

LAB 2

Extract, Transform, Load (ETL) Processing

Using the SQL Language to Create New Database Objects and Insert Data from a Source Database

Overview

In this lab you will be creating an extract – transform – load (ETL) process that executes entirely on the SQL relational database management system (RDBMS) platform. In other words, you will not be using any of the many dedicated ETL tools on the market today. Rather, you will learn to complete this task the *old-fashioned* way; using structured query language (SQL) statements. Taking this approach will extend your knowledge and increase your familiarity with the SQL language by teaching you to **create** new database objects using data definition language (DDL) statements, and by way of learning new data manipulation language (DML) statements that are used to **insert** data into those new objects.

In the first steps, you will create a new dimensional model, otherwise known as a star-schema, that's designed to reorganize and store basic retail sales transaction data sourced from the **Northwind** database you worked with in the previous lab. As discussed in lecture, a dimensional model is a specialize database schema design approach that's optimized for read performance and for easy interaction by business users. Dimensional models are optimized for facilitating a low volume of read transactions, each of which may affect a large number of observations (rows) in the base tables. This is achieved by selectively de-normalizing the schema of the source database to reduce its complexity; i.e., reduce the number of its tables. Carried to its logical end, this process results in a schema design featuring only one *dimension* table dedicated to each business entity (e.g., customer, product, employee), and one central *fact* table dedicated to storing the numerical values involved in the business process being modeled (e.g., sales transactions, purchase orders). The fact table also stores foreign-key references to each of the dimension tables. This design enables the aggregation of the numerical values (e.g., sum, count, minimum, maximum, average) so that they may be filtered and/or grouped by members of any of the dimensions. For example, the user can easily view *total sales amount by customer for the previous month*, or *average sales transactions per week by employee (sales person)*.

Exercise 1.0 – Creating the Dimensional Schema

In the first part of this exercise, you will use SQL statements to create a new database (schema) and *dimension* tables for the Customer, Employee, Product and Shipper entities, and then in the second part you will create one *fact* table to model the Orders process. To expedite this process, and to prevent any typographical errors that would result in a frustrating de-bugging effort, you should take advantage of

the code-generation capability inherent to most modern database system management interfaces like MySQL Workbench, SQL Server Management Studio, and Azure Data Studio; just to name a few.

Figure 1 illustrates that the *Create Statement* command is accessible through the context menu by right-clicking any database object (database, table, view, stored procedure, or function) in the schema navigator of MySQL Workbench. This function can be accessed through either the *Copy to Clipboard* or the *Send to SQL Editor* sub-menus.

A similar method is available in other database management interfaces like SQL Server Management Studio).

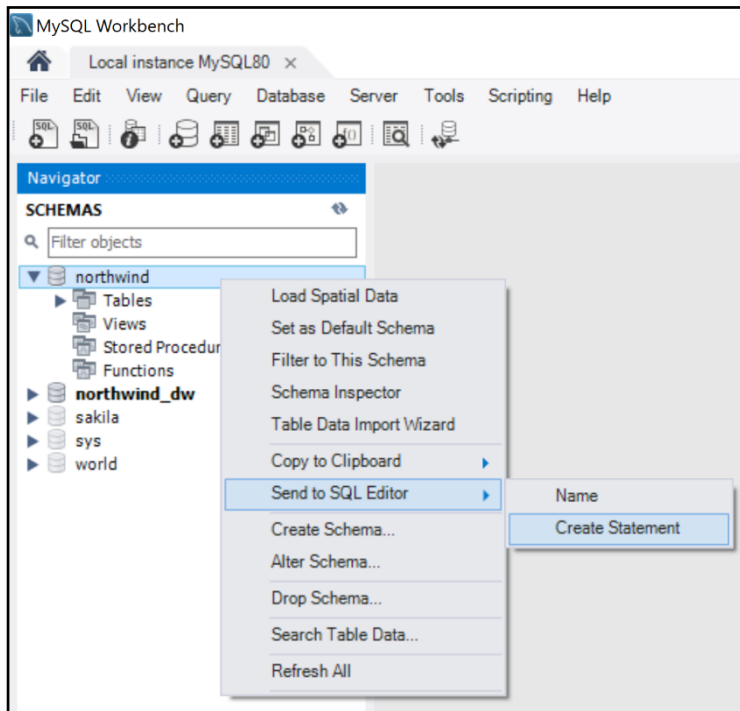


Figure 1 - Sending a Create Statement to the SQL Editor

Using this method, right-click the Northwind database (schema) to generate the CREATE DATABASE statement required to create your new database (schema). Rename the database to **Northwind_DW**.

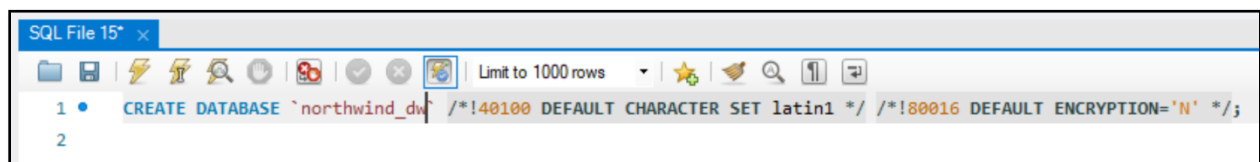


Figure 2 - Editing the CREATE DATABASE statement

Exercise 1.1 – Creating the Dimension Tables

Using the method described in the preceding section, right-click each of the following tables in the Northwind database to generate the CREATE TABLE statements required to develop your new dimension tables: *customers*, *employees*, *products* and *shippers*. The command will inject the SQL statements, one after the other, into the query window that was opened by the CREATE DATABASE

statement. In the statements you generate, first change the name of each table to ***dim_customers***, ***dim_employees***, ***dim_products*** and ***dim_shippers***. Also, to conform with customary star-schema naming conventions, rename the PRIMARY KEY column of each table to ***customer_key***, ***employee_key***, ***product_key***, ***shipper_key***, respectively. Next, remove any unnecessary columns from each CREATE TABLE statement, keeping only a subset of the columns listed in each table to be included in your new *dimension* tables.

Tip: Columns containing binary object data (e.g., images, attachments) and/or free-text are typically not well suited to online analytical processing (OLAP) databases. What's more, columns that are very sparse (i.e., have many rows containing NULL values) are of very little use, and may also be omitted.

Exercise 1.2 – Creating the Fact Table

Continue using the code-generation method described above to produce the CREATE TABLE statements needed to combine columns from the *orders*, *orders_status*, *order_details*, and *order_details_status* tables into a new table named ***fact_orders***.

Notice that the *orders* table maintains a many-to-one relationship with the *orders_status* table, and the *order_details* table maintains a many-to-one relationship with the *order_details_status* table. The *orders* table references the *orders_status* table by way of its *status_id* column, and the *order_details* table references the *order_details_status* table by way of its own *status_id* column. In other words, since each order and each order detail can have one of many statuses, the *orders* and *order_details* tables each contain many references to rows in the *orders_status* and *order_details_status* tables; respectively. Also, notice that the *orders* table maintains a one-to-many relationship with the *order_details* table by way of the *order_id* column in the *order_details* table. In other words, since one order can have many order details, the *order_details* table may contain multiple references to each row in the *orders* table.

Remember that the objective is to replace the *status_id* column in the *orders* and *order_details* tables with the *status_name* column from the *orders_status*, and *order_details_status* tables; respectively. Because both *status_name* columns will eventually coexist in the *fact_orders* table, you must rename at least one of them to disambiguate them, and to provide an accurate semantic description of their purpose and origin (e.g., *orders_status_name* and *order_details_status_name*).

Having removed the complexity associated with the *orders* table referencing the *orders_status* table, and with the *order_details* table referencing the *order_details_status* table (reducing the table count from 4 to 2), we can now focus on eliminating the complexity associated with the *order_details* table referencing the *orders* table; thereby, unifying the data into the ***fact_orders*** table. The object here is to integrate the product information (e.g., product, quantity, unit_price, discount) from the *order_details* table with the order header information from the *orders* table (e.g., customer, employee, shipper).

Exercise 2.0 – Populating the Newly-Created Dimensional Schema

In this exercise, you will *select* data *from* the **Northwind** database and, making any necessary transformations, you will then *insert* that data *into* the new **Northwind_DW** database. One of the most efficient methods for loading data into an RDBMS is to pass a *tabular result set* (i.e., a table of data in memory) directly to a SQL INSERT INTO statement. Since RDBMS like Oracle, MySQL, and Microsoft SQL Server are optimized to perform *set-based* operations this construct will load all the rows of data in the tabular result set into the destination table within a single transaction; thereby minimizing the overhead of managing multiple transactions and recording those operations in the RDBMS transaction log. To learn more about the transaction log see: [The Transaction Log \(SQL Server\) | Microsoft Docs](#).

Exercise 2.1 – Loading the Dimension Tables

Using the code-generation feature of MySQL Workbench, right-click each of the newly created dimension tables in the **Northwind_DW** database, and select *Insert Statement* from the context menu to generate the INSERT INTO statement. Next, for each dimension table's *Insert Into* statement, remove

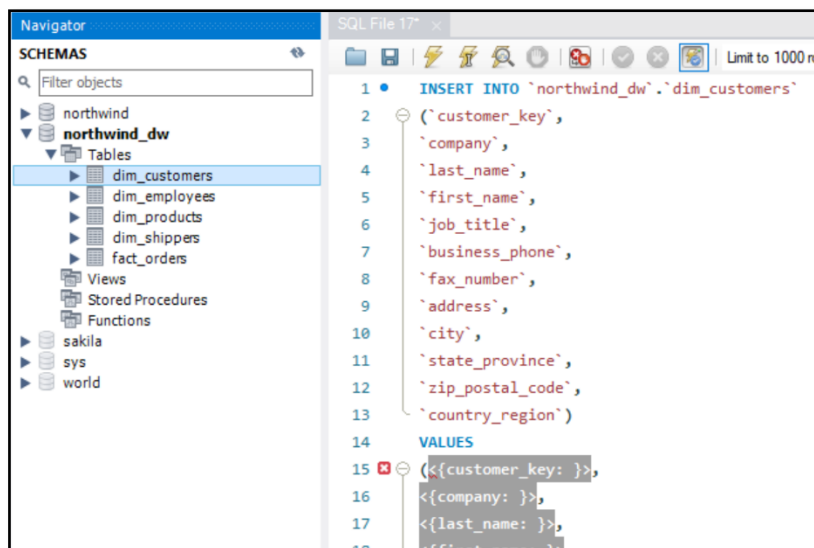


Figure 3 - The generated INSERT INTO statement

the entire VALUES(...) portion, and then right-click each of the corresponding source tables in the **Northwind** database, and choose *Select All Statement* from the context menu to send a select statement to the SQL Editor window. Finally, remove any column names from the SELECT statement that do not appear in column list of the INSERT INTO statement.

Figure 3 shows the *Insert Into* statement that was generated.

Exercise 2.2 – Loading the Orders Fact Table

With the dimension tables loaded, you now understand how data can be *extracted* from a source system using a simple SELECT statement, and how the resulting tabular result set can then be *loaded* into a destination table by simply passing it to an INSERT INTO statement. Any *transformations* that are deemed necessary can be accomplished by carefully crafting the SELECT statement. In our simple example thus far, the extent of our transformations has been to simply omit certain columns from the select list, but the task of loading our fact table will require a bit more effort.

By way of completing **Exercise 1.1**, you learned a great deal about the structure of the source tables involved in the Orders business process. You learned how *order statuses* relate to *orders*, and how *order details statuses* relate to *order details*. You also learned how *order details* relate to *orders*. This

information provides critical input towards how you will need to craft the *select* statement used to *load* data from all four of these tables into the Orders fact table. Remember that you'll need to *join* these four tables in such a manner that reflects the one-to-many relationships that exist between them, along with the directionality of those relationships.

*Tip: Remember that an **inner join** will return all rows from both tables that satisfy the condition(s) specified by the **on** clause, that a **left outer join** will return all rows from the left (first) table along with rows from the right (second) table that satisfy the condition(s) specified by the **on** clause, and that a **right outer join** will return all rows from the right (second) table along with rows from the left (first) table that satisfy the condition(s) specified by the **on** clause. These mechanics provide a concise and deterministic means for expressing the one-to-many relationships that exist between the tables involved in the **select** statement, as well as the directionality of those relationships.*

Here you will implement techniques you learned while loading the *dimension* tables. Using the code generation feature of MySQL Workbench, right-click the *fact_orders* table in the **Northwind_DW** database, and then select *Insert Statement* from the context menu to generate the INSERT INTO statement for the *fact_orders* table. Then, carefully craft an appropriate SELECT statement that *extracts* the required data from the *orders*, *orders_status*, *order_details*, and *order_details_status* tables, and use it to replace the VALUES(...) clause of the INSERT statement.

Exercise 3.0 – Creating a Dimensional Report

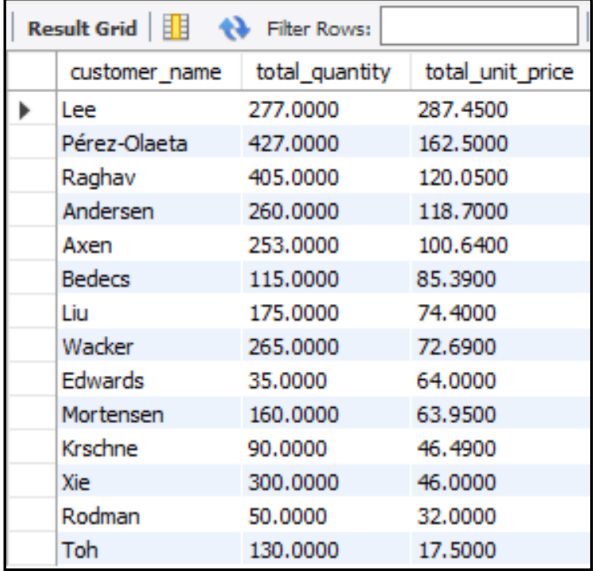
Having completed Exercises 1.0 and 2.0, you should now have a new dimensional database named Northwind_DW that contains four (4) *dimension* tables named *dim_customers*, *dim_employees*, *dim_products*, and *dim_shippers* along with one (1) *fact* table named *fact_orders*.

These tables should all contain data you *extracted* from the **Northwind** database.

To demonstrate the viability of your solution, author a SQL SELECT statement that returns:

- Each Customer's Last Name
- The total amount of the order quantity associated with each customer
- The total amount of the order unit price associated with each customer

Your result should look something like Figure 4.



	customer_name	total_quantity	total_unit_price
▶	Lee	277.0000	287.4500
	Pérez-Olaeta	427.0000	162.5000
	Raghav	405.0000	120.0500
	Andersen	260.0000	118.7000
	Axen	253.0000	100.6400
	Bedecs	115.0000	85.3900
	Liu	175.0000	74.4000
	Wacker	265.0000	72.6900
	Edwards	35.0000	64.0000
	Mortensen	160.0000	63.9500
	Krschne	90.0000	46.4900
	Xie	300.0000	46.0000
	Rodman	50.0000	32.0000
	Toh	130.0000	17.5000

Figure 4 - Result of your SQL Query