 SQL coding standards DB-9049 V2.

1. Capitalize reserved words

Correct: CREATE PROCEDURE [dbo].[ProcedureName\_sp] (@Date DATE=NULL, @Group CHAR(3)=NULL)

Incorrect: create procedure [dbo].[PROCEDURENAME\_sp] (@date date=null, @group char(3)=null)

1. Objects name (tables, fields, procedures …) Capitalization Style: Pascal case. No spaces inside objects name.

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters.

For example BackColor:

1. Use suffix or prefix for object names. Mostly from the type column in sys.objects or from BOL:

CHECK constraint

CONSTRAINT CHK\_ColumnNameMinValue CHECK (ColumnName > 10)

DEFAULT constraint. The default value must be the same type as the column—if a column is declared a DECIMAL do not provide an INTEGER default value.

CONSTRAINT DF\_ColumnName DEFAULT ('bicycle')

FOREIGN KEY constraint

CONSTRAINT FK\_TableName\_ColumnName FOREIGN KEY (ColumnName) REFERENCES Person.Person (BusinessEntityID);

UDF = SQL scalar function, SQL inline table-valued function and table-valued-function

CREATE FUNCTION dbo.FunctionName\_UDF (@Group CHAR(3))  
  
SP = SQL Stored Procedure

CREATE PROCEDURE dbo.Get10TopResellers\_SP @Group CHAR(3)

PK = PRIMARY KEY constraint

CONSTRAINT PK\_TableName\_TransactionID PRIMARY KEY CLUSTERED (TransactionID);

SN = Synonym

CREATE SYNONYM dbo.MyProduct\_SN FOR AdventureWorks2012.Production.Product;

SO = Sequence object

CREATE SEQUENCE dbo.CountBy1\_SO

START WITH 1

INCREMENT BY 1;

TT = Table type

CREATE TYPE dbo.LocationTableType\_TT AS TABLE

( LocationName VARCHAR(50)

, CostRate INT );

Table (user-defined)

CREATE TABLE dbo.CamelCase\_TABLE

AK = UNIQUE constraint (Alternate key)

ALTER TABLE Person.Password

ADD CONSTRAINT AK\_Password UNIQUE (PasswordHash, PasswordSalt);   
  
V = View

CREATE VIEW dbo.FbaMgr\_V

AS

SELECT …

|  |  |
| --- | --- |
| Object type | suffix or prefix |
| CHECK constraint | CONSTRAINT CHK\_ColumnNameMinValue CHECK (ColumnName > 10) |
| DEFAULT constraint | CONSTRAINT DF\_ColumnName DEFAULT ('bicycle') |
| FOREIGN KEY constraint | CONSTRAINT FK\_TableName\_ColumnName FOREIGN KEY (ColumnName) … |
| SQL function | CREATE FUNCTION dbo.FunctionName\_UDF (@Group CHAR(3)) |
| SQL Stored Procedure | CREATE PROCEDURE dbo.Get10TopResellers\_SP @Group CHAR(3) |
| PRIMARY KEY constraint | CONSTRAINT PK\_TableName\_TransactionID PRIMARY KEY … |
| Synonym | CREATE SYNONYM dbo.MyProduct\_SN FOR TableName; |
| Table type | CREATE TYPE dbo.LocationTableType\_TT AS TABLE |
| Table (BetaNDM) | CREATE TABLE dbo.CamelCase\_TABLE |
| Table | CREATE TABLE dbo.CamelCase |
| UNIQUE constraint | ALTER TABLE TableName ADD CONSTRAINT AK\_ColumnName UNIQUE |
| View | CREATE VIEW dbo.FbaMgr\_V |

4. Don’t create name for temporary table constraints, always create name for permanent table constraints.

CREATE TABLE #TempTable (

ID INT NOT NULL,

TradeDate DATE NOT NULL DEFAULT GETDATE(),

TradeType CHAR(1) CHECK (TradeType != 'N'),

SomeID INT UNIQUE,

PRIMARY KEY (ID, TradeDate));

GO

DROP TABLE #TempTable;

GO

CREATE TABLE dbo.PermTable\_TABLE (

ID INT NOT NULL,

TradeDate DATE NOT NULL CONSTRAINT DF\_TradeDate DEFAULT GETDATE(),

TradeType CHAR(1) CONSTRAINT CHK\_TradeType CHECK (TradeType != 'N'),

SomeID INT ,

CONSTRAINT AK\_SomeID UNIQUE (SomeID),

CONSTRAINT PK\_PermTable\_ID\_TradeDate PRIMARY KEY (ID, TradeDate));

GO

DROP TABLE dbo.PermTable\_TABLE;

5. Table names singular

6. Use IF EXISTS and PRINT statements when create/alter objects

IF EXISTS(SELECT \* FROM sys.table\_types s WHERE s.name = 'Rep\_TT')

BEGIN

PRINT 'table-valued parameter type dbo.Rep\_TT already exists';

END

ELSE

BEGIN

PRINT 'create table-valued parameter type dbo.Rep\_TT';

CREATE TYPE dbo.Rep\_TT AS TABLE

(

Rep CHAR(4) NOT NULL PRIMARY KEY

);

END;

GO

USE ALTER instead of CREATE statement

IF OBJECT\_ID('dbo.ReportPensacolaPositionHoldings\_sp') IS NULL

BEGIN

PRINT 'PROCEDURE dbo.ReportPensacolaPositionHoldings\_sp not found';

EXEC ('CREATE PROCEDURE dbo.ReportPensacolaPositionHoldings\_sp AS RETURN 0;')

END

ELSE

BEGIN

PRINT 'ALTER PROCEDURE dbo.ReportPensacolaPositionHoldings\_sp';

END;

IF OBJECT\_ID('dbo.myView\_V') IS NULL

BEGIN

PRINT 'VIEW dbo.myView\_V not found';

EXEC ('CREATE VIEW dbo.myView\_V AS SELECT 1 AS Clmn;')

END;

7. Do not use reserved words, if you can’t avoid the use square brackets like: [event]

8. Development programming style vs. production programming style.

This is an example of development code with WHERE 1=1

(It makes it easier to comment out stuff during development of queries):

SELECT

A

-- ,B

,C

,D

-- ,E

FROM TABLE

WHERE 1=1

-- and B='This'

and C='That'

-- and D is not null

This is another example of development programming style with SELECT \* :

SELECT \*

FROM TABLE

WHERE B = 'This'

--AND C = 'That'

AND D IS NOT NULL

This is how it should looks in Source Control, QA and Production:

SELECT

A

,B

,C

,D

FROM dbo.TABLE

WHERE B = 'This'

AND C = 'That'

AND D IS NOT NULL;

Do not use SELECT \* in your queries. Always write the required Column names after the SELECT statement.

9. Always define the columns names in the insert statement like below

INSERT INTO LMS\_ERROR\_LOG ( PROSPECT\_ID,METHOD\_NAME, EXCEPTION,DATE\_TIME )

VALUES (@ProspectId,@pMethodName,@pErrorMsg,getdate() );

10. Triggers: Rules: TR\_<TableName>\_<action>

Examples: TR\_Orders\_Upd

11. Indexes: Rules: IX\_<TableName>\_<columns separated by \_>

Examples: IX\_Products\_ProductID

If this is a cover index IX\_<TableName>\_<first column name>\_cover

Max column name – 20 columns.

12. Foreign Keys: Rules: FK\_<TableName1>\_<TableName2>

Example: FK\_Products\_Orderss

13. Each table must have a primary key. At least it should be an IDENTITY column named ID.

14. Use parentheses to increase readability

WHERE (color=’red’ AND (size = 1 OR size = 2))

15. Avoid using <> as a comparison operator if it possible. Note maximum IN input values is 15 (the extra value in the IN caused late filtering in execution plan).

Use ID IN(1,3,4,5) instead of ID <> 2

If you have more than 15 values use INNER JOIN:

DECLARE @in TABLE (city nvarchar(60));

INSERT INTO @in (city)

VALUES ('Boulogne-Billancourt'),('Boulogne-sur-Mer'),('Bountiful'),

('Bracknell'),('Bradenton'),('Braintree'),('Brampton'),

('Branch'),('Branson'),('Braunschweig'),('Bremerton'),

('Brisbane'),('Brossard'),('Burbank'),('Burien'),

('Burlingame'),('Burnaby'),('Bury'),('Byron'),('Calgary'),

('Caloundra'),('Camarillo'),('Cambridge'),('Campbellsville'),

('Canoga Park'),('Carnation'),('Carol Stream'),('Carrollton'),

('Carson'),('Casper'),('Cedar City'),('Cedar Park'),

('Central Valley'),('Cergy'),('Cerritos'),('Chalk Riber'),

('Chandler'),('Chantilly'),('Charlotte'),('Chatou'),

('Cheektowaga'),('Chehalis'),('Cheltenham'),('Cheyenne'),

('Chicago'),('Chula Vista'),('Cincinnati'),('Citrus Heights'),

('City Of Commerce'),('Clackamas'),('Clarkston'),('Clay'),

('Clearwater'),('Cliffside'),('Cloverdale'),('Coffs Harbour'),

('College Station'),('Colma'),('Colombes'),('Colomiers'),

('Columbus'),('Concord'),('Coronado'),('Corpus Christi'),('Seattle');

SELECT DISTINCT a.city

FROM Person.Address a

INNER JOIN @in i on (i.city = a.City);

16. Use SET NOCOUNT ON; at the beginning of stored procedures.

17. Schema-qualified object names.

Always use schema-qualified object names (Sales.Orders as opposed to just Orders) both when creating and when querying them. Different users can have different default schemas. Also, even though the savings is very minor and probably negligible, explicitly indicating a schema means you save the cost of the instructions involved in implicit resolution. Furthermore, a query plan produced by a user with one default schema can't be reused by a user with a different default schema. You'll get multiple optimizations with multiple unnecessary execution plans in cache.

SELECT

tbl1.Field01,

tbl2.Field1,

tbl1.Field02,

tbl2.Field2

FROM dbo.Table1 tb1

JOIN dbo.Table2 tbl2 ON tbl1.OrderNum = tbl2.OrderNum;

18. Place all DECLARE statements before any other code in the procedure.

19. Check the global variable @@ERROR immediately after executing a data manipulation statement (like INSERT/UPDATE/DELETE), so that you can rollback the transaction if an error occurs

Or use TRY/CATCH

20. Stored procedure should always return value. The RETURN statement is meant for returning the execution status only, but not data.

RETURN @@ERROR;

21. When a result set is not needed, use syntax that does not return a result set – output parameters.

22. Column suffixes:

The following suffixes have a universal meaning ensuring the columns can be read and understood easily from SQL code. Use the correct suffix where appropriate.

\_id—a unique identifier such as a column that is a primary key.

\_status—flag value or some other status of any type such as publication\_status.

\_total—the total or sum of a collection of values.

\_num—denotes the field contains any kind of number.

\_name—signifies a name such as first\_name.

\_seq—contains a contiguous sequence of values.

\_date—denotes a column that contains the date of something.

\_tally—a count.

\_size—the size of something such as a file size or clothing.

\_addr—an address for the record could be physical or intangible such as ip\_addr.

23. Always end a SQL statement with semicolon (;)

In the past, T-SQL didn't have a requirement to terminate all statements with a semicolon. However, standard SQL requires it. Also, with each new version of SQL Server, this becomes a requirement for more and more specific cases to avoid parsing ambiguity. SQL Server 2008 already added "not terminating statements with a semicolon" to the list of deprecated features. So do yourself a favor and if you haven't done so already, start getting used to terminating all of your statements.

24. SQL GROUP BY techniques. Here's a classic example that returns total sales per customer, in addition to returning the customer's name and address:

SELECT

C.CustomerID, C.CustomerName,

C.CustomerType, C.Address1, C.City,

SUM(S.Sales) as TotalSales

FROM

Customers C

INNER JOIN Sales S

ON C.CustomerID = S.CustomerID

GROUP BY

C.CustomerID, C.CustomerName,

C.CustomerType, C.Address1, C.City;

You should only be grouping on CustomerID, and not on all those other columns. Push the grouping down a level, into a derived table:

SELECT

C.CustomerID, C.CustomerName,

C.CustomerType, C.Address1, C.City,

S.TotalSales

FROM

Customers C

INNER JOIN

(SELECT

CustomerID, SUM(Sales) as TotalSales

FROM

Sales

GROUP BY

CustomerID) S

ON

C.CustomerID = S.CustomerID;

25. Create procedure template. It should include Jira number and how to use it example.

--- Modified Date Created By Description ---

--- =================================================================== ---

--- 02/14/2013 Yury Yershov New ---

--- =================================================================== ---

--- 07/15/2016 Dmitriy Burtsev DBP-3406 ---

--- ---

--- EXEC dbo.StoredProcedure\_sp @FundFamily = 'ABC', @Discretion = 'DE' ---

--- ---

--- calls from Production/MyBeautifulReport report ---

---

ALTER PROCEDURE [dbo].[StoredProcedure\_sp]

@FundFamily VARCHAR(1000),

@Discretion VARCHAR(20)

AS

SET NOCOUNT ON;

26. Do not use SameDatabaseName.dbo.TableName syntax.

Example (Common database):

-- get procedures with SameDatabaseName.dbo.TableName syntax

SELECT OBJECT\_NAME(sm.object\_id), sm.definition FROM sys.sql\_modules sm WHERE sm.definition LIKE ('%' + DB\_NAME() + '.%');

CREATE PROCEDURE [STG].[Vestmark\_AdvisoryAccountDetails\_File\_Update]

(

...

UPDATE Common.dbo.ACCOUNTS\_FOCUSED\_TRACKING

SET STATUS =

Reason: Sometimes we need to restore database under a different name or create a database snapshot.

It also confused some developers because we expect this syntax for objects from another database.

Another example, this code runt for BetaNDM but fail ob any other db like BetaNDMTest:

CREATE PROCEDURE [dbo].[Daily\_Import\_fba\_rtb\_summary\_sp]

. . .

--Update Normalized table so that total\_tiers has the actual number of tiers and not the normalized number of tiers.

UPDATE fba\_rtb\_normalized

SET BetaNDM.dbo.fba\_rtb\_normalized.total\_tiers = b.total\_tiers

FROM fba\_rtb\_normalized

INNER JOIN BetaNDM.dbo.fba\_rtb\_master b ON b.rate\_id = BetaNDM.dbo.fba\_rtb\_normalized.rate\_id

. . .

Msg 4104, Level 16, State 1, Procedure Daily\_Import\_fba\_rtb\_summary\_sp, Line 247

The multi-part identifier "BetaNDM.dbo.fba\_rtb\_normalized.rate\_id" could not be bound.

26. Use SQL description, when needed, for fields, tables, views, functions and procedures

CREATE TABLE [dbo].[Table\_1](

[Field1] [INT] NULL,

[Field2] [NCHAR](10) NOT NULL

) ON [PRIMARY]

GO

EXEC sys.sp\_addextendedproperty @name=N'MS\_Description', @value=N'This is description for Field1' , @level0type=N'SCHEMA',@level0name=N'dbo', @level1type=N'TABLE',@level1name=N'Table\_1', @level2type=N'COLUMN',@level2name=N'Field1'

EXEC sys.sp\_addextendedproperty @name=N'MS\_Description', @value=N'This is description for Field2' , @level0type=N'SCHEMA',@level0name=N'dbo', @level1type=N'TABLE',@level1name=N'Table\_1', @level2type=N'COLUMN',@level2name=N'Field2'

EXEC sys.sp\_addextendedproperty @name=N'MS\_Description', @value=N'This is table description' , @level0type=N'SCHEMA',@level0name=N'dbo', @level1type=N'TABLE',@level1name=N'Table\_1'

GO

27. It is always good practice to refer to database objects by a schema name and the object name, separated by a period (.).

SELECT \* FROM HumanResources.Employee;

28. Clustered Index Design Guidelines.

Clustered indexes sort and store the data rows in the table based on their key values. There can only be one clustered index per table. Consider using a clustered index for queries that do the following:

* Return a range of values by using operators such as BETWEEN, >, >=, <, and <=.
* Return large result sets.
* Use JOIN clauses; typically these are foreign key columns.
* Use ORDER BY, or GROUP BY clauses.

An index on the columns specified in the ORDER BY or GROUP BY clause may remove the need for the Database Engine to sort the data, because the rows are already sorted. This improves query performance.

Clustered indexes are not a good choice for the following attributes:

* Columns that undergo frequent changes

This causes in the whole row to move, because the Database Engine must keep the data values of a row in physical order. This is an important consideration in high-volume transaction processing systems in which data is typically volatile.

* Wide keys

Wide keys are a composite of several columns or several large-size columns. Any nonclustered indexes defined on the same table will be significantly larger because the nonclustered index entries contain the clustering key and also the key columns defined for that nonclustered index.

29. Nonclustered Index Design Guidelines

A nonclustered index contains the index key values and row locators that point to the storage location of the table data.

* Databases or tables with low update requirements, but large volumes of data can benefit from many nonclustered indexes to improve query performance. Consider creating filtered indexes for well-defined subsets of data to improve query performance, reduce index storage costs, and reduce index maintenance costs compared with full-table nonclustered indexes.
* Online Transaction Processing applications and databases that contain heavily updated tables should avoid over-indexing. Additionally, indexes should be narrow, that is, with as few columns as possible.

Large numbers of indexes on a table affect the performance of INSERT, UPDATE, DELETE, and MERGE statements because all indexes must be adjusted appropriately as data in the table changes.

Consider using a nonclustered index for queries that have the following attributes:

* Use JOIN or GROUP BY clauses.

Create multiple nonclustered indexes on columns involved in join and grouping operations, and a clustered index on any foreign key columns.

* Queries that do not return large result sets.

Create filtered indexes to cover queries that return a well-defined subset of rows from a large table.

* Contain columns frequently involved in search conditions of a query, such as WHERE clause, that return exact matches.
* If the table has a clustered index, the column or columns defined in the clustered index are automatically appended to the end of each nonclustered index on the table. This can produce a covered query without specifying the clustered index columns in the definition of the nonclustered index. For example, if a table has a clustered index on column C, a nonclustered index on columns B and A will have as its key values columns B, A, and C.
* Lots of distinct values, such as a combination of last name and first name, if a clustered index is used for other columns.
* Index key columns, excluding nonkeys, must follow the existing index size restrictions of 16 key columns maximum, and a total index key size of 900 bytes.

30. Bulk loading large data sets into SQL Server.

Minimally Logged Operations

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Table Indexes** | **Rows in table** | **Hints** | **Without TF 610** | **With TF 610** | **Concurrent possible** |
| Heap | Any | TABLOCK | Minimal | Minimal | Yes |
| Heap | Any | None | Full | Full | Yes |
| Heap + Index | Any | TABLOCK | Full | Depends (3) | No |
| Cluster | Empty | TABLOCK, ORDER (1) | Minimal | Minimal | No |
| Cluster | Empty | None | Full | Minimal | Yes (2) |
| Cluster | Any | None | Full | Minimal | Yes (2) |
| Cluster | Any | TABLOCK | Full | Minimal | No |
| Cluster + Index | Any | None | Full | Depends (3) | Yes (2) |
| Cluster + Index | Any | TABLOCK | Full | Depends (3) | No |

**Table 1:** Summary of minimal logging conditions

(1) If you are using the INSERT … SELECT method, the ORDER hint does not have to be specified, but the rows must be in the same order as the clustered index. If using BULK INSERT the order hint must be used.

(2) Concurrent loads only possible under certain conditions. See “Bulk Loading with the Indexes in Place”. Also, only rows written to newly allocated pages are minimally logged.

(3) Depending on the plan chosen by the optimizer, the nonclustered index on the table may either be fully- or minimally logged.

Choosing Between Bulk Load Methods

The following table provides an overview of the different bulk methods available in SQL Server and Integration Services.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Functionality** | **Integration Services** | | **BULK INSERT** | **BCP** | **INSERT … SELECT** |
| **SQL Dest.** | **OLE DB Dest** |
| **Protocol** | Shared Memory | TCP/IP  Named Pipes | In Memory | TCP/IP  Shared Memory  Named Pipes | In Memory |
| **Speed** | Faster / Fastest(4) | Fast / Fastest (1) | Fastest | Fast | Slow / Fastest (2) |
| **Data Source** | Any | Any | Data File Only | Data File Only | Any OLE DB |
| **Bulk API Support** | Not Native | Not ORDER  Not Native | All | All | No Hints Allowed |
| **Lock taken with TABLOCK hint on heap** | BU | BU | BU | BU | X |
| **Can transform in transit** | Yes | Yes | No | No | Yes |
| **I/O Read block Size** | Depends(3) | Depends(3) | 64 kilobytes (KB) | 64 KB | Up to 512 KB |
| **SQL Server Version** | 2005 and 2008 | 2005 and 2008 | 7.0, 2000, 2005, and 2008 | 6.0, 7.0, 2000, 2005, and 2008 | 2008 |
| **Invoked from** | DTEXEC / BIDS | DTEXEC / BIDS | Transact-SQL | Command Line | Transact-SQL |

**Table 2:** Choosing among bulk load methods

(1) If you run DTEXEC on a different server than SQL Server, Integration Services can deliver very high speed by offloading data conversions from the Database Engine.  
(2) Note that INSERT … SELECT does not allow concurrent inserts into a single table. If used to populate a single table, Integration Services will often be a faster option because you can run multiple streams in parallel.  
(3) The read block size depends on source. For text files, 128 KB block sizes are used.  
(4) SQL Server Destination will use more CPU cycles than BULK INSERT, limiting max speeds. But because it offloads the data conversion, the throughput of a single stream insert is faster than BULK INSERT.

Bulk Loading an Empty, Nonpartitioned Table

Before you load data into an empty table, it is always a good idea to drop or disable all indexes on the table. After the load is done, re-create or re-enable the indexes.

Consider using multiple bulk insert operations in parallel. A single bulk operation will only fully utilize one CPU core. BCP, BULK INSERT, and Integration Services are all capable of taking bulk update (BU) locks – if you specify the TABLOCK hint.

Bulk Loading into a Nonpartitioned Table That Already Has Data.

If you are importing a small amount of new data relative to the amount of existing data, dropping and rebuilding the indexes may be counter productive. The time required to rebuild the indexes is likely to be longer than the time saved during the bulk operation.

In contrast, if you are importing a relatively large amount of new data, dropping the indexes on the table before performing the bulk operation can increase performance, without substantially increasing the time required for indexing.

Bulk Loading with the Indexes in Place.

Tables with indexes do not support the bulk update (BU) lock – instead exclusive (X) locks are taken. Bulk loading into a table with indexes will take exclusive (X) locks at the row level. To achieve a concurrent bulk load into an indexed table, you should therefore not use the TABLOCK hint. Lock escalation to table level may occur as the number of row-level exclusive (X) locks rises above a threshold. If this happens, concurrency will drop to single threaded inserts.

ALTER TABLE Sales SET (LOCK\_ESCALATION = DISABLE);

Bulk Loading a Partitioned Table

Bulk loading into partitioned tables provides the fastest possible load option. Switching in partitions once the data loading is done as a metadata-only operation.

It is possible to take bulk update (BU) locks on the switched-out object alone and have several bulk streams operating on the same time.

You can drop and rebuild the indexes on the switched-out partition before switching it back in. This can increase bulk load speed.

Bulk loading directly into a partitioned table will trigger a sort operation, even if there are no indexes on the table or the incoming data has an ORDERED hint. This sort operation is caused by the optimizer removing the overhead of continuously opening and closing new partitions under heavy insert activity.

**1. Create temporary tables for staging.**

|  |  |
| --- | --- |
| https://i-technet.sec.s-msft.com/dynimg/IC266118.gif | Create temporary tables as switch targets. The method to perform this is described in “Partition SWITCH”.  Switch data into the temp tables:  ALTERTABLE Sales\_P  SWITCH PARTITION<X>  TO Sales\_200<X> |

**2. Apply bulk load optimization on each individual table.**

|  |  |
| --- | --- |
| https://i-technet.sec.s-msft.com/dynimg/IC347268.gif | BULKINSERT Sales\_200<X>  FROM'Sales200<X><Y>.csv'  WITH (   FIELDTERMINATOR=';'   ,ROWTERMINATOR='\n'  )  …etc… |

**3. Switch all tables back into main table.**

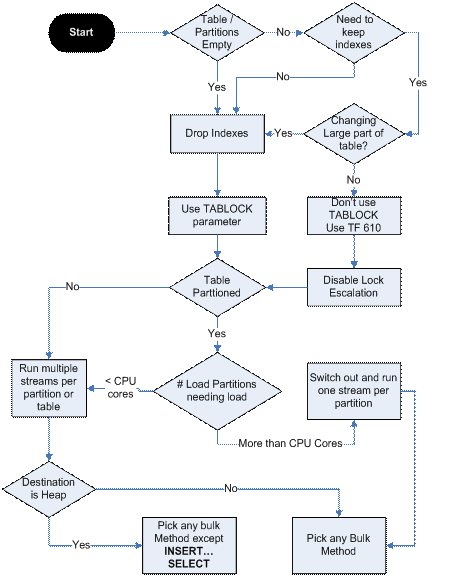
|  |  |
| --- | --- |
| https://i-technet.sec.s-msft.com/dynimg/IC266120.gif | Switch back the staging tables:  ALTERTABLE Sales\_200<X>  SWITCH TO Sales\_P  PARTITION<X> |

**4. Clean up the staging tables**

|  |  |
| --- | --- |
| https://i-technet.sec.s-msft.com/dynimg/IC266121.gif | Drop the temporary staging tables:  DROPTABLE Sales\_200<X> |

Summarizing Insert Scenarios

The following flow chart helps you solve the different bulk load scenarios.



31. BATCHSIZE and ROWS\_PER\_BATCH

BATCHSIZE, describes how many rows are committed at a time during the bulk operation. If this value is left at its default, the entire bulk operation is committed as one, big transaction. However, if this value is greater than 0, a new transaction is created and committed every time the amount of rows specified has been sent to the server.

ROWS\_PER\_BATCH, is an estimate of the total number of rows for the entire bulk load operation. The parameter is used as a hint to the query optimizer when BATCHSIZE is not specified. If you can estimate the number of rows, the optimizer can allocate the correct amount of memory resources for the batch.

If a nonzero BATCHSIZE is used in a bulk load method, you must leave the ROW\_PER\_BATCH setting as empty or zero.

This is how we get numbers of rows from file:

DECLARE @RowsNum INT, @sql\_bcp VARCHAR(8000);

CREATE TABLE #output (id INT IDENTITY(1,1), cmdoutput NVARCHAR(255) NULL);

SET @sql\_bcp = 'find /v /c "" \\snsqldbp14\MSSQLSERVER\FileTables\SQLarchive\BetaNDM\stf\_TAXL\_RATI\_20161021091115';

INSERT INTO #output ( cmdoutput ) EXEC master..xp\_cmdshell @sql\_bcp ;

IF (@@ROWCOUNT >=3)

BEGIN

SET @RowsNum = (SELECT CAST(RIGHT(cmdoutput, (LEN(cmdoutput) - CHARINDEX(':', cmdoutput) - 1)) AS INT) FROM #output WHERE cmdoutput IS NOT NULL);

PRINT @RowsNum;

END

ELSE

BEGIN

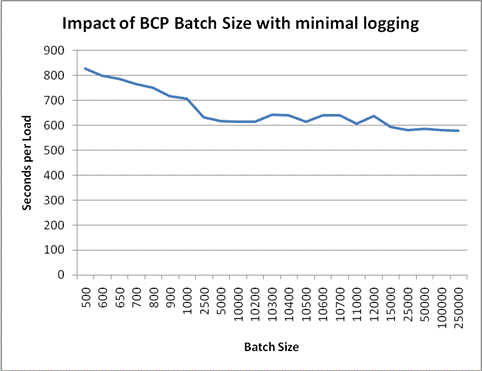
PRINT 'xp\_cmdshell fail';

END;

DROP TABLE #output;

Note: this code take about 1 second to run.

Figure 3 shows this impact on a 17-gigabyte (GB) table loading with one bulk stream. Note that for values over 10.000, little further improvement is observed.



**Figure 3:** Impact of BCP Batch Size with minimal logging

|  |  |  |
| --- | --- | --- |
| **Bulk Method** | **BATCHSIZE** | **ROWS\_PER\_BATCH** |
| Integration Services  OLE DB Destination | See “Integration Services Batch Sizes” | Rows per Batch |
| Integration Services  SQL Server Destination | See “Integration Services Batch Sizes” | Not applicable |
| BCP | -b <X> (1) | -h “ROWS\_PER\_BATCH = <X>” |
| BULK INSERT | BATCHSIZE = X | ROWS\_PER\_BATCH = <X> |
| INSERT … SELECT (2) | N/A | ROWS\_PER\_BATCH (3) |

**Table 3:** BATCHSZIE and ROWS\_PER\_BATCH from different bulk methods

(1) If BATCH\_SIZE is not specified, BCP will use 1000 as the default value   
(2) INSERT … SELECT cannot control BATCHSIZE and ROWS\_PER\_BATCH  
(3) Must use OPENROWSET with the bulk hint as the source

32. BCP and BULK INSERT KILOBYTES\_PER\_BATCH option

KILOBYTES\_PER\_BATCH = cc

Specifies the approximate number of kilobytes (KB) of data per batch as kilobytes\_per\_batch. By default, KILOBYTES\_PER\_BATCH is unknown.

Performance Considerations

If the number of pages to be flushed in a single batch exceeds an internal threshold, a full scan of the buffer pool might occur to identify which pages to flush when the batch commits. This full scan can hurt bulk-import performance. A likely case of exceeding the internal threshold occurs when a large buffer pool is combined with a slow I/O subsystem. To avoid buffer overflows on large machines, either do not use the TABLOCK hint (which will remove the bulk optimizations) or use a smaller batch size (which preserves the bulk optimizations).

Because computers vary, we recommend that you test various batch sizes with your data load to find out what works best for you.

This is how we can get file size from TSQL:

DECLARE @FileSize INT, @sql\_bcp VARCHAR(8000);

CREATE TABLE #output (id INT IDENTITY(1,1), cmdoutput NVARCHAR(255) NULL)

SET @sql\_bcp = 'for %I in (\\snsqldbp14\MSSQLSERVER\FileTables\SQLarchive\BetaNDM\stf\_TAXL\_RATI\_20161021091115) do @echo %~zI';

INSERT #output ( cmdoutput ) EXEC master..xp\_cmdshell @sql\_bcp ;

SET @FileSize = (SELECT TRY\_CAST(cmdoutput AS INT) FROM #output WHERE cmdoutput IS NOT NULL);

-- convert bytes to kilobytes

SET @FileSize = CASE WHEN @FileSize < 1024 THEN 1 ELSE @FileSize / 1024 END;

PRINT @FileSize;

DROP TABLE #output;

Note: This operation took one minute (01:04) on SNSQLDBD14S even for small files.

33. BCP on the same machine may use named pipes.

When BCP runs on the same machine as SQL Server, using local named pipes greatly speeds the process. Local pipes are an interposes communication (IPC) mechanism and completely bypass the network to optimize processes running on the same machine. BCP automatically uses a local named pipe if you do not provide a server name when you run the command.

34. Think of NULLs and the Three-Valued Logic

One of the most important best practices in T-SQL is that whenever you write T-SQL code, you should always be mindful of NULLs, as well as the fact that interaction with NULLs is special and sometimes actually counterintuitive.

There is an example:

SELECT custid, country, region, city FROM Sales.Customers WHERE region <> N'WA';

So this query will return only customers where the region is known to be different than WA and not NULL. To fix this problem and include customers from locations where the region is inapplicable, you need to add logic to the query, like so:

SELECT custid, country, region, city FROM Sales.Customers WHERE region <> N'WA' OR region IS NULL;

35. Using a NOT IN predicate to look for non-matches is very dangerous when one of the elements in the subquery can be NULL. For example, suppose that you're writing the following query to return customers who didn't place orders:

SELECT custid FROM Sales.Customers WHERE custid NOT IN (SELECT custid FROM Sales.Orders);

If any order has a NULL custid value, this query will return an empty set, even when there are customers who didn't place orders.

After you identify the problem, you can determine a solution. For example, you can explicitly eliminate the NULLs in the subquery with a filter saying WHERE custid IS NOT NULL, and therefore you end up comparing the outer custid value only with present custid values from the inner set. Another option is to use NOT EXISTS instead because it relies on two-valued logic.

SELECT

CASE WHEN 1 NOT IN ( 2, 3 ) THEN 'True'

ELSE 'Unknown or False'

END ,

CASE WHEN 1 NOT IN ( 2, 3, NULL ) THEN 'True'

ELSE 'Unknown or False'

END ;

(No column name) (No column name)

True Unknown or False

Correction with NOT EXISTS

SELECT 'True' WHERE NOT EXISTS

(SELECT tbl.Num

FROM (VALUES (2),(3),(NULL)) AS Tbl(Num)

WHERE Tbl.Num = 1);

(No column name)

True

One more example with EXCEPT:

DECLARE @T1 TABLE (COL1 char(1));

DECLARE @T2 TABLE (COL1 char(1));

INSERT INTO @T1 VALUES

('A'),

('B');

INSERT INTO @T2 VALUES

('A'),

(null);

SET ANSI\_NULLS ON

SELECT COL1 FROM @T1 WHERE COL1 NOT IN (SELECT COL1 FROM @T2)

SELECT COL1 FROM @T1 T1 WHERE NOT EXISTS (SELECT \* FROM @T2 T2 WHERE T2.COL1=T1.COL1)

SELECT COL1 FROM @T1 EXCEPT SELECT COL1 FROM @T2;

COL1

----

COL1

----

B

COL1

----

B

36. Avoid NOLOCK

The use of the NOLOCK query hint (equivalent to using the read uncommitted isolation level) is one of the most common practices in T-SQL yet one of the worst. The practice is so common because it makes queries run faster. Typically, the storage engine will first prioritize the correctness expectations from the request before performance. But when using the NOLOCK option, the storage engine will assume your main focus is speed and will therefore prefer the faster option. This can result in double-reading or skipping rows in cases that would typically not be allowed to happen. In short, you get data consistency problems with NOLOCK well beyond getting uncommitted reads.

37. Date and Time - Use language-neutral literals.

When expressing date literals, a common practice is to use a form that's language-dependent because that's what the developer is used for (e.g., the form '01/02/11'). However, depending on the language of the login connected to SQL Server and running the code, this form can be interpreted in several different ways.

The best practice is then to use a form that's language-neutral. For example, when using '20110102', SQL Server will always interpret the date as YYYYMMDD because this form is language-neutral.

38. Avoid BETWEEN for Date and Time.

You want to filter a period of date and time data such as January 2012. Some people use the following filter form:

WHERE col BETWEEN '20120101' AND '20120131 23:59:59.999'

The problem is that 999 as the millisecond unit isn't a multiplication of the precision unit for DATETIME, which is three and a third milliseconds. Therefore, the value gets rounded to the next midnight, and your range might end up including rows it isn't supposed to include. Therefore, the best practice with date and time ranges is to avoid BETWEEN and to always use the form:

WHERE col >= '20120101' AND col < '20120201'

39. Use Search Arguments (SARG)

When optimizing a query and considering whether and how to use indexes, one of the key elements the optimizer evaluates is the predicates in the query filters. When the optimizer can rely on index ordering for a given predicate, the predicate is said to be a search argument, or SARG for short. Here are a few examples:

The predicate COALESCE(T1.col1, -1) = COALESCE(T2.col1, -1) isn't a SARG, whereas the predicate T1.col1 = T2.col1 OR (T1.col1 IS NULL and T2.col1 IS NULL) is.

The predicate col1 - 1 <= @n isn't a SARG, whereas the predicate col1 <= @n + 1 is.

The predicate LEFT(col1, 3) isn't a SARG, whereas the predicate col1 LIKE 'abc%' is.

As you can see, sometimes simple revisions to your predicates can retain the meaning and allow efficient use of indexing.

40. The problem of ambiguous updates (UPDATE…FROM)

Example: populating a permanent table from a staging table.

CREATE TABLE dbo.Codes (

Code VARCHAR(5) NOT NULL ,

Description VARCHAR(40) NOT NULL ,

CONSTRAINT PK\_Codes PRIMARY KEY ( Code ) ) ;

GO

CREATE TABLE dbo.CodesStaging (

Code VARCHAR(10) NOT NULL ,

Description VARCHAR(40) NOT NULL ) ;

Now, let's populate each table with some sample data

DELETE FROM dbo.Codes ;

INSERT INTO dbo.Codes ( Code , Description )

SELECT 'AR' , 'Old description for Arkansas'

UNION ALL

SELECT 'IN' , 'Old description for Indiana' ;

DELETE FROM dbo.CodesStaging ;

INSERT INTO dbo.CodesStaging ( Code , Description )

SELECT 'AR' , 'description for Argentina'

UNION ALL

SELECT 'AR' , 'new description for Arkansas'

UNION ALL

SELECT 'IN' , 'new description for Indiana ' ;

The incoming data in our staging table has a duplicate: the code AR occurs twice, with different descriptions. Suppose that we have not detected or resolved this duplicate, and that we are updating our Codes table from the staging table.

UPDATE dbo.Codes

SET Description = s.Description

FROM dbo.Codes AS c INNER JOIN dbo.CodesStaging AS s ON c.Code = s.Code ;

SELECT Code, Description FROM dbo.Codes;

Code Description

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

AR description for Argentina

IN new description for Indiana

(2 row(s) affected)

Although two descriptions were provided for the AR code, the UPDATE…FROM command did not raise an error; it just silently updated the corresponding row in the Codes table with one of the two provided values.

Updating with CTE

;WITH c AS (

SELECT c.Code , c.Description , s.Description AS NewDescription

FROM dbo.Codes AS c INNER JOIN dbo.CodesStaging AS s ON c.Code = s.Code

)

UPDATE c SET Description = NewDescription ;

(2 row(s) affected)

SELECT Code , Description FROM dbo.Codes ;

Code Description

AR description for Argentina

IN new description for Indiana

Note that, neither in this example nor in the previous UPDATE…FROM example, can we predict which of these two values will end up in the target table – that, as usual, depends on the execution plan and, as such, is completely unpredictable.

Using MERGE to detect ambiguity

MERGE INTO dbo.Codes AS c

USING dbo.CodesStaging AS s ON c.Code = s.Code

WHEN MATCHED THEN UPDATE SET c.Description = s.Description ;

Msg 8672, Level 16, State 1, Line 37

The MERGE statement attempted to UPDATE or DELETE the same row more than once. This happens when a target row matches more than one source row. A MERGE statement cannot UPDATE/DELETE the same row of the target table multiple times. Refine the ON clause to ensure a target row matches at most one source row, or use the GROUP BY clause to group the source rows.

An ANSI-standard method to detect ambiguity:

UPDATE dbo.Codes

SET Description = (

SELECT Description

FROM dbo.CodesStaging

WHERE Codes.Code = CodesStaging.Code

)

WHERE EXISTS (

SELECT \* FROM dbo.CodesStaging AS s WHERE Codes.Code = s.Code

) ;

Msg 512, Level 16, State 1, Line 41

Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

The statement has been terminated.

If your business rules allow you to ignore ambiguities and only update that which is unambiguous, then the solution is using PARTITION BY to ignore ambiguities when updating an inline view (CTE).

;WITH c AS (

SELECT c.Code ,

c.Description ,

s.Description AS NewDescription ,

COUNT(\*) OVER ( PARTITION BY s.Code ) AS NumVersions

FROM dbo.Codes AS c

INNER JOIN dbo.CodesStaging AS s ON c.Code = s.Code

)

UPDATE c SET Description = NewDescription

WHERE NumVersions = 1 ;

(1 row(s) affected)

Or with analytical function for detecting and ignoring ambiguities:

UPDATE dbo.Codes

SET Description = s.Description

FROM dbo.Codes AS c

INNER JOIN ( SELECT Code, Description, COUNT(\*) OVER ( PARTITION BY Code ) AS NumValues

FROM dbo.CodesStaging ) AS s ON c.Code = s.Code AND NumValues = 1 ;

41. Conditions in a WHERE clause can evaluate in any order

Never assume that the conditions in your WHERE clause will evaluate in the left-to-right order in which you list them.

SELECT Subject , Body

FROM dbo.EmailMessages

WHERE ISDATE(VarcharColumn) = 1 AND CAST(VarcharColumn AS DATETIME) = '20090707';

This query might work for some data, but it can blow up at any time, and the reason is simple: the conditions in the WHERE clause can evaluate in any order, and the order can change from one execution of the query to the next. If the CAST is evaluated before the ISDATE validity check, and if the string is not a valid date, than the query will fail.

The safe way to develop such queries is to use CASE expressions to explicitly specify the order in which our conditions must evaluate:

SELECT MessageID , SenderID , ReceiverID , MessageDateAsVarcharColumn , SomeMoreData

FROM dbo.Messages

WHERE CASE WHEN ISDATE(MessageDateAsVarcharColumn) = 1

THEN CAST(MessageDateAsVarcharColumn AS DATETIME) END = '20090707';

We can get all valid Date Time rows by this statement:

SELECT MessageID , SenderID , ReceiverID , MessageDateAsVarcharColumn , SomeMoreData

FROM dbo.Messages

WHERE CASE WHEN ISDATE(MessageDateAsVarcharColumn) = 1

THEN NULL

ELSE ''

END IS NULL ;

42. Assigning values to variables - SET vs. SELECT.

SET is the ANSI standard way of assigning values to variables and SELECT is not. But you can use SELECT to assign values to more than one variable at a time.

If the query returns multiple values/rows then SET will raise an error. SELECT will assign one of the values to the variable and hide the fact that multiple values were returned (so you'd likely never know why something was going wrong elsewhere - have fun troubleshooting that one).

SELECT may leave a variable unchanged if the selected result set is empty.

CREATE TABLE dbo.Orders (

OrderID INT NOT NULL ,

OrderDate DATETIME NOT NULL ,

IsProcessed CHAR(1) NOT NULL ,

CONSTRAINT PK\_Orders PRIMARY KEY ( OrderID ));

GO

INSERT dbo.Orders ( OrderID , OrderDate , IsProcessed) VALUES

(1 , '20090420' , 'N'),

(2 , '20090421' , 'N'),

(3 , '20090422' , 'N');

GO

CREATE PROCEDURE dbo.ProcessBatchOfOrders @IDsIntervalSize INT

AS

DECLARE @minID INT , @ID INT ;

SELECT @minID = MIN(OrderID) ,

@ID = MIN(OrderID)

FROM dbo.Orders ;

WHILE @ID < ( @minID + @IDsIntervalSize )

BEGIN;

UPDATE dbo.Orders

SET IsProcessed = 'Y'

WHERE OrderID = @ID ;

-- this SELECT may leave the value-- of @ID unchanged

SELECT TOP (1) @ID = OrderID

FROM dbo.Orders

WHERE IsProcessed = 'N'

ORDER BY OrderID ;

-- PRINT is needed for debugging purposes only

PRINT @ID ;

END ;

GO

-- The execution of dbo.ProcessBatchOfOrders results in an infinite loop.

EXEC dbo.ProcessBatchOfOrders 10 ;

Replacing the SELECT with a SET removes the infinite loop.

ALTER PROCEDURE dbo.ProcessBatchOfOrders @IDsIntervalSize INT

AS

DECLARE @minID INT , @ID INT ;

SELECT @minID = MIN(OrderID) ,

@ID = MIN(OrderID)

FROM dbo.Orders ;

WHILE @ID < ( @minID + @IDsIntervalSize )

BEGIN;

UPDATE dbo.Orders

SET IsProcessed = 'Y'

WHERE OrderID = @ID ;

-- SELECT is replaced with SET

SET @ID = ( SELECT TOP (1) OrderID

FROM dbo.Orders

WHERE IsProcessed = 'N'

ORDER BY OrderID ) ;

-- PRINT is needed for debugging purposes

PRINT @ID ; END ;

GO

43. Re-runnable SQL Server Scripts for deployment. Do not drop objects.

Stored procedure:

IF OBJECT\_ID('dbo.sp\_Blitz') IS NULL

EXEC ('CREATE PROCEDURE dbo.sp\_Blitz AS RETURN 0;')

GO

ALTER PROCEDURE [dbo].[sp\_Blitz]

. . .

Table, View and other objects:

IF OBJECT\_ID('table\_name') IS NULL

BEGIN

CREATE TABLE table\_name (ID INT NULL);

END;

Type:

IF TYPE\_ID('NewType') IS NULL

BEGIN

CREATE TYPE NewType FROM INT;

END;

Index:

IF NOT EXISTS(SELECT \* FROM sys.indexes si WHERE si.name = 'IX\_MyTableName\_MyColumnName\_MyColumn2' AND si.object\_id = OBJECT\_ID('MyTableName' ))

BEGIN

CREATE NONCLUSTERED INDEX [IX\_MyTableName\_MyColumnName\_MyColumn2] ON [dbo].[MyTableName]

(MyColumnName ASC , MyColumn2 DESC);

END;

Column:

IF NOT EXISTS(SELECT \* FROM sys.columns sc WHERE sc.Name = N'CreatedDate' AND sc.object\_id = OBJECT\_ID(N'MyTableName'))

BEGIN

Statistics:

IF EXISTS (SELECT s.name FROM sys.stats s WHERE s.name = 'LastFirst' AND s.object\_id = OBJECT\_ID ('Person.Person'))

BEGIN

44. Handling nulls in CHECK constraints.

Logical conditions in CHECK constraints work differently from logical conditions in the WHERE clause. If a condition in a CHECK constraint evaluates to "unknown," then the row can still be inserted, but if a condition in a WHERE clause evaluates to "unknown," then the row will not be included in the result set.

CREATE TABLE dbo.Tablechk

(Clumn INT NULL CONSTRAINT CHK\_Tablechk\_Clumn CHECK (Clumn > 0));

GO

INSERT INTO dbo.Tablechk ( Clumn ) VALUES ( NULL );

SELECT Clumn FROM dbo.Tablechk;

Clumn

NULL

45. A TRY…CATCH example: retrying after deadlocks

The 1205 deadlock victim error can be caught by the CATCH block and the transaction can be rolled back until the threads become unlocked.

The following example shows how TRY…CATCH can be used to handle deadlocks. This first section creates a table that will be used to demonstrate a deadlock state and a stored procedure that will be used to print error information.

USE AdventureWorks2008R2;

GO

-- Verify that the table does not exist.

IF OBJECT\_ID (N'my\_sales',N'U') IS NOT NULL

DROP TABLE my\_sales;

GO

-- Create and populate the table for deadlock simulation.

CREATE TABLE my\_sales

(

Itemid INT PRIMARY KEY,

Sales INT not null

);

GO

INSERT my\_sales (itemid, sales) VALUES (1, 1);

INSERT my\_sales (itemid, sales) VALUES (2, 1);

GO

-- Verify that the stored procedure for error printing

-- does not exist.

IF OBJECT\_ID (N'usp\_MyErrorLog',N'P') IS NOT NULL

DROP PROCEDURE usp\_MyErrorLog;

GO

-- Create a stored procedure for printing error information.

CREATE PROCEDURE usp\_MyErrorLog

AS

PRINT

'Error ' + CONVERT(VARCHAR(50), ERROR\_NUMBER()) +

', Severity ' + CONVERT(VARCHAR(5), ERROR\_SEVERITY()) +

', State ' + CONVERT(VARCHAR(5), ERROR\_STATE()) +

', Line ' + CONVERT(VARCHAR(5), ERROR\_LINE());

PRINT

ERROR\_MESSAGE();

GO

The following code scripts for session 1 and session 2 run simultaneously in two separate SQL Server Management Studio connections. Both sessions try to update the same rows in the table. One of the sessions will succeed with the update operation during the first attempt, and the other session will be selected as the deadlock victim. The deadlock victim error will cause execution to jump to the CATCH block and the transaction will enter an uncommittable state. Inside the CATCH block, the deadlock victim can roll back the transaction and retry updating the table until the update succeeds or the retry limit is reached, whichever happens first.

|  |  |
| --- | --- |
| **Session 1** | **Session 2** |
| USE AdventureWorks2008R2;  GO  -- Declare and set variable  -- to track number of retries  -- to try before exiting.  DECLARE @retry INT;  SET @retry = 5;  -- Keep trying to update  -- table if this task is  -- selected as the deadlock  -- victim.  WHILE (@retry > 0)  BEGIN  BEGIN TRY  BEGIN TRANSACTION;    UPDATE my\_sales  SET sales = sales + 1  WHERE itemid = 1;  WAITFOR DELAY '00:00:13';    UPDATE my\_sales  SET sales = sales + 1  WHERE itemid = 2;  SET @retry = 0;  COMMIT TRANSACTION;  END TRY  BEGIN CATCH  -- Check error number.  -- If deadlock victim error,  -- then reduce retry count  -- for next update retry.  -- If some other error  -- occurred, then exit  -- retry WHILE loop.  IF (ERROR\_NUMBER() = 1205)  SET @retry = @retry - 1;  ELSE  SET @retry = -1;  -- Print error information.  EXECUTE usp\_MyErrorLog;    IF XACT\_STATE() <> 0  ROLLBACK TRANSACTION;  END CATCH;  END; -- End WHILE loop.  GO | USE AdventureWorks2008R2;  GO  -- Declare and set variable  -- to track number of retries  -- to try before exiting.  DECLARE @retry INT;  SET @retry = 5;  --Keep trying to update  -- table if this task is  -- selected as the deadlock  -- victim.  WHILE (@retry > 0)  BEGIN  BEGIN TRY  BEGIN TRANSACTION;    UPDATE my\_sales  SET sales = sales + 1  WHERE itemid = 2;  WAITFOR DELAY '00:00:07';    UPDATE my\_sales  SET sales = sales + 1  WHERE itemid = 1;  SET @retry = 0;  COMMIT TRANSACTION;  END TRY  BEGIN CATCH  -- Check error number.  -- If deadlock victim error,  -- then reduce retry count  -- for next update retry.  -- If some other error  -- occurred, then exit  -- retry WHILE loop.  IF (ERROR\_NUMBER() = 1205)  SET @retry = @retry - 1;  ELSE  SET @retry = -1;  -- Print error information.  EXECUTE usp\_MyErrorLog;    IF XACT\_STATE() <> 0  ROLLBACK TRANSACTION;  END CATCH;  END; -- End WHILE loop.  GO |

Implementing the "retry after deadlock" logic in a C# class.

class SqlCommandExecutor

{

public static void RetryAfterDeadlock (SqlCommand command, int timesToRetry)

{

int retryCount = 0;

while (retryCount < timesToRetry)

{

retryCount++;

try

{

command.ExecuteNonQuery();

Console.WriteLine ("Command succeeded:" + command.CommandText);

return;

}

catch (SqlException e)

{

if (e.Number != 1205)

{throw; }

Console.WriteLine ("Retrying after deadlock:" + command.CommandText);

}

}

}

}

46. When an object is created, developers should also include permissions for the appropriate groups.

use [Common]

GO

GRANT EXECUTE ON [dbo].[bal\_Trade\_ByControlNo\_sp] TO [IUser] AS [dbo];

GO

47. Return status codes from stored procedure.

USE AdventureWorks2012;

GO

-- Drop the procedure if it already exists.

IF OBJECT\_ID(N'HumanResources.usp\_DeleteCandidate', N'P') IS NOT NULL

DROP PROCEDURE HumanResources.usp\_DeleteCandidate;

GO

-- Create the procedure.

CREATE PROCEDURE HumanResources.usp\_DeleteCandidate

(

@CandidateID INT

)

AS

-- Execute the DELETE statement.

DELETE FROM HumanResources.JobCandidate

WHERE JobCandidateID = @CandidateID;

-- Test the error value.

IF @@ERROR <> 0

BEGIN

-- Return 99 to the calling program to indicate failure.

PRINT N'An error occurred deleting the candidate information.';

RETURN 99;

END

ELSE

BEGIN

-- Return 0 to the calling program to indicate success.

PRINT N'The job candidate has been deleted.';

RETURN 0;

END;

GO

CREATE PROCEDURE dbo.Foo\_SP @ID INT

AS

SET NOCOUNT ON;

SELECT Field1, Field2, Field3

FROM dbo.TableName

WHERE Field4 = @ID;

RETURN @@ERROR;

Examples of Return Codes from MSDN:

Return code value Meaning

0 Successful execution.

1 Required parameter value is not specified.

2 Specified parameter value is not valid.

3 Error has occurred getting sales value.

4 NULL sales value found for the salesperson.

48. Consider to use a stored procedure with an input and an output parameter if you return only one row of data.

CREATE PROCEDURE dbo.Foo\_SP

@ID INT,

@CustomerFName VARCHAR(50) OUTPUT,

@CustomerLName VARCHAR(50) OUTPUT,

@CustomerPhone VARCHAR(50) OUTPUT

AS

SET NOCOUNT ON;

SELECT @CustomerFName = FName, @CustomerLName = LName, @CustomerPhone = Phone

FROM dbo.TableName

WHERE Field4 = @ID;

RETURN @@ERROR;

48. Working with comma delimited string as Table-Valued Parameter

CREATE TABLE dbo.Tbl2 (

ID INT NOT NULL,

SomeField DATE NULL,

SomeOtherField FLOAT NULL,

AnotherOneField DECIMAL(10, 5) NULL)

GO

INSERT INTO dbo.Tbl2

( ID ,

SomeField ,

SomeOtherField ,

AnotherOneField

)

VALUES (1, NULL, 1.2, 2.3),

(2, GETDATE(), NULL, 4.5);

INSERT INTO dbo.Tbl1

( ID, SomeStr )

VALUES ( 1, 'String1.1'),

(1, 'String1.2'),

(2,'String2.1'),

(2, 'string2.2'),

(2,'string2.3');

GO

CREATE TYPE dbo.SomeStr\_TT AS TABLE

(SomeStr VARCHAR(30) NOT NULL PRIMARY KEY);

GO

CREATE PROCEDURE dbo.InsertList\_sp

@id INT, @SomeField DATE , @SomeOtherField FLOAT, @TVP SomeStr\_TT READONLY

AS

SET NOCOUNT ON;

INSERT INTO dbo.Tbl2

( ID ,

SomeField ,

SomeOtherField )

VALUES (@id, @SomeField, @SomeOtherField);

INSERT INTO dbo.Tbl1 ( ID, SomeStr )

SELECT @id, SomeStr

FROM @TVP;

RETURN @@ERROR;

GO

DECLARE @StrTVP AS dbo.SomeStr\_TT;

INSERT INTO @StrTVP

( SomeStr )

VALUES ('33437615'),('71810874'),('32797824'),('57918311');

DECLARE @id INT = 1, @SomeField DATE = GETDATE() , @SomeOtherField FLOAT = 1.5;

EXEC dbo.InsertList\_sp @id, @SomeField, @SomeOtherField, @StrTVP;

CREATE FUNCTION dbo.FunctionName\_UDF (@ID INT)

RETURNS TABLE

AS

RETURN (

SELECT t2.ID, t2.SomeField, t2.SomeOtherField,

list= REPLACE(( SELECT t1.SomeStr AS 'data()' FROM dbo.Tbl1 t1 WHERE t1.ID = t2.ID FOR XML PATH('')), ' ', ',')

FROM dbo.Tbl2 t2

WHERE t2.ID = @ID

)

GO

SELECT ID, SomeField, SomeOtherField, list FROM dbo.FunctionName\_UDF (1);

C# call:

System.Data.DataTable table = new DataTable();

table.Columns.Add("SomeStr", typeof(string));

table.Rows.Add("33437615");

table.Rows.Add("71810874");

table.Rows.Add("32797824");

table.Rows.Add("57918311");

using (SqlConnection connection = new SqlConnection("Data Source=NYCITLAP08;Database=Test;Integrated Security=SSPI;Persist Security Info=False;"))

{

connection.Open();

using (SqlCommand command = connection.CreateCommand())

{

command.CommandText = "dbo.InsertList\_sp";

command.CommandType = CommandType.StoredProcedure;

SqlParameter parameter0 = command.Parameters.AddWithValue("@id", 9);

SqlParameter parameter1 = command.Parameters.AddWithValue("@SomeField", DateTime.Now);

SqlParameter parameter2 = command.Parameters.AddWithValue("@SomeOtherField", 1.7);

SqlParameter parameter3 = command.Parameters.AddWithValue("@TVP", table);

parameter3.SqlDbType = SqlDbType.Structured;

parameter3.TypeName = "dbo.SomeStr\_TT";

command.ExecuteNonQuery();

}

}

49. Avoid Using Functions in WHERE Clause

There are two reasons why you want to avoid having a function call in your WHERE clause and more specifically on the columns you are filtering on in your WHERE clause. The first is the function needs to be called for every record in the result set which can slow down your query performance. The second reason which can have even more impact on query performance is the fact that if there is a function surrounding the column you are trying to filter on, any indexes on that column cannot be used.

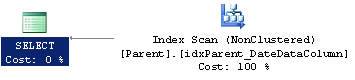
Now let's look at a simple query which would return all the records in the Parent table that are less than 30 days old. Here is one way that we could write the SQL statement.

SELECT ParentID

FROM [dbo].[Parent]

WHERE dateadd(d,30,DateDataColumn) > getdate();

Looking at the explain plan for this query we can see that the index on the DateDataColumn that we created is ignored and an index scan is performed.



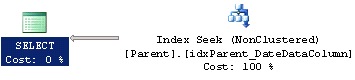
Now let's rewrite this query and move the function to the other side of the > operator. Here is the SQL statement.

SELECT ParentID

FROM [dbo].[Parent]

WHERE DateDataColumn > dateadd(d,-30,getdate());

Looking at the explain plan for this query we can see that the optimizer is now using the index and performs a seek rather than a scan.



To confirm that it is indeed faster let's take a look at the SQL Profiler results for these two queries. We can see below that when using an index, as is usually the case, we use fewer resources and our statement executes faster.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **CPU** | **Reads** | **Writes** | **Duration** |
| Function | 5 | 274 | 0 | 43 |
| No Function | 0 | 5 | 0 | 5 |

50. Join the small tables first

The order in which the tables in your queries are joined can have a dramatic effect on how the query performs. If your query happens to join all the large tables first and then joins to a smaller table later this can cause a lot of unnecessary processing by the SQL engine.

51. Use WHERE, JOIN, ORDERBY, SELECT Column Order When Creating Indexes.

When looking at an explain plan for a query you'll notice that the SQL Optimizer first parses the WHERE clause, then the JOIN clause, followed by the ORDER BY clause and finally it processes the data being selected. Based on this fact it makes sense that you would need to specify the columns in your index in this order if you want the entire index to be used. This is especially true if you are trying to create a covering index.

SELECT P.ParentID,C.ChildID,C.IntDataColumn,C.VarcharDataColumn

FROM [dbo].[Parent] P INNER JOIN

[dbo].[Child] C ON P.ParentID=C.ParentID

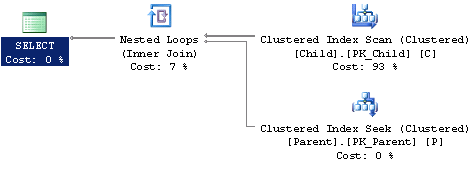
WHERE C.IntDataColumn=32433

ORDER BY ChildID

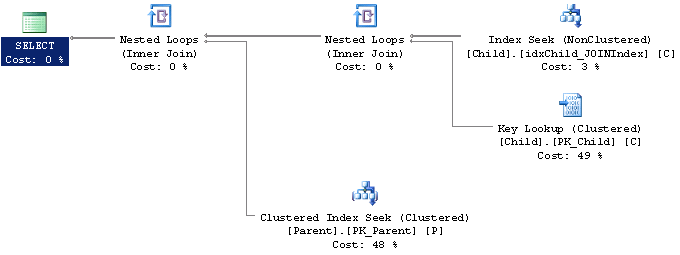
CREATE NONCLUSTERED INDEX idxChild\_JOINIndex

ON [dbo].[Child] ([IntDataColumn],[ParentID],[ChildID]) INCLUDE ([VarcharDataColumn])

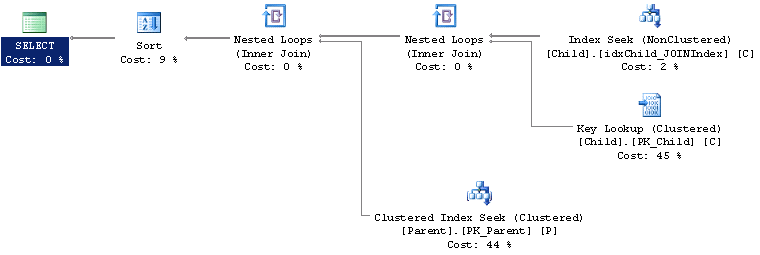
#### No Index



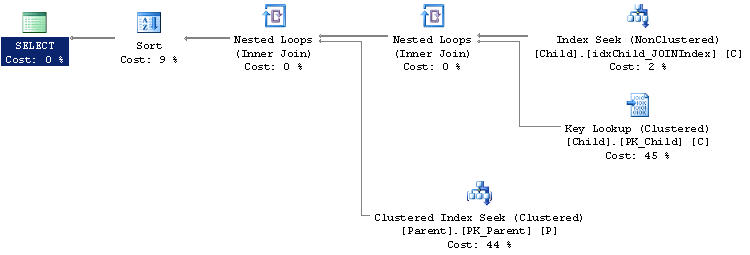
#### WHERE Index



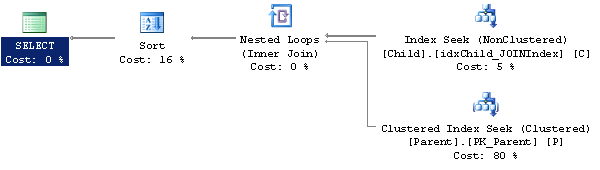
#### WHERE,JOIN Index



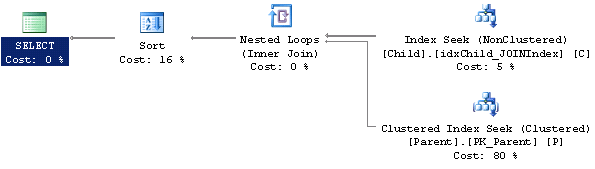
#### WHERE,JOIN,ORDER BY Index



#### WHERE,JOIN,ORDER BY, SELECT Index



#### WHERE,JOIN,ORDER BY, INCLUDE Index



It's hard to tell just from the [explain plans](https://www.mssqltips.com/sqlservertip/1873/how-to-read-sql-server-graphical-query-execution-plans/) if each step will see an improvement or not except for maybe just adding the initial index which eliminated the index scan so let's take a look at the [SQL Profiler](https://www.mssqltips.com/sql-server-tip-category/83/profiler-and-trace/) results to see the actual performance benefit.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table Type** | **CPU** | **Reads** | **Writes** | **Duration** |
| No Index | 110 | 14271 | 0 | 103 |
| WHERE Index | 0 | 129 | 0 | 2 |
| WHERE, JOIN Index | 0 | 117 | 0 | 0 |
| WHERE, JOIN, ORDER BY Index | 0 | 117 | 0 | 0 |
| WHERE, JOIN, ORDER BY, SELECT Index | 0 | 60 | 0 | 0 |
| WHERE, JOIN, ORDER BY, INCLUDE Index | 0 | 60 | 0 | 0 |

We can see from these results that as we add each column we do see the SQL engine has to perform less reads to execute the query thereby executing a little faster. The only exception to this is the step where we added the [ORDER BY](https://www.mssqltips.com/sqlservertutorial/135/select-with-order-by/) to the index but this can be attributed to the fact that we are ordering by ChildID which is a primary key so it's already sorted. The other thing we should note is that there isn't really a performance difference between adding the [SELECT](https://www.mssqltips.com/sqlservertutorial/10/select-command-for-sql-server/) column directly to the index vs. using the INCLUDE clause.

52. ISNULL vs COALESCE

ISNULL ( check\_expression , replacement\_value )

replacement\_value must be of a type that is implicitly convertible to the type of check\_expresssion.

COALESCE ( expression [ ,...n ] )

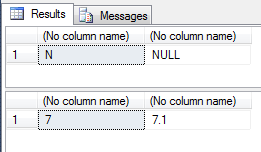
Returns the data type of expression with the highest data type precedence.

DECLARE @a CHAR(1) = NULL

SELECT ISNULL(@a, 'NULL'), COALESCE(@a, 'NULL');

DECLARE @i INT = NULL

SELECT ISNULL(@i, 7.1), COALESCE(@i, 7.1);



53. NULL and CHECK constraint

CREATE TABLE #temp (

Color VARCHAR(15) --NULL

, CONSTRAINT CK CHECK (Color IN ('Black', 'White'))

);

INSERT INTO #temp VALUES (NULL);

(1 row(s) affected)

54. Function in the CASE statement.

On runtime SQL Server will change code like this

CASE Sex

WHEN 'M' THEN 'Male'

WHEN 'F' THEN 'Female'

ELSE 'Unknown'

END

To this:

CASE

WHEN Sex = 'M' THEN 'Male'

WHEN Sex = 'F' THEN 'Female'

ELSE 'Unknown'

END

We can see how it works in select statement:

USE AdventureWorks2014;

CREATE FUNCTION dbo.GetMailUrl

(

@Email NVARCHAR(50)

)

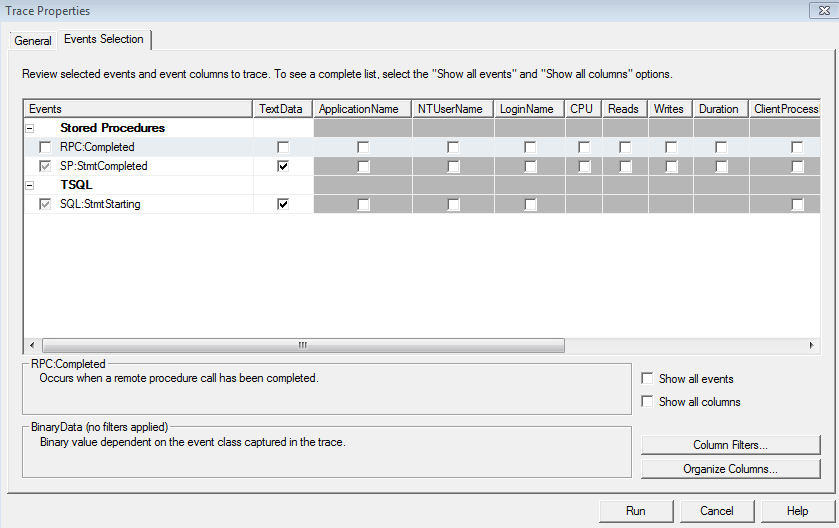
RETURNS NVARCHAR(50)

AS

BEGIN

RETURN SUBSTRING(@Email, CHARINDEX('@', @Email) + 1, LEN(@Email))

END



SELECT TOP(5) EmailAddressID

, EmailAddress

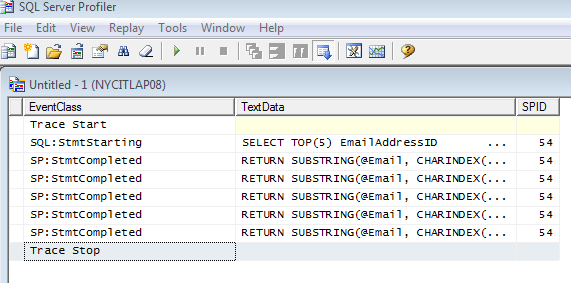
, CASE dbo.GetMailUrl(EmailAddress)

--WHEN 'microsoft.com' THEN 'Microsoft'

WHEN 'adventure-works.com' THEN 'AdventureWorks'

END

FROM Person.EmailAddress



SELECT TOP(5) EmailAddressID

, EmailAddress

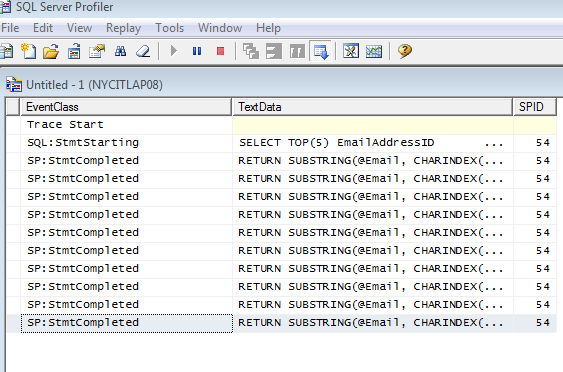
, CASE dbo.GetMailUrl(EmailAddress)

WHEN 'microsoft.com' THEN 'Microsoft'

WHEN 'adventure-works.com' THEN 'AdventureWorks'

END

FROM Person.EmailAddress



We can see that we have function call for every WHEN part of the CASE statement.

Workaround:

SELECT TOP(5) EmailAddressID

, EmailAddress

, CASE Fn.MailUrl

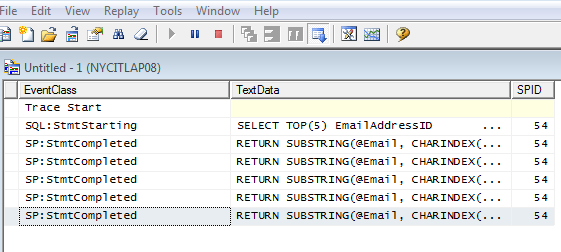
WHEN 'microsoft.com' THEN 'Microsoft'

WHEN 'adventure-works.com' THEN 'AdventureWorks'

END

FROM Person.EmailAddress pe

CROSS APPLY (SELECT dbo.GetMailUrl(pe.EmailAddress) AS MailUrl) AS Fn;



Now the function calls only one time for each row.

55. TSQL String concatenation

For simple concatenation:

CREATE TABLE #t (i CHAR(1))

INSERT INTO #t

VALUES ('1'), ('2'), ('3');

SELECT [text()] = i

FROM #t

FOR XML PATH('');

DECLARE @txt VARCHAR(50) = ''

SELECT @txt = CONCAT(@txt, i)

FROM #t;

SELECT @txt;

If you need more sophisticated select and don’t have special characters like <, >, &

SELECT t.name

, STUFF((

SELECT ', ' + c.name

FROM sys.columns c

WHERE c.[object\_id] = t.[object\_id]

FOR XML PATH('')), 1, 2, '')

FROM sys.objects t

WHERE t.[type] = 'U'

If you do have special characters like <, >, &

SELECT [name], STUFF((

SELECT ', ' + c.[name]

FROM sys.columns c

WHERE c.[object\_id] = t.[object\_id]

FOR XML PATH(''), TYPE).value('(./text())[1]', 'NVARCHAR(MAX)'), 1, 2, '')

FROM sys.objects t

WHERE t.[type] = 'U';

56. Advanced Use of the SQL Server ORDER BY Clause

Example 1

I have a big table, and I just want to sample 10 records randomly for my testing purpose, how can I do it?

USE tempdb;

-- prepare test table

IF OBJECT\_ID('dbo.product', 'U') IS NOT NULL

DROP TABLE dbo.product;

GO

CREATE TABLE dbo.product (id INT IDENTITY PRIMARY KEY ,

[name] VARCHAR(100) ,

[color] VARCHAR(20) ,

[price] INT );

GO

-- populate the table with some data,

-- there are three distinct colors for each product

;WITH L0 AS ( SELECT 1 AS c UNION ALL SELECT 1 AS c ),

L1 AS ( SELECT L0.c FROM L0 CROSS JOIN L0 AS T ),

L2 AS ( SELECT L1.c FROM L1 CROSS JOIN L1 AS T ),

L3 AS ( SELECT L2.c FROM L2 CROSS JOIN L2 AS T ),

L4 AS ( SELECT L3.c FROM L3 CROSS JOIN L3 AS T ),

L5 AS ( SELECT rn = ROW\_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) FROM L4 )

INSERT INTO dbo.product ( name, color, price )

SELECT [name] = 'Product' + CAST(rn AS VARCHAR(5)) ,

color = CASE ( CAST (CEILING(RAND(rn \* 7) \* 100000) AS INT) % 3 )

WHEN 0 THEN 'red'

WHEN 1 THEN 'green'

ELSE 'blue'

END ,

price = CAST(CEILING(RAND(rn \* 7) \* 123456) AS INT) % 56789

FROM L5;

Now let's list 10 records randomly.

-- to list top 10 records randomly, this query is good for medium to large (5K) data set.

-- for small data set, using the 2nd way

SELECT TOP 10 \* FROM dbo.product

ORDER BY DATEPART(ns, GETDATE()) % CAST(( RAND(id) \* 11111111 ) AS INT);

-- another way commonly known

SELECT TOP 10 \* FROM dbo.product ORDER BY NEWID();

Actually there is a third way using OFFSET and FETCH as shown below. This approach will always randomly grab 10 records in sequence.

SELECT \* FROM dbo.product

ORDER BY ( SELECT NULL )

OFFSET CAST(CEILING(RAND() \* 1234567) AS INT) % ( SELECT COUNT(\*) FROM dbo.product ) ROWS

FETCH NEXT 10 ROWS ONLY;

Using the same table, we will create a small sample data and then find the median value of [Price].

We will populate the table with 9 records and later with 10 records to see whether our solution works.

TRUNCATE TABLE dbo.product;

DECLARE @i INT = 1;

WHILE @i < 10

BEGIN

INSERT INTO dbo.product ( name, color, price )

SELECT 'product' + CAST(@i AS VARCHAR(3)),

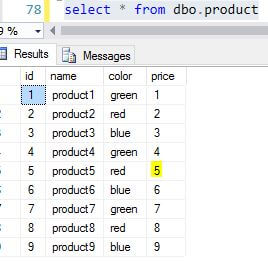
CASE @i % 3 WHEN 0 THEN 'blue' WHEN 1 THEN 'green' ELSE 'red' END ,

@i;

SET @i += 1;

END;

SELECT \* FROM dbo.product;



We can see the median value is 5 for [Price]. So to calculate this in T-SQL, we can use the following code:

-- find the row(s) with median value

SELECT \* FROM dbo.product

ORDER BY price

OFFSET ( SELECT COUNT(\*) - 1 FROM dbo.product ) / 2 ROWS

FETCH NEXT ( CASE ( SELECT COUNT(\*) FROM dbo.product ) % 2

WHEN 1 THEN 1 ELSE 2 END ) ROWS ONLY;

GO

-- find the real median value for [Price] column

WITH c AS (

SELECT \* FROM dbo.product

ORDER BY price

OFFSET ( SELECT COUNT(\*) - 1 FROM dbo.product ) / 2 ROWS

FETCH NEXT ( CASE ( SELECT COUNT(\*) FROM dbo.product ) % 2

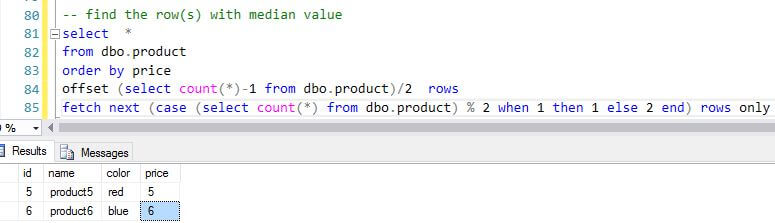
WHEN 1 THEN 1 ELSE 2 END ) ROWS ONLY

)

SELECT median = AVG(price \* 1.)

FROM c;

Assume we insert another record into the table to make it 10 records:



The key here is the OFFSET and FETCH setting, which are decided by the row count, if row count is an odd number, we set the OFFSET to (count-1)/2 and then FETCH next 1 row. But if row count is an even number, we set the same OFFSET value while set FETCH for the next 2 rows.

For an example, if row count = 9, OFFSET = (9-1)/2 = 4, FETCH = 1. If row count=10, OFFSET=(10-1)/2 = 4 (4.5 becomes 4 due to integer conversion), and FETCH=2

Question 3

In the test table created above, there are three distinct colors, Red, Green, Blue, now for my report, I want Green color product to be always on the top of my report, how can I do so in one T-SQL statement?

TRUNCATE TABLE dbo.product;

;WITH L0 AS ( SELECT 1 AS c UNION ALL SELECT 1 AS c ),

L1 AS ( SELECT L0.c FROM L0 CROSS JOIN L0 AS T ),

L2 AS ( SELECT L1.c FROM L1 CROSS JOIN L1 AS T ),

L3 AS ( SELECT L2.c FROM L2 CROSS JOIN L2 AS T ),

L4 AS ( SELECT L3.c FROM L3 CROSS JOIN L3 AS T ),

L5 AS ( SELECT rn = ROW\_NUMBER() OVER ( ORDER BY ( SELECT NULL ) ) FROM L4 )

INSERT INTO dbo.product ( name, color, price )

SELECT [name] = 'Product' + CAST(rn AS VARCHAR(5)) ,

color = CASE ( CAST (CEILING(RAND(rn \* 7) \* 100000) AS INT) % 3 )

WHEN 0 THEN 'red' WHEN 1 THEN 'green' ELSE 'blue' END ,

price = CAST(CEILING(RAND(rn \* 7) \* 123456) AS INT) % 100

FROM L3

WHERE rn <= 10;

--list records with Green product on the top

SELECT \*

FROM dbo.product

ORDER BY CASE color WHEN 'green' THEN 0 ELSE 1 END ASC;

57. Try-Catch exception with SQL Server Transaction

CREATE PROCEDURE dbo.Test1

AS

BEGIN

BEGIN TRY

BEGIN TRAN

SELECT \*

FROM TableNotExist;

COMMIT TRAN

END TRY

BEGIN CATCH

PRINT XACT\_STATE();

PRINT ERROR\_MESSAGE();

IF ( XACT\_STATE() ) = -1

BEGIN

ROLLBACK TRANSACTION;

END;

END CATCH;

END;

GO

EXEC dbo.Test1;

Msg 208, Level 16, State 1, Procedure Test1, Line 25

Invalid object name 'TableNotExist'.

Msg 266, Level 16, State 2, Procedure Test1, Line 25

Transaction count after EXECUTE indicates a mismatching number of BEGIN and COMMIT statements. Previous count = 0, current count = 1.

Now check the transaction state using DMV sys.dm\_tran\_session\_transactions.

SELECT \* FROM sys.dm\_tran\_session\_transactions;

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| session\_id | transaction\_id | transaction\_descriptor | enlist\_count | is\_user\_transaction | is\_local | is\_enlisted | is\_bound | open\_transaction\_count |
| 51 | 427214 | 0x0400000033000000 | 1 | 1 | 1 | 0 | 0 | 1 |

We can see the transaction is still open for session 51.

We should kill this transaction.

USE MASTER; KILL 51;

Changing code to catch errors. When the procedure is run it raises an error like "deferred name resolution" with a **severity below 10**, so the Try-Catch could not trap the error. To resolve this issues, we need to set an explicit transaction using "SET XACT\_ABORT ON".

CREATE PROCEDURE dbo.Test2

AS

SET XACT\_ABORT ON;

BEGIN

BEGIN TRY

BEGIN TRAN

SELECT \*

FROM TableNotExist;

COMMIT TRAN

END TRY

BEGIN CATCH

PRINT XACT\_STATE();

PRINT ERROR\_MESSAGE();

IF ( XACT\_STATE() ) = -1

BEGIN

ROLLBACK TRANSACTION;

END;

END CATCH;

END;

GO

EXEC dbo.Test2;

Msg 208, Level 16, State 1, Procedure Test2, Line 27

Invalid object name 'TableNotExist'.

Now check the transaction state using DMV sys.dm\_tran\_session\_transactions.

SELECT \* FROM sys.dm\_tran\_session\_transactions;

(0 row(s) affected)