Module 03 Notes

Data Movement – Part 1 (ETL Processing)

# ****Module Overview****

In this module, you learn about how **data is moved between relational databases**. This process involves **extracting source data, transforming** that data, and loading data into its destination. We will use an **example of a data warehouse**, but the **techniques shown are applicable to many scenarios**. This module will also **expand your knowledge** of how **stored** **procedures** are created and used in the real world!

# ****Required Software****

Install SQL Developer Edition (or equivalent), Visual Studio 2019 Community edition, and Report Builder.

# Assignment

Each week you have an assignment to perform. Let us review this week's assignment.

# Source Data

The first thing you need to perform any ETL process is a source of data. Data sources can be anything from text files to relational databases, or non-relational databases and beyond. Of course, the **more exotic the source is the more difficult it can be to access its data.**

You have likely had some experience with **importing data using tools like BCP or Bulk insert, both of which import text file data.** Text file imports are **best when the source and destination are very dissimilar**. That is because **text files provide a common data format using a globally agreed-upon "code pages."** These code pages map characters and symbols to binary values that can **easily be written by one system and read by another.**

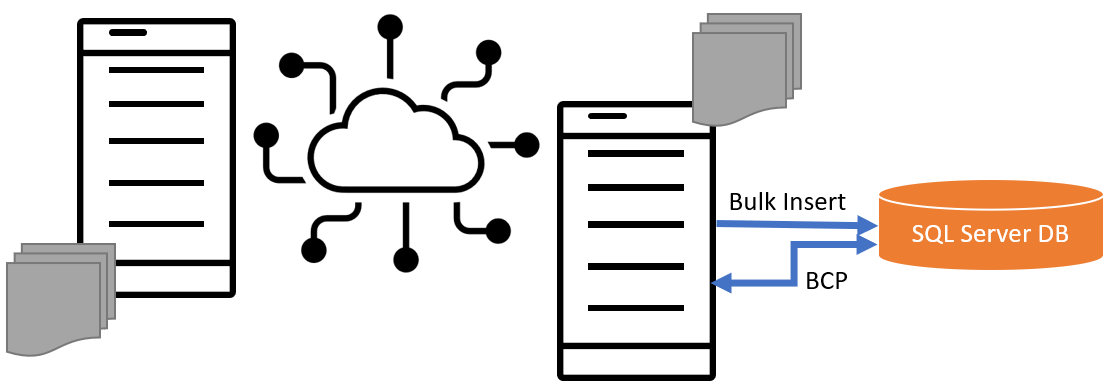


Figure. Importing text data into a SQL server

Often you will load data into a **"staging area" before loading it into its destination**. This **staging** area hold the data in its **original format** and may need to **transform**. Common transformations include **sorting, filtering, calculating, and formatting.**

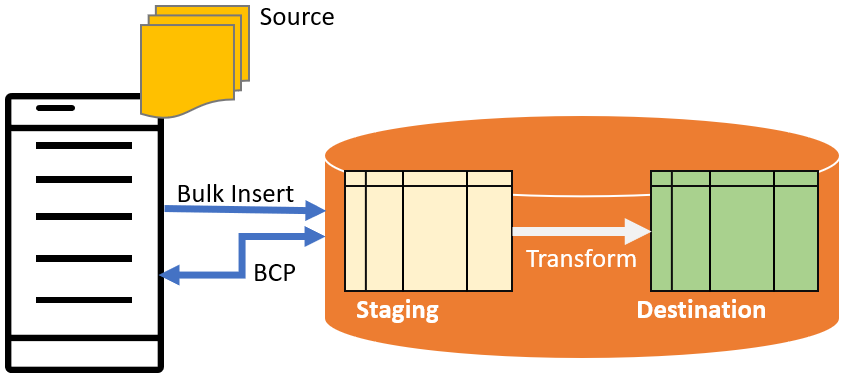


Figure. Using a staging table

## Demonstration – Creating the Source database

In this demo, you review a SQL script that creates a source database and fills it with data generated on the internet and from an existing relational database.

Files needed:

* 0\_InstPubs.sql
* 1\_SourceDatabase.sql

# Destination Data

Transformed data needs to be store at a destination. In a relational database, the destination is a set of one or more tables.

When that data is being stored for reporting purposes it may be held in **reporting tables**. These reporting tables are **often designed using "Dimensional Modeling."** A database designed using dimensional modeling is commonly called a data warehouse.

# Data Warehouses

A data warehouse is a collection of **data objects designed for the easy extraction of information**. **Relational databases** are **frequently used** for this purpose, but data warehouses can be in any form, including a series of text ﬁles. You can think of a data warehouse as a reporting database.

## Data Mart

* A data mart is also a **collection of data for a given topic**.
* **More speciﬁc** than that of a data warehouse.
* **Created for a particular process**, such as a **sales event or taking inventory**
* **Designed around departments** within the company
* Typically, **better** off deﬁning the data mart based on a **process, not a department**

## Design Patterns of Data

The three basic **patterns of data regardless of its form** are; **one-to-one, one-to-many, and many-to-many.** In a relational database the pattens happen **between columns and tables.**

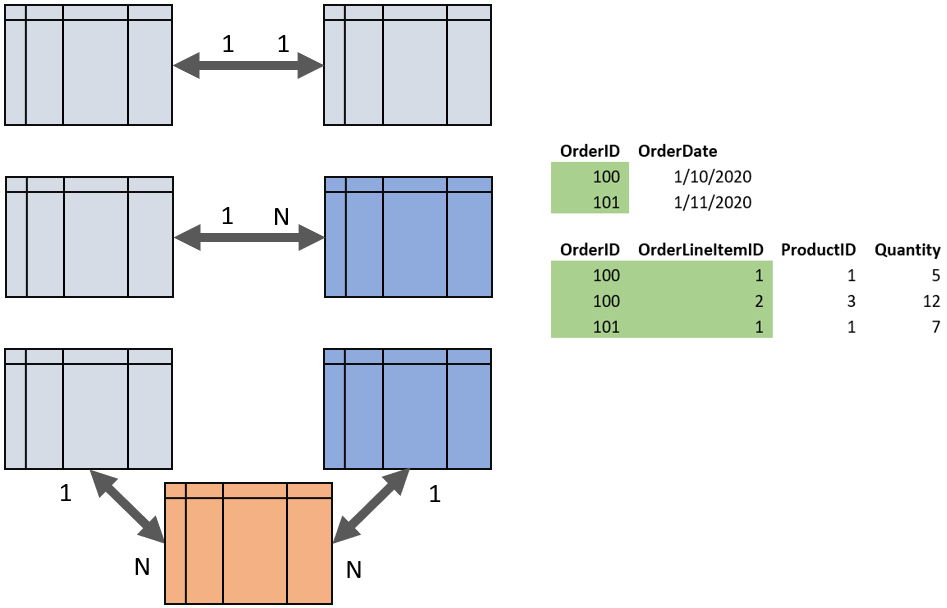


Figure. Relationships between column and table data

Here are some **basic facts** you should know as a database professional:

* Data Warehouses store **reporting data**.
* Data Warehouse Design **(OLAP) design is like OLTP**, but its focus is different
* The focus is on **report performance and simplicity**
* It should be **easy to understand** and consist of a **minimal set of tables** wherever possible
* **Make things consistent** so that **additional processes can be added later**, in what Kimball refers to as a bus architecture.
* **The OLAP designs usually follow the dimensional modeling patten**

Here is an example of both an OLTP database and its OLAP counterpoint:

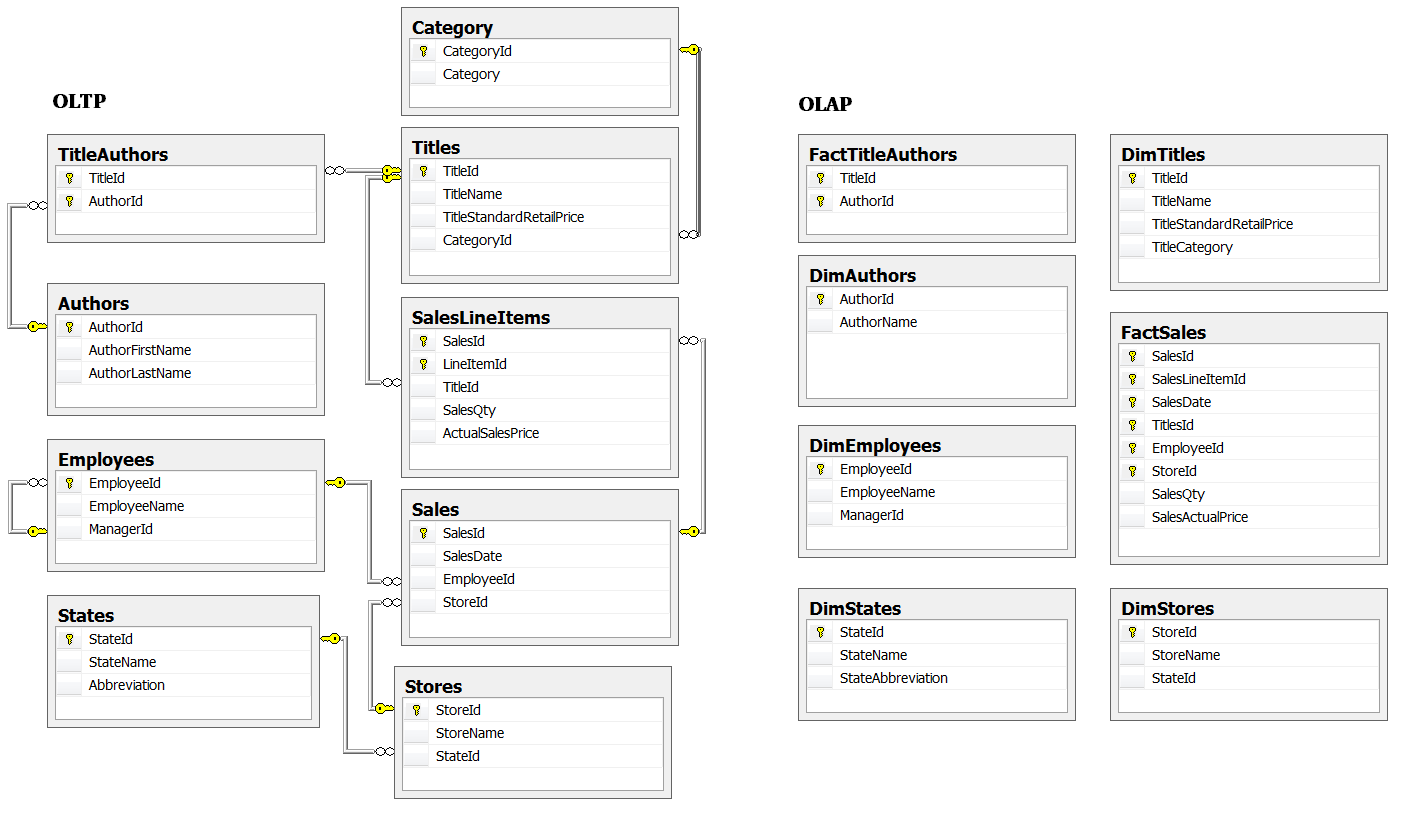


Figure. An OLTP and OPAP database diagram

## Measures

In dimensional modeling, Measures are **numeric values used to calculate aggregates for reporting** on a given process (ex. selling something)

* Always **held in a Fact Table**
* **Numeric values in dimension tables are not measures**
* Measure values **typically represent the lowest level of detail** tracked by a given process.
* The lowest level of detail **deﬁnes the granularity** of a measure in a data mart

## Fact Tables

Fact table tables contain **Zero or More measures** about a **process or a relationship.**

* **Process** fact tableswill likely have a **DateTime attribute directly associated with its measures.**
* **Look like an OLTP Event table**, but may combine data from multiple tables (ex. **denormalized** sales data)
* Typically have the **prefix "Fact"** in their name
* **Relationship** fact tables **seldom include measure** values.

## Dimension Tables

* Dimension tables **contain dimension attributes**
* Each dimension table should **always** contain **a dimensional key column and a dimensional name** **column**
* **Look like an OLTP subject table**, but may combine data from multiple tables (ex. **denormalized** titles data)
* Typically have the **prefix "Dim"** in their name

**Question:** In the following image, which columns are dimensional attributes, and which are measures?

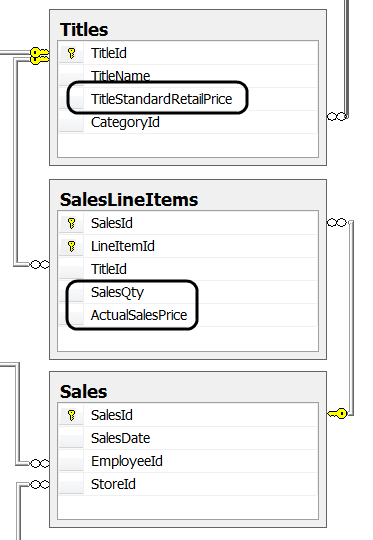


Figure. Dimensional attributes and measures examples

In reporting databases, like a data warehouse, both one-to-one relations and one-to-many relations may exist between columns. This leads to redundancy, which increases storage space to increase SELECT performance.

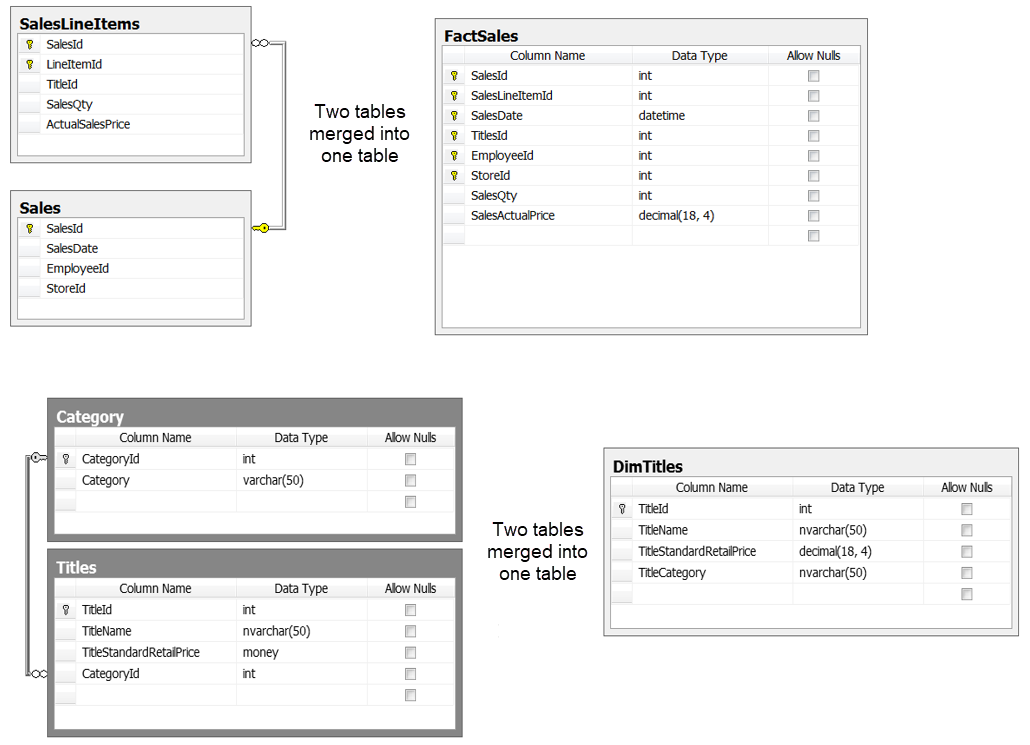


Figure. Multiple tables are often merged in a reporting database

## Single Table and Multiple Table dimensions

* **When all dimension data is in a Single table, that** dimension is **called a "star"** design pattern
* **When dimension data is in Multiple tables, that** dimension is **called a "snowﬂake"** design pattern
* **A data mart may have both star and snowflake dimensions**

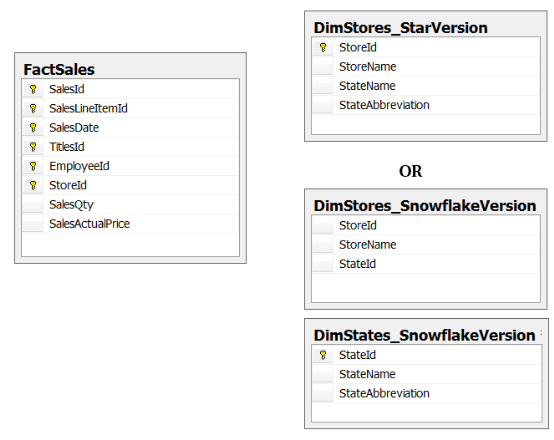


Figure. Single (star) and Multi (snowflake) table dimensional designs

**Question: What are the dimensions in this next image, and which dimensions are star or snowflake?**

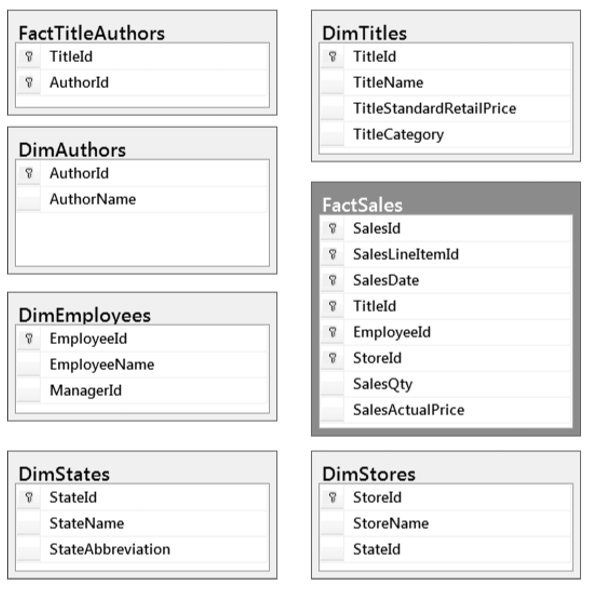


Figure. A typical OLAP database design

## Dimensional Patterns

* A dimension table's pattern **describes its relationship to a fact table**
* Choosing an **incorrect pattern may cause incorrect report data**
* Patterns are **easy to recognize** once you know what to look for

### Standard Dimensions

* A collection of **one or more tables linked directly to the primary fact table**
* The standard dimension is **seen most often**, which is why it is called standard
* Each standard dimension **should always have a key attribute and a name attribute**
* In addition to those two columns, you can **provide additional descriptive values** that help further categorize the data
* May be **either a star or snowﬂake design**

### Fact or Degenerate

* Dimensions Fact dimensions (aka degenerate dimensions) have **all their attributes stored in the fact table**

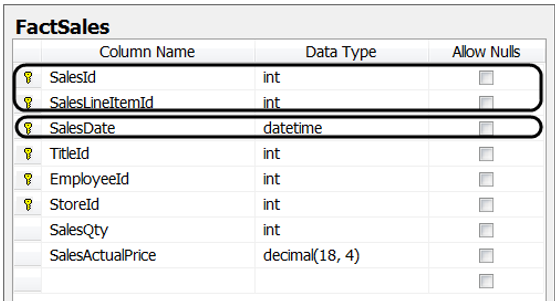


Figure. A fact table with Fact (degenerate) dimensions

### Date and Time Dimensions

* **You should create a date dimension table**
* A Date table **always** **includes a date key and a date name** but **may include month, quarter, and year**
* May also include **additional attributes such as holidays, corporate events, or ﬁscal weeks**
* Use **either date or integer** data types

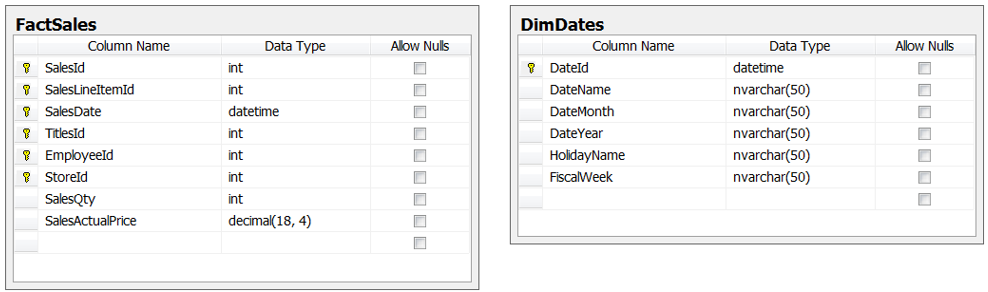


Figure. A fact table with a separate date dimension table

* A DateTime column may need to be divided in both Time and Date. If you want to report based on **hours, minutes, and seconds**, you should create a **separate "Time" table.** A separate time and date dimension table would look like this.

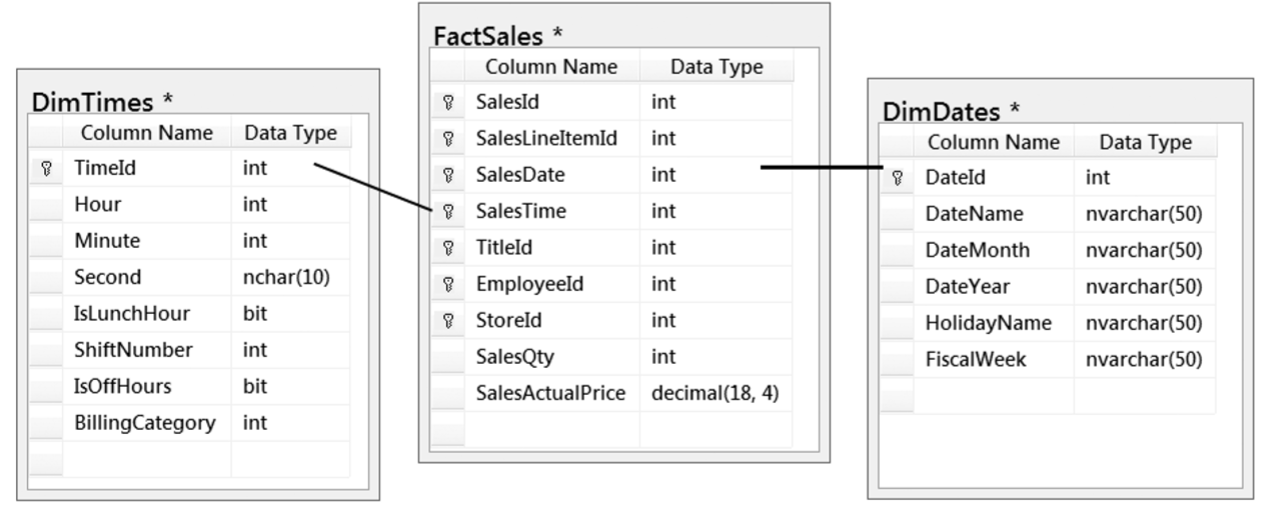


Figure. A time dimension and date dimension table connected to a fact table

### Role-Playing Dimensions

* Role-playing dimension tables are **used repeatedly for slightly different purposes** (roles)
* The **classic example** of a role-playing dimension is a **date or time dimension** that link to a fact table many times

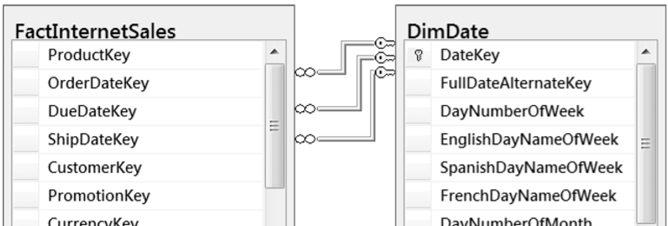
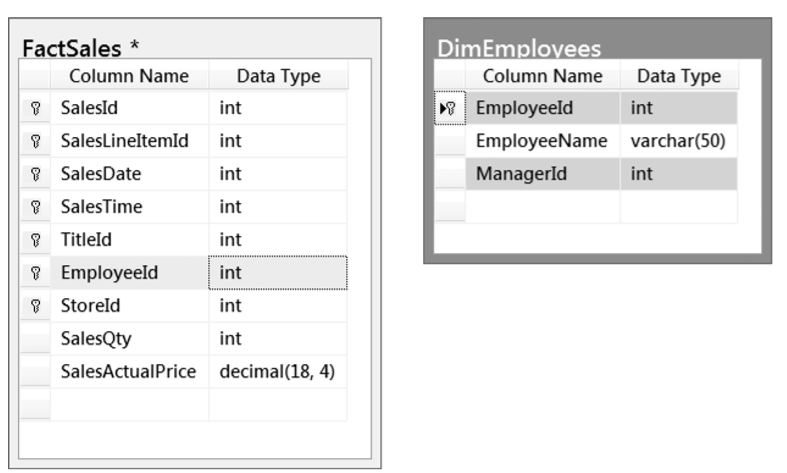


Figure. A date dimension "playing" multiple roles with the fact table

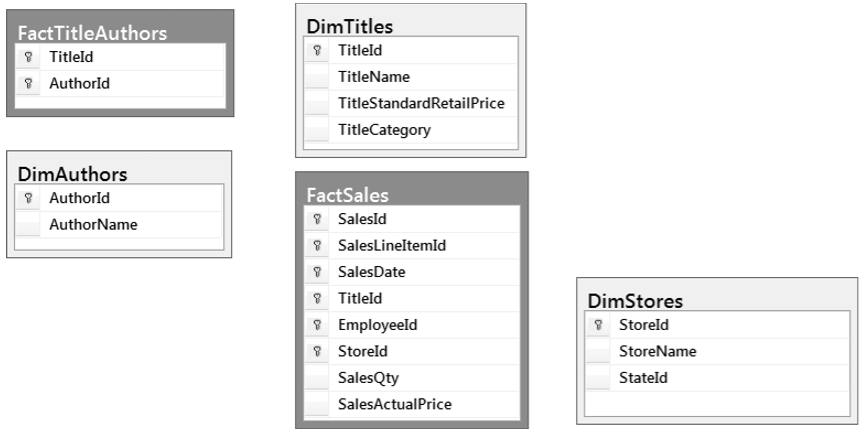
### Parent-Child Dimensions

* Parent-Child dimension tables **look like their OLTP** counterparts
* an **employee** table is a **classic example** of a **parent-child** dimension



### Many-to-Many Dimensions

* **Use a bridge table** to deﬁne the many-to-many relationship
* **Are a "Fact" table**, but not the "Primary" fact table of the data mart
* You have to **go through the bridge table and another dimension table** to get **to the primary fact table**



### Conformed Dimensions

* Dimensions that **used from multiple data marts** within a data warehouse
* Used by **different fact tables**, each based on a **different subject** (Sales vs. Inventory)

## Demonstration – Creating the DW database

In this demo, you review a SQL script that creates a data warehouse, destination, relational database.

Files needed:

* 2\_Assignment03MetadataWorksheet.xlsx
* 5\_Assignment03DWDatabase.sql

# Creating an ETL Process

**E**xtracting the **data** from its **original** location, **T**ransforming **the data to be consistent with your new data** warehouse design, and **L**oading **the data into** the new **data warehouse** location, is known as **ETL processing**. You create an ETL process using a **combination** **of** programming, such as **SQL, Java, Python, C#**, etc., and/or **specialized tools,** such as SQL Server's Integration Server (**SSIS**).

Often, you **use SQL code to do the bulk of your ETL work**, since it is specifically designed to manipulate large amounts of data. Here is an example of some simplistic ETL code (In real life this code would be much more complex).

## Creating ETL Objects

Code that you want to use repeatedly should be save in views, functions, or stored procedures. Since ETL code is use often, you should **create** these **database object to support this code**. Here are some examples:

Go

Create Or Alter View vETLDimStores

As

Select

[StoreId] = Cast( stor\_id as nChar(4) )

, [StoreName] = Cast( stor\_name as nVarchar(50) )

, [StoreCity] = Cast( city as nVarchar(50) )

, [StoreState] = Cast( state as nVarchar(50) )

From IndependentBookSellers.dbo.stores;

Go

Create Or Alter Proc pETLDimStores

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--

-- Desc: This Sproc fills the DimStores table.

-- Change Log: When,Who,What

-- 2020-01-01,RRoot,Created Sproc

--\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*--

As

Begin

Begin Try

Declare @Message varchar(1000);

Begin Tran;

Insert Into DimStores

(StoreID, StoreName, StoreCity, StoreState)

Select

[StoreId]

,[StoreName]

,[StoreCity]

,[StoreState]

From vETLDimStores;

Commit Tran;

End Try

Begin Catch

Print Error\_Message(); -- Note that this is presentation code!

End Catch

End

Go

-- Select \* From DimStores; Select \* From vETLMetadata

## Creating the ETL Logging

Logging thing that is often overlooked when creating automation tasks. However, including it allows you to track and report on the automation process. Besides, it can be created with only a small amount of effort. Here is an example:

If NOT Exists(Select \* From Sys.tables where Name = 'ETLMetadata')

Create Table ETLMetadata

(ETLMetadataID int identity Primary Key

,ETLDateAndTime datetime Default GetDate()

,ETLAction varchar(100)

,ETLMetadata varchar(2000)

);

go

Create or Alter View vETLMetadata

As

Select

ETLMetadataID

,ETLDate = Format(ETLDateAndTime, 'D', 'en-us')

,ETLTime = Format(Cast(ETLDateAndTime as datetime2), 'HH:mm', 'en-us')

,ETLAction

,ETLMetadata

From ETLMetadata;

go

## ETL Stored Procedures

Any stored procedure used in an ETL process is an ETL stored procedure. **Stored procedures provide flexibility and greater modularity** when compared to using a simple ETL script. Let's get started with a quick **review of what you have learned so far before we look at how to create these stored procedures.**

### Variables and Stored Procedures

All code in a stored procedure is in one batch. A stored procedure **may have thousands of statements**, but they all must run as a **single batch**! This means that any variable created in a stored procedure only exists while that procedure runs, for afterward that batch has ended.

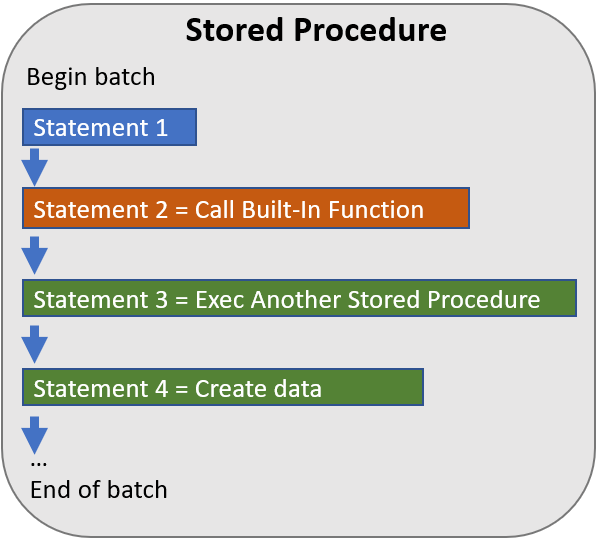


Figure. A representation of an executing the stored procedure's batch of code

For example, if I declare the variable **@Status** in a stored procedure, it will **not be available after** the stored procedure ends. So, trying to access it will cause an error!

CREATE OR ALTER PROCEDURE pAddValues

AS

BEGIN

DECLARE @Status int = 0;

END

go

EXEC pAddValues;

PRINT @Status;

Msg 137, Level 15, State 2, Line 505

Must declare the scalar variable "@Status".

**Note:** In a script file, you **cannot use GO inside of a stored procedure** but should **add it after** the procedure's code ends. This will eliminate the chances of other statements being incorrectly included in the stored procedure.

### Variables and Return Codes

Stored procedures use **return codes to indicate the status** of the procedure's tasks. The standard pattern is to return a **positive value if a task is running well or negative if it is not**. For example, if we try to divide by zero an error will occur, so we use a negative number to indicate a "negative" status. The following code will return a negative one when it attempts to divide five by zero.

CREATE OR ALTER PROCEDURE pDivideValues

AS

BEGIN

DECLARE @Status int

DECLARE @Quotient float

BEGIN TRY

SET @Quotient = 5/0;

SET @Status = 1; -- This statement is not reached!

END TRY

BEGIN CATCH

SET @Status = -1

END CATCH

RETURN @Status -- And exit the stored procedure

END

go

Declare @Status int;

Exec @Status = pDivideValues;

Print @Status;

go

/\* Results

-1

\*/

Here are some other **facts** to remember when working with return codes:

* Return codes **return the status** of a Stored Procedure
* Return code values are **always integers**
* Return codes **do not** work well for **returning** most **data** since they must be integers
* Return code values are **not displayed automatically**
* Return code's values are **captured in a "return code" variable**
* Return code variables are used **before the procedures name** when executing that procedure
* **RETURN will "Exit"** a stored procedure immediately
* **Only have one RETURN** command per stored procedure

### Variables and Input Parameters

You can **create variables to hold the argument values** and use these argument values when executing a stored procedure. **Input parameters are just variables used as "parameters."** The term "parameter" describes the task it is performing. The scope of a parameter is the stored procedure's batch. Here is a stored procedure with two parameters.

CREATE OR ALTER PROCEDURE pDivideValues

(@Value1 float, @Value2 float)

AS

BEGIN

DECLARE @Status int

DECLARE @Quotient float

BEGIN TRY

SET @Quotient = @Value1/@Value2;

PRINT @Quotient -- Add a PRINT statement to see the results

SET @Status = 1;

END TRY

BEGIN CATCH

SET @Status = -1

END CATCH

RETURN @Status

END

go

DECLARE @Status int;

EXEC @Status = pDivideValues @Value1 = 10, @Value2 = 3;

PRINT @Status;

go

/\* Results

3.33333

1

\*/

### Input Parameter and Argument Variables

When you execute or "Call" a stored procedure with parameters, **you must pass in "Arguments" (Unless the parameter has a "Default Value")**. You will often see developers create variables to hold these argument values. Since **the code that calls the stored procedure is a separate batch from the code inside of the stored procedure**, your argument **variables and parameters can have the same name**! It may look odd but be prepared to see this since it is a common practice. Here is an example:

DECLARE @Value1 float = 10 , @Value2 float = 3;

DECLARE @Status int;

EXEC @Status = pDivideValues @Value1 = @Value1, @Value2 = @Value2; -- Parameter = Argument

PRINT @Status;

go

/\* Results

3.33333

1

\*/

**Important:** **Variable assignments flow from right to left**, so the argument will always be on the right of the input parameter's name!

### Output Parameters

So far, our examples have printed the calculated values from within the stored procedure but have not returned those values to the calling code. **Using SELECT or PRINT inside of the stored procedure makes it both present and process values**. **Separating the presentation and processing of data is more flexible and maintainable**. For example, you can use multiple presentation strategies without having to change and re-test the processing code. To implement this strategy, we use "output" parameters.

You can declare **one or more output parameters** per stored procedure. However, you must also declare **one or more variables to receive each output parameter's result**. Here is an example of a data processing stored procedure with output parameters:

CREATE OR ALTER PROCEDURE pAlgebraValues

( @Value1 float = 0 -- Input

, @Value2 float = 0 -- Input

, @Sum float = 0 OUTPUT

, @Difference float = 0 OUTPUT

, @Product float = 0 OUTPUT

, @Quotient float = 0 OUTPUT

)

AS

BEGIN

DECLARE @Status int

BEGIN TRY

SET @Sum = @Value1 + @Value2;

SET @Difference = @Value1 - @Value2;

SET @Product = @Value1 \* @Value2;

SET @Quotient = @Value1 / @Value2;

SET @Status = 1;

END TRY

BEGIN CATCH

SET @Status = -1

END CATCH

RETURN @Status -- And exit the stored procedure

END

go

With our data processing code created, we now create presentation code with two input values and capture four output values. We start by **declaring variables to hold the return code, input values, and output value**. Then we call the stored procedure using those variables to either input values or capture output values and the return code. Finally, we **present the results using the return code and output variables.**

The following code shows an example of how to do this. Note that we have named the argument and parameters differently to make the code easier to read, but it is possible to use the same name. **Also note, that you must use the keyword OUTPUT next to the argument values when you execute the stored procedure.** If you forget, the argument values will remain empty!

-- Declare the variables

DECLARE @RC int – Return code

     , @Value1 float = 10 -- Input

, @Value2 float = 3 -- Input

  , @S float -- Output

  , @D float -- Output

  , @P float -- Output

  , @Q float -- Output

-- Call the stored procedure

EXEC @RC = pAlgebraValues @Value1 = @Value1 – Input Parameter <- Argument Variable

                        , @Value2 = @Value2 – Input Parameter <- Argument Variable

                        , @Sum = @S OUTPUT -- Parameter -> Argument Variable

                        , @Difference = @D OUTPUT -- Parameter -> Argument Variable

                        , @Product = @P OUTPUT -- Parameter -> Argument Variable

                        , @Quotient = @Q OUTPUT;  -- Parameter -> Argument Variable

-- Present the results:

SELECT [ReturnCode] = @RC -- Alias <- Variable

     , [Sum] = @S -- Alias <- Variable

     , [Difference] = @D -- Alias <- Variable

     , [Product] = @P -- Alias <- Variable

     , [Quotient] = @Q; -- Alias <- Variable

go

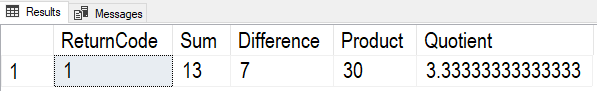


Figure. The results from executing the stored procedure

### Output Parameters and Transactions

**Output parameters can be used to capture the new identity value** of insert stored procedures. This is useful when you want to perform complex transactions processing. For now, let's start with a simple transaction before we try a complex one.

Here is a simple insert stored procedure that inserts data into a Customers table (the table has been greatly simplified for the example). We **use the system variable @@Identity to set the output variable** to the new CustomerID is generated by the IDENTITY keyword. We have added a UNIQUE constraint to the table as well, and we will later use this to demonstrate error handling.

USE TEMPDB;

go

CREATE TABLE Customers

(CustomerID int PRIMARY KEY IDENTITY

,CustomerFirstName nvarchar(100)

,CustomerLastName nvarchar(100)

,CustomerEmail nvarchar(100) UNIQUE

);

go

CREATE OR ALTER PROCEDURE pInsCustomers

(@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

,@NewCustomerID int OUTPUT

)

AS

BEGIN

DECLARE @Status int = 0;

BEGIN TRY

BEGIN TRAN;

INSERT INTO Customers (CustomerFirstName, CustomerLastName, CustomerEmail)

VALUES (@CustomerFirstName, @CustomerLastName, @CustomerEmail);

COMMIT TRAN;

SET @NewCustomerID = @@Identity;

SET @Status = +1;

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION

SET @Status = -1;

END CATCH

RETURN @Status;

END

go

With the stored procedure created, we once again execute it using argument variables. However, this **time we have included an output argument variable to capture the newly generated customer ID.**

-- Call the stored procedure

DECLARE @Status int;

DECLARE @NewID int;

EXEC @Status = pInsCustomers

@CustomerFirstName = 'Victor'

,@CustomerLastName = 'Vu'

,@CustomerEmail = 'VictorVu@MyCo.com'

,@NewCustomerID = @NewID OUTPUT;

-- Present the results

SELECT @NewID as [New CustomerID]

SELECT @Status as [Return Code Value]

SELECT CASE @Status

WHEN +1 THEN 'Insert was successful!'

WHEN -1 THEN 'Insert failed! Common Issues: Duplicate Data'

END AS [Status];

SELECT \* FROM Customers;

Go

When this code runs, it successfully inserts new customer data then captures the new customer ID of 1 and a return value of 1. The code then presents the data as shown here. Selecting from the table shows the new row of data.

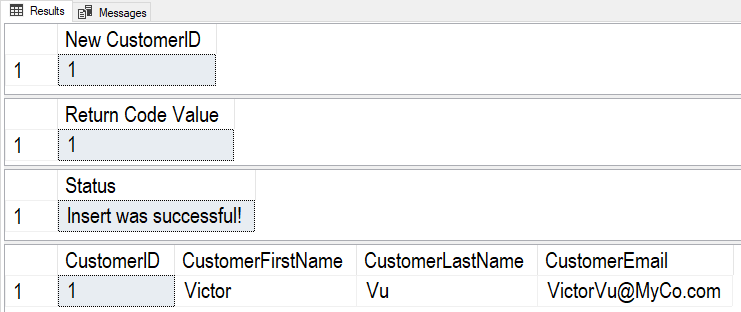


Figure. The results from executing the stored procedure and presenting the return code value of 1

However, **if we run the same code a second time it causes an error since the CustomerEmail's UNIQUE constraint rejects a duplicated email address.** The results from executing the stored procedure are a **return code value of -1,** indicating that the insert failed, and a NULL customer ID, showing that a new number was not generated. Selecting from the table still shows only one row of data.

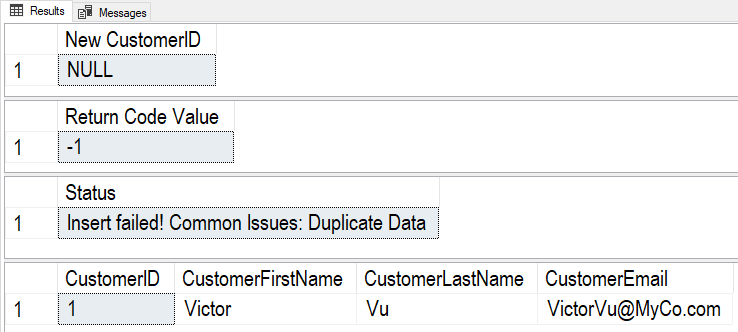


Figure. The results from executing the stored procedure.

### Lookup Stored Procedures

A **lookup stored procedure "looks" up one or more values in a table**. For example, let's say we want to look up a customer ID based on a given Email Address. In that case, we could create a lookup stored procedure like this one:

CREATE OR ALTER PROCEDURE pGetCustomerIDByEmail

(@CustomerEmail nvarchar(100)

,@CustomerID int OUTPUT

)

AS

BEGIN

DECLARE @Status int = 0;

BEGIN TRY

SELECT @CustomerID = CustomerID

FROM Customers

WHERE CustomerEmail= @CustomerEmail;

SET @Status = +1;

END TRY

BEGIN CATCH

SET @Status = -1;

END CATCH

RETURN @Status;

END

go

Now, we execute the stored procedure and capture the ID as shown here:

DECLARE @Status int;

DECLARE @CustomerID int;

EXEC @Status = pGetCustomerIDByEmail

@CustomerEmail = 'VictorVu@MyCo.com'

,@CustomerID = @CustomerID output;

SELECT @Status as [Return Code Value], @CustomerID as [CustomerID]

Executing the stored procedure returns the status of one, indicating that the lookup was successful, and **the output argument now contains the customer ID value found for that email address**.

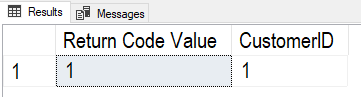


Figure 1. The results from executing the stored procedure

### Nested Stored Procedures

**Transaction stored procedures often use Lookup stored procedures**. Doing this **separates the lookup processing from the transaction processing**, once again making your code more modular. It also **makes code easier to test and maintain** code, since changes can be tested on only the stored procedure that was changed.

To help you picture the process, consider the following image. Here you see **representations of a transaction and a lookup** **stored procedure**. The transaction procedure starts by executing or calling the lookup procedure. The lookup procedure selects data from one or more tables to see if a value is found. When it is, it returns a positive return code indicating success, otherwise, it returns a negative return code. In this example, it also returns a Null value when a lookup value is not found. **The return code or Null value can be examined by the transaction procedure and a decision is made to continue or end the transaction**. Note, our example, a Null would continue with the transaction since the lookup value was not found. Using a lookup stored procedure to validate data before starting a transaction reduces overhead, which increases performance.

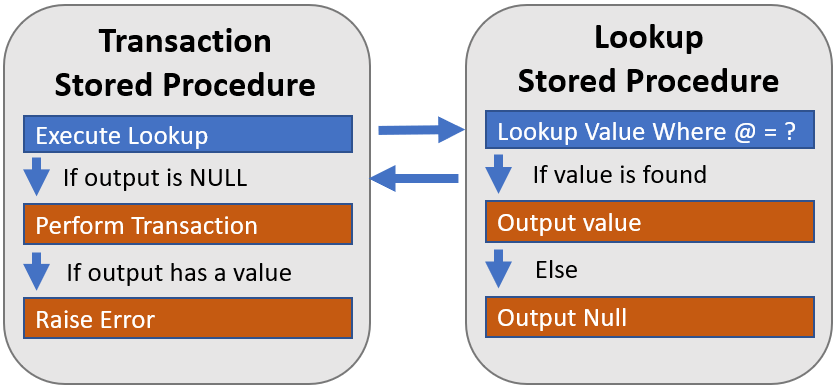


Figure. A figure representing the processing between the transaction and lookup stored procedures.

**Note:** Nesting stored procedures are not limited to transactions but are used in reporting stored procedures as well.

Using our example code, we call the pGetCustomerIDByEmail to see if a customer already exists before we try creating a new customer. To do so, we pass the lookup sproc to the email address given to the transaction sproc and await the return status and customer ID. Here is the modified code for the transaction stored procedure.

CREATE OR ALTER PROCEDURE pInsCustomers

(@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

,@NewCustomerID int OUTPUT

)

AS

BEGIN

DECLARE @Status int = 0;

DECLARE @CustomerID int = NULL; -- Will hold customer ID if it exists

BEGIN TRY

EXEC @Status = pGetCustomerIDByEmail -- Check for existing customer email

@CustomerEmail = @CustomerEmail

,@CustomerID = @CustomerID OUTPUT;

IF @Status = 1 AND @CustomerID IS NULL -- ran successfully and did NOT find an ID

BEGIN

BEGIN TRAN;

INSERT INTO Customers (CustomerFirstName, CustomerLastName, CustomerEmail)

VALUES (@CustomerFirstName, @CustomerLastName, @CustomerEmail);

COMMIT TRAN;

SET @NewCustomerID = @@Identity;

SET @Status = +1;

END

ELSE -- ran successfully, but found an ID

RAISERROR('Customer Already Exists! See CustomerID ', 15, 1);

END TRY

BEGIN CATCH

IF @@TRANCOUNT > 0 ROLLBACK TRAN;

SET @Status = -1;

DECLARE @Message nvarchar(100);

SET @Message = ERROR\_MESSAGE() + Cast(@CustomerID as varchar(10));

RAISERROR(@Message, 15, 1);

END CATCH

RETURN @Status;

END

go

Now, we test the code to see how it works! First, we **test a positive outcome** by inserting a data that should work. In this case, that means a customer that has not already been recorded. Here is our example.

-- Positive test --

DECLARE @Status int;

DECLARE @NewID int;

EXEC @Status = pInsCustomers

@CustomerFirstName = 'Sam'

,@CustomerLastName = 'Solo'

,@CustomerEmail = 'SamSolo@MyCo.com'

,@NewCustomerID = @NewID output;

SELECT @Status as [Return Code Value]

SELECT CASE @Status

WHEN +1 THEN 'Insert was successful!'

WHEN -1 THEN 'Insert failed! Common Issues: Duplicate Data'

END AS [Status];

SELECT \* FROM Customers;

Go

The results we get back show a return code of 1 and the new row in the table.

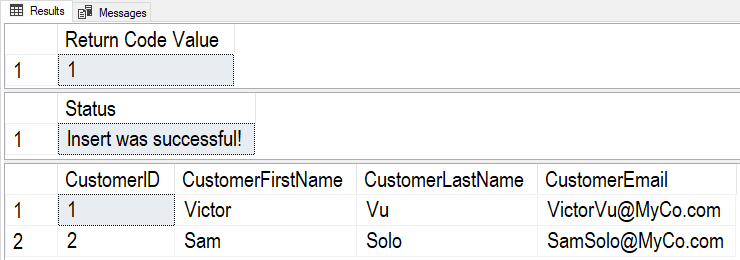


Figure. The results of executing the modified stored procedure with a successful insert

Finally, we **test the code to make sure it fails correctly**. Here we are trying to insert duplicate customer data and expect an error to occur.

-- Negative Test --

DECLARE @Status int;

DECLARE @NewID int;

EXEC @Status = pInsCustomers

@CustomerFirstName = 'Victor'

,@CustomerLastName = 'Vu'

,@CustomerEmail = 'VictorVu@MyCo.com'

,@NewCustomerID = @NewID output;

SELECT @Status as [Return Code Value]

SELECT CASE @Status

WHEN +1 THEN 'Insert was successful!'

WHEN -1 THEN 'Insert failed! Common Issues: Duplicate Data'

END AS [Status];

SELECT \* FROM Customers;

Go

When this code runs, it results in the following error message and results:

Msg 50000, Level 15, State 1, Procedure pInsCustomers, Line 32 [Batch Start Line 19]

Customer Already Exists! See CustomerID 1

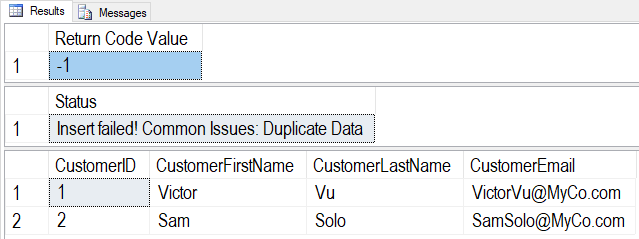


Figure. The results of executing the modified stored procedure with a failed insert

## Demonstration – Creating the ELT Objects

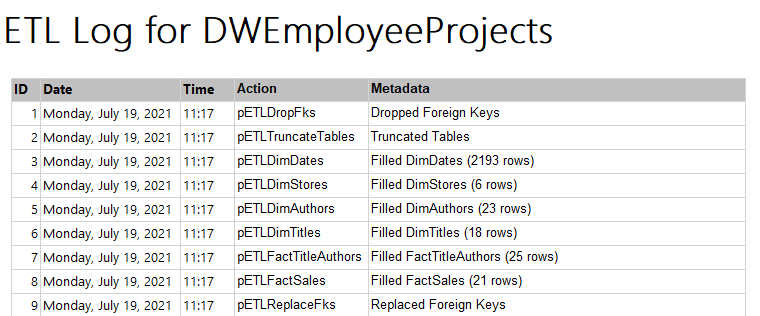
In this demo, you review a SQL script that creates ETL objects (tables, views, and stored procedures).

Files Needed:

* 4\_SQLTestScript.sql
* 5\_ETLObjects.sql

# Creating Admin Reports

ETL is usually an automated process and as such, it needs to be monitored. We can use our ETL Metadata table to create a simple admin report like this one:



## Demonstration – Creating an Administrative Report

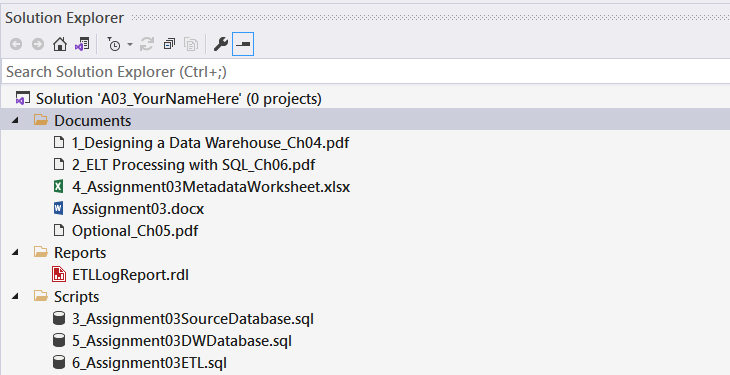
In this demo, you see how to create a simple report using Report Builder.

Files Needed: None

Software Needed: Report Builder

# Keeping Organized

Each week the assignment becomes more complex. So, you must keep your work organized to save time and avoid frustration. Visual Studio is a good tool to use for organization since it can work with most types of files and documents. For example, the assignment in this module uses Word, PDF, and Excel documents. The assignment also includes an RDL and several SQL codes files.



## Demonstration – Organizing with Visual Studio

In this demo will review how to organize files with Visual Studio.