# 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 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.

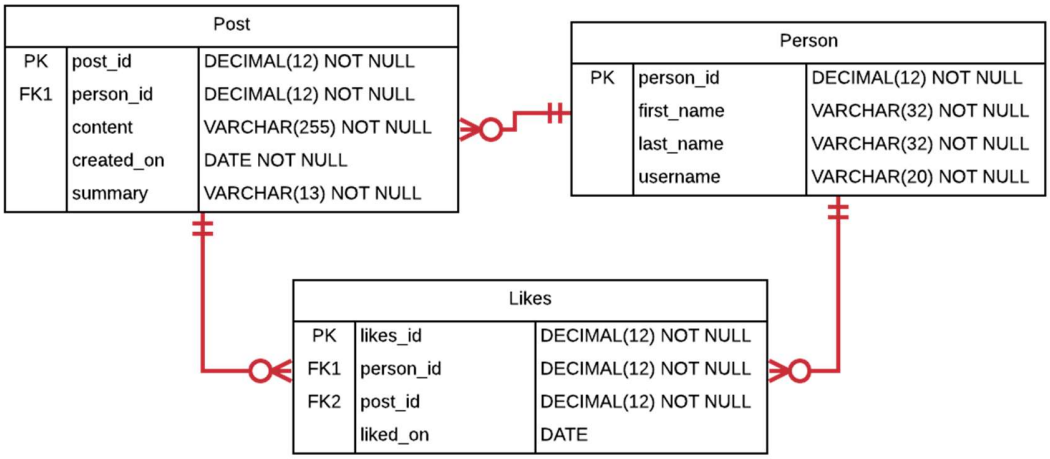
Modern relational DBMS support the creation and use of persistent stored modules, namely, stored procedures and triggers, which are widely used to perform operations critical to modern information systems. A stored procedure contains logic that is executed when a transaction invokes the name of the stored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

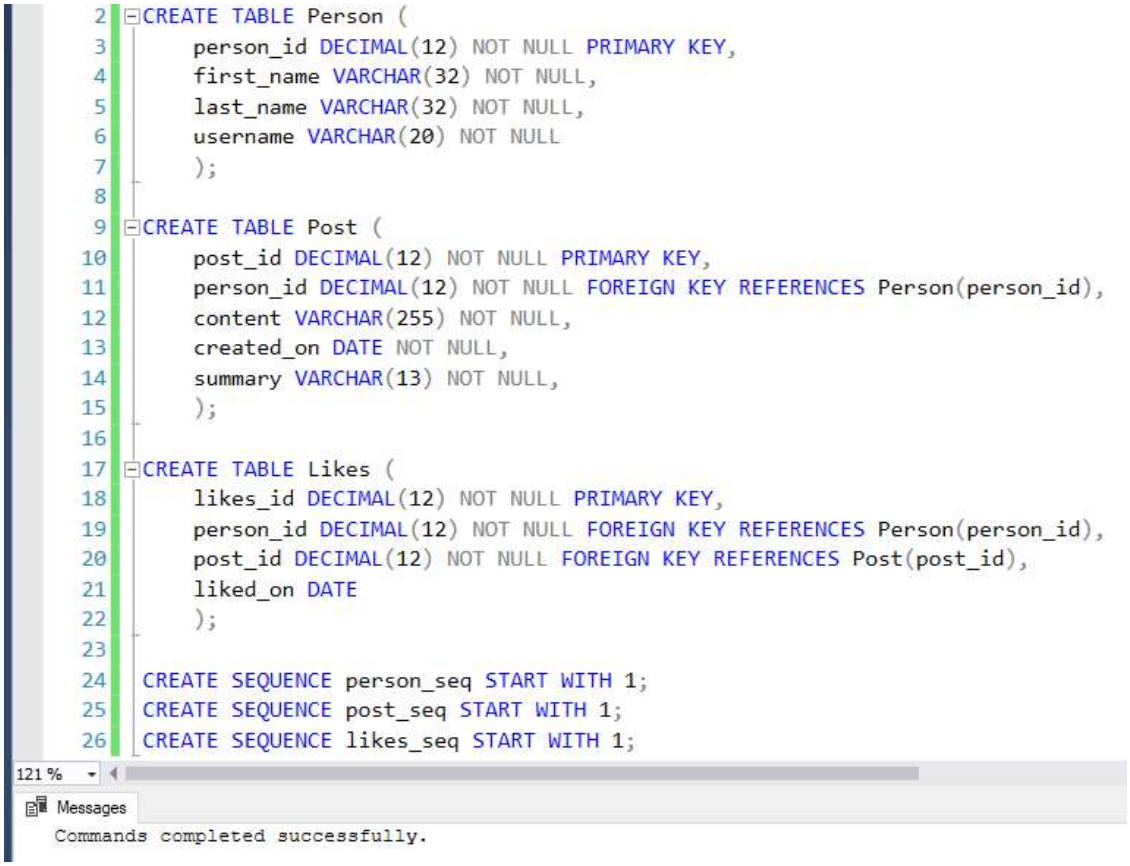
In this first section, you will work with stored procedures on this schema, which offer many significant benefits. Reusability is one significant benefit. The logic contained in a stored procedure can be executed repeatedly, so that each developer need not reinvent the same logic each time it is needed. 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.



```
CREATE TABLE Person (
    person_id DECIMAL(12) NOT NULL PRIMARY KEY,
    first_name VARCHAR(32) NOT NULL,
    last_name VARCHAR(32) NOT NULL,
    username VARCHAR(20) NOT NULL
    );

CREATE TABLE Post (
    post_id DECIMAL(12) NOT NULL PRIMARY KEY,
    person_id DECIMAL(12) NOT NULL FOREIGN KEY REFERENCES Person(person_id),
    content VARCHAR(255) NOT NULL,
    created_on DATE NOT NULL,
    summary VARCHAR(13) NOT NULL,
    );

CREATE TABLE Likes (
    likes_id DECIMAL(12) NOT NULL PRIMARY KEY,
    person_id DECIMAL(12) NOT NULL FOREIGN KEY REFERENCES Person(person_id),
    post_id DECIMAL(12) NOT NULL FOREIGN KEY REFERENCES Post(post_id),
    liked_on DATE
    );

CREATE SEQUENCE person_seq START WITH 1;
CREATE SEQUENCE post_seq START WITH 1;
CREATE SEQUENCE likes_seq START WITH 1;
```

121 %

Messages

```
Commands completed successfully.
```

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 “…”.

```sql
-- Inserts 5 people.
INSERT INTO Person (person_id, first_name, last_name, username)
VALUES (NEXT VALUE FOR person_seq, 'Abby', 'Aoe', 'AA');
INSERT INTO Person (person_id, first_name, last_name, username)
VALUES (NEXT VALUE FOR person_seq, 'Brian', 'Boe', 'BB');
INSERT INTO Person (person_id, first_name, last_name, username)
VALUES (NEXT VALUE FOR person_seq, 'Charlie', 'Coe', 'CC');
INSERT INTO Person (person_id, first_name, last_name, username)
VALUES (NEXT VALUE FOR person_seq, 'Daisy', 'Doe', 'DD');
INSERT INTO Person (person_id, first_name, last_name, username)
VALUES (NEXT VALUE FOR person_seq, 'Elliott', 'Eoe', 'EE');
```

121 %

Messages

```
(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)
```

See lines 46, 50, 55, …: SELECT is used to get person_id for user(name) creating post.

```sql
-- Inserts 8 posts.
DECLARE @content VARCHAR(255) = '';
-- Two posts by user AA.
SET @content = 'Australia is great.';
INSERT INTO Post (post_id, person_id, content, created_on, summary)
VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='AA'),
    @content, '1/13/2022', SUBSTRING(@content, 1, 10) + '...');
SET @content = 'I like Argentina.';
INSERT INTO Post (post_id, person_id, content, created_on, summary)
VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='AA'),
    @content, '1/13/2022', SUBSTRING(@content, 1, 10) + '...');
-- Two posts by user BB.
SET @content = 'Brazilian cuisine is the best.';
INSERT INTO Post (post_id, person_id, content, created_on, summary)
VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='BB'),
    @content, '1/15/2022', SUBSTRING(@content, 1, 10) + '...');
SET @content = 'Belgium has three official languages.';
INSERT INTO Post (post_id, person_id, content, created_on, summary)
VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='BB'),
    @content, '1/15/2022', SUBSTRING(@content, 1, 10) + '...');
```

121 %

Messages

```
(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)
```

```
61  -- Two posts by user CC.
62  SET @content = 'China is a country in East Asia.';
63  INSERT INTO Post (post_id, person_id, content, created_on, summary)
64  VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='CC'),
65      @content, '1/17/2022', SUBSTRING(@content, 1, 10) + '...');
66  SET @content = 'Canada is a country in North America.';
67  INSERT INTO Post (post_id, person_id, content, created_on, summary)
68  VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='CC'),
69      @content, '1/17/2022', SUBSTRING(@content, 1, 10) + '...');
70  -- One posts by user DD.
71  SET @content = 'Denmark is a Scandinavian country.';
72  INSERT INTO Post (post_id, person_id, content, created_on, summary)
73  VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='DD'),
74      @content, '1/19/2022', SUBSTRING(@content, 1, 10) + '...');
75  -- One posts by user EE.
76  SET @content = 'Ecuador has a diverse landscape.';
77  INSERT INTO Post (post_id, person_id, content, created_on, summary)
78  VALUES (NEXT VALUE FOR post_seq, (SELECT person_id FROM Person WHERE username='EE'),
79      @content, '1/21/2022', SUBSTRING(@content, 1, 10) + '...');
```

121 %

Messages

```
(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)
```

See lines 84-85, 88-89, …: SELECTs are used to get person_id for user(name) creating like and post_id for post being liked.

```
81  -- Inserts 4 likes.
82  INSERT INTO Likes (likes_id, person_id, post_id, liked_on)
83  VALUES (NEXT VALUE FOR likes_seq,
84      (SELECT person_id FROM Person WHERE username='AA'),
85      (SELECT post_id FROM Post WHERE content LIKE '%Brazilian%'), '1/24/2022');
86  INSERT INTO Likes (likes_id, person_id, post_id, liked_on)
87  VALUES (NEXT VALUE FOR likes_seq,
88      (SELECT person_id FROM Person WHERE username='BB'),
89      (SELECT post_id FROM Post WHERE content LIKE '%Canada%'), '1/25/2022');
90  INSERT INTO Likes (likes_id, person_id, post_id, liked_on)
91  VALUES (NEXT VALUE FOR likes_seq,
92      (SELECT person_id FROM Person WHERE username='CC'),
93      (SELECT post_id FROM Post WHERE content LIKE '%Denmark%'), '1/26/2022');
94  INSERT INTO Likes (likes_id, person_id, post_id, liked_on)
95  VALUES (NEXT VALUE FOR likes_seq,
96      (SELECT person_id FROM Person WHERE username='DD'),
97      (SELECT post_id FROM Post WHERE content LIKE '%Ecuador%'), '1/27/2022');
```

121 %

Messages

```
(1 row affected)

(1 row affected)

(1 row affected)

(1 row affected)
```

And to check the inserts, the following query is executed – as shown in the results, the inserts were correct:

```
SELECT *
FROM Person
LEFT JOIN Post ON Post.person_id = Person.person_id
LEFT JOIN Likes ON Likes.post_id = Post.post_id;
```

Results Messages

| | person_id | first_name | last_name | username | post_id | person_id | content | created_on | summary | likes_id | person_id | post_id | liked_on |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | Abby | Aoe | AA | 1 | 1 | Australia is great. | 2022-01-13 | Australia ... | NULL | NULL | NULL | NULL |
| 2 | 1 | Abby | Aoe | AA | 2 | 1 | I like Argentina. | 2022-01-13 | I like Arg... | NULL | NULL | NULL | NULL |
| 3 | 2 | Brian | Boe | BB | 3 | 2 | Brazilian cuisine is the best. | 2022-01-15 | Brazilian ... | 1 | 1 | 3 | 2022-01-24 |
| 4 | 2 | Brian | Boe | BB | 4 | 2 | Belgium has three official languages. | 2022-01-15 | Belgium ha... | NULL | NULL | NULL | NULL |
| 5 | 3 | Charlie | Coe | CC | 5 | 3 | China is a country in East Asia. | 2022-01-17 | China is a... | NULL | NULL | NULL | NULL |
| 6 | 3 | Charlie | Coe | CC | 6 | 3 | Canada is a country in North America. | 2022-01-17 | Canada is ... | 2 | 2 | 6 | 2022-01-25 |
| 7 | 4 | Daisy | Doe | DD | 7 | 4 | Denmark is a Scandinavian country. | 2022-01-19 | Denmark is... | 3 | 3 | 7 | 2022-01-26 |
| 8 | 5 | Elliott | Eoe | EE | 8 | 5 | Ecuador has a diverse landscape. | 2022-01-21 | Ecuador ha... | 4 | 4 | 8 | 2022-01-27 |

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.

```
-- Creates (or alters) stored procedure.
CREATE OR ALTER PROCEDURE add_michelle_stella
AS
BEGIN
  INSERT INTO Person (person_id, first_name, last_name, username)
  VALUES (NEXT VALUE FOR person_seq, 'Michelle', 'Stella', 'MS');
END;
```

Messages

```
Commands completed successfully.
```

```
-- Executes stored procedure.
EXECUTE add_michelle_stella
```

Messages

```
(1 row affected)
```

```
SELECT *
FROM Person;
```

Results Messages

| | person_id | first_name | last_name | username |
|---|---|---|---|---|
| 1 | 1 | Abby | Aoe | AA |
| 2 | 2 | Brian | Boe | BB |
| 3 | 3 | Charlie | Coe | CC |
| 4 | 4 | Daisy | Doe | DD |
| 5 | 5 | Elliott | Eoe | EE |
| 6 | 6 | Michelle | Stella | MS |

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.

```
114  -- Creates reusable stored procedure to add one person to Person table with each execution.
115  CREATE OR ALTER PROCEDURE add_person
116      @first_name VARCHAR(32), -- Person's first name.
117      @last_name VARCHAR(32),  -- Person's last name.
118      @username VARCHAR(20)    -- Person's username.
119  AS
120  BEGIN
121    INSERT INTO Person (person_id, first_name, last_name, username)
122    VALUES (NEXT VALUE FOR person_seq, @first_name, @last_name, @username);
123  END;
```

121 %

Messages

```
Commands completed successfully.
```

```
125  EXECUTE add_person 'Frank', 'Foe', 'FF';
```

121 %

Messages

```
(1 row affected)
```

```
127  SELECT *
128  FROM Person;
```

121 %

Results | Messages

| | person_id | first_name | last_name | username |
|---|---|---|---|---|
| 1 | 1 | Abby | Aoe | AA |
| 2 | 2 | Brian | Boe | BB |
| 3 | 3 | Charlie | Coe | CC |
| 4 | 4 | Daisy | Doe | DD |
| 5 | 5 | Elliott | Eoe | EE |
| 6 | 6 | Michelle | Stella | MS |
| 7 | 7 | Frank | Foe | FF |

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.

```
131 CREATE OR ALTER PROCEDURE add_post
132     -- Defines parameters.
133     @person_id DECIMAL(12), -- Person's ID (ID of poster).
134     @content VARCHAR(255),  -- Post's content.
135     @created_on DATE        -- Post's date.
136 AS
137 BEGIN
138     -- Defines variable (can only be accessed from within procedure).
139     DECLARE @summary VARCHAR(13);
140     -- Computes and assigns value to variable.
141     SET @summary = SUBSTRING(@content, 1, 10) + '...';
142     -- SQL query (or queries).
143     INSERT INTO Post (post_id, person_id, content, created_on, summary)
144     VALUES (NEXT VALUE FOR post_seq, @person_id, @content, @created_on, @summary);
145 END;
146 GO -- GO after stored procedure is necessary to 'combine DDL (data definition language)
147 -- with DML (data manipulation language)'.
```

121 %

Messages

```
Commands completed successfully.
```

```
149 EXECUTE add_post 4, 'The Dominican Republic is a Caribbean nation', '1/19/2022';
```

121 %

Messages

```
(1 row affected)
```

```
151 SELECT *
152 FROM Post;
```

121 %

Results | Messages

| | post_id | person_id | content | created_on | summary |
|---|---|---|---|---|---|
| 1 | 1 | 1 | Australia is great. | 2022-01-13 | Australia ... |
| 2 | 2 | 1 | I like Argentina. | 2022-01-13 | I like Arg... |
| 3 | 3 | 2 | Brazilian cuisine is the best. | 2022-01-15 | Brazilian ... |
| 4 | 4 | 2 | Belgium has three official languages. | 2022-01-15 | Belgium ha... |
| 5 | 5 | 3 | China is a country in East Asia. | 2022-01-17 | China is a... |
| 6 | 6 | 3 | Canada is a country in North America. | 2022-01-17 | Canada is ... |
| 7 | 7 | 4 | Denmark is a Scandinavian country. | 2022-01-19 | Denmark is... |
| 8 | 8 | 5 | Ecuador has a diverse landscape. | 2022-01-21 | Ecuador ha... |
| 9 | 9 | 4 | The Dominican Republic is a Caribbean nation | 2022-01-19 | The Domini... |

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.

```sql
CREATE OR ALTER PROCEDURE add_like
    -- Defines parameters.
    @username VARCHAR(20), -- Username of person who likes post.
    @post_id DECIMAL(12),  -- Post's ID (of liked post).
    @liked_on DATE         -- Like's date.
AS
BEGIN
   -- Defines variable.
   DECLARE @person_id DECIMAL(12);
   -- Gets person_id based upon username and assigns value to @person_id.
   SET @person_id = (SELECT person_id FROM Person WHERE username=@username);
   -- SQL query (or queries).
   INSERT INTO Likes(likes_id, person_id, post_id, liked_on)
   VALUES (NEXT VALUE FOR likes_seq, @person_id, @post_id, @liked_on);
END;
GO
```

121 %

Messages

```
Commands completed successfully.
```

```sql
EXECUTE add_like 'EE', 1, '1/28/2022';
```

121 %

Messages

```
(1 row affected)
```

```sql
SELECT *
FROM Likes;
```

121 %

Results | Messages

| | likes_id | person_id | post_id | liked_on |
|---|---|---|---|---|
| 1 | 1 | 1 | 3 | 2022-01-24 |
| 2 | 2 | 2 | 6 | 2022-01-25 |
| 3 | 3 | 3 | 7 | 2022-01-26 |
| 4 | 4 | 4 | 8 | 2022-01-27 |
| 5 | 5 | 5 | 1 | 2022-01-28 |

As can be seen, the like has been added to the Likes table; person_id 5 belongs to username ‘EE’.

# 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.

```
178  -- Creates trigger.
179  CREATE TRIGGER valid_summary_trigger
180  ON Post AFTER INSERT, UPDATE -- Is trigger on table Post; trigger fires 'AFTER INSERT, UPDATE'.
181  AS
182  BEGIN
183    DECLARE @inserted_content VARCHAR(255);
184    DECLARE @inserted_summary VARCHAR(255);
185    -- Uses INSERTED (pseudo table) to get last inserted (or updated) content and summary values.
186    SET @inserted_content = (SELECT INSERTED.content FROM INSERTED);
187    SET @inserted_summary = (SELECT INSERTED.summary FROM INSERTED);
188
189    -- Checks if substring of @inserted_content (character 1 to 10, inclusive) + '...'
190    -- is not equal to @inserted_summary:
191    -- If True, rolls back transaction in-progress and raises error;
192    -- trigger is part if INSERT or UPDATE transaction in-progress.
193    IF (SUBSTRING(@inserted_content, 1, 10) + '...') <> @inserted_summary
194    BEGIN
195      ROLLBACK;
196      -- 14 and 1 are level and state shown in raised error message.
197      RAISERROR('Summary is not first 10 characters of content followed by 3 dots', 14, 1);
198    END; -- End if block.
199  END; -- End trigger block.
```

```
121 %
Messages
Commands completed successfully.
```

```
201  DECLARE @content VARCHAR(255) = '1234567890ABC';
202  -- Valid insert - will have summary length of 10 plus '...' at end ('1234567890...').
203  INSERT INTO Post (post_id, person_id, content, created_on, summary)
204  VALUES (NEXT VALUE FOR post_seq, 5, @content, '1/21/2022', SUBSTRING(@content, 1, 10) + '...');
```

```
121 %
Messages

(1 row affected)
```

```
205 -- Invalid insert - will have summary length of 11 and no '...' at the end ('1234567890A')
206 INSERT INTO Post (post_id, person_id, content, created_on, summary)
207 VALUES (NEXT VALUE FOR post_seq, 5, @content, '1/21/2022', SUBSTRING(@content, 1, 11));
```

121 %

Messages

```
Msg 50000, Level 14, State 1, Procedure valid_summary_trigger, Line 19 [Batch Start Line 204]
Summary is not first 10 characters of content followed by 3 dots
Msg 3609, Level 16, State 1, Line 207
The transaction ended in the trigger. The batch has been aborted.
```

```
209 SELECT *
210 FROM Post;
```

121 %

Results | Messages

| | post_id | person_id | content | created_on | summary |
|---|---|---|---|---|---|
| 1 | 1 | 1 | Australia is great. | 2022-01-13 | Australia ... |
| 2 | 2 | 1 | I like Argentina. | 2022-01-13 | I like Arg... |
| 3 | 3 | 2 | Brazilian cuisine is the best. | 2022-01-15 | Brazilian ... |
| 4 | 4 | 2 | Belgium has three official languages. | 2022-01-15 | Belgium ha... |
| 5 | 5 | 3 | China is a country in East Asia. | 2022-01-17 | China is a... |
| 6 | 6 | 3 | Canada is a country in North America. | 2022-01-17 | Canada is ... |
| 7 | 7 | 4 | Denmark is a Scandinavian country. | 2022-01-19 | Denmark is... |
| 8 | 8 | 5 | Ecuador has a diverse landscape. | 2022-01-21 | Ecuador ha... |
| 9 | 9 | 4 | The Dominican Republic is a Caribbean nation | 2022-01-19 | The Domini... |
| 10 | 10 | 5 | 1234567890ABC | 2022-01-21 | 1234567890... |

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.

```sql
CREATE TRIGGER cross_table_validation_trigger
ON Likes AFTER INSERT, UPDATE -- Trigger on table Likes; trigger fires 'AFTER INSERT, UPDATE'.
AS
BEGIN
  -- Declares local variables (local to trigger).
  DECLARE @liked_on_date DATE;
  DECLARE @created_on_date DATE;
  -- Joins Likes and Post tables on post_id to get Post.created_on date;
  -- assigns INSERTED.liked_on to @liked_on_date and Post.created_on to @created_on_date.
  SELECT @liked_on_date = INSERTED.liked_on,
         @created_on_date = Post.created_on
  FROM   Post
  JOIN   INSERTED ON INSERTED.post_id = Post.post_id;

  -- Checks if @liked_on_date date is before @created_on_date.
  -- If True, rolls back transaction in-progress and raises error.
  IF @liked_on_date < @created_on_date
  BEGIN
    ROLLBACK;
    RAISERROR('liked_on date cannot be before created_on date of post', 14, 1);
  END;
END;
```

121 %

Messages

```
Commands completed successfully.
```

```sql
-- Post with post_id 1 was created 1/13/2022.
-- Valid like.
EXECUTE add_like 'FF', 1, '1/13/2022';
```

121 %

Messages

```
(1 row affected)
```

```sql
-- Invalid like.
EXECUTE add_like 'FF', 1, '1/12/2022';
```

121 %

Messages

```
Msg 50000, Level 14, State 1, Procedure cross_table_validation_trigger, Line 20 [Batch Start Line 233]
liked_on date cannot be before created_on date of post
Msg 3609, Level 16, State 1, Procedure add_like, Line 13 [Batch Start Line 233]
The transaction ended in the trigger. The batch has been aborted.
```

```sql
SELECT *
FROM Likes;
```

121 %

Results | Messages

| | likes_id | person_id | post_id | liked_on |
|---|---|---|---|---|
| 1 | 1 | 1 | 3 | 2022-01-24 |
| 2 | 2 | 2 | 6 | 2022-01-25 |
| 3 | 3 | 3 | 7 | 2022-01-26 |
| 4 | 4 | 4 | 8 | 2022-01-27 |
| 5 | 5 | 5 | 1 | 2022-01-28 |
| 6 | 6 | 7 | 1 | 2022-01-13 |

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).

```
241 -- Creates history table:
242 -- with foreign key to Post which has been changed;
243 -- old_content which will hold content before change;
244 -- new_content which will hold content after change; and change_date.
245 CREATE TABLE post_content_history (
246     post_id DECIMAL(12) NOT NULL FOREIGN KEY REFERENCES Post(post_id),
247     old_content VARCHAR(255) NOT NULL,
248     new_content VARCHAR(255) NOT NULL,
249     change_date DATE NOT NULL
250     );
```

121 %

Messages

```
Commands completed successfully.
```

```
252 -- Creates history trigger.
253 CREATE TRIGGER post_content_history_trigger
254 ON Post AFTER UPDATE -- Trigger on table Post; trigger fires 'AFTER UPDATE'.
255 AS
256 BEGIN
257   -- Declares local variables and assigns:
258   -- post_id (of changed row; gets value from pseudo table INSERTED),
259   -- content before change (gets value from pseudo table DELETED), and
260   -- content after change (gets value from pseudo table INSERTED).
261   DECLARE @post_id DECIMAL(12) = (SELECT post_id FROM INSERTED);
262   DECLARE @old_content VARCHAR(255) = (SELECT content FROM DELETED);
263   DECLARE @new_content VARCHAR(255) = (SELECT content FROM INSERTED);
264
265 -- If old and new content differ, then content was changed and content value
266 -- before change and content value after change are recoreded in history table
267 -- with corresponding post_id (foreign key) and current date (GETDATE()).
268   IF @old_content <> @new_content
269   BEGIN
270     INSERT INTO post_content_history (post_id, old_content, new_content, change_date)
271     VALUES(@post_id, @old_content, @new_content, GETDATE());
272   END;
273 END;
```

121 %

Messages

```
Commands completed successfully.
```

```
280 -- Change - history trigger will fire.
281 DECLARE @content VARCHAR(255) = 'Update';
282 UPDATE Post
283 SET content = @content, summary = (SUBSTRING(@content, 1, 10) + '...')
284 WHERE post_id = 1;
```

121 %

Messages

```
(1 row affected)

(1 row affected)
```

It was indicated that two rows were affected. This is because the row with post_id 1 was updated in the Post table and a row was added to the history table post_content_history:

```
286 SELECT *
287 FROM post_content_history;
```

121 %

Results | Messages

| | post_id | old_content | new_content | change_date |
|---|---|---|---|---|
| 1 | 1 | Australia is great. | Update | 2022-01-29 |

# 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:
column1,column2,… ➔ column3, column4…

   case_number ➔ case_description, plaintiff_first_name, plaintiff_last_name, defendant_first_name, defendant_last_name
   **Explanation:**
   case_number is a unique number assigned to each case, so case_number can be seen as the primary key of a Case table; thus, case_number can specify any attribute in a Case table. According to the task, each Case (table) has a case_description, a plaintiff (plaintiff_first_name and plaintiff_last_name), and a defendant (defendant_first_name and defendant_last_name). Thus, case_number can be used to determine any combination of case_description, plaintiff_first_name, plaintiff_last_name, defendant_first_name, and defendant_last_name: e.g.,

case_number ➔ case_description
case_number ➔ plaintiff_first_name, plaintiff_last_name
case_number ➔ plaintiff_first_name
case_number ➔ plaintiff_last_name
case_number ➔ defendant_first_name, defendant_last_name
and so on.
However, not every combination is now listed because the first functional dependency already says that case_number can determine every combination of plaintiff_first_name, plaintiff_last_name, defendant_first_name, defendant_last_name.

case_number ➔ attorney1_first_name, attorney1_last_name, attorney2_first_name, attorney2_last_name, attorney3_first_name, attorney3_last_name
**Explanation:**
Each case_number belongs to a case, each case has a plaintiff and defendant, each plaintiff and defendant may have many attorneys: so case_number must be able to determine attorneyX_first_name, attorneyX_last_name.

case_number ➔ appearance_date
**Explanation:**
A case can have multiple appearances, but 'only one court appearance per day for the same case', thus case_number determines the date (appearance_date).

case_number, appearance_date ➔ case_description, plaintiff_first_name, plaintiff_last_name, defendant_first_name, defendant_last_name**,** attorney1_first_name, attorney1_last_name, attorney2_first_name, attorney2_last_name, attorney3_first_name, attorney3_last_name, appearance_date, number_attending, decision1_description, decision2_description, extra_appearance_notes
**Explanation:**
Note: To the right of the arrow all attributes are listed except case_number and appearance_date (they are listed on the left). case_number and appearance_date serve together as the primary key for appearance (Appearance table), thus they identify appearance uniquely, because the constraint 'only one court appearance per day for the same case' was given in the task, and with it everything that has to do with the appearance.

extra_appearance_notes ➔ attorney1_first_name, attorney1_last_name, attorney2_first_name, attorney2_last_name, attorney3_first_name, attorney3_last_name, attorneyX_first_name, attorneyX_last_name
**Explanation:**
attorney**X**_first_name and attorney**X**_last_name mean attorney4_first_name, attorney5_, etc.

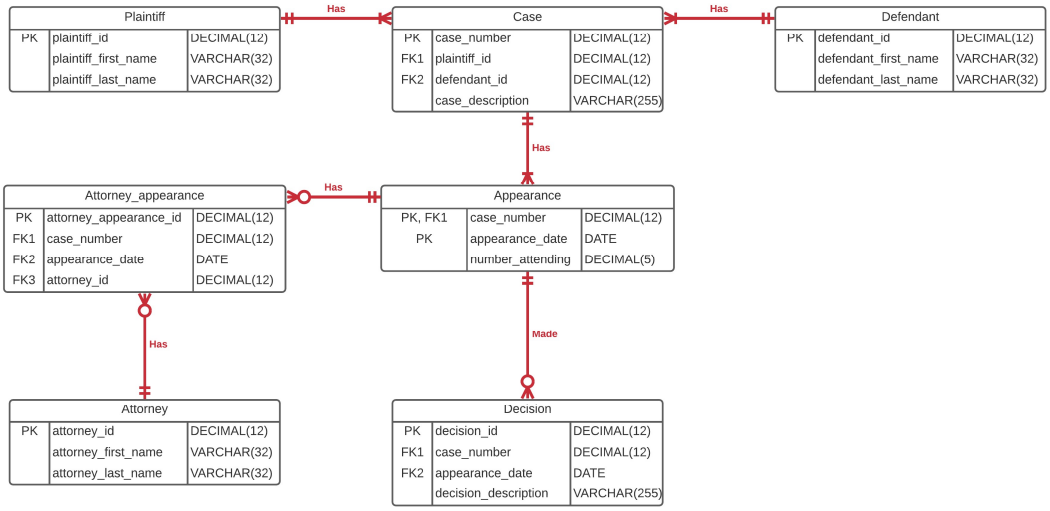
Since the task states: ‘If there are more than three attorneys (…) at a court appearance, this notes field identifies’ them, thus there is a functional dependency.

extra_appearance_notes ➔ decision1_description, decision2_description, decisionX_description

**Explanation:**

Since the task states: ‘If there are more than (…) two decisions at a court appearance, this notes field identifies’ them, thus there is a functional dependency.

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 used the exact field names from the table above so it is absolutely clear which fields were put into which table/entity. The court has a list of cases (Case table), each case has exactly one plaintiff (Plaintiff table) and defendant (Defendant table). Each plaintiff and defendant can be part of several cases, but must be part of at least one case, otherwise they are not plaintiff or defendant. Each case has one or many court appearances (Appearance table), and in each appearance one case is handled. Furthermore, the task states that each appearance can have 0 to many decisions (Decision table); and a decision must belong to one appearance. ‘Each plaintiff and defendant may retain multiple attorneys (Attorney table) for each court appearance’; and each court appearance may have many attorneys. Since there is a many-to-many relationship, a bridge table must be inserted (Attorney_appearance table) to resolve the many-to-many relationship. A primary key was inserted into

each entity, so that the rows in each entity can be uniquely identified. There is one composite primary key case_number, appearance_date in table Appearance. The foreign keys of the bridge table each reference the primary keys of the entities that the bridge table bridges. And in a one-to-many relationship, the foreign key is in the entity that is related to at most one of the other entity. And the other fields mentioned in the above table were assigned to the tables/entities.

Then the physical ERD was normalized to BCNF:

- 1NF: each table has a primary key and each cell has a single value (e.g., ID column has one number per cell and not two, three or more.
  1NF has been achieved.

- 2NF: Achieved 1NF and no partial dependencies (partial dependencies: attribute is dependent on only a portion of a composite primary key)
  2NF has been achieved (only one composite primary key is used, see Appearance table, and the non-primary key attribute number_attending depends on the entire composite primary key).

- 3NF: Achieved 2NF and no transitive dependencies (transitive dependencies: dependency between non-key attributes exist).
  Removed field extra_appearance_notes and thus achieved 3NF, because there is no other transitive dependency.

- BCNF: Achieved 3NF and no table has more than one candidate key (candidate key: minimal set of attributes that can uniquely identify rows).
  Which is already the case, BCNF achieved.