MET CS 669 Database Design and Implementation for Business
Lab 4: Procedural SQL

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 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.

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 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.

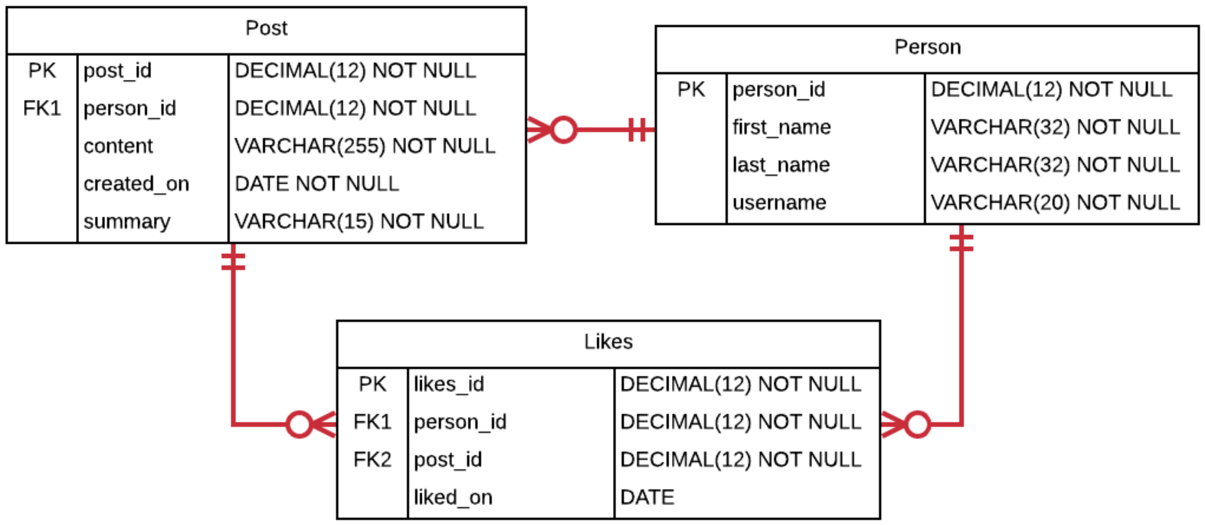
It is important for us to avoid this confusion by recognizing that there are two distinct languages within a relational DBMS – declarative and procedural – and that both are treated very differently within a DBMS in concept and in implementation. In concept, we use the SQL declarative language to tell 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.



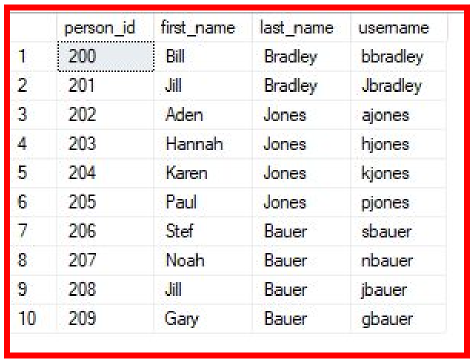
The Person table contains a primary key, the person's first and last name, and the person's username that they use to login to the social networking website. The Post table contains a primary key, a foreign key to the Person that made the post, a shortened content field containing the text of the post, a created_on date, and a summary of the content which is the first 12 characters followed by "…". For example, if the content is "Check out my new pictures.", then the summary would be "Check out my...". 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.

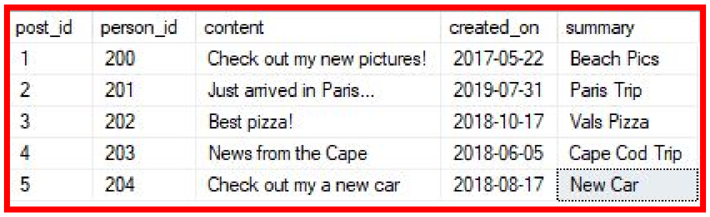
As a reminder, for each step that requires SQL, make sure to capture a screenshot of the command and the results of its execution.

Section Steps

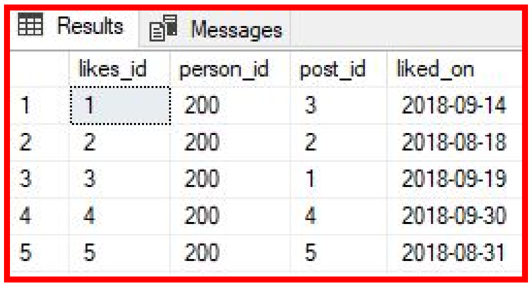
1. Create the tables in the social networking schema, including all of their columns, datatypes, and constraints. Populate the tables with data, ensuring that there are at least 5 people, at least 8 posts, and at least 4 likes. Most of the fields are self-explanatory. As far as the "content" field in 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 12 characters of the content, followed by "…".



	person_id	first_name	last_name	usemame
1	200	Bill	Bradley	bbradley
2	201	Jill	Bradley	Jbradley
3	202	Aden	Jones	ajones
4	203	Hannah	Jones	hjones
5	204	Karen	Jones	kjones
6	205	Paul	Jones	pjones
7	206	Stef	Bauer	sbauer
8	207	Noah	Bauer	nbauer
9	208	Jill	Bauer	jbauer
10	209	Gary	Bauer	gbauer

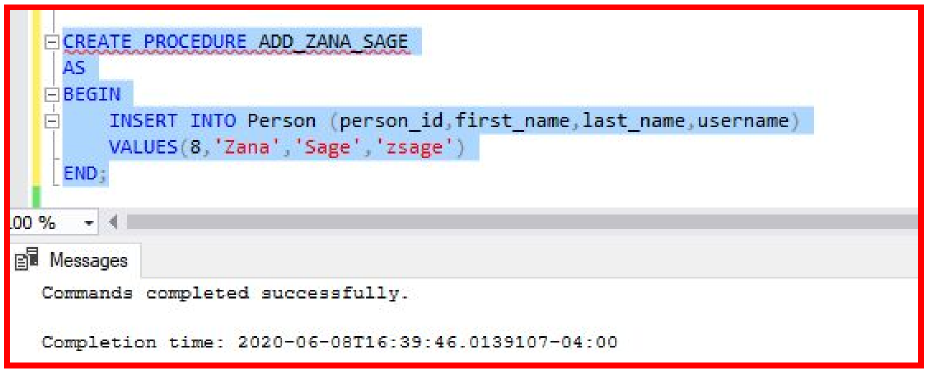


post_id	person_id	content	created_on	summary
1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	201	Just arrived in Paris...	2019-07-31	Paris Trip
3	202	Best pizza!	2018-10-17	Vals Pizza
4	203	News from the Cape	2018-06-05	Cape Cod Trip
5	204	Check out my a new car	2018-08-17	New Car



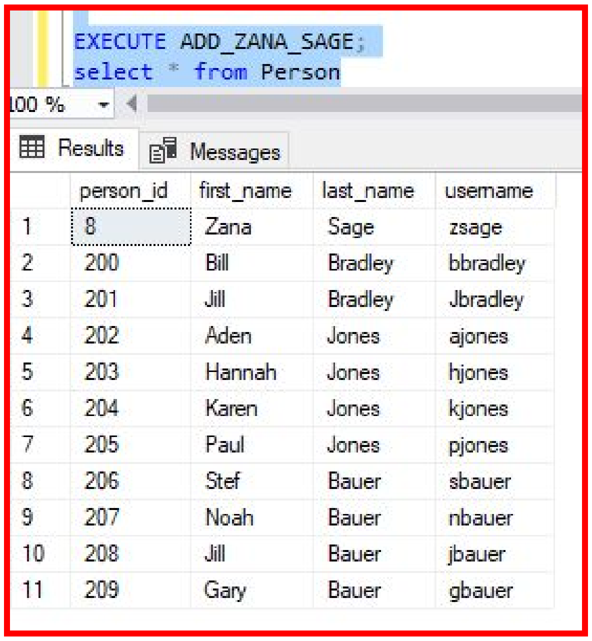
	likes_id	person_id	post_id	liked_on
1	1	200	3	2018-09-14
2	2	200	2	2018-08-18
3	3	200	1	2018-09-19
4	4	200	4	2018-09-30
5	5	200	5	2018-08-31

2. Create a stored procedure named “add_zana_sage” which adds a person named “Zana Sage” to the Person table, then execute the stored procedure. List out the rows in the Person table to show that Zana Sage has been added.



```
CREATE PROCEDURE ADD_ZANA_SAGE
AS
BEGIN
    INSERT INTO Person (person_id,first_name,last_name,username)
    VALUES(8,'Zana','Sage','zsage')
END;
```

```
Commands completed successfully.

Completion time: 2020-06-08T16:39:46.0139107-04:00
```



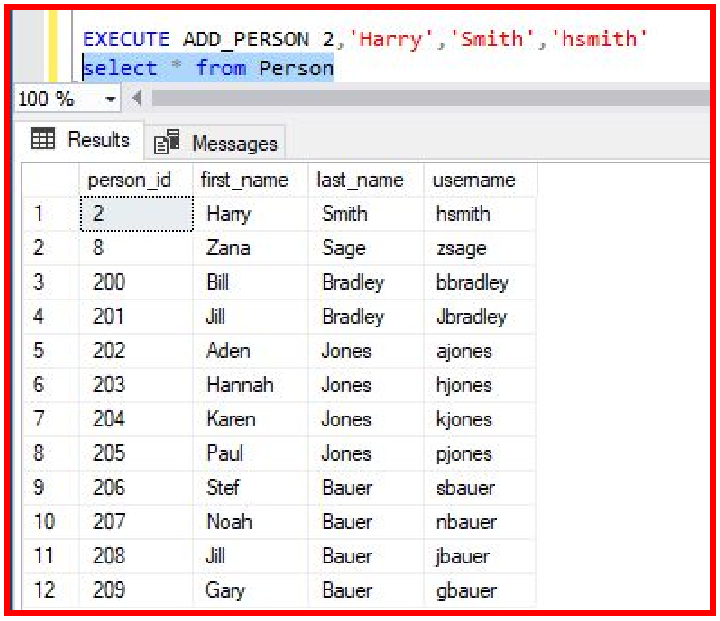
```
EXECUTE ADD_ZANA_SAGE;
select * from Person
```

	person_id	first_name	last_name	username
1	8	Zana	Sage	zsage
2	200	Bill	Bradley	bbradley
3	201	Jill	Bradley	Jbradley
4	202	Aden	Jones	ajones
5	203	Hannah	Jones	hjones
6	204	Karen	Jones	kjones
7	205	Paul	Jones	pjones
8	206	Stef	Bauer	sbauer
9	207	Noah	Bauer	nbauer
10	208	Jill	Bauer	jbauer
11	209	Gary	Bauer	gbauer

3. Attempt to execute the "add_zana_sage" procedure a second time. Summarize what the issue is from the error that occurs as a result.

   Primary Key Constraint Violation - The procedure attempted to add the same primary key into the table.

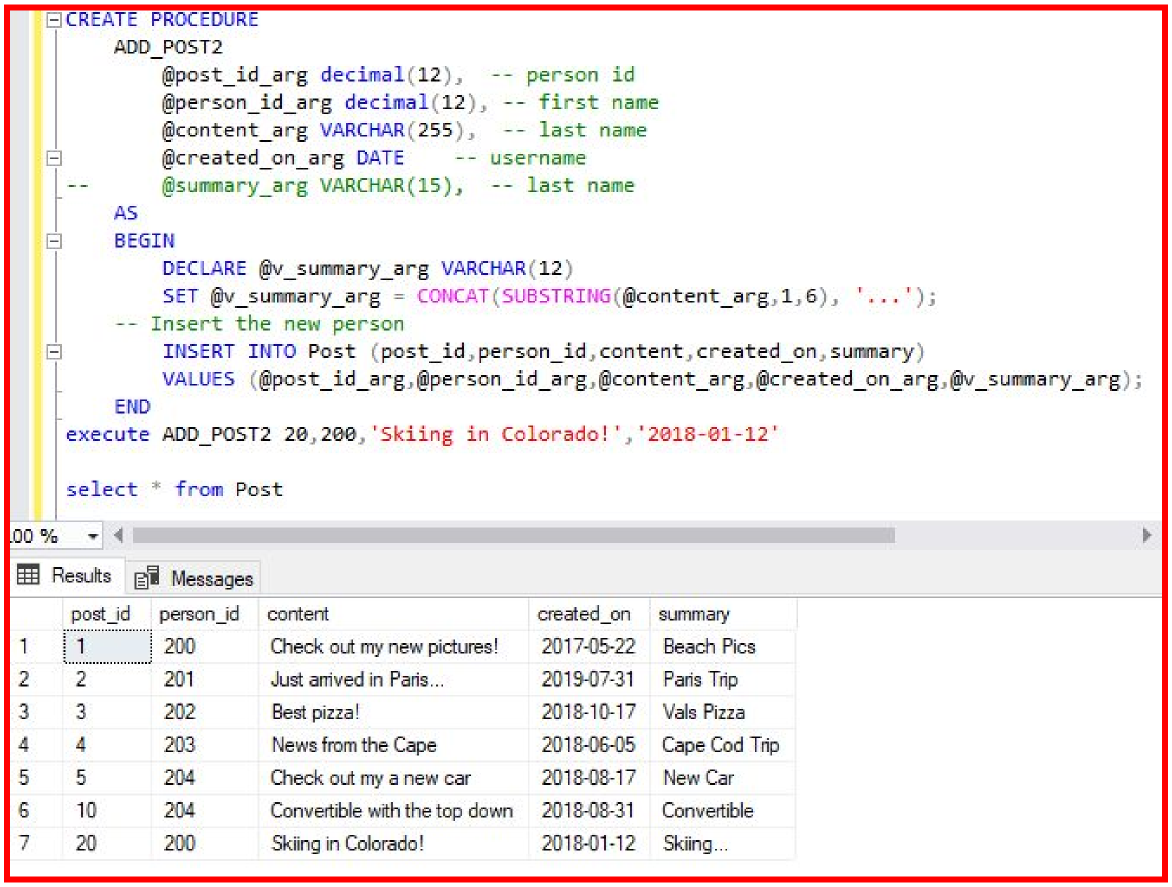
4. Create a reusable stored procedure named "add_person" that uses parameters and allows you to insert any new person into the Person table. Execute the stored procedure with a person of your choosing, then list out the Person table to show that the person was added to the table.



```sql
EXECUTE ADD_PERSON 2,'Harry','Smith','hsmith'
select * from Person
```

	person_id	first_name	last_name	username
1	2	Harry	Smith	hsmith
2	8	Zana	Sage	zsage
3	200	Bill	Bradley	bbradley
4	201	Jill	Bradley	Jbradley
5	202	Aden	Jones	ajones
6	203	Hannah	Jones	hjones
7	204	Karen	Jones	kjones
8	205	Paul	Jones	pjones
9	206	Stef	Bauer	sbauer
10	207	Noah	Bauer	nbauer
11	208	Jill	Bauer	jbauer
12	209	Gary	Bauer	gbauer

5. 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 12 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.

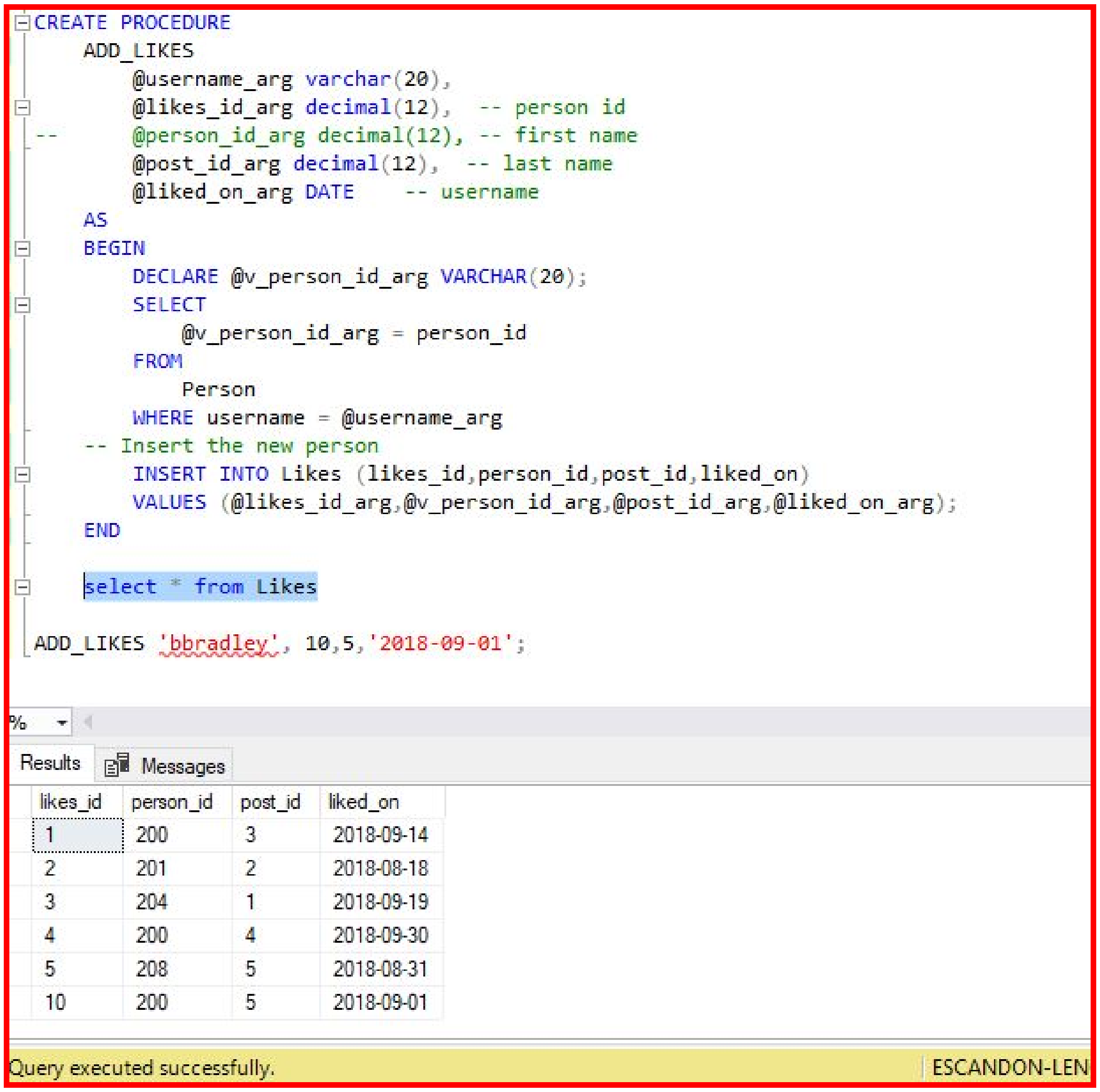


```
CREATE PROCEDURE
    ADD_POST2
        @post_id_arg decimal(12),   -- person id
        @person_id_arg decimal(12), -- first name
        @content_arg VARCHAR(255),   -- last name
        @created_on_arg DATE     -- username
--      @summary_arg VARCHAR(15),   -- last name
    AS
    BEGIN
        DECLARE @v_summary_arg VARCHAR(12)
        SET @v_summary_arg = CONCAT(SUBSTRING(@content_arg,1,6), '...');
    -- Insert the new person
        INSERT INTO Post (post_id,person_id,content,created_on,summary)
        VALUES (@post_id_arg,@person_id_arg,@content_arg,@created_on_arg,@v_summary_arg);
    END
execute ADD_POST2 20,200,'Skiing in Colorado!','2018-01-12'

select * from Post
```

	post_id	person_id	content	created_on	summary
1	1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	2	201	Just arrived in Paris...	2019-07-31	Paris Trip
3	3	202	Best pizza!	2018-10-17	Vals Pizza
4	4	203	News from the Cape	2018-06-05	Cape Cod Trip
5	5	204	Check out my a new car	2018-08-17	New Car
6	10	204	Convertible with the top down	2018-08-31	Convertible
7	20	200	Skiing in Colorado!	2018-01-12	Skiing...

I have two 'add_post' routines… one that takes the substring of 12 chars + '…' and this second one that takes 5 chars + '…' for the summary.

6. 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 parameters 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.



```
CREATE PROCEDURE
    ADD_LIKES
        @username_arg varchar(20),
        @likes_id_arg decimal(12),   -- person id
--      @person_id_arg decimal(12), -- first name
        @post_id_arg decimal(12),   -- last name
        @liked_on_arg DATE     -- username
    AS
    BEGIN
        DECLARE @v_person_id_arg VARCHAR(20);
        SELECT
            @v_person_id_arg = person_id
        FROM
            Person
        WHERE username = @username_arg
    -- Insert the new person
        INSERT INTO Likes (likes_id,person_id,post_id,liked_on)
        VALUES (@likes_id_arg,@v_person_id_arg,@post_id_arg,@liked_on_arg);
    END

    select * from Likes

ADD_LIKES 'bbradley', 10,5,'2018-09-01';
```

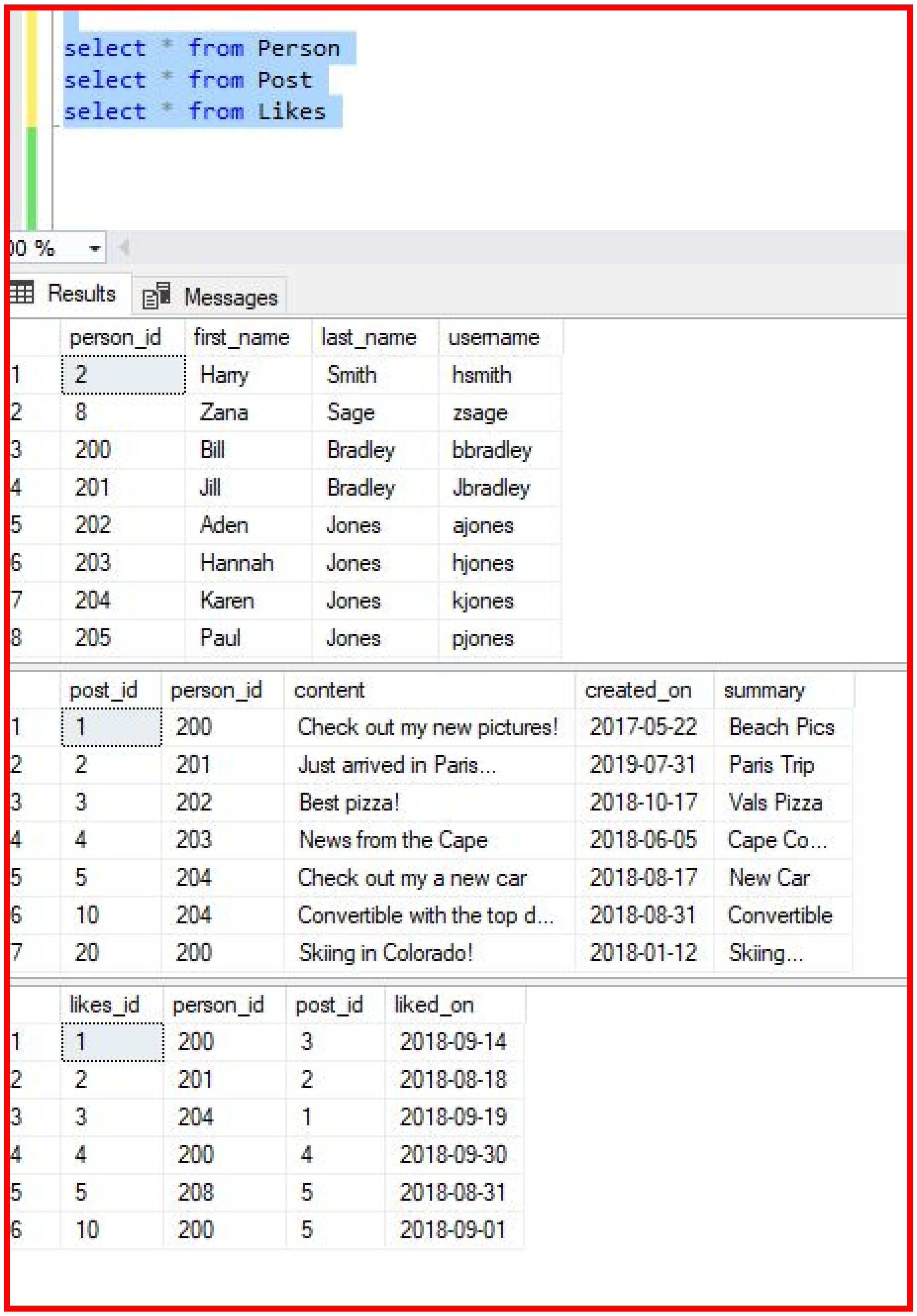
Results | Messages

likes_id	person_id	post_id	liked_on
1	200	3	2018-09-14
2	201	2	2018-08-18
3	204	1	2018-09-19
4	200	4	2018-09-30
5	208	5	2018-08-31
10	200	5	2018-09-01

Query executed successfully. ESCANDON-LEN

Added like from 'bbradley' dated september 1, 2018

7. Create a reusable stored procedure named “delete_person” that takes only one parameter, the username of a person, and deletes all record of that person from the database. This means deleting all of a person’s posts, likes, and the Person record itself. Execute the procedure to delete a person of your choosing (make sure the person has at least one post and at least one like). List out all three tables to show that all record of the person is gone.



```
select * from Person
select * from Post
select * from Likes
```

00 %

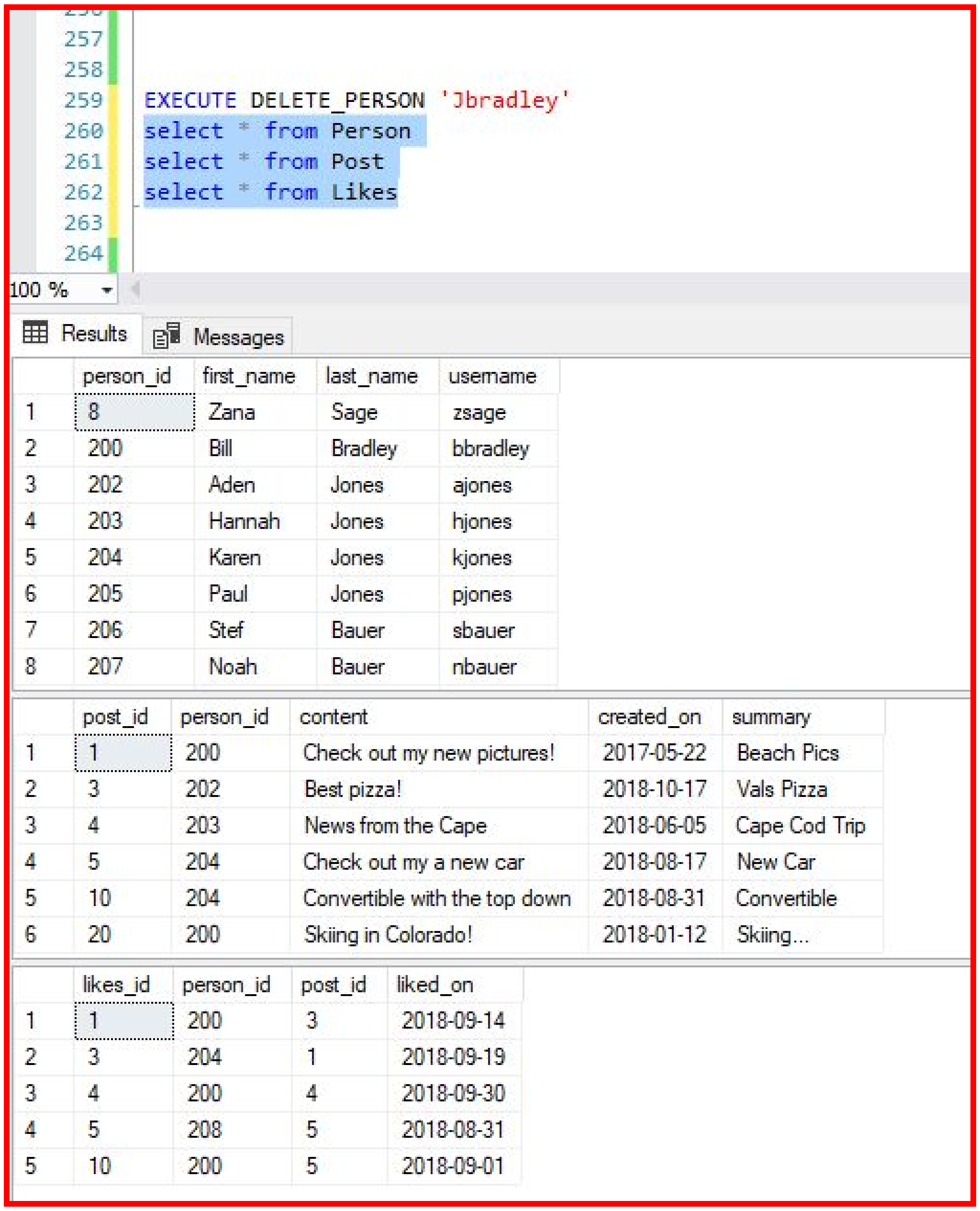
Results | Messages

	person_id	first_name	last_name	username
1	2	Harry	Smith	hsmith
2	8	Zana	Sage	zsage
3	200	Bill	Bradley	bbradley
4	201	Jill	Bradley	Jbradley
5	202	Aden	Jones	ajones
6	203	Hannah	Jones	hjones
7	204	Karen	Jones	kjones
8	205	Paul	Jones	pjones

	post_id	person_id	content	created_on	summary
1	1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	2	201	Just arrived in Paris...	2019-07-31	Paris Trip
3	3	202	Best pizza!	2018-10-17	Vals Pizza
4	4	203	News from the Cape	2018-06-05	Cape Co...
5	5	204	Check out my a new car	2018-08-17	New Car
6	10	204	Convertible with the top d...	2018-08-31	Convertible
7	20	200	Skiing in Colorado!	2018-01-12	Skiing...

	likes_id	person_id	post_id	liked_on
1	1	200	3	2018-09-14
2	2	201	2	2018-08-18
3	3	204	1	2018-09-19
4	4	200	4	2018-09-30
5	5	208	5	2018-08-31
6	10	200	5	2018-09-01

Before deletion of ‘Jbradley’, person_id: 201



```
EXECUTE DELETE_PERSON 'Jbradley'
select * from Person
select * from Post
select * from Likes
```

100 %

Results | Messages

	person_id	first_name	last_name	username
1	8	Zana	Sage	zsage
2	200	Bill	Bradley	bbradley
3	202	Aden	Jones	ajones
4	203	Hannah	Jones	hjones
5	204	Karen	Jones	kjones
6	205	Paul	Jones	pjones
7	206	Stef	Bauer	sbauer
8	207	Noah	Bauer	nbauer

	post_id	person_id	content	created_on	summary
1	1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	3	202	Best pizza!	2018-10-17	Vals Pizza
3	4	203	News from the Cape	2018-06-05	Cape Cod Trip
4	5	204	Check out my a new car	2018-08-17	New Car
5	10	204	Convertible with the top down	2018-08-31	Convertible
6	20	200	Skiing in Colorado!	2018-01-12	Skiing...

	likes_id	person_id	post_id	liked_on
1	1	200	3	2018-09-14
2	3	204	1	2018-09-19
3	4	200	4	2018-09-30
4	5	208	5	2018-08-31
5	10	200	5	2018-09-01

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.

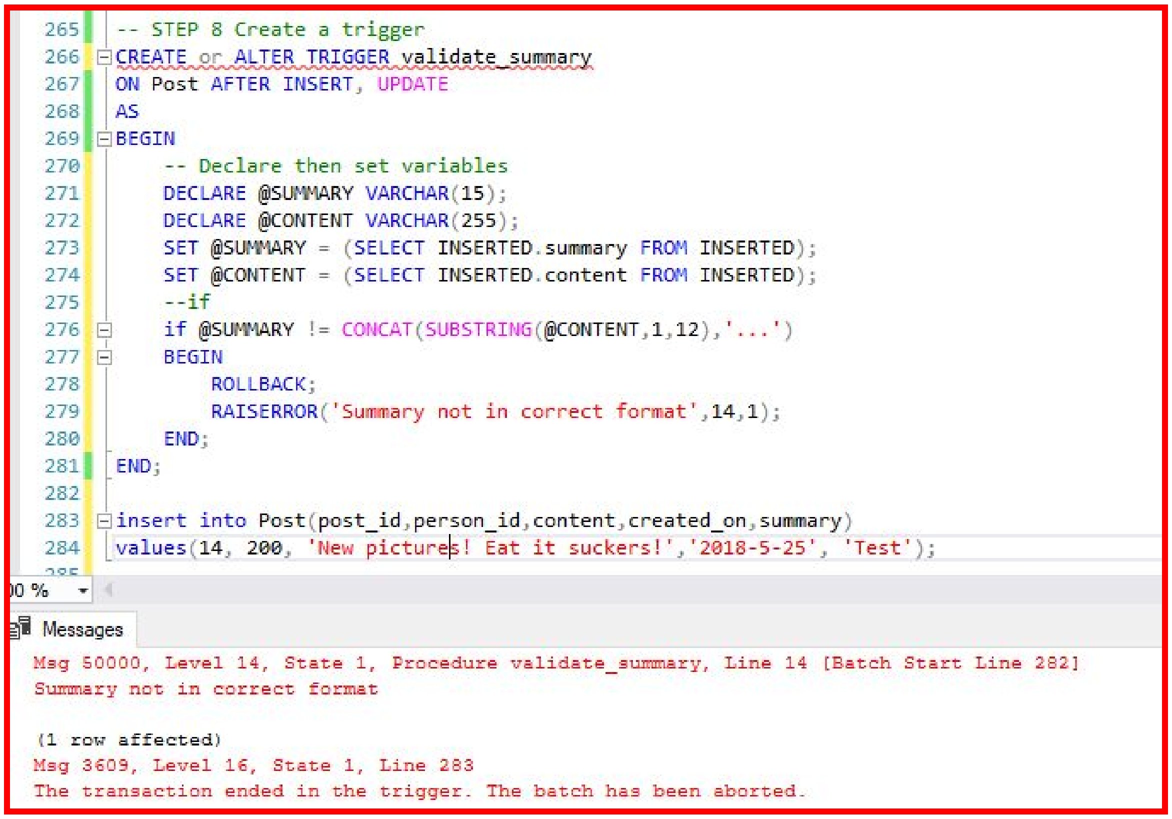
Triggers also have significant drawbacks. By default triggers execute within the same transaction as the event that caused the trigger to execute, and so any failure of the trigger results in the abortion of the overall transaction. Triggers execute additional code beyond the regular processing of the 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

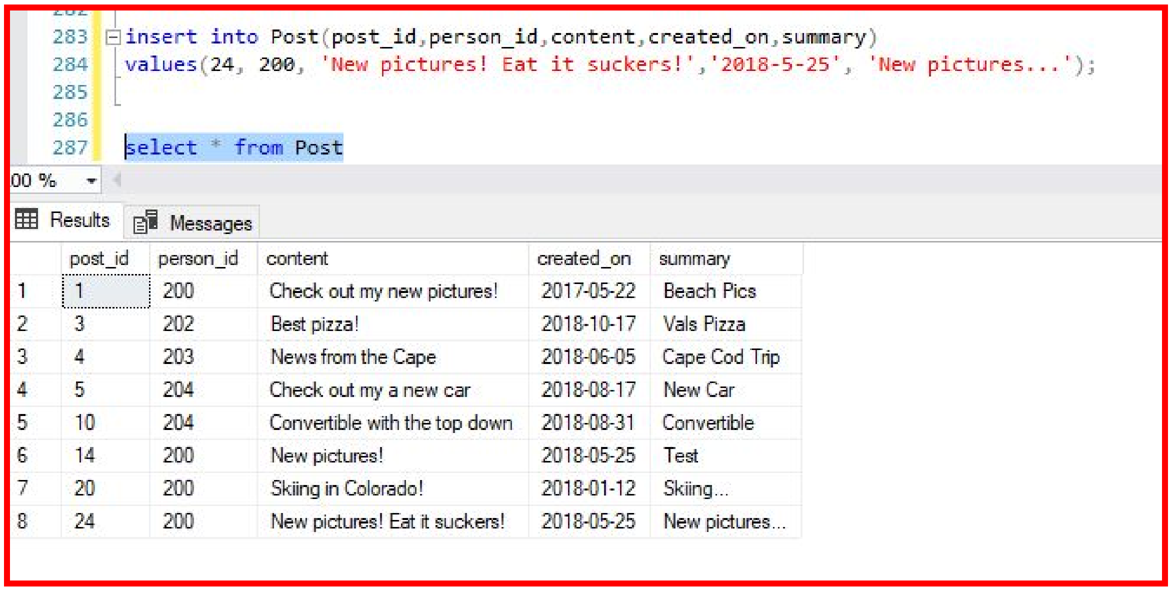
8. One practical use of a trigger is validation within a single table (that is, the validation can be performed by using columns in the table being modified). Create a trigger that validates that the summary is being inserted correctly, that is, that the summary is actually the first 12 characters of 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.



```sql
-- STEP 8 Create a trigger
CREATE or ALTER TRIGGER validate_summary
ON Post AFTER INSERT, UPDATE
AS
BEGIN
    -- Declare then set variables
    DECLARE @SUMMARY VARCHAR(15);
    DECLARE @CONTENT VARCHAR(255);
    SET @SUMMARY = (SELECT INSERTED.summary FROM INSERTED);
    SET @CONTENT = (SELECT INSERTED.content FROM INSERTED);
    --if
    if @SUMMARY != CONCAT(SUBSTRING(@CONTENT,1,12),'...')
    BEGIN
        ROLLBACK;
        RAISERROR('Summary not in correct format',14,1);
    END;
END;

insert into Post(post_id,person_id,content,created_on,summary)
values(14, 200, 'New pictures! Eat it suckers!','2018-5-25', 'Test');
```

```
Msg 50000, Level 14, State 1, Procedure validate_summary, Line 14 [Batch Start Line 282]
Summary not in correct format

(1 row affected)
Msg 3609, Level 16, State 1, Line 283
The transaction ended in the trigger. The batch has been aborted.
```

Trigger catches an error!

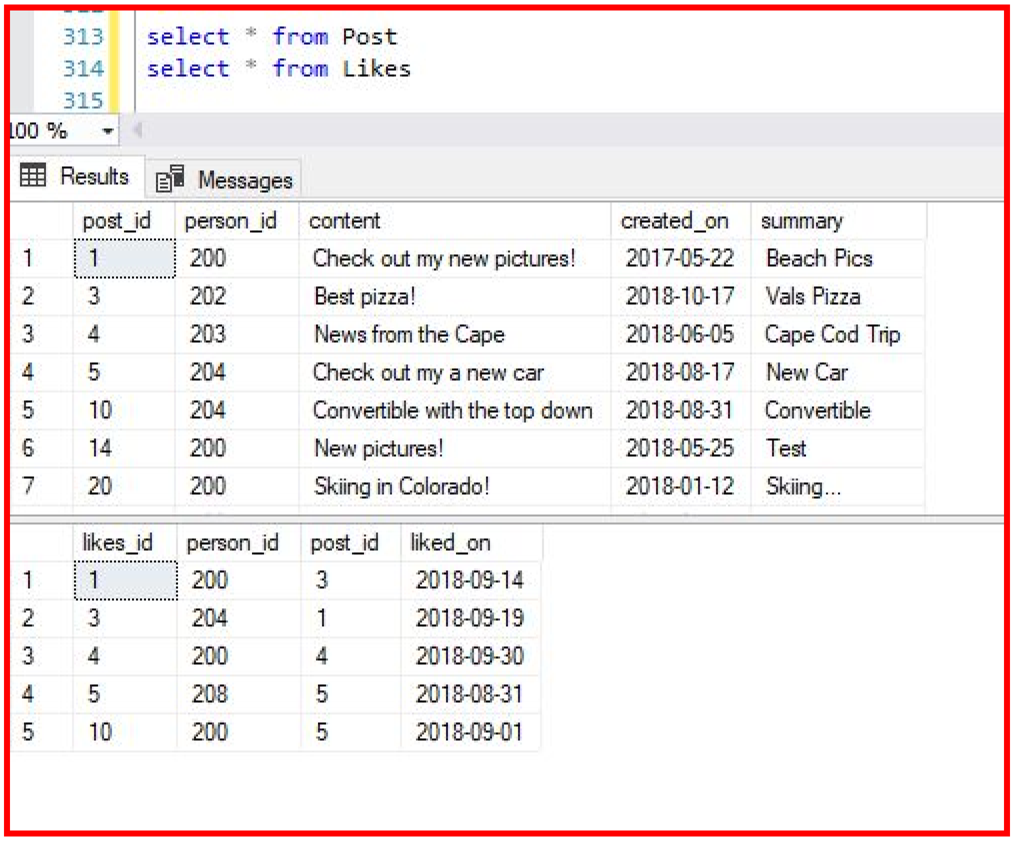


```
insert into Post(post_id,person_id,content,created_on,summary)
values(24, 200, 'New pictures! Eat it suckers!','2018-5-25', 'New pictures...');


select * from Post
```

	post_id	person_id	content	created_on	summary
1	1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	3	202	Best pizza!	2018-10-17	Vals Pizza
3	4	203	News from the Cape	2018-06-05	Cape Cod Trip
4	5	204	Check out my a new car	2018-08-17	New Car
5	10	204	Convertible with the top down	2018-08-31	Convertible
6	14	200	New pictures!	2018-05-25	Test
7	20	200	Skiing in Colorado!	2018-01-12	Skiing...
8	24	200	New pictures! Eat it suckers!	2018-05-25	New pictures...

Trigger passes post_id 24 with correct summary

9. 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.

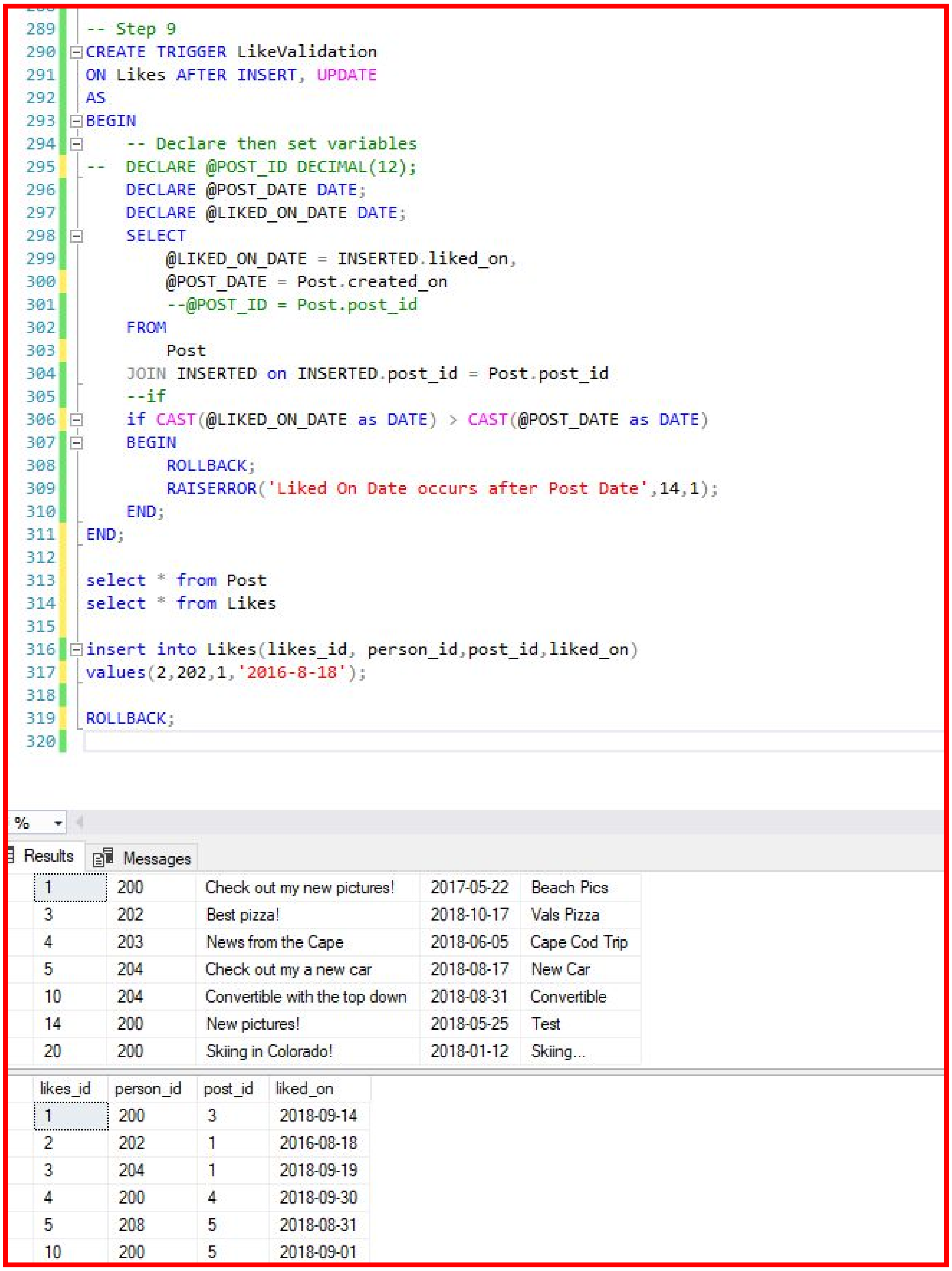


```
select * from Post
select * from Likes
```

	post_id	person_id	content	created_on	summary
1	1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	3	202	Best pizza!	2018-10-17	Vals Pizza
3	4	203	News from the Cape	2018-06-05	Cape Cod Trip
4	5	204	Check out my a new car	2018-08-17	New Car
5	10	204	Convertible with the top down	2018-08-31	Convertible
6	14	200	New pictures!	2018-05-25	Test
7	20	200	Skiing in Colorado!	2018-01-12	Skiing...

	likes_id	person_id	post_id	liked_on
1	1	200	3	2018-09-14
2	3	204	1	2018-09-19
3	4	200	4	2018-09-30
4	5	208	5	2018-08-31
5	10	200	5	2018-09-01

START



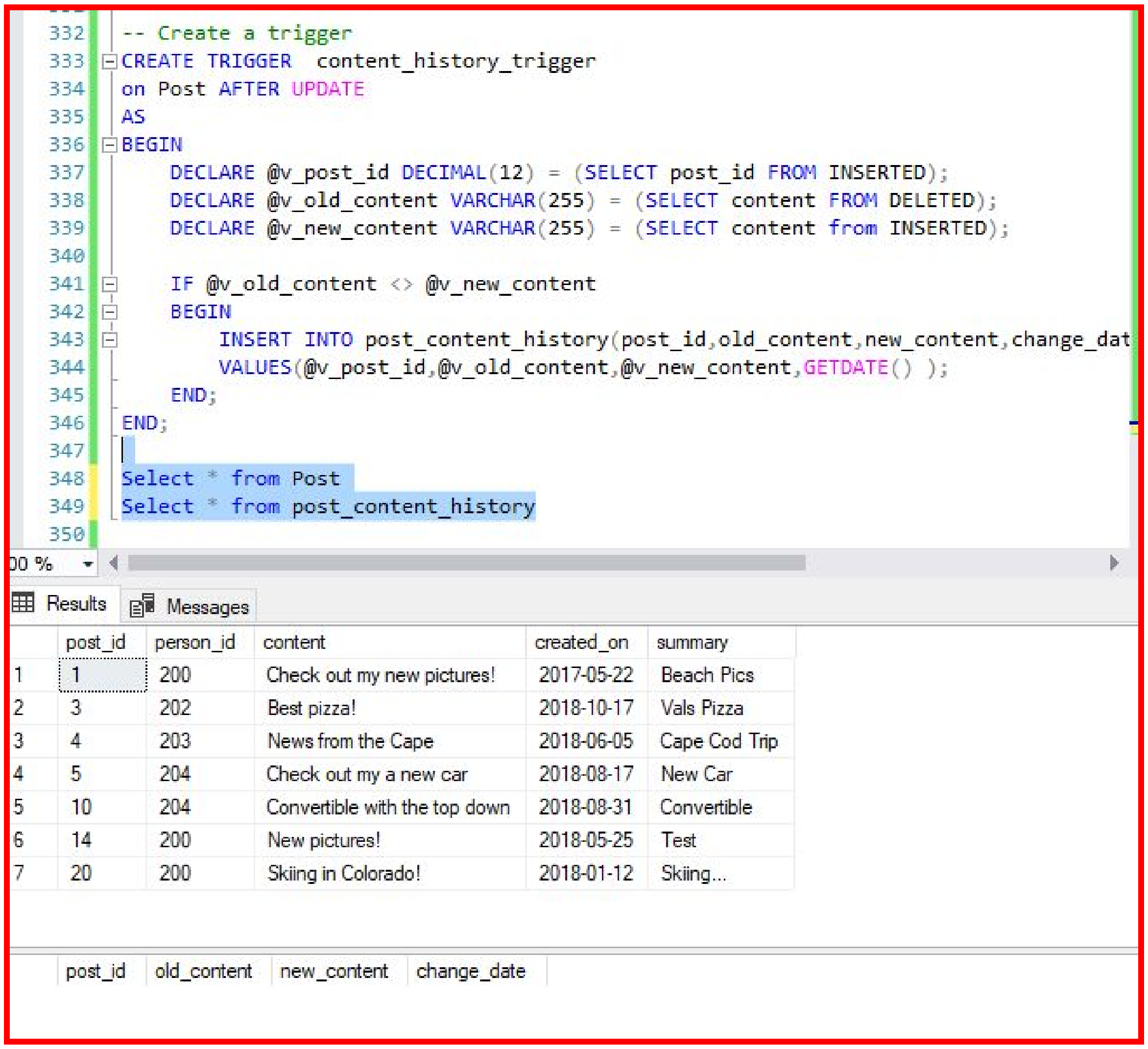
```
-- Step 9
CREATE TRIGGER LikeValidation
ON Likes AFTER INSERT, UPDATE
AS
BEGIN
    -- Declare then set variables
--  DECLARE @POST_ID DECIMAL(12);
    DECLARE @POST_DATE DATE;
    DECLARE @LIKED_ON_DATE DATE;
    SELECT
        @LIKED_ON_DATE = INSERTED.liked_on,
        @POST_DATE = Post.created_on
        --@POST_ID = Post.post_id
    FROM
        Post
    JOIN INSERTED on INSERTED.post_id = Post.post_id
    --if
    if CAST(@LIKED_ON_DATE as DATE) > CAST(@POST_DATE as DATE)
    BEGIN
        ROLLBACK;
        RAISERROR('Liked On Date occurs after Post Date',14,1);
    END;
END;

select * from Post
select * from Likes

insert into Likes(likes_id, person_id,post_id,liked_on)
values(2,202,1,'2016-8-18');

ROLLBACK;
```

1	200	Check out my new pictures!	2017-05-22	Beach Pics
3	202	Best pizza!	2018-10-17	Vals Pizza
4	203	News from the Cape	2018-06-05	Cape Cod Trip
5	204	Check out my a new car	2018-08-17	New Car
10	204	Convertible with the top down	2018-08-31	Convertible
14	200	New pictures!	2018-05-25	Test
20	200	Skiing in Colorado!	2018-01-12	Skiing...

likes_id	person_id	post_id	liked_on
1	200	3	2018-09-14
2	202	1	2016-08-18
3	204	1	2018-09-19
4	200	4	2018-09-30
5	208	5	2018-08-31
10	200	5	2018-09-01

Could not get TRIGGER to activate on error.
'Like Date' for Post 1 should not occur before 2017, but I was able to insert the like.
Must be a way to debug and step through these procedures and triggers.

10. 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).

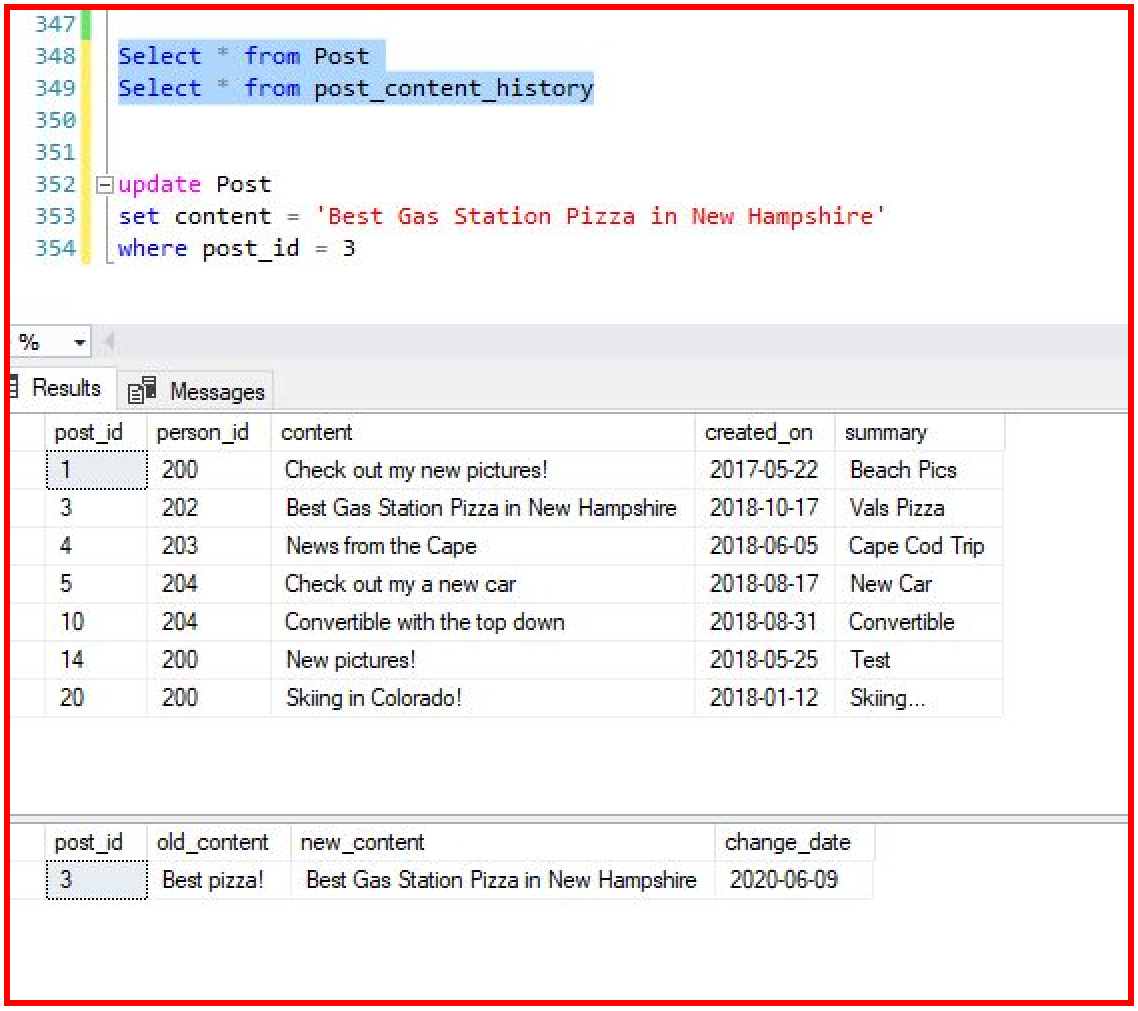


```
-- Create a trigger
CREATE TRIGGER  content_history_trigger
on Post AFTER UPDATE
AS
BEGIN
    DECLARE @v_post_id DECIMAL(12) = (SELECT post_id FROM INSERTED);
    DECLARE @v_old_content VARCHAR(255) = (SELECT content FROM DELETED);
    DECLARE @v_new_content VARCHAR(255) = (SELECT content from INSERTED);

    IF @v_old_content <> @v_new_content
    BEGIN
        INSERT INTO post_content_history(post_id,old_content,new_content,change_dat
        VALUES(@v_post_id,@v_old_content,@v_new_content,GETDATE() );
    END;
END;

Select * from Post
Select * from post_content_history
```

	post_id	person_id	content	created_on	summary
1	1	200	Check out my new pictures!	2017-05-22	Beach Pics
2	3	202	Best pizza!	2018-10-17	Vals Pizza
3	4	203	News from the Cape	2018-06-05	Cape Cod Trip
4	5	204	Check out my a new car	2018-08-17	New Car
5	10	204	Convertible with the top down	2018-08-31	Convertible
6	14	200	New pictures!	2018-05-25	Test
7	20	200	Skiing in Colorado!	2018-01-12	Skiing...

post_id	old_content	new_content	change_date

No Changes to Post table - No entries in history table.



```sql
Select * from Post
Select * from post_content_history


update Post
set content = 'Best Gas Station Pizza in New Hampshire'
where post_id = 3
```

Results | Messages

post_id	person_id	content	created_on	summary
1	200	Check out my new pictures!	2017-05-22	Beach Pics
3	202	Best Gas Station Pizza in New Hampshire	2018-10-17	Vals Pizza
4	203	News from the Cape	2018-06-05	Cape Cod Trip
5	204	Check out my a new car	2018-08-17	New Car
10	204	Convertible with the top down	2018-08-31	Convertible
14	200	New pictures!	2018-05-25	Test
20	200	Skiing in Colorado!	2018-01-12	Skiing...

post_id	old_content	new_content	change_date
3	Best pizza!	Best Gas Station Pizza in New Hampshire	2020-06-09

Trigger successfully captured a change in the Post Table.

Evaluation

Your lab will be reviewed by your facilitator or instructor with the following criteria and grade breakdown.

Criterion	A	B	C	D	F	Letter Grade
Correctness and Completeness of Results (70%)	All steps' results are entirely complete and correct	About ¾ of the steps' results are correct and complete	About half of the steps' results are correct and complete	About ¼ of the steps' results are correct and complete	Virtually none of the step's results are correct and complete	
Constitution of SQL and Explanations (30%)	Excellent use and integration of appropriate SQL constructs and supporting explanations	Good use and integration of appropriate SQL constructs and supporting explanations	Mediocre use and integration of appropriate SQL constructs and supporting explanations	Substandard use and integration of appropriate SQL constructs and supporting explanations	Virtually all SQL constructs and supporting explanations are unsuitable or improperly integrated	
					Assignment Grade:	#N/A
The resulting grade is calculated as a weighted average as listed using A+=100, A=96, A-=92, B+=88, B=85, B-=82 etc.						
To obtain an A grade for the course, your weighted average should be >=95, A- >=90, B+>=87, B >= 82, B- >= 80 etc.						

Use the **Ask the Facilitators 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.