**Lab 2 - Introduction to MySQL**

**Content:**

* Introduction to MySQL
* Introduction to the MySQL command window in MySQL Workbench
* Practice of simple MySQL commands

**Duration**: 4 teaching periods

**Learning outcome:**

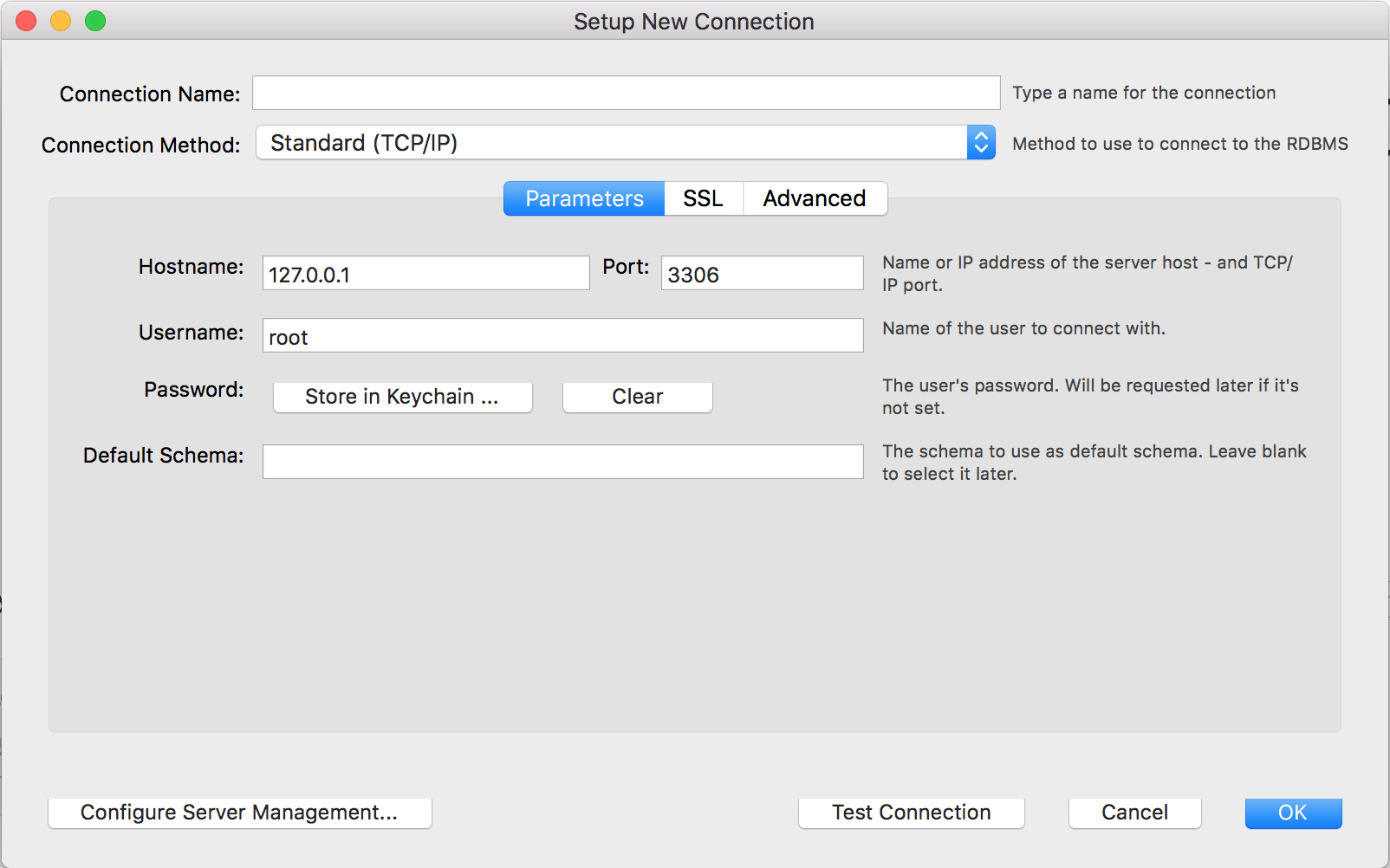
* Familiar with MySQL, how to edit and execute SQL commands in Query Editor of MySQL Workbench
* Knowledge of Data Definition Commands in MySQL
* Practice of Simple Data Query with SELECT

**Part 1: Introduction to MySQL**

**Part 2: Introduction to MySQL Workbench**

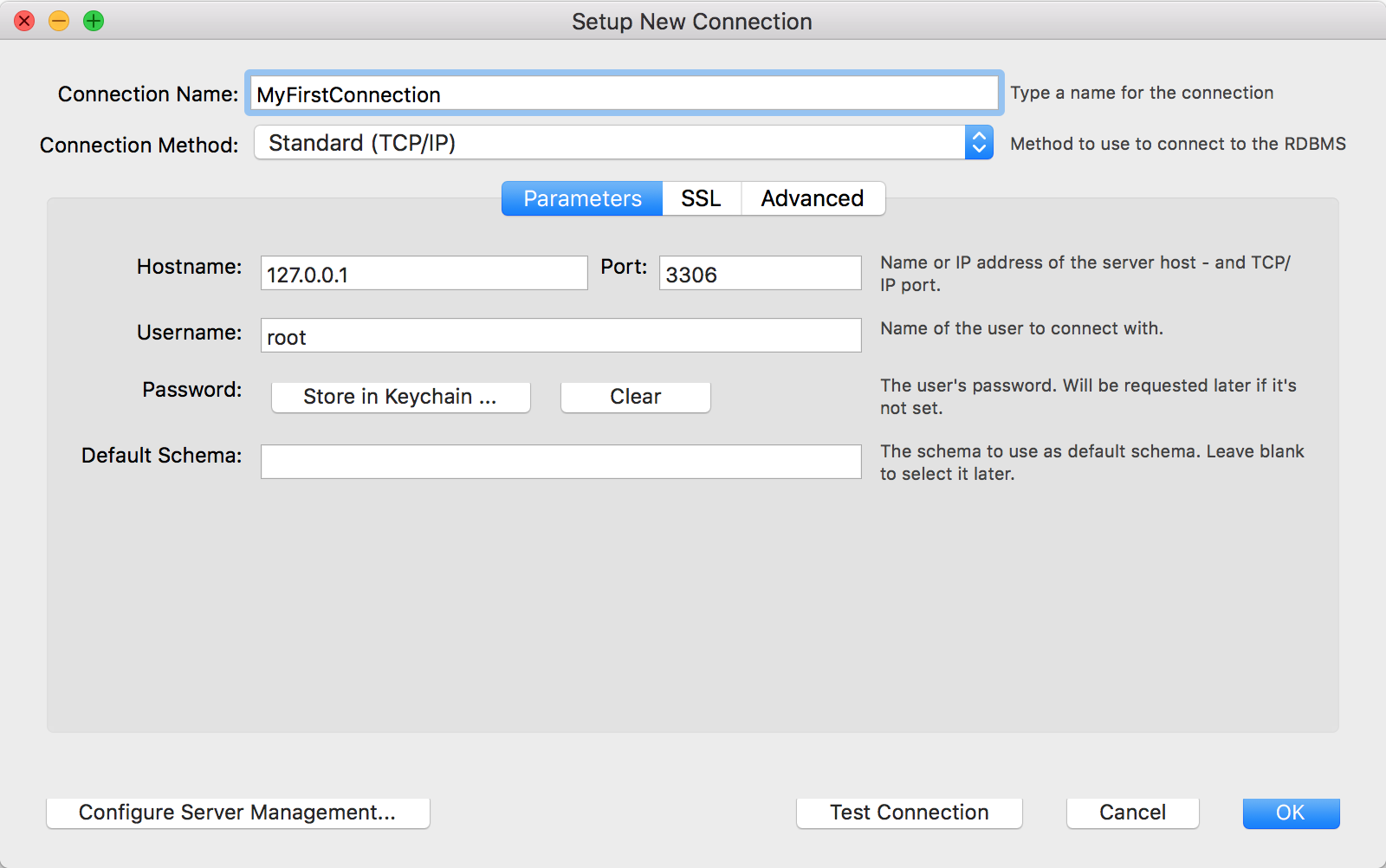
**1. Create the connection**

MySQL WorkBench allows us to create connections to manage them. This is information that when we program or install software that wants to connect to the database will use the information here to connect to. To create the connection we click on the (+) sign at the **MySQL Connections** tittle on the home screen. This opens the **Setup New Connection** form, as the following figure shows

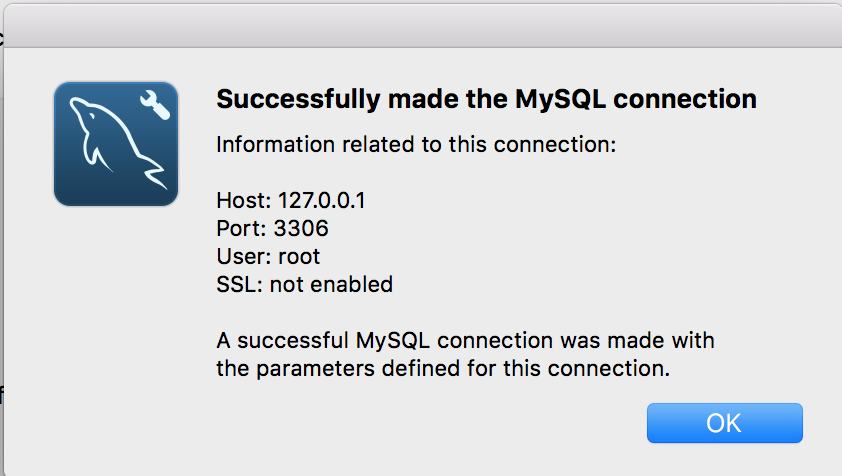


Fill out the connection details and optionally click **Configure Server Management** to execute the Server Management wizard. Click **OK** to save the connection.

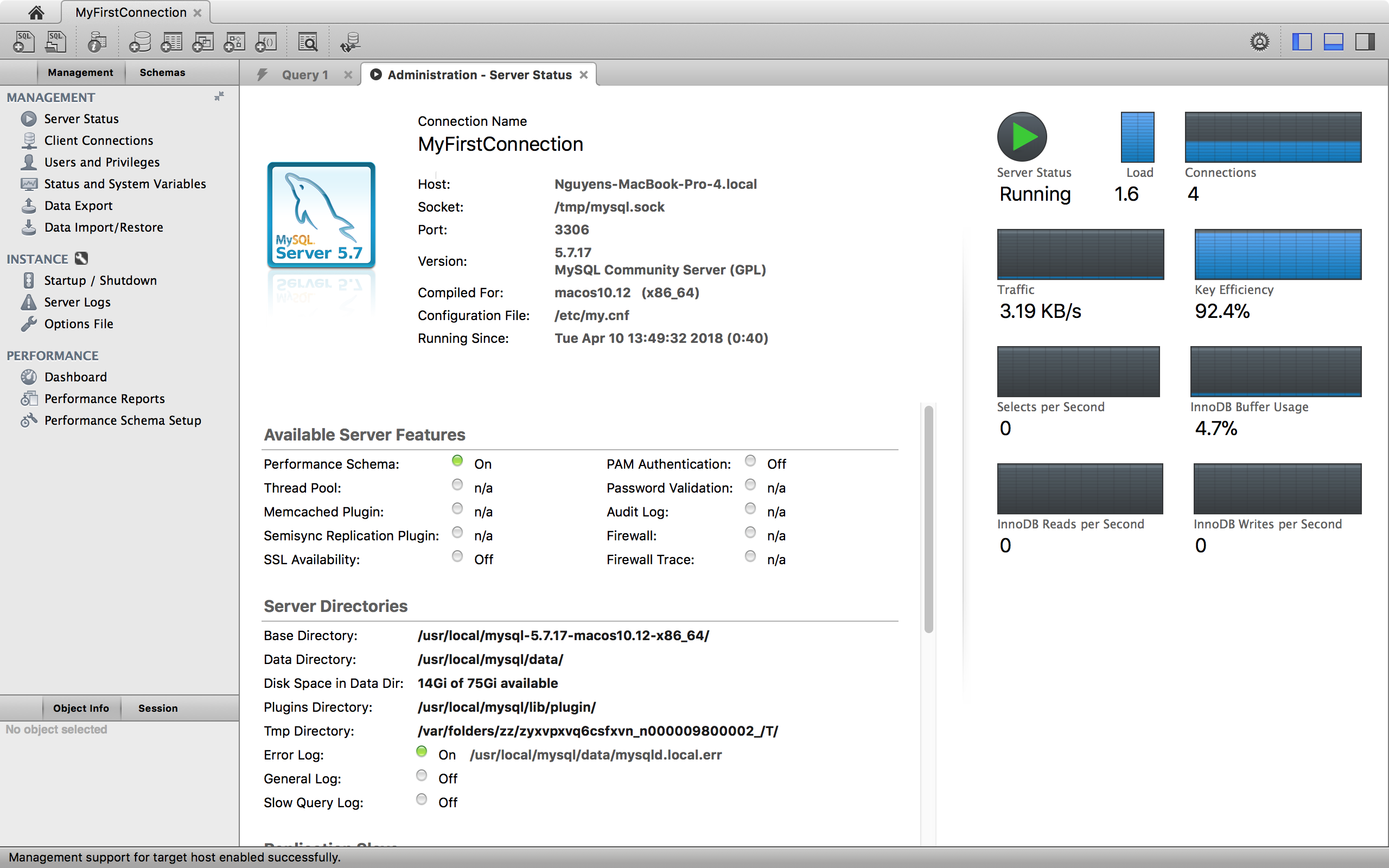
After reviewing the **Setup New Connection** information (see the figure that follows), click **Test Connection** again to make sure it still functions and then click **OK** to create the new MySQL connection



Connect well as shown below



From the home screen, click the new MySQL connection to open the SQL editor for this connection. The SQL editor is the default page. Click **Server Status** from the **Management** tab in **Navigator** panel to display the current status of the connected MySQL server instance (see the figure that follows)



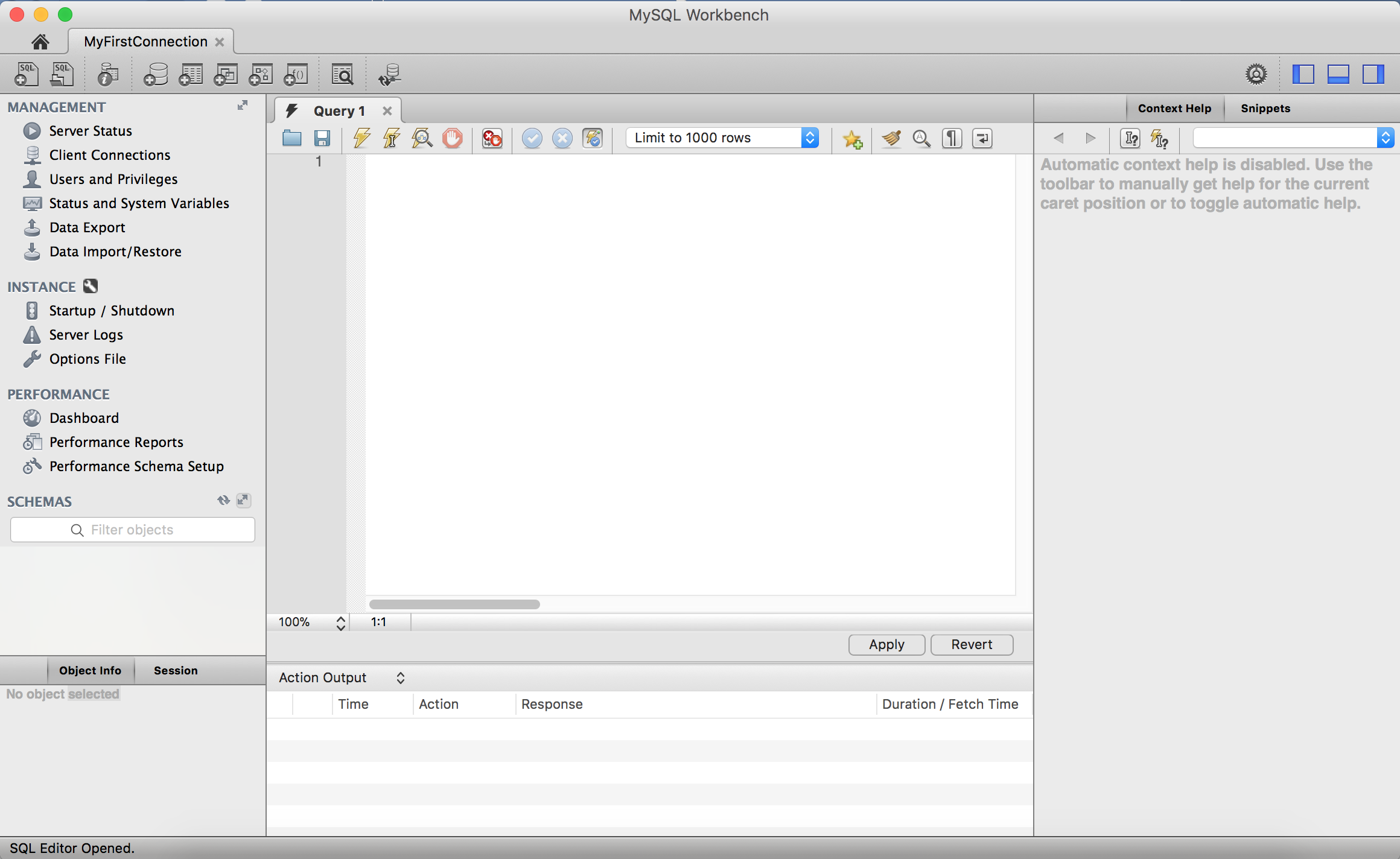
Test the other **Navigator** panel options that relate to your new MySQL connection. Check its status, MySQL logs, and measure its performance statistics from the **Dashboard**.

Notice the **Management** and **Schemas** tabs on the bottom of the Navigator panel. The Schemas view displays the schemas that are associated with your new MySQL connection. Alternatively, you can merge the Schemas and Management tabs by either clicking the merge icon on the top right of the Navigator panel, or by enabling the Show **Management Tools and Schema Tree in a single tab** SQL Editor preference.

**2. Visual SQL Editor**

The visual SQL Editor lets you build, edit and run queries, create and edit data, and view and export results. Color syntax highlighting, context sensitive help and auto-complete helps write and debug SQL statements. The integrated EXPLAIN plans provide data to help optimize the your queries. The following figure shows the main elements of the visual editor.

**Active MySQL Connection**



**Open references   
Toggle panels/sidebars**

**Statement history**

**Navigator**

**Manage your connected mySQL Server**

**Choose connected schemas**

**Collapse Schemas and Management panels**

**Refresh the Navigator**

**SQL Query menu**

**3. SQL Query Window Toolbar**

****

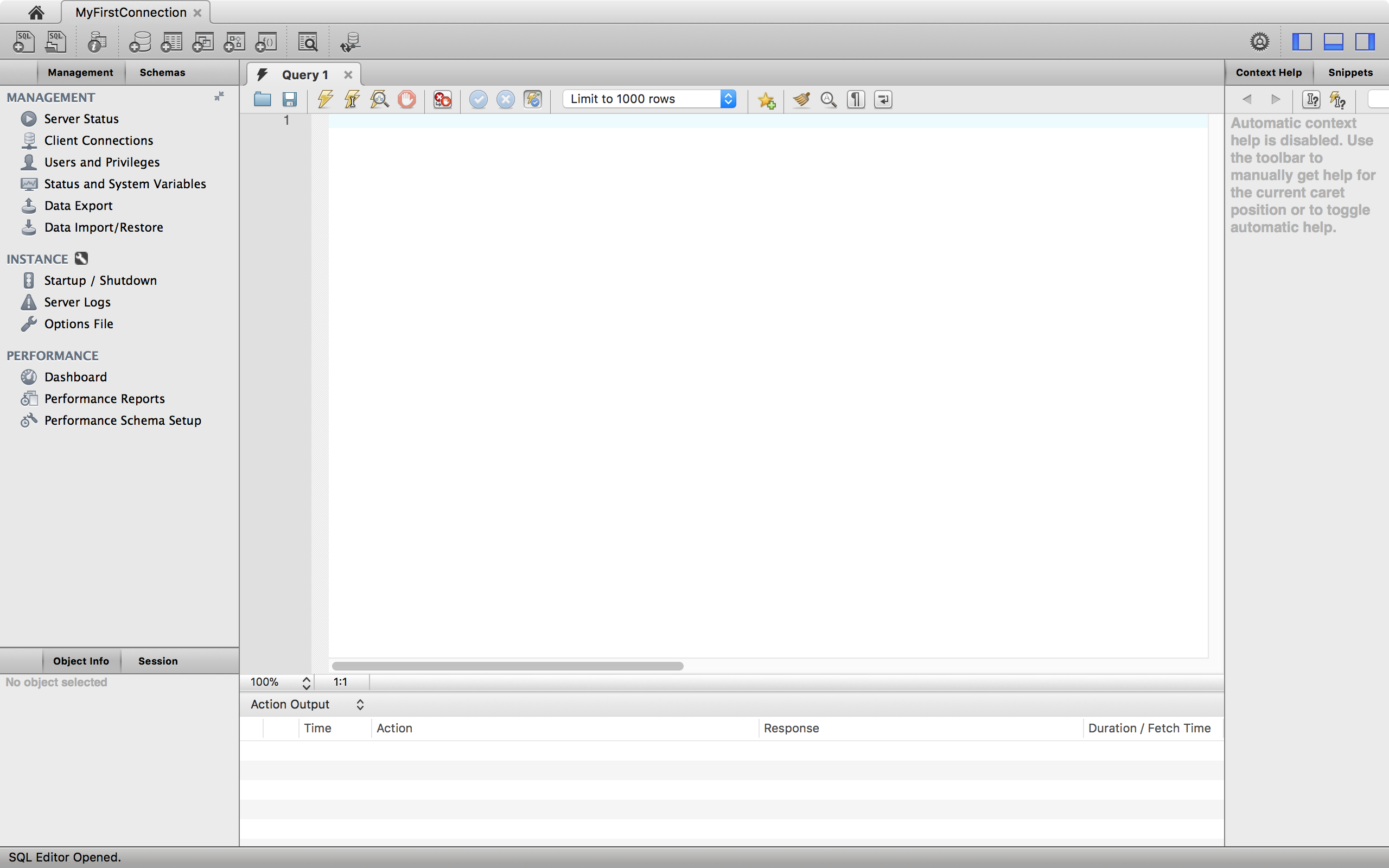
|  |  |  |
| --- | --- | --- |
| **Tool** | **Tool Function** | **Description** |
|  | **Open a Script File in this Editor** | Loads content from a saved SQL script into the SQL editor. |
|  | **Save SQL Script to File** | Saves contents from the SQL editor into a file. |
|  | **Execute SQL Script** | Executes the selected portion of the query, or the entire query if nothing is selected. |
|  | **Execute Current SQL script** | Execute the statement under the keyboard cursor. |
|  | **Explain**  **(All or Selection)** | Execute the **EXPLAIN** command on the query under the keyboard cursor.  - A "Results Grid" tab is also displayed when executing an EXPLAIN statement. Clicking it will execute the same query, as if Execute SQL Script was selected.  - Alternatively, the Visual Explain plan is already available for all executed queries. Select **Execution Plan** from the results tab to view it. |
|  | **Stop the query being executed** | Halts execution of the currently executing SQL script.  ***Note:*** *The database connection will not be restarted, and open transactions will remain open.* |
|  | **Toggle whether execution of SQL script should continue after failed statements.** | If the red “breakpoint” circle is displayed, the script terminates on a statement that fails. If you click the button  so that the green arrow is displayed, execution continues past the failed code, possibly generating additional result sets. In either case, any error generated from attempting to execute the faulty statement is recorded in the Output tab sheet. This behavior can also be set from the **SQL Execution** user preferences panel. |
|  | **Commit** | Commits the current transaction  ***Note:*** *All query tabs in the same connection share the same transactions. To have independent transactions, a new connection must be opened.* |
|  | **Rollback** | Rolls back the current transaction  ***Note:*** *All query tabs in the same connection share the same transactions. To have independent transactions, a new connection must be opened.* |
|  | **Toggle Auto-Commit Mode** | If selected, each statement will be committed independently.  ***Note:*** *All query tabs in the same connection share the same transactions. To have independent transactions, a new connection must be opened*.  Auto-commit is enabled by default, and this default behavior can be modified (disabled) under the SQL Execution user preferences panel. |
|  | **Set Limit for Executed Queries** | The default value is 1000, which appends "LIMIT 0, 1000" to SELECT queries.  The default (1000) can be changed from the SQL Execution user preferences panel. |
|  | **Save Snippet** | Save the current statement or selection to the active snippet list. |
|  | **Beautify SQL** | Beautify/reformat the SQL script. By default, SQL keywords are changed to UPPER CASE. This functionality can be changed from the **SQL Editor** user preferences panel. |
|  | **Find panel** | Show the Find panel for the editor. |
|  | **Invisible characters** | Toggle display of invisible characters, such as newlines, tabs, spaces.  A new line is represented as **[LF]**, a space as a single dot (.), and a tab as a right arrow. |
|  | **Wrapping** | Toggles the wrapping of long lines in the SQL editor window. |

**Practice:** Self-exploring Query Editor.

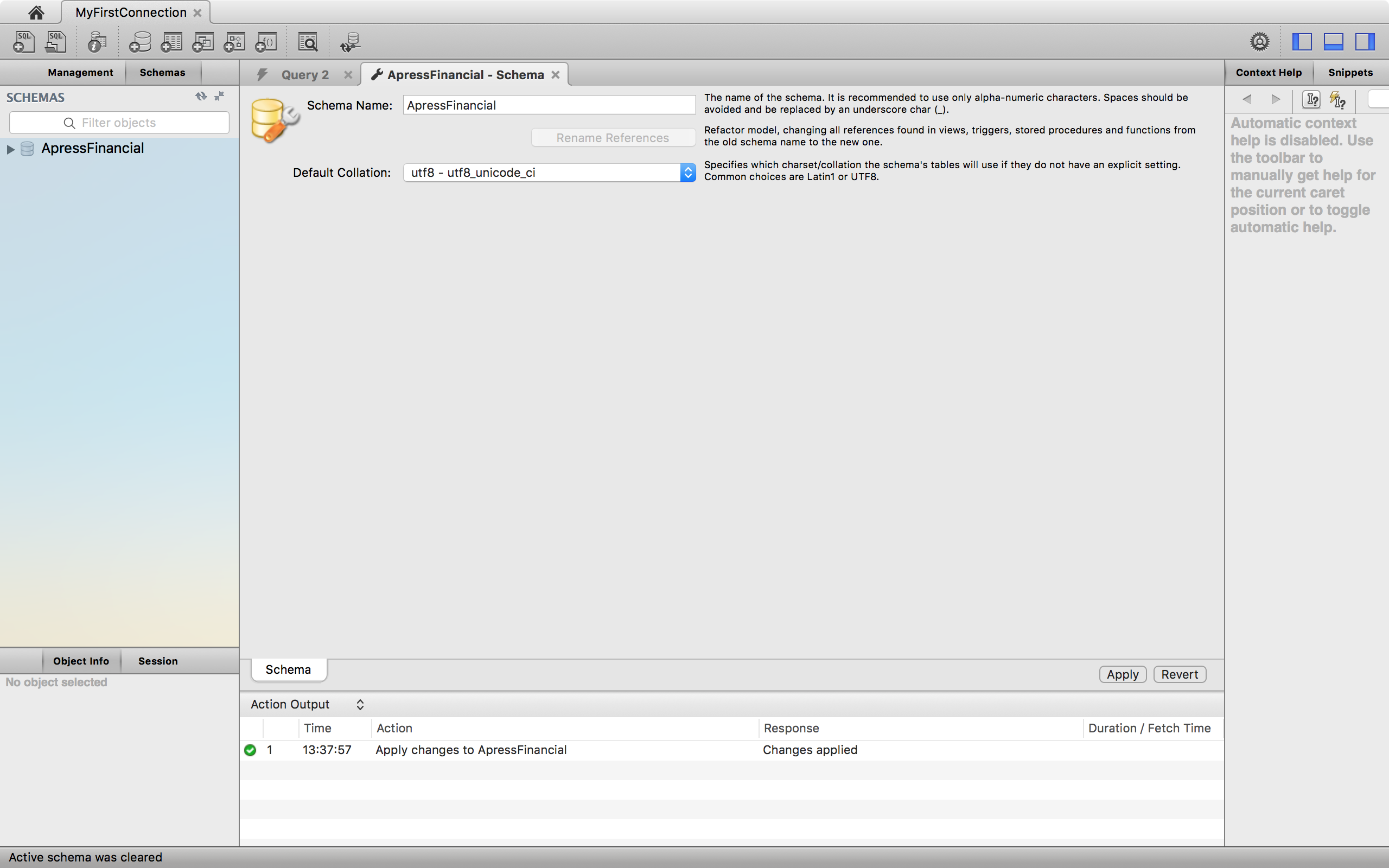
**Part 3: Data Definition Command in MySQL**

**Practice:** Defining a Table Through Query Editor

1. Create New Schema



**Click here to create new schema**



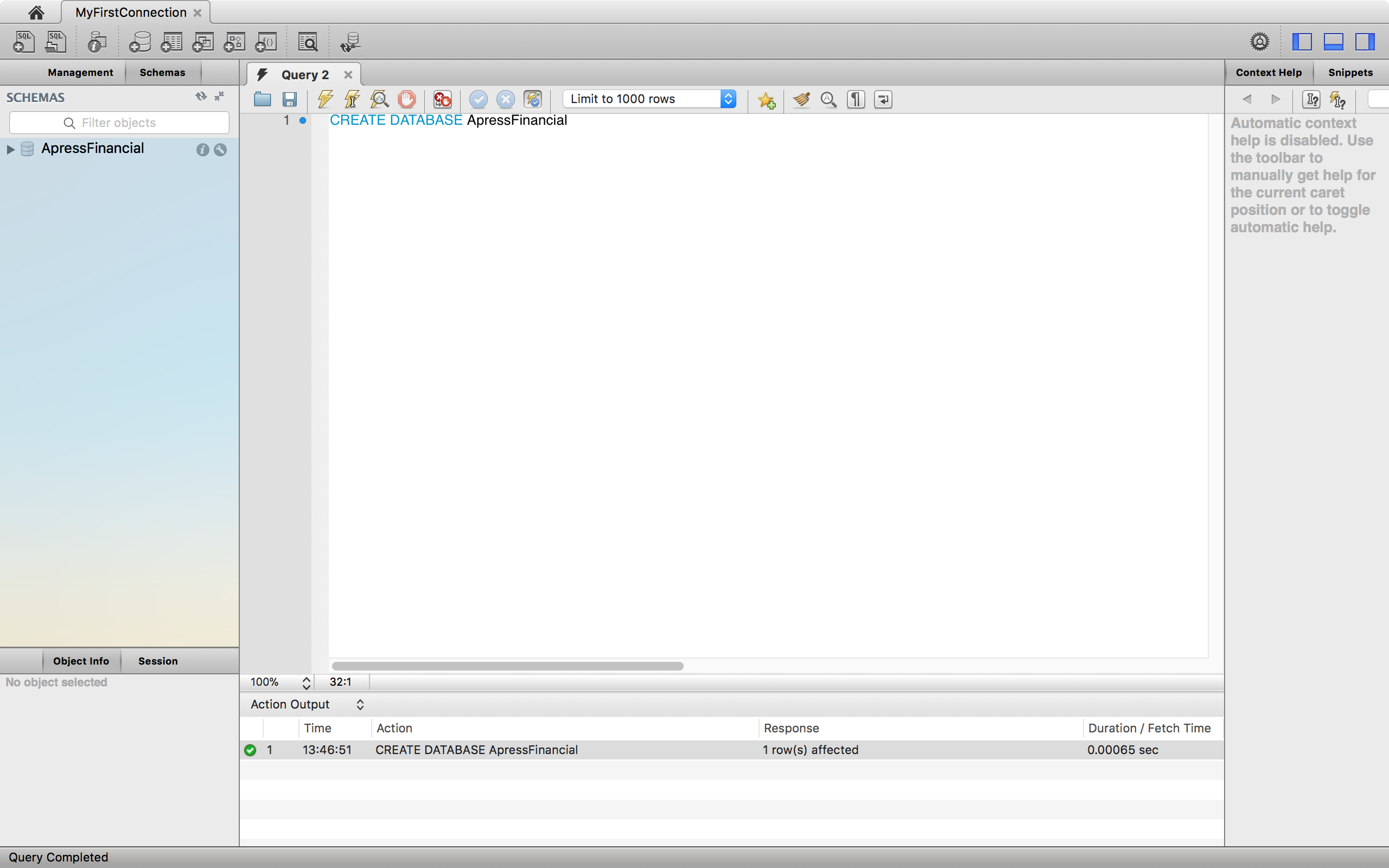
**4. New Schema is created**

**3. Click Apply to execute**

**2. Choose default collation**

**1. Fill in Schema name**

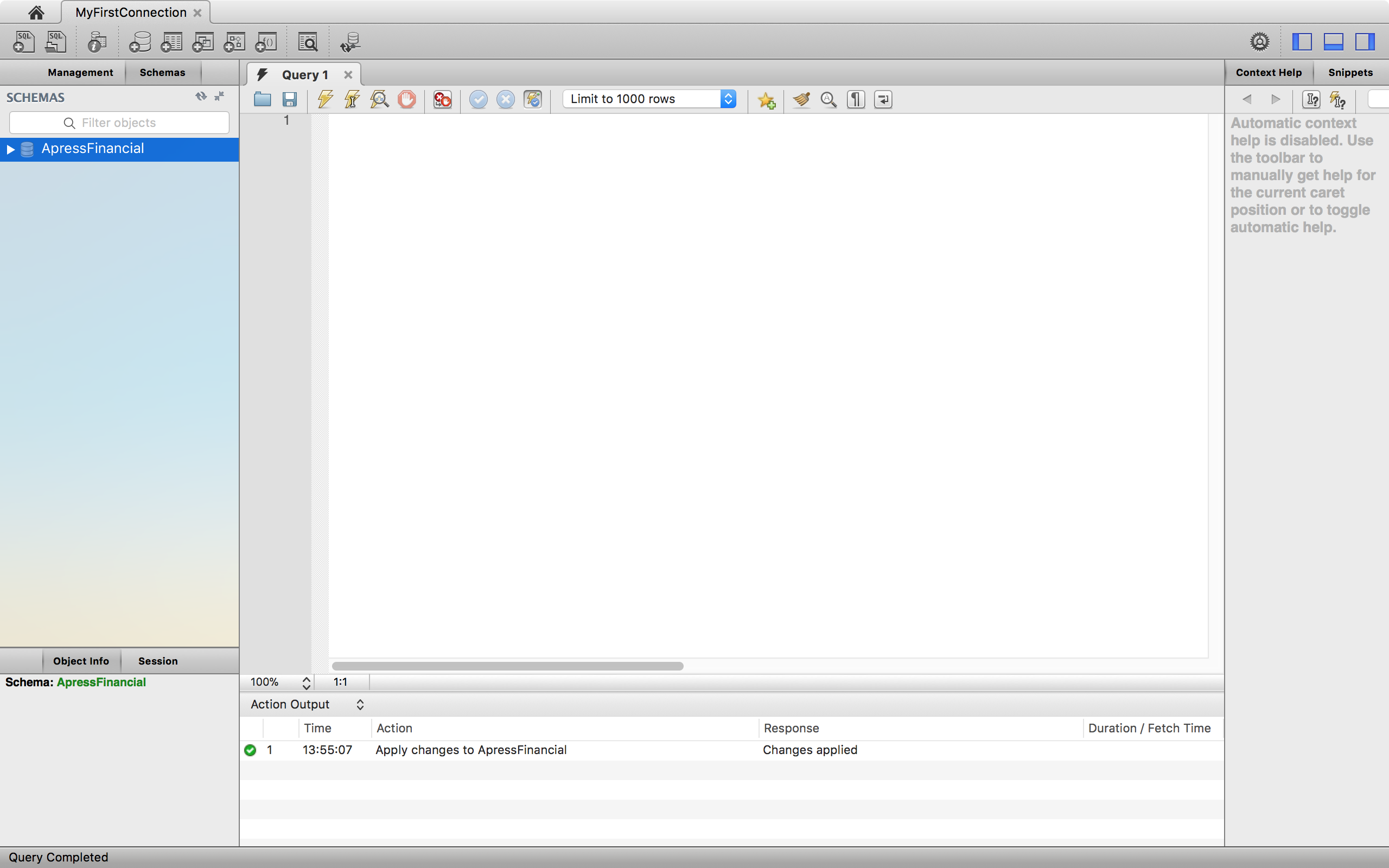
We also have another way to create New Schema



**Enter the code**

**New Schema is created**

2. Ensure that you are pointing to the **ApressFinancial** database in Query Editor, as in the following table:



For preparation, it is necessary to open CreateDatabase\_Table.sql file, double check the file path in the script, and then run it.

2. In Query Editor, enter the following code:

CREATE TABLE Transactions(

TransactionId bigint NOT NULL,

CustomerId bigint NOT NULL,

TransactionType int NOT NULL,

DateEntered datetime NOT NULL,

Amount numeric(18, 5) NOT NULL,

ReferenceDetails nvarchar(50) NULL,

Notes nvarchar(50) NULL,

RelatedShareId bigint NULL,

RelatedProductId bigint NOT NULL

);

3. Execute the code by either pressing key combination, or clicking the toolbar Execute button.

4. You should now see the following message:

../../../Desktop/Screen%20Shot%202018-04-12%20at%202.23.36%20P

5. However, you may have received an error message instead. This could be for a number of reasons, from a typing mistake through to not having the authority to create tables.

6. Now move to Navigation bar in the left side and open tab **Schemas**. If it is already open, you will have to refresh the schemas tab ../../../Desktop/Screen%20Shot%202018-04-12%20at%202.29.57%20P. You should then see the **Transactions** table alongside the **Customers** table created previously.

7. Similarly, you create the TransactionTypes as follows:

CREATE TABLE TransactionTypes(

TransactionTypeId int NOT NULL,

TransactionDescription nvarchar(30) NOT NULL,

CreditType bit NOT NULL

);

**The basic syntax for creating a table is as follows**:

CREATE TABLE tablename(

column\_name datatype [length] [NULL/NOT NULL],

column\_name datatype [length] [NULL/NOT NULL],

...

);

**The ALTER TABLE Statement**

The ALTER TABLE statement allows restrictive alterations to a table layout but keeps the contents. Columns can be added, removed, or modified using the ALTER TABLE statement. Removing a column will simply remove the data within that column, but careful thought has to take place before adding, removing, or altering a column.

Add a column:

1. First of all, open up Query Editor and ensure that you are pointing to the ApressFinancial database. Then write the code to alter the TransactionTypes table to add the new column. The format is very simple. We specify the table prefixed by the schema name we want to alter after the ALTER TABLE statement. Next we use a comma-delimited list of the columns we wish to add. We define the name, the data type, the length if required, and finally whether we allow NULLs or not. As we don’t want the existing data to have any default values, we will have to define the column to allow NULL values.

ALTER TABLE TransactionTypes

ADD AffectCashBalance bit NULL;

2. Once we’ve altered the data as required, we then want to remove the ability for further rows of data to have a NULL value. This new column will take a value of 0 or 1. Again, we use the ALTER TABLE statement, but this time we’ll add the MODIFY statement with the name of the column we wish to alter. After this statement, the results are the alterations we wish to make. Although we are not altering the data type, it is a mandatory requirement to redefine the data type and data length. After this, we can inform MySQL Workbench that the column will not allow NULL values.

ALTER TABLE TransactionTypes

MODIFY AffectCashBalance bit NOT NULL;

3. Execute the preceding code to make the **TransactionTypes** table correct.

**Creating the Remaining Tables**

We need to create the remaining four tables. We will do this as code placed in Query Editor. There is nothing specifically new to cover in this next section, and therefore only the code is listed. Enter the following code and then execute it as before. You can then move into MySQL Workbench and refresh it, after which you should be able to see the new tables.

CREATE TABLE FinancialProducts(

ProductId bigint NOT NULL,

ProductName nvarchar(50) NOT NULL

);

CREATE TABLE CustomerProducts (

CustomerFinancialProductId bigint NOT NULL,

CustomerId bigint NOT NULL,

FinancialProductId bigint NOT NULL,

AmountToCollect decimal(15,2) NOT NULL,

Frequency smallint NOT NULL,

LastCollected datetime NOT NULL,

LastCollection datetime NOT NULL,

Renewable bit NOT NULL

);

CREATE TABLE Shares(

ShareId bigint NOT NULL AUTO\_INCREMENT,

ShareDesc nvarchar(50) NOT NULL,

ShareTickerId nvarchar(50) NULL,

CurrentPrice numeric(18, 5) NOT NULL

);

CREATE TABLE SharePrices(

SharePriceId bigint NOT NULL AUTO\_INCREMENT,

ShareId bigint NOT NULL,

Price numeric(18, 5) NOT NULL,

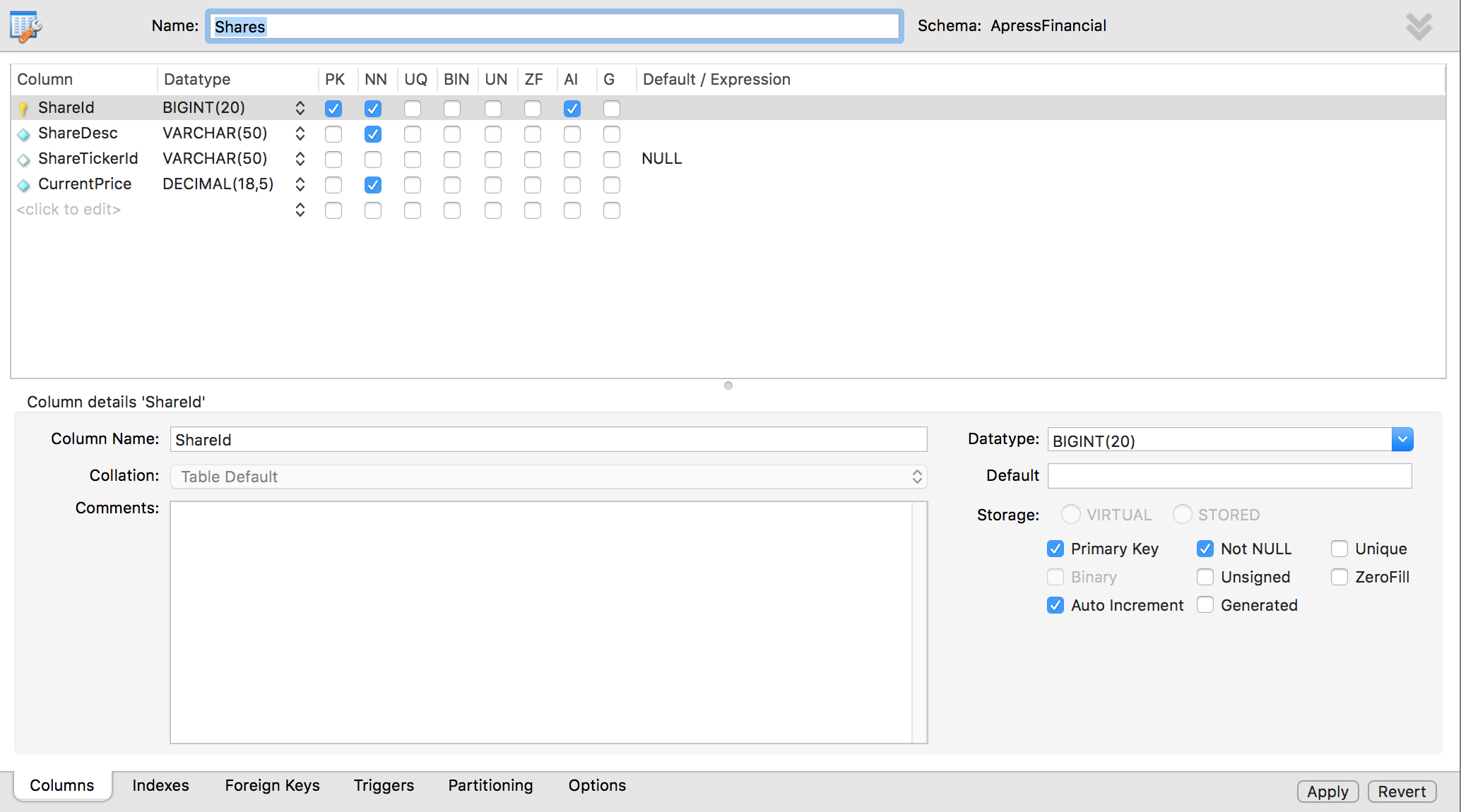
PriceDate datetime NOT NULL

);

**Practice: Defining a Primary Key**

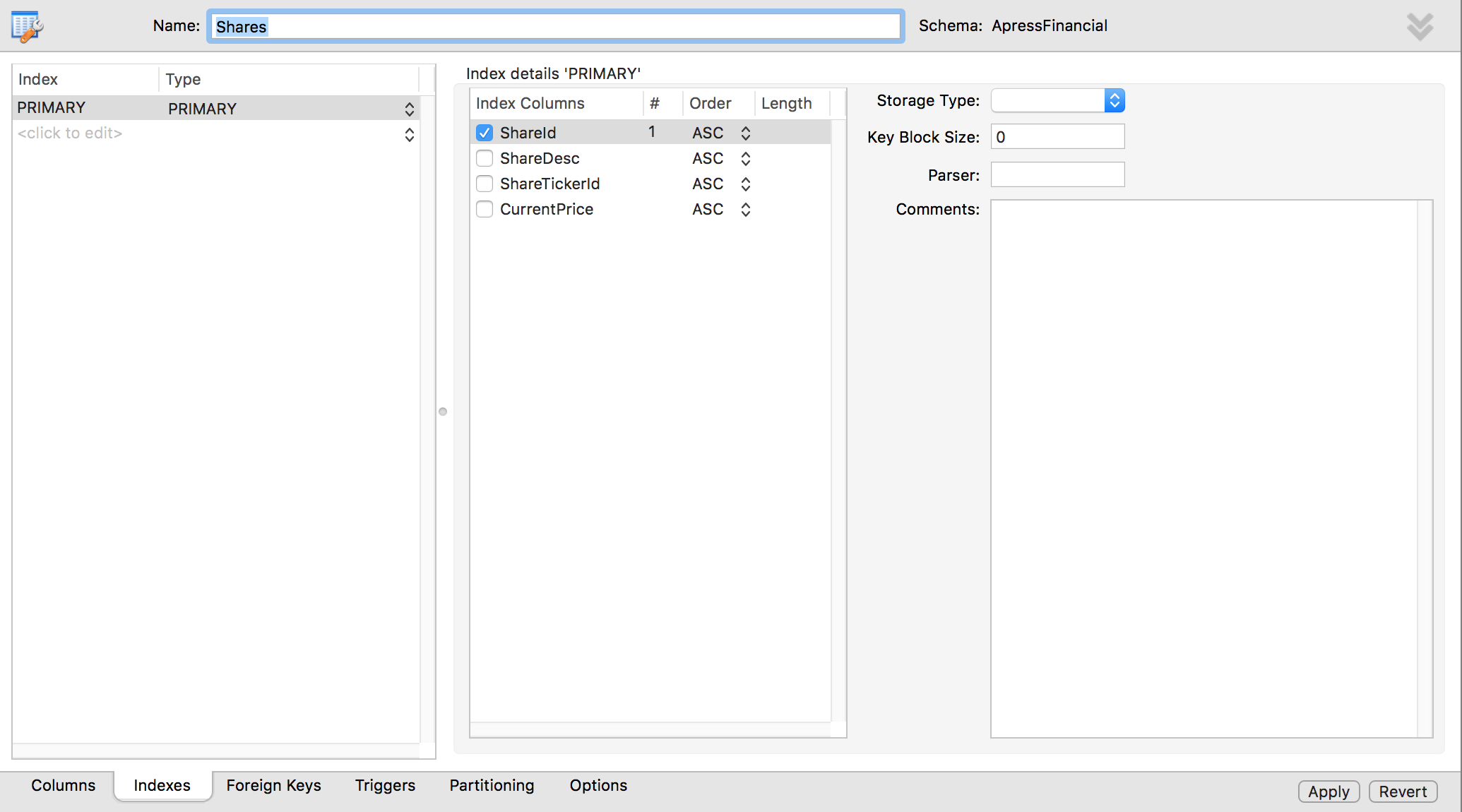
This section will demonstrate how to set a primary key of a table over the interface of MySQL Workbench.

1. In MySQL Workbench, you have navigated to the **ApressFinancial** database. Find the **Shares** table, and click on icon ../../../Desktop/Screen%20Shot%202018-04-12%20at%203.27.19%20P. Setting tab will appear for you to edit many features of the table (PK is Primary Key)



In PK column, you can check or uncheck to define Primary Key.

2. However, this is not all that happens, as you will see. Save the table modifications by clicking the Apply button. Move to Indexes tab.



Look at Type. It says Primary Key. Notice that a key definition has been created for you, with a name and the selected column, informing you that the index is unique and clustered (more on indexes and their relation to primary keys will be introduced in later labs).

That’s all there is to creating and setting a primary key. A primary key has now been set up on the Shares table. In this instance, any record added to this table will ensure that the data will be kept in ShareId ascending order, and it is impossible to insert a duplicate row of data. This key can then be used to link to other tables within the database at a later stage.

**Creating a Relationship between Tables**

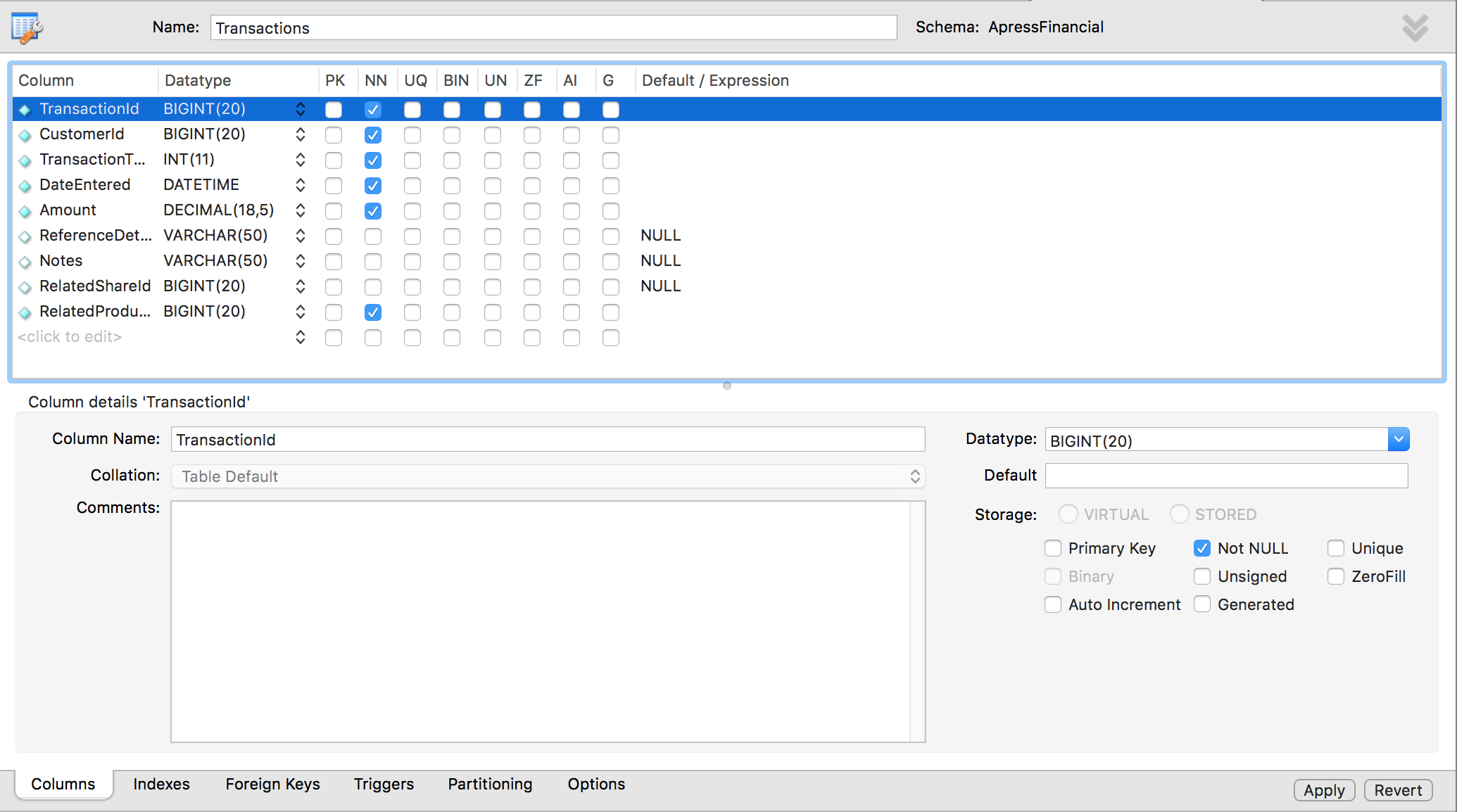
In a database schema, there are relationships, each links two tables. In this section, the first relationship that we create will be between the customer and customer transactions tables. This will be a one-to-many relationship where there is one customer record to many transaction records. Keep in mind that although a customer may have several customer records, one for each product he or she has bought, the relationship is a combination of customer and product to transactions because a new CustomerId will be generated for each product the customer buys. We will now build that first relationship.

1. In SQL Server environment, we have the **ApressFinancial** database be selected. We need to add a primary key to **Customers**. Enter the code that follows and then execute it:

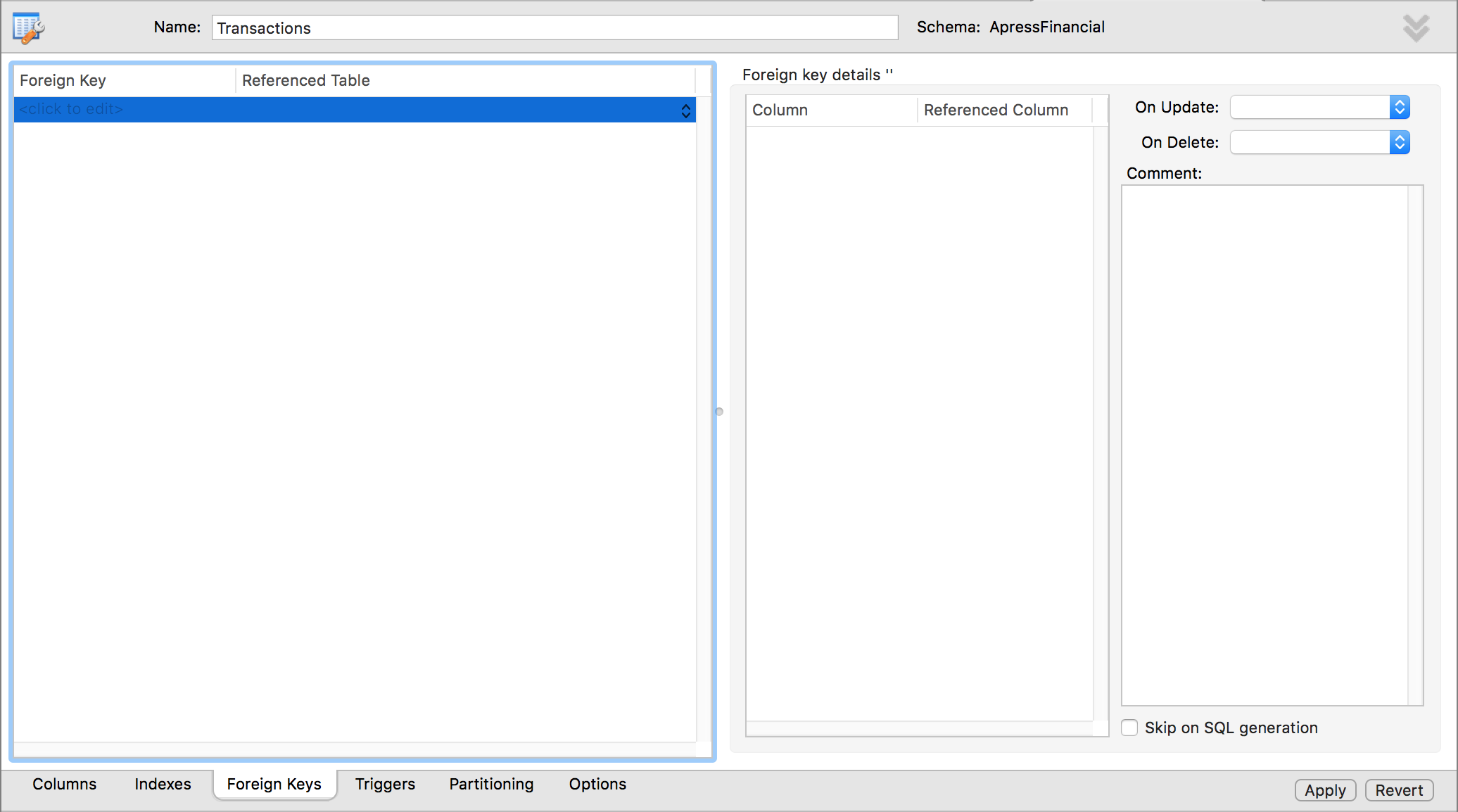
ALTER TABLE Customers

ADD CONSTRAINT PK\_Customer PRIMARY KEY (CustomerId);

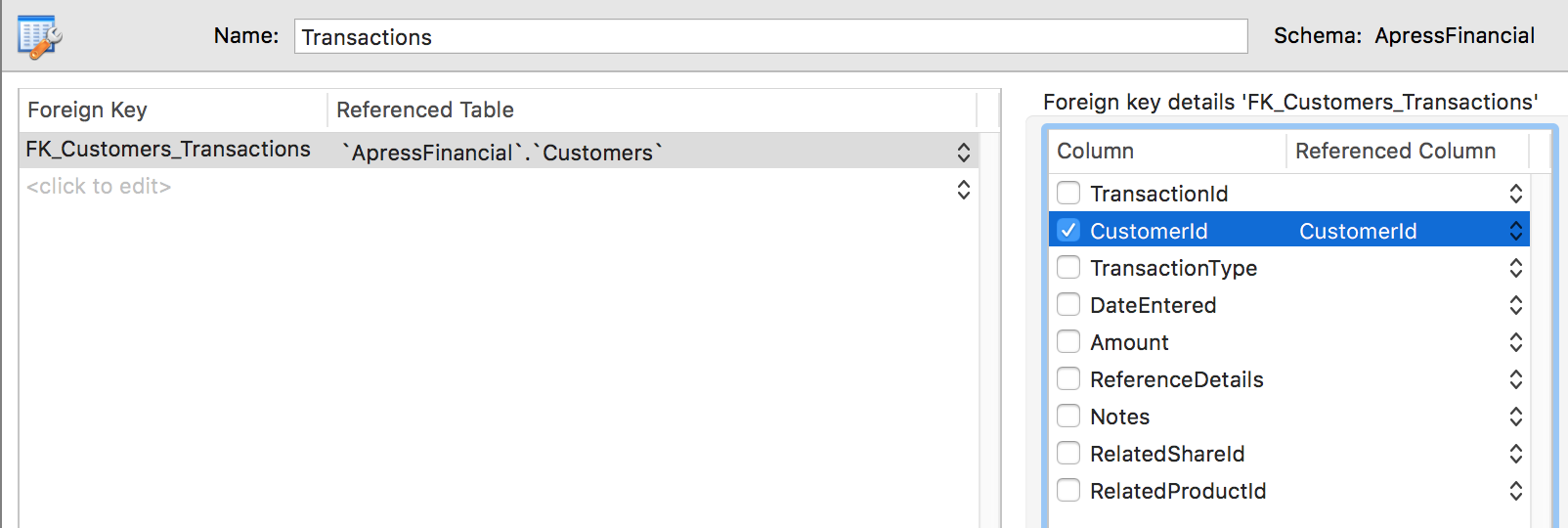
2. Open setting tab of **Transactions** table.



3. Move to the Foreign Keys. As it is empty, you need to click **<click to edit>** to add new



4. The first requirement is to change the name to make it more meaningful. Quite often you will find that naming the key **FK\_ParentTable\_ChildTable** is the best method, so in this case type **FK\_Customers\_Transactions** as the **Customers** table will be the master table for this foreign key. We also need to define the column in each table that is the link. We are linking every one customer record to many transaction records, and we can do so via the **CustomerId**. Choose **Referenced Table** is **Customers** then Choose column and referenced column. Now click Apply.



**Check Existing Data on Creation**

**Enforce Foreign Key Constraints**

**Delete Rule/Update Rule**

**Part 4: Simple Data Query in MySQL**

**Insert One Row of Data by the MySQL INSERT Statement**

INSERT [LOW\_PRIORITY | DELAYED | HIGH\_PRIORITY] [IGNORE]

[INTO] *tbl\_name*

[PARTITION (*partition\_name* [, *partition\_name*] ...)]

[(*col\_name* [, *col\_name*] ...)]

{VALUES | VALUE} (*value\_list*) [, (*value\_list*)] ...

[ON DUPLICATE KEY UPDATE *assignment\_list*]

*value:*

{*expr* | DEFAULT}

*value\_list:*

*value* [, *value*] ...

*assignment:*

*col\_name* = *value*

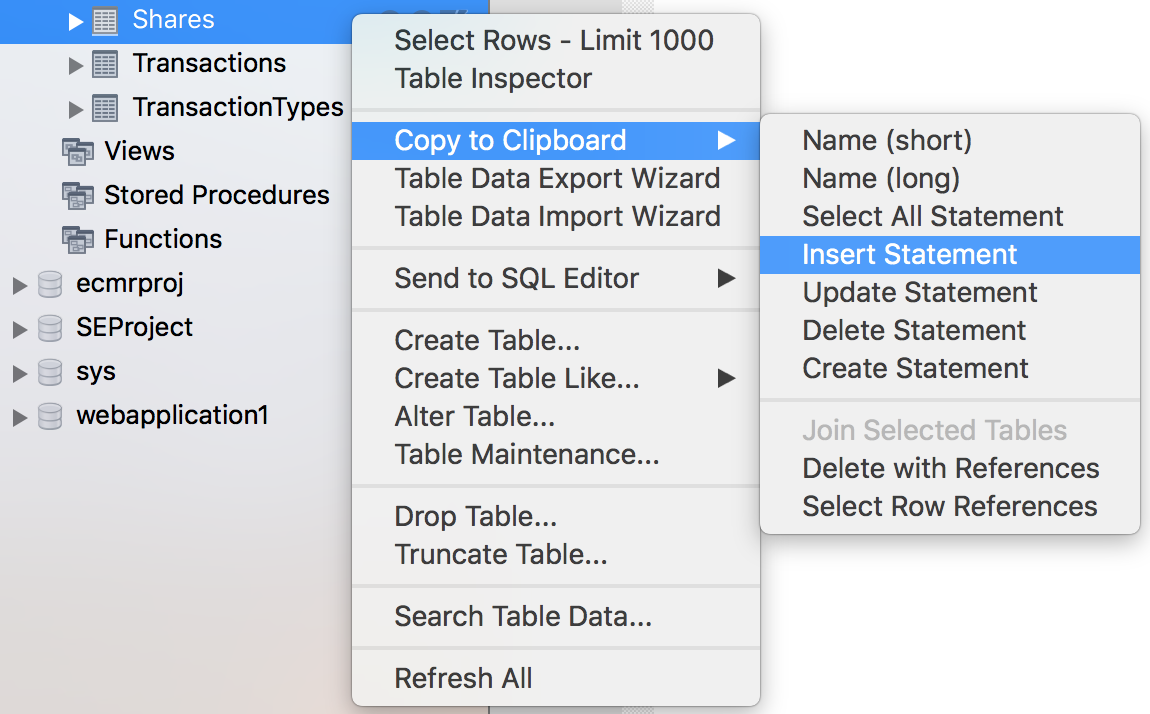
*assignment\_list:*

*assignment* [, *assignment*] ...

Example of INSERT:

1. Log in an account which can modify the **ApressFinancial** schema, open a Query Editor window,choose the **ApressFinancial** schema.

2. Right-click against the **Shares** table, select Copy to Clipboard 🡪 Insert Statement



3. Past Statement in Clipboard to Query Editor:

INSERT INTO `ApressFinancial`.`Shares`

(`ShareId`,

`ShareDesc`,

`ShareTickerId`,

`CurrentPrice`)

VALUES

(<{ShareId: }>,

<{ShareDesc: }>,

<{ShareTickerId: }>,

<{CurrentPrice: }>);

4. Modify code to:

INSERT INTO `ApressFinancial`.`Shares`

(`ShareId`,

`ShareDesc`,

`ShareTickerId`,

`CurrentPrice`)

VALUES

(1,

"ACME'S HOMEBAKE COOKIES INC",

'AHCI',

2.34125);

5. Execute the code. The following result appears:

../../../Desktop/Screen%20Shot%202018-04-12%20at%209.37.35%20P

**Default Values**

Default values are used when a large number of INSERTs for a column would have the same value entered each time. For such column, the value does not need to appear in the INSERT command.

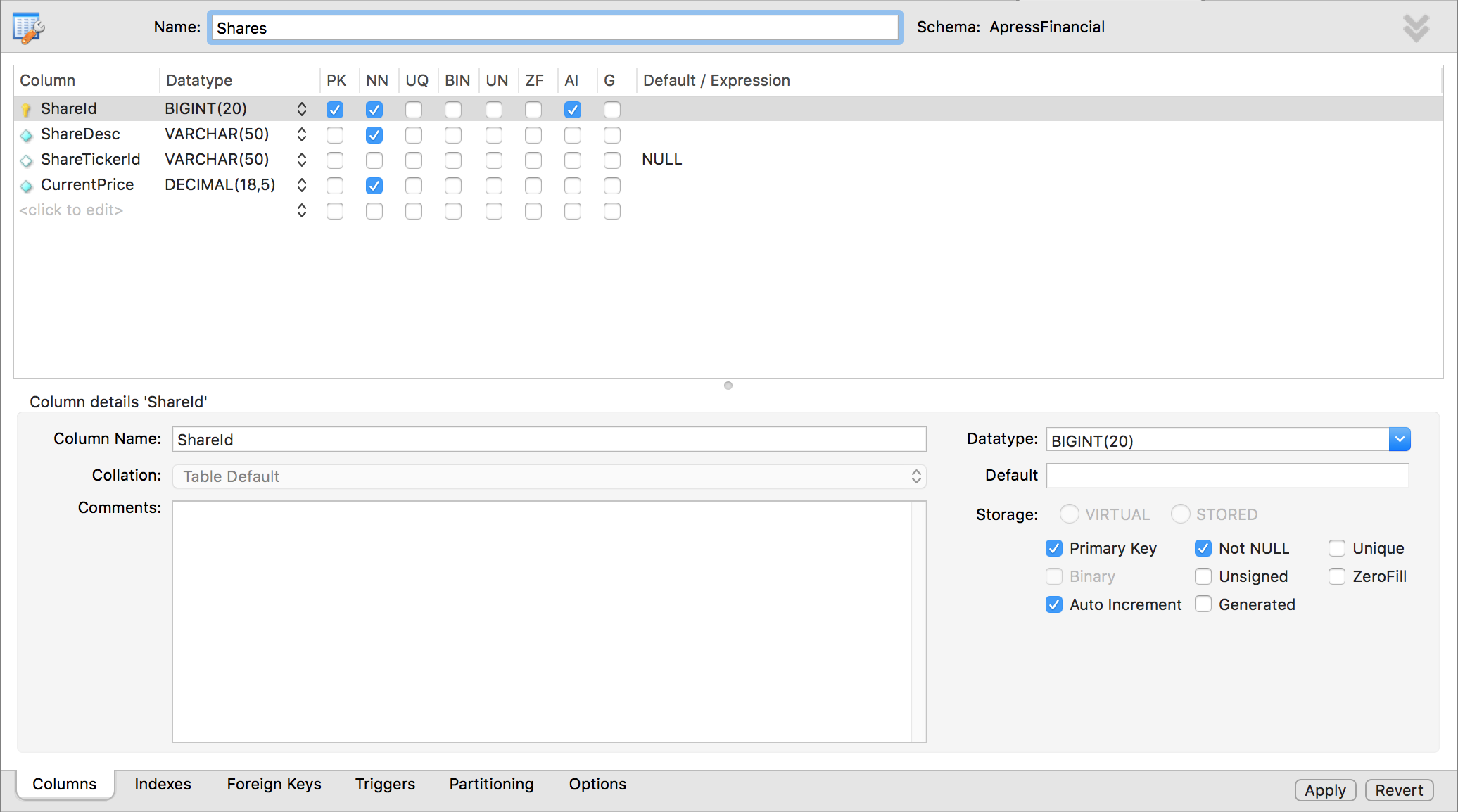
When creating the **Customers** table, column **DateAdded** is set up to be populated with a default value which is set by function **NOW()** in MySQL Workbench. When a new row is added, MySQL Workbench calls the function and assigns the returned value to the column.

ALTER TABLE Customers

MODIFY DateAdded datetime NULL DEFAULT NOW();

**Using NULL Values**

The next method for avoiding having to fill in data for every column is to allow NULL values in the columns. When defining the tables, set up NN (Not NULL) column (Check means do not allow Null value). As in the following figure, one of the columns in the **Shares** table, **ShareTickerId** does allow a NULL value to be entered into it.



In this table, ShareID is AI (Auto Increment). Thus, only two rows required values in the INSERT command. Comparing to the previous example, the code is reduced to:

INSERT INTO Shares(

ShareDesc,

CurrentPrice)

VALUES

("ACME'S HOMEBAKE COOKIES INC",

2.34125);

After executing this command, when opening the table, the result is as:



The value of NULL requires special handling within MySQL Workbench or applications that will be viewing this data. What this value actually means is that the information within the column is unknown; it is not a numeric or an alphanumeric value. Therefore, because you don’t know if it is numeric or alphanumeric, you cannot compare the value of a column that has a setting of NULL to the value of any other column, and this includes another NULL column.

**Column Constraints**

A constraint is essentially a check that MySQL Workbench places on a column to ensure that the data to be entered in the column meets specific conditions. This will keep out data that is not satisfying the condition and therefore avoid data inconsistencies. Constraints are used to keep database integrity by ensuring that a column only receives data within certain parameters.

ALTER TABLE Customers

MODIFY DateAdded datetime NULL DEFAULT NOW();

**Practice: Add constraint**

1. Ensure that Query Editor is running. Although all the examples deal with the **CustomerProducts** table, each constraint being added to the table will be created one at a time, which will allow a discussion for each point to take place. In the Query Editor pane, enter the following code, which will add a primary key to the **CustomerProducts** table. This will place the **CustomerFinancialProductId** column within the key, which will be clustered.

ALTER TABLE CustomerProducts

ADD CONSTRAINT PK\_CustomerProducts PRIMARY KEY (CustomerFinancialProductId);

2. Next we add a CHECK constraint on the AmountToCollect column. The **CustomerProducts** table is once again altered, and a new constraint added called **CK\_CustProds\_AmtCheck**. This constraint will ensure that for all rows inserted into the **CustomerProducts** table from this point on, the score must be greater than 0.

ALTER TABLE CustomerProducts

ADD CONSTRAINT CK\_CustProds\_AmtCheck

CHECK(AmountToCollect > 0);

3. Moving on to the third constraint to add to the **CustomerProducts** table, we have a DEFAULT value constraint. In other words, this will insert a value of 0 to the Renewable column if no value is entered specifically into this column. This signifies that the premium collected is a one-off collection.

ALTER TABLE CustomerProducts ALTER Renewable SET DEFAULT 0;

4. Execute then see the following result:

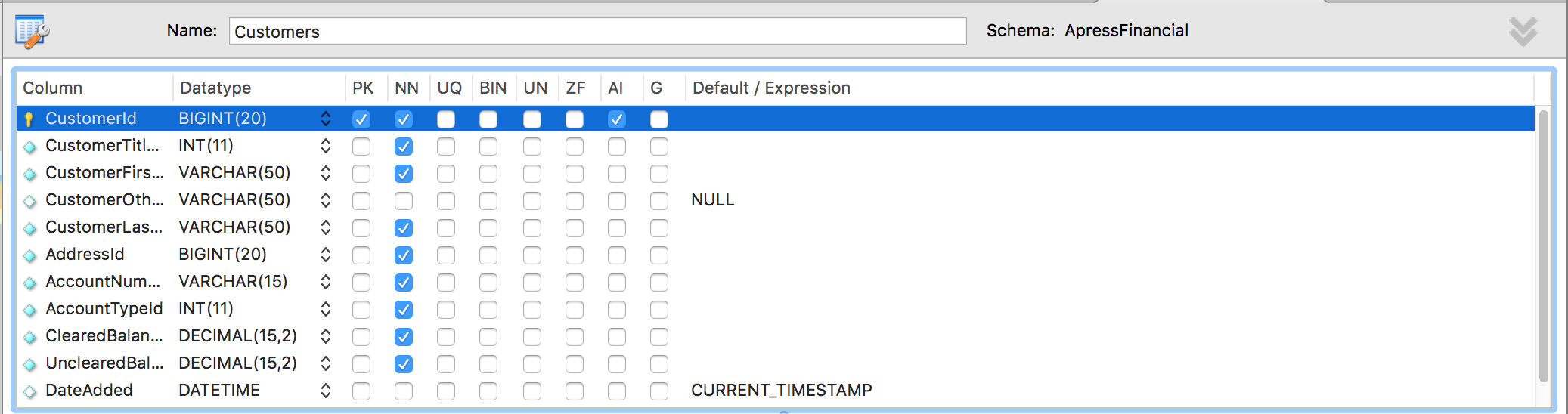
../../../Desktop/Screen%20Shot%202018-04-12%20at%2011.24.33%20P

**Dealing with Several Rows at Once**

In Query Editor, you can input a sequence of SQL commands and execute them as a batch. Each command will be treated as a single unit of work, which either completes or fails.

**Practice: Insert Several Records at Once**

1. Set up **CustomerId** of **Customers** table is Auto Increment



2. Ensure that MySQL Workbench Query Editor is up and running. In the Query Editor window, enter the following code. In this example, several customers will be added through only one INSERT statement.

INSERT INTO Customers

(CustomerTitleId,CustomerFirstName,CustomerOtherInitials,

CustomerLastName,AddressId,AccountNumber,AccountTypeId,

ClearedBalance,UnclearedBalance)

VALUES

(3,'Bernie','I','McGee',314,65368765,1,6653.11,0.00),

(2,'Julie','A','Dewson',2134,81625422,1,53.32,-12.21),

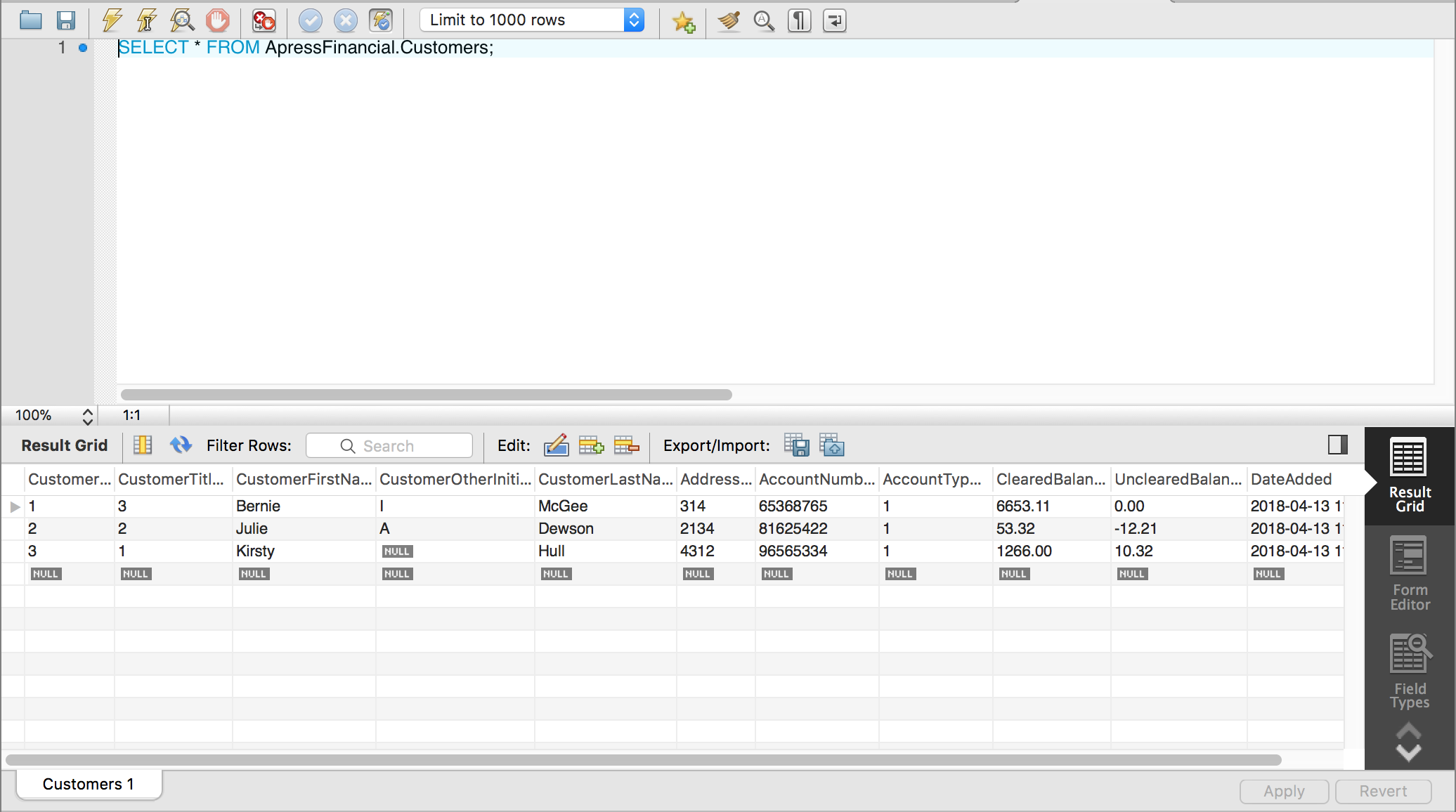
(1,'Kirsty',NULL,'Hull',4312,96565334,1,1266.00,10.32);

3. Now just execute the code in the usual way. You will see the following output in the results pane. This indicates that three rows of information have been inserted into the database, one at a time.

../../../Desktop/Screen%20Shot%202018-04-13%20at%2011.20.36%20A

**Practice: Retrieve Data in SQL Server environment**

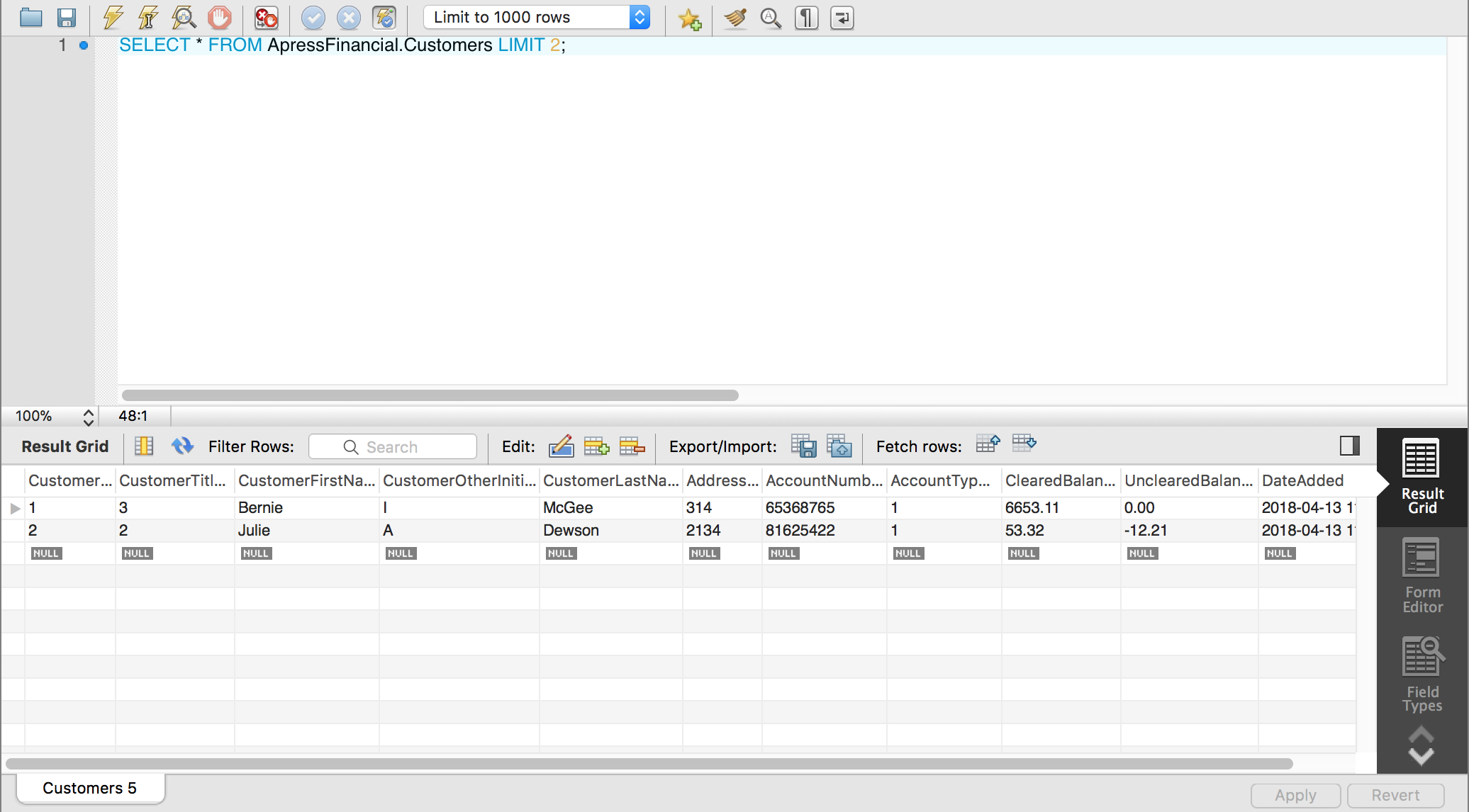
1. Ensure that MySQL Workbench environment is running. Navigate to the **ApressFinancial** schema and click the Tables node; this should then list all the tables in the left-hand pane. Find the **Customers** table and click ../../../Desktop/Screen%20Shot%202018-04-13%20at%2011.35.02%20A. This instantly opens up a new Query Editor pane like the one in the following figure, which shows all the rows that are in the **Customers** table. But how did MySQL Workbench get this data? Let’s find out.



2. Above the results, you will see the SELECT MySQL statement used to return the data. The SELECT statement will return the top 1,000 rows if you add LIMIT <number of row>.

SELECT \* FROM ApressFinancial.Customers LIMIT 1000;

3. You can alter the number next to the top clause if you want to return a smaller or a greater number of rows, and then re-execute the query. For this first time, alter this to 2, and you should see something similar to the following figure. This will return a maximum of two rows.



**The SELECT Statement**

SELECT

[ALL | DISTINCT | DISTINCTROW ]

[HIGH\_PRIORITY]

[STRAIGHT\_JOIN]

[SQL\_SMALL\_RESULT] [SQL\_BIG\_RESULT] [SQL\_BUFFER\_RESULT]

[SQL\_CACHE | SQL\_NO\_CACHE] [SQL\_CALC\_FOUND\_ROWS]

*select\_expr* [, *select\_expr* ...]

[FROM *table\_references*

[PARTITION *partition\_list*]

[WHERE *where\_condition*]

[GROUP BY {*col\_name* | *expr* | *position*}

[ASC | DESC], ... [WITH ROLLUP]]

[HAVING *where\_condition*]

[ORDER BY {*col\_name* | *expr* | *position*}

[ASC | DESC], ...]

[LIMIT {[*offset*,] *row\_count* | *row\_count* OFFSET *offset*}]

[PROCEDURE *procedure\_name*(*argument\_list*)]

[INTO OUTFILE '*file\_name*'

[CHARACTER SET *charset\_name*]

*export\_options*

| INTO DUMPFILE '*file\_name*'

| INTO *var\_name* [, *var\_name*]]

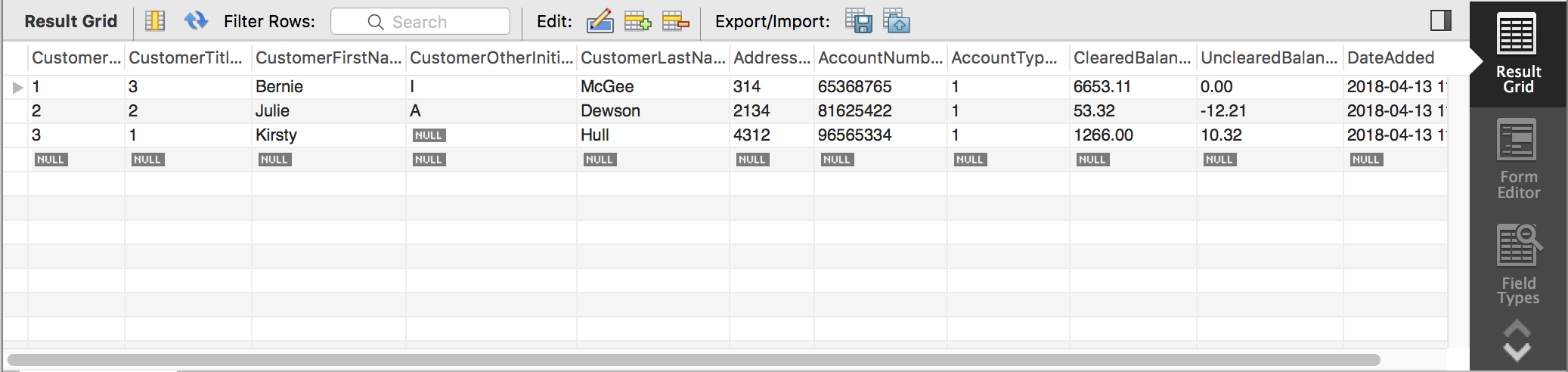
[FOR UPDATE | LOCK IN SHARE MODE]]

**Practice: The First Set of Searches**

1. Ensure that Query Editor is running and that you are within the **ApressFinancial** schema. In the Query Editor pane, enter the following SQL code:

SELECT \* FROM ApressFinancial.Customers;

2. Execute the code. You should then see something like the results shown in the following figure.

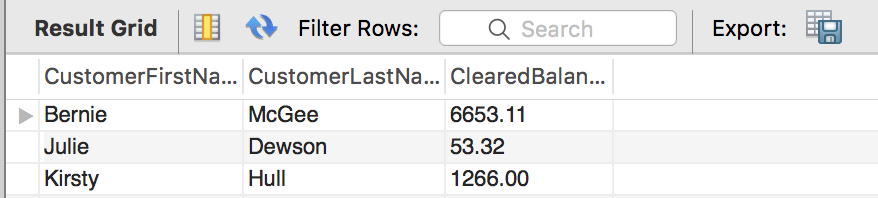


3. This is a simple SELECT statement returning all the columns and all the rows from the **Customers** table. Let’s now take it to the next stage where specific column names will be defined in the query, which is a much cleaner solution. In this instance from the **Customers** table, we would like to return a customer’s first name, last name, and current account balances. This would mean naming **CustomerFirstName**, **CustomerLastName**, and **ClearedBalance** as the column names in the query. The code will read as follows:

SELECT CustomerFirstName, CustomerLastName, ClearedBalance

FROM ApressFinancial.Customers;

4. Now execute this code, which will return the results shown in the following figure. As you can see, not every column is returned.



5. As you have seen from the examples so far, the column names, although well named from a design viewpoint, are not exactly suitable if we had to give this to a set of users. Using the same query as before, a couple of minor modifications are required to give the columns aliases. The first alias name is in quotes as it contains a space. Notice the last column also does not have AS specified because this keyword is optional.

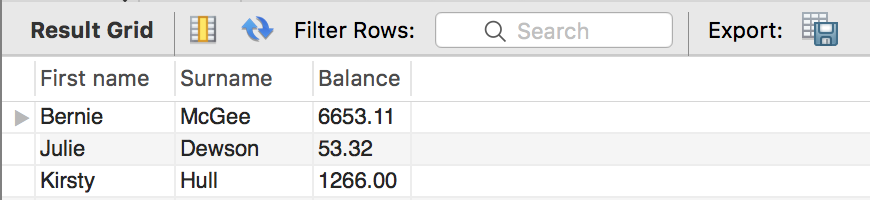
SELECT CustomerFirstName AS 'First name',

CustomerLastName AS 'Surname',

ClearedBalance AS 'Balance'

FROM ApressFinancial.Customers;

6. Execute this and the displayed output changes—much more friendly column names, as you see in the following figure.



**Practice: Limiting the Search: The Use of WHERE**

1. First of all to use a different table, let’s enter some more rows in to the **Shares** table. Enter and execute the following code:

INSERT INTO Shares

(ShareDesc, ShareTickerId,CurrentPrice)

VALUES

('FAT-BELLY.COM','FBC',45.2),

('NetRadio Inc','NRI',29.79),

('Texas Oil Industries','TOI',0.455),

('London Bridge Club','LBC',1.46);

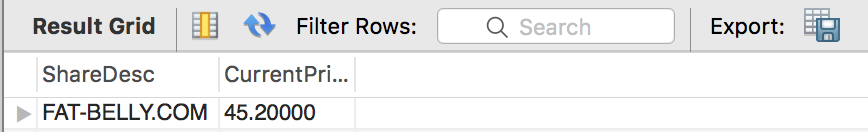
2. The requirement for this section is to find the current share price for FAT-BELLY.COM. We restrict the SELECT statement so that only the specific row comes back by using the WHERE statement, as can be seen in the following code:

SELECT ShareDesc,CurrentPrice

FROM Shares

WHERE ShareDesc = 'FAT-BELLY.COM';

3. Execute this code, and you will see that the single row for FAT-BELLY.COM is returned, as shown in the following figure.



4. To prove that we are working within an installation that is not case sensitive from a data perspective (unless you installed a different collation sequence to that described in Chapter 1), if you perform the following query, you will get the same results as displayed in the previous figure.

SELECT ShareDesc,CurrentPrice

FROM Shares

WHERE ShareDesc = 'FAT-BELLY.COm';

5. You have seen the WHERE in action using the equals sign; it is also possible to use the other relational operations in the WHERE statement. The next query demonstrates how SSE takes the WHERE condition and starts returning rows after the given point. This query provides an interesting set of results. Enter the code as detailed here:

SELECT ShareDesc,CurrentPrice

FROM Shares

WHERE ShareDesc > 'FAT-BELLY.COM'

AND ShareDesc < 'TEXAS OIL INDUSTRIES';

6. Once done, execute the code and check the results, which should resemble the following figure.



7. Let’s now bring in another option in the WHERE statement that allows us to avoid returning specific rows. This can be achieved in one of two ways: the first is by using the less than and greater than signs; the second is by using the NOT operator. Enter the following code, which will return all rows except FAT-BELLY.COM. Run both sets of code at once. This will return two sets of output, known as multiple result sets.

SELECT ShareDesc,CurrentPrice

FROM Shares

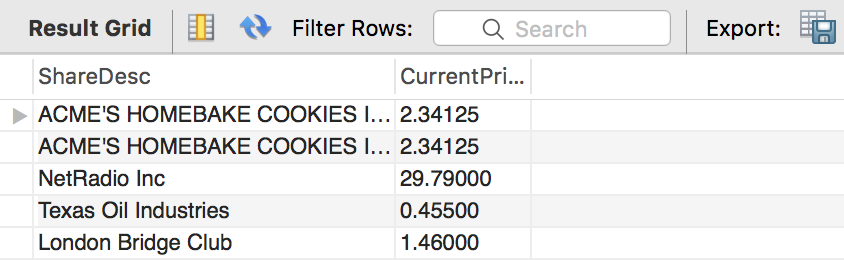
WHERE ShareDesc <> 'FAT-BELLY.COM';

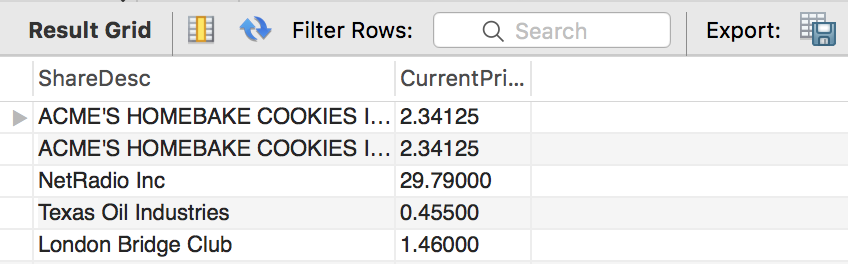
SELECT ShareDesc,CurrentPrice

FROM Shares

WHERE NOT ShareDesc = 'FAT-BELLY.COM';

8. Executing this code will produce the output shown in the following figure. Notice how in neither set of output FAT-BELLY.COM has been listed.





**End of Lab 2**