<https://mva.microsoft.com/en-US/training-courses/querying-microsoft-sql-server-2012-databases-jump-start-8241>

--ddl or data definition language: create alter drop

--dml or data manipulation language: select insert update delete (CRUD operations)

--dcl or data control language: grant revoke deny

**T-SQL enforces operator precedence**

|  |  |
| --- | --- |
| **Elements:** | **Predicates and Operators:** |
| Predicates | IN, BETWEEN, LIKE |
| Comparison Operators | =, >, <, >=, <=, <>, !=, !>, !< |
| Logical Operators | AND, OR, NOT |
| Arithmetic Operators | +, -, \*, /, % |
| Concatenation | + |

**T-SQL functions**:

String functions: SUBSTRING,LEFT, RIGHT,LEN,DATALENGTH, REPLACE,REPLICATE,UPPER, LOWER,RTRIM, LTRIM

Date and time functions: GETDATE,SYSTDATETIME,GETUTCDATE,DATEADD,DATEDIFF,YEAR,MONTH,DAY

Aggregate functions: SUM,MIN,MAX,AVG,COUNT

**T-SQL variables**:

Local variables in T-SQL temporarily store a value of a specific data type

Name begins with single @ sign

@@ reserved for system functions

Assigned a data type

Must be declared and used within the same batch

In SQL Server 2008 and later, can declare and initialize in the same statement

**DECLARE @MyVar int = 30;**

**T-SQL expressions**:

Combination of identifiers, values, and operators evaluated to obtain a single result

Can be used in SELECT statements

SELECT clause

WHERE clause

Can be single constant, single-valued function, or variable

Can be combined if expressions have same the data type

**SELECT YEAR(OrderDate) + 1 ...**

**SELECT OrderQty \* UnitPrice ...**

**T-SQL batch separators**:

Batches are sets of commands sent to SQL Server as a unit

Batches determine variable scope, name resolution

To separate statements into batches, use a separator:

SQL Server tools use the GO keyword

GO is not a SQL Server T-SQL command

**T-SQL flow control, errors and transactions**:

Used in programmatic code objects such as stored procedures, triggers, statement blocks

Flow control: IF...ELSE,WHILE,BREAK,CONTINUE,BEGIN...END

Error handling: TRY…CATCH

Transaction control: BEGIN TRANSACTION,COMMIT TRANSACTION,ROLLBACK TRANSACTION

The order in which a query is written is not the order in which it is evaluated by the server. The order is:

5: SELECT <select list>

1: FROM <table source>

2: WHERE <search condition>

3: GROUP BY <group by list>

4: HAVING <search condition>

6: ORDER BY <order by list>

USE AdventureWorks2014;

SELECT SalesPersonID, YEAR(OrderDate) AS OrderYear

FROM Sales.SalesOrderHeader

WHERE CustomerID = 29974

GROUP BY SalesPersonID, YEAR(OrderDate)

HAVING COUNT(\*) > 1

ORDER BY SalesPersonID, OrderYear;

When performance tuning, using SELECT 1 gives you stats for just speaking to sql server on network(rather than what happens inside the database engine)

**Advanced SELECT clauses (DISTINCT, aliases, CASE, and scalar functions)**

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 prior to execution of SELECT clause

SELECT DISTINCT <column list>

FROM <table or view>

**SELECT DISTINCT StoreID**

**FROM Sales.Customer;**

Using aliases to refer to columns: only ‘AS’ is the standard. Rest are here for legacy reasons

**SELECT SalesOrderID, UnitPrice, OrderQty AS Quantity**

**FROM Sales.SalesOrderDetail;**

**SELECT SalesOrderID, UnitPrice, Quantity =** **OrderQty**

**FROM Sales.SalesOrderDetail;**

**SELECT SalesOrderID, UnitPrice** **Quantity**

**FROM Sales.SalesOrderDetail;**

Using aliases to refer to tables: using alias makes name shorter and also then u can use same table multiple times(with different aliases) in joins

**SELECT SalesOrderID, ProductID FROM Sales.SalesOrderDetail AS SalesOrders;**

**SELECT SalesOrderID, ProductID FROM Sales.SalesOrderDetail SalesOrders;**

**SELECT SalesOrders.SalesOrderID, SalesOrders.ProductID FROM Sales.SalesOrderDetail AS SalesOrders;**

**T-SQL case expressions**:

Simple CASE

Compares one value to a list of possible values and returns first match

If no match, returns value found in optional ELSE clause

If no match and no ELSE, returns NULL

Searched CASE

Evaluates a set of predicates, or logical expressions

Returns value found in THEN clause matching first expression that evaluates to TRUE

T-SQL CASE expressions return a single (scalar) value

CASE expressions may be used in:

SELECT column list (behaves as calculated column requiring an alias)

WHERE or HAVING clauses

ORDER BY clause

**SELECT ProductID, Name, ProductSubCategoryID,**

**CASE ProductSubCategoryID**

**WHEN 1 THEN 'Beverages'**

**ELSE 'Unknown Category'**

**END**

**FROM Production.Product**

**Joins**:

|  |  |
| --- | --- |
| **Join Type** | **Description** |
| Cross | Combines all rows in both tables (creates Cartesian product). |
| Inner(the default join) | Starts with Cartesian product; applies filter to match rows between tables based on predicate. |
| Outer | Starts with Cartesian product; all rows from designated table preserved, matching rows from other table retrieved. Additional NULLs inserted as placeholders. |

Inner Join: As it is the default join, just specify join for inner join

Returns only rows where a match is found in both tables

Matches rows based on attributes supplied in predicate

ON clause in SQL-92 syntax

Why filter in ON clause?

Logical separation between filtering for purposes of JOIN and filtering results in WHERE

‘on’ filter is applied before the ‘where’ filter

Typically no difference to query optimizer

If JOIN predicate operator is =, also known as equi-join

**SELECT SOH.SalesOrderID, SOH.OrderDate, SOD.ProductID, SOD.UnitPrice, SOD.OrderQty**

**FROM Sales.SalesOrderHeader AS SOH**

**JOIN Sales.SalesOrderDetail AS SOD**

**ON SOH.SalesOrderID = SOD.SalesOrderID;**

Outer Join:

Returns all rows from one table and any matching rows from second table

One table’s rows are “preserved”

Designated with LEFT, RIGHT, FULL keyword

All rows from preserved table output to result set

Matches from other table retrieved

Additional rows added to results for non-matched rows

NULLs added in place where attributes do not match

Example: Return all customers and for those who have placed orders, return order information. Customers without matching orders will display NULL for order details.

Customers that did not place orders

SELECT CUST.CustomerID, CUST.StoreID, ORD.SalesOrderID, ORD.OrderDate

FROM Sales.Customer AS CUST

LEFT OUTER JOIN Sales.SalesOrderHeader AS ORD

ON CUST.CustomerID = ORD.CustomerID

WHERE ORD.SalesOrderID IS NULL;

Cross join:

Combine each row from first table with each row from second table

All possible combinations are displayed

Logical foundation for inner and outer joins

INNER JOIN starts with Cartesian product, adds filter

OUTER JOIN takes Cartesian output, filtered, adds back non-matching rows (with NULL placeholders)

Due to Cartesian product output, not typically a desired form of JOIN

Some useful exceptions:

Generating a table of numbers for testing

Example:

Create test data by returning all combinations of two inputs:

SELECT EMP1.BusinessEntityID, EMP2.JobTitle

FROM HumanResources.Employee AS EMP1

CROSS JOIN HumanResources.Employee AS EMP2;

Self Join:

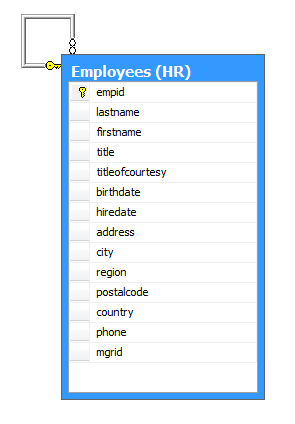
Why use self-joins?

Compare rows in same table to each other

Create two instances of same table in FROM clause

At least one alias required

Example: Return all employees and the name of the employee’s manager



Return all employees with ID of employee’s manager when a manager exists (INNER JOIN):

SELECT EMP.EmpID, EMP.LastName, EMP.JobTitle, EMP.MgrID, MGR.LastName

FROM HR.Employees AS EMP

INNER JOIN HR.Employees AS MGR

ON EMP.MgrID = MGR.EmpID;

Return all employees with ID of manager (OUTER JOIN). This will return NULL for the CEO:

SELECT EMP.EmpID, EMP.LastName,

EMP.Title, MGR.MgrID

FROM HumanResources.Employee AS EMP

LEFT OUTER JOIN HumanResources.Employee AS MGR

ON EMP.MgrID = MGR.EmpID;

Order By clause:

ORDER BY sorts rows in results for presentation purposes

Use of ORDER BY guarantees the sort order of the result

Last clause to be logically processed(processed after SELECT)

Sorts all NULLs together

ORDER BY can refer to:

Columns by name, alias or ordinal position (not recommended)

Columns not part of SELECT list unless DISTINCT clause specified

Declare sort order with ASC or DESC

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

ORDER BY OrderDate;

SELECT SalesOrderID, CustomerID, YEAR(OrderDate) AS OrderYear

FROM Sales.SalesOrderHeader

ORDER BY OrderYear;

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

ORDER BY OrderDate DESC;

WHERE clause:

WHERE clauses use predicates

Must be expressed as logical conditions

Only rows for which predicate evaluates to TRUE are accepted

Values of FALSE or UNKNOWN are filtered out

WHERE clause follows FROM, precedes other clauses

Can’t see aliases declared in SELECT clause

Can be optimized by SQL Server to use indexes

SELECT CustomerID, TerritoryID

FROM Sales.Customer

WHERE TerritoryID = 6;

SELECT CustomerID, TerritoryID

FROM Sales.Customer

WHERE TerritoryID >= 6;

SELECT CustomerID, TerritoryID, StoreID

FROM Sales.Customer

WHERE StoreID >= 1000 AND StoreID <= 1200;

Filtering data in the SELECT clause:

TOP allows you to limit the number or percentage of rows returned

Works with ORDER BY clause to limit rows by sort order

If ORDER BY list is not unique, results are not deterministic (no single correct result set)

Modify ORDER BY list to ensure uniqueness, or use TOP WITH TIES

Added to SELECT clause:

SELECT TOP (N) | TOP (N) Percent

With percent, number of rows rounded up

SELECT TOP (N) WITH TIES

Retrieve duplicates where applicable (nondeterministic)

TOP is proprietary to Microsoft SQL Server

SELECT TOP (20) SalesOrderID, CustomerID, TotalDue

FROM Sales.SalesOrderHeader

ORDER BY TotalDue DESC;

--this might show u 23 rows

SELECT TOP (20) WITH TIES SalesOrderID, CustomerID, TotalDue

FROM Sales.SalesOrderHeader

ORDER BY TotalDue DESC;

SELECT TOP (1) PERCENT SalesOrderID, CustomerID, TotalDue

FROM Sales.SalesOrderHeader

ORDER BY TotalDue DESC;

Offset: could be used for paging…It offsets the result set by the number of records specified. Here we are getting rows 21 to 30.

select \* from Production.Product

order by ListPrice, ProductID

offset 20 rows

fetch next 10 rows only;

Union: if only say union, then the combined result set would have distinct records. UNION ALL would remove duplicates

--i had to use wrapper selects as otherwise it threw an error

select \* from (

select top(2) Name,ListPrice,Color from Production.Product as p

where Color='Black'

order by ListPrice desc

) as a

Union all

select \* from(

select top(2) Name,ListPrice,Color from Production.Product as p

where Color='Red'

order by ListPrice desc

) as b

Null values:

Different components of SQL Server handle NULL differently

Query filters (ON, WHERE, HAVING) filter out UNKNOWNs

CHECK constraints accept UNKNOWNS

ORDER BY, DISTINCT treat NULLs as equals

Testing for NULL

Use IS NULL or IS NOT NULL rather than = NULL or <> NULL

SELECT CustomerID, StoreID, TerritoryID

FROM Sales.Customer

WHERE StoreID IS NULL

ORDER BY TerritoryID

SQL server data types:

SQL Server associates columns, expressions, variables, and parameters with data types

Data types determine what kind of data can be stored in the field:

Integers, characters, dates, money, binary strings, etc.

SQL Server supplies several built-in data types

Developers can also define custom types

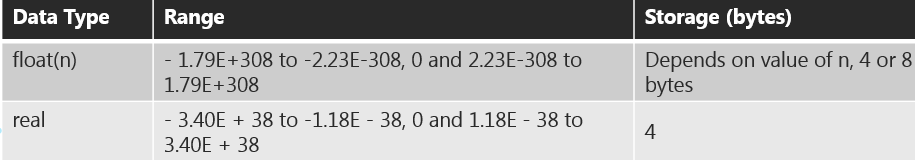
Aliases in T-SQL

User-defined types in .NET code

Built-in data types are categorized as shown in the table below

|  |  |
| --- | --- |
| **SQL Server Data Type Categories** | |
| Exact numeric | Unicode characters |
| Approximate numeric | Binary strings |
| Date and time | Other |
| Character strings |  |

Approximate Numeric types:



float(24) is the ISO synonym for float

In float(n), n is the number of bytes used to store the mantissa of the float number in scientific notation

Values of float are truncated when converted to integer types

Exact Numeric types:

|  |  |  |
| --- | --- | --- |
| **Data type** | **Range** | **Storage (bytes)** |
| tinyint | 0 to 255 | 1 |
| smallint | -32,768 to 32,767 | 2 |
| int | 2^31 (-2,147,483,648) to  2^31-1 (2,147,483,647) | 4 |
| Bigint | -2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807) | 8 |
| bit | 1, 0 or NULL | 1 |
| decimal/numeric | - 10^38 +1 through 10^38 – 1 when maximum precision is used | 5-17 |
| money | -922,337,203,685,477.5808 to 922,337,203,685,477.5807 | 8 |
| smallmoney | - 214,748.3648 to 214,748.3647 | 4 |

Decimal/numeric are functionally equivalent and use precision and scale parameters:

DECLARE @mydecimal AS DECIMAL(8,2)

Note that the bit field still uses 1 byte. Strange.

Binary String data types: for storing blobs. Say image data.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Range** | **Storage (bytes)** |
| binary(n) | 1-8000 bytes | n bytes |
| varbinary(n) | 1-8000 bytes | actual length + 2 |
| varbinary(MAX) | 1-2.1 billion (approx) bytes | actual length + 2 |

Other data types:

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Range** | **Storage (bytes)** | **Remarks** |
| rowversion | Auto-generated | 8 | Successor type to timestamp |
| uniqueidentifier | Auto-generated | 16 | Globally unique identifier (GUID) |
| xml | 0-2 GB | 0-2 GB | Stores XML in native hierarchical structure |
| cursor | N/A | N/A | Not a storage data type |
| hierarchyid | N/A | Depends on content | Represents position in a hierarchy |
| sql\_variant | 0-8000 bytes | Depends on content | Can store data of various data types |
| table | N/A | N/A | Not a storage data type, used for query and programmatic operations |

Converting strings with parse:

PARSE is new function in SQL Server 2012 that converts strings to date, time, and number types

|  |  |
| --- | --- |
| **sPARSE element** | **Comment** |
| String\_value | Formatted nvarchar(4000) input |
| Data\_type | Requested data type ouput |
| Culture | Optional string in .NET culture form: en-US, es-ES, ar-SA, etc. |

SELECT PARSE('02/12/2012' AS datetime2 USING 'en-US') AS parse\_result;

--this is the iso format as well as the ansi format. yyyy-mm-dd

select OrderDate from Sales.SalesOrderHeader

select PARSE('2011-05-31' as datetime using 'en-US')

select try\_parse('hejc' as datetime using 'en-US')

Parse is going from string to date and Format will go form date to string

SELECT FORMAT(ORDERDATE,'yyyy:MM:dd HH','sv-SE' )FROM SALES.SALESORDERHEADER

Character data types:

SQL Server supports two kinds of character data types:

Regular: CHAR, VARCHAR

One byte stored per character

Only 256 possible characters – limits language support

Unicode: NCHAR, NVARCHAR

Two bytes stored per character

65k characters represented – multiple language support

Precede characters with N' (National)

TEXT, NTEXT deprecated

Use VARCHAR(MAX), NVARCHAR(MAX) instead

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Range** | **Storage** |
| CHAR(n),  NCHAR(n) | 1-8000 characters | n bytes, padded  2\*n bytes, padded |
| VARCHAR(n), NVARCHAR(n) | 1-8000 characters | Actual length +2 bytes  2\* (Actual length) +2 bytes |
| VARCHAR(MAX), NVARCHAR(MAX) | 1-2^31-1 characters | Actual length + 2 |

CHAR, NCHAR are fixed length

VARCHAR, NVARCHAR are variable length

Character data is delimited with single quotes

SQL Server uses the + (plus) sign to concatenate characters: Concatenating a value with a NULL returns a NULL

SELECT BusinessEntityID, FirstName, LastName, FirstName + N' ' + LastName AS FullName

FROM Person.Person;

SQL Server 2012 introduces CONCAT() function: Converts NULL to empty string before concatenation

SELECT AddressLine1, City, StateProvinceID,  CONCAT(AddressLine1, ', ' + City, ', ' + PostalCode) AS Location

FROM Person.Address

Character string functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Syntax** | **Remarks** |
| SUBSTRING() | SUBSTRING (expression , start , length) | Returns part of an expression |
| LEFT(), RIGHT() | LEFT (expression , integer\_value) RIGHT (expression , integer\_value) | LEFT() returns left part of string up to integer\_value. RIGHT() returns right part of string. |
| LEN(), DATALENGTH() | LEN ( string\_expression )  DATALENGTH ( expression ) | LEN() returns the number of characters of the specified string expression, excluding trailing blanks. DATALENGTH() returns the number bytes used. |
| CHARINDEX() | CHARINDEX ( expressionToFind, expressionToSearch ) | Searches an expression for another expression and returns its starting position if found. Optional start position. |
| REPLACE() | REPLACE ( string\_expression , string\_pattern , string\_replacement ) | Replaces all occurrences of a specified string value with another string value. |
| UPPER(), LOWER() | UPPER ( character\_expression )  LOWER ( character\_expression ) | UPPER() returns a character expression with lowercase character data converted to uppercase. LOWER() converts uppercase to lowercase. |

Like predicate:

The LIKE predicate used to check a character string against a pattern

Patterns expressed with symbols

% (Percent) represents a string of any length

\_ (Underscore) represents a single character

[<List of characters>] represents a single character within the supplied list

[<Character> - <character>] represents a single character within the specified range

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

ESCAPE Character allows you to search for a character that is also a wildcard character (%, \_, [, ] for example)

SELECT ProductLine, Name,ProductNumber

FROM Production.Product

WHERE Name LIKE 'Mountain%'

Performance: If the wild card '%' is after the string, then the index on Name column would be used for searching and thus would be fast. If the wild card is at the start of the string, then index could not be used and a scan would have to be performed and would be slow.

Date and time data types:

Older versions of SQL Server supported only DATETIME and SMALLDATETIME

DATE, TIME, DATETIME2, and DATETIMEOFFSET introduced in SQL Server 2008

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Type** | **Storage (bytes)** | **Date Range** | **Accuracy** | **Recommended Entry Format** |
| DATETIME | 8 | January 1, 1753 to December 31, 9999 | 3-1/3 milliseconds | 'YYMMDD hh:mm:ss:nnn' |
| SMALLDATETIME | 4 | January 1, 1900 to June 6, 2079 | 1 minute | 'YYMMDD hh:mm:ss:nnn' |
| DATETIME2 | 6 to 8 | January 1, 0001 to December 31, 9999 | 100 nanoseconds | 'YYMMDD hh:mm:ss.nnnnnn' |
| DATE | 3 | January 1, 0001 to December 31, 9999 | 1 day | 'YYYY-MM-DD' |
| TIME | 3 to 5 |  | 100 nanoseconds | 'hh:mm:ss:nnnnnnn' |
| DATETIMEOFFSET | 8 to 10 | January 1, 0001 to December 31, 9999 | 100 nanoseconds | 'YY-MM-DD hh:mm:ss:nnnnnnn [+|-]hh:mm' |

--datettime2 works correctly due to precision/accuracy...

--datetime would round up here...anyway, u would be off by anwhere between 3 to 1/3 milliseconds and tha could play up in

--a unexpected way.So .997 is taken as the end of the day

select CAST('2013-09-13 23:59:59.999' as datetime) ;--returns 2013-09-14 00:00:00.000

select CAST('2013-09-13 23:59:59.998' as datetime) ;--returns 2013-09-13 23:59:59.997

select CAST('2013-09-13 23:59:59.997' as datetime) ;--returns 2013-09-13 23:59:59.997

select CAST('2013-09-13 23:59:59.99' as datetime) ;--returns 2013-09-13 23:59:59.990

select CAST('2013-09-13 23:59:59.999' as datetime2) ;--returns 2013-09-13 23:59:59.9990000

we should use < tomorrow in datetime predicates to avoid problems like these

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Language-Neutral Formats** | **Examples** |
| DATETIME | 'YYYYMMDD hh:mm:ss.nnn'  'YYYY-MM-DDThh:mm:ss.nnn'  'YYYYMMDD' | '20120212 12:30:15.123'  '2012-02-12T12:30:15.123'  '20120212' |
| SMALLDATETIME | 'YYYYMMDD hh:mm'  'YYYY-MM-DDThh:mm'  'YYYYMMDD' | '20120212 12:30'  '2012-02-12T12:30'  '20120212' |
| DATETIME2 | 'YYYY-MM-DD'  'YYYYMMDD hh:mm:ss.nnnnnnn'  'YYYY-MM-DD hh:mm:ss.nnnnnnn'  'YYYY-MM-DDThh:mm:ss.nnnnnnn'  'YYYYMMDD'  'YYYY-MM-DD' | '20120212 12:30:15.1234567'  '2012-02-12 12:30:15.1234567'  '2012-02-12T12:30:15.1234567'  '20120212'  '2012-02-12' |
| DATE | 'YYYYMMDD'  'YYYY-MM-DD' | '20120212'  '2012-02-12' |
| TIME | 'hh:mm:ss.nnnnnnn' | '12:30:15.1234567' |
| DATETIMEOFFSET | 'YYYYMMDD hh:mm:ss.nnnnnnn [+|-]hh:mm'  'YYYY-MM-DD hh:mm:ss.nnnnnnn [+|-]hh:mm'  'YYYYMMDD'  'YYYY-MM-DD' | '20120212 12:30:15.1234567 +02:00'  '2012-02-12 12:30:15.1234567 +02:00'  '20120212'  '2012-02-12' |

SQL Server doesn't offer an option for entering a date or time value explicitly

Dates and times are entered as character literals and converted explicitly or implicitly

For example, CHAR converted to DATETIME due to precedence

Formats are language-dependent, can cause confusion

Best practices:

Use character strings to express date and time values

Use language-neutral formats

--this works

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

WHERE OrderDate = '20110531';

--this works as well

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

WHERE OrderDate = '2011-05-31';

DATETIME, SMALLDATETIME, DATETIME2, and DATETIMEOFFSET include both date and time data

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

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

DECLARE @DateOnly DATETIME = '20120212';

SELECT @DateOnly;--returns 2012-02-12 00:00:00.000

Date values converted from character literals often omit time

Queries written with equality operator for date will match midnight

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

WHERE OrderDate = '20110531';

If time values are stored, queries need to account for time past midnight on a date

Use range filters instead of equality

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

WHERE OrderDate >= '20110531' and OrderDate< '20110601' ;

Date and time functions:

|  |  |  |
| --- | --- | --- |
| **Function** | **Return Type** | **Remarks** |
| GETDATE() | datetime | Current date and time. No time zone offset. |
| GETUTCDATE() | datetime | Current date and time in UTC. |
| CURRENT\_TIMESTAMP | datetime | Current date and time. No time zone offset. ANSI standard. |
| SYSDATETIME() | datetime2 | Current date and time. No time zone offset |
| STSUTCDATETIME() | datetime2 | Current date and time in UTC. |
| SYSDATETIMEOFFSET() | datetimeoffset | Current date and time. Includes time zone offset |

SELECT CURRENT\_TIMESTAMP;

SELECT SYSUTCDATETIME();

Functions that return part of date and times:

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Syntax** | **Return Type** | **Remarks** |
| DATENAME() | DATENAME(datepart, date) | nvarchar | Use 'year', 'month', 'day' as datepart |
| DATEPART() | DATEPART(datepart, date) | int | Use 'year', 'month', 'day' as datepart |
| DAY() | DAY(datevalue) | int |  |
| MONTH() | MONTH(datevalue) | int |  |
| YEAR() | YEAR(datevalue) | int |  |

SELECT DATENAME(year,'20120212');

SELECT DAY('20120212');

Functions that return date and time from parts:

|  |  |  |
| --- | --- | --- |
| **Function** | **Syntax** | **Return Type** |
| DATEFROMPARTS() | DATEFROMPARTS(year, month, day) | date |
| DATETIMEFROMPARTS() | DATETIMEFROMPARTS(year, month, day, hour, minute, seconds, milliseconds) | datetime |
| DATETIME2FROMPARTS() | DATETIME2FROMPARTS(year, month, day, hour, minute, seconds, fractions, precision) | Datetime2 |
| DATETIMEOFFSETFROMPARTS() | DATETIMEOFFSETFROMPARTS(year, month, day, hour, minute, seconds, fractions, hour\_offset, minute\_offset, precision) | datetime |
| SMALLDATETIMEFROMPARTS() | SMALLDATETIMEFROMPARTS(year, month, day, hour, minute) | smalldatetime |
| TIMEFROMPARTS() | TIMEFROMPARTS(hour, minute, seconds, fractions, precision) | time |
|  |  |  |

SELECT DATEFROMPARTS(2012,2,12);

SELECT DATETIME2FROMPARTS(2012,2,12,8,30,0,0,0);

Functions that modify date and time values:

|  |  |  |
| --- | --- | --- |
| **Function** | **Syntax** | **Remarks** |
| DATEADD() | DATEADD(datepart, interval, date) | Adds interval to date, returns same datatype as date |
| EOMONTH() | EOMONTH(start\_date, interval) | Returns last day of month as start date, with optional offset |
| SWITCHOFFSET() | SWITCHOFFSET(datetimeoffset, time\_zone) | Changes time zone offset |
| TODATETIMEOFFSET() | TODATETIMEOFFSET(expression, time\_zone) | Converts datetime2 into datetimeoffset |

SELECT DATEADD(day,1,'20120212');

SELECT EOMONTH('20120212');

Functions that operate on date and time values:

|  |  |  |
| --- | --- | --- |
| **Function** | **Syntax** | **Remarks** |
| DATEDIFF() | DATEDIFF(datepart, start\_date, end\_date) | Returns the number of boundaries crossed for the specified datepart |
| ISDATE() | ISDATE(expression) | Determines whether a datetime or smalldate time is a valid value |

Converting with cast:

Converts a value from one data type to another

Can be used in SELECT and WHERE clauses

ANSI standard

Truncation can occur if converting to smaller data type

CAST Example:

SELECT CAST(SYSDATETIME() AS date) AS 'TodaysDate';

Returns an error if data types are incompatible:

--attempt to convert datetime2 to int

SELECT CAST(SYSDATETIME() AS int);

Converting with convert:

Converts a value from one data type to another

Can be used in SELECT and WHERE clauses

CONVERT is specific to SQL Server, not standards-based

Style specifies how input value is converted:

Date, time, numeric, XML, etc.

Example:

SELECT CONVERT(CHAR(8), CURRENT\_TIMESTAMP,112) AS ISO\_style;

**SQL Server 2012 built-in function types**:

|  |  |
| --- | --- |
| **Function Category** | **Description** |
| Scalar | Operate on a single row, return a single value |
| Grouped Aggregate | Take one or more values but return a single, summarizing value |
| Window | Operate on a window (set) of rows |
| Rowset | Return a virtual table that can be used subsequently in a T-SQL statement |

Scalar functions:

Scalar function categories

* Configuration
* Conversion
* Cursor
* Date and Time
* Logical
* Mathematical
* Metadata
* Security
* String
* System
* System Statistical
* Text and Image

Operate on elements from a single row as inputs, return a single value as output.

Return a single (scalar) value

Can be used like an expressionin queries

May be deterministic or non-deterministic

Collation depends on input value or default collation of database

SELECT SalesOrderID, YEAR(OrderDate) AS OrderYear

FROM Sales.SalesOrderHeader;

SELECT ABS(-1.0), ABS(0.0), ABS(1.0);

SELECT CAST(SYSDATETIME() AS date);

SELECT DB\_NAME() AS current\_database;

Window functions:

Functions applied to a window, or set of rows

Include ranking, offset, aggregate and distribution functions

SELECT TOP(5) ProductID, Name, ListPrice, RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice

FROM Production.Product

ORDER BY RankByPrice;

749 Road-150 Red, 62 3578.27 1

750 Road-150 Red, 44 3578.27 1

751 Road-150 Red, 48 3578.27 1

752 Road-150 Red, 52 3578.27 1

753 Road-150 Red, 56 3578.27 1

select ROW\_NUMBER() over (order by SalesOrderDetailID) as rownum1

,\*

from Sales.SalesOrderDetail as t

order by SalesOrderDetailID

select ROW\_NUMBER() over (order by SalesOrderDetailID) as rownum1

, ROW\_NUMBER() over (partition by salesorderid order by SalesOrderDetailID) as rownum1

,\*

from Sales.SalesOrderDetail as t

order by SalesOrderDetailID

select ROW\_NUMBER() over (order by SalesOrderDetailID) as rownum1

, ROW\_NUMBER() over (partition by salesorderid order by SalesOrderDetailID) as rownum1

, SUM(UnitPrice) over (partition by salesorderid ) as totalorderprice

,\*

from Sales.SalesOrderDetail as t

order by SalesOrderDetailID

select ROW\_NUMBER() over (order by SalesOrderDetailID) as rownum1

, ROW\_NUMBER() over (partition by salesorderid order by SalesOrderDetailID) as rownum1

, SUM(UnitPrice) over (partition by salesorderid ) as totalorderprice

, SUM(UnitPrice) over () as totaltotalorderprice

,\*

from Sales.SalesOrderDetail as t

order by SalesOrderDetailID

SELECT TOP(5) ProductID, Name, ListPrice, RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice

FROM Production.Product

ORDER BY RankByPrice;

**Writing logical tests with functions**:

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

Returns a 1 when the input evaluates to any valid numeric type, including FLOAT and MONEY, otherwise returns 0

SELECT ISNUMERIC('SQL') AS isnmumeric\_result;

SELECT ISNUMERIC('101.99') AS isnmumeric\_result;

**Performing conditional tests with IIF**:

IIF returns one of two values, depending on a logical test

Shorthand for a two-outcome CASE expression

|  |  |
| --- | --- |
| **IIF Element** | **Comments** |
| Boolean\_expression | Logical test evaluating to TRUE, FALSE, or UNKNOWN |
| True\_value | Value returned if expression evaluates to TRUE |
| False\_value | Value returned if expression evaluates to FALSE or UNKNOWN |

SELECT ProductID, ListPrice,

IIF(ListPrice > 50, 'high', 'low') AS PricePoint

FROM Production.Product;

**Selecting items from a list with CHOOSE**:

CHOOSE returns an item from a list as specified by an index value

CHOOSE example:

SELECT CHOOSE (3, 'Beverages', 'Condiments', 'Confections') AS choose\_result;

**Aggregate Functions**:

* SUM
* MIN
* MAX
* AVG
* COUNT
* COUNT\_BIG
* STDEV
* STDEVP
* VAR
* VARP
* CHECKSUM\_AGG
* GROUPING
* GROUPING\_ID

Return a scalar value (with no column name)

Ignore NULLs except in COUNT(\*)

Can be used in

SELECT, HAVING, and ORDER BY clauses

Frequently used with GROUP BY clause

SELECT COUNT (DISTINCT SalesOrderID) AS UniqueOrders,

AVG(UnitPrice) AS Avg\_UnitPrice, MIN(OrderQty)AS Min\_OrderQty, MAX(LineTotal) AS Max\_LineTotal

FROM Sales.SalesOrderDetail;

Using DISTINCT with aggregate functions:

Use DISTINCT with aggregate functions to summarize only unique values

DISTINCT aggregates eliminate duplicate values, not rows (unlike SELECT DISTINCT)

Compare (with partial results):

SELECT SalesPersonID, YEAR(OrderDate) AS OrderYear,

COUNT(CustomerID) AS All\_Custs,

COUNT(DISTINCT CustomerID) AS Unique\_Custs

FROM Sales.SalesOrderHeader

GROUP BY SalesPersonID, YEAR(OrderDate);

--all orders

select \* from Sales.SalesOrderHeader

--now i want to get maximum of 1 order per customer??

--first how many customers do i have

select COUNT(distinct CustomerID) from Sales.SalesOrderHeader--returns 19119

--so i should have 19119 orders returned as a result of actual query to be answered

--group by would not work..

--select \* from Sales.SalesOrderHeader

--group by CustomerID

--we need all the order details...not only the customer id

select CustomerID from Sales.SalesOrderHeader

group by CustomerID

--i can't use distinct as it will use ditinct over the entrier row. But i can use this row\_number thing for this problem.

select

ROW\_NUMBER() over (partition by customerID order by OrderDate desc) as rn

,\*

from Sales.SalesOrderHeader

--i cant use this as rn column name is not avaiable at the time of execution of where clause

--select

--ROW\_NUMBER() over (partition by customerID order by OrderDate desc) as rn

--,\*

--from Sales.SalesOrderHeader

--where rn =1

--now i cant do this as all the ranking functions are calculated at select time

--select

--ROW\_NUMBER() over (partition by customerID order by OrderDate desc) as rn

--,\*

--from Sales.SalesOrderHeader

--where ROW\_NUMBER() over (partition by customerID order by OrderDate desc) =1

--but this would work...u can compare the count of rows returned and it would be 19119

select \* from(

select

ROW\_NUMBER() over (partition by customerID order by OrderDate desc) as rn

,\*

from Sales.SalesOrderHeader

) as a

where rn=1;

--now lets say i want to get all orders that are made in the same day. now if they are made in the same day, they will get different row\_numbers

--but if i use rank, then all the order with the same order date will come back with same rank..this will return more that 19119 rows

select \* from(

select

rank() over (partition by customerID order by OrderDate desc) as rn

,\*

from Sales.SalesOrderHeader

) as a

where rn=1;

**Group by clause**:

GROUP BY creates groups for output rows, according to unique combination of values specified in the GROUP BY clause

SELECT <select\_list>

FROM <table\_source>

WHERE <search\_condition>

GROUP BY <group\_by\_list>;

GROUP BY calculates a summary value for aggregate functions in subsequent phases

SELECT SalesPersonID, COUNT(\*) AS Cnt

FROM Sales.SalesOrderHeader

GROUP BY SalesPersonID;

Detail rows are “lost” after GROUP BY clause is processed

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

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

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

|  |  |  |
| --- | --- | --- |
| **Logical Order** | **Phase** | **Comments** |
| 5 | SELECT |  |
| 1 | FROM |  |
| 2 | WHERE |  |
| 3 | GROUP BY | Creates groups |
| 4 | HAVING | Operates on groups |
| 6 | ORDER BY |  |

If a query uses group by, then all subsequent phases operate on the group, not on the source rows

Using GROUP BY with aggregate functions:

Aggregate functions are commonly used in SELECT clause, summarize per group:

SELECT CustomerID, COUNT(\*) AS cnt

FROM Sales.SalesOrderHeader

GROUP BY CustomerID;

Aggregate functions may refer to any columns, not just those in GROUP BY clause

SELECT productid, MAX(OrderQty) AS largest\_order

FROM Sales.SalesOrderDetail

GROUP BY productid;

Filtering grouped data using HAVING Clause:

HAVING clause provides a search condition that each group must satisfy

HAVING clause is processed after GROUP BY

SELECT CustomerID, COUNT(\*) AS Count\_Orders

FROM Sales.SalesOrderHeader

GROUP BY CustomerID

HAVING COUNT(\*) > 10;

Compare HAVING to WHERE clauses:

WHERE filters rows before groups created

Controls which rows are placed into groups

HAVING filters groups

Controls which groups are passed to next logical phase

Using a COUNT(\*) expression in HAVING clause is useful to solve common business problems:

Show only customers that have placed more than one order:

SELECT Cust.Customerid, COUNT(\*) AS cnt

FROM Sales.Customer AS Cust

JOIN Sales.SalesOrderHeader AS Ord ON Cust.CustomerID = ORD.CustomerID

GROUP BY Cust.CustomerID

HAVING COUNT(\*) > 1;

Show only products that appear on 10 or more orders:

SELECT Prod.ProductID, COUNT(\*) AS cnt

FROM Production.Product AS Prod

JOIN Sales.SalesOrderDetail AS Ord ON Prod.ProductID = Ord.ProductID

GROUP BY Prod.ProductID

HAVING COUNT(\*) >= 10;

**Working with subqueries**:

Subqueries are nested queries or queries within queries

Results from inner query are passed to outer query

Inner query acts like an expression from perspective of outer query

Subqueries can be self-contained or correlated

Self-contained subqueries have no dependency on outer query

Correlated subqueries depend on values from outer query

Subqueries can be scalar, multi-valued, or table-valued

Scalar subquery returns single value to outer query

Can be used anywhere single-valued expression can be used: SELECT, WHERE, etc.

SELECT SalesOrderID, ProductID, UnitPrice, OrderQty

FROM Sales.SalesOrderDetail

WHERE SalesOrderID =

(SELECT MAX(SalesOrderID) AS LastOrder

FROM Sales.SalesOrderHeader);

If inner query returns an empty set, result is converted to NULL

Construction of outer query determines whether inner query must return a single value

Multi-valued subquery returns multiple values as a single column set to the outer query

Used with IN predicate

If any value in the subquery result matches IN predicate expression, the predicate returns TRUE

SELECT CustomerID, SalesOrderId,TerritoryID

FROM Sales.SalesorderHeader

WHERE CustomerID IN (

SELECT CustomerID

FROM Sales.Customer

WHERE TerritoryID = 10);

May also be expressed as a JOIN (test both for performance)

The keyword EXISTS does not follow a column name or other expression.

The SELECT list of a subquery introduced by EXISTS typically only uses an asterisk (\*).

SELECT CustomerID, PersonID

FROM Sales.Customer AS Cust

WHERE EXISTS (

SELECT \*

FROM Sales.SalesOrderHeader AS Ord

WHERE Cust.CustomerID = Ord.CustomerID);

SELECT CustomerID, PersonID

FROM Sales.Customer AS Cust

WHERE NOT EXISTS (

SELECT \*

FROM Sales.SalesOrderHeader AS Ord

WHERE Cust.CustomerID = Ord.CustomerID);

--in a complex query, the derived table expressions can become unreadable. so then use common table expressions.

--it is just declaring the expression before hand

with a as(

select

rank() over (partition by customerID order by OrderDate desc) as rn

,\*

from Sales.SalesOrderHeader

)

select \* from a

where rn=1

--find number of orders per customer

select

CustomerID, COUNT(\*) as ordercnt

from Sales.SalesOrderHeader

group by CustomerID

--here as well we have repeat the expression

select

case

when RevisionNumber = 3 then 'ok'

else 'not ok'

end as status

, count(\*) as ordercnt

from Sales.SalesOrderHeader

group by case

when RevisionNumber = 3 then 'ok'

else 'not ok'

end;

with a as (

select

case

when RevisionNumber = 3 then 'ok'

else 'not ok'

end as status

from Sales.SalesOrderHeader

)

select status,count(\*) from a

group by status

**Views**:

Views are saved queries created in a database by administrators and developers

Views are defined with a single SELECT statement

ORDER BY is not permitted in a view definition without the use of TOP, OFFSET/FETCH, or FOR XML

To sort the output, use ORDER BY in the outer query

View creation supports additional options beyond the scope of this class

CREATE VIEW HumanResources.EmployeeList

AS

SELECT BusinessEntityID, JobTitle, HireDate, VacationHours

FROM HumanResources.Employee;

go

SELECT \* FROM HumanResources.EmployeeList

go

Creating simple inline table-valued functions:

Table-valued functions are created by administrators and developers

Create and name function and optional parameters with CREATE FUNCTION

Declare return type as TABLE

Define inline SELECT statement following RETURN

CREATE FUNCTION Sales.fn\_LineTotal (@SalesOrderID INT)

RETURNS TABLE

AS

RETURN

SELECT SalesOrderID,

CAST((OrderQty \* UnitPrice \* (1 - SpecialOfferID))

AS DECIMAL(8, 2)) AS LineTotal

FROM Sales.SalesOrderDetail

WHERE SalesOrderID = @SalesOrderID ;

Writing queries with derived tables:

Derived tables are named query expressions created within an outer SELECT statement

Not stored in database – represents a virtual relational table

When processed, unpacked into query against underlying referenced objects

Allow you to write more modular queries

SELECT <column\_list>

FROM (

<derived\_table\_definition>

) AS <derived\_table\_alias>;

Scope of a derived table is the query in which it is defined

|  |  |
| --- | --- |
| **Derived Tables Must** | **Derived Tables May** |
| * Have an alias * Have names for all columns * Have unique names for all columns * Not use an ORDER BY clause (without TOP or OFFSET/FETCH) * Not be referred to multiple times in the same query | * Use internal or external aliases for columns * Refer to parameters and/or variables * Be nested within other derived tables |

Passing arguments to derived tables:

Derived tables may refer to arguments

Arguments may be:

Variables declared in the same batch as the SELECT statement

Parameters passed into a table-valued function or stored procedure

DECLARE @emp\_id INT = 9;

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

FROM (

SELECT YEAR(orderdate) AS orderyear, custid

FROM Sales.Orders

WHERE empid=@emp\_id

) AS derived\_year

GROUP BY orderyear;

Creating queries with common table expressions:

Use WITH clause to create a CTE:

Define the table expression in WITH clause

Reference the CTE in the outer query

Assign column aliases (inline or external)

Pass arguments if desired

WITH CTE\_year AS

(

SELECT YEAR(OrderDate) AS OrderYear, customerID

FROM Sales.SalesOrderHeader

)

SELECT orderyear, COUNT(DISTINCT CustomerID) AS CustCount

FROM CTE\_year

GROUP BY OrderYear;

Why would you use a view vs a table valued function?? Table valued function is just a view that takes parameters

**SET operators (UNION, INTERSECT, EXCEPT, APPLY)**:

The results of two input queries may be combined, compared, or operated against each other

Both sets must have the same number of compatible columns

ORDER BY not allowed in input queries, but may be used for result of set operation

NULLs considered equal when comparing sets

SET operators include UNION, INTERSECT, EXCEPT, and APPLY

<SELECT query\_1>

<set\_operator>

<SELECT query\_2>

[ORDER BY <sort\_list>]

UNION returns a result set of distinct rows combined from both sides

Duplicates removed during query processing (affects performance)

-- only distinct rows from both queries are returned

SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail

UNION

SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail

UNION ALL returns a result set with all rows from both sets

To avoid performance penalty, use UNION ALL even if you know there are no duplicates

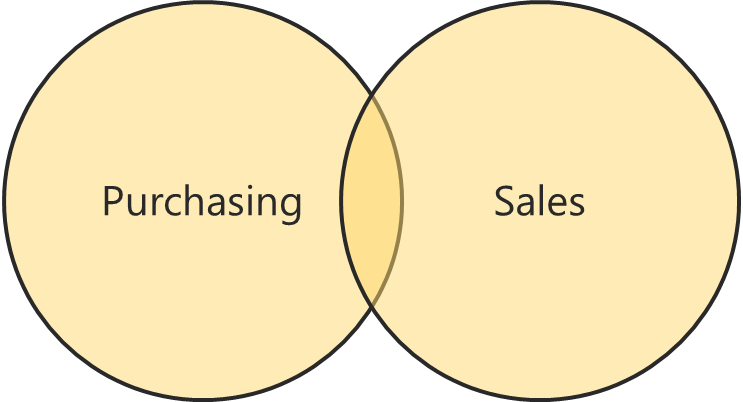
-- all rows from both queries are returned

SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail

UNION ALL

SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail

INTERSECT returns only distinct rows that appear in both result sets



**INTERSECT**

-- only rows that exist in both queries are returned

SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail

INTERSECT

SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail

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

Order in which sets are specified matters

-- only rows from Sales are returned

SELECT ProductID, OrderQty, UnitPrice FROM Sales.SalesOrderDetail

EXCEPT

SELECT ProductID, OrderQty, UnitPrice FROM Purchasing.PurchaseOrderDetail

APPLY is a table operator used in the FROM clause and can be either a CROSS APPLY or OUTER APPLY

Operates on two input tables, left and right

Right table is often a derived table or a table-valued function

OUTER APPLY is similar to LEFT OUTER JOIN between two tables

* 1. OUTER APPLY applies the right table expression to each row in left table
  2. OUTER APPLY adds rows for those with NULL in columns for right table

SELECT <column\_list>

FROM <left\_table> AS <alias>

CROSS/OUTER APPLY

<derived\_table\_expression or inline\_TVF> AS <alias>

--show all customers and last 5 order for each customer..we can use row\_number or apply...lets use apply

--apply would execute once for each row in the outer set

select

c.CustomerID,

c.AccountNumber,

o.\*

from Sales.Customer as c

outer apply(

select top(5) \* from Sales.SalesOrderHeader as sho

where sho.CustomerID = c.CustomerID

order by sho.OrderDate desc

)as o

where c.TerritoryID =3

**SQL windowing**:

Windows extend T-SQL's set-based approach

Windows allow you to specify an order as part of a calculation, without regard to order of input or final output order

Windows allow partitioning and framing of rows to support functions

Window functions can simplify queries that need to find running totals, moving averages, or gaps in data

Partitioning windows:

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

Use PARTITION BY in the OVER() clause

Without a PARTITION BY clause defined, OVER() creates a single partition of all rows

SELECT CustomerID, OrderDate, TotalDue,

SUM(TotalDue) OVER(PARTITION BY CustomerID)

AS TotalDueByCust

FROM Sales.SalesOrderHeader;

**CustomerID OrderDate TotalDue TotalDueByCust**

**---------- -------------------------- -------- --------------**

**11000 2007-08-01 00:00:00.000 3756.989 9115.1341**

**11000 2007-10-01 00:00:00.000 2587.8769 9115.1341**

**11000 2006-09-01 00:00:00.000 2770.2682 9115.1341**

**11001 2007-08-01 00:00:00.000 2674.0227 7054.1875**

**11001 2006-11-01 00:00:00.000 3729.364 7054.1875**

**11001 2007-04-01 00:00:00.000 650.8008 7054.1875**

Defining window functions:

A windows function is a function applied to a window, or set of rows

Window functions include aggregate, ranking, distribution, and offset functions

Window functions depend on set created by OVER()

Windows aggregate functions:

Similar to grouped aggregate functions such as SUM, MIN, MAX, etc.

Applied to windows defined by OVER clause

Support partitioning, ordering, and framing

Window ranking functions:

Ranking functions require a windows order clause

Partitioning is optional

To display results in sorted order still requires ORDER BY!

|  |  |
| --- | --- |
| **Function** | **Description** |
| RANK | Returns the rank of each row within the partition of a result set. May include ties and gaps. |
| DENSE\_RANK | Returns the rank of each row within the partition of a result set. May include ties but will not include gaps. |
| ROW\_NUMBER | Returns a unique sequential row number within partition based on current order. |
| NTILE | Distributes the rows in an ordered partition into a specified number of groups. Returns the number of the group to which the current row belongs. |

Window offset functions:

Window offset functions allow comparisons between rows in a set without the need for a self-join

Offset functions operate on an position relative to the current row, or to the start or end of the window frame

|  |  |
| --- | --- |
| **Function** | **Description** |
| LAG | Returns an expression from a previous row that is a defined offset from the current row. Returns NULL if no row at specified position. |
| LEAD | Returns an expression from a later row that is a defined offset from the current row. Returns NULL if no row at specified position. |
| FIRST\_VALUE | Returns the first value in the current window frame. Requires window ordering to be meaningful. |
| LAST\_VALUE | Returns the last value in the current window frame. Requires window ordering to be meaningful. |

--what is the current balance of the account per a given row..this query did not had good performance before sql2012

select \* from Transactions

where AccountID = 29825

select \*

,SUM(Amount) over (

partition by AccountID

order by TransactionDate, TransactionID

rows between unbounded preceding and current row

) as FinalBalance

from Transactions

where AccountID = 29825

order by AccountID , TransactionDate, TransactionID

--i want to 1 step backwards in the set

select \*

,lag(Amount,1,0) over (partition by AccountID order by TransactionDate, TransactionID)

,SUM(Amount) over (

partition by AccountID

order by TransactionDate, TransactionID

rows between unbounded preceding and current row

) as FinalBalance

from Transactions

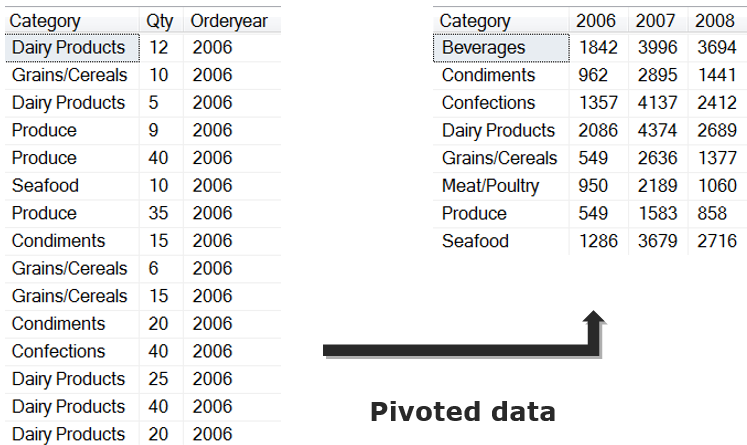
where AccountID = 29825

order by AccountID , TransactionDate, TransactionID

**What is pivoting**?

Pivoting data is rotating data from a rows-based orientation to a columns-based orientation

Distinct values from a single column are projected across as headings for other columns - may include aggregation



Pivoting includes three phases:

1. Grouping determines which element gets a row in the result set
2. Spreading provides the distinct values to be pivoted across
3. Aggregation performs an aggregation function (such as SUM)

SELECT Category, [2006],[2007],[2008]

FROM ( SELECT Category, Qty, Orderyear

FROM Sales.CategoryQtyYear) AS D

PIVOT(SUM(QTY) FOR orderyear

IN([2006],[2007],[2008])

) AS pvt;

SELECT VendorID, [250] AS Emp1, [251] AS Emp2, [256] AS Emp3, [257] AS Emp4, [260] AS Emp5

FROM

(SELECT PurchaseOrderID, EmployeeID, VendorID

FROM Purchasing.PurchaseOrderHeader) p

PIVOT

(

COUNT (PurchaseOrderID)

FOR EmployeeID IN

( [250], [251], [256], [257], [260] )

) AS pvt

ORDER BY pvt.VendorID;

VendorID Emp1 Emp2 Emp3 Emp4 Emp5

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

1492 2 5 4 4 4

1494 2 5 4 5 4

1496 2 4 4 5 5

Note that u generally don’t use pivot in sql as it mainly in used in excel or reporting services. So pivoting is generally done up in the application layer.

CREATE TABLE pvt (VendorID int, Emp1 int, Emp2 int,

Emp3 int, Emp4 int, Emp5 int);

GO

INSERT INTO pvt VALUES (1,4,3,5,4,4);

INSERT INTO pvt VALUES (2,4,1,5,5,5);

INSERT INTO pvt VALUES (3,4,3,5,4,4);

GO

SELECT VendorID, Employee, Orders

FROM

(SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5

FROM pvt) p

UNPIVOT

(Orders FOR Employee IN

(Emp1, Emp2, Emp3, Emp4, Emp5)

)AS unpvt;

GO

VendorID Employee Orders

1 Emp1 4

1 Emp2 3

**Writing queries with grouping sets**:

GROUPING SETS subclause builds on T-SQL GROUP BY clause

Allows multiple groupings to be defined in same query

Alternative to use of UNION ALL to combine multiple outputs (each with different GROUP BY) into one result set

SELECT <column list with aggregate(s)>

FROM <source>

GROUP BY

GROUPING SETS(

(<column\_name>),--one or more columns

(<column\_name>),--one or more columns

() -- empty parentheses if aggregating all rows

);

SELECT TerritoryID, CustomerID, SUM(TotalDue) AS TotalAmountDue

FROM Sales.SalesOrderHeader

GROUP BY

GROUPING SETS((TerritoryID),(CustomerID),());

**TerritoryID CustomerID TotalAmountDue**

**--------------- ----------- --------------**

**NULL 30116 211671.2674**

**NULL 30117 919801.8188**

**NULL 30118 313671.5352**

**NULL NULL 123216786.1159**

**3 NULL 8913299.2473**

**6 NULL 18398929.188**

**9 NULL 11814376.0952**

**1 NULL 18061660.371**

**7 NULL 8119749.346**

--lets say we want to get total sales by customer and territory

SELECT

C.CustomerID

,C.TerritoryID

,SUM(SOD.LineTotal) AS TOTALSALES

FROM Sales.Customer AS C

INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID

INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID

GROUP BY

C.CustomerID

,C.TerritoryID

--WHAT IS THE TOTAL SALES IN TERRITORY INDEPENDENT OF THE CUSTOMER

SELECT

C.CustomerID

,C.TerritoryID

,SUM(SOD.LineTotal) AS TOTALSALES

FROM Sales.Customer AS C

INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID

INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID

GROUP BY

C.CustomerID

,C.TerritoryID

UNION ALL

SELECT

NULL

,NULL

,SUM(SOD.LineTotal) AS TOTALSALES

FROM Sales.Customer AS C

INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID

INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID

ORDER BY CustomerID

--now we are writting a lot of code. instead use grouping sets...EASY AND WE CAN ADD MORE GROUPING SETS EASILY

SELECT

C.CustomerID

,C.TerritoryID

,SUM(SOD.LineTotal) AS TOTALSALES

FROM Sales.Customer AS C

INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID

INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID

GROUP BY GROUPING SETS(

(C.CustomerID,C.TerritoryID)

,(C.TerritoryID)

,()

)

ORDER BY CustomerID, TerritoryID

**CUBE and ROLLUP**:

CUBE provides shortcut for defining grouping sets given a list of columns

All possible combinations of grouping sets are created

SELECT TerritoryID, CustomerID, SUM(TotalDue) AS TotalAmountDue

FROM Sales.SalesOrderHeader

GROUP BY CUBE(TerritoryID, CustomerID)

ORDER BY TerritoryID, CustomerID;

ROLLUP provides shortcut for defining grouping sets, creates combinations assuming input columns form a hierarchy

SELECT TerritoryID, CustomerID, SUM(TotalDue) AS TotalAmountDue

FROM Sales.SalesOrderHeader

GROUP BY ROLLUP(TerritoryID, CustomerID)

ORDER BY TerritoryID, CustomerID;

**DML statements (INSERT, UPDATE, and DELETE)**:

Using INSERT to add data:

The INSERT...VALUES statement inserts a single row by default

INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)

VALUES (N'Square Yards', N'Y2', GETDATE());

GO

Table and row constructors add multi-row capability to INSERT...VALUES

INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)

VALUES

(N'Square Feet', N'F2', GETDATE()),

(N'Square Inches', N'I2', GETDATE());

Using INSERT with DEFAULT constraint:

DEFAULT constraints are used to assign a value to a column when none is specified in the INSERT statement

Defaults are defined in CREATE or ALTER TABLE statement

INSERT can omit columns which have defaults defined

Applies to IDENTITY and NULL too

VALUES clause can use DEFAULT keyword

If no default constraint assigned, NULL inserted

If you added a field called Country with a DEFAULT constraint of ‘USA’ you could INSERT data using the following command

INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate, Country)

VALUES (N'Square Miles', N'M2', GETDATE(), DEFAULT);

Using INSERT with SELECT and EXEC:

INSERT...SELECT is used to insert the result set of a query into an existing table

INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)

SELECT Name, UnitMeasureCode, ModifiedDate

FROM Sales.TempUnitTable

WHERE ModifiedDate < '20080101';

INSERT...EXEC is used to insert the result of a stored procedure or dynamic SQL expression into an existing table

INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)

EXEC Production.Temp\_UOM

@numrows = 5, @catid=1;

Using SELECT INTO:

SELECT...INTO is similar to INSERT...SELECT but SELECT...INTO creates a new table each time the statement is executed

Copies column names, data types, and nullability

Does not copy constraints or indexes

SELECT Name, UnitMeasureCode, ModifiedDate

INTO Production.TempUOMTable

FROM Production.UnitMeasure

WHERE orderdate < '20080101';

Using IDENTITY:

IDENTITY property of a column generates sequential numbers automatically for insertion into a table

Can specify optional set seed and increment values

Only one column in a table may have IDENTITY property defined

IDENTITY column is omitted in INSERT statements

Functions provided to retrieve last generated value

Creates a table with using the IDENTITY property with a starting number of 100 and incremented by 10 as each row is added

CREATE TABLE Production.IdentityProducts(

productid int IDENTITY(100,10) NOT NULL,

productname nvarchar(40) NOT NULL,

categoryid int NOT NULL,

unitprice money NOT NULL)

Using SEQUENCES:

Sequence objects new in SQL Server 2012

Independent objects in database

More flexible than the IDENTITY property

Can be used as default value for a column

Manage with CREATE/ALTER/DROP statements

Retrieve value with the NEXT VALUE FOR clause

-- Define a sequence

CREATE SEQUENCE dbo.InvoiceSeq AS INT START WITH 5 INCREMENT BY 5;

-- Retrieve next available value from sequence

SELECT NEXT VALUE FOR dbo.InvoiceSeq;

Using UPDATE to modify data:

Updates all rows in a table or view

Set can be filtered with a WHERE clause

Set can be defined with a JOIN clause

Only columns specified in the SET clause are modified

Updates the ModifiedDate using a the GETDATE function for the record that has ‘M2’ in the UnitMeasureCode

UPDATE Production.UnitMeasure

SET ModifiedDate = (GETDATE())

WHERE UnitMeasureCode = 'M2'

If no WHERE clause is specified, all records in the Production.UnitMeasure will be updated

Using MERGE to modify data:

MERGE modifies data based on one of the following conditions

When the source matches the target

When the source has no match in the target

When the target has no match in the sourace

MERGE INTO schema\_name.table\_name AS TargetTbl

USING (SELECT <select\_list>) AS SourceTbl

ON (TargetTbl.col1 = SourceTbl.col1)

WHEN MATCHED THEN

UPDATE SET col2 = SourceTbl.col2

WHEN NOT MATCHED THEN

INSERT (<column\_list>)

VALUES (<value\_list>);

--SELECT SOME DATA INTO A NEW TABLE

--SELECT

--ProductID

--,Name

--INTO PRODUCTS

--FROM Production.Product;

--RETURN AS XML

SELECT

\*

FROM PRODUCTS

WHERE NAME LIKE 'A%'

FOR XML PATH

--RETURN AS XML

SELECT

\*

FROM PRODUCTS

WHERE NAME LIKE 'A%'

FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')

--RETURN AS XML...CHANGE NAMES

SELECT

PRODUCTID AS "@id"--WE HAVE TO USE DOUBLE INVERTED COMMAS AS @ HAS SPECIAL MEANING IN SQL...AND @ IS IS USED TO SPECIFY THAT

--WE WANT THIS TO BE A ATTRIBUTE RATHER THAN AN ELEMENT

,NAME AS "@name"

FROM PRODUCTS

WHERE NAME LIKE 'A%'

FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')

--UPDATES USING XML...KEEP THE CASING TO LOWER CASE WHEN USING XML

DECLARE @Xml XML = N'

<PRODUCTS>

<PRODUCT id="1" name="Adjustable Race!" />

<PRODUCT id="879" name="All-Purpose Bike Stand!" />

<PRODUCT id="712" name="AWC Logo Cap!" />

<PRODUCT name="Ramneek!" />

</PRODUCTS>';

SELECT

XT.XC.value('@id','INT') AS PRODUCTID

,XT.XC.value('@name','NVARCHAR(1000)') AS NAME

FROM @Xml.nodes('/PRODUCTS/PRODUCT') AS XT(XC);

WITH SRC AS (

SELECT

XT.XC.value('@id','INT') AS PRODUCTID

,XT.XC.value('@name','NVARCHAR(1000)') AS NAME

FROM @Xml.nodes('/PRODUCTS/PRODUCT') AS XT(XC)

)

--NOW MERGE THIS DATA INTO THE PRODUCTS TABLE

MERGE INTO PRODUCTS AS DEST

USING SRC ON SRC.PRODUCTID = DEST.PRODUCTID

WHEN NOT MATCHED THEN

INSERT (NAME) VALUES(SRC.NAME)

WHEN MATCHED THEN

UPDATE SET NAME = SRC.NAME;

SELECT

PRODUCTID AS "@id"--WE HAVE TO USE DOUBLE INVERTED COMMAS AS @ HAS SPECIAL MEANING IN SQL...AND @ IS IS USED TO SPECIFY THAT

--WE WANT THIS TO BE A ATTRIBUTE RATHER THAN AN ELEMENT

,NAME AS "@name"

FROM PRODUCTS

WHERE NAME LIKE 'A%' OR NAME LIKE 'RaM%'

FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')

--aLSO MERGE FOR DELETES.

DECLARE @Xml XML = N'

<PRODUCTS>

<PRODUCT id="1" name="Adjustable Race!" />

<PRODUCT id="879" name="All-Purpose Bike Stand!" />

<PRODUCT id="712" delete="true" name="AWC Logo Cap!" />

<PRODUCT id="1000" name="Ramneek!" />

</PRODUCTS>';

WITH SRC AS (

SELECT

XT.XC.value('@id','INT') AS PRODUCTID

,XT.XC.value('@name','NVARCHAR(1000)') AS NAME

,ISNULL(XT.XC.value('@delete','BIT'),0) AS DODELETE

FROM @Xml.nodes('/PRODUCTS/PRODUCT') AS XT(XC)

)

--NOW MERGE THIS DATA INTO THE PRODUCTS TABLE

MERGE INTO PRODUCTS AS DEST

USING SRC ON SRC.PRODUCTID = DEST.PRODUCTID

WHEN NOT MATCHED THEN

INSERT (NAME) VALUES(SRC.NAME)

WHEN MATCHED and src.DODELETE=0 THEN

UPDATE SET NAME = SRC.NAME

WHEN MATCHED and src.DODELETE=1 THEN

delete;

SELECT

PRODUCTID AS "@id"--WE HAVE TO USE DOUBLE INVERTED COMMAS AS @ HAS SPECIAL MEANING IN SQL...AND @ IS IS USED TO SPECIFY THAT

--WE WANT THIS TO BE A ATTRIBUTE RATHER THAN AN ELEMENT

,NAME AS "@name"

FROM PRODUCTS

WHERE NAME LIKE 'A%' OR NAME LIKE 'RaM%'

FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')

Using DELETE to remove data:

DELETE operates on a set

Set may be filtered with a WHERE clause

Deletion of each row is logged in database's transaction log

DELETE may be rolled back if statement issued within a user-defined transaction or if an error is encountered

DELETE FROM Production.UnitMeasure

WHERE UnitMeasureCode = 'Y2';

If no WHERE clause is specified, all records in the Production.UnitMeasure will be deleted

Using TRUNCATE TABLE to remove all data:

TRUNCATE TABLE clears the entire table

Storage is physically deallocated, rows not individually removed

Minimally logged

Can be rolled back if TRUNCATE issued within a transaction

TRUNCATE TABLE will fail if the table is referenced by a foreign key constraint in another table

TRUNCATE TABLE Production.UnitMeasure

Note: if you have a lot of rows, then doing a ‘delete from’ without a where clause will take a long time. Then use the ‘truncate’ option

**PRIMARY KEY constraint**:

A ***PRIMARY KEY*** is an important concept of designing a database table as it provides an attribute or set of attributes used to uniquely identify each row in the table

A table can only have one primary key which is created using a primary key constraint and enforced by creating a unique index on the primary key columns

A column that participates in the primary key constraint cannot accept null values

To add a PRIMARY KEY constraint to an existing table use the following command

ALTER TABLE Production.TransactionHistoryArchive

ADD CONSTRAINT PK\_TransactionHistoryArchive\_TransactionID

PRIMARY KEY CLUSTERED (TransactionID);

**FOREIGN KEY constraint**:

A ***FOREIGN KEY*** is a column or combination of columns that are used to establish a link between data in two tables. The columns used to create the primary key in one table are also used to create the foreign key constraint and can be used to reference data in the same table or in another table

A foreign key does not have to reference a primary key, it can be defined to reference a unique constraint in either the same table or in another table

To add a FOREIGN KEY constraint to an existing table use the following command

ALTER TABLE Sales.SalesOrderHeaderSalesReason

ADD CONSTRAINT FK\_SalesReason

FOREIGN KEY (SalesReasonID)

REFERENCES Sales.SalesReason (SalesReasonID)

ON DELETE CASCADE

ON UPDATE CASCADE ;

**UNIQUE constraints**:

A ***UNIQUE constraint*** is created to ensure no duplicate values are entered in specific columns that do not participate in a primary key

Creating a UNIQUE constraint automatically creates a corresponding unique index

To create a UNIQUE constraint while creating a table use the following command

CREATE TABLE Production.TransactionHistoryArchive4

(TransactionID int NOT NULL,

CONSTRAINT AK\_TransactionID UNIQUE(TransactionID) );

**CHECK constraints**:

A ***CHECK constraint*** is created in a table to specify the data values that are acceptable in one or more columns

To create a CHECK constraint after creating a table use the following command

ALTER TABLE DBO.NewTable

ADD ZipCode int NULL

CONSTRAINT CHK\_ZipCode

CHECK (ZipCode LIKE '[0-9][0-9][0-9][0-9][0-9]‘);

**DEFAULT constraints**:

A ***DEFAUT constraint*** is a special case of a column default that is applied when an INSERT statement doesn't explicitly assign a particular value. In other words, the column default is what the column will get as a value by default

To create a DEFAULT constraint on an existing table use the following command

ALTER TABLE Sales.CountryRegionCurrency

ADD CONSTRAINT Default\_Country

DEFAULT ‘USA’ FOR CountryRegionCode

**DML triggers**:

A ***DML*** ***trigger*** is a special type of stored procedure that automatically fires when any valid DML event takes place regardless of whether or not any rows are affected

DML triggers can include complex T-SQL statements used to enforce business rules and provide data integrity when an INSERT, UPDATE, or DELETE command is executed

The following trigger will prints a message to the client when anyone tries to add or change data in the Customer table

CREATE TRIGGER reminder1 ON Sales.Customer

AFTER INSERT, UPDATE

AS RAISERROR ('Notify Customer Relations', 16, 10);

Note: we might want to use the new throw statement instead of the raiseerror used here

Note: if the rules are too complex to be enforced by constraints, then use trigger

Note: other use could be to update data in some denormalized tables. But if performance is hampered, then you might want to use some sort of background job instead of a trigger to update the denormalized tables.

**OUTPUT clause**:

The ***OUTPUT*** clauseis used to return information from, or expressions based on, each row affected by an INSERT, UPDATE, DELETE, or MERGE statement. These results can be returned to the processing application for use in such things as confirmation messages or archiving

The following example deletes all rows in the ShoppingCartItem table. The clause OUTPUT **deleted.\*** specifies that all columns in the deleted rows, be returned to the calling application which in this case was the Query Editor

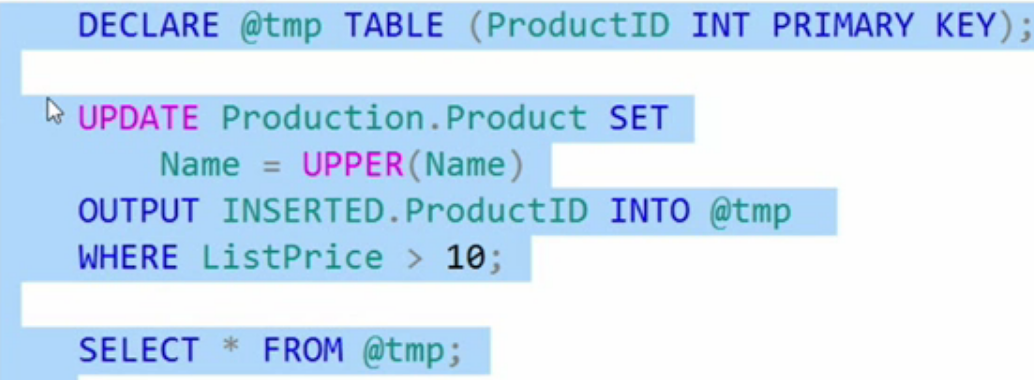
DELETE Sales.ShoppingCartItem OUTPUT DELETED.\* WHERE ShoppingCartID = 20621;

--Verify the rows in the table matching the WHERE clause have been deleted.

SELECT COUNT(\*) AS [Rows in Table]

FROM Sales.ShoppingCartItem

WHERE ShoppingCartID = 20621;



**T-SQL batches**:

T-SQL batches are collections of one or more T-SQL statements sent to SQL Server as a unit for parsing, optimization, and execution and are terminated with the GO clause

Batches are also boundaries for the scope of a variable

Some statements (e.g., CREATE FUNCTION, CREATE PROCEDURE, CREATE VIEW) may not be combined with others in the same batch

CREATE VIEW HumanResources.EmployeeList

AS

SELECT BusinessEntityID, JobTitle, HireDate, VacationHours

FROM HumanResources.Employee;

GO

Batches are parsed for syntax as a unit

Syntax errors cause the entire batch to be rejected

Runtime errors may allow the batch to continue after failure, by default

Batches can contain error-handling code

--Valid batch

INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)

VALUES (N'Square Footage', N‘F4', GETDATE()),

(N'Square Inches', N‘I2', GETDATE());

GO

--Invalid batch

INSERT INTO dbo.t1 VALUE(1,2,N'abc');

INSERT INTO dbo.t1 VALUES(2,3,N'def');

GO

T-SQL Variables:

Variables are objects that allow storage of a value for use later in the same batch

Variables are defined with the DECLARE keyword and begin with @

Beginning with SQL Server 2008, variables can be declared and initialized in the same statement

Variables are always local to the batch in which they're declared and go out of scope when the batch ends

--Declare,initialize, and use a variable

DECLARE @SalesPerson\_id INT = 5;

SELECT OrderYear, COUNT(DISTINCT CustomerID) AS CustCount

FROM (

SELECT YEAR(OrderDate) AS OrderYear, CustomerID

FROM Sales.SalesOrderHeader

WHERE SalesPersonID = @SalesPerson\_id

) AS DerivedYear

GROUP BY OrderYear;

Working with Variables:

Values can be assigned with a SET command or a SELECT statement

SET can only assign one variable at a time. SELECT can assign multiple variables at a time

When using SELECT to assign a value, make sure that exactly one row is returned by the query

--Declare and initialize variables

DECLARE @numrows INT = 3, @catid INT = 2;

--Use variables to pass parameters to procedure

EXEC Production.ProdsByCategory

@numrows = @numrows, @catid = @catid;

GO

Working with Synonyms:

A synonym is an alias or link to an object stored either on the same SQL Server instance or on a linked server

Synonyms can point to tables, views, procedures, and functions

Synonyms can be used for referencing remote objects as though they were located locally, or for providing alternative names to other local objects

Use the CREATE, ALTER, and DROP commands to manage synonyms

-- Create a synonym for the Product table in AdventureWorks CREATE SYNONYM dbo.MyProduct

FOR AdventureWorks.Production.Product;

GO

-- Query the Product table by using the synonym.

SELECT ProductID, Name

FROM MyProduct

WHERE ProductID < 5;

GO

**T-SQL control-of-flow language**:

SQL Server provides additional language elements that control the flow and execution of T-SQL statements in batches, stored procedures, and multi-statement functions

Control-of-flow elements allow you to specify statements need to be performed in a specified order or not at all

The default is for statements to execute sequentially, however you can use IF…ELSE, BEGIN…END, WHILE, RETURN, and others to control the flow of your batch files or stored procedures

IF OBJECT\_ID ('Production.Product', 'U') IS NOT NULL

PRINT 'I am here and contain data, so don’t delete me'

**IF…ELSE**:

IF…ELSE uses a predicate to determine the flow of the code

The code in the IF block is executed if the predicate evaluates to TRUE

The code in the ELSE block is executed if the predicate evaluates to FALSE or UNKNOWN

Very useful when combined with the EXISTS operator

IF OBJECT\_ID ('Production.Product', 'U') IS NOT NULL

PRINT 'I am here and contain data, so don’t delete me'

ELSE

PRINT 'Table not found, so feel free to create one'

GO

**WHILE**:

WHILE enables statements to execute in the WHILE block as long as the predicate evaluates to TRUE and doesn’t stop executing until the predicate evaluates to FALSE or UNKNOWN

Execution can be altered by BREAK or CONTINUE

DECLARE @BusinessEntID AS INT = 1, @Title AS NVARCHAR(50);

WHILE @BusinessEntID <=10

BEGIN

SELECT @Title = JobTitle FROM HumanResources.Employee

WHERE BusinessEntityID = @BusinessEntID;

PRINT @Title;

SET @BusinessEntID += 1;

END;

**Structured exception handling**:

Structured exception handling allows a centralized response to runtime errors

TRY to run a block of commands and CATCH any errors

Execution moves to the CATCH block of commands when an error occurs

No need to check every statement to see if an error occurred

If error you decide whether the transaction should be rolled back, errors logged, etc.

Not all errors can be caught by TRY / CATCH:

Syntax or compile errors

Some name resolution errors

**Querying the ERROR object**:

Common ERROR object properties and ERROR object functions:

|  |  |  |
| --- | --- | --- |
| **Property** | **Function to Query** | **Description** |
| Number | ERROR\_NUMBER | Unique number assigned to  the error |
| Message | ERROR\_MESSAGE | Error message text |
| Severity | ERROR\_SEVERITY | Severity class (1-25) |
| Procedure Name | ERROR\_PROCEDURE | Name of the procedure or  trigger that raised the error |
| Line Number | ERROR\_LINE | Number of the line that  raised the error in the batch, procedure, trigger, or function |

Values returned correspond to sys.messages view

TRY and CATCH blocks:

TRY block defined by BEGIN TRY...END TRY statements

Place all code that might raise an error between them

No code may be placed between END TRY and BEGIN CATCH

TRY and CATCH blocks may be nested

CATCH block defined by BEGIN CATCH...END CATCH

Execution moves to the CATCH block when catchable errors occur within the TRY block

BEGIN TRY

-- Generate a divide-by-zero error.

SELECT 1/0;

END TRY

BEGIN CATCH

SELECT

ERROR\_NUMBER() AS ErrorNumber

,ERROR\_SEVERITY() AS ErrorSeverity

,ERROR\_STATE() AS ErrorState

,ERROR\_PROCEDURE() AS ErrorProcedure

,ERROR\_LINE() AS ErrorLine

,ERROR\_MESSAGE() AS ErrorMessage;

END CATCH;

GO

ErrorNumber ErrorSeverity ErrorState ErrorProcedure ErrorLine ErrorMessage

8134 16 1 NULL 3 Divide by zero error encountered.

Not all errors can be caught by TRY / CATCH:

Syntax, compilation errors, and name resolution errors

BEGIN TRY

-- Table does not exist; object name resolution

-- error not caught.

SELECT \* FROM IDontExist;

END TRY

BEGIN CATCH

SELECT

ERROR\_NUMBER() AS ErrorNumber

,ERROR\_MESSAGE() AS ErrorMessage;

END CATCH

Invalid object name 'IDontExist'.

THROW statement:

SQL Server 2012 provides the new THROW statement

Successor to the RAISERROR statement

Does not require defining errors in the sys.messages table

THROW allows choices when handling errors:

Handle specific errors in the local CATCH block

Pass errors to another process

Use THROW:

With parameters to pass a user-defined error

Without parameters to re-raise the original error (must be within a CATCH block)

BEGIN TRY

SELECT 100/0 AS 'Problem';

END TRY

BEGIN CATCH

PRINT 'Code inside CATCH is beginning'

PRINT 'MyError: ' + CAST(ERROR\_NUMBER()

AS VARCHAR(255));

THROW;

END CATCH

(0 row(s) affected)

Code inside CATCH is beginning

MyError: 8134

Msg 8134, Level 16, State 1, Line 6

Divide by zero error encountered.

**Transactions**:

A transaction is a group of tasks defined as a unit of work that must succeed or fail together – no partial completion is permitted

--Two tasks that make up a unit of work INSERT INTO Sales.SalesOrderHeader...

INSERT INTO Sales.SalesOrderHeader...

INSERT INTO Sales.SalesOrderDetail...

Individual data modification statements are automatically treated as standalone transactions

User transactions can be managed with T-SQL commands:

BEGIN/ COMMIT/ROLLBACK TRANSACTION

SQL Server uses locking mechanisms and the transaction log to support transactions

The need for transactions: issues with batches:

Some runtime errors during a batch may result in unacceptable partial success:

Part of the batch succeeds and part fails, leaving behind the results from the part of the batch that succeeded

Simple error handling within a batch cannot repair partial success

--Batch without transaction management

BEGIN TRY

INSERT INTO Sales.SalesOrderHeader... --Insert succeeds

INSERT INTO Sales.SalesOrderDetail... --Insert fails

END TRY

BEGIN CATCH

--First row still in Sales.SalesOrderHeader Table

SELECT ERROR\_NUMBER()

...

END CATCH;

Transactions extend batches:

Transaction commands identify blocks of code that must succeed or fail together and provide points where database engine can roll back, or undo, operations:

BEGIN TRY

BEGIN TRANSACTION

INSERT INTO Sales.SalesOrderHeader... --Succeeds

INSERT INTO Sales.SalesOrderDetail... --Fails

COMMIT TRANSACTION -- If no errors, transaction completes

END TRY

BEGIN CATCH

--Inserted rows still exist in Sales.SalesOrderHeader SELECT ERROR\_NUMBER()

ROLLBACK TRANSACTION --Any transaction work undone

END CATCH;

BEGIN TRANSACTION marks the starting point of an explicit, user-defined transaction

Transactions last until a COMMIT statement is issued, a ROLLBACK is manually issued, or the connection is broken and the system issues a ROLLBACK

Transactions are local to a connection and cannot span connections

In your T-SQL code: Mark the start of the transaction's work

BEGIN TRY

BEGIN TRANSACTION -- marks beginning of transaction

INSERT INTO Sales.SalesOrderHeader... –-Completed

INSERT INTO Sales.SalesOrderDetail... –-Completed

...

COMMIT ensures all of the transaction's modifications are made a permanent part of the database

COMMIT frees resources, such as locks, used by the transaction

In your T-SQL code: If a transaction is successful, commit it

BEGIN TRY

BEGIN TRAN -- marks beginning of transaction

INSERT INTO Sales.SalesOrderHeader...

INSERT INTO Sales.SalesOrderDetail...

COMMIT TRAN -- mark the transaction as complete

END TRY

A ROLLBACK statement undoes all modifications made in the transaction by reverting the data to the state it was in at the beginning of the transaction

ROLLBACK frees resources, such as locks, held by the transaction

Before rolling back, you can test the state of the transaction with the XACT\_STATE function

In your T-SQL code: If an error occurs, ROLLBACK to the point of the BEGIN TRANSACTION statement

BEGIN CATCH

SELECT ERROR\_NUMBER() --sample error handling

ROLLBACK TRAN

END CATCH;

SQL Server does not automatically roll back transactions when errors occur

To roll back, either use ROLLBACK statements in error-handling logic or enable XACT\_ABORT

XACT\_ABORT specifies whether SQL Server automatically rolls back the current transaction when a runtime error occurs

When SET XACT\_ABORT is ON, the entire transaction is terminated and rolled back on error, unless occurring in TRY block

SET XACT\_ABORT OFF is the default setting

Change XACT\_ABORT value with the SET command:

SET XACT\_ABORT ON;

**System catalog views**:

Built-in views that provide information about the system catalog

Use standard query methods to return metadata

Column lists, JOIN, WHERE, ORDER BY

Some views are filtered to display only user objects, some views include system objects

--Pre-filtered to exclude system objects

SELECT name, object\_id, schema\_id, type, type\_desc

FROM sys.tables;

--Includes system and user objects

SELECT name, object\_id, schema\_id, type, type\_desc

FROM sys.objects;

Information schema views:

Views stored in the INFORMATION\_SCHEMA system schema

Return system metadata per ISO standard, used by third-party tools

Maps standard names (catalog, domain) to SQL Server names (database, user-defined data type)

SELECT TABLE\_CATALOG, TABLE\_SCHEMA,

TABLE\_NAME, TABLE\_TYPE

FROM INFORMATION\_SCHEMA.TABLES;

SELECT VIEW\_CATALOG, VIEW\_SCHEMA, VIEW\_NAME, TABLE\_CATALOG, TABLE\_SCHEMA, TABLE\_NAME, COLUMN\_NAME

FROM INFORMATION\_SCHEMA.VIEW\_COLUMN\_USAGE;

SELECT VIEW\_CATALOG, VIEW\_SCHEMA, VIEW\_NAME, TABLE\_CATALOG, TABLE\_SCHEMA, TABLE\_NAME, COLUMN\_NAME

FROM INFORMATION\_SCHEMA.VIEW\_COLUMN\_USAGE

WHERE COLUMN\_NAME = 'BusinessEntityID'

**System metadata functions**:

Return information about settings, values, and objects in SQL Server

Come in a variety of formats

Some marked with a @@ prefix, sometimes incorrectly referred to as global variables: @@VERSION

Some marked with a () suffix, similar to arithmetic or string functions: ERROR\_NUMBER()

Some special functions marked with a $ prefix: $PARTITION

Queried with a standard SELECT statement:

SELECT @@VERSION AS SQL\_Version;

SELECT SERVERPROPERTY('ProductVersion') AS version;

SELECT SERVERPROPERTY('Collation') AS collation;

Querying DMVs and functions:

Dynamic management views are queried like standard views:

SELECT session\_id, login\_time, program\_name

FROM sys.dm\_exec\_sessions

WHERE is\_user\_process = 1;

Dynamic management functions are queried as table-valued functions, including parameters:

SELECT referencing\_schema\_name, referencing\_entity\_name, referencing\_class\_desc

FROM sys.dm\_sql\_referencing\_entities(

'Sales.SalesOrderHeader', 'OBJECT');

GO

About dynamic management objects:

The nearly 200 dynamic management views (DMVs) and functions return server state information

DMVs include catalog information as well as administrative status information, such as object dependencies

DMVs are server-scoped (instance-level) or database-scoped

Requires VIEW SERVER STATE or VIEW DATABASE STATE permission to query DMVs

Underlying structures change over time, so avoid writing SELECT \* queries against DMVs

Categories include;

|  |  |
| --- | --- |
| **Naming pattern** | **Description** |
| db | Database-related information |
| exec | Query execution-related information |
| io | I/O statistics |
| os | SQL Server Operating System (SQLOS) information |
| tran | Transaction-related information |

**Stored procedures**:

Use the EXECUTE or EXEC command before the name of the stored procedure

Pass parameters by position or name, separated by commas when applicable

--no parameters so lists all database

EXEC sys.sp\_databases;

--single parameter of name of table

EXEC sys.sp\_help N'Sales.Customer';

--multiple named parameters

EXEC sys.sp\_tables

@table\_name = '%',

@table\_owner = N'Sales';

Common system stored procedures:

Database engine procedures can provide general metadata

sp\_help, sp\_helplanguage

sp\_who, sp\_lock

Catalog procedures can be used as an alternative to system catalog views and functions:

|  |  |
| --- | --- |
| **Name** | **Description** |
| sp\_databases | Lists databases in an instance of SQL Server |
| sp\_tables | Returns a list of tables or views, except synonyms |
| sp\_columns | Returns column information for the specified objects |

Unlike system views, there is no option to select which columns to return

Executing system stored procedures:

System stored procedures:

Marked with an sp\_ prefix

Stored in a hidden resource database

Logically appear in the sys schema of every user and system database

Best practices for execution include:

Always use EXEC or EXECUTE rather than just calling by name

Include the sys schema name when executing

Name each parameter and specify its appropriate data type

--This example uses EXEC, includes the sys schema name, --and passes the table name as a named Unicode parameter --to a procedure accepting an NVARCHAR(776)

--input parameter.

EXEC sys.sp\_help @objname = N'Sales.Customer';

Creating procedures that return rows:

Stored procedures can be wrappers for simple or complex SELECT statements

Procedures may include input and output parameters as well as return values

Use CREATE PROCEDURE statement:

CREATE PROCEDURE <schema\_name.proc\_name>

(<parameter\_list)

AS

SELECT <body of SELECT statement>;

Change procedure with ALTER PROCEDURE statement

No need to drop, recreate

Creating procedures that accept parameters:

Input parameters passed to procedure logically behave like local variables within procedure code

Assign name with @prefix, data type in procedure header

Refer to parameter in body of procedure

CREATE PROCEDURE Production.ProdsByProductLine

(@numrows AS int, @ProdLine AS nchar)

AS

SELECT TOP(@numrows) ProductID,

Name, ListPrice

FROM Production.Product

WHERE ProductLine = @ProdLine;

--Retrieve top 50 products with product line = M

EXEC Production.ProdsByProductLine 50, 'M'

NOTE: SET NOCOUNT ON

Nocount on so that after each statement we don’t return number of rows affected back to the client. Reduces the data sent back to the client.

**Writing well-performing queries**:

Only retrieve what you need

In the SELECT clause, only use needed columns – avoid \*

Use a WHERE clause, filter to return only needed rows

Improve search performance of WHERE clause

Avoid expressions that manipulate columns in the predicate

Minimize use of temporary tables or table variables

Use windowing functions or other set-based operations when possible

Avoid cursors and other iterative approaches

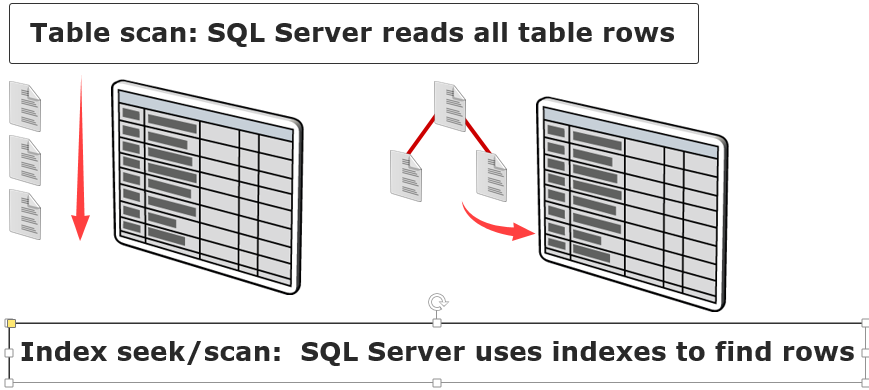
Work with your DBA to arrange good indexes to support filters, joins, and ordering

Learn how to address tasks with different query approaches to compare performance

Indexing in SQL Server:

SQL Server accesses data by using indexes or by scanning all rows in a table

Indexes also supports ordering operations such as grouping, joining, and ORDER BY clauses



SQL Server indexes: performance considerations:

Check query execution plans to see if indexes are present and being used as expected

For query writers who are not DBAs or database developers, the ability to recognize problems with indexes, such as the use of table scans when you expect an index to be used, can be very helpful in tuning an application

Note: while indexes improve the searching of data, there is a slight hit while inserting data.

Distribution statistics:

Distribution statistics describe the distribution and the uniqueness, or selectivity, of data

Statistics, by default, are created and updated automatically

Statistics are used by the query optimizer to estimate the selectivity of data, including the size of the results

Large variances between estimated and actual values might indicate a problem with the estimates, which may be addressed through updating statistics

Avoiding cursors:

Cursors contradictthe relational model, which operates on sets

Cursors typically require more code than set-based approach

Cursors typically incur more overhead during execution than a comparable set-based operation

Alternatives to cursors:

Windowing functions

Aggregate functions

Appropriate uses for cursors:

Generating dynamic SQL code

Performing administrative tasks

**What is an execution plan**?

Review of the process of executing a query:

Parse, resolve, optimize, execute

An execution plan includes information on which tables to access, which indexes, what joins to perform

If statistics exist for a relevant column or index, then the optimizer will use them in its calculations

SQL Server tools provide access to execution plans to show how a query was executed or how it would be executed

Plans available in text format (deprecated), XML format, and graphical renderings of XML

Plan viewer accessible in results pane of SSMS

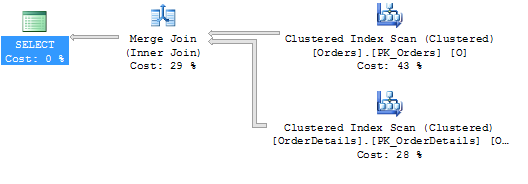
**Actual and estimated execution plans**:

Execution plans graphically represent the methods that SQL Server uses to execute the statements in a T-SQL query

SSMS provides access to two forms of execution plans:

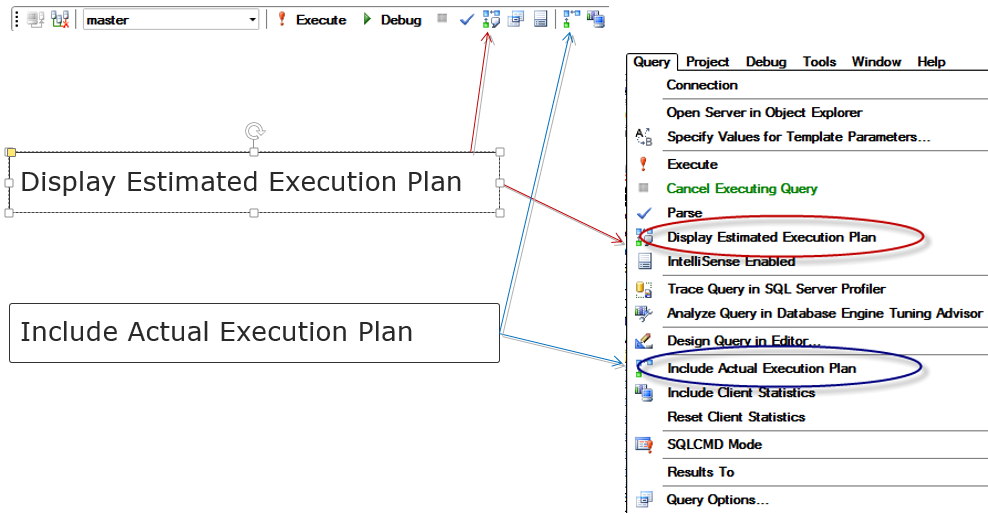
**Estimated** execution plans do not execute the query. Instead, they display the plan that SQL Server would likely use if the query were run.

**Actual** execution plans are returned the next time the query is executed. They display the plan that was actually used by SQL Server



Viewing graphical execution plans:

Enable execution plan viewers in SSMS



Interpreting the execution plan:

Read the plan right to left, top to bottom

Hover the mouse pointer over an item to see additional information

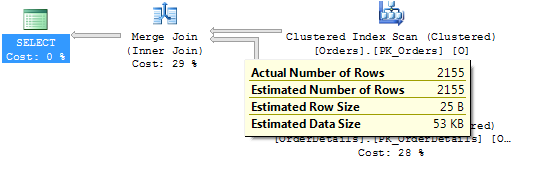
Percentages indicate cost of operator relative to total query

Thickness of lines between operators indicates relative number of rows passing through

For issues, look for thick lines leading into high-cost operators

In an actual execution plan, note any differences between estimated and actual values

Large variances may indicate problems with estimates



Displaying Query Statistics:

SQL Server provides detailed runtime information about the execution of a query

STATISTICS TIME will show time spent parsing and compiling a query

SET STATISTICS TIME ON;

STATISTICS IO will show amount of disk activity generated by a query

SET STATISTICS IO ON;

**Query performance**: