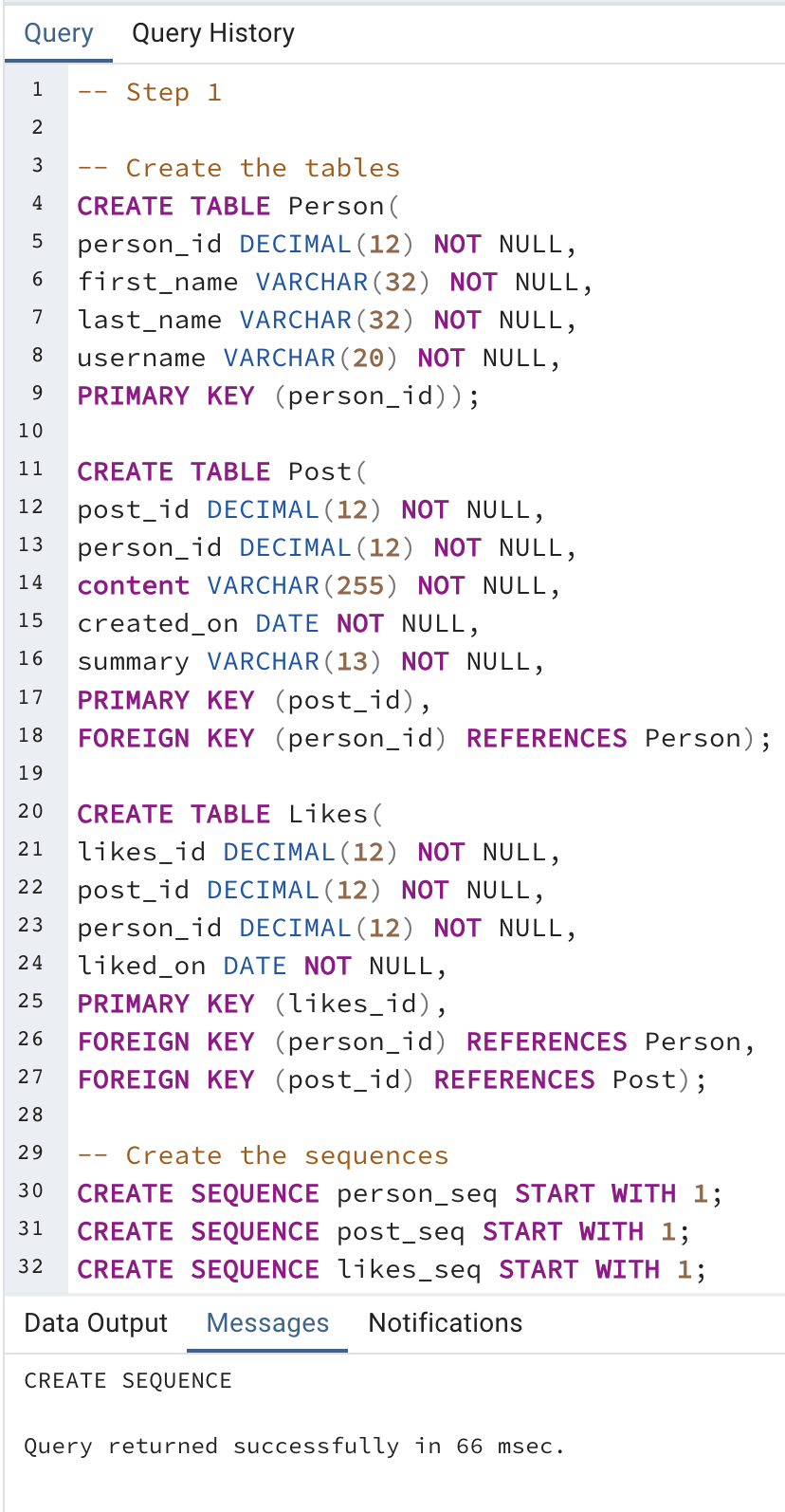
***Section One – Stored Procedures***

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 “…”.

A screen shot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

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.

A screenshot of a computer code

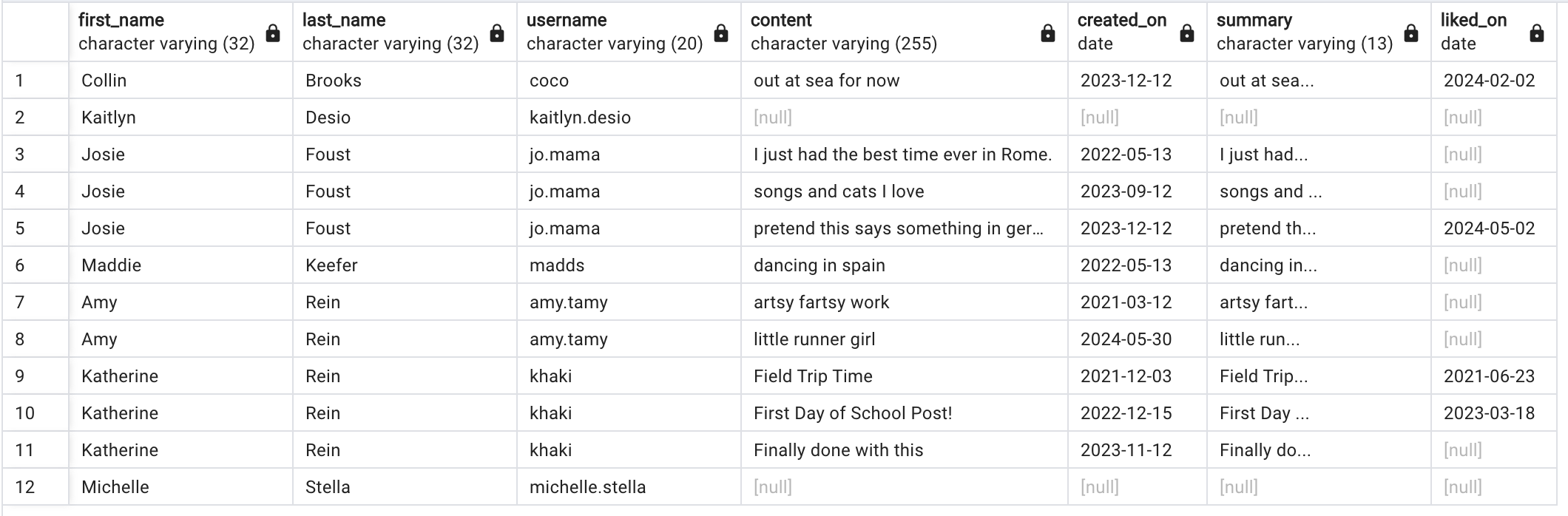
Description automatically generated

A screenshot of a computer

Description automatically generated

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.





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.

A screenshot of a computer

Description automatically generated

***Section Two – Triggers***

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.

A screenshot of a computer

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

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).

A screenshot of a computer program

Description automatically generated

***Section Three – Normalization***

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….
2. 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

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

case\_number 🡺 courtID

case\_number is a unique identifier. This means that for each value where there can only be one per case we have a functional dependency. There can only ever be one case description, plaintiff, defendant, and court per case. Therefore, the unique identifier for the case creates a functional dependency between those values and the case\_number.

case\_number, appearance\_date 🡺 number\_attending, extra\_appearance\_notes

case\_number, appearance\_date 🡺 attorney1\_first\_name, attorney1\_last\_name

case\_number, appearance\_date 🡺 attorney2\_first\_name, attorney2\_last\_name

case\_number, appearance\_date 🡺 attorney3\_first\_name, attorney3\_last\_name

case\_number, appearance\_date 🡺 decision1\_description, decision2\_description

Since each case only can be heard once per day, we can find out all of the information about a certain case appearance if given the case\_number and appearance\_date.

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.

A diagram of a system

Description automatically generated

I chose to create a court table to generalize this for many courts if needed. The case\_number is what ties pretty much everything together. The court\_appearanceID is essentially case\_number and appearance\_date combined. This is how decision gets added to the main table. I made each decision need its own decision\_id so that I could diagram it correctly. I did the same thing with the attorneys. Each table that is functionally dependent upon the case\_number has its own table and with the case\_number as a foreign key. The court\_appearance and decision tables are representing their own partial dependencies with case\_number and appearance\_date. This initial set up does not have any transitive dependencies or non-candidate determinants.