MODULE 1

Lesson 1

**The Basic Architecture of SQL Server**

**Relational Databases**

* SQL Server is a relational database management system
* Databases contain data and objects, including tables, views, stored procedures, user accounts, and other management objects.
* Each database has multiple tables
* Tables are joined together to extract meaningful information

In addition to databases that are created to store information, SQL Server includes 5 system databases:

1. **master**: the system configuration database
2. **model**: the template database. SQL Server will apply any changes made in model to new databases
3. **msdb**: used by SQL Server Agent to Schedule jobs and alerts
4. **tempDb**: a temporary store for data such as work tables. This database is dropped and recreated each time SQL Server restarts , which means that any temporary tables will be lost when SQL Server closes down
5. **resource** : a hidden, read-only database that contains all the system objects for other databases

Before you can execute queries, or insert or delete information from a database, you must connect to the database. You need security credentials to log on to SQL Server, and a database account with permissions to access data object.

**Client Server Databases**

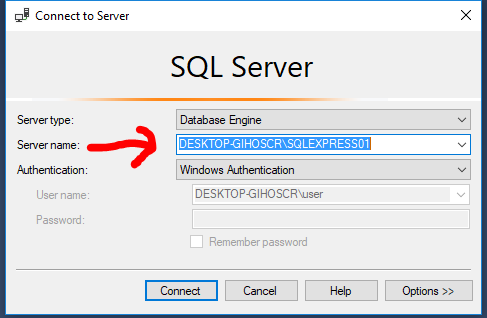
SQL Server is a client server system. This means that the client software, which includes SQL Server Management Studio and Visual Studio, is separate from the SQL Server Database Engine.

When the client application sends requests to the database engine as T-SQL statement, SQL performs the necessary file access, memory management and processor utilization on behalf of the client. The client never has direct access to the database files – unlike, for example, a desktop database application

* client software is separate from the server database engine
* Client/Server refers to the separation of functionality –not where the software is actually located
* Client software and server database engine can be on the same machine
* Database can access data in other databases over a network

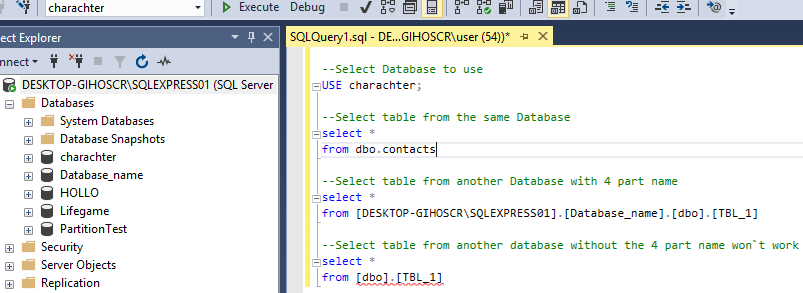
In production environments, the client software runs on a separate machine to the database engine. Indeed, there could be multiple clients accessing the same database engine.

Wherever the client and server software is located, it makes no difference to the way you write T-SQL code. On the logon screen you just specify the SQL Server that you want to connect to.

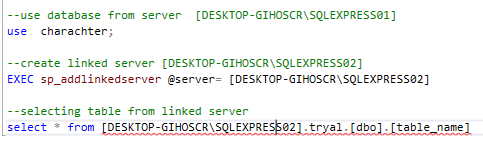


You can also refer to other data bases in a T-SQL script by using its four-part name which has the format

**Instance.Database.Schema.Object**.



Connecting to a remote server requires the remote instance to be set up as a linked server. In T-SQL , you add a linked server using **sp\_addlinkserver**. Altough **sp\_addlinkserver** takes a number of optional arguments, in its simplest form you could connect to the server in the previous example using the statement **exec** **sp\_addlinkserver = serverName**



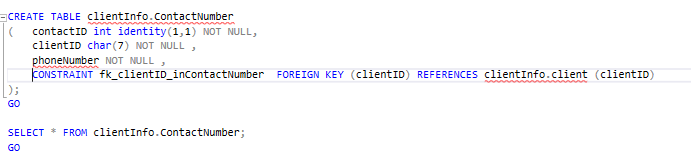
**Queries**

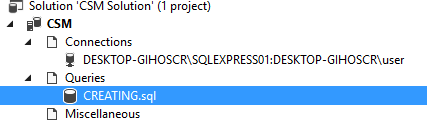
T-sql is a set-based language, which means it does not extract data row by row, but instead extracts data from tables that normally contain many rows. Only after it has retrieved the table does SQL Server filter data to produce a subset of the table if that is what the query has requested.

This makes SQL Server highly efficient in dealing with large volumes of data, but it means you have to think in sets to write efficient T-SQL code.

T-SQL scripts are stored in script files with a **.sql** extension. Inside each file, you can divide the script into batches, each batch concluding with the **GO** keyword. SQL Server runs each batch in its entity before it starts the next one.

* T.SQL is a set-based language
* T.SQL IS written in scripts with .sql extension
* GO keyword separates batches





Lesson 2

**SQL Server Editions and versions**

**SQL Server Versions**

|  |  |
| --- | --- |
| **version** | **Release Year** |
| 2016 | 2016 |
| 2014 | 2014 |
| 2008 R2 | 2010 |
| 2008 | 2008 |
| 2005 | 2005 |
| 200 | 2000 |
| 7 | 1998 |
| 6.5 | 1996 |
| 6 | 1995 |
| 4.2.1 | 1994 |
| 4.2 | 1992 |
| 1.1 | 1991 |
| 1 | 1989 |

SQL Server was first developed in 1989 for the OS/2 operating system.

SQL Server 4.2 and later versions where developed to run on windows.

Although the name may suggest otherwise, SQL Server 2008 R2 is not a service pack for SQL Server 2008. It is an independent version (number 10.5) with enhanced multiserver management capabilities, in addition to new business intelligence (BI) features.

|  |  |
| --- | --- |
| **Main Editions** | **Other Editions** |
| Enterprise | Developer |
| Standard | Express |
| Business Inteligence | Compact |
| Azure SQL Database |  |

**SQL Server Editions**

SQL SERVER is available in different editions with different feature sets that target various business scenarios. In SQL Server 2012 release, Microsoft streamlined the number of editions from previous versions.

**Four main editions are available :**

1. **Enterprise:** This is SQL Server`s flagship edition containing all features, including Business Intelligence, support for data warehousing, and high availability.
2. **Standard:** This includes the database engine, as well as core reporting and analytics capability. Standard supports 16 processor cores but does not include all the high availability, security and data warehousing features found in the Enterprise edition.
3. **Business Intelligence:** This includes the core database engine, along with the full Business Intelligence functionality of analytics, reporting, and integration services. However, like the Standard edition, it supports 16 processor cores and does not offer all the high availability, security and data warehousing features of the Enterprise edition.
4. **Express:** This is a free version of SQL Server and is limited to four processor cores, 1GB of memory per instance, and 10GB maximum storage per database.

In addition to the editions described above, SQL Server also runs in the cloud, in one of two ways :

You can install SQL Server on a cloud-based virtual machine that your organization has provisioned and integrated with its infrastructure. When properly set up, SQL Server works as if it were on a server on your network.

Secondly, it could be an Azure SQL Database. This is Software as a Service (SaaS) and allows you to use SQL Server without physical server or a cloud-based virtual machine. You can add or remove performance as required, making this highly scalable option.

Lesson 3

**Getting Started with SQL Server Management Studio**

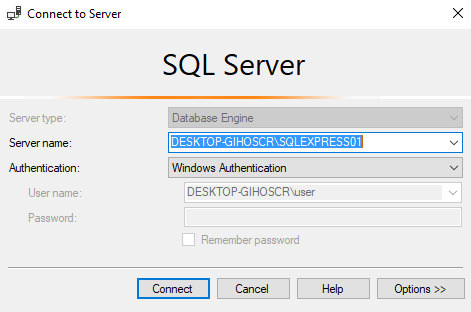
SSMS is an integrated management, development, and querying application that has many features for exploring and working with your database.

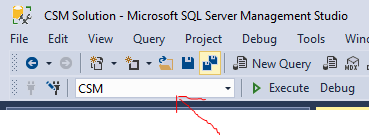
You can start SSMS in one of two ways:

1. USE the **SSMS** shortcut on the window Start menu.
2. Type **ssms.exe** in a command prompt window

By default, SSMS will display a Connect to server dialog box where you can specify the server (or instance)

Name, together with your security credentials.

To specify the database you want to connect to, click the **option** button to open the Connection properties dialog box. Alternatively, you can select the database after you have connected.

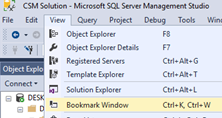
 or with script 

You can explore many SSMS features without connecting to a SQL Server instance. You can also cancel the Connection to Server dialog box if you want to connect to a server later.

With SSMS running, you can explore some of its settings found in **tools**, Options. You can change the default font, enable line numbering for scripts, and control the behaviour of its many windows.

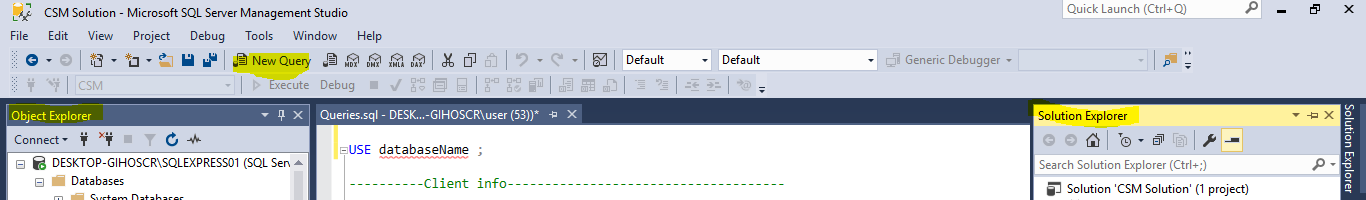
Useful windows include:

Query Editor



Object Explorer

Solution Explorer

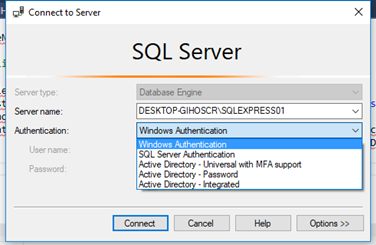
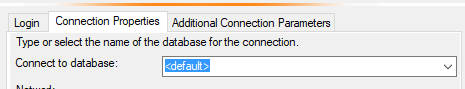


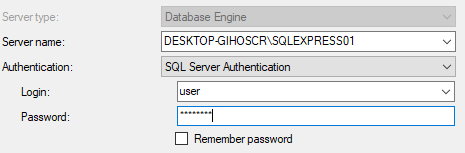
**Connecting to SQL Server**

To connect to an instance of SQL SERVER, you need to specify several items, regardless of how you connect:

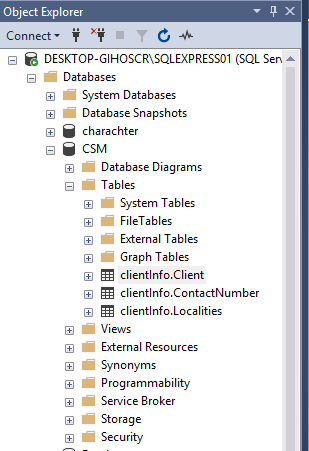
* The instance name: This must be in the format : **hostname\instancename**
* Database name: A default database can be assigned to a logon. If you don`t specify a database, you will connect to the master database or the default database designated by the database administrator
* Authentication : Windows Authentication or SQL SERVER Authentication

Account must be provisioned by a database administrator



**Working with Object Explorer**



Object Explorer is a graphical tool for managing SQL Server

instances and database. It is one of several SSMS windows

available from the **View** menu. Object Explorer provides direct

interaction with most SQL Server data objects such as tables,

views, and stored procedures. To display context-sensitive help

for an object in the tree view, right-click an object, such as a table.

The available options including query and script generators for

object definitions.

Operations performed in SSMS require appropriate permissions

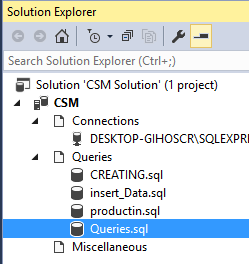
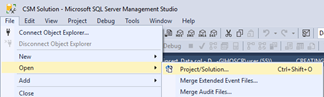
granted by a database administrator. Being able to see an object

or command does not necessarily imply permission to use the

object or issue the command. Selecting objects in the Object Explorer pane ddoes not change any connections made in other windows.

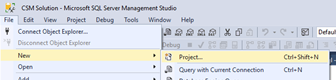
**Script Files and Projects**

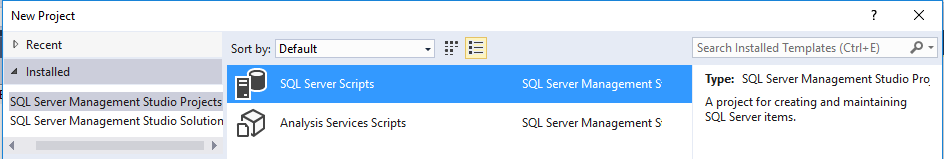
SSMS allows you to create and save T-SQL code in text files with a **.sql** file extention. In addition to working on individual script files, SSMS lets you organize files into solutions and projects. You can keep scripts for one project together, saving time by opening or closing all the files at the same time, you can open solutions, projects, or script files from SSMS or File Explorer.

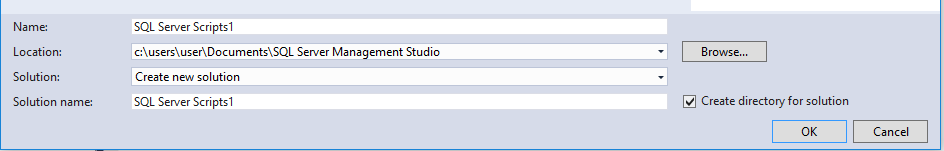


|  |  |  |
| --- | --- | --- |
| Object | Parent | Description |
| Solution | - | A solution is a conceptual container for projects. Solutions have a **.ssmssln** extention, and are always displayed at the top of the hierarchy. |
| Project | Solution | Projects contain queries (T-SQL scripts), database connection metadata, and other miscellaneous files. You can file any number of projects within a solution.Projects have a **.ssmssqlproj** extension. |
| Script | Project | T-SQL script files with a **.sql** extension are the basic files used to work with SQL Server. |

To create a new solution :







**Execute Queries**

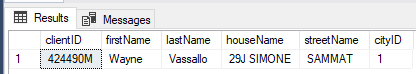
To execute T-SQL code in SSMS

* Select the portion of code you want to execute.
* If you do not select anything, the entire script will run.

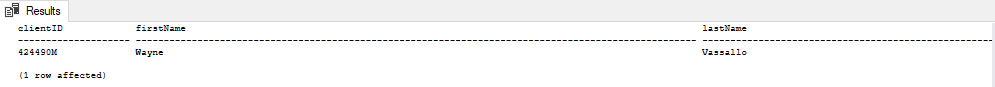
By default, SSMS will display the result in a new pane of the query window. To change the location and appearance of the result, click **Tool**, **Options**. CTRL-R brings up or removes the result pane.

SSMS enables results to be displayed in three different ways:

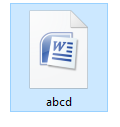
* **Grid:** A spreadsheet-like view, with row numbers and columns you can resize. Use **CTRL+D** to select Grid layout before executing query.



* **Text:** A Windows Notepad-like display that pads column widths. Use **CTRL+T** (Changed in SQL Server 2016 as it used to be CTRL+F) to select Grid layout before executing query.

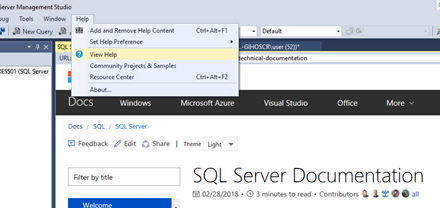


* **File:** Saves query results to a text with a **.rpt** extension. When you execute the query, you will be prompted for a location to save the file. You can then open the file with any application that reads text files. To send result to file, USE **CTRL+SHIFT+F** before running query.



**Using SQL Server 2016 Technical Documentation**

SQL Server 2016 Technical Documentation (Sometimes abbreviated as BOL(Books Online)) is the product documentation for SQL Server. BOL includes helpful information on SQL Server`s architecture and concepts, in addition to syntax reference for T-SQL. You can start BOL from the Help menu in SSMS, or from the query window. For context-sensitive help for T-SQL keywords, select the keyword and press **F1**. You can also view SQL Server 2016 Technical Documentation on the Microsoft website.



Before SQL Server 2014 there was a setup option to install SQL Server Book Online locally. In SQL Server 2016, you must download and install the SQL Server 2016 Technical Documentation separately.

The first time you invoke help, you will be prompted to specify whether you want to view SQL Server 2016 Technical Documentation locally or online. With online help, you will always access the latest information.

Module 2

Lesson 1

**Introducting T-SQL**

**About T-SQL**

T.SQL is Microsoft`s implementation of the industry standard Structured Language. The language was originally developed to support the new relational data model at International Business Machines (IBM) in the early 1970`s. Since then, SQL has become a standard of the American National Standards Institute (**ANSI**) and the International Organization for Standardization (ISO) in the 1980`s.]

An important concept to understand when working with T-SQL is that it is not a procedural language but a set-based and declarative language. When you write a query to retrieve data from SQL Server, you describe the data you wish to display; you do not tell SQL Server how to retrieve it. Instead of supplying a procedural list of steps to take, you provide the attributes of the data you seek.

Example you want to retrieve a list of customers who are located in Portland, a procedural approach might look like this:

**Procedual Approach**

Loop through each row in the data.

If the city is Portland, return the row; otherwise do nothing.

Move to next row.

End loop.

This procedural code has to contain logic to retrieve the data-to inspect the data to see if it meets your needs-and will be doing this for all the data in the table, whether or not it is relevant.

With a declarative language such as T-SQL, you will provide the attributes and values that describe the set you wish to retrieve. You do not have to specify how to retrieve the data, but you should identify what the data is.

For example , see the following pseudo-code :

**Declarative Language**

Return all the customers whose city is Portland

With T-SQL, the SQL Server 2016 Database Engine will determine the optimal path to access the data and return a matching set. Your role is to learn to write an efficient and accurate T-SQL code so you can properly describe the set you wish to retrieve.

* Transact-SQL is commonly referred as T-SQL
  + The querying language of SQL 2016
* SQL is declarative
  + Describe what you want, not the individual steps
* Structured Query Language (SQL)
  + Developed by IBM in the 1970`s
  + Adopted ANSI & ISO standard bodies
  + Widely used in the industry
    - PL/SQL (Oracle) SQL Procedural Language (IBM)

**Categories of T-SQL**

T-SQL statements can be organized into three categories:

**Data Manipulation Language (DML)** is the set of T-SQL statements that focuses on querying and modifying data. This includes SELECT, and modification statements such as INSERT, UPDATE, and Delete.

**Data Definition Language (DDL)** is the set of T-SQL statements that handles definition and life cycle of the database objects, such as tables, views and procedures. This includes statements such as CREATE, ALTER, and DROP.

**Data Control Language (DCL)** is the set of T-SQL statements used to manage security permissions for users and objects. DCL includes statements such as GRANT, REVOKE and DENY.

|  |  |  |
| --- | --- | --- |
| DML | DDL | DCL |
| Data Manipulation Language | Data Definition Language | Data Control Language |
| Used to query and manipulate data | used to define database objects | used to manage security permissions |
| SELECT, INSERT, UPDATE, DELETE | CREATE, ALTER, DROP | GRANT, REVOKE, DENY |

**TSQL Language Elements**

Like many programming languages, T-SQL Contains elements that you will use in queries.

You will use Predicates to filter rows, Operators to perform comparisons, Functions & Expressions to manipulate data or retrieve system information and Comment to document your code.

If you need to go beyond writing SELECT statements to create stored procedures, triggers, and other objects, you might use elements such as control-of-flow statements, variables to temporarily store values, and batch separators.

* Predicates & Operators
* Functions
* Variables
* Expressions
* Batch Separators
* Control of Flow
* Comments

**T-SQL Language Elements: Predicates & Operators**

The T-SQL language provides elements for specifying and evaluating logical expressions. In SELECT statements, you can use logical expressions to define filters for WHERE and HAVING clauses. You will write these expressions using predicates and operators.

* **Predicates**

Is an expression that evaluates to TRUE, FALSE, or UNKNOWN. Predicates are used in the search condition of [WHERE](https://docs.microsoft.com/en-us/sql/t-sql/queries/where-transact-sql) clauses and [HAVING](https://docs.microsoft.com/en-us/sql/t-sql/queries/select-having-transact-sql) clauses, the join conditions of [FROM](https://docs.microsoft.com/en-us/sql/t-sql/queries/from-transact-sql) clauses, and other constructs where a Boolean value is required.



* **IN**: used to determine whether a value matches any value in a list or subquery.
  + For example, WHERE day IN (1,5,6,10).
* **BETWEEN**: used to specify a range of values.
  + For example, WHERE rate BETWEEN 3 AND 7
* **LIKE**: used to match characters against a pattern.
  + For example, WHERE surname LIKE ‘%mith%’

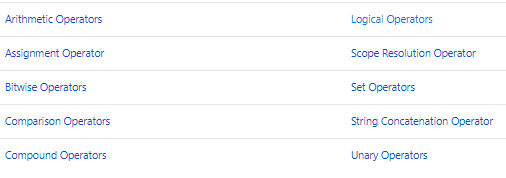
 During pattern matching, regular characters must exactly match the characters specified in the character string ( Example: WHERE surname LIKE ‘Smithx’). The data type of the column can be any character or date data type. There are certain characters within the pattern, called wildcard characters.

four types of wildcards; they are:

* 1. **Percent sign (%)**: It is used to represent or search any string of zero or more characters.
  2. **Underscore (\_)**: It is used to represent or search a single character.
  3. **Bracket ([]):** It is used to represent or search any single character within the specified range.
  4. **Caret (^):** It is used to represent or search any single character not within the specified range.
  + **Escape( '%\%%' ESCAPE '\'; ) :** used within an SQL statement to tell the driver that the escaped part of the SQL string should be handled differently**.**
* **Exists Function**: Specifies a subquery to test for the existence of rows. The exists function and subquery as an argument and returns and returns true if the subquery returns one or more rows, and returns false if it returns zero rows. The EXISTS operator operates on a subquery and returns a Boolean value either TRUE or FALSE.
  + TRUE if the subquery returns at least one row
  + FALSE if no rows are returned by the subquery
* **NOT EXISTS**: NOT EXISTS works like EXISTS, except the WHERE clause in which it is used is satisfied if no rows are returned by the subquery. Example: WHERE NOT EXISTS (SELECT NULL)
* **ANY / SOME , ALL**: The operators ANY and ALL are always used in combination with comparison operators. scalar\_expression { = | <> | != | > | >= | !> | < | <= | !< } ALL ( subquery ) The ANY/SOME operator evaluates to true if the result of an inner query contains at least one row that satisfies the comparison. The ALL Operator evaluates to true if the evaluation of the table column in an inner query returns all values of the column.
* **IS [NOT] NULL**: Determines whether a specified expression is NULL / NOT NULL. If the value of expression is NULL, IS NULL returns TRUE; otherwise, it returns FALSE. If the value of expression is NULL, IS NOT NULL returns FALSE; otherwise, it returns TRUE. Example: WHERE Weight < 10.00 OR Color IS NULL
* **Etc...**

# Operators

An operator is a symbol specifying an action that is performed on one or more expressions.



# Comparison Operators

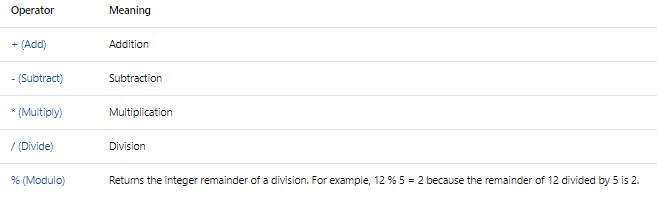
Comparison operators test for equality and non equality .Operators can be used on all expressions except expressions of the text, ntext, or image data types.

# 

# Logical Operators : For testing the validity of a condition

# Arithmetic Operators : Arithmetic operators perform mathematical operations



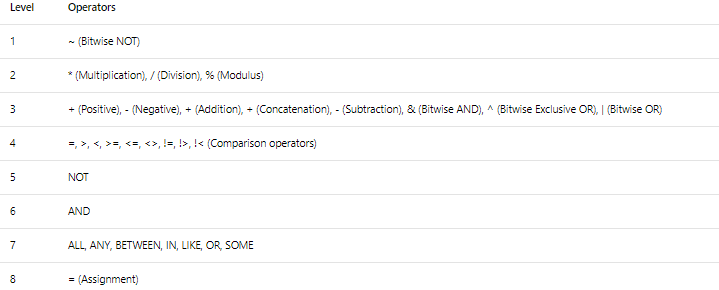
# String Concatenation

# An operator in a string expression that concatenates two or more character or binary strings, columns, or a combination of strings and column names into one expression (a string operator). For example SELECT 'book'+'case'; returns bookcase or SELECT (LastName + ', ' + FirstName) AS Name; returns wayne vassallo.

# Assignment

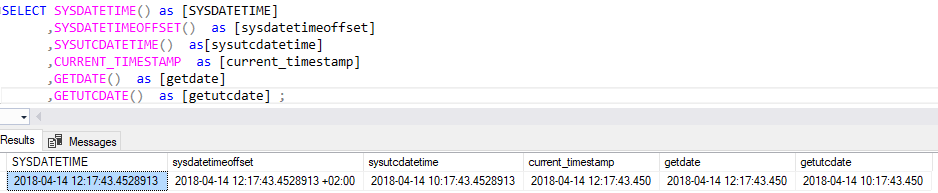
# For setting a value: ‘ = ‘ ex: DECLARE @MyCounter INT; SET @MyCounter = 1;

As with other mathematical environments, when a complex expression has multiple operators, operator precedence determines the sequence in which the operations are performed. The order of execution can significantly affect the resulting value:



**T-SQL Language Elements: Functions**

SQL Server 2016 provides a wide variety of functions for your T-SQL queries. They range from scalar functions, such as SYSDATETIME, which returns a single-valued result



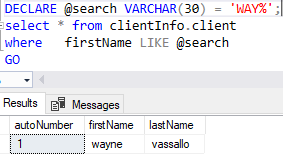
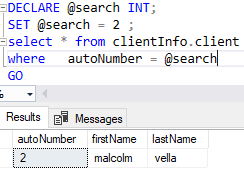
, the others that operate on and return entire sets such as windowing functions. (Window functions belong to a type of function known as a ‘set function’, which means a function that applies to a set of rows. The word ‘window’ is used to refer to the set of rows that the function works on.)

**T-SQL Language Elements : Variables**

Like many programming languages, T-SQL provides a means of temporary storing a value of a specific data type. However, unlike other programming environments, all user-created variables are local to the T-SQL batch that created them-and are visible only to that batch. There are no global or public variables available to SQL Server users.

To create a variable in T-SQL, you must give name, data type, and initial value. The name must start with a single @ symbol, and the data type must be system-supplied or user-defined, and stored in the database your code will run against. @@ reserved for system functions.

If your variable is not initialized in the DECLARE statement, it will be crated with a value of NULL and you can subsequently assign a value with the SET statement.

If persistent storage or global visibility for a value is needed, consider creating a table in a database for that purpose. SQL Server provides both temporary and permanent storage in databases.

**T-SQL Language Elements: Expressions**

T-SQL provides combinations of identifiers, symbols and operators that are evaluated by SQL Server to return a single result. These combinations are known as expressions, offering a useful and powerful tool for your queries. In SELECT statements you may use expressions :

* In the SELECT clause to operate on and/or manipulate columns.
* As CASE expression to replace values matching a logical expression with another value.
* In the WHERE clause to construct predicates for filtering rows
* As table expressions to create temporary sets used for further processing

Expressions may be based on a scalar (single-value) function, on a constant value, or on variables. Multiple expressions may be joined using operators if they have the same data type, or if the data type can be converted from a lower precedence to a higher precedence (for example, int to money).

The following example of an expression operates on a column to add an integer to the the result of the YEAR function on datetime column

Expression:

SELECT YEAR(orderdate) AS currentyear, YEAR(orderdate) + 1 AS nextyear FROM Sales.orders;

**T-SQL Language Elements: Control of Flow, Errors and Transactions**

While T-SQL is primarily a data retrieval language and not a procedural language, it does support a limited set of statements that provide some control of flow during execution.

Some of the commonly-used control statements include:

* **IF...ELSE** , for providing branching control based on a logical test.
* **WHILE**, for repeating a statement or block of statements while a condition is true
* **BEGIN.....END**, for defining the extents of a block T-SQL statements
* **TRY....CATCH**, for defining the structure of exception handling (error handling)
* **THROW**, for raising an exception and transferring execution to a CATCH block
* **BEGIN TRANSACTION**, for marking a block of statements as part of an explicit transaction. Ended by COMMIT TRANSACTION or ROLLBACK TRANSACTION.

|  |  |  |
| --- | --- | --- |
| Control of flow | Error Handling | Transaction Control |
| IF..ELSE | TRY | BEGIN TRANSACTION |
| WHILE | CATCH | ROLLBACK TRANSACTION |
| BREAK | THROW | COMMIT TRANSACTION |
| CONTINUE |  | ROLLBACK WORK |
| BEGIN...END |  | SAVE TRANSACTION |
| WAITFOR |  |  |

**T-SQL Language Elements: Comments**

For single lines: -- (double dash)

For a block comment: /\* \*/

EXAMPLE :

--insert Comment single line

/\*

THIS IS

A BLOCK

COMMENT

\*/

**T-SQL Language Elements: Batch Seperators**

SQL Server client tools, such as SSMS, send command to the database engine in sets called batches.

For most simple query purposes, batch seperators are not used, because you will be submitting a single query at a time. However, when you need to create and manipulate objects, you might need to separate statements into distinct batches.

* Batches are sets of commands sent to SQL Server a a unit
* Batchhes determine variable scope, name resolution
* To separate statements into batches, use a separator :
* SQL server tools use the GO keyword
* GO is not an SQL Server T-SQL command and it can be changed for the current query in QUERY | QUERY options or globally in Tools |options |Query Execution
* GO [count] executes the preceding batch [count] times

The following is an example of a CREATE TABLE and CREATE VIEW statement in the same batch:

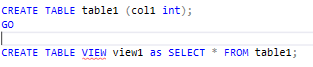
CREATE TABLE table1 (col1 int);

CREATE TABLE VIEW view1 as SELECT \* FROM table1;

The previous example returns the following error:

Msg 156, Level 15, State 1, Line 275 Incorrect syntax near the keyword 'VIEW'.

To resolve the error insert GO batch separator between the two CREATE statements



User-declared variables are considered local to the batch in which they are declared. If a variable is declared in one batch and referenced in another, the second batch would fail.

√

DECLARE @custid INT = 5;

SELECT custid,companyName , contactName

FROM Sales.Customers

WHERE custid = @custid

X

DECLARE @custid INT = 5;

GO

SELECT custid,companyName , contactName

FROM Sales.Customers

WHERE custid = @custid

Lesson 2

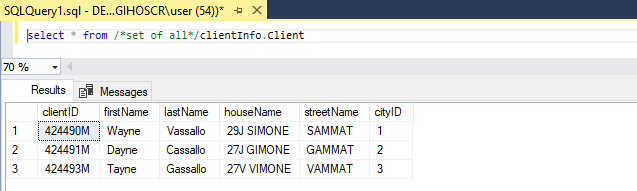
**Understanding Sets**

**The Set theory and SQL Server**

A set is defined as a ‘collection of definite, distinct objects considered as a whole’

For example a Customer table represents a Set –specifically, the set of all customers. You will see that the results of a SELECT statement also form a set. It is important to think of the entire set, instead of individual members, at all times. Working with sets requires thinking in terms of operations that occur ‘all at once’ instead of one at a time.

All members of a set must be (distinct) unique. In SQL Server, uniqueness is typically implemented using keys, such as a Primary Key column. However, when you start working with subsets of data, it’s important to know you can uniquely address each member of a set. There is no predefined order in a set, you must use ORDER BY clause if you need the return results in a certain order.



SET – TABLE CLIENT

Each row: Elements of a set called Members

**Set Theory applied to SQL Server Queries**

**There are a few considerations and recommendations to be aware of when writing efficient T-SQL queries:**

* Act on all elements at once – Query the whole table
* Use set based processing – Tell the engine what you want to retrieve
* Avoid cursors or loops – Do not process each item individuallt
* Members of a set must be unique – Define unique keys in a table
* No defined orders to result set –Use ORDER BY clause if results need to be ordered

Lesson 3

**Understanding Predicate Logic**

**Predicate Logic & SQL Server**

Predicate is a property or expression) that is either true or false. It is also known as a Boolean expression.

Predicates usually are embedded in a statement that does something wih the true or false result, such as WHERE clause to filter rows; a Case expression to match a value; or even a column constraint governing the range of acceptable values for that column in a table’s definition.

In most real world environments, you need to account for missing or unknown values, and extend your understanding of predicates from two possible outcomes (true or false) to three – true , false or unknown.

**Predicate Logic Applied to SQL Queries**

Uses for predicates

Filtering data in queries (in WHERE & HAVING clauses)

Providing conditional logic to CASE expressions

Joining tables (in the ON filter)

Defining subqueries (ex: in EXIST tests)

Enforcing data integrity

Control of flow (predicates have uses outside SELECT statements, such as in CHECK constraints to limit values permitted in a column, and in control-of-flow element such as IF statement.)

Lesson 4

**Understanding the Logical Order of Operations in Select Statements**

**Elements of a SELECT Statement**

The order in which you write a statement is not necessarily that in which the database engine will evaluate and process it .Database engines may optimize their execution of a query. Providing the accuracy of the result (as determined by the logical order) is retained.

|  |  |  |
| --- | --- | --- |
| Element | Expression | Role |
| SELECT | <select list> | Defines which columns to return |
| FROM | <table source> | Defines table(s) to query |
| WHERE | <search condition> | Filters returned data using a predicate |
| GROUP BY | <group by list> | Arrange rows by groups |
| HAVING | <earch condition> | Filter groups by predicates |
| ORDER BY | <order list> | sorts the result |
|  |  |  |

Not all elements will be present in every SELECT query. However, when an element is present, it will always be evaluated in the same order, with respect to the others present.

**Logical Query Processing**

The order in which a SELECT statement is written is not that in which it is evaluated and processed by the SQL Database Engine.

Example :

SELECT EmployeeID , YEAR(orderDate) AS OrderYear

FROM sales.orders

WHERE CustomerID = 71

GROUP BY EmployeeID , YEAR(orderDate)

HAVING COUNT(\*) > 1

ORDER BY EmployeeID, OrderYear

Notice the alias OrderYear made in the SELECT statement was used in the ORDER BY clause but not in the GROUP BY clause. That is because of the logical order the SELECT statement is evaluated after the other steps because the run-time order evaluation determines what data is available to clauses downstreamfrom one another. As follows :

**Logical Order**

FROM sales.orders

WHERE CustomerID = 71

GROUP BY EmployeeID , YEAR(orderDate)

HAVING COUNT(\*) > 1

SELECT EmployeeID , YEAR(orderDate) AS OrderYear

ORDER BY EmployeeID, OrderYear

Module 3

Lesson 1

**Writing Simple SELECT Statements**

**Retrieving Columns from a TABLE or View**

The SELECT clause specifies the columns from the source table(s) or view(s) that you want to return as the result set of the query. In addition to columns from the source table, you can add others in the form of calculated expressions.

The FROM clause specifies the name of the table or view that is the source of the columns in the SELECT clause. To avoid errors in table or view name resolution, it is best to include the SCHEMA & OBJECT- for example Sales.Customer.

If the table or view name contains irregular characters, such as spaces or other special characters, you need to delimit, or enclose the name. T-SQL supports the use of the ANSI standard double quotes “SALES Order Details”, and the SQL Server specific square brackets [Sales Order Details].

End all statements with a semicolon (;) character. In SQL Server 2016, semicolon is optional terminator for most statements. However, future version will require its use. For current usages when a semicolon is required, such as some common table expressions (CTEs) and some Service Broker statements, the error message returned for a missing semicolon are often cryptic. Therefore, you should adopt the practice of terminating all statements with semicolon.

**Display Columns**

To display columns in a query, you need to create a comma-delimited list. The order of the columns in your list will determine their display in the output, regardless of the order in which you have defined in the source table.

T-SQL supports the use of the asterisk, or “star” character (\*) to retrieve all columns from the source table. While the asterisk is suitable for quick test, avoid using it in production work, as changes made to the table will cause the query to retrieve all current columns in the table`s current defined order. This could cause a bug or other failures in reports or applications expecting a known number of columns returned in a defined order. Furthermore, returning data that is not needed can slow down your queries and cause performance issues if the source table contains a large number of rows.

By using explicit columns list in your SELECT clause, you will always achieve the desired results, providing the columns exist in the table. If a column is dropped, you will receive an error that will help identify the problem and fix your query.

**Using Calculated Expression**

A SELECT statement can perform calculations and manipulations which can change the source column data, and use a built in T-SQL functions.

As the result will appear in a new column, repeated once per row of the result set, calculated expressions in a SELECT clause must be scalar- they must return only a single value.

Calculated expressions can operate on other columns in the same row, on built-in functions, or a combination of the two:

**Calculated expression**

SELECT unitprice , qty , (unitprice \* qty)

FROM sales.OrderDetails;

The result appears as follows

Unitprice qty

---------- -------- ------------

14.00 12 168.00

9.8 10 98.00

To provide a name to the new column use a column alias : (unitprice \* qty) as TOTAL

**Create a calculated Column**

SELECT empid , lastname , hiredate , YEAR(hiredate)

FROM HR.Employees;

The result :

Empid lastname hiredate

---------- -------- ------------ -----------

1 Davis 2002-05-01 2002

2 Funk 2002-08-01 2002

Calculations are scalar, returning one value per row

|  |  |
| --- | --- |
| Operator | Description |
| + | ADD or Concatinate |
| - | Subtract |
| \* | Multiply |
| / | Divide |
| % | Modulo |
|  |  |

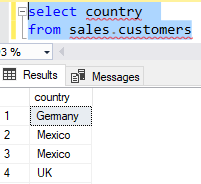
Lesson 2

**Eliminating Duplicates with Distict**

**SQL Sets and Duplicate Rows**

T-SQL query results are not always true (unique) sets. The row retrieved by a query are not a guaranteed to be unique, even when they come from a source table that uses a primary key to differentiate each row.

For example, consider a query that returns country names from the Sales.Customers table:

****

It shows duplicate county names and it would give a wrong answer to the question ‘’ How many countries are represented among our customers?’’

****

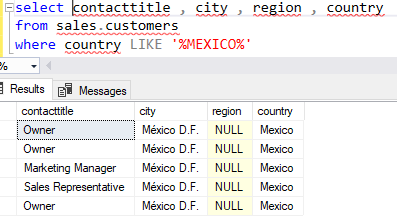
The reason for this output is that by default, a SELECT clause contains a hidden default ALL statement:

****

The query will return one result for each row in the Sales.Customers table; however as only the country column is specified, you will see just this column for 91 rows.

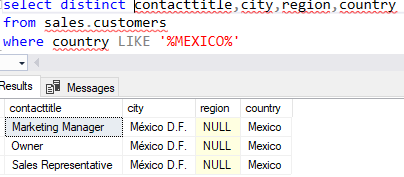
**Understanding Distinct**

Replacing the default SELECT ALL clause with SELECT DISTINCT will filter out duplicates in the result set and specifies that the result set must contain only unique rows. Distinct option operates only on the set of columns returned by the SELECT clause. It does not take into account any other unique columns in the source table. Distinct also operates on all the columns in the SELECT list not just the first one.

** Notice duplicate rows**

The select statement returned 5 rows which 3 of them are duplicates

Whit the SELECT DISTINCT 3 rows where returned instead of 5 as duplicates where removed

****

The logical order of operations also ensures

that Distinct operator will remove rows that

may have already been processed by WHERE,

* DISTINCT Specifies that only unique rows can appear in the result set
* Removes duplicates based on column list results, not source table
* Provides uniqueness across set of selected columns
* Removes rows already operated on by WHERE,HAVING and GROUP BY clauses
* Some queries may improve performance by filtering out duplicates before execution of SELECT clause

HAVING and GROUP BY clauses.

Lesson 3

**Using Column and Table Aliases**

**Use Aliases to Refer to Columns**

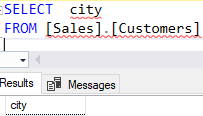
Column aliases can be used to reliable columns when returning the result of a query. In T-SQL, there are 3 multiple methods of creating a column alias, with identical output results.

**AS keyword :** SELECT city AS cities

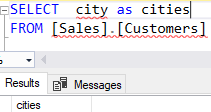
**Alias with Equals Sign :** SELECT city = cities

**Alias with following column name (not recommended):** SELECT city cities

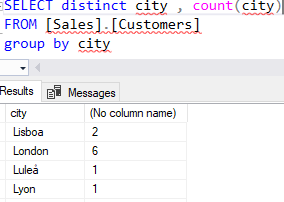
When retrieving data from a table or view, a T-SQL query will name each column after its source.

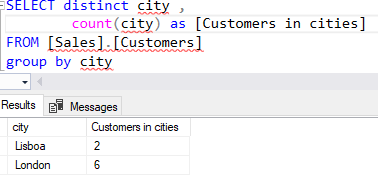


You can reliable columns by using aliases in the SELECT clause.



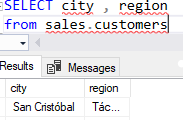
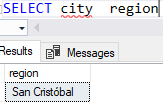
However, columns created with expressions will not be named automatically. Column aliases can be used to provide custom column headers.





**Accidental Alias**

Column aliases can also be accidentally created, by omitting a comma between two column names in the SELECT list :

 (without the comma)

**Use Aliases to Refer to Tables**

Aliases can also be used in the From clause to refer to a table; this can improve readability and save redundancy when referring to a tale elsewhere in a query.

Table Alias using **AS** :

SELECT city region

from sales.customers as cs

Table and column Aliases combined :

SELECT cs.city , cs.region

from sales.customers as cs

**The impact of Logical Processing Order on Aliases**

When using aliases an issue can arise. Aliases created in the SELECT clause may not be referred to in others in the query- such as WHERE or HAVING clause. This is due to the logical order query processing. The WHERE and HAVING clause are processed before the SELECT clause. An Exception to this is the ORDER BY clause.

select city as c

from sales.customers

order by c

Lesson 4

**Writing Simple CASE Expressions**

A CASE expression extends the ability of a SELECT clause to manipulate data as it is retrieved. Often you need to substitute a value of a column with another value. In real-world environments, CASE is often used to help make cryptic data that is held in a column more meaningful.

A CASE expression returns a scalar value(single-valued) based on conditional logic, often with multiple conditions. As scalar value, it may be used wherever single values can be used. CASE can be used in any statement or clause that allows a valid expression. For example, you can use CASE in statements such as SELECT, UPDATE, DELETE and SET, and in clauses such as select\_list, IN, WHERE, ORDER BY, and HAVING..

**Using CASE Expressions in SELECT Clauses**

CASE expressions are not statements, nor do they specify the control of programmatic flow. Instead they are used in SELECT and other clauses to return the result of an expression. In SELECT clause, CASE behaves as calculated column requiring an alias.

**Forms of CASE Expressions**

In T-SQL CASE expressions may take one of two forms- simple CASE expressions or searched(Boolean) CASE.

* The simple CASE expression compares an expression to a set of simple expressions to determine the result.
* The searched CASE expression evaluates a set of Boolean expressions to determine the result.

Both formats support an optional ELSE argument.

### Using a SELECT statement with a simple CASE expression

Within a SELECT statement, a simple CASE expression allows for only an equality check; no other comparisons are made. The following example uses the CASE expression to change the display of product line categories to make them more understandable.

SELECT ProductNumber, Category =

CASE ProductLine

WHEN 'R' THEN 'Road'

WHEN 'M' THEN 'Mountain'

WHEN 'T' THEN 'Touring'

WHEN 'S' THEN 'Other sale items'

ELSE 'Not for sale'

END,

Name

FROM Production.Product

ORDER BY ProductNumber;

### SELECT statement with a searched CASE expression

Within a SELECT statement, the searched CASE expression allows for values to be replaced in the result set based on comparison values. The following example displays the list price as a text comment based on the price range for a product.

SELECT ProductNumber, Name, "Price Range" =

CASE

WHEN ListPrice = 0 THEN 'Mfg item - not for resale'

WHEN ListPrice < 50 THEN 'Under $50'

WHEN ListPrice >= 50 and ListPrice < 250 THEN 'Under $250'

WHEN ListPrice >= 250 and ListPrice < 1000 THEN 'Under $1000'

ELSE 'Over $1000'

END

FROM Production.Product

ORDER BY ProductNumber ;

Module 4

**Querying Multiple Tables**

Lesson 1

**Understanding Joins**

**The FROM Clause and Virtual Tables**

FROM clause determines which table or tables will be the source of rows for the query.

FROM clause is processed first, so as a result, any table aliases you create there may be referenced in the SELECT clause. Table aliases are optional, except in the case of a self join query.

**WITHOUT ALIAS**

select contactname , qty

FROM sales.customers

JOIN Sales.CustOrders ON SALES.Customers.custid = SALES.CustOrders.custid

**WITH ALIAS**

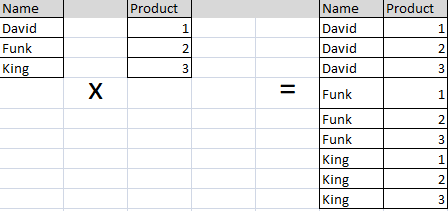
select c.contactname , co.qty

FROM sales.customers AS c

JOIN Sales.CustOrders AS co ON c.custid = co.custid

**Join terminology: Cartesian Product**

Cartesian product occurs when two input tables are joined without considering any logical relationships between them. With no information about relationships, the SQL Server query will output all possible combination of rows.

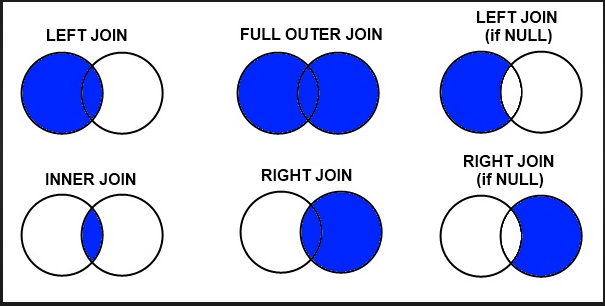


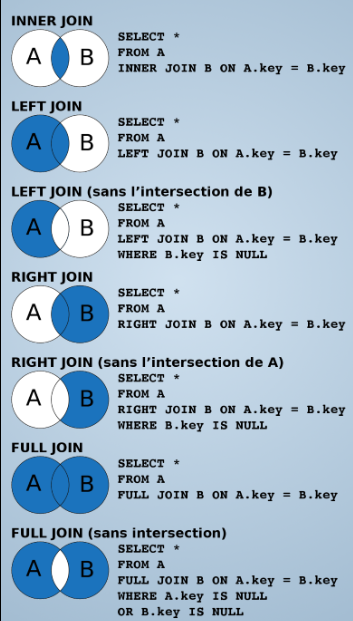
**Overview of Join Types**

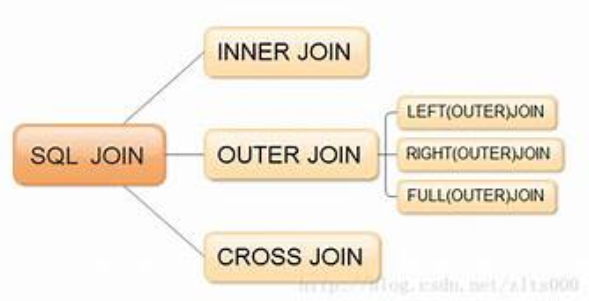
* A Cross join operator (CROSS JOIN) adds all possible combinations of the two input tables rows to virtual table. Any filtering of the rows will happen in a WHERE clause.

A cross-join that does not have a 'where' clause gives the Cartesian product. Cartesian product result-set contains the number of rows in the first table, multiplied by the number of rows in second table. (Resulting in a higher dimension in the resulting set). Both the joins give same result. Cross-join is SQL 99 join and Cartesian product is Oracle Proprietary join.

* An inner join operator(INNER JOIN, or just JOIN) first creates a Cartesian product, and then filters the results using the predicate supplied in the ON clause, removing any rows from the virtual table that do not satisfy the predicate. The inner join is a very common type of join that retrieves rows with matching attributes across tables, such as Customers to Orders by a common custid.
* An outer join operator (LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN) first create a Cartesian product, and like an inner join, filters the result to find rows, that match in each table. However, all rows from one table are preserved, and added back to the virtual table after the initial filter is applied. NULLs are placed on attributes where no matching values are found.







Lesson 2

**Querying with Inner Joins**

**Understanding Inner Joins**

To retrieve data that has been stored across multiple tables, you will often need to reassemble it via inner join queries. An inner join begins its logical processing phase as a Cartesian product, which is then filtered to remove any rows the don`t match the predicate.

--ANSI SQL-89 SYNTAX

SELECT c.companyname, o.orderdate

FROM sales.customers AS c, sales.orders AS o

WHERE c.custid = o.custid;

--ANSI SQL-92 SYNTAX

SELECT c.companyname, o.orderdate

FROM sales.customers AS c join sales.orders AS o

ON c.custid = o.custid;

* Returns only Rows where a match is found in both input tables
* Matches rows based in attributes supplied in predicate
  + ON clause in SQL-92 syntax(preferred)
  + WHERE clause in SQL-89 syntax
* Why filter in ON clause?
  + Logical separation between filtering for purposes of join and filtering results in WHERE
  + Typically no difference to query optimizer
* If join predicate operator is - , also known as equi-join

**Inner Join Examples**

**Join based on single matching attribute**

SELECT c.companyname, o.orderdate

FROM sales.customers AS c

JOIN sales.orders AS o

ON c.custid = o.custid;

**Join based on multiple matching attributes**

**(composite join)**

SELECT DISTINCT e.city, e.country

FROM sales.customers AS c

JOIN sales.orders AS e

ON c.city = e.city

AND c.country = e.country

**Join including more than two tables**

SELECT c.companyname, o.orderdate

FROM sales.customers AS c

JOIN sales.orders AS o

ON c.custid = o.custid

JOIN sales.ordersDetails AS od

ON o.orderid = od.orderid;

Lesson 3

**Querying with Outer Joins**

**Understanding Outer Joins**

OUTER JOIN is used to join two tables even if there is not a match. An OUTER JOIN can be used to return a list of all the customers and the orders even if no orders have been placed for some of the customers. A keyword, RIGHT or LEFT, is used to specify which side of the join returns all possible rows**.** NULLs are added where there is no match.

* Returns all rows from one table and any matching rows from the second table
* One table`s rows are ‘preserved’
  + Designated with LEFT, RIGHT, FULL keyword
  + All rows from preserved table output the result set
* Matches from other table retrieved
* Additional rows added to results for no matched rows
  + NULLs added in places where attribute do not match

The next example returns a list of all the customers and the SalesOrderID for the orders that have been placed, if any.

SELECT c.CustomerID, s.SalesOrderID

FROM Sales.Customer c

LEFT OUTER JOIN Sales.SalesOrderHeader s ON c.CustomerID = s.CustomerID

The query joins all customers to the orders placed in 2002. Then the results are restricted to those where there is no match.

SELECT c.CustomerID, s.SalesOrderID

FROM Sales.Customer c

LEFT OUTER JOIN Sales.SalesOrderHeader s ON c.CustomerID = s.CustomerID

and s.OrderDate between '1/1/2002' and '12/31/2002'

WHERE s.SalesOrderID IS NULL

Lesson 4

**Querying with Cross Joins and Self Joins**

**Understanding Cross Joins**

* Cross join queries create a Cartesian product.
* Cross JOIN combine each row from first table with each row from second table
* All possible combinations output
  + Inner join starts with Cartesian product, adds filter
  + Outer Join takes Cartesian output, adds back no-matching rows(with Null placeholders)
* Due to Cartesian product output, not typically a desired form of join
* Some useful exceptions:
  + Creating a table of numbers, with a row for each possible value in range
  + Generating a large volume of data for testing. When cross joined to itself a table with 100 rows can readily generate 10,000 output rows

**CROSS JOIN EXAMPLE**

SELECT c.companyname , o.orderdate

FROM sales.customers AS c CROSS JOIN sales.orders AS o;

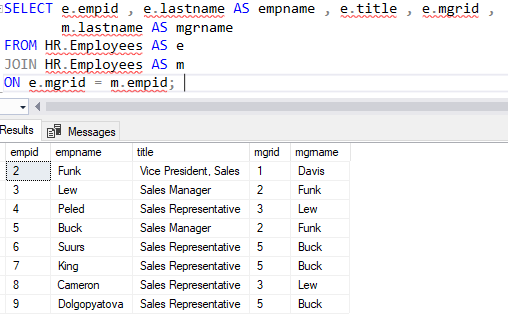
**Understanding Self Joins**

A table can be joined to itself in a self-join.

Use a self-join when you want to create a result set that joins records in a table with other records in the same table. To list a table two times in the same query, you must provide a table alias for at least one of instance of the table name to help the query processor determine whether columns should present data from the right or left version of the table.

The following example query retrieves employees and their matching manager information from the Employees table joined to itself:

**SELF JOIN EXAMPLE**



Module 5

**Sorting and Filtering Data**

Lesson 1

**Sorting Data**

**Using the ORDER BY clause**

* In the logical order the ORDER BY clause is the last phase of SELECT statement to be executed, therefore it has access to aliases defined in the SELECT list.
* ORDER BY enables you to control the sorting of rows as they are output from the query to the client application in (default) ascending or descending (ASC | DESC), optionally, limit the rows returned to a specified range (OFFSET-FETCH).
* The order in which rows are returned in a result set are not guaranteed unless an ORDER BY clause is specified.
* If the query uses a DISTINCT option, any columns in the ORDER BY list must be included in the SELECT list.
* ORDER BY may also include a COLLATE clause, which provides a way to sort by a specific character collation, instead of the collation of the column in the table.
* Each column may be provided with a separate order

### Basic syntax

### Specifying a single column defined in the select list

SELECT ProductID, Name FROM Production.Product

WHERE Name LIKE 'Lock Washer%'

ORDER BY ProductID;

**Specifying a column that is not defined in the select list**

SELECT ProductID, Name, Color

FROM Production.Product

ORDER BY ListPrice;

#### Specifying an alias as the sort column

#### SELECT name, SCHEMA\_NAME(schema\_id) AS SchemaName

FROM sys.objects

WHERE type = 'U'

ORDER BY SchemaName;

#### Specifying an expression as the sort column

SELECT BusinessEntityID, JobTitle, HireDate

FROM HumanResources.Employee

ORDER BY DATEPART(year, HireDate);

#### Specifying both ascending and descending order

SELECT LastName, FirstName FROM Person.Person

WHERE LastName LIKE 'R%'

ORDER BY FirstName ASC, LastName DESC ;

### Specifying a collation

CREATE TABLE #t1 (name nvarchar(15) COLLATE Latin1\_General\_CI\_AI)

GO

INSERT INTO #t1 VALUES(N'Sánchez'),(N'Sanchez'),(N'sánchez'),(N'sanchez');

-- This query uses the collation specified for the column 'name' for sorting.

SELECT name

FROM #t1

ORDER BY name;

-- This query uses the collation specified in the ORDER BY clause for sorting.

SELECT name

FROM #t1

ORDER BY name COLLATE Latin1\_General\_CS\_AS;

Lesson 2

**Filtering data with Predicates**

**Filtering data in the WHERE clause with Predicates**

* To limit the rows that are returned by a query, you will need to add a WHERE clause to your SELECT statement, following the FROM clause.
* WHERE clauses are constructed from a search condition which, in turn, is written as a predicate expression. The predicate provides a logical filter through which each row must pass.
* Only rows returning TRUE in the predicate will be output to the next logical phase of the query. Therefore values of FALSE or UNKNOWN will be filtered out.
* Remember that, logically, the WHERE clause is the next phase in query execution after FROM, so it will be processed before other clauses such as the SELECT, which means the WHERE clause will be unable to refer to column aliases.
* Can be optimised by SQL Server to use indexes.
* Can reduce network traffic and client memory usage.

**WHERE clause syntax**

WHERE <search\_condition>

**Typical WHERE clause**

WHERE <column> <Predicates or Operators> <expression>

**Filtering example**

SELECT orderid , custid , YEAR(orderdate) AS orderyeat

FROM sales.orders

WHERE YEAR(orderdate) = 2006;

Lesson 3

**Filtering data with TOP & OFFSET-FETCH**

**Filtering in the SELECT Clause Using the TOP option**

* Limits the rows returned in a query result set to a specified number of rows or percentage of rows
* When TOP is not used in conjunction with the ORDER BY clause it returns the set of rows in an undefined order.
* Use this clause to specify the number of rows returned from a SELECT statement or affected by an INSERT, UPDATE, MERGE, or DELETE statement.
* TOP cannot be used in an UPDATE and DELETE statements on partitioned views.
* TOP cannot be combined with OFFSET and FETCH in the same query expression (in the same query scope)

Basic syntax

-- Select the first 10 employees hired most recently.

SELECT TOP(10)JobTitle, HireDate

FROM HumanResources.Employee

ORDER BY HireDate DESC;

#### -- Using TOP with a variable.

DECLARE @p AS int = 10;

SELECT TOP(@p)JobTitle, HireDate, VacationHours

FROM HumanResources.Employee

ORDER BY VacationHours DESC;

#### -- Specifying a percentage

SELECT TOP(5)PERCENT JobTitle, HireDate

FROM HumanResources.Employee

ORDER BY HireDate DESC;

-- Including tie values

SELECT TOP(10) WITH TIES JobTitle, HireDate

FROM HumanResources.Employee

ORDER BY HireDate DESC;

Limiting the rows affected by DELETE, INSERT, or UPDATE

#### Using TOP to limit the number of rows deleted

DELETE FROM Purchasing.PurchaseOrderDetail

WHERE PurchaseOrderDetailID IN

(SELECT TOP 10 PurchaseOrderDetailID

FROM Purchasing.PurchaseOrderDetail

ORDER BY DueDate ASC);

#### Using TOP to limit the number of rows inserted

The OUTPUT clause displays the rows that are inserted into the EmployeeSales table.

INSERT TOP(5)INTO dbo.EmployeeSales

OUTPUT inserted.EmployeeID, inserted.FirstName, inserted.LastName, inserted.YearlySales

SELECT sp.BusinessEntityID, c.LastName, c.FirstName, sp.SalesYTD

FROM Sales.SalesPerson AS sp

INNER JOIN Person.Person AS c

ON sp.BusinessEntityID = c.BusinessEntityID

WHERE sp.SalesYTD > 250000.00

ORDER BY sp.SalesYTD DESC;

#### Using TOP to limit the number of rows updated

UPDATE HumanResources.Employee

SET VacationHours = VacationHours + 8

FROM (SELECT TOP 10 BusinessEntityID FROM HumanResources.Employee

ORDER BY HireDate ASC) AS th

WHERE HumanResources.Employee.BusinessEntityID = th.BusinessEntityID

**Filtering in the ORDER BY Clause Using OFFSET-FETCH**

* While the TOP option is used by many SQL Server professionals as a method of retrieving only a certain range of rows, it also has disadvantages:
* TOP is proprietary of T-SQL and SQL Server.
* TOP does not support skipping a range of rows.
* Because TOP depends on an ORDER BY clause, you cannot establish the rows filtered by TOP and another to determine the output display

Like TOP, OFFSET-FETCH enables you to return only a range of the rows selected by your query. However it adds the functionality to supply a starting point (an offset) and a value to specify how many rows you would like to return (a fetch value). This provides a convenient technique for paging through results

When paging, you need to consider that each query with an OFFSET-FETCH clause runs independently of any previous or subsequent query. There is no server-side state maintained, and you will need to track your position through a result set via client-side code.

## Limitations in Using OFFSET-FETCH

* ORDER BY is mandatory to use OFFSET and FETCH clause.
* OFFSET clause is mandatory with FETCH. You can never use, ORDER BY … FETCH.
* TOP cannot be combined with OFFSET and FETCH in the same query expression.
* The OFFSET/FETCH rowcount expression can be any arithmetic, constant, or parameter expression that will return an integer value. The rowcount expression does not support scalar sub-queries.

**OFFSET** { integer\_constant | offset\_row\_count\_expression } { ROW | ROWS }

Specifies the number of rows to skip, before starting to return rows from the query expression. The argument for the OFFSET clause can be an integer or expression that is greater than or equal to zero. You can use ROW and ROWS interchangeably.

**FETCH** { FIRST|NEXT } <rowcount expression> { ROW|ROWS } ONLY

Specifies the number of rows to return, after processing the OFFSET clause. The argument for the FETCH clause can be an integer or expression that is greater than or equal to one. You can use ROW and ROWS interchangeably. Similarly, FIRST and NEXT can be used interchangeably.

**Syntax**

[ORDER BY { order\_by\_expression [ ASC | DESC ] } [ ,...n][<offset\_fetch>] ]

<offset\_fetch> =

{OFFSET { integer\_constant | offset\_row\_count\_expression } { ROW | ROW }

 [FETCH { FIRST | NEXT } {integer\_constant | fetch\_row\_count\_expression } { ROW | ROWS } ONLY]}

**Example 1**Skip first 10 rows from the sorted result set and return the remaining rows.

SELECT firstName + ' ' + lastName

FROM Employees

ORDER BY firstName OFFSET 10 ROWS;

**Example 2-** Skip first 10 rows from the sorted result set and return next 5 rows.

SELECT firstName + ' ' +lasttName

FROM Employees

ORDER BY firstName OFFSET 10 ROWS FETCH NEXT 5 ROWS ONLY;

**Example 3-** Retrieving the 50 most recent rows as determined by the order date.

Select orderid , custid , empid , orderdate

FROM sales.orders

ORDER BY orderdate DESC

OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY;

Lesson 4

**Working with Unknown Values**

**Three-Valued Logic**

In SQL Server not all data being compared may be presented. You need to plan for and act on the possibility that some data is missing or unknown. SQL Server will mark the missing value as NULL. A NULL is neither TRUE nor FALSE but it is marked for UNKNOWN, which represents the third value in the three-valued logic.

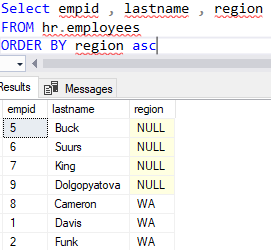
You can determine whether X > Y is TRUE or FALSE when you know the values of both X and Y. But what does SQL Server return for the expression X > Y when Y is missing?

SQL Server will return an UNKNOWN marked as NULL. You will need to account for possible presence of NULL in your predicate logic, and in the values stored in columns marked with NULL. You will need to write queries that use three-valued logic to account for three possible outcomes- TRUE, FALSE and UNKNOWN.

**Handling NULL in Queries**

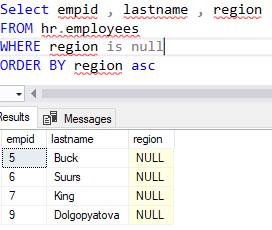
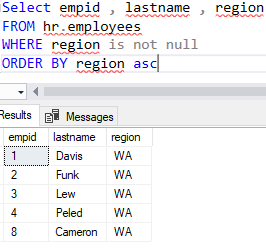
Query filters, such as ON, WHERE, and the HAVING clause, treat NULL like a FALSE result

The ORDER BY clause sorts the NULLs together and first (when ASC) – A behaviour you cannot override.



To test for null values in a query, use **IS NULL** or **IS NOT NULL** in the WHERE clause

.

Module 6

**Working with SQL Server Data Types**

Lesson 1

**Introducing SQL Server Data Types**

**SQL Server Data Types**

SQL Server defines a set of system data types for storing data in columns, holding values in variables, operating on data in expressions, and passing parameters stored procedures.

Data types specify the type, length, precision, and scale of data.

Developers might also extend the supplied set by creating aliases to built-in types and even by producing new user-defined types using Microsoft.NET Framework.

SQL Server data types can be grouped into the following categories:

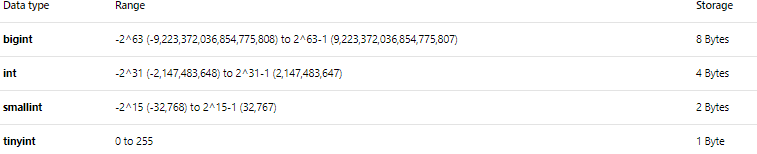
1. **Exact numeric**. These data types store data with precision, either as:
   1. Integers-whole numbers with varying degrees of capacity.
   2. Decimals-decimal numbers with control over both the total number of digits stored and the number of digits to the right of the decimal place.
2. **Approximate numeric**. These data types allow inexact values to be stored, typically for use in scientific calculations.
3. **Binary strings**. These data types allow binary data to be stored, such as byte streams or hashes, to support common applications.
4. **Other data types**. This catch-all category includes several special types that fall outside the other categories. Some of these data types can be used as column data types (and therefore accessible to queries). This category also includes data types not used for storage, but rather for special operations, such as cursor manipulation or creating table variables for further processing. If you are a report writer, you may encounter data types used for columns, such as **uniqueidentifier** and **xml** data types.

**Numeric Data Types**

Numeric data types fall into one of two subcategories- **exact numeric** and **approximate numeric**.

**1.Exact numeric data types**:

* **Integer data types.**



* **Decimal data types.**

These data types are specified with the total number of digits to be stored(precision) and the number of digits to the right of the decimal place(scale). The larger the precision, the greater the storage cost.

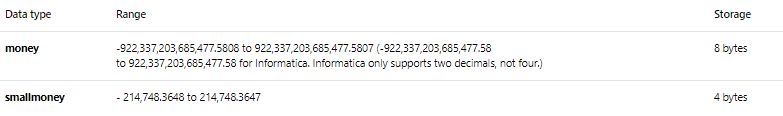
The maximum total number of decimal digits that will be stored, both to the left and to the right of the decimal point is 38. The precision must be a value from 1 through the maximum precision of 38. The default precision is 18.

****

NOTE that there is no difference between the decimal data type and the numeric data type- decimal is the ISO standard-compilation name for the data type; numeric is used for backward compatibility with earlier versions of SQL Server.

* **Money data types.**

Data types that represent monetary or currency values.

****

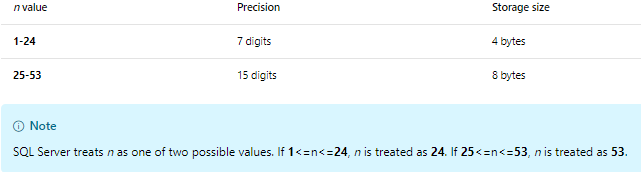
* **Boolean data types.**

The BIT data type is used to store Boolean values (true/false) which are treated by SQL Server as numeric values- 1 for TRUE and 0 FRO false.

**2.Approximate numeric data types**:

These are less accurate, but have more capacity than the exact numeric data type. The approximate numeric data types store values in scientific notation which, because of lack precision, loses accuracy.

* The **float** data type takes an optional parameter of the number of bits used to store the mantissa of the float number in scientific notation.



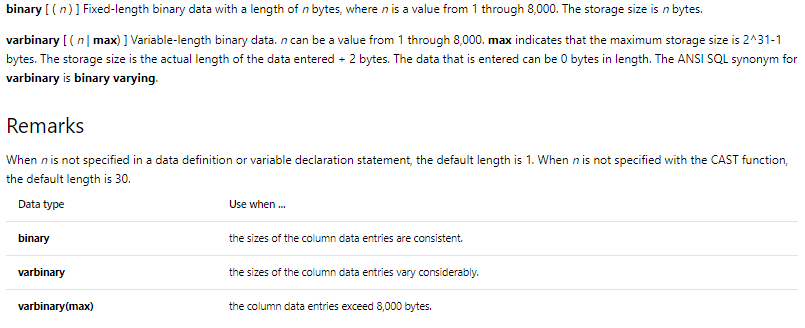
* The **rea**l data type is a synonym for float data type with a mantissa value of 24 (that is **float(24)** )

**3.Binary string Data Types**

Binary string data types allow a developer to store binary information, such as serialized files, images, byte streams, and other specialized data.

If you are considering using the binary data types, note the differences in range and storage requirements, compared with numeric and character string data.

You can choose between fixed-width and variable-width binary strings.

****

The **image** data type is also a Binary string type but marked for removal in future version of SQL Server. **Varbinary(max)** should be used instead.

**Converting to Binary Data Type :**

The two leading characters ‘0x’ indicates that the output is a binary string.

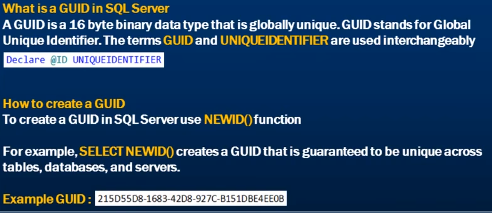
SELECT CAST (1234 AS binary(4)) AS Result;



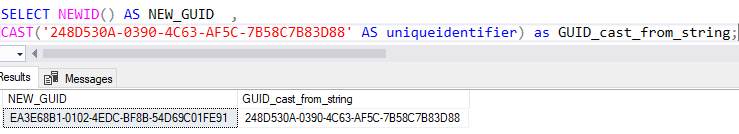
**4.Other Data Types**

In addition to numeric and binary types, SQL Server also supplies some other data types for specialized used cases, such as storage and processing of XML, generation and storage of globally unique identifiers (GUIDs), the representation of hierarchies, and more:

* The xml data type allows the storage and manipulation of Extensible Markup Language data (XML). The advantage of the xml data type over storing XML in a character data type is that the xml data type allows XML nodes and attributes to be queried within a T-SQL query using XQury expressions. The xml data type also optionally allows an XML schema to be enforced. Each instance of xml data type can store up to 2GB of data
* The uniqueidentifier data type allows the generation and storage of globally unique identifiers (GUIDs) stored as 16-byte value. Values for the uniqueidentifier data type can be generated within SQL Server by using NEWID() system function; they can also be generated by external applications or converted for string values.



Creating GUIDs for the uniqueidentifier Data Type



* The hierarchyid data type is used to simplify the recording and querying of hierarchical relationships between rows in the same table-for example, the levels an organisation chart or a bill of materials. SQL server stores hierarchyid as a variable-lenght binary data type; the hierarchy is exposed through built-in functions.
* The rowversion data type stores an automatically generated 8-byte binary value in a table that increments each time a row is inserted or updated. Rowversion values do not store date or time information, but can be used to detect weather a row has been changed since it was last read by a client application (for instance when implementing optimistic locking).
* The spatial data types are special complex data types for dealing with geometric and geographical data.
  + The geometry data type is used to store data in a Euclidean (flat) coordinate system. Arrays of coordinates defining lines, polygons and other simple geometric shapes can be stored in the geometry data type. Special built-in methods are available for carrying out operations on geometry data.
  + The geography data type is used to store data in a round-earth coordinate system such as GPS latitude and longitude coordinates. As with the geography data type, shape definitions can be stored in the geography type, then in built-in methods used to operate geography data.
* The sql\_variant type is a special type that may be used to store data of any other built-in datatype for instance, enabling integer, decimal and character data for typical database designs, and its use may indicate design problems. The sql\_variant data type is listed here for completeness.

The following data types may not be used as a data type columns in tables or views, they are used as variables or parameters for stored procedures:

* The cursor data type is used to reference a cursor object, which allows row-by-row preccesing of a data set.
* The table data type is used to define a table variable or stored procedure parameter, which has many of the properties of a standard database table but exists only in the context of the session for which it was created. Table data type are typically used to temporary store the results of T-SQL statements for further processing later.

**Data type precedence**

When combining or comparing different data types in your queries, such as in a WHERE or JOIN clause, SQL Server will need to convert one value from its data type to another value. Which data type is converted depends on the precedence between the two.

SQL Server uses the following precedence order for data types:

When an operator combines two expressions of different data types, the rules for data type precedence specify that the data type with the lower precedence is converted to the data type with the higher precedence. If the conversion is not a supported implicit conversion, an error is returned. When both operand expressions have the same data type, the result of the operation has that data type.

Any implicit conversion is transparent to the user; therefore, if it fails(such as when your operation requires converting between data types for which no implicit conversion exists), you will need to explicitly convert data type by using CAST or CONVERT functions.

**When are Data Types Converted?**

* When data is moved, compared, or combined with other data
* During variable assignment
* When using any operator that involves operands of different types
* When T-SQL code explicitly converts one data type to another, using CAST or CONVERT function

**Implicit Conversion Example – Integer Data types**

DECLARE @myTinyInt AS tinyint = 25;

DECLARE @mybigInt AS bigint = 99999999999;

SELECT @myTinyInt + @mybigInt;

****

SQL Server will automatically attempt to perform an implicit conversion from a lower-precedence data type to a higher precedence data type. Implicit data type conversion is transparent to the user, unless the conversion fails.

Lesson 2

**Working with Character Data**

**Character Data types**

**Character data types in SQL Server are categorised by two characteristics:**

* **Support for either fixed-width or variable-width data:**
  + Fixed width data is always stored at a consistent size, regardless of the number of characters in the character data. Any unused space is filled with padding.
  + Variable-width data is stored at the size of the character data, plus a small overhead.
* **Support for either a single-bite character or a multi-bite character set:**
  + A single-byte character set supports up to 256 different characters, stored as a one byte per character. By default, SQL Server uses the ASCII character set to interpret this data.
  + A multi-byte character set supports more 65,000 different characters by sorting each character as multiple bytes-typically two bytes per character, but sometimes more. SQL Server uses the UNICODE USC-2 character set to interpret this data.

The four available character data types support all possible combinations of these characteristics:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Type | Fixed Width? | Variable Width? | Single-Byte Characters? | Multi-Byte Characters? |
| char | yes |  | yes |  |
| nchar | yes |  |  | yes |
| varchar |  | yes | yes |  |
| nvarchar |  | yes |  | yes |

Definition for columns or variables takes an optional value that defines the maximum length of the character data to be stored. You will almost always need to specify a value for the string length if the maximum length is not supplied; the default value is one character.

The varchar and nvarchar data types support the storage of very long strings of characters data by using max for its value. Use of varchar(max) and nvarchar(max) replaces the use of the deprecated text and ntext types

|  |  |  |
| --- | --- | --- |
| Data Type | Range | Storage |
| char(n) | 1-8000 characters | n bytes, padded 2\*n bytes, padded |
| nchar(n) | 1-4000characters | n bytes, padded 2\*n bytes, padded |
| varchar(n) | 1-8000 characters | Actual lenght + 2 bytes |
| nvarchar(n) | 1-8000 characters | Actual lenght + 2 bytes |
| varchar(max) | up to 2GB | Actual lenght + 2 bytes |
| nvarchar(max) |  | Actual lenght + 2 bytes |
|  |  |  |
|  |  |  |

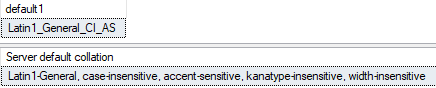
Note : All character data is delimited with a single quotation marks.

* + Single-byte character data is indicated with quotation marks alone-for example ‘SQL Server’.
  + Multi-byte character data is indicated by single quotation marks with the prefix N (for National)-for example N’SQL Server’. Then N prefix is always required, even when inserting the data into a column or a variable with a multi-byte type.

**Collation**

A collation is a collection of properties that determine several aspects of character data, including:

* Language or locate, from which is derived:
  + Character set
  + Sort order
* Case sensitivity - CS
* Accent sensitivity - AS



Note: A default collation is configured during the installation of SQL Server, but can be overridden on a per-database or per-column bases. As you will see, you might also override the current collation for some character data by explicitly setting a different collation in your query.

When querying, it is important to be aware of the collation settings for your character data-for example whether it is case-sensitive or not.

The COLLATE can be used to override the collation of a column and force different collation to be applied when the query run; this example forces a case sensitive and accent-sensitive comparison using Latin1\_General sort rules and character table by adding COLLATE clause to the WHERE clause.



CASE SENSITIVE:

select firstname from hr.employees

where firstname collate Latin1\_General\_CS\_AS like 'D%'

CASE SENSITIVE:

select firstname from hr.employees

where firstname collate Latin1\_General\_CS\_AS like 'd%' NO REULT

(unless there would be a name starting with ‘d’)



NOTE : Note that the database-level collation settings apply to database object names(such as tables and views) as well as to character data.

Example, in database with a case-sensitive default collation, the table names ‘’HR.Employees’’ and ‘’HR.employees’’ would refer to 2 different objects. In a database with Case-Insensitive ‘’HR.Employees’’ and ‘’HR.employees’’ would refer to the same object.

View COLLATION

all SQL SERVER collations that can be used:

SELECT name, description FROM fn\_helpcollations();

To view the server default collation:

SELECT CONVERT (varchar, SERVERPROPERTY('collation')) as default1 ;

or

EXECUTE sp\_helpsort;

All collations supported by SQL 2017

SELECT name, description FROM sys.fn\_helpcollations();

View databases collation:

SELECT name, collation\_name FROM sys.databases;

OR

SELECT CONVERT (varchar, DATABASEPROPERTYEX('TSQL2012', 'collation'));

View columns collation:

SELECT name, collation\_name FROM sys.columns

WHERE name = N'<insert character data type column name>';

**String Concatenation**

The CONCAT function takes at least two or more data values as arguments and returns a string value with the input value concatenated together.

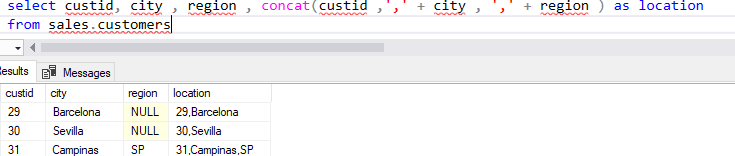
If any of the input data values is not of a character data type, it will be implicitly converted to a character data type.

Any NULL values will be converted to an empty string.

The + (plus) operator and the CONCAT function can both be used in concatenate strings in SQL 2016

**Using CONCAT**

Converts input values to strings and converts NULL to empty string

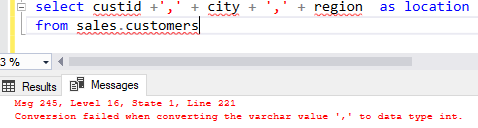


Notice when there is a null the **‘,’** is not inserted

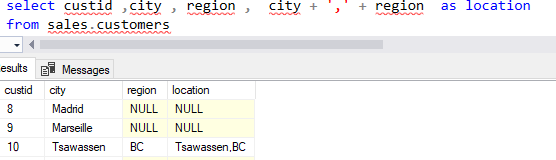
**Using the + (plus)**

No conversion of NULL or data type

No conversion of data type:



No conversion of NULLs, when any of the string values concatenated with the + operator is NULL, the output string will be NULL.



**Character String Functions**

In addition to retrieving character data as is from SQL Server, you may also need to extract portions of text or determine the location of characters within a string. SQL Server provides a number of built in-functions to accomplish these tasks.



Some of these functions include:

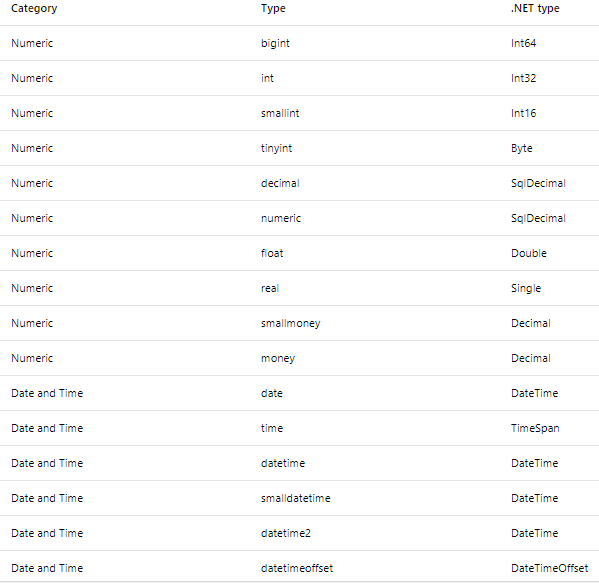
* **Format** –

Syntax:

FORMAT ( value, format [, culture ] )

Allows you for locale-aware formatting of date/time and number values to a character string based on a .NET format string, with an optional culture parameter.

The following table lists the acceptable data types for the *value* argument together with their .NET Framework mapping equivalent types.



This example shows the use of the FORMAT function to format money value as currency in various locales:

DECLARE @m money = 120.595

SELECT @m as unformated\_value ,

FORMAT (@m, 'C','zh-cn') as zh\_cn\_currency,

FORMAT (@m, 'C','en-us' )as en\_us\_currency,

FORMAT (@m, 'C','de-de')as de\_de\_currency;



* **Substring** –

Allows you to return part of a character string given a starting point and a number of characters to return.

(Returns part of a character, binary, text, or image expression in SQL Server)

Syntax:

SUBSTRING ( expression ,start , length )



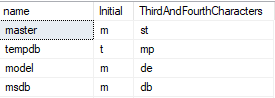
### Using SUBSTRING with a character string

SELECT name,

SUBSTRING(name, 1, 1) AS Initial ,

SUBSTRING(name, 3, 2) AS ThirdAndFourthCharacters

FROM sys.databases



* **LEFT and Right allows you to return a number of characters from the left or right of a string-**

SELECT LEFT('abcdefg',2); SELECT RIGHT('abcdefg',2);

-- --

Ab fg

* **LTRIM and RTRIM allows you to Return a character string after truncating all trailing spaces**

SELECT RTRIM('Four spaces are after the period in this sentence. ') + 'Next string.';

****

* **TRIM**

Removes the space character char(32) or other specified characters from the start or end of a string.

TRIM ( [ characters FROM ] string )

* **LEN and DATALENGHT**

Allows you to query metadata about the number of characters or the numbers of stored bytes in a string

--('!...# . test ') ORIGINAL NO OF CHARACTERS 15, WITH SPACE ON THE RIGHT 19

SELECT

TRIM( '.,! ' FROM '!...# . test ') AS TRIMMED,

DATALENGTH( '!...# . test ') AS BYTES\_NOT\_TRIMMED ,

DATALENGTH (TRIM( '.,! ' FROM '!...# . test ')) AS BYTES\_TRIMMED,

LEN( '!...# . test ') AS CHARACHTERS\_NOT\_TRIMMED,

LEN (TRIM( '.,! ' FROM '!...# . test ')) AS CHARACHTERS\_TRIMMED

****

* **CHARINDEX**

Allows you to query the start position of a string within another string. If the target string is not found, CHARINDEX returns 0

CHARINDEX ( expressionToFind , expressionToSearch [ , start\_location ] )

### Returning the starting position of an expression

### The following example returns the position at which the sequence of characters bicycle starts in the DocumentSummary column of the Document

DECLARE @document varchar(64);

SELECT @document = 'Reflectors are vital safety' +' components of your bicycle.';

SELECT CHARINDEX('bicycle', @document);



### Searching from a specific position

The following example uses the optional start\_location parameter to start looking for vital at the fifth character of the DocumentSummary

DECLARE @document varchar(64);

SELECT @document = 'Reflectors are vital safety' + ' components of your bicycle.';

SELECT CHARINDEX('vital', @document, 5);



* **Replace**

Allows you to substitute one string for another within a target string.

REPLACE ( string\_expression , string\_pattern , string\_replacement )

SELECT REPLACE('abcdefghicde','cde','xxx');

------------

abxxxfghixxx

DECLARE @n VARCHAR(20) = '123-1213/234'

SELECT REPLACE(REPLACE (@n , '-' , '') , '/', '')



* **UPPER and LOWER**

For performing character case conversions

UPPER ( character\_expression ) / LOWER ( character\_expression )

SELECT UPPER(RTRIM(LastName)) + ', ' + FirstName AS Name

FROM HR.EMPLOYEES

ORDER BY LastName;



* **REVERSE**

Returns the reverse order of a string value.

SELECT FirstName, REVERSE(FirstName) AS Reverse

FROM HR.EMPLOYEES



AN EXAMPLE on this lesson: how to find how many times a specific letter is shown in a row ?

select LEN('ABCDEEEEFG') - LEN(REPLACE('ABCDEEEFG','E',''))

RESULT = 4

And many other more ,........

**The like predicate**

The LIKE character string can be used to check a character string for a match with a pattern

**% (percentage)** represents a string of any length





**\_ (underscore)** represents a single character



**[<Charachter>-<Character>]** represents a single character within the supplied list

SELECT FirstName

FROM HR.EMPLOYEES

where FIRSTNAME like N'[a-z]%'



SELECT FirstName

FROM HR.EMPLOYEES

where FIRSTNAME like N'[RDG]%'

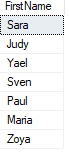


**[^<Character list or range>]** represents a single character not in the specific list or range.

SELECT FirstName

FROM HR.EMPLOYEES

where FIRSTNAME like N'[^RDG]%'



Escape – used to set an escape character meaning you can search for a character that is a wild card character but to treat it literal, rather than a wild card.

--searching for 10%

where @s like '10\%' ESCAPE '\'

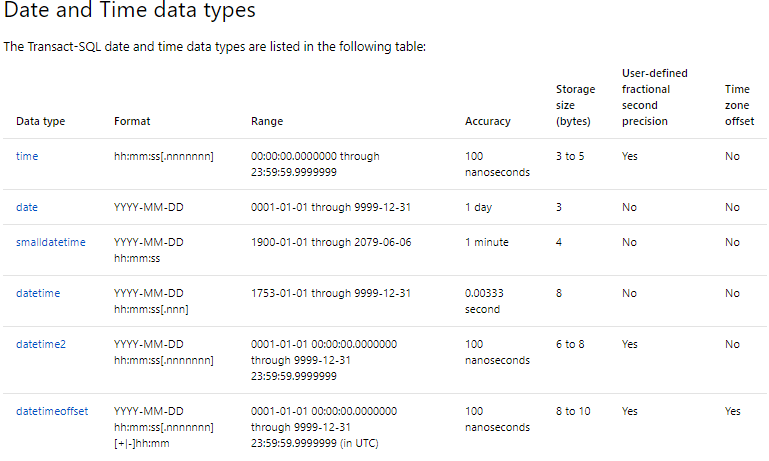
Lesson 3

**Working with Date and Time**

Before SQL Server 2008, there were only two data types for date and time data : datetime and smalldatetime.

In SQL Server 2008, Microsoft introduced four new data types: datetime2, date , time and datetimeoffset.

In SQL Server 2012, Microsoft introduced new functions working with partial data from date and time data types (such as DATEFROMPARTS) and performing calculations on dates (such as EOMONTH).



**Entering Date and Time Data Types Using Strings**

To use date and time data in your queries, you will need to use character strings (often referred to as string literals) which are delimited, like all other strings in SQL Server, with single quotes. SQL Server will implicitly convert the string literals to date and time values. (You might also explicitly convert string literals with the T-SQL CAST and CONVERT function.

SQL Server can interpret a wide variety of string literal formats as dates but, for consistency and to avoid issued with other languages or nationality interpretation; it is recommended that you use a neutral format, such as ‘YYYYMMDD’. To represent February 12, 2014, you would use the literal ‘20140212’,

String Literals Example (Returens order date of August 25, 2007):

SELECT orderid,custid,empid,orderdate

from sales.orders

where orderdate = '20070825'



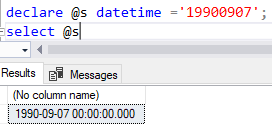
**Working Seperatly with Date and Time**

Datetime, smalldatetime, datetime2, and datetimeoffset include date and time data

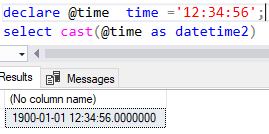
If only date is specified, time set to midnight (all zeros)

If only time is specified , date set to base date (January 1,1900)

Datetime with no time :



Default Date Example:



**Querying Date and Time Values**

When querying date and time types, you might need to consider both the date and time portions of the data to return the results you expected.

In this example, a user is trying to query all the sales orders with an order date of August 25, 2007:

Midnight Time Values Example

select orderdate

from sales.orders

WHERE orderdate = '20070825'



By the previous example we can see that as only the date was specified; as mentioned before the time will turn into midnight (all zero) ,therefore only orderdates that happened in midnight would show in the result set.

One way to be certain of returning all the orders for August 25,2007-regardelss of the time portion of the orderdate column-would be to query the data with a range, rather than a single value:

select orderdate

from sales.orders

WHERE orderdate >= '20070825'

AND orderdate < '20070826';

**Date and Time Functions**

SQL Server provides a number of functions designed to manipulate date and time data:

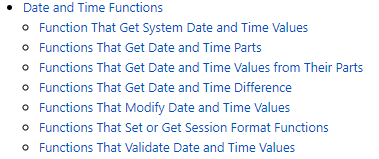
Functions that return current date and time, offering you choices between various return types, in addition to wether to include or exclude time zone information.

Functions that return parts of date and time values, enabling you to extract only the portion you of a date or time that your query requires. Note that DATENAME and DATEPART offer functionality similar to one another. The difference between them is the return type.

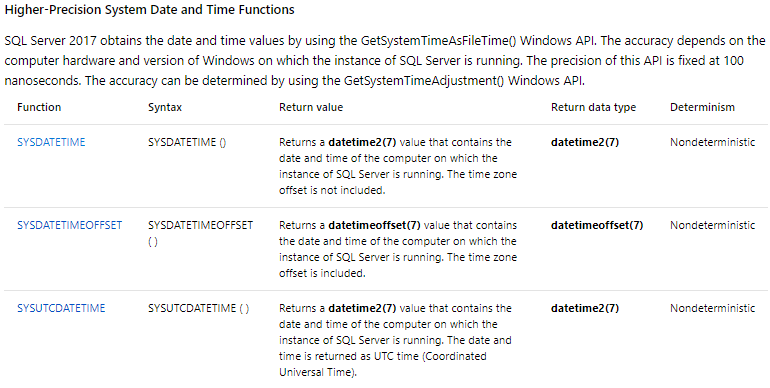
Functions that return date and time typed data from components such as separately supplied year, month, day. This offers alternative to providing dates as sring literals. Note that these functions require all parts of the target date/time data to be provided.

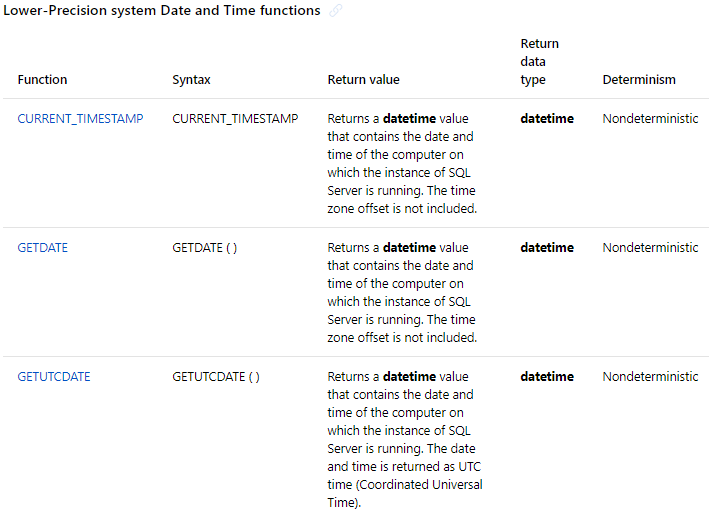
Functions that modify date and time values, including increment dates, to calculate the last day of a month, and to alter time zone offset information.

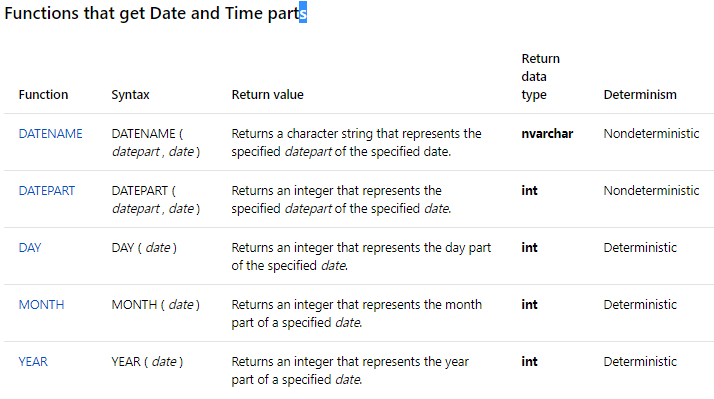
Functions that examine date and time values, returning metadata or calculations about intervals between input dates.

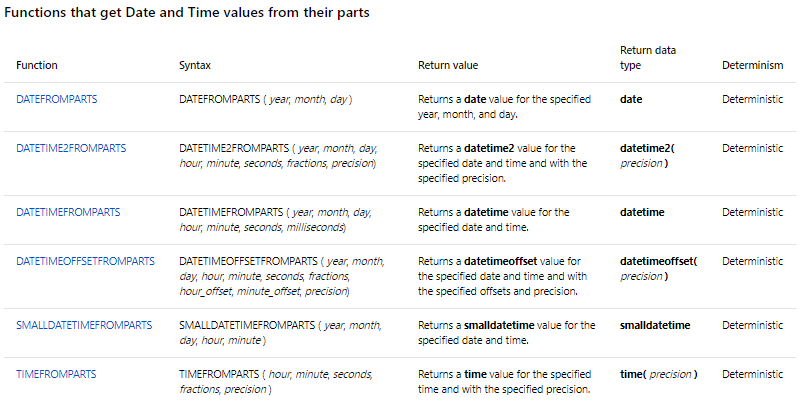


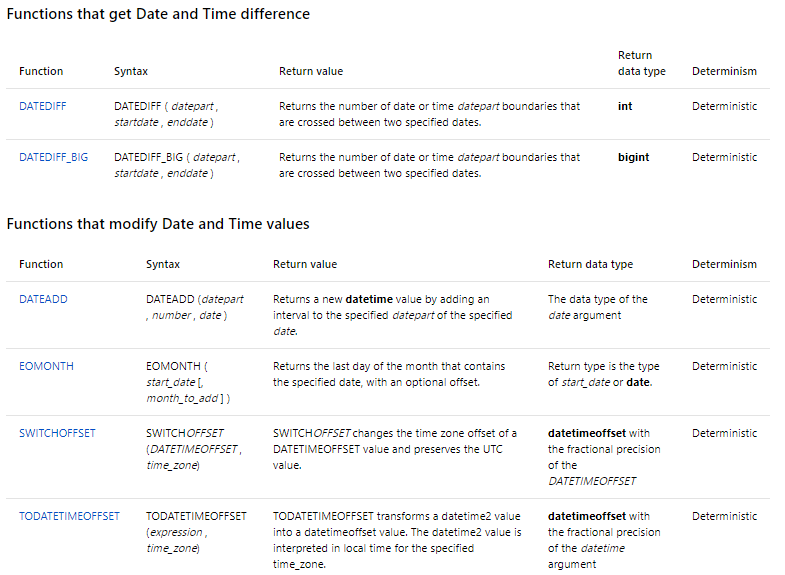
### Functions that get system Date and Time values

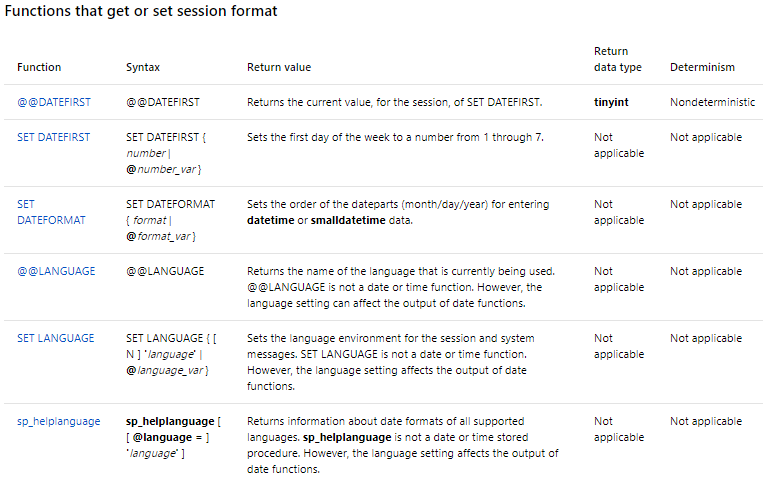


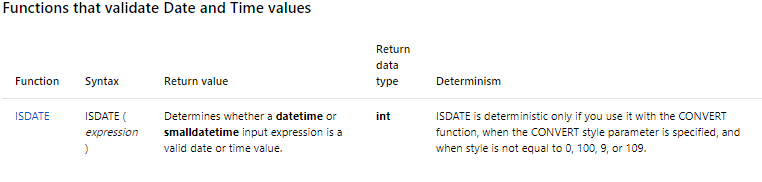












**AN EXTRA :**

AT TIME ZONE

**AT TIME ZONE** implementation relies on a Windows mechanism to convert **datetime** values across time zones.

## Syntax

inputdate AT TIME ZONE timezone

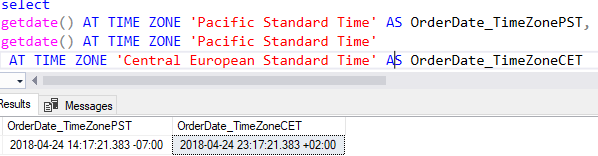
inputdate  
Is an expression that can be resolved to a **smalldatetime**, **datetime**, **datetime2**, or **datetimeoffset** value.

timezone  
Name of the destination time zone. SQL Server relies on time zones that are stored in the Windows Registry. All time zones installed on the computer are stored in the following registry hive: **KEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Time Zones**. A list of installed time zones is also exposed through the [sys.time\_zone\_info (Transact-SQL)](https://docs.microsoft.com/en-us/sql/relational-databases/system-catalog-views/sys-time-zone-info-transact-sql?view=sql-server-2017) view.

## Return Types : Returns the data type of  datetimeoffset

## Return Value : The datetimeoffset value in the target time zone.

Example :



Module 7

**Using DML to Modify Data**

Lesson 1

**Adding data to tables**

**Using INSERT to ADD Data**

In SQL the INSERT statement is used to add one or more rows to a table.

INSERT syntax

INSERT [INTO] <Table or view> [ (column\_list) ] - - column list is optional but this way it`s safer

VALUES ( [ColumnName or an expression or DEFAULT or NULL])

Whit this form, called INSERT VALUES, you can specify the columns that will have values placed in them and the order in which the data will be presented for each row inserted in the table. In addition, you can provide the values for those columns as a coma separated list.

When inserting values, the keyword DEFAULT is used when you have no data to insert in that column which has a default constraint so that the default value will be used.

When inserting values, the keyword NULL is used when you have no data to insert in a nullable column and no default constraint is specified.

INSERT VALUES EXAMPLE

This is an example to show you how you can insert more than one row at a time and how DEFAULT and NULL work, and also that there is no need to insert the value of a column where there is an autoincrement. Unless you want to turn off the autoIncrementing feature by issuing the statement Set

Identity\_Insert [TableName] On, as in:

Set Identity\_Insert [TableName] On

-- --------------------------------------------

Insert TableName (pkCol, [OtherColumns])

Values(pkValue, [OtherValues])

-- ---- Don't forget to turn it back off ------

Set Identity\_Insert [TableName] Off

CREATE TABLE #temp\_tbl\_products

(

productID int IDENTITY(1,1),

productName varchar(100) not null,

productAmmount int,

productDescription varchar(100) default 'N/A' not null,

productMadeFrom varchar(100) null

);

insert into #temp\_tbl\_products

(productName,productAmmount ,productDescription,productMadeFrom)

values ('',6,Default,NULL)

('',6,Default,NULL) ,

('',10,'China','description in here'),

('',0,'Denmark','yo mama');

SELECT \* FROM #temp\_tbl\_products



**Using INSERT with Data Providers**

T-SQL also supports using the output of other operations to provide value for INSERT.

You can pass the result of a SELECT clause or the output of a procedure to the INSERT clause.

INSERT with SELECT EXAMPLE (insert rows from another table)

insert into #temp\_tbl\_products

(productName,productAmmount ,productDescription,productMadeFrom)

SELECT productname , unitprice ,COALESCE([description],'N/A') , madeFrom

FROM production.products;

The COALESCE was used so that;

if the column description from the source table returned null values

‘N/A’ would be the default inserted as the column we are going to insert into is not null

Stating that:

You must provide column values or Default, or Null, for each column.

Result sets from stored procedures (or even dynamic batches) may also be used as input to an INSERT statement. This form of INSERT, called INSERT EXEC, is conceptually similar to INSERT SELECT and will present the same considerations.

Inserting rows into a Table from a Stored Procedure

CREATE PROC proc\_select\_productions

AS

SELECT productname , unitprice ,'N/A' , NULL

from production.products AS ThisDB;

insert into #temp\_tbl\_products

(productName,productAmmount ,productDescription,productMadeFrom)

exec proc\_select\_productions

SELECT \* FROM #temp\_tbl\_products

**Using SELECT INTO**

In T-SQL, you can use SELECT INTO statement to create and populate a new table with the results of a SELECT query.

Select into cannot be used to insert rows into an existing table.

A new table is created, with a schema defined by the column in the SELECT list.

Each column in the new table will have the same name, data type, and nullable as the corresponding column (or expression) in the select list.

It does not copy constraints or indexes of the source table.

(continuing on the previous codes with temporary tables)

select \* into #newTable

from #temp\_tbl\_products

or another example

select productname , unitprice

INTO newTable

from production.products

where unitprice > 10

NOTE : the use of SELECT INTO requires permission to create table objects in the destination database. Do not try to put this clause inside a view, because it will only work once. If a table cannot be created when the view is activated, an error ill occur after the first of the view.

Lesson 2

**Modifying and Removing Data**

**Using Update to Modify Data**

UPDATE statement is used to change existing data in a table or a view.

UPDATE operates on a set of rows, either defined by a condition in a where clause or defined in a join. It uses a SET clause that can perform one or more assignments separated by commas, to allocate new values to the target. The WHERE clause in an UPDATE statement has the same structure as a WHERE clause in a SELECT statement.

NOTE: An UPDATE without a WHERE clause, and/or a join, will target all rows that are not filtered out of the operation. Use UPDATE statement with caution.

UPDATE EXAMPLE (updating a list price where product10 needs to have productunitprice plus + 25)

UPDATE Production.Product

SET unitPrice = unitPrice + 25

where productid = 10;

**Using Merge To Modify Data**

Performs insert, update, or delete operations on a target table based on the results of a join with a source table.

MERGE modifies data, based on one or more conditions:

* When the source data matches the data in the target, it updates data.
* When a source data has no match in the target, inserts data
* When target data has no match in the source, it deletes the target data.

**(Performance Tip:** The conditional behaviour described for the MERGE statement works best when the two tables have a complex mixture of matching characteristics. For example, inserting a row if it does not exist, or updating the row if it does match. When simply updating one table based on the rows of another table, improved performance and scalability can be achieved with basic INSERT, UPDATE, and DELETE statements. For example:

INSERT tbl\_A (col, col2)

SELECT col, col2

FROM tbl\_B

WHERE NOT EXISTS (SELECT col FROM tbl\_A A2 WHERE A2.col = tbl\_B.col);**)**

NOTE: Because the T-SQL implementation of MERGER support the:

**WHEN NOT MATCHED BY SOURCE** clause,

MERGE is more than an upsert operation-because it also deletes, it is a delupsert or something similar.

**MERGE Example**

---creating temporary table product

CREATE TABLE #temp\_products

(

productID int not null,

productName varchar(100) not null,

productQuantity int not null,

comment varchar(100) default null,

);

insert into #temp\_products

(productID,productName ,productQuantity ,comment)

values (1,'hat',6,NULL),

(2,'t-shirt',6,NULL) ,

(3,'jersey',10,NULL),

(4,'shirt',20,NULL);

----inserting all data from #temp\_products into a new created temporary table #temp\_products\_backup

select \*

into #temp\_products\_backup

from #temp\_products

---adding more products in #temp\_products

insert into #temp\_products

(productID,productName ,productQuantity ,comment)

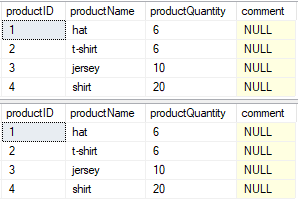
values (5,'jacket',21,NULL),

(6,'boxers',9,NULL)

NOW WE HAVE #temp\_products with more items than then #temp\_products\_backup

SELECT \* from #temp\_products

SELECT \* from #temp\_products\_backup



* When mathched if there is ‘removed’ in comment then insert getdate else leave comment as it is
* When not matched by target insert data
* When not matched by source add ‘removed’ to the comment

MERGE #temp\_products\_BACKUP AS trgt --target

USING #temp\_products AS src --source

--matching login

on trgt.productid = src.productid

WHEN MATCHED

--PRODUCTS ARE IN BOTH TABLES , update target table

THEN UPDATE SET COMMENT = ( CASE WHEN TRGT.comment LIKE '%Removed%'

THEN cast(getdate() as varchar(20))

ELSE TRGT.COMMENT

END

)

WHEN NOT MATCHED by TARGET

--BY TARGET I.E records are in source but not Target

THEN INSERT (productID , productName, ProductQuantity, comment)

values(src.productID , src.productName, src.ProductQuantity, cast(getdate() as varchar(20)))

WHEN NOT MATCHED BY SOURCE

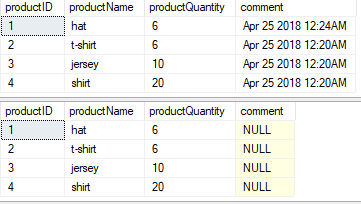
--i.e RECORDS are IN TARGET BUT NOT IN SOURCE

THEN UPDATE SET comment = trim(replace(trgt.comment, ' removed' , '') )+' Removed '

;

SELECT \* from #temp\_products

SELECT \* from #temp\_products\_backup



Lesson 3

**Generating Automatic Column Values**

**Using IDENTITY**

You may need to automatically generate sequential values for a column in a table.

SQL Server provides two mechanisms for generating values:

the IDENTITY property, for all versions of SQL Server and the sequence object in 2012- 2016.

Each mechanism can be used to provide sequential numbers when rows are inserted into a table. With the sequence object, the number variable can be used in multiple tables.

To use the IDENTITY property, define a column using a numeric data type with a scale of 0 – meaning whole numbers only—and include the IDENTITY keyword.

create TABLE #tmp\_table

(

automaticNumber INT IDENTITY(1,1) (seed (STARTING WITH),increment( INCREMENT BY))

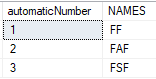
NAMES VARCHAR(20)

)

When an IDENTITY property is defined on a column , INSERT statement against the table do not reference the IDENTIFY COLUMN. SQL Server generates a value using the next available value for the column.

INSERT INTO #tmp\_table

VALUES ('FF'),('FAF'),('FSF');



If a value must be explicitly assigned to an IDENTITY column, the SET IDENTITY INSERT statement must be executed to override the default behaviour of the IDENTITY column.

SET IDENTITY\_INSERT #tmp\_table ON;

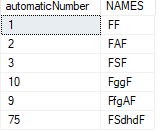
Note: Columns have to be specified in the INSERT statement

INSERT INTO #tmp\_table (automaticNumber,NAMES )

VALUES (10,'FggF'),(9,'FfgAF'),(75,'FSdhdF')

SET IDENTITY\_INSERT #tmp\_table OFF;

SELECT \* FROM #tmp\_table



When a value is assigned to a column by the IDENTITY property, the value may be retrieved like any other value in a column. Values generated by the IDENTITY property are unique within a table. However, without a constraint on the column (such as PRIMARY KEY or UNIUE constraint), uniqueness is not enforced after the value has been generated.

To return the most recently assigned value within the same session scope, such as a stored procedure, use IN SELECT the SCOPE\_IDENTITY() function. The legacy @@IDENTITY function will return the last value generated during the session, but it does not distinguish scope. You can use SCOPE\_IDENTITY() for most purposes.

To reset the IDENTITY property by assigning a new seed, use the DBCC CHECKIDENT statement.

**USING Seuquences**

The IDENTITY property is used to generate a sequence of values for a column within a table, however, the IDENTITY property is not suitable for coordinating values across multiple tables within a database. Database administrators and Developers need to create tables manually to provide a pool of sequential value across tables.

SQL Server 2012 provides the new sequence object, an independent database object that has more flexibility than the IDENTITY property, and can be referenced by multiple tables within a database. The sequence object is created and managed with typically DDL statements such as CREATE, ALTER, and DROP. SQL Server provides a command for retrieving the next value in a sequence.

Execute the following statement to view the properties of an existing sequence.

SELECT \* FROM sys.sequences WHERE name = 'TestSequence' ;

### Creating a sequence that decreases by 1 Creating a sequence that increases by 1

CREATE SEQUENCE Test.CountByNeg1 CREATE SEQUENCE Test.CountByNeg1

START WITH 0 START WITH 0

INCREMENT BY -1 ; INCREMENT BY -1 ;

### Creating a sequence using all arguments

### CREATE SEQUENCE Test.DecSeq

### AS decimal(3,0)

START WITH 125

INCREMENT BY 25

MINVALUE 100

MAXVALUE 200

CYCLE

CACHE 3 ;

Execute the following statement to see the first value and keeps increasing.

SELECT NEXT VALUE FOR Test.DecSeq;

Sequence example :

CREATE TABLE #temp\_table1

(

tblID INT PRIMARY KEY,

tblname varchar(20)

)

CREATE SEQUENCE mysequence

AS INT

START WITH 1

INCREMENT BY 1

insert into #temp\_table1 (tblID, tblname)

values (NEXT VALUE FOR mysequence , 'john')

Module 8

**Using Built-In Functions**

Lesson 1

**Writing Queries with Built-In Functions**

**SQL Server Built-in Function Types**

|  |  |
| --- | --- |
| Functional Categor | Description |
| Scalar | Operate on a single row, return single value |
| Grouped Aggregates | Take one or more input values, return single summarizing value |
| Window | Operate on a window (set) of rows |
| Rowset | Return a virtual table that can be used in a T-SQL statement |

**Scalar Functions**

* Operates on elements from a single row as inputs, return a single value as output
* Return a single scalar value
* Can be used like an expression in queries
* May be deterministic or nondeterministic
* Collation depends on input value or default of database

|  |
| --- |
| Scalar function categories  The following example of the YEAR function shows a typical use of a scalar function in a SELECT clause. The function is calculated once per row, using a column from the row as its input.  Select Function in a Select Clause  SELECT orderid, orderdate , YEAR(ORDERDATE) AS orderyear  FROM sales.orders |
| configuration |
| conversion |
| date and time |
| logical |
| mathematical |
| matadata |
| security |
| system statistical |
| text and image |

The following example uses the system metadata function DB\_NAME() to return the name of the database currently in use by the user`s session;

Metadata Function

Select DB\_NAME() as current\_database



**Aggregate Functions**

* Functions that operate on sets, or rows, of data
* Summarize input rows
* Without GROUP BY clause, all rows are arranged as one group

Grouped aggregate functions operate on sets of rows defined in a GROUP BY clause and return a summarized result. Examples include SUM, MIN, MAX, COUNT, and AVG. In the absence of a GROUP BY clause, all rows are considered one set; aggregation is performed on all of them.

The following example uses a COUNT function and a SUM function to return aggregate values without a GROUP BY clause:

Aggregate Function

SELECT COUNT(\*) AS numorders , SUM(unitprice) AS totalSales

FROM sales.OrderDetails



**Window Functions**

* Functions applied to a window, or set of rows
* Include ranking, offset, aggregates, and distribution functions

Window functions allow you to perform calculations against a user-defined set or window, or rows. They include ranking, offset, aggregates, and distribution functions. Windows are defined using the OVER clause, then window function are applied to the sets defined.

This example uses the RANK function to calculate a ranking based on the unitprice, with the highest price ranked at 1, the next highest ranked 2, and so on;

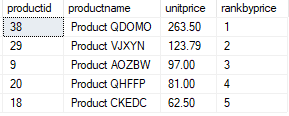
Window Function

SELECT TOP(5) productid, productname , unitprice,

RANK() OVER(ORDER BY unitprice DESC) AS rankbyprice

FROM Production.Products

ORDER BY rankbyprice;



**Rowset Functions**

Rowset functions return a virtual table that can be used elsewhere in the query and take parameters specific to the rowset function itself. They include OPENDATASOURCE, OPENQUERY, OPENROWSET, and OPENXML.

For example, the OPENQUERY function enables you to pass query to a linked server. It takes the system name of the linked server and the query expression as parameters. The results of the query are returned as a rowset, or virtual table, to the query containing the OPENQUERY function.

Lesson 2

**Using Conversion Functions**

**Implicit and Explicit data conversions**

* Implicit conversion occurs automatically and follows data type precedence
* USE explicit conversion:
  + When implicit conversion fail or is not permitted
  + To override data type precedence
* Explicitly convert between types with CAST or CONVERT functions
* Watch for truncation

SQL Server may implicitly convert data types, following the precedence rule for type conversion. However, you might need to override the type precedence, or force a conversion where an implicit conversion might fail.

To accomplish this you can use the CAST and CONVERT functions, in addition to the TRY\_CONVERT function.

Some considerations when converting between data types include:

* Collation – When CAST or CONVERT returns a character string from a character string input, the output uses the same collation. When converting from a noncharacter type to a character, the return value uses the collation of the database. The COLLATE option may be used with CAST or CONVERT to override this behaviour.
* Truncation – When you convert between character or binary types and different data types, data may be truncated, it might appear a cut off, or an error could be thrown because the result is too short to display. The end result depends on the data type involved. For example, conversion from an integer with a two-digit value to a char(1) will return an ‘’\*’’ which means the character type was too small to display the results.

**Converting with Cast**

Converts a value from one data type to another; CAST is an ANSI-Standard function and is therefore recommended over the SQL Server-specific CONVERT function. As CAST is a scalar function, you may use it in SELECT and WHERE clauses.

Converting with cast Example

CAST(<value> AS <datatype> ) select getdate() as [getdate],CAST(getdate() AS date) as[date];



**Converting with CONVERT**

Unlike the ANSI-standard CAST function, the CONVERT function is proprietary to SQL Server and is therefore not recommended. However, because of its additional capability to format to return value, you may occasionally still need to us it.

As with CAST, CONVERT is a scalar function, you may use CONVERT in SELECT and WHERE clauses.

CONVERT syntax

CONVERT(<dataType>,<[value]>,<optional\_style\_number>);

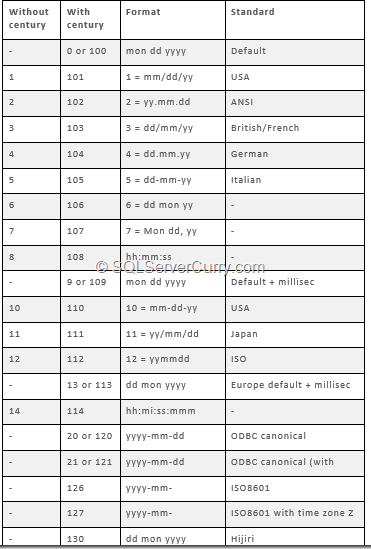
The style number argument causes CONVERT to format the return data according to a specific set of options. These cover a wide range of date and time styles, in addition to styles for numeric, XML and binary data. Some date and time examples include:

Convert example ;

SELECT CONVERT(CHAR(8),CURRENT\_TIMESTAMP,12),

CONVERT(CHAR(11),CURRENT\_TIMESTAMP,13);





**Converting Strings with PARSE**

Converts strings to date, time, and number types.

Parse requires a string which must be in a form recognizable to SQL Server and returns a value of a specific data type.

Parse Syntax

SELECT PARSE('<STRING VALUE>',<datatype>[USING <culture\_code>]);

The culture parameter must be in the form of a valid .NET Framework culture code, such as **‘en-US’** for US ENGLISH, **‘es-ES’** for Spanish, and so on. If the culture parameter is commited, the settings for the current user session will be used.

PARSE example

SELECT PARSE('02/12/2012' AS datetime2 USING 'en-US')



**Converting with TRY\_PARSE and TRY\_CONVERT**

When using CONVERT or PARSE an error may occur of the input value cannot be converted to the specified output type.

For example, if February 31, 2012(an invalid date) is passed to convert, a runtime error is raised.

SELECT CONVERT(datetime2,'20120231')



SQL Server 2016 provides conversion functions to address this. TRY\_PARSE and TRY\_CONVERT will attempt a conversion, just like PARSE and CONVERT, respectively. However, instead of raising a runtime error, failed conversion return NULL.

SELECT TRY\_PARSE('20120231' AS datetime2 )



Lesson 3

**Using Logical Functions**

**Writing Logical Test with Functions**

ISNUMERIC – tests whether an input expression is a valid numeric data type.

A useful function for validating data type of an expression is ISNUMERIC. This tests an input expression and returns a 1 if the expression is convertible to any numeric type, including integers, decimals, money, floating point and real. If the value is not convertible to a numeric type ISNUMERIC returns 0.

Writing Logical Tests with Functions

Select empid, lastname, postalcode

FROM HR.Employees

WHERE ISNUMERIC(postalcode) = 1;



**Performing Conditional Tests with IIF**

IIF accepts three parameters – a logical test to perform,

A value to return if the test evaluates to True

And a value to return if the test evaluates to False or Unknown

**IIF Syntax**

IIF ( boolean\_expression, true\_value, false\_value )

You can think of IIF as a shorthand approach to writing CASE statement with two possible return values, As with CASE you may nest an IIF function within another IIF function, down to a maximum level of 10.

**IIF Example**

select productid , unitprice,

IIF ( unitprice > 50, 'high', 'low' ) as pricepoint

from Production.Products;



**Selecting items from a List with CHOOSE**

CHOOSE returns the value of an item at a specific index in a list.

CHOOSE returns an item from a list, selecting the item that matches an index value.

Note the selected Item has to be a string data type.

CHOOSE Example

SELECT CHOOSE(2 , unitprice , productname)

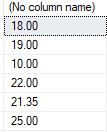
FROM Production.Products

Msg 235, Level 16, State 0, Line 30

Cannot convert a char value to money. The char value has incorrect syntax.

SELECT CHOOSE(1 , cast(unitprice as varchar(20)) , productname)

FROM Production.Products



Lesson 4

**Using Functions to Work with NULL**

**Converting NULL with ISNULL**

ISNULL replaces NULL with a specified value

Not standard use COALESCE instead

Example :

SELECT custid , city , ISNULL(region, 'N/A') AS Region

FROM Sales.Customers;



**USING COALESCE to Return Non-NULL Values**

COALESCE takes as its input one or more expressions, and returns the first non-NULL argument it finds.

With only two arguments, COALESCE behaves like ISNULL. However with more than two arguments, COALESCE can be used as an alternative to a multipart CASE expression using ISNULL.

If all arguments are NULL , COALESCE returns NULL.

COALESCE Example :

The following example returns customers with region where available, and adds a new column combining country,region and city, replacing NULL regions with a space:

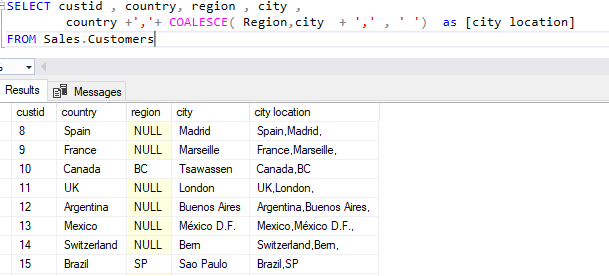
SELECT custid , country, region , city ,

country +','+ COALESCE( Region + ',' , ' ') + city as [city location]

FROM Sales.Customers;



COALESCE Example with multiple expressions

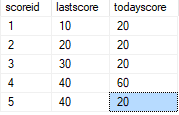


**Using NULLIF to Return NULL if Value Matches**

NULLIF compares two expressions

Returns NULL if both expressions are equal

Returns the first argument if the two arguments are not equal

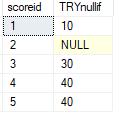


Example : having

When both expressions are equal it returns NULL, when not equal it returns the first expression

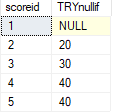
select scoreid , nullif(lastscore , todayscore) as TRYnullif

from #tmp\_tbl2



select scoreid , nullif(lastscore , '10' ) as TRYnullif

from #tmp\_tbl2



Module 9

**Grouping and Aggregating Data**

Lesson 1

**Using Aggregate Functions**

**Working with Aggregate Functions**

When working with aggregate functions, you need to consider the following:

Aggregate functions return a single (scalar) value and can be used in SELECT statements where a single expression is used, such as SELECT , HAVING , and ORDER BY clauses.

Aggregate functions ignore NULLs, except when using COUNT(\*).

Aggregate functions in a SELECT list do not generate column alias. You may wish to use the AS clause to provide one.

Aggregate functions in a SELECT clause operate on all rows passed to the SELECT phase. If there is no GROUP BY clause, all rows will be summaraized.

To extend beyond the built-in functions, SQL Server provides a mechanism for user-defined aggregate functions via the .NET Common Language Runtime(CLR)

**Built-in Aggregate Functions**

|  |  |  |
| --- | --- | --- |
| Commonly | Statistical | Other |
| SUM | STDEV | CHECKSUM\_AGG |
| MAX | STDEVP | GROUPING |
| MIN | VAR | GROUPING\_ID |
| AVG | VARP |  |
| COUNT |  |  |
| COUNT\_BIG |  |  |

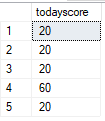
SQL Server provides many built-in aggregate functions.

Commonly used functions include :

|  |  |  |
| --- | --- | --- |
| Comonly | Syntax | Description |
| SUM | SUM(<expression>) | Totals all the non-NULL numeric values in column |
| MAX | AVG(<expression>) | averages all the non-NULL numeric values in column (SUM/COUNT) |
| MIN | MIN(<expression>) | Returns the smallest number, earliest date/time, or first-occurring string (according to collation sort rules) |
| AVG | MAX(<expression>) | Returns the largest number, latest date/time, or last-occurring string (according to collation sort rules) |
| COUNT OR COUNT\_BIG | COUNT(\*) or COUNT(<EXPRESSION>) | With (\*), counts all rows including those with NULL values, When a column is specified as <expression>, returns count of non-NULL rows for that column. COUNT returns an intl COUNT\_BIG returns a BIG\_INT |

**Aggregate Example**

select todayscore from #tmp\_tbl2;

****

select max(todayscore) as maxx , min(todayscore) as minn ,

avg(todayscore) as avgg, COUNT(todayscore) as count

from #tmp\_tbl2;

Note the above example does not have a GROUP BY clause.

Therefore, all rows will be summarised by the aggregate formula in the SELECT clause.

****

When using aggregates in a SELECT clause, all columns referenced in the SELECT list must be used as inputs for an aggregate function, or be referenced in a GROUP BY clause.

select todayscore, max(todayscore) as maxx , min(todayscore) as minn ,

avg(todayscore) as avgg, COUNT(todayscore) as countt

from #tmp\_tbl2

Msg 8120, Level 16, State 1, Line 36

Column '#tmp\_tbl2.todayscore' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

select todayscore, max(todayscore) as maxx , min(todayscore) as minn ,

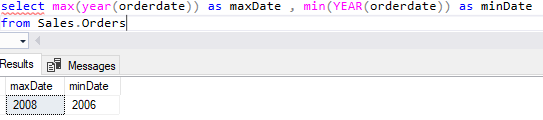
avg(todayscore) as avgg, COUNT(todayscore) as countt

from #tmp\_tbl2

GROUP BY todayscore



Aggregate with Functions :



**Using DISTINCT with Aggregate Functions**

Use DISTINCT with aggregate functions to summarize only unique values

Distinct aggregates eliminates duplicate values, not rows (Unlike SELECT DISTINCT)

Summarizing Distinct Values:

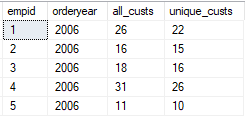
SELECT empid , YEAR(orderdate) AS orderyear ,

COUNT(custid) as all\_custs ,

COUNT (DISTINCT custid) as unique\_custs

FROM Sales.Orders

GROUP BY empid , YEAR(orderdate);



Note the difference in each row between the COUNT of custid and the DISTINCT COUNT.

All\_cust simply returns all rows except those containing a NULL (as ONLY COUNT(\*) counts NULLs)

Unique\_custs excludes duplicate custid (repeat customer) and returns a count of unique customers,

Answering the question: How many customers per employee?

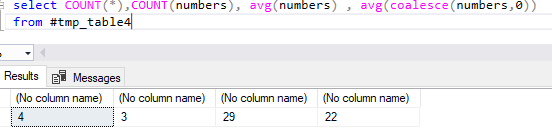
**Using Aggregate Functions with Null**

With the exception of COUNT used with the (\*) option,

T-SQL aggregate functions ignore NULLs. This means, for example, that a SUM function will add only non-NULL values. NULLs do not evaluate to zero.

The presence of NULLs in a column may lead to inaccurate computations for AVG, which will sum only populated rows and divide that sum by the number of non-NULL rows. There may be a difference in the result between AVG(<column>) and (SUM(<column>)/COUNT(\*)).

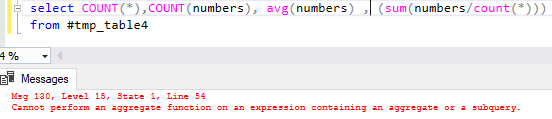
Example :



AVG(coalesce... replaces nulls with 0

select \* from #tmp\_table4



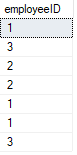
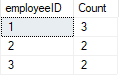


Lesson 2

**Using the GROUP BY Clause**

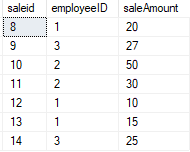
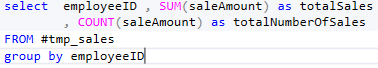
**Using the GROUP BY Clause**

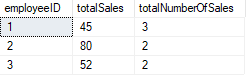
When your SELECT statement is processed, after the FROM clause and WHERE clause (If present) have been evaluated, a virtual table is created. The contest of the virtual table is now available for further processing. You can use the GROUP BY clause to subdivide the result of the preceding query phases into groups of rows.

Group By creates groups and places rows into each group as determined by unique combinations of the elements specified in the clause.

Once the GROUP BY clause has been processed and rows have been associated with a group, subsequent phase of the query must aggregate any elements of the source row that do not appear in the GROUP BY list, This will have an impact on how you write your SELECT and HAVING clauses.



**GROUP BY and the Logical Order of Operation**

|  |  |  |
| --- | --- | --- |
| Logical Order | Phase | comments |
| 5 | SELECT |  |
| 1 | FROM |  |
| 2 | WHERE | Operates on rows |
| 3 | GROUP BY | Create groups |
| 4 | HAVING | operates on groups |
| 6 | ORDER BY |  |

If a query uses a GROUP BY, all subsequent phases operate on the groups, not source rows

HAVING, SELECT & ORDER BY must return a single value per group

All columns in SELECT, HAVING, & ORDER BY, must appear in the GROUP BY clause or be inputs to aggregate expressions.

The following query will return an error because orderdate is not an input to GROUP BY, and its data has been ‘lost’ following the FROM clause:

SELECT empid , orderdate , COUNT(\*) AS CNT

FROM Sales.Orders

GROUP BY empid

Msg 8120, Level 16, State 1, Line 68

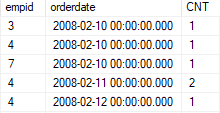
Column 'Sales.Orders.orderdate' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

As orderdate is now an input to GROUP BY the query will work

SELECT empid , orderdate , COUNT(\*) AS CNT

FROM Sales.Orders

GROUP BY empid , orderdate



**GROUP BY Workflow**

Initially, the WHERE clause is processed followed by the GROUP BY.

The following pictures will show the result of the where clause, followed by being performed on the results.

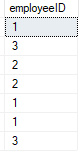
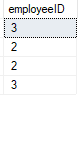
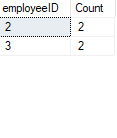
select employeeID , COUNT(employeeID) AS [Count]

FROM #tmp\_sales

where employeeID != 1

group by employeeID

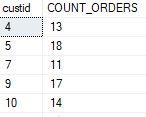
Lesson 3

**Filtering Groups with HAVING**

**Filtering Grouped Data Using the HAVING Clause**

|  |  |  |
| --- | --- | --- |
| Logical Order | Phase | comments |
| 5 | SELECT |  |
| 1 | FROM |  |
| 2 | WHERE | Operates on rows |
| 3 | GROUP BY | Create groups |
| 4 | HAVING | operates on groups |
| 6 | ORDER BY |  |

A HAVING clause enables you to create a search condition, conceptually similar to the predicate of a WHERE clause, which then tests each group returned by the GROUP BY clause.



Example: Having Count(\*) bigger then 10

SELECT custid , COUNT(\*) AS COUNT\_ORDERS

FROM Sales.Orders

GROUP BY custid

HAVING COUNT(\*) >=10

**Compare HAVING to WHERE**

While both clauses filter data, it is important to remember that WHERE operates on rows returned by the FROM clause. If a GROUP BY....HAVING section exists in your query following a WHERE clause, the WHERE clause will filter rows before GROUP BY is processed- potentially limiting the groups that can be created.

A HAVING clause is processed after GROUP BY and only operates on groups, no detail rows.

Summerising:

* A WHERE clause controls which rows are available to the next phase of the query
* A HAVING clause controls which groups are available to the next phase of the query

Note: WHERE and HAVING clauses are not mutually exclusive.

Module 10

**Using Subqueries**

Lesson 1

**Write Self-Contained Subqueries**

**Working with Subqueries**

A subquery is a SELECT statement nested, or embedded, within another query. The nested query, which is the subquery, is the inner query. The query containing the nested query is the outer query.

The purpose of a subquery is to return results to the outer query. The form of the result will determine whether the subquery is a scalar or multi-valued subquery:

* Scalar subueries, like scalar functions, return a single value. Outer queries need to be written to process a single result.
* Multi-valued subqueries return a result much like a single column table. Outer queries need to be written to handle multiple possible results.

In addition to the choice between scalar and multi-valued subqueries, you may choose to write self-contained subqueries or others that are correlated with the outer query.

* Self-contained subqueries can be written as stand-alone queries, with no dependencies on the outer query. A self-contained subquery is processed once, when the outer query runs and passes its result to that outer query.
* Correlated subqueries reference one or more columns from the outer query and therefore depend on it. Correlated Subqueries Cannot be run seperatley from the outer query.

**Writing Scalar Subqueries**

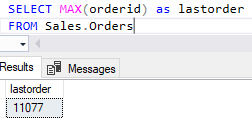
A scalar subquery is an inner SELECT statement within an outer query, written to return a single value. Scalar subquries may be used anywhere in an outer T-SQL statement where a single-valued expression is permitted- Such as in a SELECT, WHERE, HAVING, or even FROM clauses.

To write a scalar subquey, consider the following guidelines:

* To denote a query as a subquery, enclose it in parentheses ( ).
* Multiple levels of subquery are supported in SQL Server, but up to 32 levels are supported.
* If the subquery returns an empty set, the result of the subquery is converted and returned as a NULL. Ensure your outer query can handle a NULL, in addition to other expected results.

**Example:**

**INNER QUERY**  (notice it returned only a single value. That is a scalar subquery)



**OUTER AND INNER QUERY**

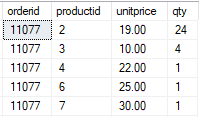
SELECT orderid , productid , unitprice , qty

FROM Sales.OrderDetails

WHERE orderid = ( SELECT MAX(orderid) as lastorder

FROM Sales.Orders

);



Because the outer query used an = operator in the predicate of the WHERE clause, and the subquery returned a single value, the query ran correctly. If an outer query is written to expect a single value, such by using simple equality operators (=, <, > and <>, for example), and the inner query returns more than one result, an error will be returned.

**Writing Multi-Valued Subqueries**

A multi-valued subquery may return more than one result, in the form of a single-column set.

It is suitable to return results to IN predicate, as in the following example

SELECT custid , orderid

FROM Sales.Orders

WHERE custid IN(SELECT custid

FROM Sales.Customers

WHERE country = N'Mexico' );

In this example, if you were to execute only the inner query, you would return the following list of custids for customers in the country of Mexico.

****

SELECT custid

FROM Sales.Customers

WHERE country = N'Mexico'

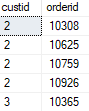
SQL Server will pass those results to the outer query, logically rewritten as follows:

SELECT custid , orderid

FROM Sales.Orders

WHERE custid IN (2,3,13,58,80);

The outer query will continue to process the SELECT statement with the following partial results:



You may find scenarios in which multi-valued subqueries are written as SELECT statements using JOINs

For example the previous subquery might be written as follows, with the same result and comparable performance.

Subquery Rewritten AS JOIN

SELECT a.custid , a.orderid

FROM Sales.Orders AS a

INNER JOIN Sales.Customers as b

ON a.custid = b.custid

WHERE country = N'MEXICO'

Note: in some cases, the database engine will interpret a subquery as a join and execute it accordingly. As you learn more about SQL Server internals, such as execution plan, you may be able to see queries interpreted this way.

Lesson 2

**WRITING Correlated Subqueries**

**Working with Correlated Subqueries**

Like Self-contained subqueries, correlated subqueries are SELECT statements nested within an outer query. They may also be written as scalar or multi-valued subqueries. They are typically used to pass a value from the outer subquery to the inner subquery, to be used as a parameter there.However, unlike self-contained subqueries, correlated subqueries depend on the outer query to pass values into the subquery as a parameter. This leads to some special considerations when planning their use:

* Correlated subqueries cannot be executed separately from the outer query. This complicates testing and debugging
* Unlike self-contained subqueries which are processed once, correlated subqueries will run multiple times. Logically, the outer query runs first, and for each row returned, the inner query is processed.

The following example uses a correlated subquery to return the orders with the last order date for each employee.

The subquery accepts an input value from the outer query, uses the input in its where clause, and returns a scalar result to the outer query:

SELECT orderid , empid , orderdate

FROM Sales.Orders as o1

WHERE orderdate = ( SELECT MAX(orderdate)

FROM Sales.orders AS O2

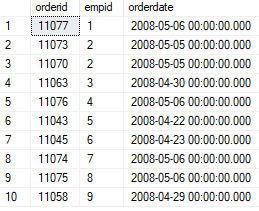
WHERE O2.empid = o1.empid)

order by empid,orderdate ;

Line numbers have been added for subsequent explanation. They don`t indicate the order in which the steps are logically processed.

|  |  |  |
| --- | --- | --- |
| Line No. | Statement | Description |
| 1 | SELECT orderid , empid , orderdate | Column returned by the outer query |
| 2 | FROM Sales.Orders as o1 | Source table for the outer query. Note the alias |
| 3 | WHERE orderdate = | Predicate used to evaluate the outer rows against the result of the inner query |
| 4 | ( SELECT MAX(orderdate) | Column returned by the inner query. Aggregate function returns a scalar value. |
| 5 | FROM Sales.orders AS O2 | Source table for the inner query. Note the alias |
| 6 | WHERE O2.empid = o1.empid ) | Correlated of empid from the inner query. This value will be supplied for each row in the outer query |
| 7 | order by empid,orderdate ; | Sorts the result of the outer query |

The query returns the following results. Note that some employees appear more than once, because they are associated with multiple orders on the latest orderdate.



**Writing Correlated Subqueries**

To write correlated Subqueries, Consider the following guidlines

* Write the outer query to accept the appropriate return result from the inner query. If the inner query will be scalar, you can use equality and comparison operators, such as =, >, <. < >, in the WHERE clause. If the inner query might return multiple values, use an IN predicate. Plan and handle NULL results.
* Identify the column from the outer query that will be passed to the correlated subquery. Declare an alias for the table that is the source of the column in the outer query.
* Identify the column from the inner table that will be compared to the column from the outer table. Create an alias for the source table, as you did for the outer query.
* Write the inner query to retrieve values from the source, based on the input value from the outer query. For example, use the outer column in the WHERE clause of the inner query.

The correlation between the inner and outer queries occurs when the outer value is passed to the inner query for comparison. It`s this correlation that gives the subquery its name.

Lesson 3

**Using the EXISTS Predicate with Subqueries**

**Working with EXISTS**

* When a subquery is used with the keyword EXISTS, it functions as an existence test

True or False – no rows passed back to outer query

* EXISTS evaluates to TRUE or FALSE (not unknown)
  + If any rows are returned by the subquery, EXISTS returns TRUE
  + If no rows are returned EXISTS returns FALSE

**Syntax**

WHERE [NOT] EXISTS (SUBQUERY)

The following queries will return the same result on with using COUNT and the other using EXISTS

Using COUNT in a Subquery

select empid , lastname

FROM HR.Employees as e

WHERE (SELECT COUNT(\*)

FROM HR.Employees as f

where f.empid = e.empid

)>0;

Using EXISTS in a subquery

select empid , lastname

FROM HR.Employees as e

WHERE exists (select \* from HR.Employees as f

WHERE f.empid = e.empid);

In the first example, the subquery must count every occurrence of each empid found in the SALES.Orders table and compare the count result to zero, simply to indicate that the employee has associated rders.

In the second query, EXISTS returns TRUE for an empid as soon as one has been found in the Sales.Orders table-a complete accounting of each occurrence is unnecessary.

Note: From the perspective of logical processing, the two query forms are equivalent. From a performance perspective, the database engine may treat the queries differently as it optimizes them for execution.

Another useful application of EXIST is negating it with NOT, as in the following query example, which will return any customer who has never placed an order.

NOT EXISTS EXAMPLE

SELECT custid , companyname

FROM Sales.Customers AS C

WHERE NOT EXISTS (SELECT \*

FROM Sales.Orders AS O

WHERE O.custid = C.custid);

SQL Server will not have to return data about the related orders for customers who have placed orders. If a customer ID is found in the sales.Orders table, NOT EXISTS evaluates to FALSE and the evaluation quickly completes.

When writing queries that use EXISTS with subqueries, consider the following:

* The keyword EXISTS directly follows WHERE. No column name(or other expression) needs to precede it unless NOT is also used.
* Within the subquery following the ESISTS, the SELECT list only needs to contain (\*). No rows are returned by the subquery, so no columns need to be specified.

Module 11

**Using Table Expressions**

Lesson 1

**Using VIEWS**

**Writing Queries That Return Results in Views**

A view is a named table expression whose definition is stored as metadata in a SQL Server database. Views can be used as a source for queries in much the same way as tables themselves. However, views can be used as a source for queries in much the same way as tables themselves. However do not persistently store data; the definition of view is unpacked at runtime and the source objects are queried.

Note in an indexed view, data is materialized in the view.

To write a query that uses a view as its data source, use the two-part view name wherever the table source would be used, such as in From or Join clause;

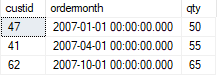
SELECT <select\_list>

FROM <view\_name>

ORDER BY <sort\_list>;

Note that ORDER BY clause is used in this simple syntax to emphasize the point that as a table expression, ther is no order included in the definition of a view.

The following example uses a simple view whose definition is stored in the TSQL database. Note that there is no way to determine that the FROM clause references a view and not a table.

 SELECT custid , ordermonth , qty 

FROM sales.custorders;

The apparent similarity between a table and a view provides as important benefit-an application can be written to use views instead of the underlying tables, shielding the application from changes to the table. Providing the view continues to present the same structure to the calling application, the application will receive consistent results. Views can be considered as an application program interface (API) to a database for purposes of retrieving data.

Administrators can also use views as a security layer, granting users permissions to select from a view without providing permission to underlying source tables.

**Creating simple views**

To use a view in your query, it must be created by a database developer or administrator with appropriate permission in the database.

To store a view definition, use the CREATE VIEW T-SQL statement to name and store a single SELECT Statement, Note that the ORDER BY clause is not permitted a view definition unless the view uses a TOP, OFFSET-FETCH, or FOR XML element.

CREATE VIEW syntax

CREATE VIEW [,schema\_name.view\_name>] [colum\_alias\_list]

[WITH <VIEW\_options> ]

AS select\_statement;

The following example creates the view names Sales.CustOrders that exists in the TSQL sample database. Most of the code within the example makes up the definition of the select statement itself:

SELECT

o.custid,DATEADD(month ,DATEDIFF(month , 0 , o.orderdate) , 0) AS month , O.orderdate

FROM Sales.Orders AS o

JOIN Sales.OrderDetails AS od

ON o.orderid = od.orderid

ORDER BY month

You can query system metadata by querying system catalogue views such as sys.views.

To query a view, refer it in the FROM clause of a SELECT statement, as you would refer to a table.

Lesson 2

**Using Inline TVFs**

**Writing Queries That Use Inline TVFs**

Inline TVFs are named table expressions whose definitions are stored persistently in a database that can be queried in much the same way as a view. This enables reuse and centralized management of code in a way that is not possible for derived tables and CTEs as query-scoped table

Note: SQL Server supports several types of user-defined functions. In addition to inline TVFs, users can create scalar functions, multi-statement TVFs, and functions written in the .NET Common Language Runtime (CLR).

One of the key distinctions between views and inline TVFs is that the latter can accept input parameters. Therefore you may think of inline TVFs conceptually as parameterized views and choose to use them in place of views when flexibility of input is preferred.

* TVFs are named table expressions with definitions stored in a database
* TVFs return virtual table to the calling query
* SQL Server provides two types of TVFs:
  + Inline, based on a single SELECT statement
  + Multi-statement, which creates and loads a table variable
* Unlike views TVFs support input parameters
* Inline TVFs may be thought of as parameterized views

Create Simple Inline TVFs

To use inline TVFs in your queries, it must be created by a database developer or administrator with appropriate permission in the database.

To store an inline TVF view definition:

* USE the CREATE FUNCTION T-SQL statement to name and store a single SELECT statement with optional parameters.
* Use RETURNS TABLE to identify this function as a TVF.
* Enclose the SELECT statement inside parentheses following the RETURN keyword to make this an inline function.

**CREATE FUNCTION syntax for inline Table-Valued Functions**

CREATE FUNCTION <Inline\_Function\_Name, sysname, FunctionName>

(

-- Add the parameters for the function here

<@param1, sysname, @p1> <Data\_Type\_For\_Param1, , int>,

<@param2, sysname, @p2> <Data\_Type\_For\_Param2, , char>

)

RETURNS TABLE

AS

RETURN

(

-- Add the SELECT statement with parameter references here

SELECT 0

)

**Inline Table-valued Function Example**

CREATE FUNCTION production.TopNProducts

( @t AS INT

)

RETURNS TABLE

AS

RETURN

(

SELECT TOP(@t) productid , productname , unitprice

FROM Production.Products

ORDER BY unitprice DESC

)

GO

**Retrieving from inline TVFs**

* Select FROM function
* Use two-part name
* Pass in parameters

Example :

select \* from [Production].[TopNProducts](50)

WHERE unitprice < 100

To delete the function : drop function <function name>

Lesson 3

**Using Derived Tables**

**Writing Queries with Derived Tables**

Like subqueries, you create derived tables in the FROM clause of an outer SELECT statement. Unlike subqueries, you write derived tables using a name expression that is logically equivalent to a table and may be referenced as a table elsewhere in the outer query. Derived tables allow you to write T-SQL statements that are more modular, helping you break down complex queries into more manageable parts. Using derived tables in your queries can also provide workarounds for some of the restrictions imposed by the logical order of query processing, such as the use of column aliases.

To create a derived table, write the inner query between parentheses, followed by an AS clause and a name for the derived table:

Derived Table syntax

SELECT <[OUTER QUERY LIST]>

FROM (SELECT <[INNER QUERY COLUMN LIST]>

FROM <[table source]> AS [derived table alias]

);

The following example uses a derived table to retrieve information about orders placed per year by distinct customers. The inner query builds a set of orders and places them into the derived table`s derived year. The outer query operates on the derived table and summarizes the results

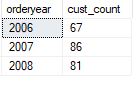
Derived Table Example :

Select orderyear , COUNT(distinct custid) as cust\_count

FROM(SELECT YEAR(orderdate) as orderyear , custid

FROM Sales.Orders) AS derived\_year

GROUP BY orderyear



When writing Queries that uses derived tables, consider the following:

* Derived tables are not stored in the database. Therefore, no special security privileges are requested to write queries using derived tables, other than the rights to select from the source objects.
* A derived table is created at the time of execution of the outer query and goes out of scope when the outer query ends
* Derived tables do not necessary have impact on performance, compared to the same query expressed differently. When the query is processed the statement is unpacked and evaluated against the underlying database objects.

**Guidelines for Derived Tables**

* The nested SELECT statement that defines the derived table must have an alias assigned to it. The outer query will use the alias in the SELECT statement in much the same way you refer to aliases tables joined in a FROM clause.
* All columns referenced in the derived table`s SELECT clause should be assigned aliases, a best practice this is not always required in T-SQL. Each alias must be unique within the expression. The column aliases may be declared inline with the column or externally to the clause. You will see examples of this in the next topic.
* The SELECT statement that defines the derived table expression may not use an ORDER BY clause unless it also includes a TOP operator, an OFFSET/FETCH clause or a FOR XML clause. As a result there is no order provided by the derived table. You sort the results in the outer query.
* The SELECT statement that defines the derived table may be written to accept arguments in the form of local variables. If the SELECT statement is embedded in a stored procedure, the argument may be written as parameters for the procedure. You will see examples of this later in the module.
* Derived tables expressions that are nested within an outer query can contain other derived tables expressions. Nesting is permitted, but it is not recommended due to increase complexity and reduced readability.
* A derived table may not be referred to multiple times within an outer query. If you need to manipulate the same result, you will need to define the derived table expression every time, such as on each side of a JOIN operator.

**Using Aliases for Column Names in Derived Tables**

To create aliases, you can use one of two methods-inline or external:

* **Column aliases defined inline**

Select orderyear , COUNT(distinct custid) as cust\_count

FROM (

SELECT YEAR(orderdate) as orderyear , custid

FROM Sales.Orders

) AS derived\_year

GROUP BY orderyear

* **Column aliases defined externally**

Select orderyear , COUNT(distinct custid) as cust\_count

FROM (

SELECT YEAR(orderdate), custid

FROM Sales.Orders

) AS derived\_year (orderyear , custid)

GROUP BY orderyear

Note: When using external aliases, if the inner query is executed separately, the aliases will not be returned to the outer query. For ease of testing and readability, it is recommended that you use inline rather than external aliases.

**Nesting and Reusing Derived Tables**

Since a derived is itself a complete query expression, that query can refer to a derived table expression. This creates a nesting scenario, which while possible, is not recommended for reasons of code maintenance and readability.

Nested Derived Tables

Select orderyear , cust\_count

FROM

(SELECT orderyear , COUNT(distinct custid) AS cust\_count

FROM

(SELECT year(orderdate) as orderyear , custid

FROM Sales.Orders

) AS derived\_year\_3

GROUP BY orderyear

)as derived\_count\_2

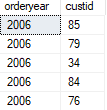
WHERE cust\_count >80

* Logically, the innermost wuery is processed first, returning these partial results as derived\_year\_1

(SELECT year(orderdate) as orderyear , custid

FROM Sales.Orders

) AS derived\_year\_3



* Next, the middle query runs, grouping and aggregateing the resuls into derived\_count\_2

(SELECT orderyear , COUNT(distinct custid) AS cust\_count

FROM

(SELECT year(orderdate) as orderyear , custid

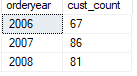
from Sales.Orders) AS derived\_year\_3

GROUP BY orderyear

)as derived\_count\_2

As you can see, while nesting derived tables adds complexity.

While nesting derived tables is possible, references to the same derived table from multiple clauses of an outer query can be challenging. Since the table is defined in the FROM clause, subsequent phases of the query can see it, but it cannot be referenced elsewhere in the same FROM clause.



* Finally the outer query runs filtering the output



Example, a derived table defined in a FROM clause may be referenced in a WHERE clause, but not in a JOIN in the same FROM clause that defines it. The derived table must be defined separately, and multiple copies of the code maintained. For an alternative approach that allows reuse without maintaining separate coppies of the derived table definition.

Lesson 4

**Using CTEs**

**Writing Queries with CTEs**

* CTEs are named table expressions defined in a query. Like subqeries and derived tables, CTEs provide a mean to break down query problem into smaller, more modular units.
* When writing queries CTEs , consider the following guidelines:
* Like derived tables, CTEs are limited in scope to the execution of the outer query. When the outer query ends, so does the CTE`S life span.
* CTEs require a name for the table expression, In addition to unique names for each of the columns referenced in the CTEs SELECT clause.
* CTEs may use inline or external aliases for columns.
* Unlike derived tables, a CTE may be referenced multiple times in the same query with one definition. Multiple CTEs may also be defined in the same WITH clause.
* CTEs support recursion, in which the expression is defined with a reference to itself.

**Creating Queries with Common Table Expressions**

**CTE Syntax**

WITH <CTE\_NAME>

AS ( <CTE\_DEFINITION>)

**CTE Example**

WITH CTE\_Year

AS

(

SELECT Year(orderdate) as orderyear , custid

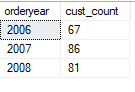
FROM sales.Orders

)

SELECT orderyear , COUNT(distinct custid) as cust\_count

FROM CTE\_Year

GROUP BY orderyear



Module 12

**Using set Operators**

Lesson 1

**Writing Queries with the UNION Operator**

**Interaction Between Sets**

SQL Server provides several operators that act on sets, each of which has a different effect on the input sets. The set operators have a number of common features that you need to understand before starting using them:

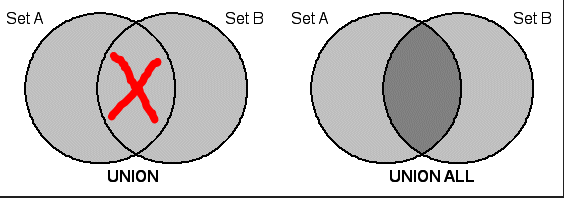
* Each input set is the result of a query, which may include any SELECT statement components you have already learned about, except an ORDER BY clause.
* The input sets must have the same number of columns and the columns must have compatible data types. The column data types, if not initial compatible, must be made compatible through conversion-this may be implicit if the data type support it, otherwise an explicit conversion must be required (using convert or cast).
* A NULL in one set is treated as equal to a NULL in another, despite what you have learned comparing NULLs earlier in this course.
* Each operator can be thought as having two forms: DISTINCT and ALL.

Example, UNION DISTINCT eliminates duplicate rows while combining two sets, UNION ALL combines all rows including duplicates. Not all set operators support both forms in SQL Server 2016

* Note: When working with set operators, it is useful to remember that, in set theory, a set does not provide a sort of order and includes only distinct rows. If you need the result sorted, you should add an ORDER BY to the final result, as you may not use it inside the input queries.

**Using the UNION Operator**

By using the UNION operator, you can combine rows from one input set with rows from another into a resulting set. If a row appears in either of the input sets, it will be returned in the output. Duplicate rows are eliminated by the UNION operator



**Using UNION all Operator**

It returns a result set with all rows from both input sets.

To avoid the performance penalty use UNION ALL over UNION when you are certain that there are no duplicates or when duplicate exists and are required

**Example( using the same column):**

select custid from Sales.Orders select custid from Sales.Orders select custid from Sales.Orders union union ALL

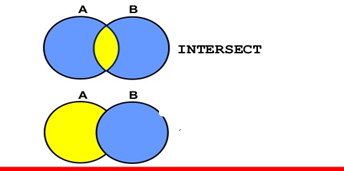
 select custid from Sales.Orders select custid from Sales.Orders  

Rows (has a lot of duplicates) Distinct Rows (FILTERS DUPLICATES) ALL ROWS TOGHETHER 830 + 830

Lesson 1

**Using EXCEPT and INTERSECT**

While UNION and UNION ALL combine all rows from input sets, you might need to return either only those rows in one set but not the other-or only rows in both sets.

****

EXCEPT

**Using the INTERSECT Operator**

Returns rows that appear in both input sets (it will return one row of each not duplicate as if in distinct)

select country,city

from Production.Suppliers

INTERSECT

select country,city

from Sales.Customers

** returned 5 rows**

**Using the Except Operator**

EXCEPT returns only distinct rows that appear in the left side but not in the right

select country,city

from Sales.Customers

except

select country,city 

from Production.Suppliers

select country,city

from Production.Suppliers

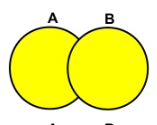
Except

select country,city 

from Sales.Customers

**Explaining UNION, UNION ALL , EXCEPT and INTERSECT**

UNION



1,2,3,4,5

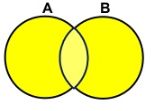
1,2,6,7,8

UNION

**=**

1,2,3,4,5,6,7,8

UNION ALL



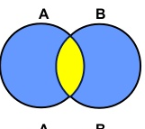
1,1,2,2,3,4,5,6,7,8

**=**

UNION

1,2,6,7,8

1,2,3,4,5



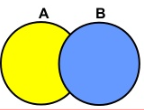
1,2,3,4,5

1,2,6,7,8

INTERSECT

**=**

1,2



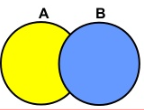
1,2,3,4,5

1,2,6,7,8

EXCEPT

**=**

3,4,5



1,2,6,7,8

1, 2,3,4,5

**=**

6,7,8

EXCEPT

Notice:

The total rows returned as result by the intersect and both except together, will sum up to the amount of the union result.

Lesson 3

**USING APPLY**

An alternative to combining rows from the two sets, SQL Server provides a mechanism to apply a table expression from one set on each row in the other set. In this lesson, you will learn how to use the APPLY operator to process rows in one set using rows in another.

**Using APPLY operator**

The APPLY operator enables queries that evaluates rows in one input set against the other expression that defines the second input set. Apply is a table operator, not a set operator, you will use APPLY in a FROM clause, like JOIN, rather than as a set operator that operates on two compatible result set of queries.

The APPLY operator is similar to a correlated subquery in that it applies a correlated table expression to each row from a table. However, APPLY differs from correlated subqueries by returning a table-valued result rather than a scalar or multi-valued result. For example, the table expression could be a TVF; you can pass elements from the left rows as input parameters to the TVF.

NOTE: When describing input tables used with APPLY, the terms ‘left’ and ‘right’ are used in the same way as they are with the JOIN operator, based on the order in which they appear, relative to one another in the FROM clause.

**The CROSS APPLY operator**

CROSS APPLY applies the right table source to each row in the left table source

Only rows with results in both the left and right table source are returned.

Most INNER JOIN statements can be rewritten as CROSS APPLY statements

CROSS APPLY ; INNER JOIN EXAMPLE

SELECT O.orderid,orderdate

FROM Sales.Orders AS O

INNER JOIN Sales.OrderDetails AS OD

ON O.orderid = OD.orderid;



SELECT O.orderid, O.orderdate, od.unitprice , od.productid

FROM Sales.Orders AS O

CROSS APPLY ( SELECT OD.productid , od.unitprice

FROM Sales.OrderDetails as od

where o.orderid = od.orderid

) AS od



BOTH RETURNED THE SAME RESULT

**The OUTER APPLY operator**

The OUTER APPLY applies the right table source to each row in the left.

ALL rows from the left row are returned – values from the right table source are returned where they exist, otherwise NULL is returned.

Most LEFT OUTER JOIN statements can be rewritten as OUTER APPLY.

**LEFT OUTER JOIN**



select DISTINCT s.country as supplier\_country ,

a.country as cusomer\_country

FROM Production.Suppliers as s

left outer join sales.customers as a

on s.country = a.country

**OUTER APPLY**

select DISTINCT s.country as supplier\_country ,

c.country as cusomer\_country

FROM Production.Suppliers AS s

OUTER APPLY (

SELECT country

FROM Sales.Customers as c

WHERE s.country = c.country

) as c ;

**Both return same results**

**CROSS APPLY and OUTER APPLY Feature**

There are many similarities between CROSS APPLY and INNER JOIN, and OUTER APPLY, and LEFT JOIN.

However, the APPLY operation enables some types of query to be executed which could not be written using JOINs. These queries rely on the left table source being processed before being applied to the right table source. TWO examples shown in this topic are using a query returning top results for each input value and TVFs as the right table source.

**OUTER APPLY: Three Most Recent Orders Per Customer**

SELECT c.custid , topOrders.orderid , topOrders.orderdate

FROM sales.Customers as c

OUTER APPLY ( SELECT top(3) orderid , cast(o.orderdate as date) as orderdate

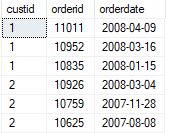
FROM Sales.Orders as o

where o.custid = c.custid

ORDER BY orderdate DESC , orderid desc

) as topOrders

If CROSS APPLY was used customers with no orders would not appear in the result set.



**CROSS APPLY calling a Table Valued Function TVF**

**Top 3 highest product price of each supplier**

--CREATING TVF

CREATE FUNCTION production.TopProductsByShipper

( @t AS INT

)

RETURNS TABLE

AS

RETURN

(

SELECT TOP(3) productid , productName , unitprice

FROM Production.products as od

where od.supplierid = @t

order by unitprice desc

)

/--/--/--/--/

select s.supplierid , s.contactname , p.productid , p.unitprice

from Production.Suppliers as s

CROSS APPLY

production.TopProductsByShipper(s.supplierid) as p

This **example uses the supplierid column from the left** input table **as an input parameter to the TVF** !!

Module 13

**Using Window Ranking Offset, and Aggregat Functions**

Lesson 1

**Creating Windows with Over**

**SQL Windowing**

SQL server provides windows as a method for applying functions to set of rows. There are many applications of this technique that solve common problems in writing T-SQL queries. For example, using windows allows the easy generation of row numbers in a result set and the calculation of running totals. Windows also provide an efficient way to compare values in one row with values in another without needing to join a table itself using an inequality operator.

There are several core elements of writing queries that use windows:

* Windows allow you to specify an order to rows that will be passed to a window function, without affecting the final order of the query output.
* Windows include a partitioning feature, which enables you to specify that you want to restrict a function only to rows that have the same value as the current row.
* Windows provide a framing option. It allows you to specify a further subset of rows within a window partition by setting upper and lower boundaries for the window frame, which presents rows to the window function.

The following examples uses an aggregate window function to calculate a running total, this illustrates the use of the elements:

-- Creating VIEW TABLE Sales.CategoryQtyYear use database TSQL2012

CREATE VIEW Sales.CategoryQtyYear AS

SELECT C.categoryname , SUM(OD.qty) AS Qty , YEAR(O.orderdate) as orderyear

FROM Production.Categories AS C

INNER JOIN Production.Products AS P

ON C.categoryid = p.categoryid

INNER JOIN Sales.OrderDetails AS OD

ON P.productid = OD.productid

INNER JOIN Sales.Orders AS O

ON OD.orderid = O.orderid

GROUP BY categoryNAME , YEAR(O.orderdate)

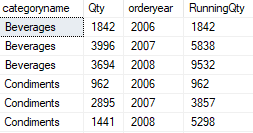
**Runing total example**

select categoryname , Qty , orderyear ,

SUM(QTY) OVER(PARTITION BY categoryname ORDER BY orderyear

ROWS UNBOUNDED PRECEDING ) AS RunningQty

FROM Sales.CategoryQtyYear



**Windowing Components**

In order to use windows and window functions in T-SQL you will always use one of the subclauses that create and manipulate windows-the OVER subclauses. Additionally, you may need to create partitions with the PARTITION BY option, and even further restrict which rows are applied to a function with framing options. Therefore, understanding the relationship between these components is vital.

The general relationship can be expressed as a sequence, with one element further manipulating the rows output by the element:

* The OVER clause determines the result set that will be used by the window function. An OVER clause with no partition defined is unrestricted. It returns all rows to the function.
* A PARTITION BY clause, if present, restricts the result to those with the same value in the partitioned columns as the current row. For example, PARTITION BY custid restricts the window to rows with the same custid as the current row. PARTITION BY builds in the OVER clause and cannot be used without OVER. (An OVER clause without a window partition is considered one partition).

select categoryname , Qty , orderyear ,

ROW\_NUMBER()

OVER(PARTITION BY categoryname ORDER BY orderyear)

AS PARTITIONED

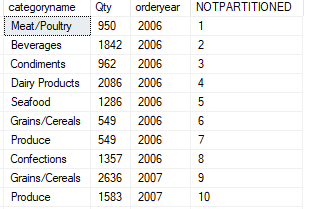
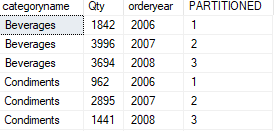
FROM Sales.CategoryQtyYear

select categoryname , Qty , orderyear ,

ROW\_NUMBER() OVER(ORDER BY orderyear )

AS NOT PARTITIONED

FROM Sales.CategoryQtyYear

* A ROW or Range clause creates a window frame within the window partition, which allows you to set a starting ending boundary around the rows being operated on. A frame requires an ORDER BY subclause within the OVER clause.

The following example, also seen in the previous topic, aggregates the Qty column against a window in the OVER clause defined by partitioning on the category column, sorting on the orderyear and framing by a boundary at the first row and boundary at the current row. This creates a ‘moving window’, where each row is aggregated with other rows of the same category value, from the oldest row by orderdate, to the current row

Window example:

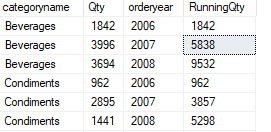
select categoryname , Qty , orderyear ,

SUM(QTY) OVER(PARTITION BY categoryname ORDER BY orderyear

ROWS BETWEEN UNBOUNDED PRECEDING AND current ROW)

AS RunningQty

FROM Sales.CategoryQtyYear;



NOTE : A single query can use multiple window function, each with its own OVER clause. Each clause determines its own partitioning, ordering and framing.

5838 + 3694 = 9532

1842 + 3996 = 5838

0 + 1842 = 1842

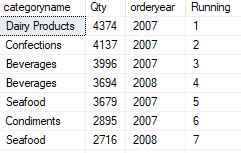
**Using OVER**

The OVER clause defines the window, or set of rows that will be operated on by window function. The OVER clause includes partitioning, ordering, and framing, where each is applicable.

Used alone, the OVER clause does not restrict the result set passed to the window function. Used with a PARTITION BY subclause, OVER restricts the set to those rows with the same values in the partitioning elements.

The following example shows the use of OVER without an explicit window partition to define an unrestricted window that will be used by the ROW-NUMBER function. All rows will be numbered, using an ORDER BY clause, which is required by ROW\_NUMBER. The row numbers will be displayed in a new column named Running:

select categoryname , Qty , orderyear ,



ROW\_NUMBER() OVER( ORDER BY qty desc ) AS Running

FROM Sales.CategoryQtyYear

**Partitioning Windows**

Partitioning a window limits a set to rows with the same value in the partitioning column.

Example, the following code snippet shows the use of PARTITION BY to create a window partition by category.

PARTITION BY code snippet:

<function\_name>() OVER( ORDER BY category)

AS yoy have learned, if no partition is defined, then the OVER() clause returns all the rows from the underlying query’s result set to the window.

The following example builds on the one you saw in the previous topic. It adds a PARTITION BY subclause to the OVER clause, creating a window partition for the rows with the matching Category values. This allows the ROW\_NUMBER function to number each set of years per category separately. Note that an ORDER BY subclause has been added to the PVER clause to provide meaning to the ROW\_NUMBER.



select categoryname , Qty , orderyear ,

ROW\_NUMBER() OVER(PARTITION BY categoryname

ORDER BY qty desc

) AS Running

FROM Sales.CategoryQtyYear

**Ordering and Framing**

As you have learned, you use window partitions to define a subset of rows within the outer window defined by OVER. In similar approach, window framing allows you to further restrict the rows available to the window function. You can think of a frame as a moving window over the data. Starting and Ending at positions you define.

To define window frames, use the ROW or RANGE subclauses to provide a starting and an ending boundary.

For example, to set a frame that extends from the first row in the partition to the current row ( such as to create a moving window for a running total), follow these steps

1. Define an OVER clause with PARTITION BY element.
2. Define an ORDER BY subclause to the over CLAUSE. This will cause the concept of ‘first row to be meaningful’.
3. Add the ROWS BETWEEN subclause, setting the starting boundary using:**UNBOUNDED PRECEDING .**

**UNBOUNDED** means go all the way to the boundary in the direction specified as **PRECEDING** (before).

Add the **CURRENT ROW** element indicate the ending boundary is the row being calculated.

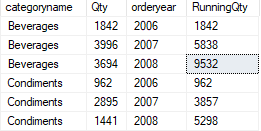
select categoryname , Qty , orderyear ,

sum(qty) OVER( PARTITION BY categoryname ORDER BY orderyear ASC

ROWS between UNBOUNDED PRECEDING and current row )

AS RunningQty

FROM Sales.CategoryQtyYear



Lesson 2

**Exploring Windows Functions**

**Defining Window Functions**

* A window function is a function applied to a window or set of rows.
* Window function include aggregate, ranking, distribution, and offset functions.
* Window functions depend on set created by OVER()

SQL Server functions can be found in the following categories:

* Aggregate functions, such as SUM , which operator window returns a single row
* Ranking functions, such as RAN, which depends on a sort order and return a value representing the rank of row, with respect to other rows in windows.
* Distribution functions, such as CUME\_DIST, wich calculates the distribution of a value in a window of rows.
* Offset functions, such as LEAD, which return values from other rows relative to the position of the current row.

When use in window scenarios, these functions depend on the result set returned by the OVER clause and any further restrictions you provide within OVER, such as partitioning and framing.

The following example uses the RANK function to calculate a rank of each row by unitprice, from high to low value.

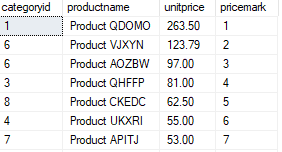
NOTE that there is no explicit window partition clause defined.

select categoryid , productname , unitprice,

row\_number() over( order by unitprice desc) as pricemark

FROM Production.Products

order by pricemark



For comprehension the following example adds a partition on category ( and adds category id to the final ORDER BY clause) Note the RANKING is calculated per partition.

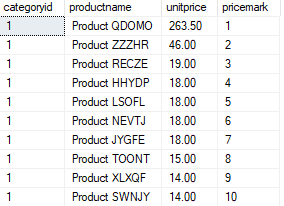
select categoryid , productname , unitprice

,row\_number() over( PARTITION BY categoryid order by unitprice desc) as pricemark

FROM Production.Products

order by categoryid , pricemark

**NOTICE in the result there are unit price values wich are ties but as a ROW\_NUMBER it keeps numbering**

****

**NOTE: in the code notice the** CAST**(**100. \*QTY/sum(qty)

OVER(PARTITION BY custid **)** AS NUMERIC(8,2)) AS OFTOTAL

* The AS NUMERIC(8,2))of CAST is set after the OVER is defined OVER(PARTITION BY custid **)** AS NUMERIC(8,2))

To get the values into a decimal state the data type is converted to NUMERIC(8,2) and the sum

* 100. \*QTY/sum(qty) notice there is a ‘.’ after the 100.

**Window Aggregte Functions**

* Similar to grouped aggregate functions
  + SUM, MAX, and so on
* Applied to window s defined by OVER clause.
* Window aggregate functions support partitioning, ordering, and framing.
* Unlike other window functions ordering is not required with aggregate functions, unless you are also specifying a frame.

**Example :**

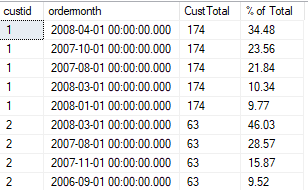
SELECT custid , ordermonth , SUM(QTY) OVER(PARTITION BY custid) AS CustTotal,

CAST(100. \*QTY/sum(qty) OVER(PARTITION BY custid )

AS NUMERIC(8,2)) AS [% of Total]

FROM Sales.CustOrders

ORDER BY custid , [% of Total] DESC

****

**Window Ranking Function**

Window Ranking Function returns a value representing the rank of a row with respect to other rows in the window. To accomplish this, ranking functions require an ORDER BY element within the OVER clause, to establish the position of each row within the window.

Note: Remember the ORDER BY element within the OVER clause affects only the processing of rows by the window function. To control the display order of the results, add an OREDER BY clause to the end of the SELECT statement, as with other queries.

The difference between RANK and DENSE\_RANK is the handling of rows when there are tie values.

Example : RANK and DENSE\_RANK

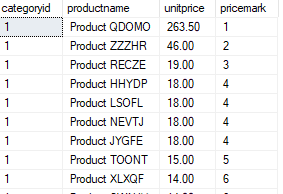
SELECT categoryid , productname , unitprice,

DENSE\_RANK() OVER( PARTITION BY categoryid

ORDER BY unitprice desc) as pricemark

FROM Production.Products

ORDER BY categoryid , pricemark



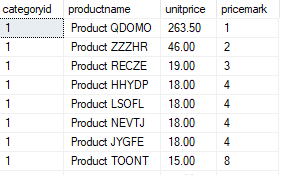
SELECT categoryid , productname , unitprice,

RANK() OVER( PARTITION BY categoryid

ORDER BY unitprice desc) as pricemark

FROM Production.Products

ORDER BY categoryid , pricemark

****

**Window distribution Functions**

Window distribution function performs statistical analysis on data, and require a window order clause

Rank distribution performed with PERCENT\_RANK and CUME\_DIST

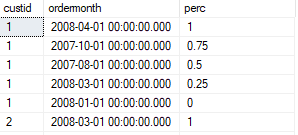
EXAMPLE OF PERCENT\_RANK() :

SELECT custid , ordermonth ,

PERCENT\_RANK() OVER(PARTITION BY custid ORDER BY QTY) AS CustTotal,

FROM Sales.CustOrders

ORDER BY custid , CustTotal DESC



Distribution Functions return a rank of a row, but instead of being expressed as an ordinary number, its expressed as a ratio between 0 and 1. I

SQL Server provides inverse distribution with the PERCENTILE\_CONT and PERCENTILE DISC Functions.

Windows Offset Functions

Windows offset functions give access to values located in rows other than the current rows, without the need to join the table itself.

Offset function operate in a position that is either **relative to the current row**, **or relatively to the starting or ending** boundary of the window frame. **LAG and LEAD** operate on an **offset to current row**. **FIRST\_VALUE and LAST\_VALUE** operate on an **offset from the window.**

NOTE: Since FIRST\_VALUE and LAST\_VALUE Operate on offsets from the window frame, it is important to remember to specify framing option to other than the default of RANGE BETWEEN UNBOUND PRECEDING AND CURRENT ROW.

Example using the LEAD function to compare year-over-year sales. The offset is 1, returning the next row`s value. Lead returns 0 if a NULL is found in the next row`s value, such as when there are no sales in the past.

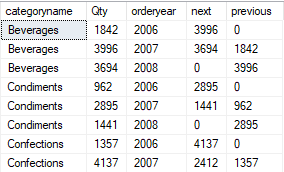
select categoryname , Qty , orderyear ,

lead(Qty , 1,0 )over( PARTITION BY categoryname order by orderyear ) as [next] ,

lag(Qty , 1,0 )over( PARTITION BY categoryname order by orderyear ) as previous

FROM Sales.CategoryQtyYear

order by categoryname ,orderyear



Module 14

**Pivoting and Grouping Sets**

Lesson 1

**Writing Queries with PIVOT and UNPIVOT**

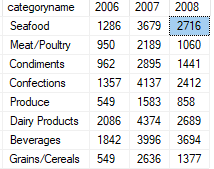
**What is Pivoting?**

Pivoting data in SQL Server rotates its display from a rows-based orientation to a columns-based orientation. It does this by consolidating values in a column to a list of distinct values, and then projecting values in a column to a list of distinct values, and then projecting that list across as column headings. Typically, this includes aggregation to column values in the new columns.

Example: The partial source data below lists repeating values for Category and Orderyear, along with values for QTY, for each instance of a Category/Orderyear pair:



To analyze this by category and year, you might want to arrange the values to be displayed as follows, Summing the qty column along the way :



In the pivoting process, each distinct year was created as a column header, and values in the Qty column were grouped by Category and aggregated.

**Elements of PIVOT**

The T-SQL PIVOT table operator, introduced in Microsoft SQL Server 2005, operates on the output of the FROM clause in a SELECT statement. To use PIVOT, you need to supply three elements to the operator.



* GROUPING: in the FROM clause, you need to provide the input columns. From those columns PIVOT will determine which column(s) will be used to group the data for aggregation. This is based on looking at which columns are not being used as other elements in the pivot

from ( select categoryname , qty , orderYear from Sales.categotyQtyYear ) as CQY

* SPREADING: you need to provide a comma-delimited list of values to be used as the column headings for the pivoting data. The values need to occur in the source data.
  + SELECT categoryname , [2006] , [2007] , [2008]
* AGGREGATION: You need to provide aggregation function (SUM,and so on) to be performed on the group rows.
  + PIVOT ( sum(qty) For orderyear IN ([2006] , [2007] , [2008])) AS PVT;

Additionally you need to provide a table alias to the result table of the PIVOT operator.

In the previous example, orderYear is the column providing the spreading values, qty is used for aggregation, and categoryname for grouping. Orderyear values are enclosed in delimiters to indicate that they are identifiers of columns in the result.

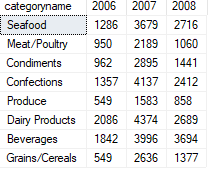
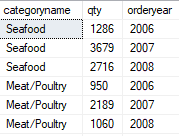
Note: Any attributes in the source subquery, that are not used for aggregation or spreading, will be used as grouping elements- be sure that no unnecessary attributes are included in the subquery.

**Writing Queries with UNPIVOT**

Unpivoting datat is the logical reverse of pivoting data. Instead of turning rows into columns, unpivot turns column into rows. This is useful in taking data that has already been pivoted (with or without using a T-SQL PIVOT operator) and returning it to row-oriented tabular display. SQL Server provides the UNPIVOT table operator to accomplish this.

When unpivoting data, one or more columns is defined as the source to be converted into rows. The data in those columns is spread to be converted into rows. The data in those columns is spread, or split, into one or more new rows, depending on how many rows have been unpivoted.

In the following source data, three columns will be unpivoted. Each orderyear value will be copied into a new row and associated with its categoryname value. Any NULLs will be removed in the process and no row is created:

Note: Unpivoting does not restore the original data. Detail-level data was lost during the aggregation process in the original pivot. UNPIVOT has no ability values to return to original detail values.

To use the UNPIVOT operator, you need to provide three elements:

* Source columns to be unpivoted.
* A name for the new column that will display the unpivoted values.
* A name for the column that will display the names of the unpivoted values.



Note: As we PIVOT, you will define the output of the UNPIVOT table operator as a derived table and provide its name.

Lesson 2

**Working with Grouping Sets**

You can us the GROUP BY clause in a SELECT statement to arrange rows in groups, typically to support aggregations. However if you need to group by different attributes at the same time, for example to report at different levels, you will need multiple queries combined with UNION ALL. SQL Server 2008 and later provides the GROUPING SETS subclause to GROUP BY, which enables multiple sets to be returned in the same query

**Writing queries with Grouping Sets**

If you need to produce aggregates of multiple groupings in the same query, you can use the GROUPING SETS subclause of the GROUP BY clause.

GROUPING SETS provide an alternative to using UNION ALL to combine results from multiple individual queries, each with its own GROUP BY clause.

Whit GROUPING SETS, you can specify multiple combinations of attributes on which to group, The following example uses GROUPING SETS to aggregate on the Category and Cust columns, in addition to the empty parentheses notation to aggregate all rows, as in the following example:

select categoryname , custid , sum(qty) as totalqty

FROM sales.categorySales

group by

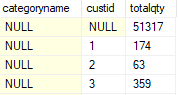
GROUPING SETS

(

(categoryname) , (custid) , ()

)

order by categoryname , custid;





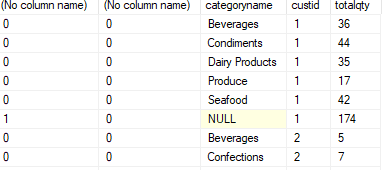
With GROUPING SETS, you can specify which attributes to group on and their order. If you want to group on any possible combination of attributes instead, see the topic on CUBE and ROLLUP later in this lesson.

The GROUPING\_ID function can be used to associate result rows with their grouping sets, as follows:

select GROUPING\_ID(categoryname) as groupCat, GROUPING\_ID(custid) as groupCast, categoryname , custid , sum(qty) as totalqty

FROM sales.categorySales

group by CUBE (categoryname , custid);



.......



As you can see the GROUPING\_ID function returns a 1 when a row is aggregated as part of the current grouping set and a 0 when it is not. In the first row, borh grpCat and grpCust return 0; therefore, the row is part of the grouping set(Categoryname , custid).

GROUPING\_ID can also take a multiple columns as inputs and return a unique integer bitmap, compared of combined bits, per grouping sets.

Module 15

**Executing Stored Procedures**

Lesson 1

**Querying Data with Stored Procedures**

**Examining Stored Procedures**

Stored procedures are named collections of T-SQL statements created with the **CREATE PROCEDURE** command. They encapsulate many server and database commands, and can provide a consistent application programming interface (API) to client applications using input parameters, Output parameters and return values.

Because in this course focuses primarily on retrieving results from databases through **SELECT** statements, this lesson will only cover the use of stored procedures that encapsulate SELECT queries. However, it might be useful to not that stored procedures can also include **INSERT, UPDATE, DELETE,** and other valid T-SQL commands.

They can also be used to provide an interface layer between a database and an application. Using such a layer, developers and administrators can ensure that all activity is performed by trusted code modules that validate input and handle errors appropriately.

Elements of such an API would include:

* Views or table-valued functions as wrappers for a simple retrieval.
* Stored procedures for retrieval when complex validation or manipulation is required.
* Stored procedures for inserting, updating or deleting rows.

In addition to encapsulate code and making it easier to maintain, this approach provides a security layer. Users may be granted access to objects rather than the underlying tables themselves. This ensures that users might only use the provided application to access data rather than other tools.

Stored Procedures also offer other benefits, including network and database performance improvements.

**Executing Stored Procedures**

* To execute a stored procedure, use the EXECUTE command or its shortcut EXEC, followed by the 2 part-name of the procedure. Your reporting tool may provide a graphical interface for selecting procedures by name, which will invoke the EXEC command for you.
* If the procedure accepts parameters, pass them as name-value pairs. For example if the parameter is called custid and the value is 5, use this form: @castid = 5. Multiple parameters are separated with commas

EXEC Production.ProdcutsBySuppliers

@supplierid = 1;

EXEC Production.ProdcutsBySuppliers

@supplierid = 1 , @numrows = 3;

* Pass parameters of the appropriate data type to the stored procedure. For example, if a procedure accepts NVARCHAR, pass the Unicode character string format N’string’.
* If the procedure encapsulates a simple SELECT statement, no additional elements are needed to execute it. If the procedure includes an OUTPUT parameter, additional steps will be required.

Lesson 2

**Passing Parameters to Stored Procedures**

**Passing Input Parameters to Stored Procedures**

Stored procedures can be written to accept input parameters to provide great flexibility. Procedures declare their parameters by name and data type in the header of the CREATE PROCEDURE statement, and then use the parameter as local variables in the body of the procedure.

For example, an input parameter might be used in the predicate of a WHERE clause or as the value in a TOP operator.

**EXAMPLE :**

createing a stores procedure that will accept 1 parameter as an input in @top

CREATE PROCEDURE sales.topQtySales

(@top AS INT)

AS

SELECT \*

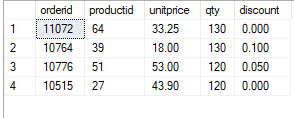
FROM Sales.OrderDetails

order by qty desc , unitprice desc

offset 0 rows fetch next @top rows only;

This will call the stored procedure and pass the parameter

EXEC Sales.topQtySales @top = 4; OR EXEC Sales.topQtySales 4;



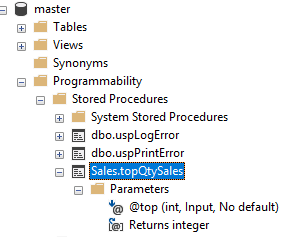
* To pass multiple input parameters, separate the name-value pairs with commas:

EXEC procedure\_Name @1stParameter = 4 , @2ndParameter = 6;

* To delete procedure:

DROP PROCEDURE sales.topQtySales

If you have not been provided with the names and data types of the parameters for the procedures, you will be executing, you can typically discover them yourself, assuming you have permissions to do so. SQL Server Managment Studio (SSMS) displays a parameters folder below each stored procedure that lists the names, types, and direction (input/output) of each definition parameter. Or hover the mouse on the procedure name.





**Working with OUTPUT Parameters**

So far in this module, we have seen procedures that return results through an embedded SELECT statement. SQL Server also gives you the capability to return a **scalar value** trough a parameter marked as an OUTPUT parameter. This has several benefits: a procedure can return a result set via SELECT statement and provide an additional value such as row count, to the calling application. For some specific scenarios where **only** a **single value** is desired, a procedure that returns an OUTPUT parameter can perform faster than a procedure that returns the scalar value in a result set.

There are two aspects to working with stored procedures using output parameters:

1. The procedure itself must mark a parameter with the OUTPUT keyword in the parameter decleration.

Creating a Stored Procedure with an OUTPUTParameter Example:

--THIS WILL CREATE A STORED PROCEDURE THAT:

--INPUTS A PARAMETER FOR PRODUCT ID

--OUTPUTS the number of orders that have been made for that specific Product ID

CREATE PROCEDURE sales.productNumbOfOrders

( @productID as int , @numbOfOrders as int OUTPUT)

AS

BEGIN

SELECT @numbOfOrders = count(productid)

from Sales.OrderDetails

where productid = @productID

end;

1. The T-SQL batch that calls the procedure must provide additional code to handle the output parameter. The code includes a local variable that acts as a container for the value that will be returned by the procedure when it executes. The parameter is added to the EXEC statement, marked with the OUTPUT keyword. After the stored procedure has completed, the variable will contain the value output parameter set inside the procedure.

The following example declares a local variable to be passed as the output parameters, executes a procedure, and then examines the variable with a SELECT statement:

Executing a Stored Procedure with OUTPUT Parameter Example

declare @count int

exec sales.productNumbOfOrders 10, @count output

select @count as [number of orders]



Lesson 3

**Creating Simple Stored Procedures**

**Creating Procedures that Returns Rows**

Stored procedures in SQL Server are used for many tasks, including system configuration and maintenance, in addition to data manipulation. As previously mentioned, there are advantages to create procedures to standardize access to data. To do that, you can create a stored procedure that is a wrapper for a SELECT statement, which might include any of the data manipulations you have already learned in this course. You can also modify design of procedure with ALTER PROCEDURE statement with no need to drop and recreate procedure and reassign security permissions.

**Example of a procedure that returns rows:**

CREATE PROCEDURE sales.OrderSummaries

AS

SELECT O.orderid , O.custid , O.empid , O.shipperid ,

CAST(O.orderdate AS DATE) AS orderdate,

SUM(OD.qty) as quantity,

CAST(SUM(OD.qty \* OD.unitprice \* (1 - OD.discount)) AS NUMERIC(12,2)) AS ordervalue

FROM Sales.Orders AS O INNER JOIN Sales.OrderDetails AS OD

ON O.orderid = OD.orderid

GROUP BY o.orderid, O.custid , O.empid , O.shipperid , o.orderdate;

**Executing a Procedure to Return Rows:**

EXEC Sales.OrderSummaries;



To modify the designe of the procedure, such as to change columns in the SELECT list or add an ORDER BY clause, use the ATLTER PROCEDURE (abbreviated ALTER PROC) statement and supply the full new code for the procedure.

Altering a Stored Procedure That Returns Rows

ALTER PROCEDURE sales.OrderSummaries

AS

SELECT O.orderid , O.custid , O.empid , O.shipperid ,

CAST(O.orderdate AS DATE) AS orderdate,

SUM(OD.qty) as quantity,

CAST(SUM(OD.qty \* OD.unitprice \* (1 - OD.discount)) AS NUMERIC(12,2)) AS ordervalue

FROM Sales.Orders AS O INNER JOIN Sales.OrderDetails AS OD

ON O.orderid = OD.orderid

GROUP BY o.orderid, O.custid , O.empid , O.shipperid , o.orderdate

🡪ORDER BY orderid , orderdate;

Changing the procedure with ALTER PROCEDURE is preferable than using DROP PROCEDURE to delete it, and then using CREATE PROCEDURE to rebuild it with a new definition. By altering it in place, security permissions do not need to be reassigned.

**Creating Procedures That Accepts Parameters**

A stored procedure that accepts input parameters provides added flexibility to its use. To define input parameters in your own stored procedures, declare them in the header of the CREATE PROCEDURES statement, then refer to them in the body of the stored procedure. Define the parameters with a @ prefix in the name, then assign them a data type.

Note: parameters may also be assigned default values, including NULL.

Example of a stored Procedure That Accepts Parameters:

CREATE PROCEDURE sales.OrderSummarieEmployee

(@empid AS INT)

AS

SELECT O.orderid , O.custid , O.empid , O.shipperid ,

CAST(O.orderdate AS DATE) AS orderdate,

SUM(OD.qty) as quantity,

CAST(SUM(OD.qty \* OD.unitprice \* (1 - OD.discount)) AS NUMERIC(12,2)) AS ordervalue

FROM Sales.Orders AS O INNER JOIN Sales.OrderDetails AS OD

ON O.orderid = OD.orderid

🡪WHERE empid = @empid

GROUP BY o.orderid, O.custid , O.empid , O.shipperid , o.orderdate

ORDER BY orderid , orderdate;

Executing a Stored Procedure That Accepts Parameters:

EXECUTE Sales.OrderSummarieEmployee @empid = 5

Lesson 4

**Working with Dynamic SQL**

**Creating Procedures that Returns Rows**

Dynamic SQL provides a mechanism for constructing a character string that is passed to SQL Server, interpreted as a command, and executed. Why would you want to do this? You might not know all the values necessary for your query until execution time-such as taking the results of one query and using them as inputs to another (for example, a pivot query) or an administrative maintenance routine that accepts object names at runtime.

T-SQL supports two methods for building dynamic SQL expressions-using the EXECUTE command (or its shortcut EXEC) with a string or invoking the system-stored procedure sp\_executesql :

1. The EXECUTE or EXEC command Supports the use of a string as an input in the following form, but does not support parameters, which need to be combined in the input string:

The following example shows how individual strings may be concatenated to form a command:

**Dynamic SQL Example**

DECLARE @sqlstring as VARCHAR(1000);

set @sqlstring='SELECT empid,' + 'lastname ' + ' FROM HR.employees;'

EXEC(@sqlstring);

GO

1. The system-stored procedure sp\_executesql supports string input for the query, in addition to input parameters.

The following example shows a simple string with a parameter passed to sp\_executesql:

**Passing Dynamic SQL with sp\_executesql**

DECLARE @sqlcode AS NVARCHAR(256) = N'SELECT GETDATE() AS dt'

EXEC sys.sp\_executesql @statement = @sqlcode;

It is important to know that EXEC cannot accept parameters and does not promote query plan reuse. Therefore, it is preferred that you use sys.sp\_executesql for passing dynamic SQL to SQL Server.

* Dynamic SQL is T-SQL code assembled into a character string, interpreted as a command, and executed.
* Dynamic SQL provides flexibility for administrative and programming tasks.
* Two methods for dynamically executing SQL statements:
  + EXEC command can accept a string as input in parentheses; no parameters may be passed in.
  + System-stored procedure sp\_executesql (preferred) supports parameters
* Beware of risks from invalid inputs in dynamic SQL.

**Writing Queries with Dynamic SQL**

This topic focuses on the preferred method of executing dynamic SQL, calling sp\_executessql. Constructing and executing dynamic SQL with sp\_executessql is preferred over using EXEC because EXEC cannot take parameters at runtime. In addition, sp\_executessql generates execution plans that are more likely to be reused than EXEC. Perhaps most important, though, using sp\_executessql can provide a line of defence against SQL injection attacks, by defining data types of parameters. And can also use output parameters marked with the OUTPUT keyword, which you learned earlier in this module.

To use sp\_executessql, provide a character string value that contain the query code as a parameter, as in the following example:

DECLARE @sqlcode AS NVARCHAR(256) = N'SELECT GETDATE() AS dt'

EXEC sys.sp\_executesql @statement = @sqlcode;

To use sp\_executesql with parameters, provide the query code, in addition to two additional parameters:

* @smtm a Unicode string variable to hold the query text
* @params, a Unicode string that holds comma-separated names and data types

In addition to these two variables, you will declare and assign variables to hold the value for the parameters you wish to pass in to sp\_executesql.

This example uses sp\_executesql to dynamically generate a query that returns an employee`s information based on an empid value:

DECLARE @sqlstring AS NVARCHAR(1000);

DECLARE @empid AS INT;

SET @sqlstring = N'SELECT empid, lastname FROM HR.employees WHERE empid = @empid;'

EXEC sp\_executesql @statement = @sqlstring, @params = N'@empid AS INT' , @empid = 5;



Module 16

**Programming with T-SQL**

Lesson 1

**T-SQL Programming Elements**

**Introducing T-SQL batches**

T-SQL batches are collections of one or more T-SQL statements that are submitted to SQL Server by a client as a single unit. SQL Server operates on all the statements in a batch at the same time when parsing, optimising, and executing the code.

If you are a report writer tasked primarily with writing SELECT statements and not procedures, it is still important to understand batch boundaries, because they will affect your work with variables and parameters in stored procedures and other routines. As you will see, you must declare a variable in the same batch in which it is referenced. It is important, therefore, to recognize what is contained in a batch.

Batches are delimited by the client application-how you mark the end of a batch will depend on the settings of your client. For example the default batch terminator in SQL Server Management Studio (SSMS) is the keyword GO. This is not a T-SQL keyword, but is one recognized by SSMS to indicate the end of a batch.

When working with T-SQL batches, there are two important considerations to keep in mind:

* Batches are boundaries for variables scope, which means that the variable defined in one batch may only be referenced by other code in the same batch.
* Some statements, typically data definition statements such as CREATE VIEW, CREATE FUNCTION, CREATE PROCEDURE, etc.. may not be combined with others in the same batch.

create PROCEDURE procedureName

AS

.....

GO

create VIEW viewreName

AS

.....

GO

**Working with Batches**

As you may have seen, batches are collections of T-SQL statements submitted as a unit in SQL Server for parsing, optimising and execution. Understanding how batches work will be useful in identifying error messages behaviour.

When a batch is submitted by a client ( such as when you press the Execute button in SSMS), the batch is parsed for syntax errors by the SQL Server engine. Any errors found will be no partial execution of statements within the batch.

If the batch passes the syntax check, then SQL Server proceeds with additional steps-resolving object names, checking permissions and optimizing the code for execution. Once this process completes and execution begins, statements succeed or fail individually. This is an important contrast to syntax checking. If a runtime error occurs on one line, the next line may be executed, unless you`ve added error handling code.

For example, the following batch contains a syntax error in the first line:

INSERT INTO t1 VALUE(1,2,N'abc'); 🡨 VALUE instead of VALUES

INSERT INTO t1 VALUES(2,3,N'def');

GO

Uppon submitting the batch, the following error returned:



The error occurred in line 1, but the entire batch is rejected, and an execution does not continue with line 2. Even if the error occurred in the second line, the first line would not be executed the entire batch would be rejected.

The following example will show an error which will occur in the third line of the first batch so the first batch will not execute due to the error in it but the following batch will execute.

INSERT INTO #t1 VALUEs(1,2,N'abc');

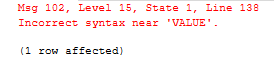
INSERT INTO #t1 VALUEs(2,3,N'def');

INSERT INTO #t1 VALUE(3,3,N'abc');

go

INSERT INTO #t1 VALUES(4,4,N'def');

GO



**Introducing T-SQL Variables**

In T-SQL, as with other programming languages, variables are objects that allow temporary storage of a value for later use. In T-SQL variable must be declared before they can be used. They may be assigned a value, or initialised, when they are declared. Declaring a variable includes providing a name and a data type as shown below.

Variables must be declared in the same batch in which they are referenced. In other words, all T-SQL variables are local in scope to the batch, both in visibility and lifetime. Only other statements in the same batch can see a variable declared in the batch. A variable is automatically destroyed (goes out of scope) when the batch ends.

The following example shows the use of variables to store values that will be passed to a stored procedure in the same batch:

**Using Variables**

--Declare and initialize the variables.

DECLARE @numrows INT = 3, @catid INT = 2;

--Use variables to pass the parameters to the procedure

EXEC Production.ProdsByCategory

@numrows = @numrows , @catid = @catid;

GO

**Working with Variables**

Once you have declared a variable, you must initialize it, or assign it a value. You can do that in three ways:

* In SQL Server 2008 or later, you may initialize a variable using the DECLARE statement.

DECLARE @numrows INT = 3, @catid INT = 2;

* In any version of SQL Server, you may assign a single (scalar) value using the SET statement.

DECLARE @numrows INT , @catid INT

SET @numrows = 3 SET @catid = 2 ;

* In any version of SQL Server, you can assign a value to a variable using SELECT statement. Be sure that the SELECT statement returns exactly one row. An empty result will leave the variable with its own original value; more than one result will cause an error.

DECLARE @numrows INT ;

SELECT @numrows = SUM(QTY) FROM Sales.OrderDetails;

The following example shows the three ways of declaring and assigning values to variables:

DECLARE @totalQty INT ;

DECLARE @string nvarchar(20) = N'whithout set';

DECLARE @number2 INT set @number2 = 2;

SELECT @totalQty = SUM(QTY) FROM Sales.OrderDetails;

SELECT @totalQty as totalQty , @STRING , @number2 as [with set];

GO



**Working with synonyms**

In SQL Server, synonyms provide a method for creating a link or alias, to an object stored in the same database or even another instance of SQL Server. Objects that might have synonyms defined for them include tables, views, stored procedures, and user-defined functions.

Synonyms can be used to make a remote object appear local or to provide an alternative name for a local object.

For example, synonyms can be used to provide an abstraction layer between client code and the actual database object used by code. The code references objects by their aliases, regardless of the object`s actual name.

Note: You can create a synonym which points an object that does not yet exist. This is called deffered name resolution. The SQL Server engine will not check for the existence of the actual object until the synonym is used to runtime.

To manage synonyms, use DDL commands CREATE SYNONYM, ALTER SYNONYM, DROP SYNONYM.

To create a synonym you must have a CREATE SYNONYM permission as well as permission to alter the schema in which the synonym will be stored.

**A synonym is useful to:**

1. Use an object from another database in the same server with an alias to have a shorter name such as :

While on database1 you need to use a table from database2

Instead of using the 3 part name 🡪 SELECT \* FROM database2.schema.table2

You can create a synonym to be used as an alias:

CREATE SYNONYM tbl2\_Synonym FOR database2.schema.table2;

SELECT \* FROM tbl2\_Synonym; 🡨 using synonym alias;

1. Use an object from another server with an alias to have a shorter name such as

Instead of using the 4 part name. The server must be linked.

Note: A synonym alias is only created in the database it is made in. To use the same synonym in another database you have to be recreated in the other database.

Lesson 2

**Controlling Program Flow**

**Understanding T-SQL Control-of-Flow Language**

SQL Server provides additional language elements that control the flow of execution of T-SQL Statements, used in batches, stored procedures and multistatement functions. These Control of Flow elements mean you can programmatically determine whether or not to execute statements and programmatically determin the order of those statements that should be executed.

These elements include, but not limited to:

* IF...ELSE, which executes code based on a Boolean expression
* WHILE, which creates a loop that executes providing a condition true.
* BEGIN...END, which defines a series of T-SQL statements that shoul be executed together.
* Other keywords (for example, BREAK, CONTINUE, WAITFOR, and RETURN), which are used to support T-SQL control-of-flow operations.

**Working with IF...ELSE**

The IF...ELSE structure is used in T-SQL to conditionally execute a block of code based on a predicate. The IF statement determines whether or not the following statement or block (id BEGIN...END is used) executes. If the predicate evaluates TRUE, the code in the block is executed. If the predicate evaluates to FALSE or UNKNOWN, the block is not executed, unless the option ELSE keyword is used to identify another block code.

For example, the following IF statements, without an ELSE, will only execute the statement between BEGIN and END if the predicate evaluates TRUE, indicating that the object exists. If it evaluates to FALSE or UNKNOWN, no action is taken and execution resumes after END statement:

**IF example**

IF OBJECT\_id('HR.EMPLOYEES') IS NULL

BEGIN

PRINT 'the specified object does not exist';

END;

With the use of ELSE, you have another execution option when IF predicate evaluates to FALSE or UNKNOWN :

**IF...ELSE Example**

IF OBJECT\_id('HR.EMPLOYEES') IS NULL

BEGIN

PRINT 'the specified object does not exist';

END

ELSE

BEGIN

PRINT 'the specified object exist';

END;

Within data manipulation operations, using IF with EXISTS keyword can be a useful tool for efficient existence checks:

IF EXISTS (SELECT \* FROM sales.EmpOrders WHERE empid = 5 )

BEGIN

PRINT 'Employee has associated orders';

END;

**Working with WHILE**

The WHILE statement is used to execute code in a loop based on a predicate.As long as the predicate evaluates to TRUE the statement in the while loop will keep repeating itself. Like IF statement, the WHILE statement determines whether the following statement or block (if BEGIN...END is used) executes. The loop ends when the predicate evaluates to FALSE or UNKNOWN. Typically, you control the loop with a variable tested by the predicate and manipulated in the body of the loop itself.

The following example uses the @empid variable in the predicate and changes its value in the BEGIN...END block:

DECLARE @empid AS INT = 1 ,@lastname AS NVARCHAR(20);

WHILE @empid <=5

BEGIN

SELECT @lastname = lastname FROM HR.employees

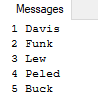
WHERE empid = @empid

PRINT @lastname

SET @empid += 1;

END;

NOTE: REMEBER that if SELECT returns UNKNOWN, the variable retains its current value. If there is no employee with an ID equal to @empid, the variable doesn’t change from one iteration to another. This would lead to an infinite loop.



For additional options within a WHILE loop, you can use the CONTINUE and BREAK keywords to control the flow.

Module 17

**Implementing Error Handling**

Lesson 1

**Implementing T-SQL ERROR Handling**

**Errors and Error Messages**

An error indicates a problem or notable issue arises during a database operation.

Each error includes the following elements:

* Error number: Unique number identifying specific error
* Error message: Text describing the error
* Severity: Numeric indication of seriousness from 1 to 25
* State: Internal state code for the database engine condition
* Procedure: The name of the stored procedure or trigger in which the error occurred
* Line number: Which statement in the batch or procedure generated the error

Errors can be generated by the SQL Server Database Engine in response to an event or failure at the system level; or you can generate application errors in your Transact-SQL code.

**System Errors**

System errors are predefined, and you can view them in the sys.messages system view. When a system error occurs SQL Server may take automatic remedial action, depending on the severity of the error. For example, when a high-severity occurs, SQL Server may take a database offline or even stop the database engine service.

**Custom Errors**

You can generate errors in Transact-SQL code to respond to application-specific conditions or to customize information sent to client applications in response to system errors. These application errors can be defined inline where they are generated, or you can predefine them in the sys.messages tablealongside the system-supplied errors. The error numbers used for custom errors must be 50001 or above.

To **add** a custom error messages to sys.messages, use sp\_addmessage.

To **delete** a custom error message usesp\_dropmessage**.**

The user for the message must be a member of the sysadmin or serveradmin fixed server roles.

**sp\_addmessage Syntax**

Exec sp\_addmessage [ @msgnum= ] msg\_id , [ @severity= ] severity , [ @msgtext= ] 'msg'

[ , [ @lang= ] 'language' ]

[ , [ @with\_log= ] { 'TRUE' | 'FALSE' } ]

[ , [ @replace= ] 'replace' ]

**sp\_dropmessage Syntax**

exec sp\_dropmessage [ @msgnum = ] message\_number

[ , [ @lang = ] 'language' ]

Note: Unless **all** is specified for language, all localized versions of a message must be dropped before the U.S. English version of the message can be dropped

**Sp\_addmessage syntax Example**

exec sys.sp\_addmessage 50001 , 10 , N'unsepected value entered' ;



In addition to being able to define custom error messages, members of the sysadmin server role can also use an additional parameter, **@with\_log**. When set to TRUE, the error will be recognised in the Window Application log. When set to TRUE, the error will also be recorded in the server error log. Be judicious with the use of the @with\_log option because network and system administrators tend to dislike applications that are ‘chatty’ in the system logs. However, **if the error needs to be trapped by an alert, the error must first be written to the Window Application log**.

Note that raising system errors is not supported.

Messages can be replaced without deleting them first by using the @replace = ‘replace ’ option.

The messages are customizable and different once can be added for the same error number for multiple languages, based on a language\_id value (NOTE: English messages are language\_id 1033.)

**Raising Errors Using RAISERROR**

Both PRINT and RAISERERROR can be used to return information on warning messages to applications.

RAISERROR allows applications to raise an error that could then be caught by the calling process.

**RAISERROR**

The ability to raise errors in T-SQL makes error handling in the application easier, because it is sent like any other system error.

RAISERROR is used to:

* Help Troubleshoot T-SQL code
* Check the values of data
* Return messages that contain variable text.

Note that using PRINT statement is similar to raising an error severity 10 (will be in black colour instead of read).

RAISERROR (N'This is message %s %d.', -- Message text.

10, -- Severity,

1, -- State,

N'number' , -- First argument.

5 ); -- Second argument.



**Substitution Placeholders and Message Number**

Note that, in the message shown in the example on the slide, %d is a placeholder for a number and %s is a placeholder for a string. Note also that a message number was not mentioned. When errors with message strings are raised using this syntax, they always have error number 50000.

**Raising Errors Using THROW**

The THROW statement offers a simpler method of raising errors in code. Errors must have an error number of at least 50000

**THROW**

THROW differs from RAISERROR in several ways:

* Errors raised by THROW are always severity 16
* The messages returned by THROW are not related to any entries in sys.sysmessages.
* Errors raised by THROW only cause transaction abort when used in conjunction with SET XACT\_ABORT ON and the session is terminated.

THROW 51000, 'The record does not exist.', 1;



**Using @@ERROR**

@@ERROR is a system variable that holds the error number of the last error that has occurred. One significant challenge with @@ERROR is that the value it holds is quickly reset as each additional statement is executed.

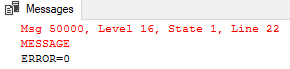
**@@ERROR Example:**

RAISERROR(N'MESSAGE' , 16 , 1);

IF @@ERROR <> 0

PRINT 'ERROR=' + cast(@@ERROR AS VARCHAR(8));

You might expect that, when the code is executed, it would return he error number in a printed string. However, when the code is executed, it returns:



Note that the error was raised but the message printed was ‘ERROR = 0’. In the first line of the output, you can see that the error, as expected, was actually 50000, with a message passed to RAISERROR. This is because the IF statement that follows the RAISERROR statement was executed successfully and caused the @@ERROR to be reset.

For this reason when working with @@ERROR, it is important to capture the error number into a variable as soon as it is raised, and then continue processing with the variable.

**Capturing @@ERROR into variable:**

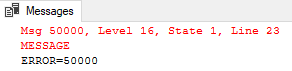
DECLARE @Errorvalue INT;

RAISERROR(N'MESSAGE' , 16 , 1);

SET @Errorvalue = @@ERROR

IF @Errorvalue <> 0

PRINT 'ERROR=' + cast(@Errorvalue AS VARCHAR(8));



**Centralising Error Handling**

One other significant issue using @@ERROR for error handling is that it is difficult to centralize within your T-SQL code. Error handling tends to end up scattered throughout the code. It would be possible to centralize error handling using @@ERROR to some extent, by using labels and GOTO statements. However, this would be frowned by most developers today as a poor coding practice.

**Creating Alerts When Error Occur**

For certain categories of errors, administrators might create SQL Server alerts, because they wish to be notified as soon as these occur. This can even apply to user-defined error messages. For example, you might want to raise an alert whenever a transaction log fills. Alerting is commonly used to bring high severity errors (such as severity 19 and above) to the attention of administrator.

**Raising ALERTS**

Alerts can be created for specific error messages. The altering service works by registering itself a callback service with the event logging service. This means alerts only work on logged errors.

There are 2 ways to make error raise an alert-you can use the WITH LOG option when raising the error or the message can be alerted to make it logged by executing sp\_altermessage. The WITH LOG option affects only the current statement. Using sp\_altermessage changes the error behaviour for all future use.

Lesson 2

**Implementing Structured Exception Handling**

**TRY/CATCH Block Progrmming**

Structured exception handling is more powerful than error handling based on the @@ERROR system variables. It allows you to prevent code from being littered with error handling code and to centralize that error handling code.

Centralization of error handling code also means you can focus more on the purpose of the code rather than the error handling it contains.

**TRY Block and CATCH Block**

When using structured exception handling, code that might raise an error is placed within a TRY block. TRY blocks are enclosed by BEING TRY and END TRY statements.

Should a catchable error occur (most errors can be caught), execution control moves to the CATCH block.

The CATCH block is a series of T-SQL statements enclosed by BEGIN CATCH and END CATCH statements.

Note that, while BEGIN CATCH and END TRY are separate statements, the BEGIN CATCH must immediately follow the END TRY

* No code may be placed between END TRY and BEGIN CATCH.
* TRY and CATCH blocks may be nested

**Current Limitations**

High level languages often offer a try/catch/finally

construct, and are often used to release resources implicity. There is no equivalent FINALLY block in T-SQL.

**Error Handling Functions**

CATCH blocks make the error-related information available throughout the duration of the CATCH block. This includes subscopes, such as stored procedures, run from within the CATCH block.

You should recall that, when programming with @@ERROR, the value held by the @@ERROR system variable was reset as soon as the next statement was executed.

Another key advantage of structured exception handling in T-SQL is that a series of error handling functions has been provided and these retain their values throughout the CATCH block. Separate functions provide each property of an error that has been raised.

This means you can write generic error handling stored procedures that can still access the error-related information.

**Catchable vs Noncatchable Errors**

Not all errors can be caught by TRY/CATCH blocks within the same scope where the TRY/CATCH block exists. Often errors that cannot be caught in the same scope can be caught in a surrounding scope. For example, you might not be able to catch an error within the stored procedure that contains the TRY/CATCH block. However, you are likely to catch that error in a TRY/CATCH block in the code that called the stored procedure where the error occurred.

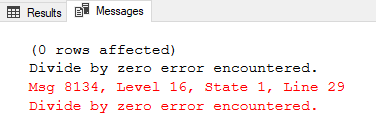
**Common Noncatchable Errors**

Common example of noncatchable errors are:

* Compile errors, such as syntax errors, that prevent a batch from compiling.
* Statement level recompilation issues that usually relate to deferred name resolutition. For example, you could create a stored procedure where the error occurred.

**Rethrowing Errors Using THROW**

If the THROW statement is used in a CATCH block without any parameters, it will rethrow the error that caused the code to enter the CATCH block. You can use this technique to implement error logging in the database by catching errors and logging their details, and then throwing the original error to the client application, so that it can be handled there.



BEGIN TRY

SELECT 1/0

END TRY

BEGIN CATCH

PRINT ERROR\_MESSAGE();

THROW ;

END CATCH;

**Errors in Managed Code**

It is important to realise that any errors not handled in the managed code are passed back to the calling T-SQL code. Whenever any error that occurs in managed code is returned to SQL Server, it will appear to be a 6522 error. Error can be nested and that particular error will be wrapping the real cause of the error.

Another rare but possible cause of errors in managed code would be that the code could execute a RAISERROR T-SQL statement via SqlCommand object.

Module 18

**Implementing Transactions**

Lesson 1

**Transactions and the Database Engine**

**Defining Transactions**

Earlier in this course, you learned that a batch was a collection of T-SQL statements sent to SQL Server as a unit for parsing, optimization, and execution. A transaction extends a batch from a unit submitted to the database engine to a unit of work performed by the database engine. A transaction is a sequence of T-SQL statements performed in an all-or-nothing fashion by SQL Server.

A transaction is a single unit of work. If a transaction is successful, all of the data modifications made during the transaction are committed and become a permanent part of the database. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.

SQL Server operates in the following transaction modes:

**Autocommit transactions**

Individual data modification statement (INSERT, CREATE , DELETE , ETC,....) submitted separately from other commands are automatically wrapped in a transaction by SQL Server . These single statement transactions are automatically committed when the statement succeeds, or are automatically rolled back when the statement encounters a runtime error.

**Explicit transactions**  
Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.

**Implicit transactions**  
A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.

**Batch-scoped transactions**  
Applicable only to multiple active result sets (MARS), a Transact-SQL explicit or implicit transaction that starts under a MARS session becomes a batch-scoped transaction. A batch-scoped transaction that is not committed or rolled back when a batch completes is automatically rolled back by SQL Server.



The primary characteristic of a transaction is that all activity within a transaction`s boundaries must either succeed or all fail – no partial completion is permitted. User transactions are typically defined to encapsulate operations that must logically occur together, such as entries into related tables as part of a single business operation.

For example, the following batch inserts data into two tables using INSERTS statements that are part of a single order-processing operation:

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (67,9,'2006-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (68,10,'2006-08-22');

INSERT INTO #orderDetails (orderID , productID , unitPrice , qty)

VALUES (1,2,15.20,20);

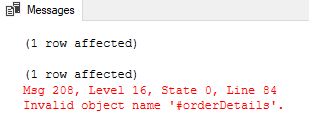
INSERT INTO #simpleOrders (custID,empID,orderdate)

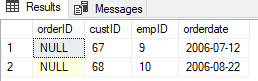
VALUES (69,11,'2006-08-12');

GO

There was an error in the third INSERT.:

* The first 2 where inserted
* The third one had an error
* The fourth on did not execute as the process stopped in the third one





Business rules might dictate that an order is complete only if the data was successfully inserted into both tables.

As you have seen in the previous example and will see in the next lesson, a runtime error in this batch might result in data being inserted in one table but not in the other. Enclosing both INSERT statements in a user-defined transaction provides the ability to undo the data insertion in one table if the INSERT statement in the other table fails. A simple batch does not provide this capability

SQL Server manages resources on behalf of transactions while they are active.

These resources might include locks and entries in the transaction log to allow SQL Server to undo changes made by the transaction, should a rollback be required.

**The Need for Transactions: Issues with Batches**

While batches of T-SQL statements provide a unit of code submitted to server, they do not include logic for dealing with partial success when a runtime error occurs, even with the use of structured exceptions handling`s TRY/CATCH blocks.

The following example illustrates this problem:

**Code without transaction**

BEGIN TRY

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (67,9,'2006-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (68,10,'20-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (69,11,'2006-08-12');

END TRY

BEGIN CATCH

SELECT ERROR\_NUMBER() as Errnum , ERROR\_MESSAGE() as Errmsg;

END CATCH;

GO

This is the SELECT statement in the BEGIN CATCH



This is the data that entered in the table



The first insert succeeded but the subsequent one failed. The new row entered in #simpleOrders will persist after the end of the batch, even after the execution branches to the CATCH block. This issue applies to any successful statements, if a later statement fails with a runtime error.

NOTE: Remember that syntax or name-resolution errors (**noncatchable errors)** cause the entire batch to return an error, preventing any execution. Runtime errors only occur after the batch has been submitted, parsed, planned, and compiled for execution.

To work around this situation you will need to direct SQL Server to treat the batch as a transaction.

**Transaction Extend Batches**

As you have seen, runtime errors encountered during the execution part of simple batches create the possibility of partial success, which is not typically a desired outcome. To address this, you will add code to identify the batch as a transaction by placing the batch between BEGIN TRANSACTION and COMMIT TRANSACTION statements. You will also add error-handling code to roll back the transaction should an error occur. This error-handling code will undo the partial changes made before the error occurred.

The following example shows the addition of TCL commands to address possibility of an error occurring after some work has been performed:

**Transaction Example:**

BEGIN TRY

BEGIN TRANSACTION;

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (67,9,'2006-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (68,10,'20-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (69,11,'2006-08-12');

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

SELECT ERROR\_NUMBER() as Errnum , ERROR\_MESSAGE() as Errmsg;

ROLLBACK TRANSACTION;

END CATCH;

GO

Within the TRY block, the INSERT statements are wrapped by BEGIN TRANSACTION and COMMIT TRANSACTION statements. This identifies the INSERT statements as a single unit of work that must be succeed or fail together. If no runtime error occurs the transaction commits and the result of each INSERT is allowed.

If an error occurs during the execution of the first insert statement the execution branches to the CATCH block, bypassing the other INSERT statements. The ROLLBACK statement in the CATCH block terminates the transaction, releasing its resources.

If an error occurs during the execution of the second or other insert statement the execution branches to the CATCH block. Because the first INSERT completed successfully and added rows to the table, the ROLLBACK statement is used to undo the successful INSERTY operation.

Lesson 2

**Controlling Transactions**

**BEGIN TRANSACTION**

Marks the starting point of an explicit, local transaction. Explicit transactions start with the BEGIN TRANSACTION statement and end with the COMMIT or ROLLBACK statement.

If you are using T-SQL structure exception handling you will want to begin the transaction inside a TRY block. Within the exception handler, you may decide whether to COMMIT or ROLLBACK the transaction depending on its outcome.

When you identify your own transactions with BEGIN TRANSACTION, consider the following:

* Once you initiate a transaction, you must properly end it. Use COMMIT TRANSACTION on success or ROLLBACK TRANSACTION on failure.
* While transactions may be nested, inner transactions will be rolled back, even if committed, if the outer transaction rolls back. Therefore are not typically useful in user code.
* Transactions last until a COMMIT TRANSACTION or a ROLLBACK TRANSACTION is issued, or until the originating connection is dropped, at which point SQL Server will roll back the transaction automatically.
* A transaction`s scope is the connection in which it was started. Transactions cannot span connections.
* SQL Server may take a hold locks on resources during the life span of the transaction. To reduce concurrency issues, consider keeping your transaction as short as possible.

**COMMIT TRANSACTION**

COMMIT ensures all of the transaction`s modifications are made a permanent part of the database and

frees resources, such as locks used by the transaction.

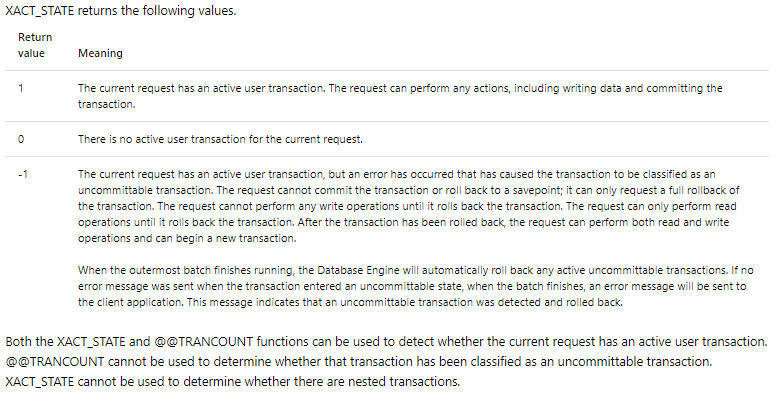
If you are using T-SQL structured except handling, you will want to COMMIT TRANSACTION inside the TRY block in which it began.

**ROLLBACK TRANSACTION**

It undoes all modifications made in the transaction by retrieving the data to the state it was in at the beginning of the transaction.IT frees resources, such as locks used by the transaction.

If you are using T-SQL structured except handling, you will want to ROLLBACK TRANSACTION inside the CATCH block in which it began.

Before issuing a ROLLBACK command, you might wish to test to see if a transaction is active. You can use the T-SQL XACT\_STATE function to determine if there is an active transaction to be rolled back. This can help avoid errors being raised inside the CATCH



XACT\_STATE Example :

BEGIN TRY

BEGIN TRANSACTION;

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (67,9,'2006-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (68,10,'20-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (69,11,'2006-08-12');

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

SELECT ERROR\_NUMBER() as Errnum , ERROR\_MESSAGE() as Errmsg;

IF( XACT\_STATE()) = -1

BEGIN

ROLLBACK TRANSACTION;

END;

ELSE PRINT 'SOMETHING';

END CATCH;

GO

**@@TRANCOUNT**

Returns the number of BEGIN TRANSACTION statements that have occurred on the current connection

**Using XACT\_ABORT**

Specifies whether SQL Server automatically rolls back the current transaction when a Transact-SQL statement raises a run-time error.

By default the XACT\_ABORT is off. Change the XACT\_ABORT setting with the SET command :

SET XACT\_ABORT ON;

After finishing the transaction do not forget to turn it back off unless you want it ON.

When SET XACT\_ABORT is ON, the entire transaction is terminated and rolled back on error, unless the error occurs in a TRY block. An error in a TRY block leaves the transaction open but not commitable despite the setting of XACT\_ABORT.

In this example it will roll back as there is an error and XACT\_ABORT is ON Example :

SET XACT\_ABORT On;

BEGIN TRANSACTION;

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (67,9,'2006-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (68,10,'20-07-12');

INSERT INTO #simpleOrders (custID,empID,orderdate)

VALUES (69,11,'2006-08-12');

Go

To view the current setting for this setting, run the following query.

DECLARE @XACT\_ABORT VARCHAR(3) = 'OFF';

IF ( (16384 & @@OPTIONS) = 16384 ) SET @XACT\_ABORT = 'ON';

SELECT @XACT\_ABORT AS XACT\_ABORT;