University of Washington

iSchool Info 330

# Module 06 - Database Administration

In this module, we will look at the common tasks and topics of database administration**.**

## Outline

Here is a general outline of what we will be doing this module:

|  |
| --- |
| **Module06: Database Administration** |
| Session01 Lectures and Labs < 110 mins |
| Security - 30 |
| Lab 1: Setting Security - 20 |
| Backups - 20 |
| Lab 2: Backups and Restores - 20 |
| Session02 - Lab |
| Work on Assignment06 |
| Session03 Lectures and Labs < 110 mins |
| Replication and Distribution Technologies - 20 |
| ETL Processing - 30 |
| Lab 3: ETL Processing - 20 |
| Indexing and Performance Tuning - 30 |

**Note**: Times are only estimates and may change without notice!

# Session01 < 110 mins

In this session, we explore **how to perform security and backups, two of the most common DBA tasks**.

## Security - 30

Security involves **authentication, authorization, and encryption**. Authentication determines the **identity** of someone or something trying to access a resource. Authorization determines what kind of **actions** an identity can perform on a resource. Encryption determines **what can be seen** in a resource with or without an encryption key.

### Authentication

"Authentication is a process in which the **credentials provided are compared** to those on file in a database of authorized users' information on a local operating system or within an authentication server. If the credentials match, the process is completed, and the user is granted authorization for access." ( <http://searchsecurity.techtarget.com/definition/authentication>) (links to external site)

#### SQL Server Authentication

"SQL Server supports **two authentication modes**, Windows authentication mode and mixed mode.

* **Windows** authentication is the **default** and is often **referred to as integrated security** because this SQL Server security model is tightly integrated with Windows. Specific Windows user and group accounts are trusted to log in to SQL Server. Windows users who have already been authenticated do not have to present additional credentials.
* **Mixed** mode supports authentication both by Windows and by SQL Server. Username **and password pairs are maintained within SQL Server**." (<https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/authentication-in-sql-server> )(links to external site)

To authenticate into SQL server, you must have a Login. Logins come in either a Windows or SQL variety.

"SQL Server supports three types of logins:

* A local **Windows user** account or trusted domain account. SQL Server relies on Windows to authenticate the Windows user accounts.
* **Windows group**. Granting access to a Windows group grants access to all Windows user logins that are members of the group.
* **SQL Server login**. SQL Server stores both the **username** and a **hash of the password** in the master database, by using internal authentication methods to verify login attempts." (<https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/authentication-in-sql-server> )(links to external site)

With Windows authentication the username and password are stored on a Microsoft **Windows Active Directory domain server**. With SQL Authentication the name and password are stored in Microsoft SQL server's Master database.

You can use the system view syslog ins to see the logins created on an instance of SQL Server (figure 1).

Select \* From Sys.syslogins;

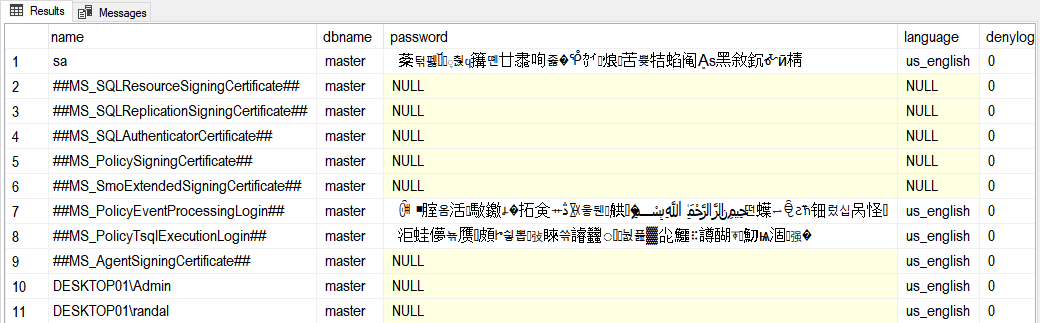


Figure 1: The results of the previous SQL statement

**SQL logins** will have a **hashed password**, while Windows logins will use a null value in the password field.

#### Window vs. SQL Authentication

The advantage of using **Windows** authentication is that, once a user has a domain account, they can use it to **access multiple resources** on the Network; not just Microsoft SQL server, but email servers and network shares.

The advantage of using a **SQL** Authentication login is that you **do not need a Windows domain account** to access SQL data. Using SQL Authentication means that the login will have **access to fewer resources** in your organization's Network, and is **often used when creating web applications** where you do not want each new customer to have to create a Windows domain but do want each customer to have individual access to SQL server resources.

#### Question: Why would SQL Logins be preferred in a Web application?

The **code** used to create an MS SQL login differs depending on if you are creating a Windows or SQL login.

CREATE LOGIN [DESKTOP01\randal] FROM WINDOWS

go

-- OR --

CREATE LOGIN [webuser] WITH PASSWORD = 'Pa$$word';

go

## Authorization

Authorization determines what you have access to. In Microsoft SQL Server, authorization is configured using database Users accounts and permissions.

#### Database Users Accounts

"A user is a database level security principal. **Logins** must be **mapped to a database user** to connect to a database. A login **can be mapped to different databases** as different users but can only be mapped as one user in each database." (<https://docs.microsoft.com/en-us/sql/relational-databases/security/authentication-access/create-a-database-user>) (links to external site)

For each database I want a user to access I must create a user account (Figure 2.)

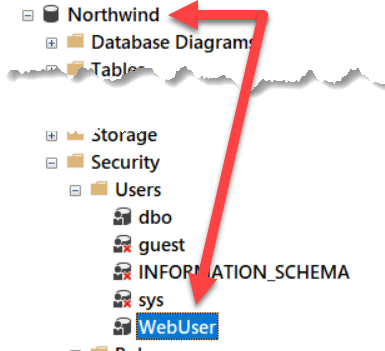


Figure 2: A user account in the Northwind Database.

While there are many options, **in most cases,** you will either use a Windows or SQL **login to map directly to a database user account**. The **code** used to create a database user account is the same no matter that you are assigning it to either a Windows or SQL login.

USE [Northwind]

go

CREATE USER [Randal] FOR LOGIN [DESKTOP01\randal] WITH DEFAULT\_SCHEMA=[dbo]

go

CREATE USER [WebUser] FOR LOGIN [webuser] WITH DEFAULT\_SCHEMA=[dbo]

go

Each time I create a new database, I must add users to it before they can access it!

Create Database SecDemoDB;

go

Use SecDemoDB;

go

CREATE USER [WebUser] FOR LOGIN [webuser] WITH DEFAULT\_SCHEMA=[dbo]

go

You can find out what login and user account you are using with this code (figure 3):

Select

[My DB User Name] = USER

,[My Login Name] = Suser\_Sname()

,[Name of DB I am connected to] = DB\_Name()

,[Name of Server] = @@ServerName

Go

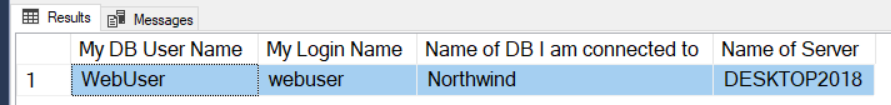


Figure 3: The results of the previous SQL statement

#### Permissions

In order for a user to access a resource (table, view, function, stored procedure, etc.), they must be **granted** permissions. Permission can also be denied or revoked.

When permission is granted, a user can access the resource depending on what type of permission was granted. Permissions on a **table** include the right to **select, insert, update, or delete** the data in the table. Permissions on a **stored procedure** only include the ability to execute the store.

Create Table Students (ID int, Name varchar(50));

go

Create View vStudents AS Select ID, Name From Students;

go

Create Proc pInsStudents AS Insert Into Students(ID, Name) Values (1,'TestInsert');

go

Create Proc pUpdStudents AS Update Students Set ID = 100, Name = 'TestUpdate';

go

Create Proc pDelStudents AS Delete From Students Where ID = 100;

go

Deny Select, Insert, Update, Delete On Students To webuser;

Grant Select On vStudents To webuser;

Grant Execute On pInsStudents To webuser;

Grant Execute On pUpdStudents To webuser;

Grant Execute On pDelStudents To webuser;

SQL Server has several **built-in roles**. These roles can be used to group user accounts. The **Public role automatically includes all the users of a database**. **Permissions** given to a role are **automatically apply to all user** accounts in that role.

Deny Select, Insert, Update, Delete On Students To Public;

Grant Select On vStudents To Public;

Grant Execute On pInsStudents To Public;

Grant Execute On pUpdStudents To Public;

Grant Execute On pDelStudents To Public;

When someone is **granted** permission, they can use those permissions to **perform the granted action**. When someone is **denied** permission, they **cannot perform the denied action**.

For **example,** and the following SQL code, **any member of the Public role will no longer be able to select data from the vStudents view**. Members of the Public role would **include the WebUser**, who was earlier granted select permission automatically since they are part of the Public role!

Deny Select On vStudents To Public;

When someone has their permission **revoked** they are **neither granted nor denied** the ability to perform an action. Having a revoked status means that if a user has been granted or denied access from some other role (group) they belong to, they will be allowed or denied the ability to perform an action based on the permissions of that other role! In this current example, this means that **WebUser**, who was earlier granted select permission, can once again access the view!

Revoke Select On vStudents To WebUser;

In addition to the built-in database roles, you can **create your own custom roles** and assign permissions to those.

Use SecDemoDB;

go

CREATE ROLE [DevAppUsers]

go

You would then **add database users** to the new custom roll you created and set the permission you desired.

-- Add the users to the role

ALTER ROLE DevAppUsers ADD MEMBER WebUser;

go

Grant Select On vStudents To DevAppUsers;

## Lab 1: Setting Security - 20

In this lab you will create a new database, table, view, and insert stored procedure. You will then deny permissions on the table but grant access on the view and stored procedure for the Public role.

You will work on your own for the first 10 minutes, then we will review the answers together in the last 10 minutes.

**Note**: This lab should be done individually or in groups of three or less.

### Step 1: Create the database

Run the following SQL code into a code window, then modify it to use your own name instead of "YourNameHere." Afterward, execute the code to make the Module 06 labs database.

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

-- Title: Mod06 Labs Database

-- Author: YourNameHere

-- Desc: This file demonstrates how to process data in a database

-- Change Log: When,Who,What

-- 2017-01-01,YourNameHere,Created File

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

Use Master;

go

If Exists(Select Name from SysDatabases Where Name = 'MyLabsDB\_YourNameHere')

Begin

Alter Database [MyLabsDB\_YourNameHere] set Single\_user With Rollback Immediate;

Drop Database MyLabsDB\_YourNameHere;

End

go

Create Database MyLabsDB\_YourNameHere;

go

Use MyLabsDB\_YourNameHere;

go

### Step 2: Create the table

Run the following SQL code into a code window to create a table;

Create Table Categories

([CategoryID] [int] IDENTITY(1,1) NOT NULL

,[CategoryName] [nvarchar](100) NOT NULL

);

go

### Step 3: Create a base view for the table

Create and run SQL code into a code window to create a base view for the table;

### Step 4: Create an insert stored procedure

Use the provided stored procedure template to create code an insert procedure for the table.

Create Procedure <pTrnTableName>

(<@P1 int = 0>)

/\* Author: <YourNameHere>

\*\* Desc: Processes <Desc text>

\*\* Change Log: When,Who,What

\*\* <2017-01-01>,<Your Name Here>,Created Sproc.

\*/

AS

Begin

Declare @RC int = 0;

Begin Try

Begin Transaction

-- Transaction Code --

Commit Transaction

Set @RC = +1

End Try

Begin Catch

Rollback Transaction

Print Error\_Message()

Set @RC = -1

End Catch

Return @RC;

End

go

/\* Testing Code:

Declare @Status int;

Exec @Status = pTrnTableName @P1 = 1;

Print @Status;

\*/

### Step 5: Test the View and Stored Procedure

Test your code by executing the stored procedure and selecting the view.

### Step 6: Set permissions

Deny the public role all access to the table, allow select permissions on the view, and allow execute permissions on the stored procedure.

**Note**: If you are using the iSchool SQL Server you can test your permissions with the WebUser login (password = 'sql'), by creating a user account in your database.

### Step 7: Review Your Work

Now, you will review your work with your instructor.

## Backups - 20

After you have created tables views stored procedures user accounts functions and set permissions a lot of effort has been performed. Instead of risking the chance of losing all the hours of effort you should consider backing up the database.

**A database backup will not only capture the data in the database, but all the programming objects, users and roles, and permission settings**.

#### Demo: Here is some code that your instructor will demonstrate:

Create Database BackupDemoDB;

go

Use BackupDemoDB;

go

Create Table BackupDemo (Id int);

go

Insert into BackupDemo(Id) Values(1);

-- Check the data

go

Select \* from BackupDemo;

-- Backup the entire database (daily, weekly)

Backup Database BackupDemoDB To Disk ='C:\\_SQLDev\BackupDemoDB\_Full.bak';

go

-- Add more data after the table

Insert into BackupDemo Values(2)

go

-- and back up the new inserted data using a log backup (6 hours, daily)

Backup Log BackupDemoDB To Disk ='C:\\_SQLDev\BackupDemoDB\_Log.bak';

go

-- Check the data

Select \* from BackupDemo;

go

-- test the Full DB backup

Use Master;

go

Restore Database BackupDemoDB From Disk ='C:\\_SQLDev\BackupDemoDB\_Full.bak'

With Recovery, Replace -- Both are optional but recommend

go

-- Check the data

Use BackupDemoDB;

go

Select \* from BackupDemo;

Go

'This will not work because the DB has already RECOVERED'

Restore Log BackupDemoDB From Disk ='C:\\_SQLDev\BackupDemoDB\_Log.bak';

go

-- test the Full DB and Log backup

Use Master;

go

Restore Database BackupDemoDB From Disk ='C:\\_SQLDev\BackupDemoDB\_Full.bak'

With NoRecovery, Replace

go

Restore Log BackupDemoDB From Disk ='C:\\_SQLDev\BackupDemoDB\_Log.bak'

With Recovery;

go

-- Check the data

Use BackupDemoDB;

go

Select \* from BackupDemo;

go

# Session02 - Lab

This session gives you time to work on the assignment task. **See the Assignment document for details**

# Session03 Lectures and Labs < 110 mins

In this session, you will learn about ways of extracting, transforming, and loading data. There are a lot of ways to do this, but we will focus first on one of the earliest ways of doing this in MS SQL Server and then on newer technologies and techniques. We will also take a quick look at the topic of indexing a performance turning.

## Replication and Distribution Technologies - 20

"Replication is a set of technologies for copying and distributing data and database objects from one database to another and then synchronizing between databases to maintain consistency. Use replication to **distribute data to different locations and to remote or mobile users over local and wide area networks, dial-up connections, wireless connections, and the Internet**." (<https://docs.microsoft.com/en-us/sql/relational-databases/replication/sql-server-replication>) (links to external site)

### Microsoft SQL Server Replication

Microsoft has a technology called Replication as part of Microsoft SQL server. However, this technology is quite old and has been replaced with other newer Technologies over the last decade.

Still **the concept of replication** has not changed, the idea is that you **extract data from one location and load it into another location**. Typically, these locations are on different computers in different parts of your Network or different parts of the world.

#### "Types of Replication

SQL Server 2000 supports three types of replication: **snapshot, transactional, and merge**. **Snapshot** replication is the **periodic replication of an entire set of data** as of a specific moment in time from a local server to remote servers. You would typically use this type of replication in databases where the amount of data to be replicated is small and the source data is static. You can grant remote servers a limited capability to update the replicated data. **Transactional** replication is the replication of an **initial snapshot** of data to remote servers **plus** the replication of individual transactions occurring at the local server that **incrementally modify data** contained in the initial snapshot. These replicated transactions are applied to the replicated data at each remote server to keep the data on the remote server synchronized with the data on the local server. You use this type of replication when you must keep the data current on the remote servers. You can grant remote servers a limited capability to update the replicated data. **Merge replication** is the replication of an **initial snapshot** of data to remote servers **plus** the replication of **changes** that occur **at any remote server** back to the local server for synchronization, **conflict resolution**, and **re-replication** to remote servers. You use merge replication when numerous changes are made to the same data, or when remote offline **computers need to operate autonomously**, such as in the case of a mobile user. "(*Microsoft SQL Server 2000 System Administration Training Kit, 2001, MSPress)*

### Stored Procedures

We have used transactional stored procedures to insert, update, and delete data in a table, but they can also be used to insert update and delete data in multiple tables.

For example, let's say that you have a Customers table, and somebody updates the customer's name or email address. If you would like to store the original values someplace you write not just one transactional statement, but a second statement to copy that original data to a new location (usually in a separate column or separate "tracking" table).

Here is an example:

Use MyLabsDB\_YourNameHere;

go

Create -- Drop

Table Customers

(CustomerID int Primary Key identity

,CustomerFirstName nvarchar(100)

,CustomerLastName nvarchar(100)

,CustomerEmail nvarchar(100)

);

go

Create -- Drop

Table CustomersChangeTracker

(CustomerChangeTrackerID int Primary Key identity

,CustomerChangeTrackerDateTime datetime Default(GetDate())

,CustomerID int

,CustomerFirstName nvarchar(100)

,CustomerLastName nvarchar(100)

,CustomerEmail nvarchar(100)

,TransactionType char(1) Check (TransactionType in ('i','u','d'))

);

go

Create -- Drop

Procedure pInsCustomersChangeTracker (

@CustomerID int

,@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

,@TransactionType char(1)

)

AS

Begin -- Skipping header,try-catch, and transaction code for demo

Insert Into CustomersChangeTracker

(CustomerID, CustomerFirstName, CustomerLastName, CustomerEmail, TransactionType)

Values(@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail, @TransactionType);

End

go

Create -- Drop

Procedure pInsCustomers (

@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

AS

Begin -- Skipping header,try-catch,and transaction code for demo

-- Step 1: Update the current data

Insert Into Customers(CustomerFirstName, CustomerLastName, CustomerEmail)

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

-- Step 2: Copy the Inserted data

Exec pInsCustomersChangeTracker

@@IDENTITY, @CustomerFirstName, @CustomerLastName, @CustomerEmail, 'i';

End

go

Create -- Drop

Procedure pUpdCustomers (

@CustomerID int

,@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

AS

Begin -- Skipping header, try-catch, and transaction code for demo

-- Step 1: Update the current data

Update Customers

Set CustomerFirstName = @CustomerFirstName

,CustomerLastName = @CustomerLastName

,CustomerEmail = @CustomerEmail

Where CustomerID = @CustomerID;

-- Step 2: Copy the Updated data

Exec pInsCustomersChangeTracker

@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail, 'u';

End

go

Create -- Drop

Procedure pDelCustomers (

@CustomerID int

,@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

)

AS

Begin -- Skipping header, try-catch, and transaction code for demo

-- Step 1: Copy the changed data

Exec pInsCustomersChangeTracker

@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail, 'd';

-- Step 2: Delete the current data

Delete From Customers

Where CustomerID = @CustomerID;

End

go

-- Testing Code --

Exec pInsCustomers

@CustomerFirstName = 'Bob'

,@CustomerLastName = 'Smith'

,@CustomerEmail = 'BSmith@MyCo.com';

Select \* From Customers;

Select \* From CustomersChangeTracker;

go

-- Since am changed data in two tables I use the IDENT\_CURRENT() function

-- to get the Identity number generated by a spacific table.

Declare @ID int;

Select @ID = IDENT\_CURRENT('Customers');

Exec pUpdCustomers

@CustomerID = @ID

,@CustomerFirstName = 'Robert'

,@CustomerLastName = 'Smith'

,@CustomerEmail = 'BSmith@MyCo.com';

Select \* From Customers;

Select \* From CustomersChangeTracker;

go

Declare @ID int;

Select @ID = IDENT\_CURRENT('Customers');

Exec pDelCustomers

@CustomerID = @ID

,@CustomerFirstName = 'Robert'

,@CustomerLastName = 'Smith'

,@CustomerEmail = 'BSmith@MyCo.com';

Select \* From Customers;

Select \* From CustomersChangeTracker;

go

Each time a customer's data is updated in this manner, the tracking table or column is also updated, thereby tracking the history of changes to your data.

### Triggers

Triggers are a special type of stored procedure that is applied to tables and views. Triggers invoke SQL code when table data is modified with an INSERT, UPDATE, or DELETE action. Data modification events cause triggers to fire automatically. Usually, this involves making a copy of data into a tracking table or updating another table in the database with new calculated values. Complex transformations can be applied to data as it is copied.

Create -- Drop

Trigger tTrackCustomers On Customers

After Insert, Update, Delete

AS

Begin

Select \*, 'Data in Inserted' as [Note] From Inserted;

Select \*, 'Data in Deleted' as [Note] From Deleted

If ((Select count(\*) From Inserted) = 1) And ((Select count(\*) From Deleted) = 0)

Select 'Trigger: It was an Insert';

If ((Select count(\*) From Inserted) = 1) And ((Select count(\*) From Deleted) = 1)

Select 'Trigger: It was an Update';

If ((Select count(\*) From Inserted) = 0) And ((Select count(\*) From Deleted) = 1)

Select 'Trigger: It was a Delete';

End

go

-- Test the Trigger!

Truncate Table Customers; -- Does not "Fire" the trigger!

go

Insert into Customers (CustomerFirstName, CustomerLastName, CustomerEmail)

Values ('a','b','ab@TestCo.com');

go

Declare @ID int;

Select @ID = IDENT\_CURRENT('Customers');

Update Customers Set CustomerFirstName = 'AA' Where CustomerID = @ID;

go

Declare @ID int;

Select @ID = IDENT\_CURRENT('Customers');

Delete From Customers Where CustomerID = @ID;

go

-- Configure the trigger to use the tracking table

Alter

Trigger tTrackCustomers On Customers

After Insert, Update, Delete

AS

Begin

Declare

@CustomerID int

,@CustomerFirstName nvarchar(100)

,@CustomerLastName nvarchar(100)

,@CustomerEmail nvarchar(100)

If (((Select count(\*) From Inserted) = 1) And ((Select count(\*) From Deleted) = 0))

BEGIN -- IF

Select

@CustomerID = CustomerID

,@CustomerFirstName = CustomerFirstName

,@CustomerLastName = CustomerLastName

,@CustomerEmail = CustomerEmail

From Inserted

Exec pInsCustomersChangeTracker

@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail, 'i';

END -- IF

If (((Select count(\*) From Inserted) = 1) And ((Select count(\*) From Deleted) = 1))

BEGIN -- IF

Select

@CustomerID = CustomerID

,@CustomerFirstName = CustomerFirstName

,@CustomerLastName = CustomerLastName

,@CustomerEmail = CustomerEmail

From Inserted

Exec pInsCustomersChangeTracker

@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail, 'u';

END -- IF

If (((Select count(\*) From Inserted) = 0) And ((Select count(\*) From Deleted) = 1))

BEGIN -- IF

Select

@CustomerID = CustomerID

,@CustomerFirstName = CustomerFirstName

,@CustomerLastName = CustomerLastName

,@CustomerEmail = CustomerEmail

From Deleted

Exec pInsCustomersChangeTracker

@CustomerID, @CustomerFirstName, @CustomerLastName, @CustomerEmail, 'd';

END -- IF

End

go

-- Test the new trigger configuration

Truncate Table Customers; -- Does not "Fire" the trigger!

Truncate Table CustomersChangeTracker; -- Does not "Fire" the trigger!

go

Insert into Customers (CustomerFirstName, CustomerLastName, CustomerEmail)

Values ('a','b','ab@TestCo.com');

Select \* From CustomersChangeTracker;

go

Declare @ID int;

Select @ID = IDENT\_CURRENT('Customers');

Update Customers Set CustomerFirstName = 'AA' Where CustomerID = @ID;

Select \* From CustomersChangeTracker;

go

Declare @ID int;

Select @ID = IDENT\_CURRENT('Customers');

Delete From Customers Where CustomerID = @ID;

Select \* From CustomersChangeTracker;

## ETL Processing - 30

The ETL process involves **extracting** vital information from sources, **transforming** data into clean, consistent, and usable data, and then **loading** it into another location (usually a reporting structure).

**The typical steps** involved in creating an **ETL process** include:

1. Deciding between **reload** a table with all data **or** loading it **incrementally**.
2. **Locating** the data to extract and **examining** its contents for validity, conformity, and completeness.
3. **Transform** it to fit your destination tables.
4. **Load** the data into your reporting tables.

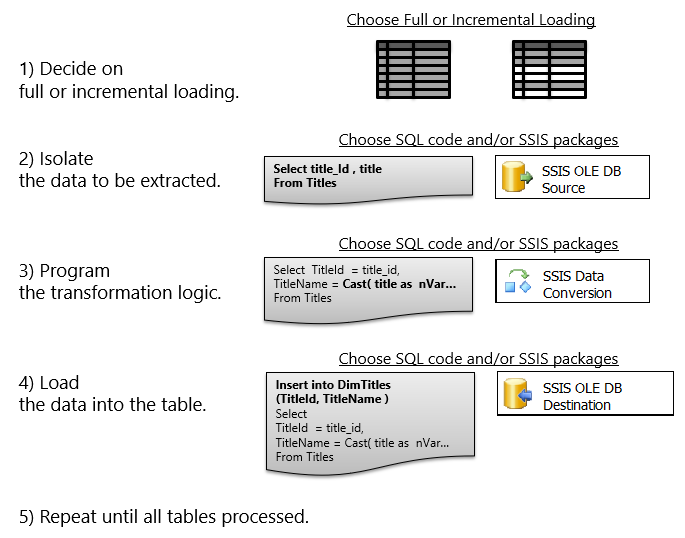


Figure: The typical steps ETL processing (SQL Code or ETL Software)

As an example, consider creating a reporting table for the Pubs.dbo.Titles table.

-- Here's the code for the source table

USE [pubs]

GO

CREATE TABLE [dbo].[titles] (

[title\_id] [dbo].[**tid**] PRIMARY KEY

,[title] [**varchar](80)** NOT Null

,[type] **[char](12)** NOT Null

,[pub\_id] [char](4) Null REFERENCES [dbo].[publishers] ([pub\_id])

,[price] **[money]** Null

,[advance] [money] Null

,[royalty] [int] Null

,[ytd\_sales] [int] Null

,[notes] [varchar](200) Null

,[pubdate] [**datetime**] NOT Null

)

-- Here is the code for the destination table

Use Master;

go

Create -- Drop

Database MyReportDB;

go

Use MyReportDB;

go

Create Table [dbo].[DimTitles] (

[TitleKey] [int] IDENTITY(1,1) PRIMARY KEY

,[TitleId] [nchar](6) NOT Null

,[TitleName] [nvarchar](100) NOT Null

,[TitleType] [nvarchar](100) NOT Null

,[PublisherName] [nvarchar](100) NOT Null

,[TitlePriceUSD] [nvarchar](100) NOT Null

,[PublishedDate] [nvarchar](100) NOT Null

)

Data transformations to the new table must consider:

1. There are a **different number of columns** between the two tables
2. **Column names** have been changed
3. Column **data types** have been **changed**
4. The **foreign key references a surrogate** column
5. Values may need cleaning (**remove or change incorrect or suspect data**)
6. Values may need to be in a more **readable or consistent** format.
7. Dealing with **Null specifications**

During the **ETL process**, you create code and programming structures to transform the data in the source to the format of the destination. **To accomplish** that you need **three things**:

* You need to **extract** the data from the data **source**
* You need to **transform** the data into the destination format
* You need to **load** the data into the **destination** table

Of course, you may not need to extract every column from the original table so, you can just **leave out the columns you do not want** in the select clause.

-- 1) There are a different number of columns between the two tables

Select

[title\_id]

, [title]

, [type]

**, [pub\_name]**

, [price]

, [pubdate]

From [Pubs].[dbo].[Titles] as t

Join [Pubs].[dbo].[**Publishers**] as p

On t.pub\_id = p.pub\_id;

You may want to **change the name** of your source columns to match the names of your destination columns by using a column alias. This makes it easier for people to see the correlation between your sources and destinations.

-- 2) Column **names** have been changed

Select

[TitleId] = title\_id

,[TitleName] = title

,[TitleType] = type

,[PublisherName] = p.pub\_name

,[TitlePriceUSD] = price

,[PublishedDate] = pubdate

From [Pubs].[dbo].[Titles] as t

Join [Pubs].[dbo].[Publishers] as p

On t.pub\_id = p.pub\_id;

In cases where the data type is different between the source and destination tables you can add code to **convert the data types and nulls** as follows.

Insert Into DimTitles

(TitleId, TitleName, TitleType, PublisherName, TitlePriceUSD, PublishedDate)

Select

[TitleId] = Cast([title\_id] as nchar(6))

,[TitleName] = Cast([title] as nvarchar(100))

,[TitleType] = Cast([type] as nvarchar(100))

,[PublisherName] = p.pub\_name

,[TitlePriceUSD] = **IsNull**(**Format**([price], 'C', 'en-us'), 'NA') -- null replacement

,[PublishedDate] = **IsNull**(**Cast**(pubdate as nvarchar(100)),'NA') -- null replacement

From [Pubs].[dbo].[Titles] as t

Join [Pubs].[dbo].[Publishers] as p

On t.pub\_id = p.pub\_id;

You would next take this insert code and create a stored procedure around it. The stored procedure would include a deletion statement to clear the reporting table (DimTitles) before it is refilled with both existing and new data from the original transaction table (Titles).

Create Procedure pETLDimTitles

/\* Author: RRoot

\*\* Desc: Processes Flush and Fill ETL on DimTitles

\*\* Change Log: When,Who,What

\*\* 2017-01-01,RRoot,Created Sproc.

\*/

AS

Begin

Declare @RC int = 0;

Begin Try

Begin Transaction

-- Transaction Code --

-- Step 1: Clear the Old Data

Delete From DimTitles;

-- Step 2: Load Current Data

Insert Into DimTitles

(TitleId, TitleName, TitleType, PublisherName, TitlePriceUSD, PublishedDate)

Select

[TitleId] = Cast([title\_id] as nchar(6))

,[TitleName] = Cast([title] as nvarchar(100))

,[TitleType] = Cast([type] as nvarchar(100))

,[PublisherName] = p.pub\_name

,[TitlePriceUSD] = IsNull(Format([price], 'C', 'en-us'), 'NA') -- null replacement

,[PublishedDate] = IsNull(Cast(pubdate as nvarchar(100)),'NA') -- null replacement

From [Pubs].[dbo].[Titles] as t

Join [Pubs].[dbo].[Publishers] as p

On t.pub\_id = p.pub\_id;

Commit Transaction

Set @RC = +1

End Try

Begin Catch

Rollback Transaction

Print Error\_Message()

Set @RC = -1

End Catch

Return @RC;

End

go

Exec pETLDimTitles;

go

Select \* From DimTitles;

## Lab 3: ETL Processing - 20

In this lab you will create and test an ETL script.

You will work on your own for the first 10 minutes, then we will review the answers together in the last 10 minutes.

**Note**: This lab should be done individually or in groups of three or less.

### Step 1: Create Tables

Use the following SQL code to create a reporting table in your Mod05Labs\_YouNameHere database.

-- Create Database MyLabsDB\_RRoot; -- As Needed

go

Use MyLabsDB\_RRoot;

go

CREATE -- Drop

TABLE [dbo].[DimProducts](

[ProductKey] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,

[ProductID] [int] NOT NULL,

[ProductName] [nvarchar](100) NOT NULL,

[ProductCategoryName] [nvarchar](100) NOT NULL,

[ProductStdPriceUSD] [nvarchar](100) NOT NULL,

[ProductIsDiscontinued] [nchar](1) NOT NULL -- ('y or n')

)

### Step 2: Create and Test Transformation Code

Test the following SQL transformation code and note what is does.

Declare @BitTest bit = 1, @DecimalTest money = $1.99;

Select iif(@BitTest = 0, 'y', 'n'); -- Convert Bit to Character

Select Format(@DecimalTest, 'C', 'en-us'); -- Convert Character with $

### Step 3: Create an ETL Stored Procedure

Create and test an ETL stored procedure that will delete all the data in the DimProducts Reporting table and fill it with the transformed data from the Northwind.dbo.Products table.

Delete From DimProducts; -- Clear table of current data

go

Select

ProductID = ProductId

,ProductName = ProductName -- Convert to nVarchar(100)

,ProductCategoryName = CategoryName -- Convert to nVarchar(100)

,ProductStdPrice = UnitPrice -- Convert to nVarchar(100) with $

,ProductIsDiscontinued = Discontinued -- Convert to character ('y' or 'n')

From Northwind.dbo.Products as p Join Northwind.dbo.Categories as c

On p.CategoryID = c.CategoryID

### Step 4: Review Your Work

Now, you will review your work with your instructor.

## Indexing and Performance Tuning - 30

Indexes in databases are **like indexes in books**. In a book, an index allows you to find information quickly without reading the entire book. In a database, an index **allows the database program to find data in a table without scanning the entire table**.

An index in a book is a list of words with the page numbers that contain each word. An index in a database is a list of values in a table with the storage locations of rows in the table that contain each value.

Indexes can be created on either a **single column or a combination of columns** in a table.

SQL Server **automatically creates indexes for certain types of constraints** (for example, PRIMARY KEY and UNIQUE constraints). You can further customize the table definitions by creating indexes that are independent of constraints.

### Table and Index Architecture

Clustered tables are tables that have a clustered index. The data rows are stored in order based on the clustered index column(s).

Heaps are tables that have no clustered index. The data rows are not stored in any particular order, and there is no specific order to the sequence of the data pages of the table.

Common characteristics of an index are:

* Unique (duplicate records not allowed)
* Composite (index made of multiple columns)
* Sort order (the order that data is stored)

### Index Performance

The performance benefits of indexes, however, do come with a cost:

* Tables with indexes **require more storage space** in the database.
* Commands that **insert, update, or delete data can take longer** and require more processing time to maintain the indexes.

When you design and create indexes, you should ensure that the performance benefits outweigh the extra cost in storage space and processing resources. Do not create indexes for every column in a table since the majority of databases are dynamic; that is, records are added, deleted, and changed regularly.

Here is some example code:

CREATE DATABASE IndexDemo

Go

Use IndexDemo

Print('\*\*\* Heaps and Clusters \*\*\*')

Go

-- Heaps --

-- By default, data is placed in pages in the first available space.

-- This may start as being sequential, but the sequence is NOT maintained.

-- This type of organization is referred to as a HEAP.

-- When a table is first made, the configuration of that table is a Heap.

CREATE TABLE PhoneList

( Id int, Name varchar(50), Extension char(5))

Go

INSERT INTO PhoneList VALUES (1, 'Bob Smith', '11')

INSERT INTO PhoneList VALUES (2, 'Sue Jones', '12')

INSERT INTO PhoneList VALUES (3, 'Joe Harris', '13')

Go

SELECT \* FROM Phonelist

-- While the table may appear to be sequentially organized,

-- if you remove a row and then add one back you will see

-- that SQL Server uses the first available slot in a page.

DELETE FROM PhoneList WHERE Id = 2

Go

INSERT INTO PhoneList VALUES (4, 'Tim Thomas', '#14')

Go

SELECT \* FROM Phonelist

-- This is not a big deal since you can have it display the results sequentially

-- using in order by statements. However, this causes the server to sort

-- the results before returning them.

'Note: Turn on the Execution Plan'

SELECT \* FROM Phonelist

Go

SELECT \* FROM Phonelist ORDER BY [Id]

Go

SELECT \* FROM Phonelist ORDER BY [Name]

-- Clustering --

-- If you would like to have SQL Server maintain the sequence on the page

-- you can add a Clustered Index to the table, and it will do so.

CREATE CLUSTERED INDEX ci\_Id ON PhoneList(Id)

Go

-- Now the table will be physically sorted on that Indexed column.

-- This will improve performance on some of your queries,

-- but will not help all of them.

SELECT \* FROM Phonelist

Go

SELECT \* FROM Phonelist ORDER BY [Id]

Go

SELECT \* FROM Phonelist ORDER BY [Name]

-- If you believe that more users will search by Name then by Id,

-- you may want to place the Clustered index on the Name column instead.

-- However, the data pages can be sorted only one way at a time.

-- So, you will have to drop the current Clustered Index before you can

-- make a new one.

DROP INDEX PhoneList.ci\_Id

Go

CREATE CLUSTERED INDEX ci\_Name ON PhoneList(Name)

Go

-- See how the statements preform now.

SELECT \* FROM Phonelist

Go

SELECT \* FROM Phonelist ORDER BY [Id]

Go

SELECT \* FROM Phonelist ORDER BY [Name]