all of their columns, datatypes, and constraints. Create sequences for each table; these will be used to generate the primary and foreign key values in Step #2.

Query	Query History	Data output	Messages	Notifications
<pre> 1 CREATE TABLE Person (2 person_id DECIMAL(12) NOT NULL PRIMARY KEY, 3 first_name VARCHAR(32) NOT NULL, 4 last_name VARCHAR(32) NOT NULL, 5 username VARCHAR(20) NOT NULL); 6 7 CREATE TABLE Post (8 post_id DECIMAL(12) NOT NULL PRIMARY KEY, 9 person_id DECIMAL(12) NOT NULL, 10 content VARCHAR(255) NOT NULL, 11 created_on DATE NOT NULL, 12 summary VARCHAR(13) NOT NULL, 13 FOREIGN KEY (person_id) REFERENCES Person(person_id)); 14 15 CREATE TABLE Likes (16 likes_id DECIMAL(12) NOT NULL PRIMARY KEY, 17 person_id DECIMAL(12) NOT NULL, 18 post_id DECIMAL(12) NOT NULL, 19 liked_on DATE, 20 FOREIGN KEY (person_id) REFERENCES Person(person_id), 21 FOREIGN KEY (post_id) REFERENCES Post(post_id)); 22 23 CREATE SEQUENCE person_seq START WITH 1; 24 CREATE SEQUENCE post_seq START WITH 1; 25 CREATE SEQUENCE likes_seq START WITH 1; </pre>		CREATE TABLE	Query returned successfully in 52 msec.	

Query	Query History	Data output	Messages	Notifications
<pre> 1 CREATE TABLE Person (2 person_id DECIMAL(12) NOT NULL PRIMARY KEY, 3 first_name VARCHAR(32) NOT NULL, 4 last_name VARCHAR(32) NOT NULL, 5 username VARCHAR(20) NOT NULL); 6 7 CREATE TABLE Post (8 post_id DECIMAL(12) NOT NULL PRIMARY KEY, 9 person_id DECIMAL(12) NOT NULL, 10 content VARCHAR(255) NOT NULL, 11 created_on DATE NOT NULL, 12 summary VARCHAR(13) NOT NULL, 13 FOREIGN KEY (person_id) REFERENCES Person(person_id)); 14 15 CREATE TABLE Likes (16 likes_id DECIMAL(12) NOT NULL PRIMARY KEY, 17 person_id DECIMAL(12) NOT NULL, 18 post_id DECIMAL(12) NOT NULL, 19 liked_on DATE, 20 FOREIGN KEY (person_id) REFERENCES Person(person_id), 21 FOREIGN KEY (post_id) REFERENCES Post(post_id)); 22 23 CREATE SEQUENCE person_seq START WITH 1; 24 CREATE SEQUENCE post_seq START WITH 1; 25 CREATE SEQUENCE likes_seq START WITH 1; </pre>		CREATE SEQUENCE	Query returned successfully in 65 msec.	

Query Query History

```
1 CREATE TABLE Person (  
2     person_id DECIMAL(12) NOT NULL PRIMARY KEY,  
3     first_name VARCHAR(32) NOT NULL,  
4     last_name VARCHAR(32) NOT NULL,  
5     username VARCHAR(20) NOT NULL);  
6  
7 CREATE TABLE Post (  
8     post_id DECIMAL(12) NOT NULL PRIMARY KEY,  
9     person_id DECIMAL(12) NOT NULL,  
10    content VARCHAR(255) NOT NULL,  
11    created_on DATE NOT NULL,  
12    summary VARCHAR(13) NOT NULL,  
13    FOREIGN KEY (person_id) REFERENCES Person(person_id));  
14  
15 CREATE TABLE Likes (  
16     likes_id DECIMAL(12) NOT NULL PRIMARY KEY,  
17     person_id DECIMAL(12) NOT NULL,  
18     post_id DECIMAL(12) NOT NULL,  
19     liked_on DATE,  
20     FOREIGN KEY (person_id) REFERENCES Person(person_id),  
21     FOREIGN KEY (post_id) REFERENCES Post(post_id));  
22  
23 CREATE SEQUENCE person_seq START WITH 1;  
24 CREATE SEQUENCE post_seq START WITH 1;  
25 CREATE SEQUENCE likes_seq START WITH 1;  
26  
27 SELECT * FROM Person;  
28 SELECT * FROM Post;  
29 SELECT * FROM Likes;
```

Data output

Messages

Notifications

post_id

[PK] numeric (12)

person_id

numeric (12)

content

character varying (255)

created_on








date

summary

character varying (13)

Query Query History

```
1 CREATE TABLE Person (  
2     person_id DECIMAL(12) NOT NULL PRIMARY KEY,  
3     first_name VARCHAR(32) NOT NULL,  
4     last_name VARCHAR(32) NOT NULL,  
5     username VARCHAR(20) NOT NULL);  
6  
7 CREATE TABLE Post (  
8     post_id DECIMAL(12) NOT NULL PRIMARY KEY,  
9     person_id DECIMAL(12) NOT NULL,  
10    content VARCHAR(255) NOT NULL,  
11    created_on DATE NOT NULL,  
12    summary VARCHAR(13) NOT NULL,  
13    FOREIGN KEY (person_id) REFERENCES Person(person_id));  
14  
15 CREATE TABLE Likes (  
16     likes_id DECIMAL(12) NOT NULL PRIMARY KEY,  
17     person_id DECIMAL(12) NOT NULL,  
18     post_id DECIMAL(12) NOT NULL,  
19     liked_on DATE,  
20     FOREIGN KEY (person_id) REFERENCES Person(person_id),  
21     FOREIGN KEY (post_id) REFERENCES Post(post_id));  
22  
23 CREATE SEQUENCE person_seq START WITH 1;  
24 CREATE SEQUENCE post_seq START WITH 1;  
25 CREATE SEQUENCE likes_seq START WITH 1;  
26  
27 SELECT * FROM Person;  
28 SELECT * FROM Post;  
29 SELECT * FROM Likes;
```

Data output		Messages		Notifications		
						
likes_id	person_id	post_id	liked_on			
[PK]						
numeric (12)	numeric (12)	numeric (12)	date			

Query
Query History

```

1 CREATE TABLE Person (
2     person_id DECIMAL(12) NOT NULL PRIMARY KEY,
3     first_name VARCHAR(32) NOT NULL,
4     last_name VARCHAR(32) NOT NULL,
5     username VARCHAR(20) NOT NULL);
6
7 CREATE TABLE Post (
8     post_id DECIMAL(12) NOT NULL PRIMARY KEY,
9     person_id DECIMAL(12) NOT NULL,
10    content VARCHAR(255) NOT NULL,
11    created_on DATE NOT NULL,
12    summary VARCHAR(13) NOT NULL,
13    FOREIGN KEY (person_id) REFERENCES Person(person_id));
14
15 CREATE TABLE Likes (
16     likes_id DECIMAL(12) NOT NULL PRIMARY KEY,
17     person_id DECIMAL(12) NOT NULL,
18     post_id DECIMAL(12) NOT NULL,
19     liked_on DATE,
20     FOREIGN KEY (person_id) REFERENCES Person(person_id),
21     FOREIGN KEY (post_id) REFERENCES Post(post_id));
22
23 CREATE SEQUENCE person_seq START WITH 1;
24 CREATE SEQUENCE post_seq START WITH 1;
25 CREATE SEQUENCE likes_seq START WITH 1;
26
27 SELECT * FROM Person;
28 SELECT * FROM Post;
29 SELECT * FROM Likes;

```

Data output Messages Notifications

person_id	first_name	last_name	username
[PK] numeric (12)	character varying (32)	character varying (32)	character varying (20)

2. **Populate Tables** – Populate the tables with data, ensuring that there are at least 5 people, at least 8 posts, and at least 4 likes. Make sure to use sequences to generate the primary and foreign key values. Most of the fields are self-explanatory. As far as the “content” field in Post, make them whatever you like, such as “Take a look at these new pics” or “Just arrived in the Bahamas”, and set the summary as the first 10 characters of the content, followed by “...”.

Query
Query History

```

1 INSERT INTO person
2 VALUES
3     (nextval('person_seq'), 'Mike', 'Judes', 'mikej'),
4     (nextval('person_seq'), 'Ryan', 'Bryant', 'R.Bryant'),
5     (nextval('person_seq'), 'Lindsay', 'Gambin', 'Linsday_G'),
6     (nextval('person_seq'), 'Sophia', 'Ng', 'Sophia_Ng'),
7     (nextval('person_seq'), 'Marinda', 'Dean', 'Marinda.D');
8

```

Data output Messages Notifications

INSERT 0 5

Query returned successfully in 3 min 58 secs.

```

1 INSERT INTO person
2 VALUES
3     (nextval('person_seq'), 'Mike', 'Judes','mikej'),
4     (nextval('person_seq'), 'Ryan', 'Bryant','R.Bryant'),
5     (nextval('person_seq'), 'Lindsay', 'Gambin','Linsday_G'),
6     (nextval('person_seq'), 'Sophia', 'Ng','Sophia_Ng'),
7     (nextval('person_seq'), 'Marinda', 'Dean','Marinda.D');
8
9 SELECT * FROM Person;

```

Data output Messages Notifications				
	person_id [PK] numeric (12)	first_name character varying (32)	last_name character varying (32)	username character varying (20)
1	1	Mike	Judes	mikej
2	2	Ryan	Bryant	R.Bryant
3	3	Lindsay	Gambin	Linsday_G
4	4	Sophia	Ng	Sophia_Ng
5	5	Marinda	Dean	Marinda.D

ry Query History

```

INSERT INTO Post
VALUES
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Mike'),
    'The weather is nice today', CAST('20-SEP-2022' AS DATE), 'The weathe...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Lindsay'),
    'Check in Miami Beach', CAST('21-SEP-2022' AS DATE), 'Check in M...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Marinda'),
    'Amazing Sunset', CAST('18-SEP-2022' AS DATE), 'Amazing Su...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Sophia'),
    'I am coming home', CAST('19-SEP-2022' AS DATE), 'I am comin...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Ryan'),
    'Just bought new iphone', CAST('12-SEP-2022' AS DATE), 'Just bough...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Mike'),
    'I am so angry', CAST('25-SEP-2022' AS DATE), 'I am so an...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Sophia'),
    'I love swimming', CAST('23-SEP-2022' AS DATE), 'I love swi...'),
    (nextval('post_seq'),(SELECT person_id FROM Person WHERE first_name = 'Lindsay'),
    'Going to cruise ship', CAST('26-SEP-2022' AS DATE), 'Going to c...');

```

Data output Messages Notifications

INSERT 0 8

Query returned successfully in 5 secs 274 msec.

Query Query History

```
1 SELECT * FROM Post;
```

Data output Messages Notifications

	post_id [PK] numeric (12)	person_id numeric (12)	content character varying (255)	created_on date	summary character varying (13)
1	1	1	The weather is nice tod...	2022-09-20	The weathe...
2	2	3	Check in Miami Beach	2022-09-21	Check in M...
3	3	5	Amazing Sunset	2022-09-18	Amazing Su...
4	4	4	I am coming home	2022-09-19	I am comin...
5	5	2	Just bought new iphone	2022-09-12	Just bough...
6	6	1	I am so angry	2022-09-25	I am so an...
7	7	4	I love swimming	2022-09-23	I love swi...
8	8	3	Going to cruise ship	2022-09-26	Going to c...

Query Query History

```
36 INSERT INTO Likes
37 VALUES
38     (nextval('likes_seq'), (SELECT person_id FROM Person WHERE first_name = 'Mike'),
39     (SELECT post_id FROM Post WHERE summary = 'Check in M...'), CAST('22-SEP-2022' AS DATE)),
40
41     (nextval('likes_seq'), (SELECT person_id FROM Person WHERE first_name = 'Lindsay'),
42     (SELECT post_id FROM Post WHERE summary = 'I am comin...'), CAST('21-SEP-2022' AS DATE)),
43
44     (nextval('likes_seq'), (SELECT person_id FROM Person WHERE first_name = 'Sophia'),
45     (SELECT post_id FROM Post WHERE summary = 'Just bough...'), CAST('18-SEP-2022' AS DATE)),
46
47     (nextval('likes_seq'), (SELECT person_id FROM Person WHERE first_name = 'Ryan'),
48     (SELECT post_id FROM Post WHERE summary = 'Going to c...'), CAST('29-SEP-2022' AS DATE));
49
50
51
52
53
```

Data output Messages Notifications

INSERT 0 4

Query returned successfully in 59 msec.

Query

Query History

1

SELECT * FROM Likes;

Data output

Messages

Notifications

≡+

📄

▼





📋

🗑️

🗄️

⬇️

📈

	likes_id [PK] numeric (12) 	person_id numeric (12) 	post_id numeric (12) 	liked_on date 
1	1	1	2	2022-09-22
2	2	3	4	2022-09-21
3	3	4	5	2022-09-18
4	4	2	8	2022-09-29

3. **Create Hardcoded Procedure** – Create a stored procedure named “add_michelle_stella” which has no parameters and adds a person named “Michelle Stella” to the Person table. Execute the stored procedure, and list out the rows in the Person table to show that Michelle Stella has been added.

Query	Query History
1	CREATE OR REPLACE PROCEDURE add_michelle_stella()
2	AS
3	\$proc\$
4	BEGIN
5	INSERT INTO Person(person_id,first_name,last_name,username)
6	VALUES (nextval('person_seq'), 'Michelle', 'Stella', 'michelle.S');
7	END;
8	\$proc\$ LANGUAGE plpgsql;
9	
10	CALL add_michelle_stella();

Data output	Messages	Notifications
CREATE PROCEDURE		
Query returned successfully in 48 msec.		

Query Query History

```
1 CREATE OR REPLACE PROCEDURE add_michelle_stella()
2 AS
3 $proc$
4 BEGIN
5     INSERT INTO Person(person_id,first_name,last_name,username)
6     VALUES (nextval('person_seq'), 'Michelle', 'Stella', 'michelle.S');
7 END;
8 $proc$ LANGUAGE plpgsql;
9
10 CALL add_michelle_stella();
```

Data output Messages Notifications

CALL

Query returned successfully in 65 msec.

Query Query History

```
1 SELECT * FROM Person;
2
3
4
5
```

Data output Messages Notifications



	person_id [PK] numeric (12)	first_name character varying (32)	last_name character varying (32)	username character varying (20)
1	1	Mike	Judes	mikej
2	2	Ryan	Bryant	R.Bryant
3	3	Lindsay	Gambin	Linsday_G
4	4	Sophia	Ng	Sophia_Ng
5	5	Marinda	Dean	Marinda.D
6	6	Michelle	Stella	michelle.S

4. Create Reusable Procedure – Create a reusable stored procedure named “add_person” that uses parameters and allows you to insert any new person into the Person table. Execute the stored procedure with a person of your choosing, then list out the Person table to show that the person was added to the table.

Query Query History

```
1 CREATE OR REPLACE PROCEDURE add_person(  
2     first_name_arg VARCHAR(32),  
3     last_name_arg  VARCHAR(32),  
4     username_arg   VARCHAR(20))  
5     LANGUAGE plpgsql  
6 AS  
7 $resuableproc$  
8 BEGIN  
9     INSERT INTO Person(person_id, first_name, last_name, username)  
10    VALUES (nextval('person_seq'), first_name_arg, last_name_arg, username_arg);  
11 END;  
12 $resuableproc$;  
13  
14 CALL add_person('Logan', 'Harnish', 'Logan.H100');
```

CREATE PROCEDURE

Query returned successfully in 35 msec.

Query Query History

```
1 CREATE OR REPLACE PROCEDURE add_person(  
2     first_name_arg VARCHAR(32),  
3     last_name_arg  VARCHAR(32),  
4     username_arg   VARCHAR(20))  
5     LANGUAGE plpgsql  
6 AS  
7 $resuableproc$  
8 BEGIN  
9     INSERT INTO Person(person_id, first_name, last_name, username)  
10    VALUES (nextval('person_seq'), first_name_arg, last_name_arg, username_arg);  
11 END;  
12 $resuableproc$;  
13  
14 CALL add_person('Logan', 'Harnish', 'Logan.H100');
```

CALL

Query returned successfully in 38 msec.

15	
16	<code>SELECT * FROM Person;</code>

	person_id [PK] numeric (12)	first_name character varying (32)	last_name character varying (32)	username character varying (20)
1	1	Mike	Judes	mikej
2	2	Ryan	Bryant	R.Bryant
3	3	Lindsay	Gambin	Linsday_G
4	4	Sophia	Ng	Sophia_Ng
5	5	Marinda	Dean	Marinda.D
6	6	Michelle	Stella	michelle.S
7	7	Logan	Harnish	Logan.H100

5. Create Deriving Procedure – Create a reusable stored procedure named “add_post” that uses parameters and allows you to insert any new post into the Post table. Instead of passing in the summary as a parameter, derive the summary from the content, storing the derivation temporarily in a variable (which is then used as part of the insert statement). Recall that the summary field stores the first 10 characters of the content followed by “...”. Execute the stored procedure to add a post of your choosing, then list out the Post table to show that the addition succeeded.

Query	Query History
1	<code>CREATE OR REPLACE PROCEDURE add_post(</code>
2	<code> p_person_id DECIMAL(12),</code>
3	<code> p_content VARCHAR(255),</code>
4	<code> p_created_on DATE)</code>
5	<code> LANGUAGE plpgsql</code>
6	<code>AS</code>
7	<code>\$\$</code>
8	<code>DECLARE</code>
9	<code> v_summary VARCHAR(13);</code>
10	<code>BEGIN</code>
11	<code> v_summary := SUBSTRING(p_content FROM 1 FOR 10) '...';</code>
12	<code> INSERT INTO Post (post_id, person_id, content, created_on, summary)</code>
13	<code> VALUES (nextval('post_seq'), p_person_id, p_content, p_created_on, v_summary);</code>
14	<code>END;</code>
15	<code>\$\$;</code>
16	
17	

Data output	Messages	Notifications
CREATE PROCEDURE		
Query returned successfully in 83 msec.		

```
17 CALL add_post(3, 'this mountain is so high', CAST('10-SEP-2022' AS DATE));
18
```

Data output Messages Notifications

CALL

Query returned successfully in 31 secs 766 msec.

Query Query History

```
1 SELECT * FROM Post;
2
```

Data output Messages Notifications

	post_id [PK] numeric (12)	person_id numeric (12)	content character varying (255)	created_on date	summary character varying (13)
1	1	1	The weather is nice tod...	2022-09-20	The weathe...
2	2	3	Check in Miami Beach	2022-09-21	Check in M...
3	3	5	Amazing Sunset	2022-09-18	Amazing Su...
4	4	4	I am coming home	2022-09-19	I am comin...
5	5	2	Just bought new iphone	2022-09-12	Just bough...
6	6	1	I am so angry	2022-09-25	I am so an...
7	7	4	I love swimming	2022-09-23	I love swi...
8	8	3	Going to cruise ship	2022-09-26	Going to c...
9	9	3	this mountain is so high	2022-09-10	this mount...

6. **Create Lookup Procedure** – Create a reusable stored procedure named “add_like” that uses parameters and allows you to insert any new “like”. Rather than passing in the person_id value as a parameter to identify which person is liking which post, pass in the username of the person. The stored procedure should then lookup the person_id and store it in a variable to be used in the insert statement. Execute the procedure to add a “like” of your choosing, then list out the Like table to show the addition succeeded.

Query Query History

```
1 CREATE OR REPLACE PROCEDURE add_like(  
2     p_username VARCHAR(20),  
3     p_post_id DECIMAL(12),  
4     p_liked_on DATE)  
5     LANGUAGE plpgsql  
6 AS  
7 $$  
8 DECLARE  
9     v_person_id DECIMAL(12);  
10 BEGIN  
11     SELECT person_id  
12     INTO v_person_id  
13     FROM Person  
14     WHERE username = p_username;  
15  
16     INSERT INTO Likes (likes_id, person_id, post_id, liked_on)  
17     VALUES (nextval('likes_seq'), v_person_id, p_post_id, p_liked_on);  
18 END;  
19 $$;
```

```
21 CALL ADD_LIKE ('Logan.H100', 7, CAST('29-SEP-2022' AS DATE));  
22  
23  
24
```

Data output Messages Notifications

CALL

Query returned successfully in 105 msec.

```
23 SELECT * FROM Likes;  
24
```

Data output Messages Notifications

	likes_id [PK] numeric (12)	person_id numeric (12)	post_id numeric (12)	liked_on date
1	1	1	2	2022-09-22
2	2	3	4	2022-09-21
3	3	4	5	2022-09-18
4	4	2	8	2022-09-29
5	5	7	7	2022-09-29

Section Two – Triggers

Section Background

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 accesses 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.

Section Steps

7. Single Table Validation Trigger – One practical use of a trigger is validation within a single table (that is, the validation can be performed by using columns in the table being modified). Create a trigger that validates that the summary is being inserted correctly, that is, that the summary is actually the first 10 characters of the content followed by "...". The trigger should reject an insert that does not have a valid summary value. Verify the trigger works by issuing two insert commands – one with a correct summary, and one with an incorrect summary. List out the Post table after the inserts to show one insert was blocked and the other succeeded.

a. *Create Trigger.*

Query	Query History
<pre>1 CREATE OR REPLACE FUNCTION validate_summary_func() 2 RETURNS TRIGGER LANGUAGE plpgsql 3 AS 4 \$trigfunc\$ 5 BEGIN 6 RAISE EXCEPTION USING MESSAGE = 'The summary is being inserted incorrectly.', 7 ERRCODE = 22000; 8 END; 9 \$trigfunc\$; 10 11 CREATE TRIGGER validate_summary_trg 12 BEFORE UPDATE OR INSERT ON Post 13 FOR EACH ROW WHEN(NEW.summary != SUBSTRING(New.content FROM 1 FOR 10) '...') 14 EXECUTE PROCEDURE validate_summary_func(); 15 16</pre>	
Data output	Messages
<pre>CREATE TRIGGER Query returned successfully in 134 msec.</pre>	

b. *Insert correct summary format.*

```

15
16 INSERT INTO Post(post_id, person_id, content, created_on, summary)
17 VALUES (nextval('post_seq'), (SELECT person_id FROM Person WHERE first_name = 'Ryan'),
18        'It is raning outside',
19        CAST('10-SEP-2022' AS DATE),
20        SUBSTRING('It is raning outside' FROM 1 FOR 10) || '...');
21
22 INSERT INTO Post(post_id, person_id, content, created_on, summary)
23 VALUES (nextval('post_seq'), (SELECT person_id FROM Person WHERE first_name = 'Sophia'),
24        'I am feeling so hot now',
25        CAST('11-SEP-2020' AS DATE),
26        SUBSTRING('I am feeling so hot now' FROM 1 FOR 5) || '...');

```

Data output Messages Notifications

INSERT 0 1

Query returned successfully in 93 msec.

c. Insert incorrect summary format.

```

21
22 INSERT INTO Post(post_id, person_id, content, created_on, summary)
23 VALUES (nextval('post_seq'), (SELECT person_id FROM Person WHERE first_name = 'Sophia'),
24        'I am feeling so hot now',
25        CAST('11-SEP-2020' AS DATE),
26        SUBSTRING('I am feeling so hot now' FROM 1 FOR 5) || '...');
27
28 SELECT * FROM Post;

```

Data output Messages Notifications

ERROR: The summary is being inserted incorrectly.
CONTEXT: PL/pgSQL function validate_summary_func() line 3 at RAISE
SQL state: 22000

d. View Post table.

28 SELECT * FROM Post;

Data output Messages Notifications

	post_id [PK] numeric (12)	person_id numeric (12)	content character varying (255)	created_on date	summary character varying (13)
1	1	1	The weather is nice tod...	2022-09-20	The weathe...
2	2	3	Check in Miami Beach	2022-09-21	Check in M...
3	3	5	Amazing Sunset	2022-09-18	Amazing Su...
4	4	4	I am coming home	2022-09-19	I am comin...
5	5	2	Just bought new iphone	2022-09-12	Just bough...
6	6	1	I am so angry	2022-09-25	I am so an...
7	7	4	I love swimming	2022-09-23	I love swi...
8	8	3	Going to cruise ship	2022-09-26	Going to c...
9	9	3	this mountain is so high	2022-09-10	this mount...
10	11	2	It is raning outside	2022-09-10	It is rani...

8. Cross-Table Validation Trigger – Another practical use of a trigger is cross-table validation (that is, the validation needs columns from at least one table external to the table being updated). Create a trigger that blocks a “like” from being inserted if its “liked_on” date is before the post’s “created_on” date. Verify the trigger works by inserting two “likes” – one that passes this validation, and one that does not. List out the Likes table after the inserts to show one insert was blocked and the other succeeded.

a. Create Trigger.

Query	Query History
<pre> 1 CREATE OR REPLACE FUNCTION liked_on_func() 2 RETURNS TRIGGER LANGUAGE plpgsql 3 AS 4 \$\$ 5 DECLARE 6 v_valid_liked_on DATE; 7 BEGIN 8 SELECT post.created_on 9 INTO v_valid_liked_on 10 FROM Post 11 WHERE Post.post_id = New.post_id; 12 13 IF New.liked_on < v_valid_liked_on THEN 14 RAISE EXCEPTION USING MESSAGE = 'The liked_on date should be after the post created_on date', 15 ERRCODE = 22000; 16 END IF; 17 RETURN NEW; 18 END; 19 \$\$; 20 21 CREATE TRIGGER liked_on_trg 22 BEFORE UPDATE OR INSERT ON Likes 23 FOR EACH ROW 24 EXECUTE PROCEDURE liked_on_func(); </pre>	
Data output	Messages
<p>CREATE TRIGGER</p> <p>Query returned successfully in 75 msec.</p>	

b. Insert like – passed validation.

<pre> 26 INSERT INTO likes (likes_id, person_id, post_id, liked_on) 27 VALUES (nextval('likes_seq'), 5, 4, CAST('21-SEP-2022' AS DATE)); 28 29 INSERT INTO likes (likes_id, person_id, post_id, liked_on) 30 VALUES (nextval('likes_seq'), 1, 3, CAST('10-SEP-2022' AS DATE)); 31 </pre>		
Data output	Messages	Notifications
<p>INSERT 0 1</p> <p>Query returned successfully in 104 msec.</p>		

c. Insert like – not passed validation.

```

28
29 INSERT INTO likes (likes_id, person_id, post_id, liked_on)
30 VALUES (nextval('likes_seq'), 1, 3, CAST('10-SEP-2022' AS DATE));
31
32
33 SELECT * FROM post;

```

Data output Messages Notifications

ERROR: The liked_on date should be after the post created_on date
 CONTEXT: PL/pgSQL function liked_on_func() line 11 at RAISE
 SQL state: 22000

d. View Likes table data.

```

26 INSERT INTO likes (likes_id, person_id, post_id, liked_on)
27 VALUES (nextval('likes_seq'), 5, 4, CAST('21-SEP-2022' AS DATE));
28
29 INSERT INTO likes (likes_id, person_id, post_id, liked_on)
30 VALUES (nextval('likes_seq'), 1, 3, CAST('10-SEP-2022' AS DATE));
31
32 SELECT * FROM Likes;
33

```

Data output Messages Notifications

	likes_id [PK] numeric (12)	person_id numeric (12)	post_id numeric (12)	liked_on date
1	1	1	2	2022-09-22
2	2	3	4	2022-09-21
3	3	4	5	2022-09-18
4	4	2	8	2022-09-29
5	5	7	7	2022-09-29
6	7	5	4	2022-09-21

9. History Trigger – Another practical use of trigger is to maintain a history of values as they change. Create a table named `post_content_history` that is used to record updates to the content of a post, then create a trigger that keeps this table up-to-date when updates happen to post contents. Verify the trigger works by updating a post's content, then listing out the `post_content_history` table (which should have a record of the update).

a. Create `post_content_history` table.

```

1 CREATE TABLE post_content_history(
2     post_id DECIMAL(12) NOT NULL,
3     old_content VARCHAR(255) NOT NULL,
4     new_content VARCHAR(255) NOT NULL,
5     old_summary VARCHAR(13) NOT NULL,
6     new_summary VARCHAR(13) NOT NULL,
7     change_date DATE NOT NULL,
8     FOREIGN KEY (post_id) REFERENCES Post(post_id));

```

Data output Messages Notifications

CREATE TABLE

Query returned successfully in 129 msec.

```

10 SELECT * FROM post_content_history;

```

Data output Messages Notifications



post_id	old_content	new_content	old_summary	new_summary	change_date
numeric (12)	character varying (255)	character varying (255)	character varying (13)	character varying (13)	date

b. Create Trigger.

```

12 CREATE OR REPLACE FUNCTION post_content_history_func()
13     RETURNS TRIGGER LANGUAGE plpgsql
14 AS
15 $$
16 BEGIN
17     IF OLD.content <> NEW.content THEN
18         INSERT INTO post_content_history (post_id, old_content, new_content, old_summary,
19             new_summary, change_date)
20         VALUES (NEW.post_id, OLD.content, NEW.content,
21             OLD.summary, SUBSTRING(NEW.content FROM 1 FOR 10) || '...', CURRENT_DATE);
22     END IF;
23     RETURN NEW;
24 END;
25 $$;
26
27 CREATE TRIGGER post_content_history_trg
28 BEFORE UPDATE ON Post
29 FOR EACH ROW
30 EXECUTE PROCEDURE post_content_history_func();

```

Data output Messages Notifications

CREATE TRIGGER

Query returned successfully in 113 msec.

c. Update post.

```

33 UPDATE Post
34 SET content = 'I just completed 10 km run.'
35 WHERE post_id = 2;
36
37

```

Data output Messages Notifications

UPDATE 1

Query returned successfully in 69 msec.

d. View post_content_history table.

```

37 SELECT * FROM post_content_history;
38

```

Data output Messages Notifications

	post_id	old_content	new_content	old_summary	new_summary	change_date
	numeric (12)	character varying (255)	character varying (255)	character varying (13)	character varying (13)	date
1	2	Check in Miami Beach	I just completed 10 km...	Check in M...	I just com...	2022-09-30

e. *View Post table after updating.*

```
37 SELECT * FROM Post;
```

38

Data output

Messages

Notifications

	post_id [PK] numeric (12)	person_id numeric (12)	content character varying (255)	created_on date	summary character varying (13)	
1	1	1	The weather is nice tod...	2022-09-20	The weathe...	
2	3	5	Amazing Sunset	2022-09-18	Amazing Su...	
3	4	4	I am coming home	2022-09-19	I am comin...	
4	5	2	Just bought new iphone	2022-09-12	Just bough...	
5	6	1	I am so angry	2022-09-25	I am so an...	
6	7	4	I love swimming	2022-09-23	I love swi...	
7	8	3	Going to cruise ship	2022-09-26	Going to c...	
8	9	3	this mountain is so high	2022-09-10	this mount...	
9	11	2	It is raning outside	2022-09-10	It is rani...	
10	2	3	I just completed 10 km...	2022-09-21	Check in M...	

Section Three – Normalization

Section Background

Normalization is the standard method of reducing data redundancy in a table. When applied to every table in a database schema, redundancy, and the accompanying problems, can be significantly minimized. In this section, you have a chance to apply normalization to a scenario.

Section Steps

10. Creating Normalized Table Structure – For this question, you create a set of normalized tables based upon the scenario given, and also identify some functional dependencies between the given fields.

This scenario involves a court which handles cases between a plaintiff and defendant. Here are some rules the govern how the court operates.

- The court has a list of cases it's working with at any one time.
- Each case has one plaintiff and one defendant.
- Each case has one or more court appearances, where the plaintiff, defendant, and their attorneys attend and decisions are made about the case.
- There can be only one court appearance per day for the same case. There may be multiple appearances on the same day, but only for different cases.
- Each plaintiff and defendant may retain multiple attorneys for each court appearance.
- Multiple decisions about the case may be made at each court appearance.
- Every decision at a court appearance is assigned a number, such as decision1, decision2, and so on. This way the decision can be formally referred to by its number for an appearance.
- In a similar fashion, every attorney attending a court appearance is assigned a number, such as attorney1, attorney2, and so on.

Currently, after a court appearance is held, the court saves information a spreadsheet with each the following fields.

Field	Description
case_number	This is a unique number assigned to each case. Court staff refer to a case by this number.
case_description	This is an explanation of what the case is about.
plaintiff_first_name	This is the first name of the plaintiff in the case.
plaintiff_last_name	This is the last name of the plaintiff in the case.
defendant_first_name	This is the first name of the defendant in the case.
defendant_last_name	This is the last name of the defendant in the case.
attorney1_first_name	This is the first name of an attorney that represents the plaintiff or defendant at the court appearance.
attorney1_last_name	This is the last name of an attorney that represents the plaintiff or defendant at the court appearance.
attorney2_first_name	This is the first name of an attorney that represents the plaintiff or defendant at the court appearance.
attorney2_last_name	This is the last name of an attorney that represents the plaintiff or defendant at the court appearance.
attorney3_first_name	This is the first name of an attorney that represents the plaintiff or defendant at the court appearance.
attorney3_last_name	This is the last name of an attorney that represents the plaintiff or defendant at the court appearance.
appearance_date	This is the date a court appearance was held.
number_attending	This is the number of people attending the court appearance.

decision1_description	This is the first decision made at the court appearance, if any.
decision2_description	This is the second decision made at the court appearance, if any.
extra_appearance_notes	If there are more than three attorneys or more than two decisions at a court appearance, this notes field identifies them. Additional appearance related information may also be stored here.

The court would like to upgrade to using a relational database to store their information going forward.

- a. Identify all functional dependencies in the set of fields listed above in the spreadsheet. These can be listed in the form of:

Before identifying all functional dependencies in the set of fields list above, I want to eliminating repeating group (1NF)

Case_number, case_description, plaintiff_firstname, plaintiff_lastname, defendant_firstname, defendant_lastname, attorney_firstname, attorney_lastname, appearance_date, number_attending, decision_discription, extra_appearance_notes.

- According to rules” Each case has one or more court appearances, where the plaintiff, defendant, and their attorneys attend, and decisions are made about the case” I identify functional dependencies below

Case_number, appearance_date → plaintiff_first_name, plaintiff_last_name, attorney_first_names, attorney_last_names, defendant_first_name, defendant_last_name, decision_descriptions, number_attending and extra_appearance_notes.

- plaintiff_first_name, plaintiff_last_name, attorney_first_name, attorney_last_name, defendant_first_name, defendant_last_name, decision_description, number_attending and extra_appearance_notes values are all dependent on- they are determined by the combination of Case_number and appearance_date.

Case_number → plaintiff_first_name, plaintiff_last_name, defendant_first_name, defendant_last_name, case_description.

- I identify this functional dependency from the rule “Each case has one plaintiff and one defendant”

Appearance_date → decision_descriptions, attorney_firstnames, attorney_lastnames

- I identify this functional dependency from the rule “Multiple decisions about the case may be made at each court appearance.” and “every attorney attending a court appearance is assigned a number”.

- b. Suggest a set of normalized relational tables derived from how the court operates and the fields they store. Create a DBMS physical ERD representing this set of tables, which contains the entities, primary and foreign keys, attributes, relationships, and relationship constraints. You may add synthetic primary keys where needed. Make sure that the tables are normalized to BCNF, and to explain your choices.

I have suggested 6 tables:

Case_Number (Case), Defendant, Plaintiff, CourtAppearance (Court Appearance), Decision_Description (Decision), Attorney.

- According to rules the govern how the court operates. I have derived some database structural rules.
 1. *Each case is associated with one plaintiff, each plaintiff is associated with one or many cases. (1:M relationship)*

I define this structural rule according to “Each case has one plaintiff and one defendant” and each plaintiff can associate with many cases.

2. *Each case is associated with one defendant, each defendant is associated with one or many cases. (1:M relationship)*

I define this structural rule according to “Each case has one plaintiff and one defendant”. and, each defendant can associate with many cases.

3. *Each case has one or more court appearance, each appearance is associated with one case. (1:M relationship)*

I define this structural rule according to “Each case has one or more court appearances”

4. *Each court appearance made one or many decisions, each decision is associated with one or many court appearance. (M:N relationship)*

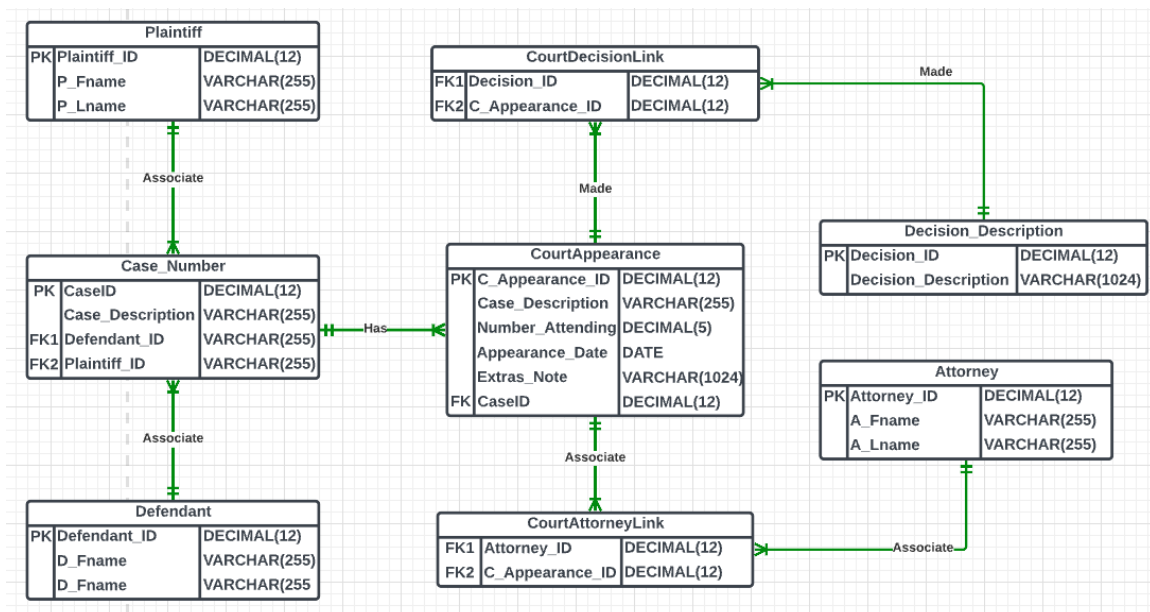
I define this structural rule according to “Multiple decisions about the case may be made at each court appearance”.

5. Each court appearance is associated with one or many attorneys, each attorney is associated with one or many appearances. (M:N relationship).

I define this structural rule according to “Each plaintiff and defendant may retain multiple attorneys for each court appearance.”

Because database structural rule # 4 and 5 are M:N relationship so I will need to create linking entities to convert it to 1:M relationship

Here is my ERD representing set of tables.



Evaluation

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

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

5 points per day will be subtracted for late submissions. Submissions beyond 5 days late will not be accepted. Please contact your facilitator for any exceptions.

Criterion	A	B	C	D	F
Section 1: Sequences (10%)	Sequences have been correctly defined for all tables that need them. All new primary key values are generated using sequences instead of hardcoding values, and foreign key values are populated through use of sequences.	Sequences have been correctly defined for most tables that need them. Most new primary key values are generated using sequences instead of hardcoding values, and most foreign key values are populated through use of sequences.	Sequences have been correctly defined for most some that need them. Some new primary key values are generated using sequences instead of hardcoding values, and some foreign key values are populated through use of sequences.	Sequences have not been defined for most tables that need them. Most new primary key value are hardcoded, and most foreign key values are hardcoded.	No sequences have been created, or all primary and foreign values are hardcoded.
Section 1: Stored Procedures (25%)	All stored procedures accomplish their purpose and completely fulfill their requirements. All stored procedures compile and are executable in the relational database they are written for.	Most stored procedures accomplish their purpose and fulfill their requirements. Most stored procedures compile and are executable in the relational database they are written for.	Some stored procedures accomplish their purpose and somewhat fulfill their requirements. Some stored procedures compile and are executable in the relational database they are written for.	Most stored procedures do not accomplish their purpose, and most requirements are left unfulfilled. Most stored procedures do not compile and are not executable in the relational database they are written for.	The stored procedures are missing, or they do not accomplish any of their purpose and do not fulfill any requirements. The stored procedures may not compile. The syntax may be incorrect.

Section 2: Triggers (25%)	All triggers accomplish their purpose and completely fulfill their requirements. All triggers compile and are executable in the relational database they are written for.	Most triggers accomplish their purpose and fulfill their requirements. Most triggers compile and are executable in the relational database they are written for.	Some triggers accomplish their purpose and somewhat fulfill their requirements. Some triggers compile and are executable in the relational database they are written for.	Most triggers do not accomplish their purpose, and most requirements are left unfulfilled. Most triggers do not compile and are not executable in the relational database they are written for.	The triggers are missing, or they do not accomplish any of their purpose and do not fulfill any requirements. The triggers may not compile. The syntax may be incorrect.
Section 3: Functional Dependencies (5%)	All functional dependencies have been identified. The dependency between the determinant and determined attributes is entirely accurate for all dependencies.	Most functional dependencies have been identified. The dependency between the determinant and determined attributes is accurate for most dependencies.	Some functional dependencies are identified. The dependency between the determinant and determined attributes is accurate for some of the dependencies.	Most functional dependencies are not identified. The dependency between the determinant and determined attributes is inaccurate for most dependencies.	No functional dependencies have been identified, or the dependency between the determinant and determined attributes is inaccurate for all dependencies.
Section 3: Normalization (25%)	Every table in the solution is normalized to BCNF. The design accurately represents the scenario. The relationships between all tables in the solution are accurate and are enforced with foreign keys. Useful primary keys have been provided for all tables. The original table could be reconstructed with joins between the tables. No information has been lost.	Most tables in the solution are normalized to BCNF. The design represents the scenario mostly accurately. The relationships between most tables in the solution are accurate and are enforced with foreign keys. Useful primary keys have been provided for most tables. The original table could be mostly reconstructed with joins between the tables. Little information has been lost.	Some tables in the solution are normalized to BCNF. The design represents the scenario somewhat accurately. The relationships between some tables in the solution are accurate and are enforced with foreign keys. Useful primary keys have been provided for some tables. The original library could be partially reconstructed with joins between the tables. Some information has been lost.	Some tables in the solution are partially normalized to BCNF. The design represents the scenario mostly inaccurately. The relationships between tables in the solution are mostly inaccurate and are not enforced with foreign keys. The primary keys provided are mostly not useful. Most information has been lost.	No tables in the solution are normalized. The design represents the scenario entirely inaccurately. The relationships between tables in the solution are inaccurate. Primary keys may not have been provided. Information needed to reconstruct the original table has been lost.
Overall Presentation (10%)	The explanations supporting all answers are excellent. Queries or other SQL commands have been executed to demonstrate correctness for all answers that need them.	The explanations supporting the answers are good. Queries or other SQL commands have been executed to demonstrate correctness for most answers that need them.	The explanations supporting the answers are satisfactory. Queries or other SQL commands have been executed to demonstrate correctness for some answers that need them.	The explanations supporting the answers are minimal. Queries or other SQL commands have not been executed to demonstrate correctness for most answers that need them.	Explanations for all answers are missing. Queries or other SQL commands have not been executed to demonstrate correctness.

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