Final Project 7-1: Databases Artifact

CS-499-Q5048 20EW5

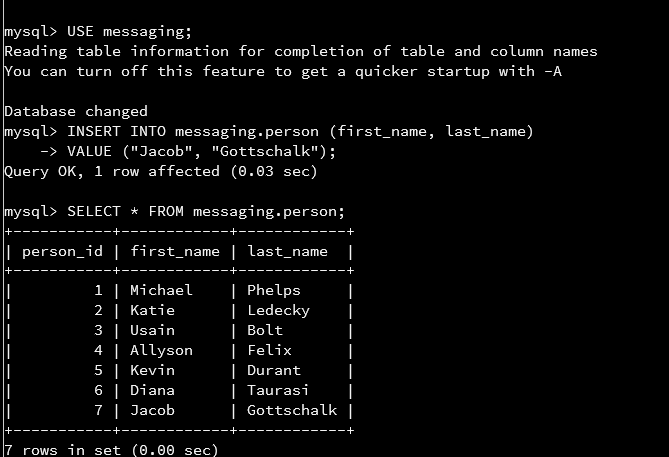
*Original Assingment Name:*

*Final Project 7-1: Database Management Report*

*DAD-220-Q4567 18EW4*

Jacob Gottschalk

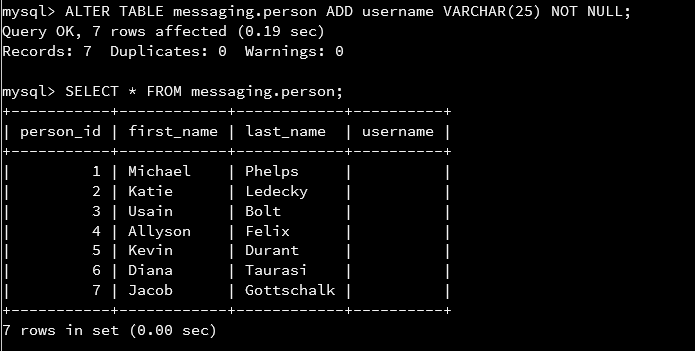
Task 1



Throughout this project, I will be working with tables and queries belonging to the messaging database. In task one I started by accessing the messaging database with the USE messaging; SQL statement. As seen by the Database changed statement returned, I have successfully accessed the messaging database.

As directed by task 1, I added a new record to this table with the use of the INSERT INTO prompt. This prompt is followed by the table name (messaging.person) in which the information is to be entered. Following the table name in parentheses I’ve listed the fields in which I will pass a value into. In this case, the person\_id field is automatically incremented for each new record being entered, so I did not have to specify that field name which means that I only have to specify the first\_name and last\_name fields. The constructor VALUE indicates the values within the parenthesis following are to be entered in their respective fields. Since first\_name and last\_name have the datatype of VARCHAR, quotation marks surround each value to be entered. The statement ends with a semicolon to indicate that the SQL command is complete.

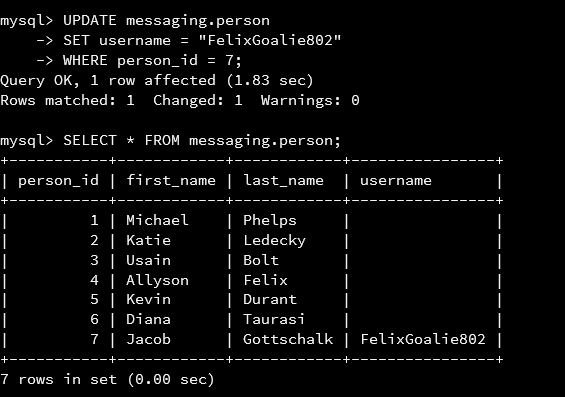
Task 2



Using the ALTER TABLE command, I’m changing the structure of the table in some way. The table in which I’m altering is the messaging.person table, and I’m adding the column username to the table. This column will have a VARCHAR datatype with a length of 25 characters, meaning that a string of letters, numbers, or some symbols can be inputted into the table as long as they do not extend past 25 characters. If a value is entered into this field longer than 25 characters, the value will truncate (cut off a not store) values after the 25 figures. NOT NULL indicates that this field cannot be left empty when inserting new records into the table in the future (but an existing value will have this field empty until it is updated).

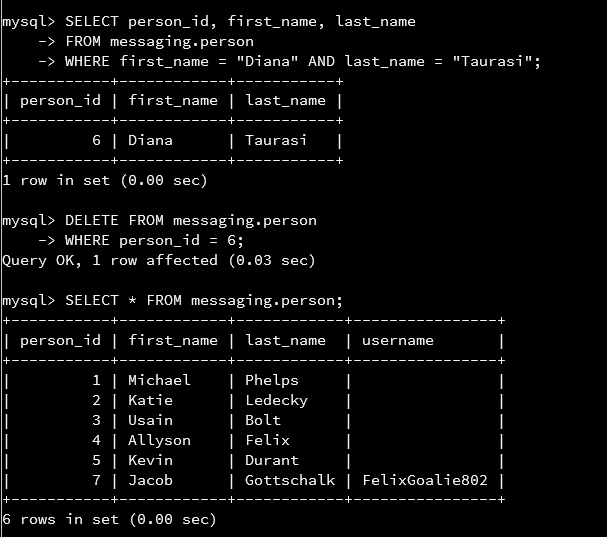
SELECT \* FROM shows all the data stored in each field for the messaging.person table.

Task 3



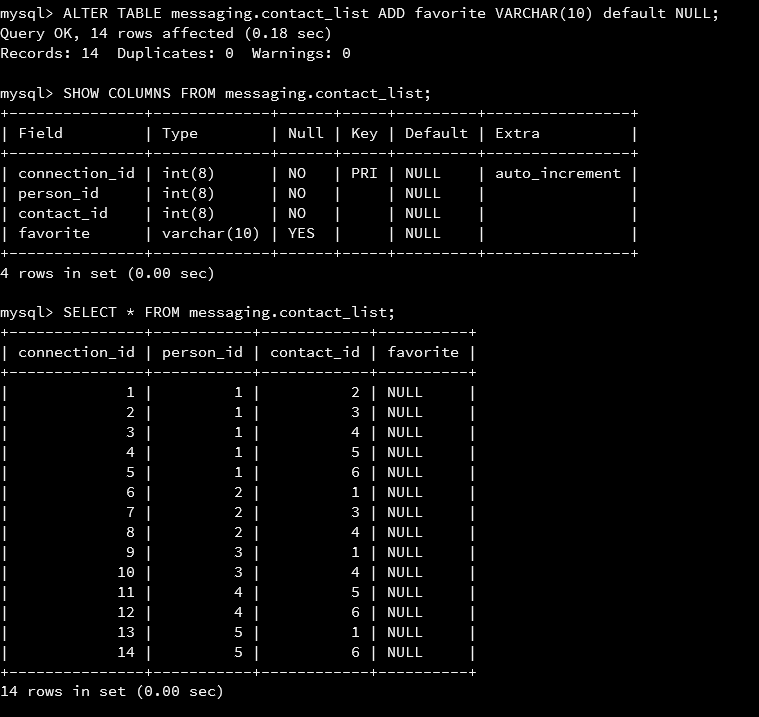
Task 3 required me to update the *messaging.person* table to add a value in the new column created in task 2. The *SET* command tells the system to make the field (in this case *username*) to equal the string *“FelixGoalie802”*. Using the *WHERE* constraint limits the *SET* value to only apply to the record in which *person\_id* is equal to 7. The results are shown in the table following the update statement.

Task 4



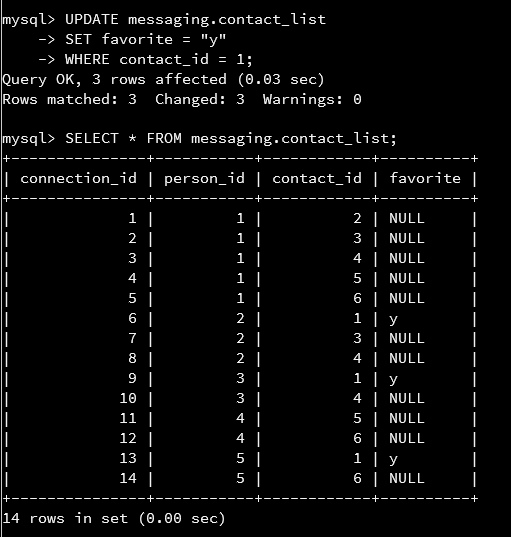
For task 4, I first ran a *SELECT* query to show the *person\_id* with constraints *WHERE* the *first\_name = “Diana” AND last\_name = “Taurasi”*. I did this to setup a *DELETE* statement with one constraint condition. The *DELETE* statement takes the resulting *person\_id* from my *SELECT* query and removes the record from the *person* table in which the *person\_id = 6* (Diana Taurasi). When deleting records, it is important to understand exactly what you are deleting to prevent unintended data loss. Backing up the database is good practice before deleting anything from a database.

Task 5



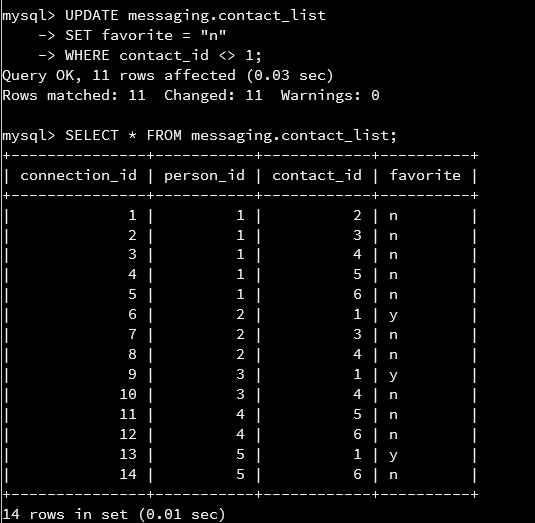
Task 5 is using the *ALTER* command to add a new field called *favorite* to the *messaging.contact\_list* table. A difference between this column being entered into this table versus the earlier alter statement is that the *favorite* filed can be left *NULL* (empty) when records are added to the table. The word *default* in this statement indicate the *NULL* value will appear in this column for any records in which data is not entered.

Task 6



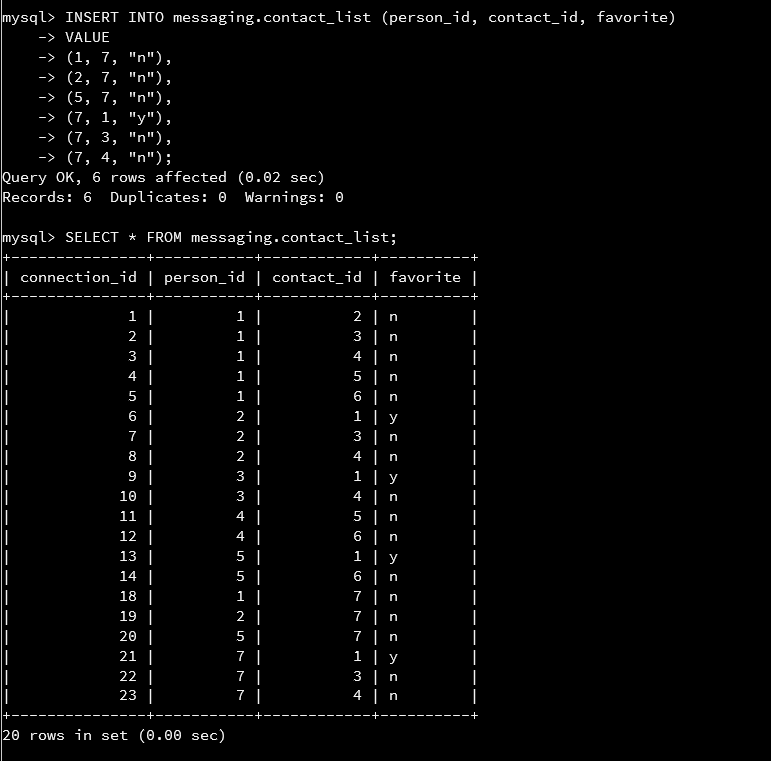
In this task, I am updating all records for the *favorite* column to change the *NULL* value to a “*y*” (for yes) if the *contact\_id* column had a value of “*1*” stored. This *contact\_id* of value “*1*” indicates the Michael Phelps is that person’s favorite contact.

Task 7



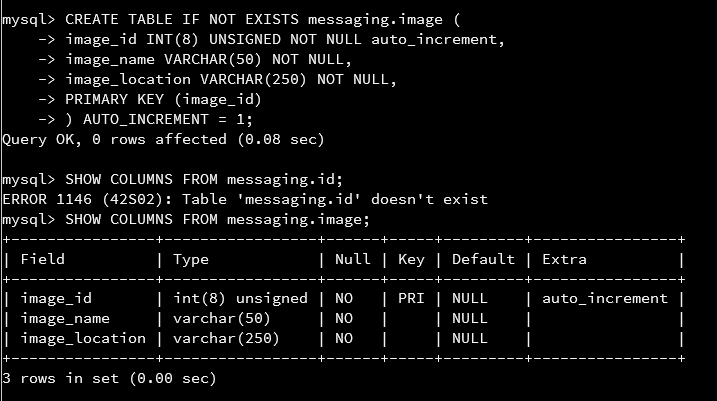
Much like in task 6, I am running an *UPDATE* SQL statement to set all favorite values to “*n*” (for no) when the *contact\_id* in the record does not equal “*1*”. The “< >” symbols indicate the value following is greater-than or less-than the specific value (i.e. *contact\_id* is not equal to “*1”*). The resulting *SELECT* statement now shows that all records for the *favorite* column either have a value of *“y”* or *“n”*.

Task 8



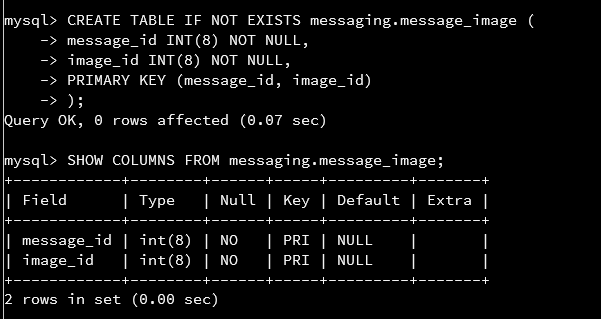
For task 8 I entered six records using the *INSERT INTO* command specifiying to add data values to *person\_id, contact\_id,* and *favorite* columns for the new records. These records are entered in the same fashion as the values entered earleir in this project. As you can see, the record with a *person\_id* of “*7”* and a *contact\_id* of “*1”* has a *favorite* value of “*y”* to follow the data parameters. It should be mentioned that I initally ran an *INSERT* comman and entered 3 new records prior to this, but realized I made a mistake and deleted the records from the table, which explains the gap between *connection\_id 14* and *connection\_id* *18*. I could have used an *UPDATE* statement to fix this mistake, but my *INSERT* statement for this assignment would have been incorrect.

Task 9



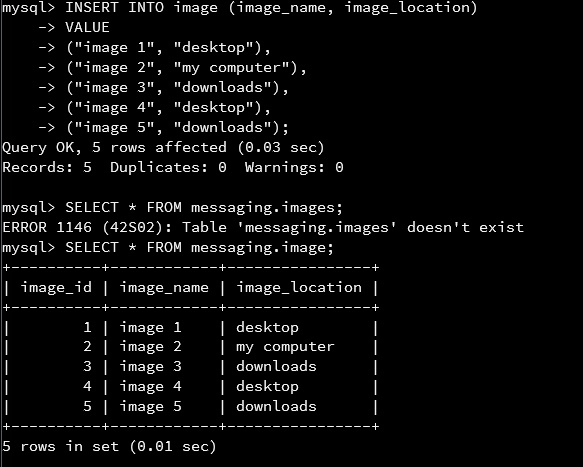
The *CREATE TABLE IF NOT EXISTS* is a way to add a new table to a database without overriding a table that may already exist under this the name entered. If the *IF NOT EXISTS* clause was not present in the statement, the system could write over an existing table with the same name which would delete all of the data stored in that table (including table structure). *Messaging.image* specifies that the new table will be named *image*. Column names and each column’s parameters are specified within the parentesis. *UNSIGNED* is a constraint on the *INT* datatype that only allows valuese that are not negative integers. *AUTO\_INCREMENT* instructs the system to automatically increase the value of that field by 1 over the previous record value in that column for each new record inserted into the table. A *PRIMARY KEY* will be assigned to the field name contained within the parenthesis (in this case *image\_id*). The primary key allows other tables to refer to this field to create relationships between tables in the database for querying. *AUTO\_INCREMENT = 1* indicates that the incrementation increases from value 1.

Task 10

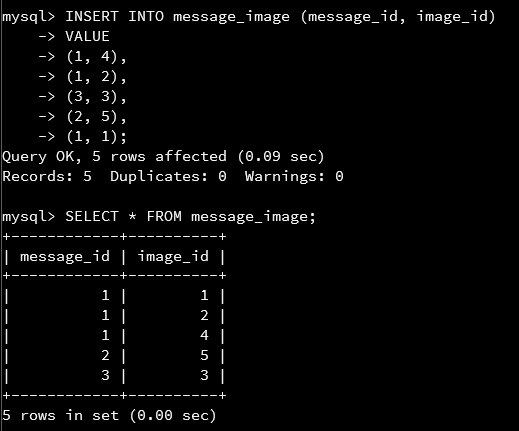


This *CREATE* statement makes a new *messaging.message\_image* table. In this table there are two primary keys which allow it to act as a passthrough table to connect two other tables. All information that can be obtained from this table can be found on other tables, but the connection between the two otherwise unrelated tables (*image* and *message*) are illustrated on this table.

Task 11

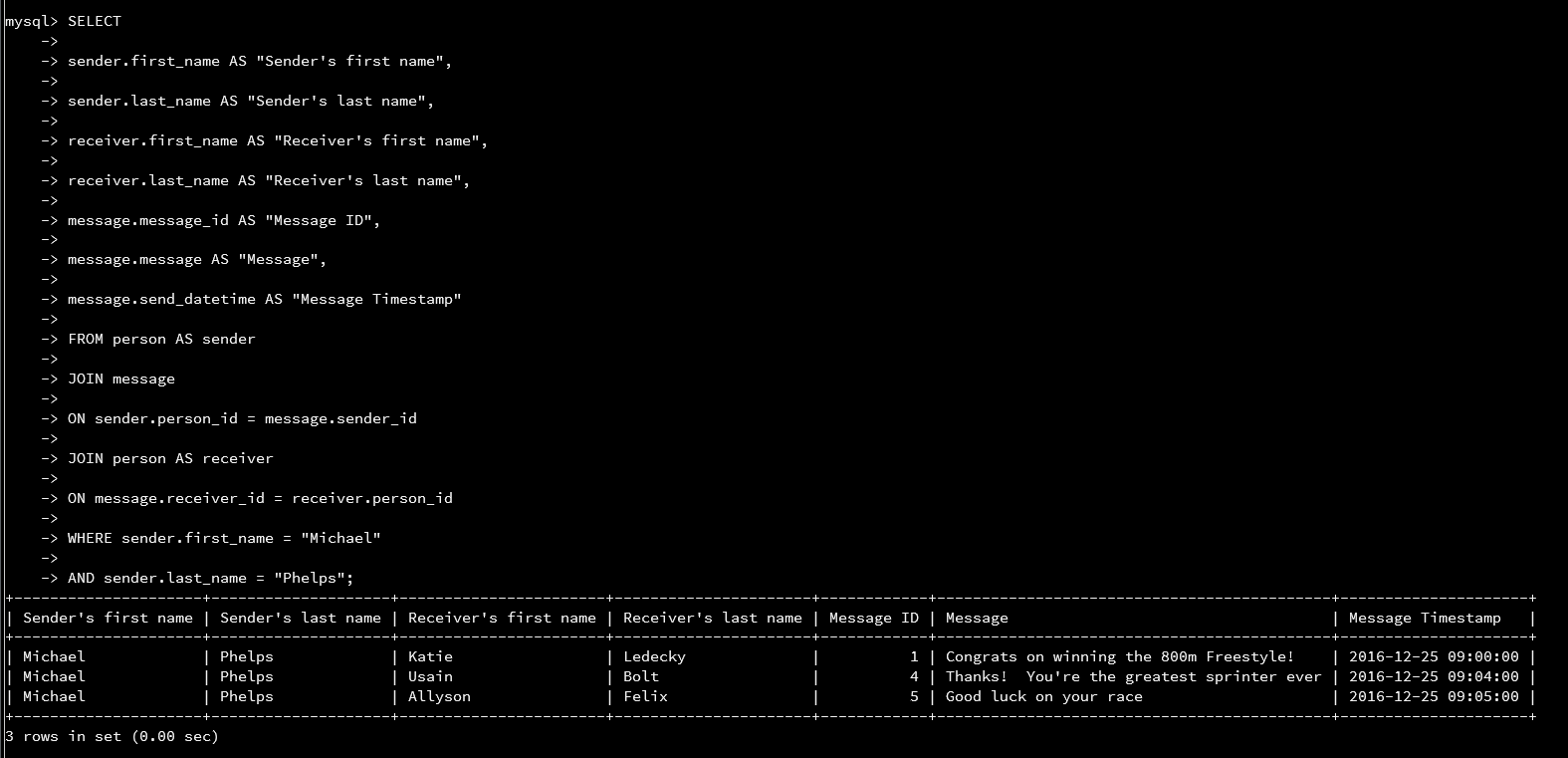


Task 12



Task 12 and Task 13 were two more *INSERT* statements that work the same as previous examples. The commands were to populate data into the new formed tables in order to run queries in the next three tasks.

Task 13

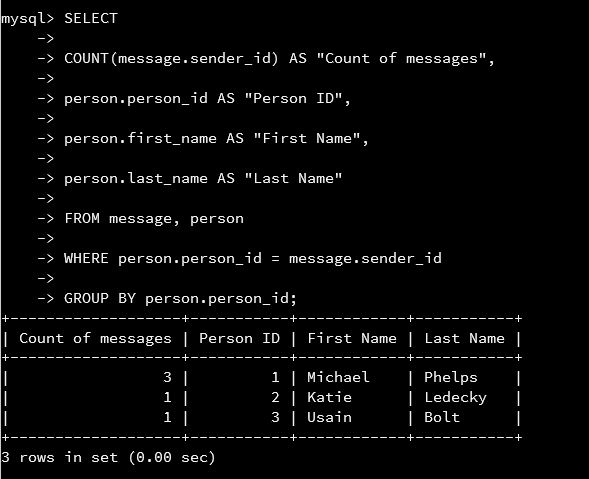


The query in task 13 was tricky as it requried converting one table into two alias tables in order to complete to task at hand. If you jump half way down into the *FROM* and *JOIN* statements, you will see that the person table has an alias of *sender* and *receiver*. This allows for the *person* table to be accessed to display the sender’s first and last name, as well as the receiver’s first and last name. Without giving each of these tables an alias with the *AS* function, both the sender and receiver’s first and last names would return with values matching the sender’s first and last name (e.g. first four columns would read “Michael” “Phelps” “Michael” “Phelps”). At this point we will jump back up to the column names that are to be shown in this *SELECT* query. As you can see, the first four columns each call their own table and field as opposed to each calling the same table (two calling the *person* table with column name *first\_name* and two calling the *person* table with column name *last\_name*). Each column displayed in the query has an alias to make the name of the column more easily readable.

To achive displaying all of the information in the query, tables have to be connected. As stated above, the *person* table was given two different aliases within the *FROM* and *JOIN* statements. Essencially, through these aliases, the person table and it’s data were cloned for just this query. This allowed for retreaving different data from the same columns of the *person table*. When connecting these tables, the fields with the same datatype were connected, and the connections to the *person* table was maid on it’s aliases.

I used the *WHERE* constraint to limit results where the sender’s first name was *Michal AND* last name was *Phelps*. Instead of using the first and last name, I could have used the sender’s *person\_id* number, but I opted not to with this small of a database.

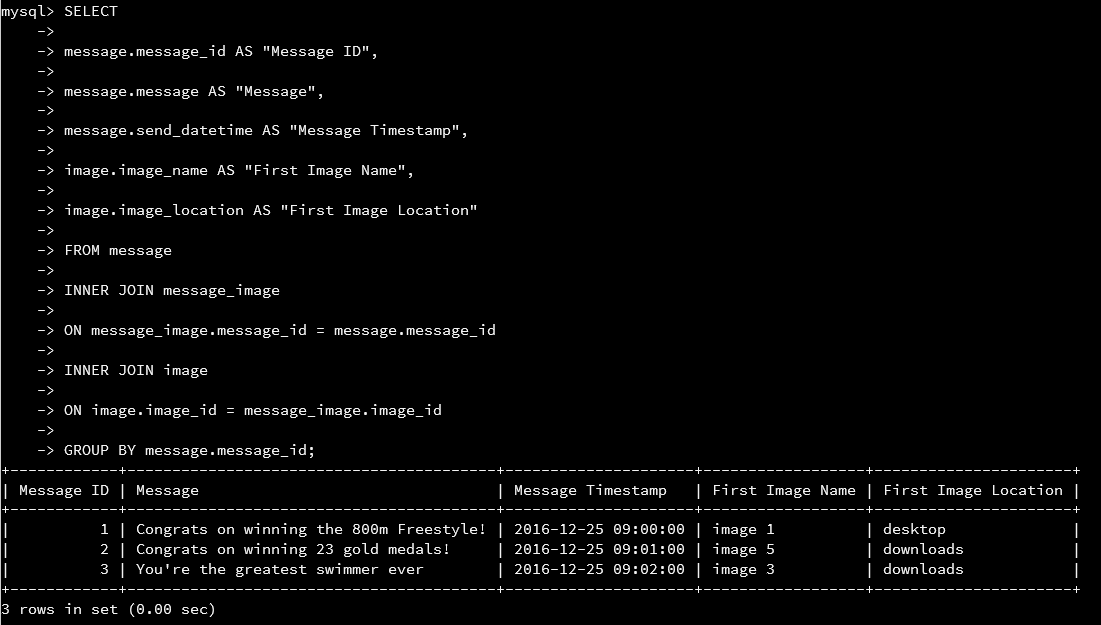
Task 14



Task 14 is another query that is counting the number of messages sent by each person. Using the *COUNT()* function, the *message.sender\_id* is being counted by combining the number of fields that contain the same value. The rest of the fields to be displayed in this SQL statement are constructed as they are in previous query statements in this project. In the *FROM* section of this statement, you can see two tables separated by a comma. This is another way to join tables without using the *JOIN* statement. I decided to do connect these tables this way specifically to show that it is possible, but using the *JOIN* statement does increase readability particularly as the number of tables becomes increasingly higher. In this statement the *WHERE* clause acts like to *ON* cause when connecting tables with the *JOIN* statement. This table is joining the *person.person\_id* to the *message.sender\_id*. The *GROUP BY* constructor is used to display each of the *person\_id*s that have sent messages. Without this *GROUP BY* function, the query would return a single row table with a message count of 5 with the person id being 1.

The results of the query shows that there were 3 different senders and each sender’s message count is displayed.

Task 15



The use of *INNER JOIN* in task 15 is essentially the only difference in the SQL statement. *INNER JOIN* works the same as *JOIN* (the terms are interchangeable). This query connects 3 tables. The *message\_image* table is a way to connect the *message* and *image* table that would otherwise not have a related fields. ­*message.image* does not contribute data into this query result, but if it was omitted from the query then the returned table would not contain data. I used the *GROUP BY* clasue to order the messages by the message id number and only show the first instance under that message id (i.e. you will not see duplicate message id numbers in this query table).

The results of this table show the message ID with what was said in the message, the time at which it was sent, the first image contained in the message (if there was more than 1 image), and the location of the first image. This query will only show messages that have images attached to them, so it shows only 3 out of the 5 messages in the database.

**Benefits of SQL**

SQL has many benefits for being query language of choice. First, SQL is a universal language that can be used with any number of different languages. This is an important attribute as it allows the developer more flexibility in selecting an approriate language for the rest of the program being developed. Additionally, SQL is realatively easy to learn and is far easier the master than other programming languages. SQL can also manage a large amount of data very quickly. This benefits programs utilizing SQL as it won’t cause a system to encounter performance issues while the query is running. The portability of SQL is a benefit as it can be utilized on many types of devices to access database information. Finally, SQL can be utilized in client/server platforms as well as in internet usage since it has very well defined standards.