|  |
| --- |
| **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 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.

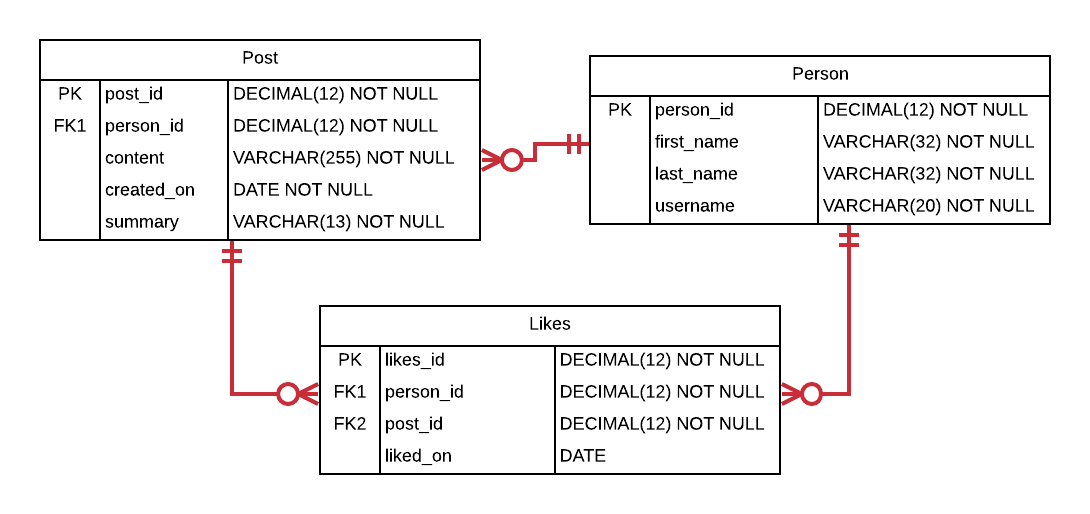
It is important for us to avoid this confusion by recognizing that there are two distinct languages within a relational DBMS – declarative and procedural – and that both are treated very differently within a DBMS in concept and in implementation. In concept, we use the SQL declarative language to tell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

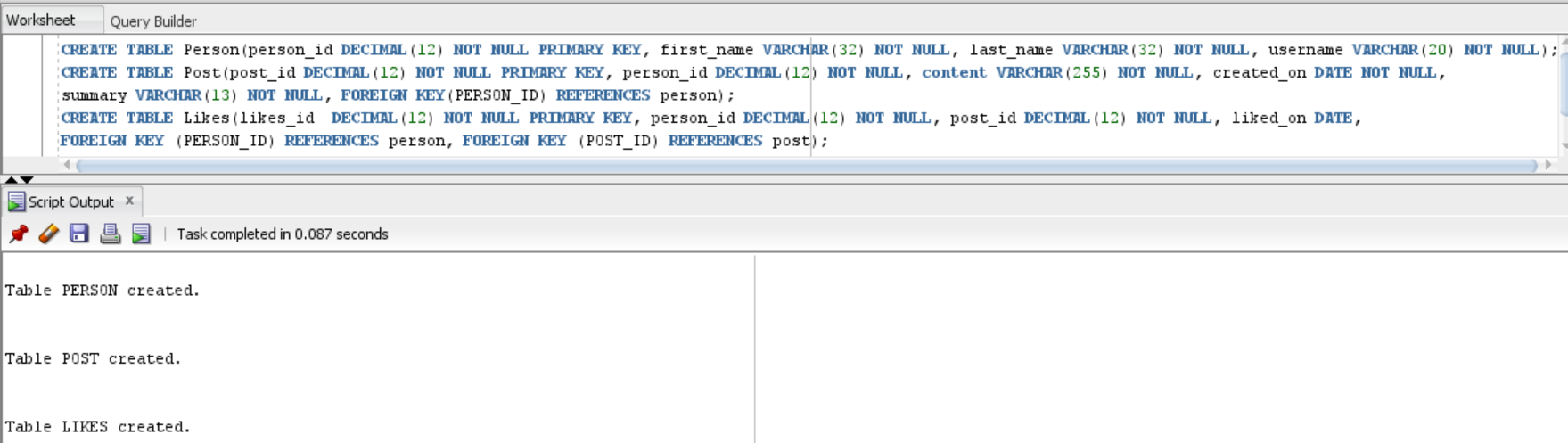
In this first section, you will work with stored procedures on this schema, which offer many significant benefits. Reusability is one significant benefit. The logic contained in a stored procedure can be executed repeatedly, so that each developer need not reinvent the same logic each time it is needed.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

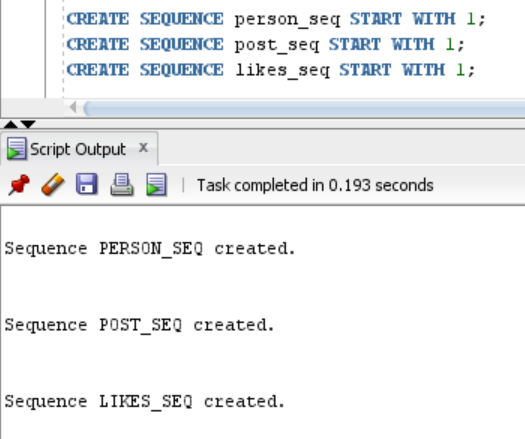
You will also learn to work with sequences in this section, which are the preferred means of generating synthetic primary keys for each of your tables.

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

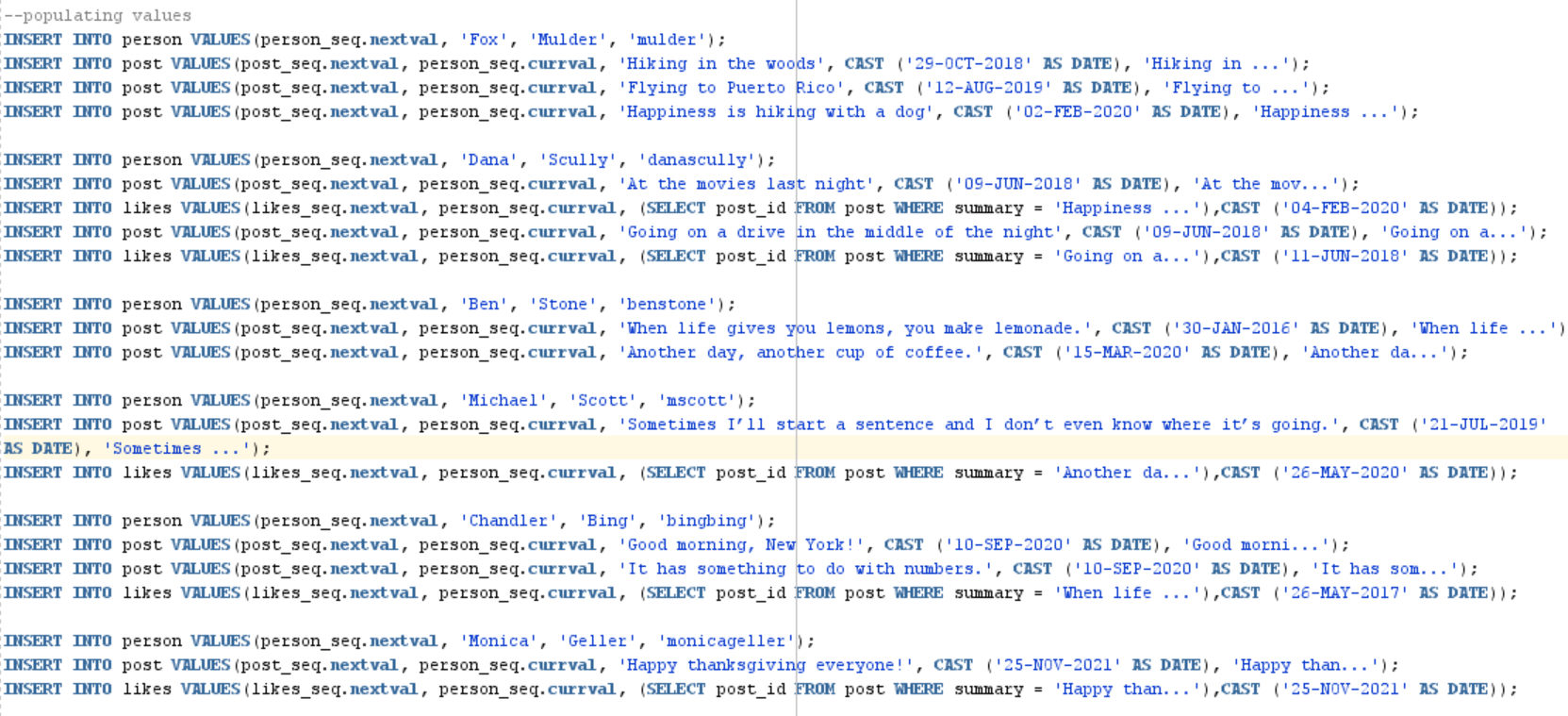
**Section Steps**

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.

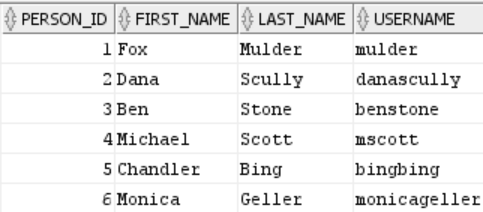




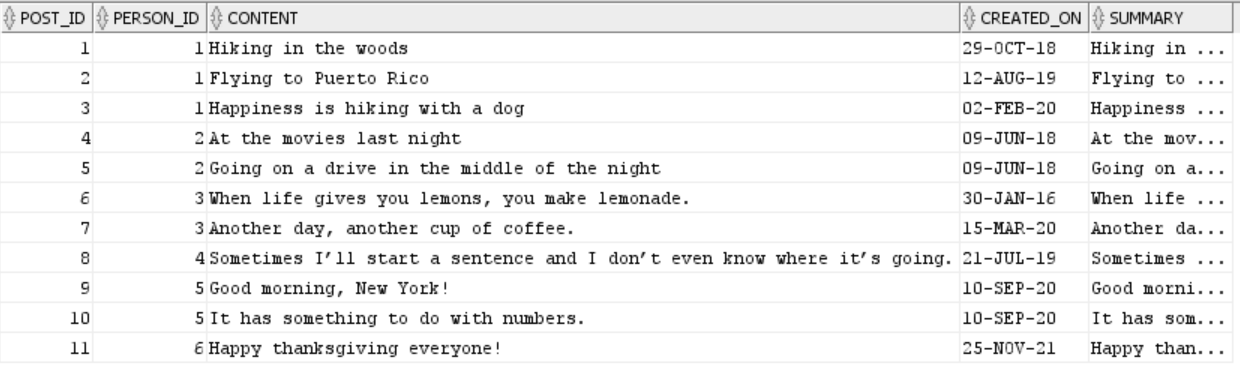
1. *Populate Tables –* Populate the tables with data, ensuring that there are at least 5 people, at least 8 posts, and at least 4 likes. Make sure to use sequences to generate the primary and foreign key values. Most of the fields are self-explanatory. As far as the “content” field in Post, make them 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 “…”.



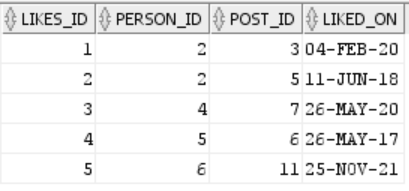
Person table:

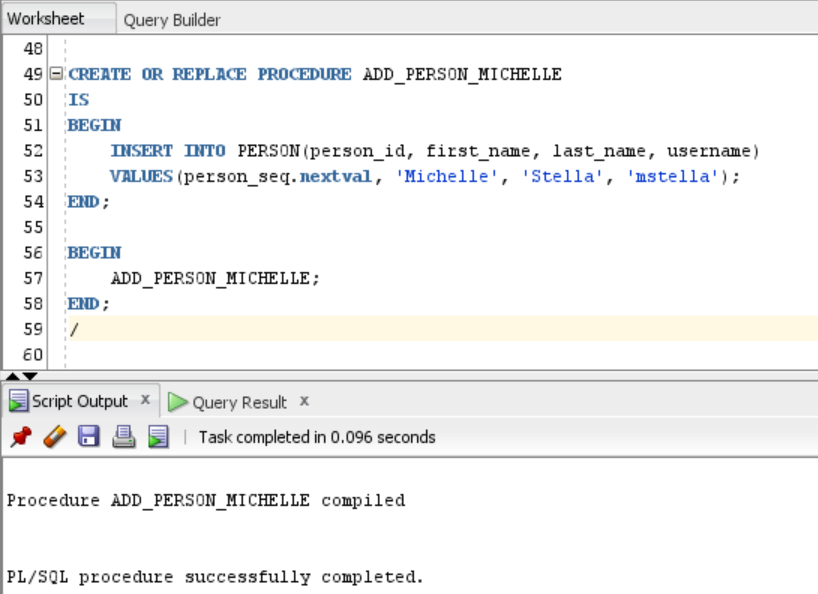


Post table:

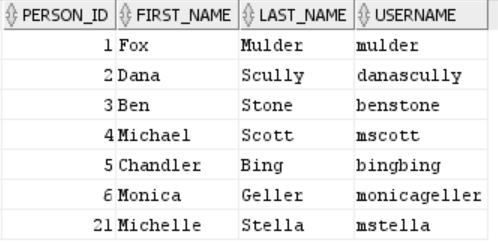


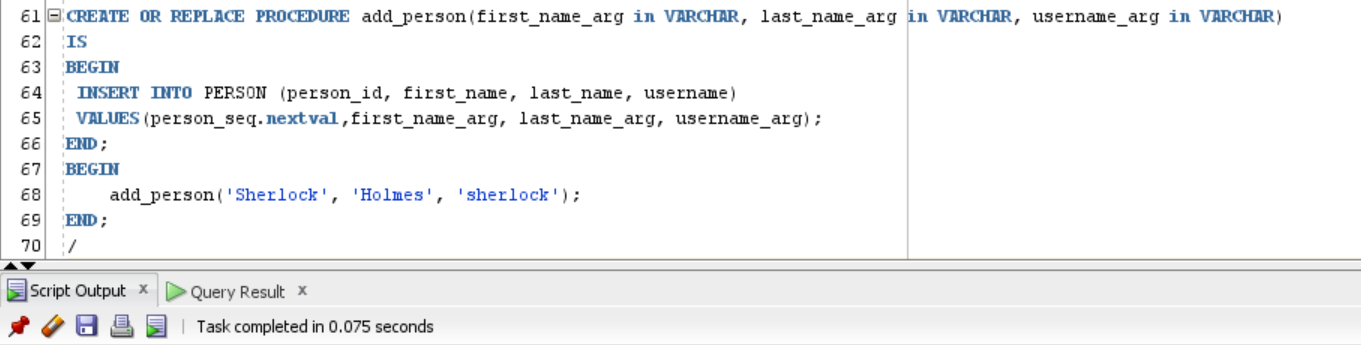
Likes table:



1. *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.  
   

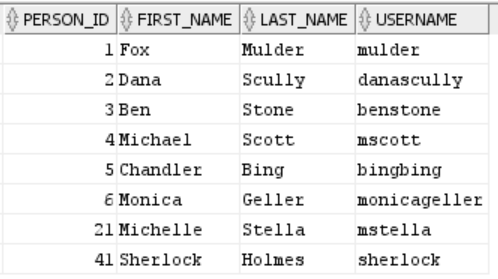
SELECT \* FROM Person; The person “Michelle Stella” has been listed in the last row of the Person table.

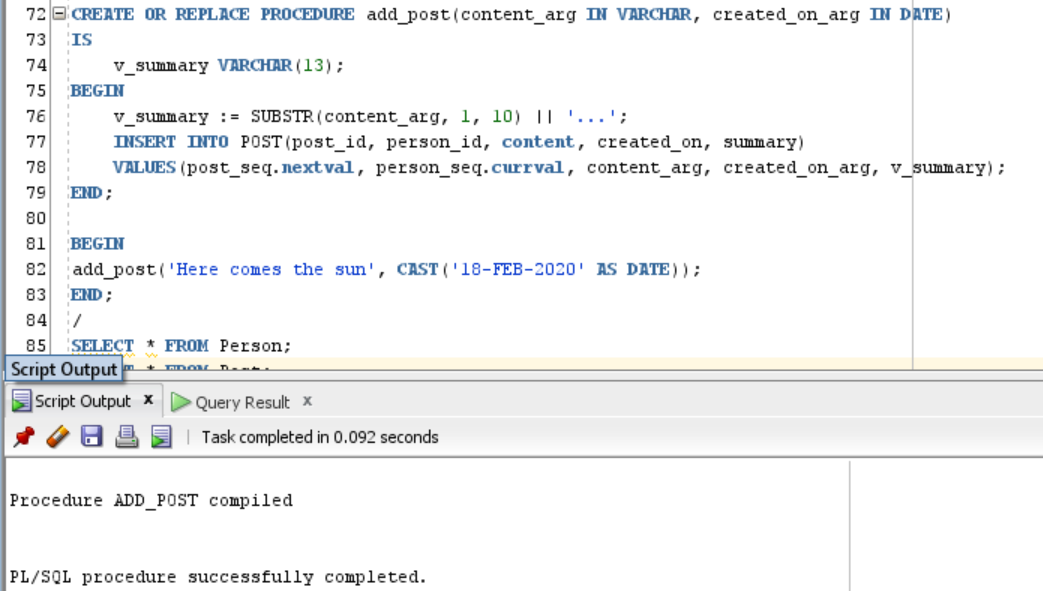


1. *Create Reusable Procedure –* Create a reusable stored procedure named “add\_person” that uses parameters and allows you to insert any new person into the Person table. Execute the stored procedure with a person of your choosing, then list out the Person table to show that the person was added to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

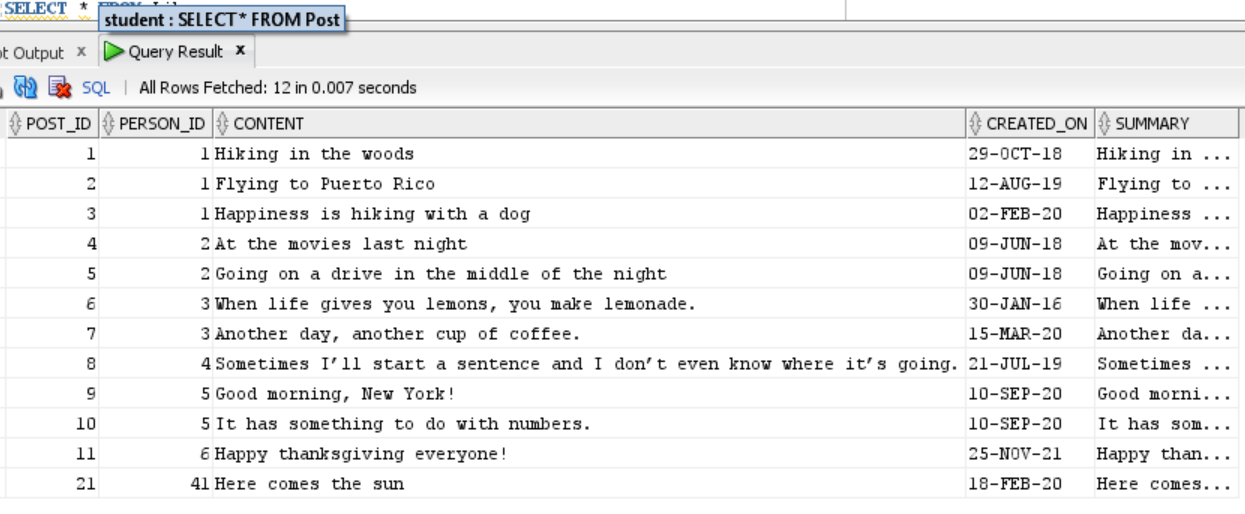


SELECT \* FROM Person; The person “Sherlock Holmes” has been listed in the last row of the Person table.

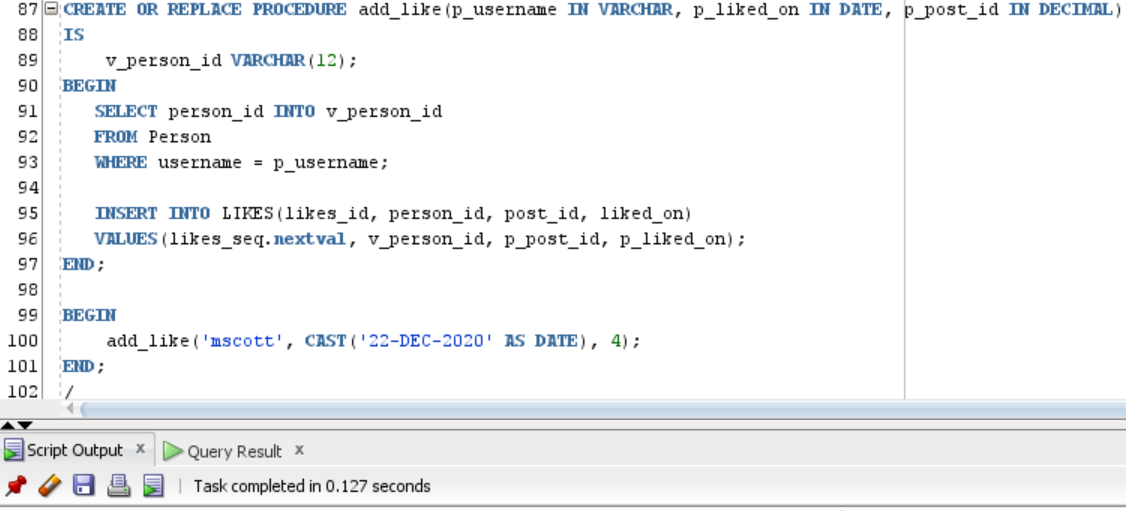


1. *Create Deriving Procedure –* Create a reusable stored procedure named “add\_post” that uses parameters and allows you to insert any new post into the Post table. Instead of passing in the summary as a parameter, derive the summary from the content, storing the derivation temporarily in a variable (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.  
   

The summary “Here comes…” for the content “Here comes the sun” has been derived into the last row of the Post table.

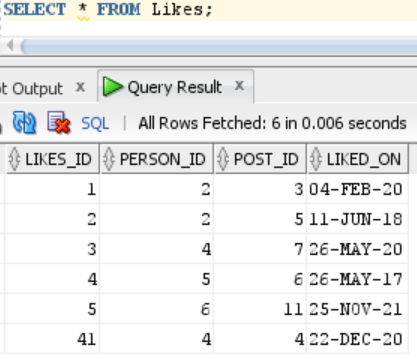


1. *Create Lookup Procedure –* Create a reusable stored procedure named “add\_like” that uses parameters and allows you to insert any new “like”. Rather than passing in the person\_id value as a parameter to identify which person is liking which post, pass in the username of the person. The stored procedure 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.





The add\_like procedure was invoked referencing “mscott” username, post\_id = 4, and liked\_on date as 22-DEC-2020. This has been inserted as the last row in the Likes table with likes\_id = 44.



The username “mscott” is for the person named Michael Scott whose person\_id = 4. The user liked post\_id = 4 which is a post with content reading “At the movies last night.”

**Section Two – Triggers**

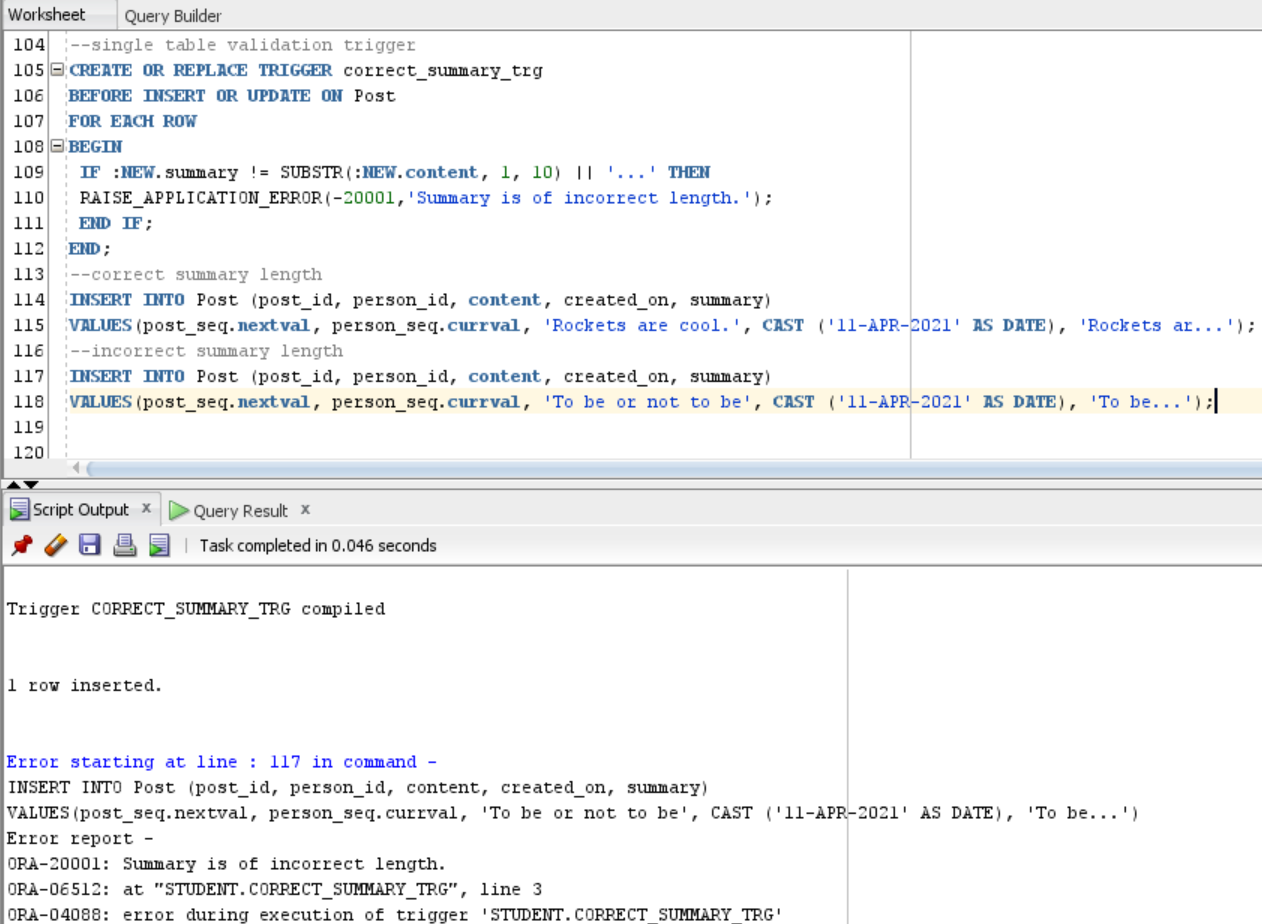
**Section Background**

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.

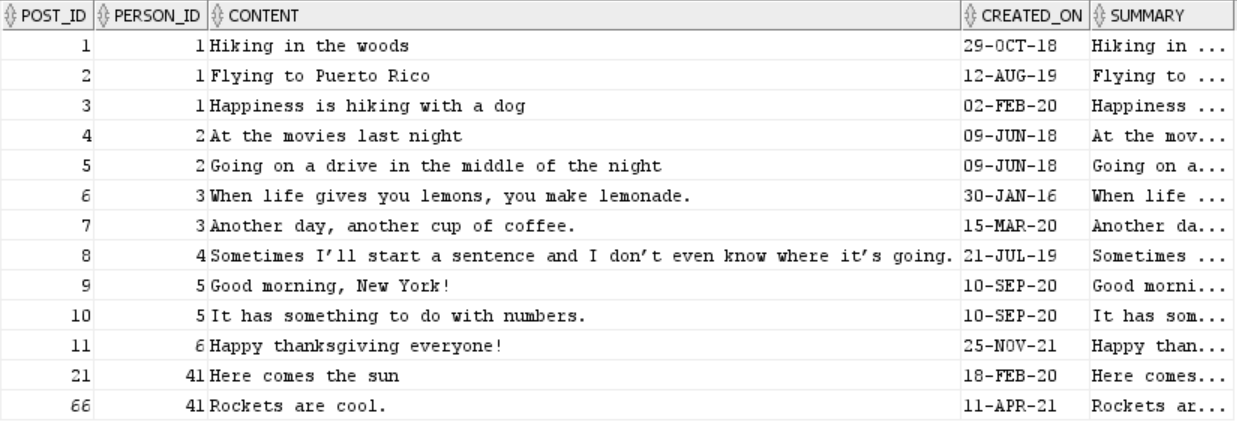
Triggers also have significant drawbacks. By default triggers execute within the same transaction as the event that caused the trigger to execute, and so any failure of the trigger results in the abortion of the overall transaction. Triggers execute additional code beyond the regular processing of the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

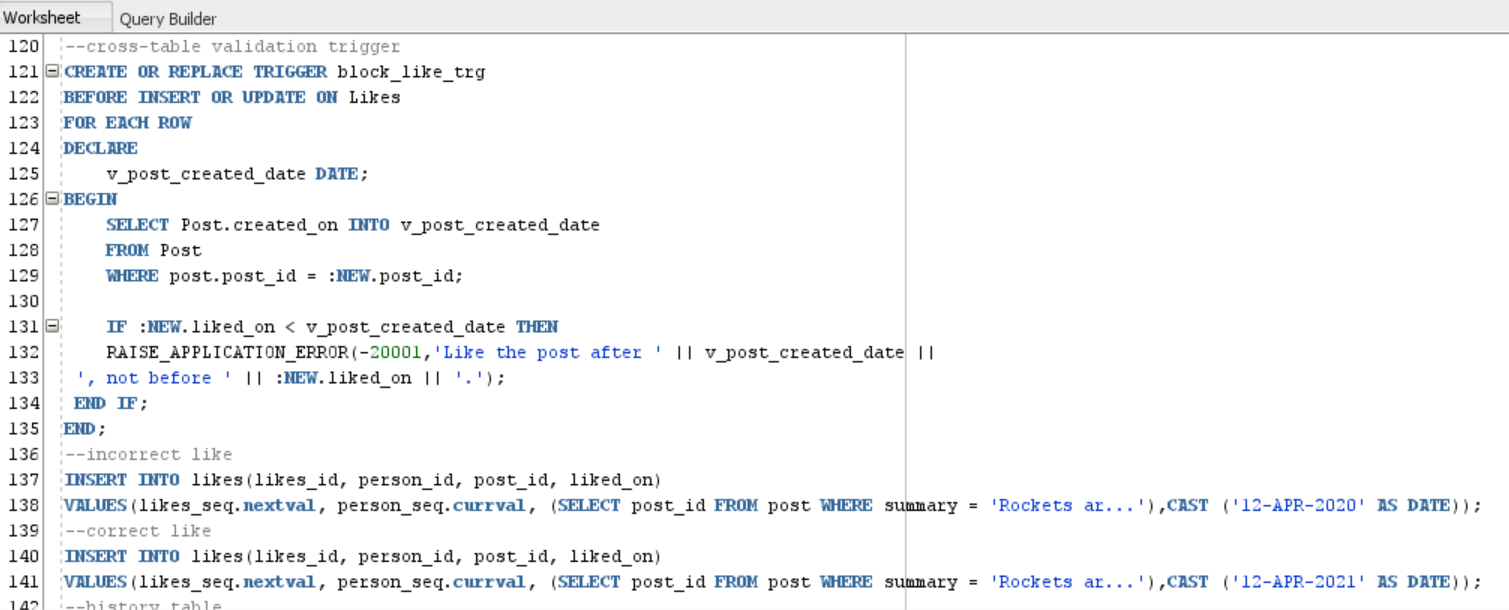
**Section Steps**

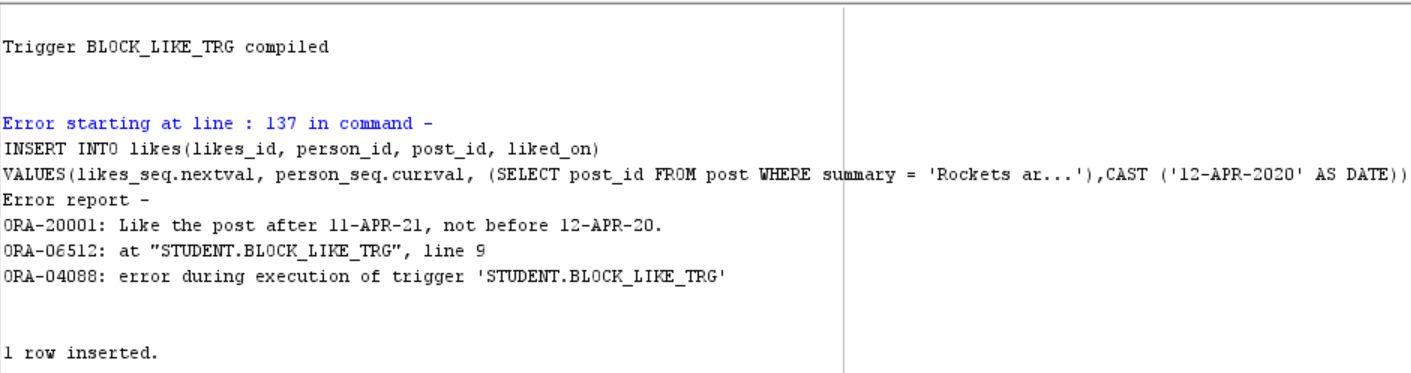
1. *Single Table Validation Trigger –* One practical use of a trigger is validation within a single table (that is, the validation can be performed by using columns in the table being modified). Create a trigger that validates that the summary is being inserted correctly, that is, that the summary is 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.  
   

The INSERT INTO statement that has summary of correct length has been inserted in the last row of the Post table and the statement that has incorrect summary length has not been added as it was rejected.



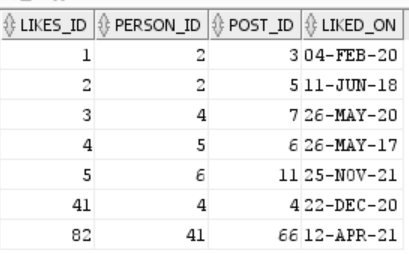
1. *Cross-Table Validation Trigger –* Another practical use of a trigger is cross-table validation (that is, the validation needs columns from at least one table external to the table being updated). Create a trigger that blocks a “like” from being inserted if its “liked\_on” date is before the post’s “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.



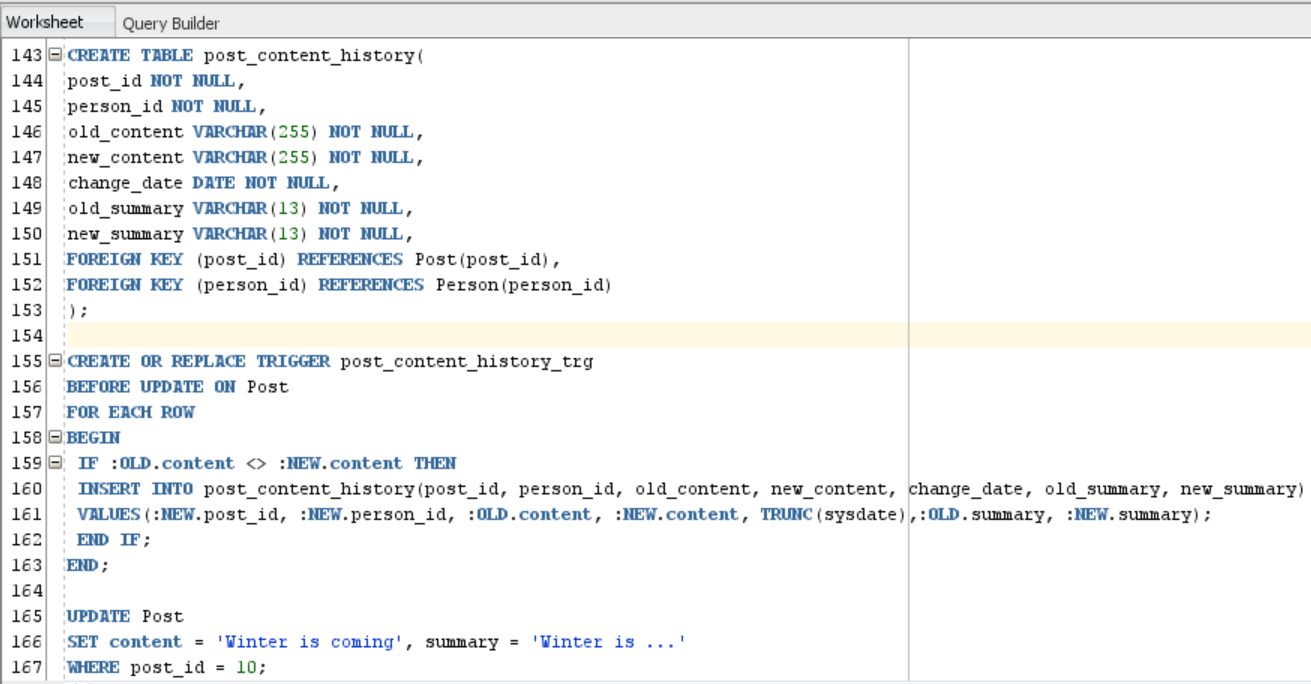


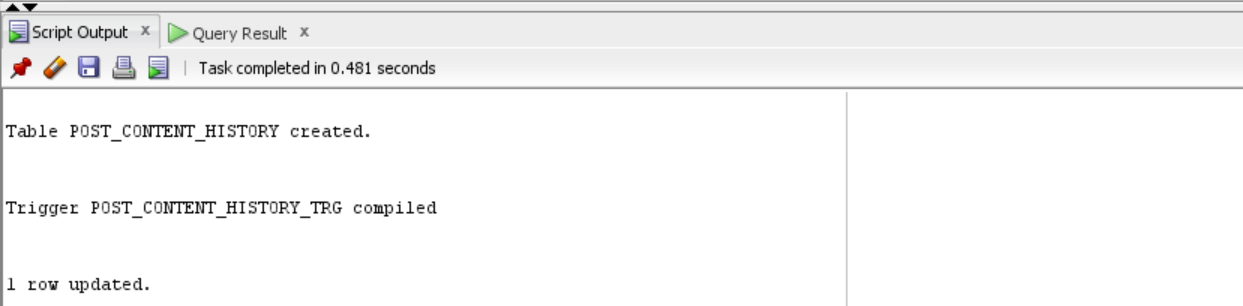
The first INSERT INTO statement tries to insert a like on 12-APR-**2020** for the post with summary “Rockets ar…”, but this post’s created\_on date is a year ahead i.e., 11-APR-**2021**. Hence, the error report “ORA-20001: Like the post after 11-APR-21, not before 12-APR-20.”

The second INSERT INTO statement tries to insert a like on 12-APR-2021, which is in the same year as the post’s created\_on date and on the next day. This is a valid insertion, so only this statement has been inserted as the last row in the Likes table.

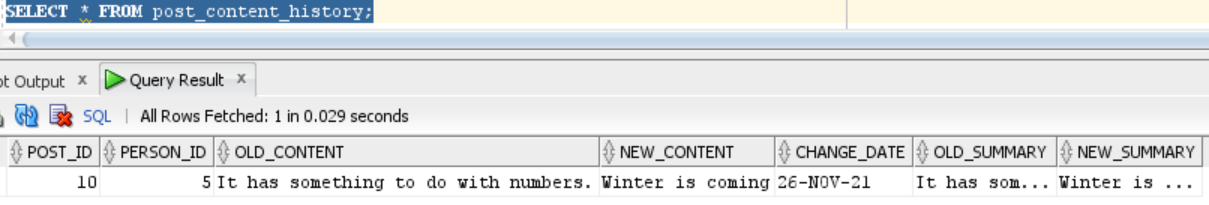


1. *History Trigger –* Another practical use of trigger is to maintain a history of values as they change. Create a table named post\_content\_history that is used to record updates to the content of a post, then create a trigger that keeps this table up-to-date when updates happen to post contents. 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).





The history table:



**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**

1. *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.

1. 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…  
     
   Make sure to explain your reasoning for the functional dependency choices.

case\_number 🡺 case\_description

case\_number 🡺 plaintiff\_first\_name, plaintiff\_last\_name, defendant\_first\_name, defendant\_last\_name

case-number, appearance date 🡺 decision1\_description, decision2\_description, attorney1\_first\_name, attorney1\_last\_name, attorney2\_first\_name, attorney2\_last\_name, attorney3\_first\_name,

attorney3\_last\_name, number\_attending, extra\_appearance\_notes

1. 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.

The case\_number will act as a primary key that uniquely identifies case\_description, plaintiff, and defendant. A defendant is also identified by their plaintiff, so it is better to branch them into one table.

The case\_number and the appearance\_date together identify decisions, number of people attending, attorneys, and any extra appearance notes. A case number may have more than one attorney, decision, and different number of people attending the same case on each appearance date. There can be a connection between the Attorney table and the Decision table and a connection between the Attorney table and the Appearance table.

Diagram

Description automatically generated

# 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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.