The Data Loading Performance Guide

SQL Server Technical Article

Writers: Thomas Kejser, Peter Carlin and Stuart Ozer

Technical Reviewers and contributors: Sunil Agarwal, Ted Lee, David Schwartz, Chris Less, Lindsey Allen, Hermann Daeubler, Juergen Thomas,Sanjay Mishra, Denny Lee, Peter Carlin, Lubor Kollar, David Schwartz, Ted Lee, Sunil Agarwal

Special Thanks: Henk van der Valk (Unisys), Alexei Khalyako, and Marcel van der Holst

Published: January 2009

Applies to: SQL Server 2008 and SQL Server 2005

Summary: This document described techniques for bulk loading large data sets into SQL Server. It covers both the available techniques as well as methodologies to performance tune and optimize the bulk loading process.

Introduction

The white paper describes load strategies for achieving high-speed data modifications of a Microsoft® SQL Server® database.

Before we get into the details of the bulk load methods, we will provide some background information on "minimal logging" in general.

The next two sections: "Bulk Load Methods" and "Other Minimally Logged and Metadata Operations" provide an overview of two key and interrelated concepts for high-speed data loading: bulk loading and metadata operations.

After this background knowledge, we describe how these methods can be used to solve customer scenarios. Script examples illustrating common design pattern are found in "Solving Typical Scenarios with Bulk Loading" Special consideration must be taken when you need to load and read data concurrently in the same table. The section "Bulk Load, NOLOCK Queries, and Read Committed Snapshot Isolation" describes methods you can use to achieve concurrent loading and reading.

This white paper concludes with troubleshooting hints in "Optimizing Bulk Load".

Understanding Minimally Logged Operations

To support high-volume data loading scenarios, SQL Server implements minimally logged operations. Unlike fully logged operations, which use the transaction log to keep track of every row change, minimally logged operations keep track of extent allocations and metadata changes only. Because much less information is tracked in the transaction log, a minimally logged operation is often faster than a fully logged operation if logging is the bottleneck. Furthermore, because fewer writes go the transaction log, a much smaller log file with a lighter I/O requirement becomes viable.

Understand that an operation can be a bulk load operation without being minimally logged. For example, you can bulk load data into clustered indexes or even heaps without getting minimal logging. Minimal logging typically provides an extra speed benefit, but even without the minimal logging, bulk load has less overhead than traditional row by row inserting of data.

Contrary to the SQL Server myths, a minimally logged operation *can* participate in a transaction. Because all changes in allocation structures are tracked, it is possible to roll back minimally logged operations.

Minimally logged operations are available only if your database is in bulk-logged or simple recovery mode. For more information, see "Operations That Can Be Minimally Logged" (http://msdn.microsoft.com/en-us/library/ms191244.aspx). Note that performing a bulk operation in a bulk-logged database has impact on the backup strategy for the database. For more information about the implications, see Backup Under the Bulk-Logged Recovery Model [http://msdn.microsoft.com/en-us/library/ms190692.aspx] (http://msdn.microsoft.com/en-us/library/ms190692.aspx).

Trace Flag 610

SQL Server 2008 introduces trace flag 610, which controls minimally logged inserts into indexed tables. The trace flag can be turned on by using one of the following methods;

- Adding to the SQL Server startup parameters.
    - For more information, see (http://msdn.microsoft.com/en-us/library/ms345416.aspx) in SQL Server Books Online.

- Running
    - This enables the trace flag for a specific session. This is useful if you want to enable 610 for only a subset of load scenarios on the instance, and it applies only to the Transact-SQL connection that issues it.
    - Use turns on the trace flag for all connections to the server until it is turned off or until the next server restart.
    - For more information about using DBCC to enable trace flags, see (http://msdn.microsoft.com/en-us/library/ms187329.aspx) in SQL Server Books Online.

Before you start using this trace flag, be aware of the limitation described in the previous section.

Not every row inserted in a cluster index with trace flag 610 is minimally logged. When the bulk load operation causes a new page to be allocated, all of the rows sequentially filling that new page are minimally logged. Rows inserted into pages that are allocated before the bulk load operation occurs are still fully logged, as are rows that are moved as a result of page splits during the load. This means that for some tables, you may still get some fully logged inserts.

If trace flag 610 causes minimal logging to occur, you should generally see a performance improvement. But as always with trace flags, make sure you test for your specific environment and workload.

Consider these two examples:

Example 1: You have a table clustered in a key that contains even integer key values 0-16. The table has four leaf pages, the pages are not full, and they can hold two more rows on each page.

You bulk load eight new rows, with uneven key values 115. The new rows fit in the existing pages. The illustration below shows how this table will look before and after the load operation.

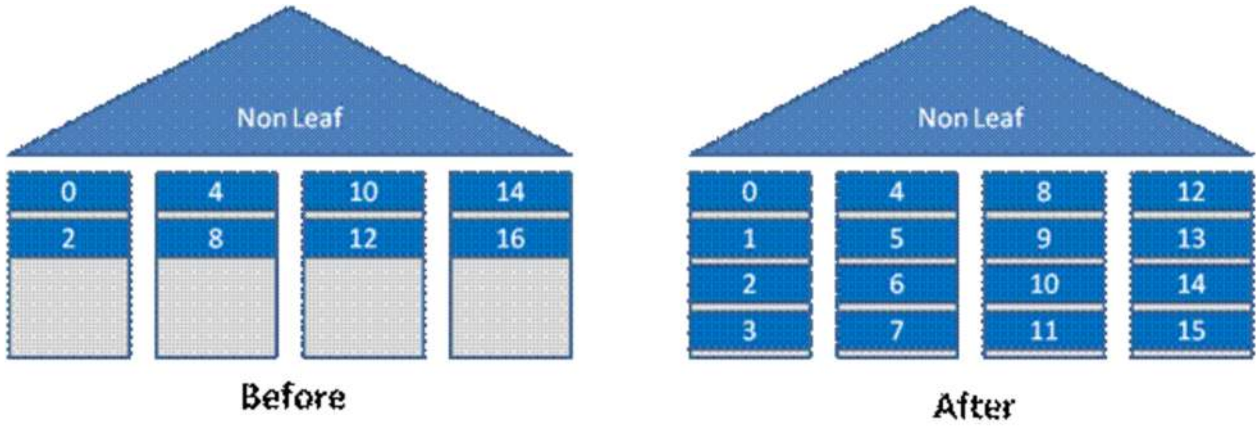


Figure 1: A fully logged insert under trace flag 610

In this example, no new pages are allocated and trace flag 610 will not give you any minimal logging.

Example 2: Consider an alternative scenario: The table initially now has two pages, both full, containing the key values 0-7. You bulk load rows with key values 8-16.

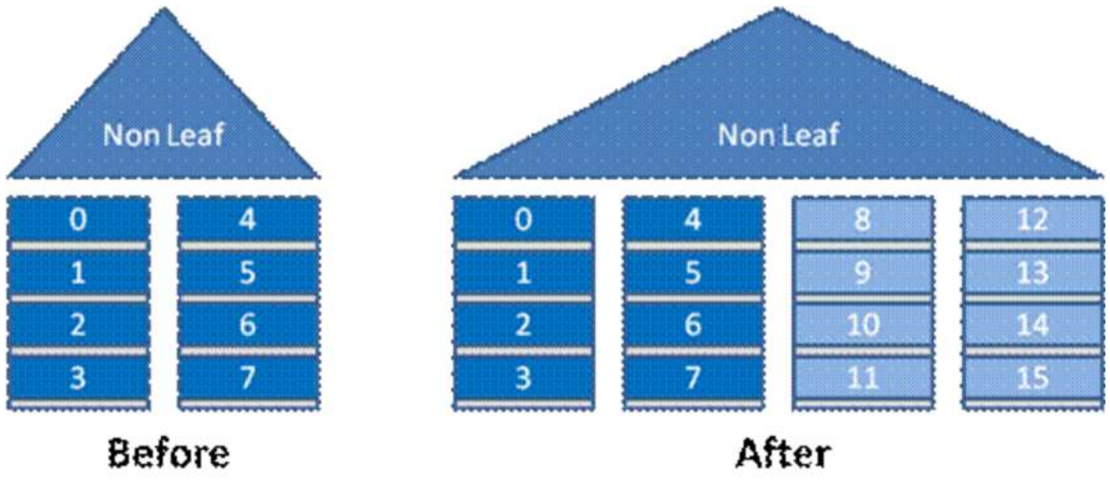


Figure 2: A minimally logged insert under trace flag 610

In this example, the pages holding key values 8-15 (in light blue above) will be minimally logged with trace flag 610.

One way to check how many new pages are allocated to a table is to query **sys.dm_db_partition_stats**. The following query will list the number of pages in each index and table.

Transact-SQL Copy Code

```
SELECT OBJECT_NAME(p.object_id) AS object_name
       , i.name AS index_name
       , ps.in_row_used_page_count
FROM sys.dm_db_partition_stats ps
JOIN sys.partitions p
       ON ps.partition_id = p.partition_id
JOIN sys.indexes i
       ON p.index_id = i.index_id
       AND p.object_id = i.object_id
```

By comparing the output before and after you run the bulk load operation, you can see how many new pages were allocated.

If you are using trace Flag 610 and populating btrees, it is a best practice to use the largest BATCHSIZE possible. SQL Server might allocate a significant number of new pages, *per batch*, per partition being populated. And under some circumstances this allocation activity, even if pages are deallocated later, can actually cause more overall I/O activity under trace flag 610 than without the trace flag. However, when you are loading a small number of partitions and utilizing a large batch size, trace flag 610 can provide substantial increase in throughput compared to fully logged inserts. For more information about minimal logging behavior under trace flag 610, see the SQL Server Storage Engine blog [http://blogs.msdn.com/sqlserverstorageengine/archive/2008/10/24/new-update-on-minimal-logging-for-sql-server-2008.aspx] (http://blogs.msdn.com/sqlserverstorageengine/archive/2008/10/24/new-update-on-minimal-logging-for-sql-server-2008.aspx).

I/O Impact of Minimal Logging Under Trace Flag 610

When you commit a bulk load transaction that was minimally logged, all of the loaded pages must be flushed to disk before the commit completes. Any flushed pages not caught by an earlier checkpoint operation can create a great deal of random I/O. Contrast this with a fully logged operation, which creates sequential I/O on the log writes instead and does not require loaded pages to be flushed to disk at commit time.

If your load scenario is small insert operations on btrees that do not cross checkpoint boundaries, and you have a slow I/O system, using minimal logging can actually slow down insert speeds.

Summarizing Minimal Logging Conditions

To assist you in understanding which bulk load operations will be minimally logged and which will not, the following table lists the possible combinations.

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

- If you are using the 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 the order hint must be used.
- Concurrent loads only possible under certain conditions. See "". Also, only rows written to newly allocated pages are minimally logged.
- Depending on the plan chosen by the optimizer, the nonclustered index on the table may either be fully- or minimally logged.

Bulk Load Methods

To provide fast data insert operations, SQL Server ships with several of standard bulk load methods. This section discusses the following methods in detail:

- **Integration Services Data Destinations**
- **BCP**
- **BULK INSERT**
- **INSERT … SELECT**
- **SELECT INTO**

In this paper , the term "bulk load" refers to the use of any of the methods described in this section. The term "BULK INSERT" (in upper case) refers to the specific Transact-SQL based bulk load method described under "BULK INSERT".

Choosing among these methods requires an understanding of their abilities and limitations. In this chapter, we will briefly describe the choices you have for bulk loading data both from outside of the SQL Server Database Engine and inside the engine itself.

In addition, similar bulk load techniques are supplied by programming interfaces to SQL Server, including the **SQLBulkCopy** class in ADO.NET, **IRowsetFastload** in OLE DB, and the SQL Server Native Client ODBC library. While these topics are beyond the scope of this white paper, the best practices around using the programming interfaces, along with many of their available settings and options, match those techniques discussed for the methods covered by this paper.

Integration Services Data Destinations

SQL Server Integration Services provides the most flexible choice for bulk loading data into SQL Server. Data can be read from any data source that is compatible with Integration Services, transformed and converted in memory, and bulk loaded directly into SQL Server without staging it to disk. Because Integration Services is a separate process, potentially on another computer, you can offload much of the CPU-intensive transformation work from SQL Server and move this into Integration Services. This allows you to scale out the bulk operation, allowing for very high throughput.

Two different data destinations both provide minimally logged, bulk load capabilities into SQL Server: The native format, SQL Server destination and the general format, OLE DB destination.

SQL Server Destination

The SQL Server destination is the fastest way to bulk load data from an Integration Services data flow to SQL Server. This destination supports all the bulk load options of SQL Server – except ROWS_PER_BATCH.

Be aware that this destination requires shared memory connections to SQL Server. This means that it can only be used when Integration Services is running on the same physical computer as SQL Server.

OLE DB Destination

The OLE DB destination supports all of the bulk load options for SQL Server. However, to support ordered bulk load, some additional configuration is required. For more information, see "Sorted Input Data". To use the bulk API, you have to configure this destination for "fast load".

The OLE DB destinationcan use both TCP/IP and named pipes connections to SQL Server. This means that the OLE DB destination, unlike the SQL Server destination, can be run on a computer other than the bulk load target. Because Integration Services packages that use the OLE DB destination do not need to run on the SQL Server computer itself, you can scale out the ETL flow with workhorse servers.

BCP

BCP (Bulk Copy Program) is the command-line tool used to both extract from and import into SQL Server. The tool is built using the bulk API and allows you to quickly insert data from text files directly into SQL Server. Additionally, BCP makes it possible to export data from SQL Server tables or queries into text files.

BCP can read the SQL Server native format from text files. This is a very fast option that requires minimal parsing of the text file input.

BULK INSERT

The BULK INSERT command is the in-process method for bringing data from a text file into SQL Server. Because it runs in process with Sqlservr.exe, it is a very fast way to load data files into SQL Server.

BULK INSERT cannot be used to export data, only to import it. But apart from this limitation, it has the same abilities as BCP. BULK INSERT is invoked from Transact-SQL, which makes it ideally suited for use in stored procedures, Transact-SQL based ETL, and SQL Server Agent jobs.

SELECT INTO

The SELECT INTO statement produces a new table based on the result of a SELECT statement. The newly created rows are minimally logged, providing a very fast way to load new data. There is no way to control BATCHSIZE or ROWS_PER_BATCH for this statement.

Another limitation of SELECT INTO is that the target table must reside on the default filegroup. Although you can temporarily change the default filegroup while the SELECT INTO statement is running, this may not be viable in your scenario.

INSERT … SELECT

SQL Server 2008 introduces a new way to perform minimally logged insert operations, allowing Transact-SQL based INSERT statements to be minimally logged under certain circumstances.

There are some limitations to this feature. First of all, none of the bulk load parameters, like commit size, check constraints, and trigger firing can be used with this method. Secondly, the target table is locked with an exclusive (X) lock, while the other bulk load methods (Integration Services, BCP, and BULK INSERT) described here issue the bulk update (BU) lock. Because exclusive (X) locks are not compatible with other exclusive (X) locks, only one insert can run at the same time. For more information about bulk update (BU) and exclusive (X) locks, see Lock Modes [http://msdn.microsoft.com/en-us/library/ms175519.aspx] (http://msdn.microsoft.com/en-us/library

/ms175519.aspx) in SQL Server Books Online.

Even though the insert operation itself is single threaded, INSERT … SELECT can still be used to drive high parallelism. The SELECT part of the statement is still fully parallelizable by the SQL Server Database Engine. Because this method can transform data in transit, operations bound by the speed of the SELECT statement can utilize the SQL Server Database Engine's optimizer to achieve high-speed Transact-SQL based transformations.

Whether an INSERT… SELECT operation into clustered indexes is minimally logged or not depends on the state of trace flag 610. INSERT…SELECT operations into heaps can be minimally logged even without trace flag 610.

Heaps

An INSERT statement that takes its rows from a SELECT operation and inserts them into a heap is minimally logged when a WITH (TABLOCK) hint is used on the destination table

The following syntax should be used to achieve minimal logging into heaps.

Transact-SQL Copy Code

```
INSERT INTO <DestinationTable> (<Columns>) WITH (TABLOCK)
SELECT <Columns> FROM <SomeStatement>
```

Note that the destination heap *need not be empty.* Coupled with fact that the destination table can reside in any filegroup (or even on a partition scheme), this technique offers more flexibility than SELECT INTO.

Clustered Indexes

If you perform an INSERT … SELECT operation into a clustered index, the operation is minimally logged if trace flag 610 is active. Note that the minimal logging is used even if the TABLOCK hint is not specified. This means that you can have multiple, concurrent INSERT … SELECT statements inserting into a table at the same time – all in minimally logged mode. There are some limitations to this feature. For more information, see "Bulk Loading with the Indexes in Place".

Additionally, a bulk load operation will be minimally logged even *without* the trace flag when *both* of the following conditions are true:

- A WITH (TABLOCK) hint is specified on the target table
- The target table is empty

In the above case, a schema modification (Sch-M) lock will be taken on the target, preventing any concurrency. This is a case of the ordered insert. For more information, see "Sorted Input Data".

Reading Text Files or OLE DB Data with INSERT … SELECT

It is possible to use the INSERT … SELECT command to read any OLE DB enabled data source into SQL Server. This is done by using the OPENROWSET command as the source SELECT statement.

By using OPENROWSET in this manner, you can mimic the behavior of BULK INSERT with INSERT…SELECT. You can even use this option to join or filter the incoming data prior to inserting.

Be aware that BULK INSERT allows you to run multiple, minimally logged streams into the same heap – INSERT…SELECT only allows one stream.

OPENROWSET also allows you to control many of the batch parameters for the insert destinations. The following parameters can be controlled when OPENROWSET is used as input for INSERT…SELECT:

- ORDER
- ROWS_PER_BATCH, but BATCH_SIZE
- IGNORE_CONSTRAINTS and IGNORE_TRIGGERS
- FIRSTROW and LASTROW
- KEEPDEFAULTS and KEEPIDENTITY

For more information, see OPENROWSET [http://msdn.microsoft.com/en-us/library/ms190312.aspx] (http://msdn.microsoft.com/en-us/library/ms190312.aspx) in SQL Server Books Online.

Write Your Own

ODBC, OLE DB, .NET and DB-library for SQL Server all allow you to write your own bulk load method. A full description of the bulk load API and abilities is outside the scope of this paper.

For information about writing your own bulk load method, see the following:

- SqlBulkCopy Class (.NET Framework Class Library) [http://msdn.microsoft.com/en-us/library /system.data.sqlclient.sqlbulkcopy.aspx]
- Performing Bulk Copy Operations [http://msdn.microsoft.com/en-us/library/ms130809.aspx]
- Bulk-Copying Rowsets How-to Topics (OLE DB) [http://msdn.microsoft.com/en-us/library /ms403307.aspx]
- Performing Bulk Copy Operations (ODBC) [http://msdn.microsoft.com/en-us/library/ms130792.aspx]
- Bulk Copying with the SQL Server ODBC Driver How-to Topics (ODBC) [http://msdn.microsoft.com /en-us/library/ms403302.aspx]

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.

A Word About BATCHSIZE and ROWS_PER_BATCH

Two bulk load parameters need some additional clarification: BATCHSIZE and ROWS_PER_BATCH.

The first, 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. The BATCHSIZE parameter is typically used to gradually commit rows, to avoid a full restart of the batch if something goes wrong. However, be aware that the restart does not happen automatically. It is up to the application programmer to keep track of how many rows have already been inserted. Note that low values of BATCHSIZE cause additional overhead for committing transaction, which will impact performance. 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.

A nonzero BATCHSIZE is most useful when you are inserting into indexed tables. A SORT operation must be performed internally once for each index in each batch if the data is not ordered, so a smaller BATCHSIZE allows sorts to occur in memory rather than spilling to disk. However, this will also create more fragmentation in the resulting index, which can impact query performance if ordered range scans are common. Fragmentation while you are inserting to a clustered index can be avoided if the inserted data is presorted as described in "Sorted Input Data".

It is best to test a variety of BATCHSIZE settings for your particular scenario to see impact on load performance and resulting fragmentation. In addition, nonzero BATCHSIZE settings are useful if you are *not* using TABLOCK and want to prevent lock escalation. This is common in situations where you expect concurrent reader queries during a load, and don't want them to block. Even if you use other techniques to prevent lock escalation (by using ALTER TABLE, for example), a nonzero BATCHSIZE will be essential to preventing too much memory from being consumed by locks in a large load.

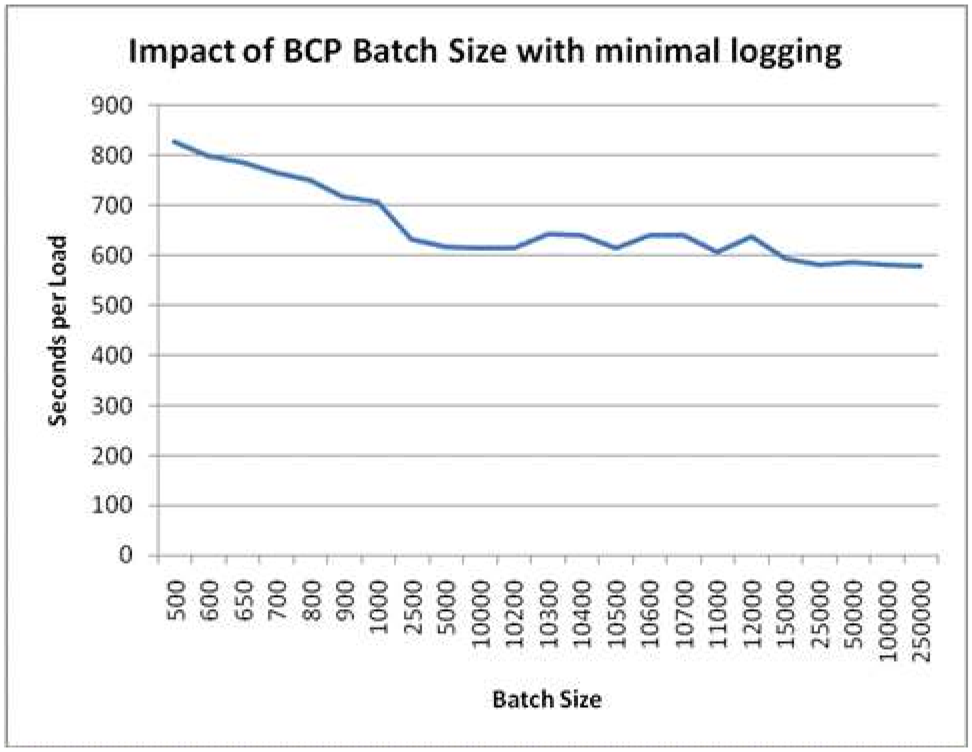


Figure 3: Impact of BCP Batch Size with minimal logging

The second parameter, 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 BATCHSIZEis 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.

The confusing issue about BATCHSIZE and ROWS_PER_BATCH is that they are referred to by different names, depending on which bulk method you use. The table below helps you translate.

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

Integration Services Batch Sizes

Integration Services does not handle batch sizes in the same way that the bulk load method does.

Integration Services will by default create one batch per pipeline buffer. After the buffer is flushed, the batch is committed. It is possible to override this behavior by changing the value of the **Maximum Insert Commit Size** in the Data Destination.

The following table summarizes the behavior of Maximum Insert Commit Size (MICS).

Maximum Insert Commit Size	**Effect**
MICS > buffer size	Setting is ignored. One commit is issued for every buffer.
MICS = 0	The entire batch is committed in one big batch. Behaves just like **BATCHSIZE** = 0.
MICS < buffer size	Commit is issued every time MICS rows are sent. Commits are *also* issued at the end of each buffer.

Note that the only way to get batch sizes larger than the buffer size is to change the size of the buffers in the properties of the data flow.

Data Compression and Bulk Load

When data is bulk loaded into a compressed table or partition, both page-level and row-level compression are generally done at bulk load time.

However, you should be aware of an exception: When bulk loading into a page compressed heap, you must use the TABLOCK hint to achieve page compression. If the TABLOCK hint is not used, only row level compression is done on the heap.

Using page compression on a bulk target will generally slow down bulk load speeds – especially if the I/O system is able to supply enough speed to saturate the CPUs.

The following illustrates the effect of compression on a 50M row table on a server that is not I/O bound:

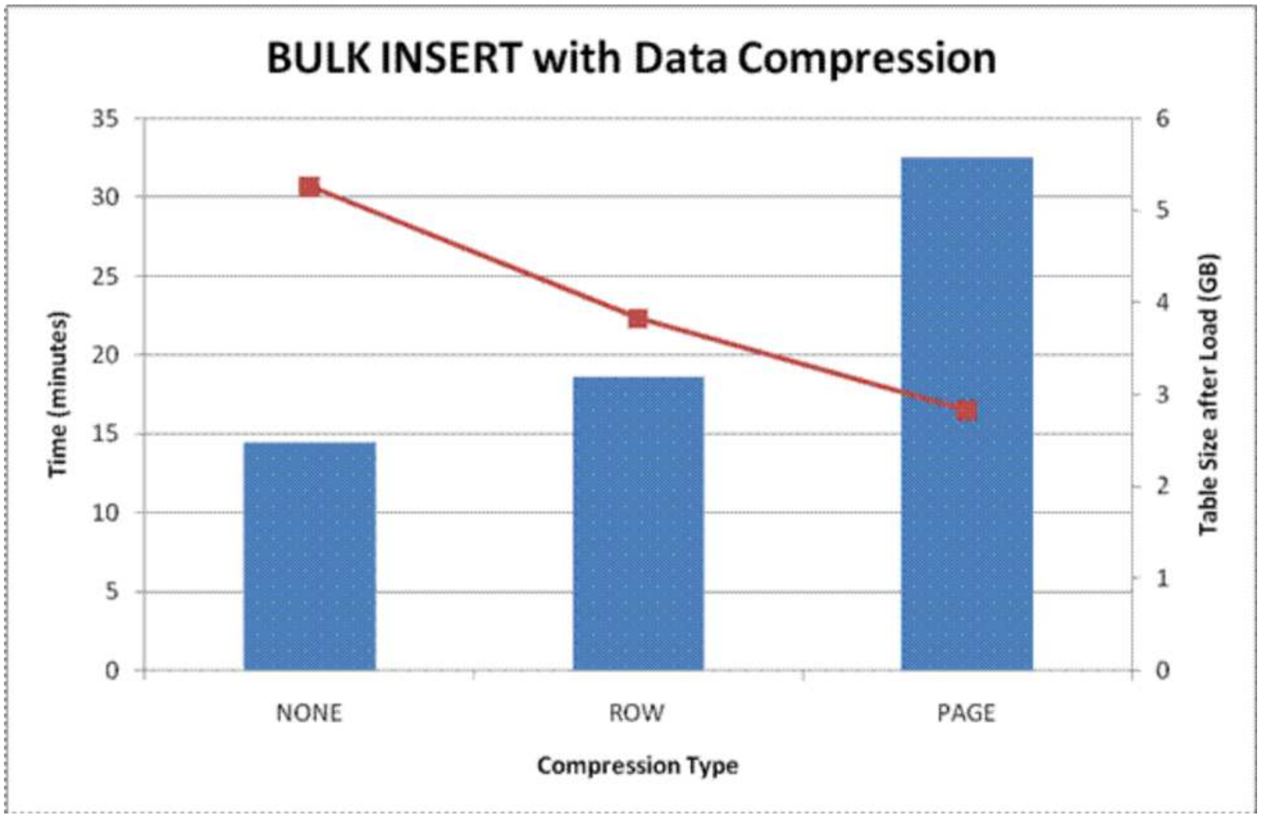


Figure 4: BULK INSERT with data compression

The red line shows the table size, and the blue bars show the time taken.

Other Minimally Logged and Metadata-Only Operations

In addition to insert operations, there exists a series of other operations that are minimally logged. When these other operations are combined with the insert statements, many data warehouse load scenarios can be implemented completely as minimally logged operations.

Although bulk load operations are often minimally logged, this is not always the case. For example, bulk loading data into a table that already has indexes and data is not minimally logged, even though it is still a fast bulk load. In this paper we will look at all types of bulk load, both the minimally logged and the fully logged.

In this section we will also describe metadata-only operations. These are operations that, like SWITCH and TRUNCATE, appear to change a lot of data, even though they only make small changes in the transaction log or data files.

DROP TABLE

The dropping of a table is a metadata-only operation. Dropping a table allows you to very quickly remove many rows without incurring the logging overhead of a DELETE statement.

MERGE

The Transact-SQL MERGE command offers minimal logging of the inserted rows. Unless you run MERGE on a heap, this minimal logging ability will only be used if trace flag 610 is enabled.

TRUNCATE TABLE

Just like DROP TABLE, TRUNCATE TABLE is a metadata-only operation. Be aware that certain limitations apply to TRUNCATE TABLE. These are:

- You cannot truncate a table that is referenced by foreign key constraints.
- You can truncate the entire table only, not a single partition (but see "Deleting All Rows from a Partition or Table").

CREATE INDEX, DROP INDEX, and REBUILD INDEX

The write operation executed when creating, dropping, or rebuilding an index is a minimally logged operation. Even though lots of data is written to the data file, only very little is written to the transaction log.

In the enterprise editions of SQL Server, the creation of indexes is parallelized. However, be aware that the parallel create of the index can cause data fragmentation.

Partition SWITCH

SQL Server 2005 introduced partitions and partition switching. Partition switching does not physically move the data; it is a metadata-only operation. It allows you to load many similar data streams in parallel into multiple tables. After this action is complete, issue the SWITCH command to create one, large table.

Updates to a table are not possible in minimally logged mode and neither are deletes on a subset of rows. However, as we will see in "Solving Typical Scenarios with Bulk Loading", the SWITCH command can be used to emulate some of these operations.

Creating a switch target that can be switched to and back into the main table can be complex. To use SWITCH, you must match the metadata and allocation of the table exactly. You need to perform the following operations to match the table metadata:

1. **Create a copy of the table, matching columns exactly.**

The obvious way to perform this is to use the CREATE TABLE statement to match the source table definition.

You can also use the following technique to clone the table definition.

Transact-SQL Copy Code

```
SELECT TOP(0) *
INTO <TargetTable>
FROM <SourceTable>
```

However, the newly created table will be allocated in the default filegroup. You can move the table by creating a clustered index on it and dropping the index again.

Transact-SQL Copy Code

```
CREATE CLUSTERED INDEX IX_temp
ON <TargetTable> (<Col>) ON [<FileGroup>]
DROP INDEX <TargetTable>.IX_Temp
```

1. **Match indexes on the target.**

After you have copied the source table definition, you need to match all indexes from the source on the target. The following index settings must be matched:

- Compression Settings
- Filegroup Allocation

1. **Create a constraint on the target.**

The last requirement is to create a constraint on the target. This constraint must match the requirements of the partition function of the target. At this point, you have to careful.

For the first partition in the table (with the lowest values), use the following template.

Transact-SQL Copy Code

```
ALTER TABLE <TargetTable>
ADD CONSTRAINT CK_<X>
CHECK (<PartCol> < <LeftValue>)
```

For the partitions between the first and the last, this is the format of the constraint.

Transact-SQL Copy Code

```
ALTER TABLE <TargetTable>
ADD CONSTRAINT CK_<X>
CHECK (<PartCol> IS NOT NULL
 AND <PartCol> >= <LeftValue>
 AND <PartCol> < <RightValue>)
```

Notice the IS NOT NULL here. NULL values are always part of the first partition, and because NULL is not comparable with NULL, you must explicitly have the constraint remove it.

Finally, for the last partition, you use the following template.

Transact-SQL Copy Code

```
ALTER TABLE <TargetTable>
ADD CONSTRAINT CK_<X>
CHECK (<PartCol> IS NOT NULL
 AND <PartCol> >= <LeftValue>)
```

The above examples all assume that the partition function uses the RANGE LEFT.

1. **Perform the SWITCH.**

You can now perform the SWITCH command itself.

Transact-SQL Copy Code

```
ALTER TABLE <SourceTable>
SWITCH PARTITION <X> TO <TargetTable>
```

To help you perform all of these above steps, the following CodePlex tool is available: SQL Server Partition Management Tool (CodePlex) [http://www.codeplex.com/sql2005partitionmgmt.aspx] (http://www.codeplex.com/sql2005partitionmgmt).

Solving Typical Scenarios with Bulk Loading

In this section, we will look at some typical load scenarios and provide the templates required to solve them. We will use an arbitrary **Sales** fact table as an example. The following scenarios will be discussed:

- Bulk loading an empty, nonpartitioned table
- Bulk loading into a nonpartitioned table that already has data
- Bulk loading a partitioned table
- Deleting all rows from a partition or table
- Deleting a large amount of rows in a partition or table
- Updating a large part of the data in a partition or table

In each scenario we will look at the tradeoffs you have to consider to maximize the speed of the operation.

In the examples we will use the BULK INSERT method for illustration purposes. However, the guidance applies for the other bulk load methods too.

Bulk Loading an Empty, Nonpartitioned Table

Loading data into a nonpartitioned table, while a simple operation, can be optimized in several ways.

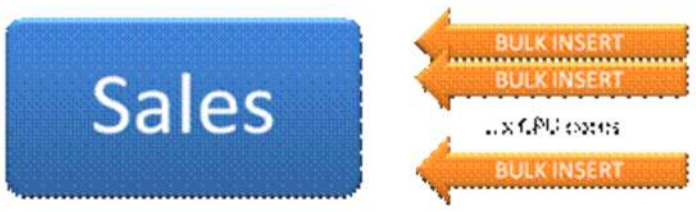
Before you load data into an empty table, it is always a good idea to drop or disable all indexes on the table. After the load is done, re-create or re-enable the indexes.

If your source data is a table inside SQL Server, using SELECT INTO or INSERT … SELECTprovides a quick and easy way to achieve this. If your source data is located outside of SQL Server, you can use BCP, BULKINSERT, INSERT…SELECT or Integration Services as appropriate.

Multiple, concurrent insert operations for heaps are possible only when the chosen bulk method issues bulk update (BU) locks on the table. Two bulk update (BU) locks are compatible, and hence two bulk operations can run at the same time.

If your I/O system is fast, consider using multiple bulk insert operations in parallel. A single bulk operation will only fully utilize one CPU core. As we will see in the other scenarios, utilizing multiple, concurrent bulk streams is the key to scaling bulk loading.

In this scenario, both INSERT … SELECT and SELECT INTO have a drawback. Both of these operations take an exclusive (X), table level lock on the destination. This means that only one bulk load operation can run at a given time, limiting scalability. However, BCP, BULK INSERT, and Integration Services are all capable of taking bulk update (BU) locks – if you specify the TABLOCK hint.

 Sales BULK INSERT BULK INSERT ...x CPU cores BULK INSERT	Perform multiple bulk load operations of the form: BULKINSERT Sales FROM'Sales<X>.csv' WITH (FIELDTERMINATOR=';' ,ROWTERMINATOR='\n' , TABLOCK)

Performing parallel bulk load into a single heap scales relatively well. Our performance testing from ETL World Record [http://blogs.msdn.com/sqlperf/archive/2008/02/27/etl-world-record.aspx] (http://blogs.msdn.com/sqlperf/archive/2008/02/27/etl-world-record.asp) gives this curve.

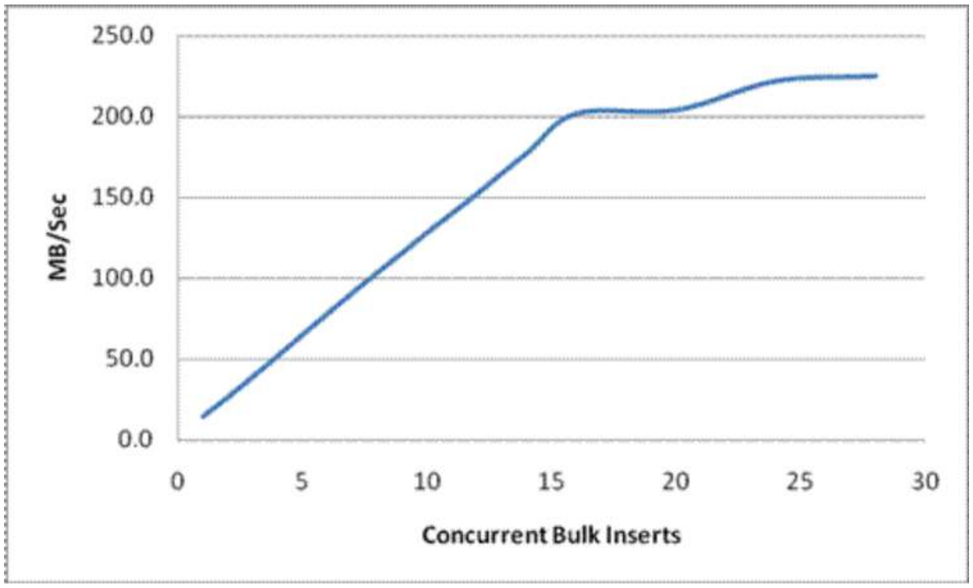


Figure 5: Bulk insert on single heap with a scalability curve

We achieve near linear scalability up to 16 concurrent bulk operations, depending on data sizes. After this, contention of the internal allocation data structure in SQL Server will start to become a bottleneck. This will show up as waiting for ALLOC_FREESPACE_CACHE in **sys.dm_os_latch_stats**. Note that these numbers vary depending on the size of the rows being inserted.

If you require linear scale beyond this point, partitioning the table may provide better throughput. Using partitions, you can concurrently load more than one table at the same time and recombine the tables using partition switching as described in "Bulk Loading a Partitioned Table".

Sorted Input Data

A special case needs mentioning here. If you ultimately require a clustered index on the destination, and the source data is ordered by the same key as the clustered index, you can specify the ORDER hint and keep the clustered index on the table while bulk loading. Using the ORDER hint will speed up bulk load significantly because it eliminates the internal sort step when loading a clustered index. However, only do this if the data is already ordered when it arrives from the source; sorting it before inserting will not increase speeds.

If you start from an empty table, the operation will be minimally logged. However, if there is already data in the table, the operation will be logged unless you use trace flag 610.

It is possible to specify the ORDER hint and TABLOCK at the same time. This will not allow you to run multiple streams into the same table (a schema modification (Sch-M) lock is taken). However, it WILL be much faster for a single stream – comparable with single stream bulk loading of heaps – and eliminate the need for a separate CREATE INDEXstep (involving an expensive sort) following the load.

For a single stream, the following bulk operation was comparable for both heaps and clustered indexed on **OrderDate**.

Transact-SQL Copy Code

```
BULK INSERT Sales FROM 'C:\temp\Sales200401'
WITH (
  FIELDTERMINATOR = ';'
  , ROWTERMINATOR = '\n'
  , TABLOCK
  , ORDER(OrderDate)
)
```

Similar performance was observed inserting into a clustered index and heap in this case. Note that unless you enable trace flag 610, the only way such an operation will be minimally logged is if the target table is empty *and* you specify a BATCHSIZE = 0. For a nonzero BATCHSIZE, only the first batch (commit) will be minimally logged.

Inserting sorted input data into a clustered index minimized fragmentation. However, sorting data can consume a lot of memory. If you want to minimize fragmentation while at the same time minimizing the memory footprint of the necessary sort, you can use the following trick:

1. Create a temporary table with same schema as the final target.
2. Ensure that the temporary table has a clustered index that matches the index of the final target.
3. Bulk load data into the staging table. Optionally, use trace flag 610 if this demonstrates a performance benefit.
    - The bulk loaded data will now reside in the staging table. It will be sorted, but potentially fragmented.
4. Use the INSERT … SELECT bulk load method to move data from the staging table to the final target.
    - Because the staging table is already sorted, the data will arrive presorted to the final target
    - This will keep fragmentation to a minimum.

The above trick will consume more I/O than just loading the data directly to the target. However, fragmentation will be kept to a minimum, which can increase read speeds for workloads heavy on range and table scans.

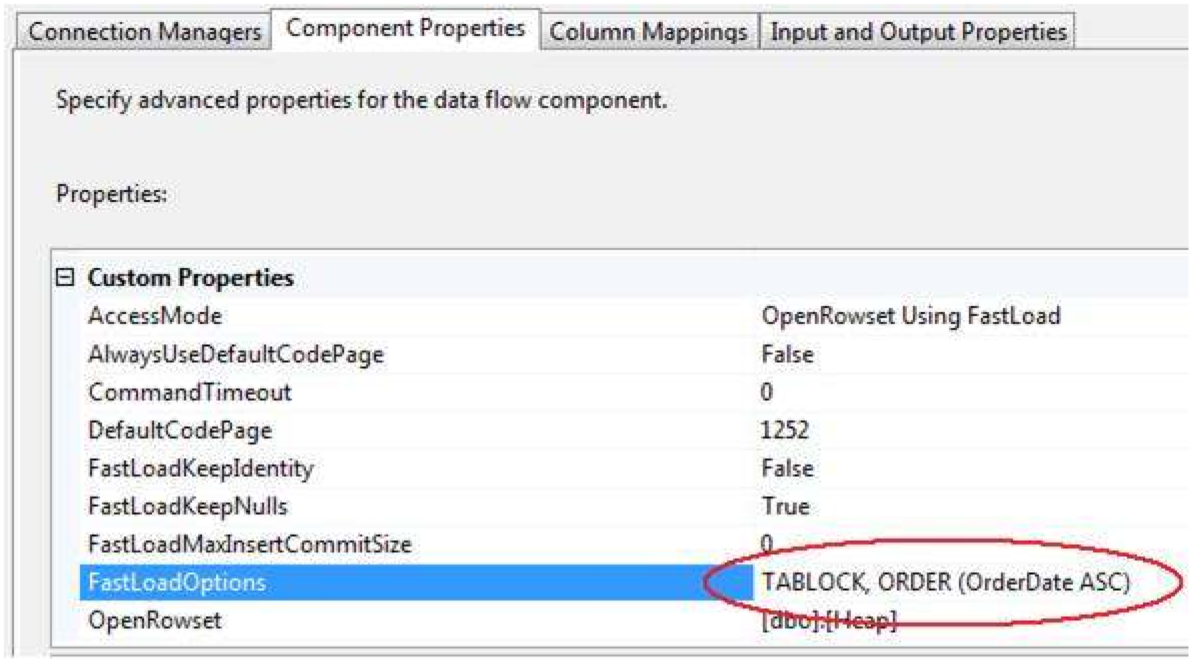
Note that when bulk loading a clustered index using small batches that can potentially fit in memory, SQL Server estimates sort memory needed by assuming that on average, variable-length columns are populated at half-capacity. If data sizes for any batch are larger than the memory estimate, the sort will spill to disk and degrade performance. You can detect sort spill events by using SQL Server Profiler to track the Sort Warning event class.

If, instead of the half-capacity estimate, your **varchar** columns tend to be populated closer to full length, you can help SQL Server avoid spilling sorts to disk by using the following workaround:

When creating the temporary table noted in step 1 above, increase the size of the variable-length columns by a factor of 3. Do not change the column length settings of the permanent table. This will increase the memory allocated for the sort step to more than enough needed.

Configuring the Integration Services OLE DB Destination for Sorted Input

While the GUI does not directly expose it, it is possible to configure the OLE DB Destination in Integration Services to support a sorted input stream. This is done from the advanced properties of the component.



Specify advanced properties for the data flow component.

Properties:

Custom Properties

Property	Value
AccessMode	OpenRowset Using FastLoad
AlwaysUseDefaultCodePage	False
CommandTimeout	0
DefaultCodePage	1252
FastLoadKeepIdentity	False
FastLoadKeepNulls	True
FastLoadMaxInsertCommitSize	0
FastLoadOptions	TABLOCK, ORDER (OrderDate ASC)
OpenRowset	[dbo].[Heap]

Figure 6: Configuring OLE DB Destination for sorted input

Add the ORDER hint, as illustrated above, to the **FastLoadOptions** property.

Bulk Loading into a Nonpartitioned Table That Already Has Data

If the table you are loading contains no indexes, the guidance from the previous section applies. Be aware that an existing table cannot be loaded with the SELECT INTO method.

When you are loading into a table that has indexes, take special care. First, see Guidelines for Optimizing Bulk Import [http://msdn.microsoft.com/en-us/library/ms177445.aspx] (http://msdn.microsoft.com/en-us/library/ms177445.aspx) in SQL Server Books Online. This topic contains guidelines on when to drop indexes before bulk loading to the table. However, be aware that the guidelines in SQL Server Books Online apply only for single-stream bulk load operations. As concurrency increases and number of cores goes up, dropping indexes, concurrently reloading them, and *then* rebuilding indexes can be faster than inserting the data in the existing indexes. Also, be aware that bulk loading into a clustered index is only minimally logged if the table is empty or if trace flag 610 is utilized. This means that multiple, concurrent bulk loads are not minimally logged.

You should test both scenarios: Dropping and re-creating the indexes and loading data with the indexes in place. If you choose to use the concurrent reload of the entire table, the guidance from the previous section applies.

Bulk Loading with the Indexes in Place

Bulk loading data to a table that already has indexes is a very different operation than bulk loading to heaps without indexes.

When you bulk load into table with one or more indexes, you should consider the following.

Indexed Tables Do Not Support Bulk Update (BU) Locks

Tables with indexes do not support the bulk update (BU) lock – instead exclusive (X) locks are taken. Exclusive (X) locks, unlike bulk update (BU) locks, are not compatible with other locks of the same type. Bulk loading into a table with indexes will take exclusive (X) locks at the row level. While it is possible to issue a TABLOCK hint when bulk loading to an indexed table, this just causes all the exclusive (X) locks to appear at the table level, blocking other bulk load threads into the same table. To achieve a concurrent bulk load into an indexed table, you should therefore *not* use the TABLOCK hint.

Find the Right BATCHSIZE

You should experiment with BATCHSIZE to maximize bulk throughput. While BATCHSIZE = 0 if useful for inserting into heap without indexes, it should be not used for tables that have indexes. Unless your input data is presorted (and you specify the ORDER hint), the optimizer will perform a sort operation on the input data before the batch is inserted. If this sort does not fit in memory, I/O activity is generated in **tempdb**, slowing down insert speeds. A smaller BATCHSIZE ensures that sorts occur in memory, but the drawback is that page splits and index fragmenting will occur if incoming data is in random order. The BATCHSIZE will also impact the ability to load data concurrently with queries against the same table.

Avoid Lock Escalation

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. It is possible to prevent this escalation from happening; the technique depends on the SQL Server version:

- For SQL Server 2005, lock escalation generally occurs at 5000 locked allocated. There are exceptions. See this
    - Setting a BATCHSIZE to a number lower than this will disable escalation
    - Alternatively, you can use trace flag 1211 to completely disable lock escalation on the server.
- For SQL Server 2008, it is also possible to completely disable lock escalation on a table level.

Use Nonoverlapping Input Streams

When running multiple bulk streams into a table with a clustered index, you should seek to have nonoverlapping ranges in the key column of your input data. Because the bulk load process uses row-level locks, two streams with overlapping values can block each other, decreasing concurrency.

Use Trace Flag 610

If trace flag 610 is enabled, inserts into a table with indexes will generally be a minimally logged operation. Tables with clustered indexes support multiple bulk load stream inserting concurrently, as long as these streams contain nonoverlapping data. If data overlap is detected, streams will block (but not deadlock).

The following example on the **Sales** table illustrates how you can achieve maximum, multistream throughput into a clustered index with BULK INSERT.

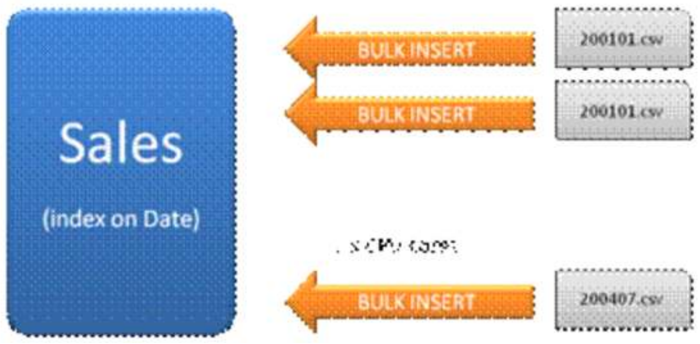
1. **Disable lock escalation on the table.**

Transact-SQL Copy Code

```
ALTER TABLE Sales SET (LOCK_ESCALATION = DISABLE)
```

In SQL Server 2005 you can disable lock escalation by setting BATCHSIZE to 5000. Alternatively, you can run with trace flag 1211 (which completely disables lock escalation on the server). For more information about trace flag 1211, see Lock Escalation in SQL2005 [http://blogs.msdn.com/sqlserverstorageengine/archive/2006/05/17/Lock-escalation.aspx] (http://blogs.msdn.com/sqlserverstorageengine/archive/2006/05/17/Lock-escalation.aspx).

1. **Start multiple BULK INSERT commands with nonoverlapping ranges.**



Partition input data with nonoverlapping ranges in the **Date** column:

```
BULKINSERT Sales
FROM'200101.csv'
WITH (
 FIELDTERMINATOR=';'
 ,ROWTERMINATOR='\n'
)
BULKINSERT Sales
```

	FROM'200102.csv' WITH (FIELDTERMINATOR=';' ,ROWTERMINATOR='\n') *… etc…* Do *not* issue the TABLOCK hint for the individual streams.

Test with different BATCHSIZE values to see where optimal throughput is reached, given your I/O subsystem and memory resources.

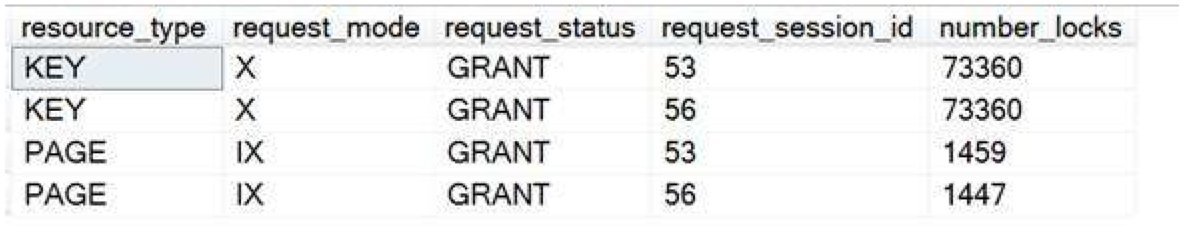
3) Check locking and concurrency

You should now see multiple, nonblocked bulk load streams. Each of these streams should hold row-level exclusive (X) locks. Try a query like the one below.

Transact-SQL Copy Code

```
SELECT lck.resource_type, lck.request_mode
       , lck.request_status , lck.request_session_id
       , COUNT(*) number_locks
FROM sys.dm_tran_locks lck
INNER JOIN sys.partitions par
       ON lck.resource_associated_entity_id = par.hobt_id
INNER JOIN sys.objects obj
       ON par.object_id = obj.object_id
WHERE obj.name = 'Sales'
GROUP BY lck.resource_type, lck.request_mode
       , lck.request_status, lck.request_session_id
```

This returns the following.



resource_type	request_mode	request_status	request_session_id	number_locks
KEY	X	GRANT	53	73360
KEY	X	GRANT	56	73360
PAGE	IX	GRANT	53	1459
PAGE	IX	GRANT	56	1447

Figure 7: Listing the locks taken by a bulk load operation

You can see the X locks on the rows (as indicated by KEY under **resource_type**) and the intent exclusive (IX) locks on the newly allocated data pages.

Bulk Loading a Partitioned Table

Bulk loading into partitioned tables provides the fastest possible load option. It allows allowing many concurrent bulk load operations to be executed in parallel with minimum coordination overhead. Switching in partitions once the data loading is done as a metadata-only operation.

When you are bulk loading a partitioned table, it is generally best to switch out the partition you want to work on before starting the bulk operation. Switching out has some strong advantages:

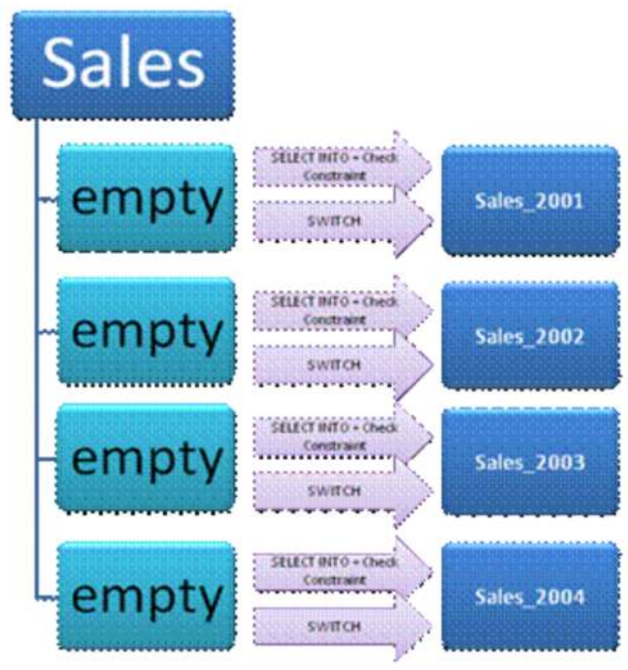
- It is possible to take bulk update (BU) locks on the switched-out object alone and have several bulk streams operating on the same time.
- You can drop and rebuild the indexes on the switched-out partition before switching it back in. This can increase bulk load speed.

- 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.

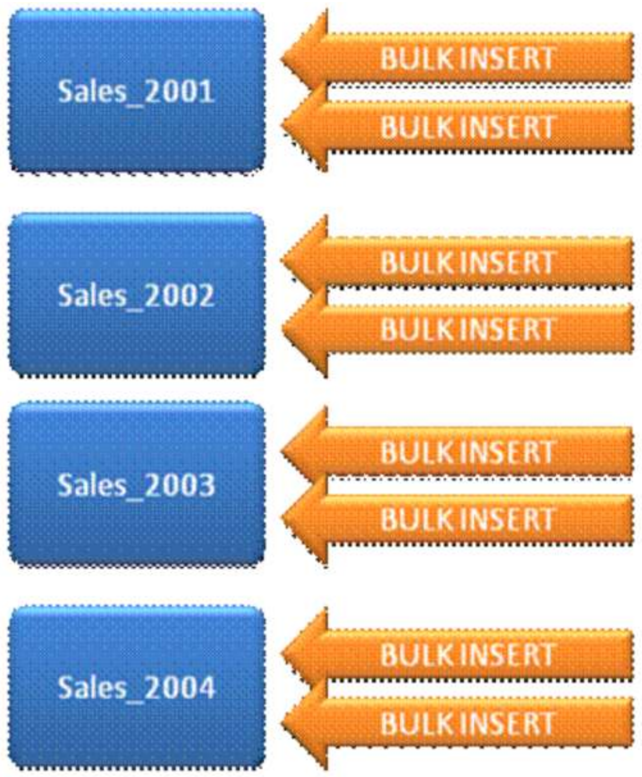
After a partition has been switched out, it behaves exactly like a normal table and all the guidance from the previous sections apply. However, do remember to add a constraint matching the target partition (and enable CHECK CONSTRAINTS in the load options) before starting the bulk load operation. Alternatively, add the check constraint after loading the table, although this will require a table scan. If you do not add the constraint, you will not be able to switch the partition back in.

However, there is a special case you should consider: If you plan to insert data into several partitions at the same time, you can optimize further. In this case, you switch out each partition into its own table. On each of these tables you then use one or more concurrent bulk load operations. This method is illustrated below:

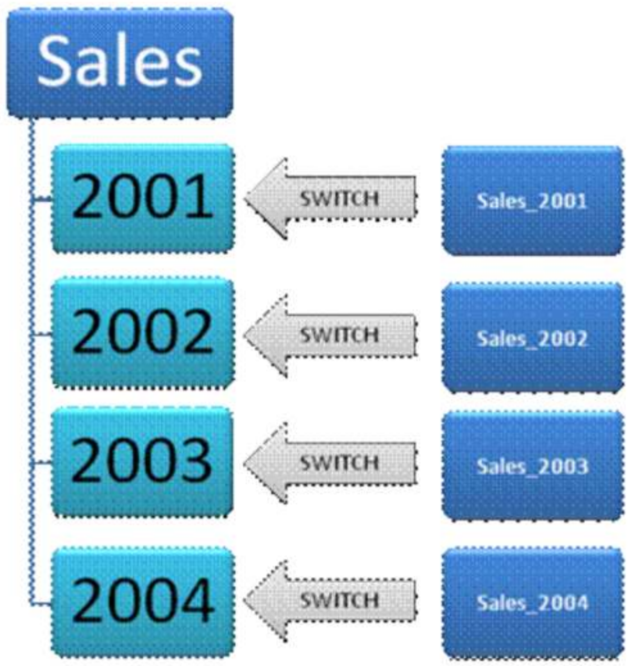
1. **Create temporary tables for staging.**

 Sales empty SELECT INTO - Check Constraint SWITCH Sales_2001 empty SELECT INTO - Check Constraint SWITCH Sales_2002 empty SELECT INTO - Check Constraint SWITCH Sales_2003 empty SELECT INTO - Check Constraint SWITCH Sales_2004	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>

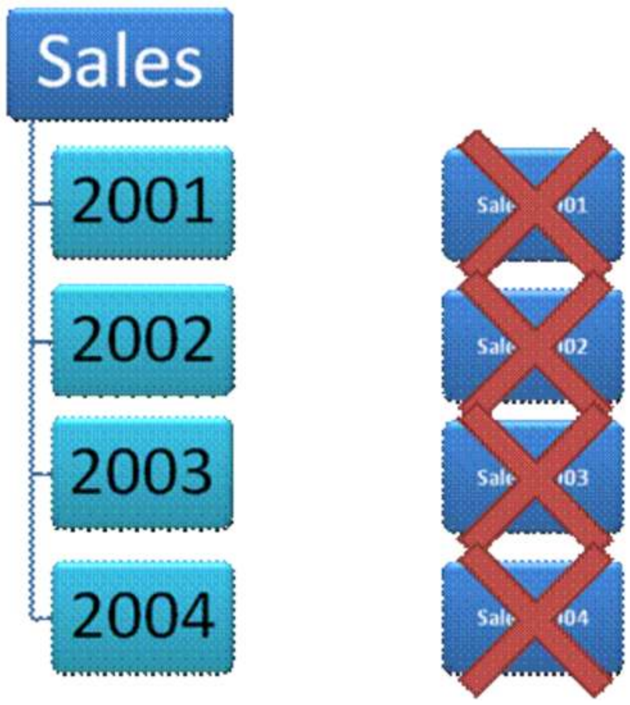
1. **Apply bulk load optimization on each individual table.**

 Sales_2001 Sales_2002 Sales_2003 Sales_2004 BULK INSERT	BULKINSERT Sales_200<X> FROM'Sales200<X><Y>.csv' WITH (FIELDTERMINATOR=';' ,ROWTERMINATOR='\n') …etc…

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

 Sales 2001 2002 2003 2004 SWITCH Sales_2001 Sales_2002 Sales_2003 Sales_2004	Switch back the staging tables: ALTERTABLE Sales_200<X> SWITCH TO Sales_P PARTITION<X>

1. **Clean up the staging tables**

 Sales 2001 2002 2003 2004 Sale 01 Sale 02 Sale 03 Sale 04	Drop the temporary staging tables: DROPTABLE Sales_200<X>

Summarizing Insert Scenarios

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

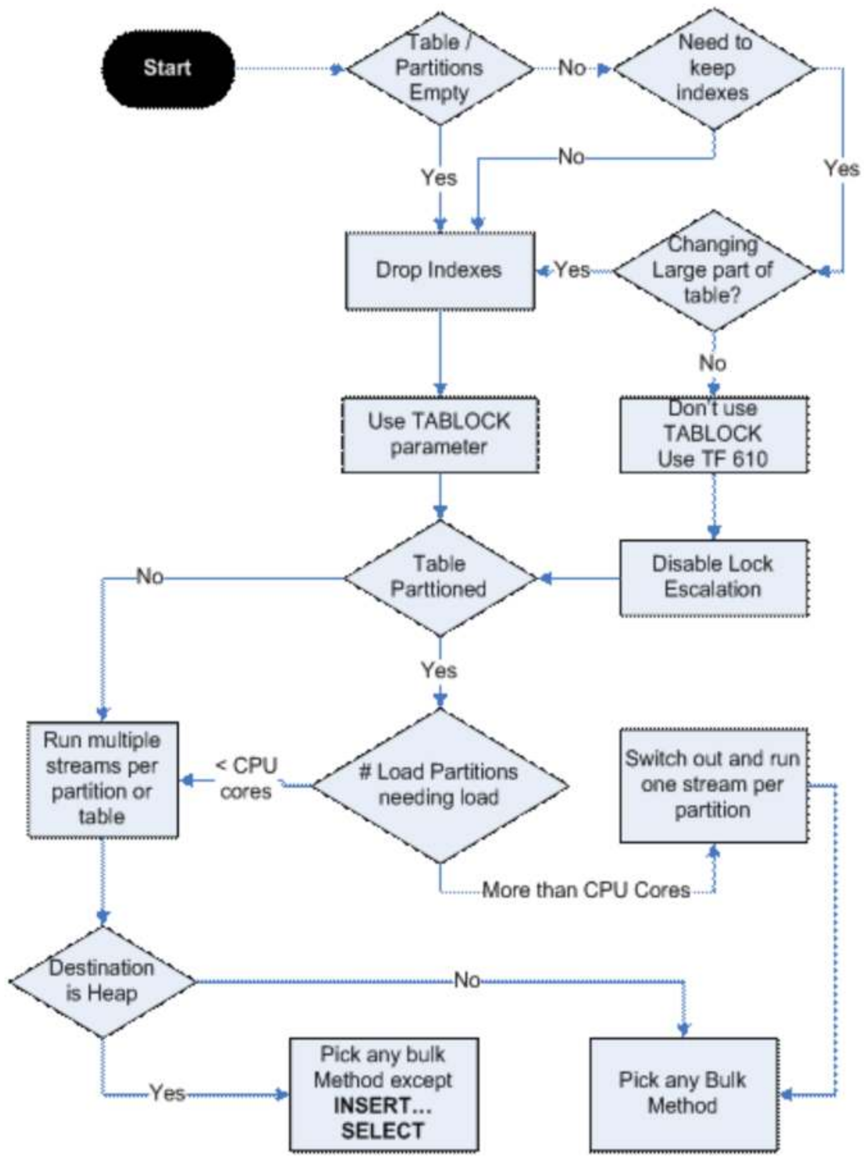


Figure 8: Bulk load method decision flowchart

Deleting All Rows from a Partition or Table

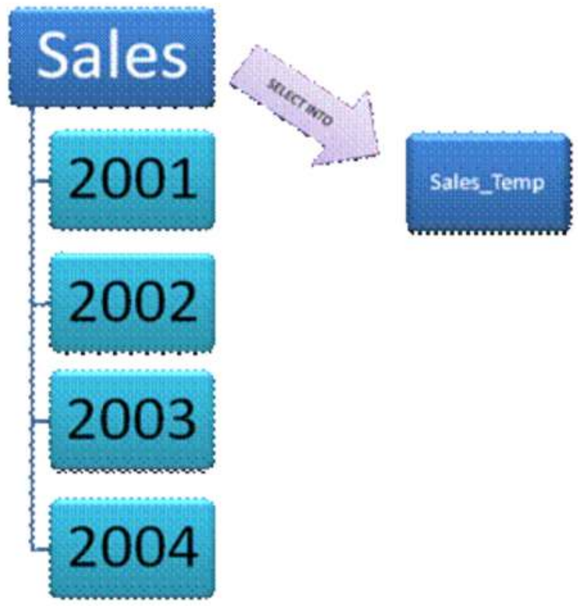
This scenario, while simple, is often overlooked. The fastest way to remove all data from a table is to simple execute a truncate statement.

Transact-SQL Copy Code

```
TRUNCATE TABLE Sales
```

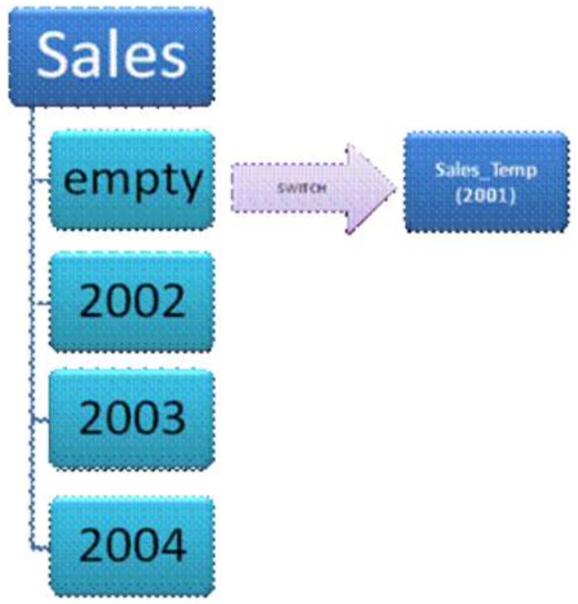
Removing data from a partition is only slightly more complicated:

1. **Create a temporary, empty table to hold the partition data.**

 Sales 2001 2002 2003 2004 SELECT INTO Sales_Temp	Create a temporary table as switch target. The method to perform this is described in “Partition SWITCH”.

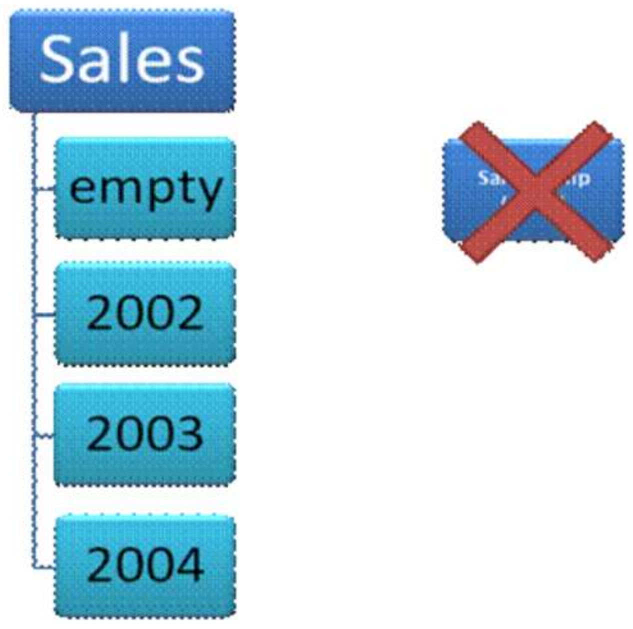
If the source table has indexes, you should also re-create these indexes on the temporary table.

1. **SWITCH the partition you want to delete to the temporary table.**

 Sales empty 2002 2003 2004 SWITCH Sales_Temp (2001)	ALTERTABLE Sales_P SWITCH PARTITION 1 TO Sales_Temp

The source partition in now empty.

1. **DROP or TRUNCATE the temporary table.**

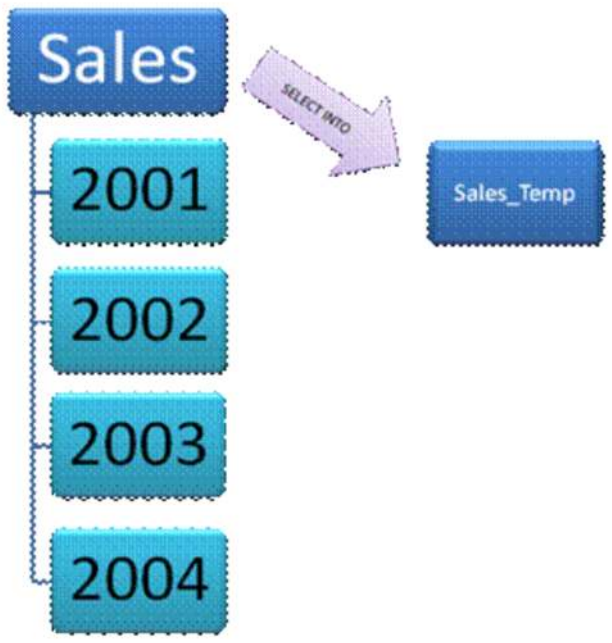
	If you no longer need the temporary table, delete it: DROPTABLE Sales_Temp Alternatively, truncate the table to reuse it for deleting more partitions: TRUNCATE TABLE Sales_Temp

Deleting a Large Amount of Rows in a Partition or Table

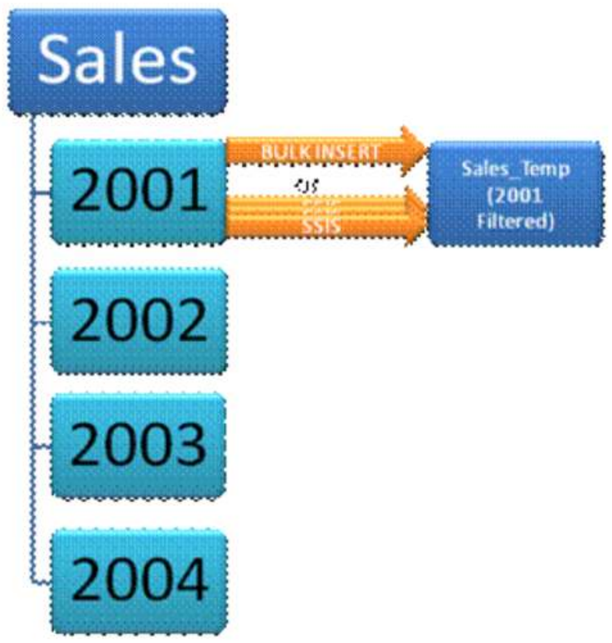
So far, we have seen how we can insert and to certain extend delete data in minimally logged mode and/or bulk load mode.

There are cases when you don't want to use TRUNCATE on a full partition but still want to remove a large number of the rows in it. The DELETE statement will be fully logged. However, you can achieve an effect that is similar to that of DELETE with bulk load. This is described below.

1. **Create a copy of the main table.**

 Sales SELECT INTO Sales_Temp 2001 2002 2003 2004	Create a temporary table as switch target. The requirements are described in "Partition SWITCH".

1. **Bulk load the rows you want to keep into the temporary partition.**

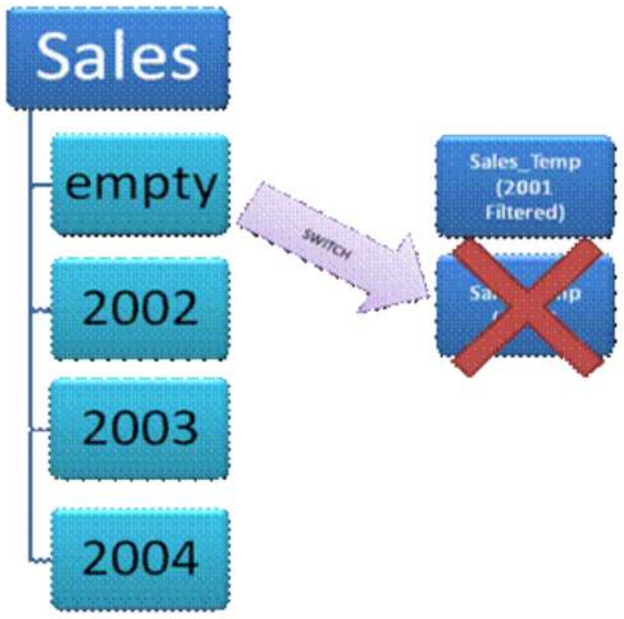


To perform the bulk insert, you can use one of two methods. The INSERT … SELECT method:

```
INSERTINTO Sales_Temp WITH (TABLOCK)

SELECT*FROM Sales

WHERE OrderDate >='20010101'

  AND OrderDate <'20020101'

  AND<Keep Criteria>
```

Alternatively, you can use Integration Services to achieve many concurrent, streams into the **Sales_Temp** table.

1. **Switch out and truncate the old data**

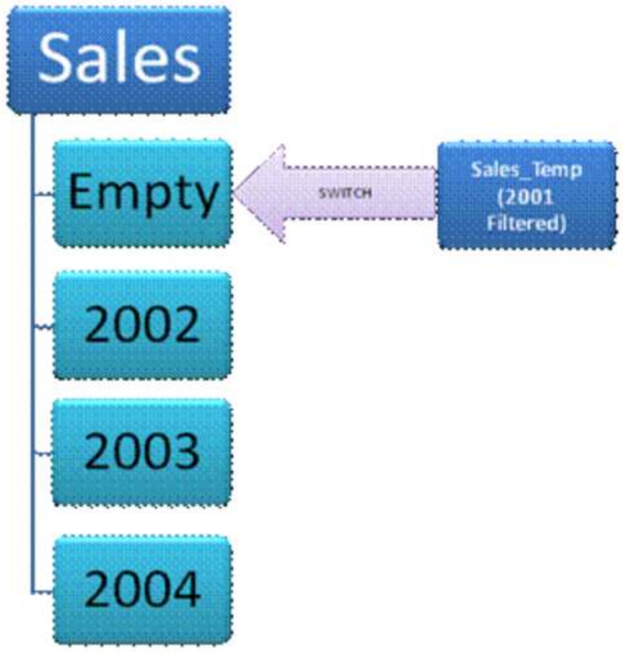


If you want to make the operation transactional, start a transaction now:

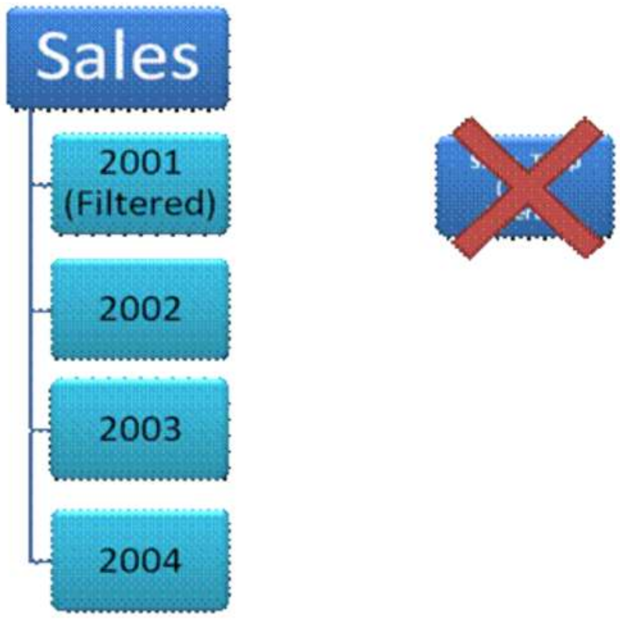
```
BEGINTRAN
```

Use the technique described in "Deleting All Rows from a Partition or Table" to remove the rows from the main table.

1. **Rebuild indexes and use SWITCH.**

 Sales Empty SWITCH Sales_Temp (2001 Filtered) 2002 2003 2004	Rebuild indexes and constraints on **Sales_Temp** to match the main table. The partition management tool described in "Partition SWITCH" will assist you with this task. Switch in **Sales_Temp:** ALTERTABLE Sales_Temp TO Sales SWITCH PARTITION 1

1. **Drop the temporary table.**

	Drop the temporary table: DROPTABLE Sales_Temp If you used a transaction, commit it now: COMMIT TRAN

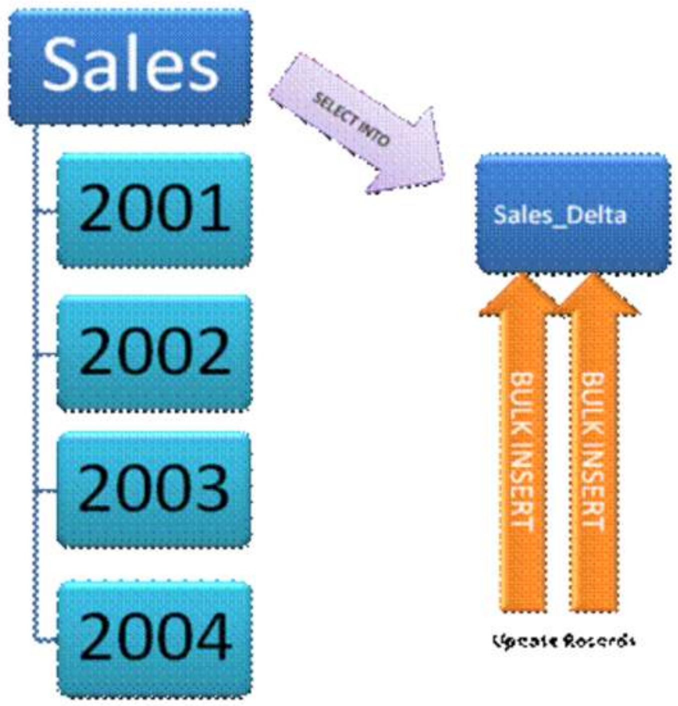
Updating a Large Part of the Data in a Partition or a Table

So far, we have seen how we can insert and to a certain extent delete data in minimally logged mode.

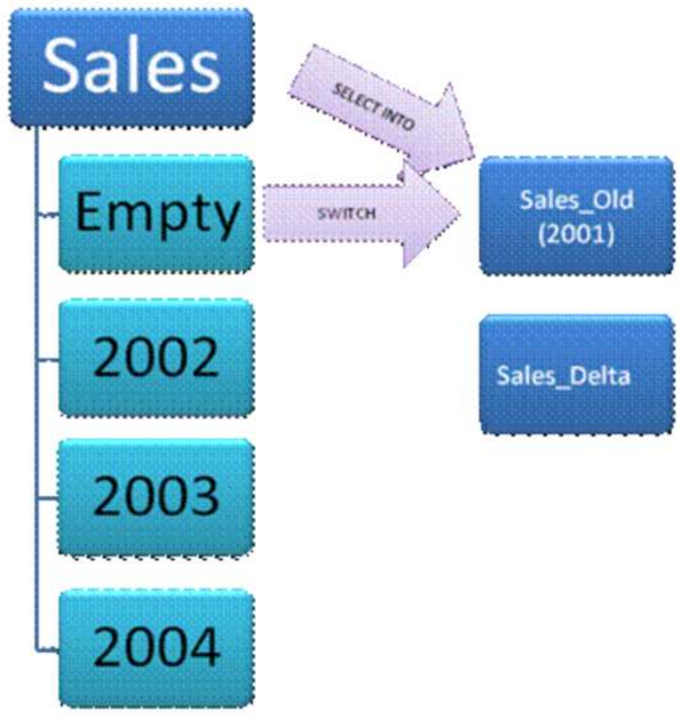
Since neither DELETE nor UPDATE can be minimally logged, it is not possible to change existing rows in a table or partition in minimally logging mode. However, it is sometimes possible to simulate both UPDATE and DELETE statements by using bulk loads.

Assume that you need to delete and/or update a large amount of rows in a table or partition. You estimate that the total number of rows changed by your operation is more than 10-20% of the total table size. In this case, the following technique may be useful to you.

1. **Write a change log to a temporary table.**

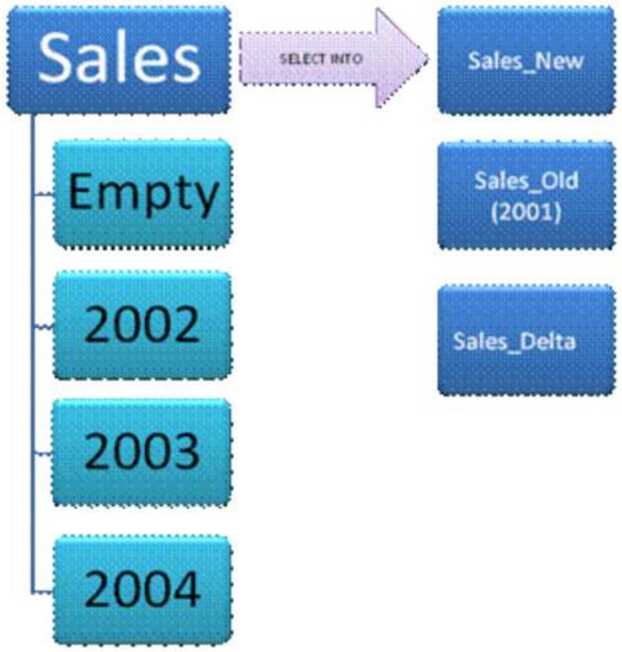
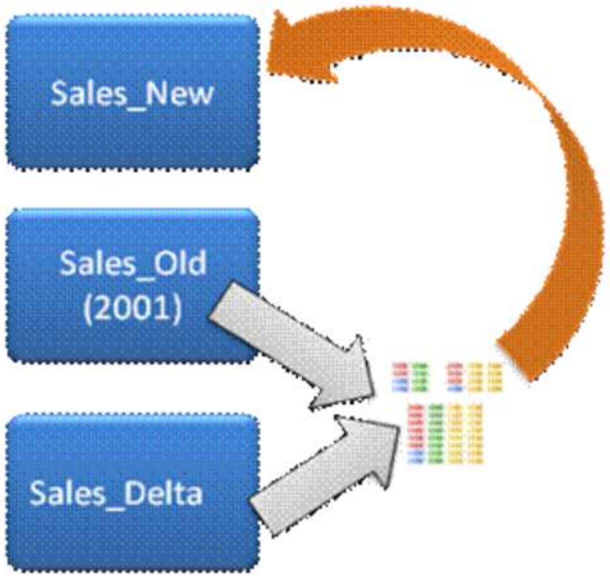
 Sales 2001 2002 2003 2004 SELECT INTO Sales_Delta BULK INSERT BULK INSERT Update Records	Create a temporary delta table as a switch target. The method to perform this is described in "Partition SWITCH". Bulk load the update records into **Sales_Delta.** Write the new value you that want to be there after the change.

1. **If the table is partitioned, switch out the partition you want to change.**

 Sales Empty 2002 2003 2004 SELECT INTO SWITCH Sales_Old (2001) Sales_Delta	Create a temporary switch target table to hold the old records. The method to perform this is described in "Partition SWITCH". Switch out the partition you want to change: ALTERTABLE Sales_P SWITCH PARTITION 1 TO Sales_Old You now have a **Sales_Delta** table and a **Sales_Old** table.

If the table is not partitioned, skip this step and use the original table instead of **Sales_New** as one of the source in the next step.

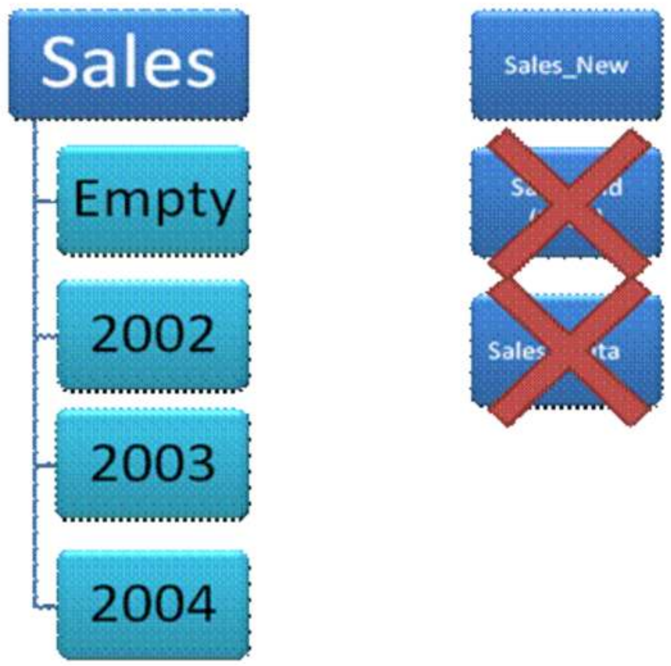
1. **Merge the change log with the original table and bulk load to temporary table.**

 Sales
SELECT INTO
Sales_New
Empty
Sales_Old (2001)
2002
Sales_Delta
2003
2004 Performing the merge: 
	Create a temporary table as switch target. The method to perform this is described in Partition SWITCH.
	
	Perform the merge between the old records and the delta. This is easily done with Integration Services merge- or the lookup component.
	
	Alternatively, you can perform the merge using Transact-SQL with a statement like:
	
	SELECT o.PrimaryKey
	  ,COALESCE(d.Col1, o.Col1)
	  , COALESCE(d.Col2, o.Col3)
	  ... etc...
	FROM Sales_Old o
	LEFTJOINSales_Delta d

When you are done with this step, the table **Sales_New** should be populated and contain the correct rows.

You can still drive high parallelism by starting several, concurrent merge operations all inserting to **Sales_New**. Each operation should work on its own, distinct subset of the **Sales_Delta** table.

1. **Delete the change log and original data.**



If you want the change to be a transaction, this is the point where you issue:

```
BEGINTRAN
```

Drop the delta table generated in step 1:

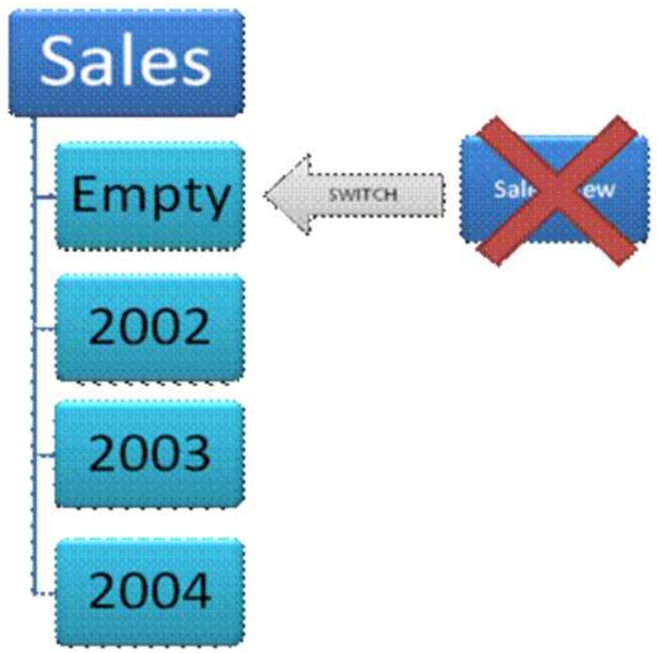
```
DROP TABLE Sales_Delta
```

Drop the original data switch-out table created in step 2:

```
DROP TABLE Sales_Old
```

If you are using a nonpartitioned table, truncate **Sales** instead of dropping **Sales_Old**.

1. **SWITCH in the result of the merge**



Use the SWITCH command to move the new version of the table into the old version:

```
ALTERTABLE Sales_New
SWITCHTO Sales_P PARTITION 1
```

Drop the **Sales_New** table to clean up:

```
DROPTABLE Sales_New
```

If you want the change to be a transaction, issue a commit:

```
COMMIT TRAN
```

That's it: You have now performed a large update in minimally logged mode.

Bulk Load, NOLOCK Queries, and Read Committed Snapshot Isolation

A popular way to read data from tables that are undergoing potentially blocking load operations is to issue queries with a NOLOCK hint. Also known as 'Dirty Reads', NOLOCK queries execute at the Read Uncommitted isolation level and run the risk of delivering incomplete or inconsistent results sets. However, because they don't take shared locks, they are not blocked by most writers.

Read committed snapshot isolation (RCSI) was introduced in SQL Server 2005 as a new mechanism to prevent queries reading data to block, or be blocked by, other queries modifying data in the same tables. It is a powerful alternative to NOLOCK because it guarantees a complete, transactionally consistent view of the data and does not require a special hint. While RCSI originally targeted scenarios common to OLTP workloads, the feature can be a powerful tool in data warehouse workloads or scenarios involving large-scale bulk insert operations.

RCSI is enabled as a database-wide setting. When enabled, reader queries do not acquire shared locks on rows, pages or tables, and as a result they are not blocked by X or BU-locks taken by others. Instead, new or modified rows in a table carry a 17-byte version identifier, and the before-images of any rows being changed by a transaction (updated or deleted) are copied to **tempdb** using the row versioning mechanisms within SQL Server. Reader queries consider only those rows that were committed as of the start of the query – by ignoring any later version numbers and referencing **tempdb** for appropriate earlier versions of rows.

Transaction semantics under RCSI are identical to that under ordinary read-committed isolation, with the difference being that reader queries always see data as of the start of the query, rather than being blocked until competing writers complete. As a result, applications require no changes to run under RCSI and will behave the same. Transactions requiring a REPEATABLE READ, or a SELECT with HOLDLOCK, will still request locks and be potentially blocked by concurrent writers. Also, RCSI should not be confused with the snapshot isolation level feature that was introduced in SQL Server 2005, which also makes use of row versioning.

For databases that might incur large amounts of UPDATE or DELETE activity, RCSI can create additional contention and bandwidth demand on **tempdb**. However, under RCSI, INSERT (including BULK INSERT), has no impact on **tempdb** and imposes no additional overhead aside from the 17 additional bytes added to each row. (Note that these 17 bytes are also noncompressible).

Although readers under RCSI are not affected by X locks, there are two bulk load situations that will block RCSI queries (as well as NOLOCK queries) in SQL Server 2008:

- When populating a heap with TABLOCK using BULK INSERT, INSERT-SELECT, or other bulk load operations. This is because the heap load acquires a BULK OPERATION intent exclusive (IX) lock and the NOLOCK or RCSI readers acquire a BULK OPERATION shared (S) lock. To bulk load a heap and permit concurrent readers even using RCSI or NOLOCK, you must eliminate the TABLOCK hint and thus give up minimal logging.
- When populating an clustered index with TABLOCK using BULK INSERT, INSERT…SELECT or other bulk load operations. This is because the load acquires a schema modification (Sch-M) lock and the readers acquire a shared schema (Sch-S) lock. The alternative is to use trace Flag 610 to achieve the minimal logging and avoid the TABLOCK hint.

Aside from the two exceptions above with RCSI enabled, BULK INSERT (including INSERT…SELECT) can be used with a TABLOCK hint, or it can be used with normal lock escalation rules, and in neither situation will readers be blocked. The performance will be similar to running queries with a NOLOCK hint, with the advantage that readers will see a transactionally consistent view of the data in the table, unlike under NOLOCK. And even with a number of long running reader queries concurrently active against a table, new INSERT operations will not be blocked by the readers.

To enable RCSI on a database, you must execute the following command while no other connections are active on the database being modified.

Transact-SQL Copy Code

```
      ALTER DATABASE <dbname> SET READ_COMMITTED_SNAPSHOT ON
```

To determine whether RCSI is active on an existing database, use the following code.

Transact-SQL Copy Code

```
SELECT name, is_read_committed_snapshot_on
FROM sys.databases
WHERE name = '<dbname>'
```

Before deciding to use RCSI in a workload, be sure to have a good understanding of the frequency of UPDATE and DELETE operations and ensure that **tempdb** is configured to handle that portion of the workload. Also recognize that the reduced blocking between readers and writers might create more I/O contention between newly-concurrent reader-and-writer queries, so ensure that your I/O bandwidth on data drives is also sufficient.

Optimizing Bulk Load

We will now dig deeper into the techniques used to optimize the bulk load process. SQL Server can support a very high throughput during bulk load. Under these conditions, all resources must be able to keep up.

There are several areas you can optimize the bulk load. First and foremost, parallelism is the key optimization for bulk loading that allows you to scale near linearly with the number of CPU cores.

In performance troubleshooting, wait stats and performance counters provide the data points needed for root cause analysis. Typical performance bottlenecks are treated in the sections “Relevant Wait Types” and “Performance Counters”.

Some areas of performance tuning and bottleneck elimination require a deeper treatment. The sections, “Performance Counters Optimizing Network”, “Scheduler Contention”, “PFS Contention”, describe these situations. Last but not least we end this section with some guidance on “Optimizing I/O and File Layout”.

Parallelism and Partitioning Input

To optimize throughput during bulk operations, parallelism is essential. This is best achieved by running several bulk load command at the same time. Because the each bulk operation is single threaded, multiple copies must be run to utilize all CPU cores. For each bulk command you run, one CPU core can be driven to 100% load. Hence, optimal performance, with all CPU cores at 100%, is achieved by running as many parallel bulk commands as your have CPU cores available for bulk operations.

When designing parallel operations, you must also consider the impact of one operation finishing before another. Ideally, the input data should be partitioned in equal-sized pieces. This ensures that all parallel jobs finish approximately at the same time.

The following diagram illustrates how a skew in the input distribution can make run times longer.

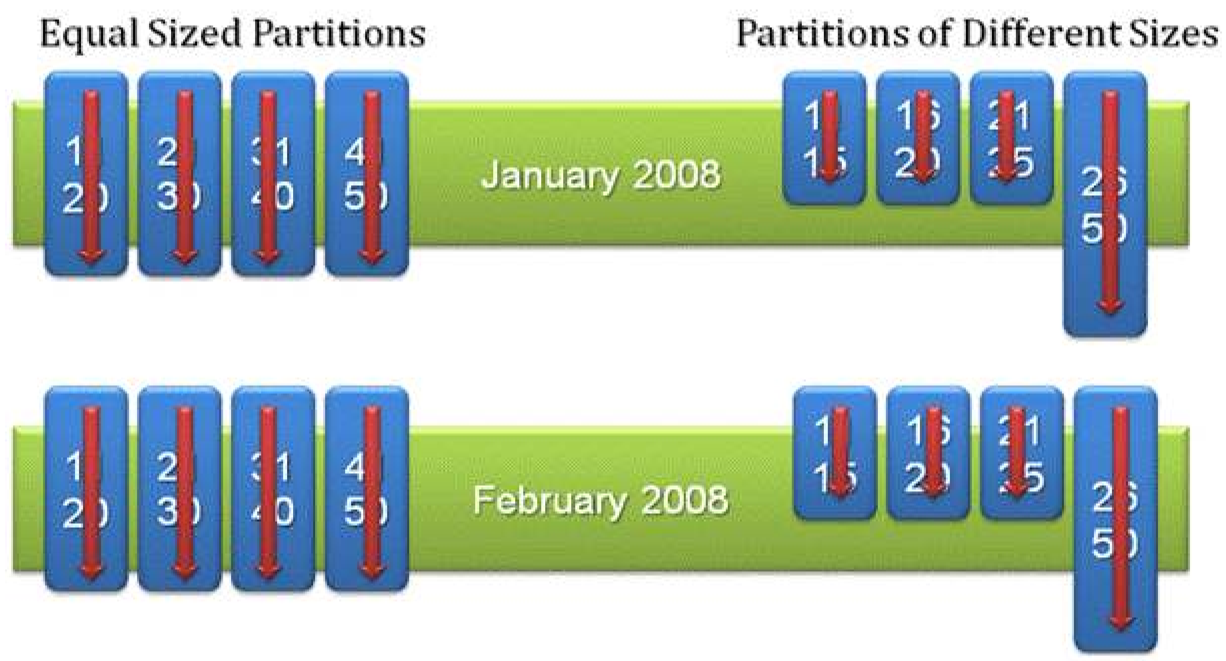


Figure 9: Balancing partition sizes

Relevant Wait Types

When you seek to drive bulk throughput higher, you should collect the data from **sys.dm_os_wait_stats** to determine whether any resources are blocking your CPUs. Focus on the longest wait times first and gradually optimize bottlenecks away. The following TechNet script is useful for this purpose: Retrieve Waitstat Snapshots [http://www.microsoft.com/technet/scriptcenter/scripts/sql/sql2005/waitstats/sql05vb048.mspx?mfr=true] (http://www.microsoft.com/technet/scriptcenter/scripts/sql/sql2005/waitstats/sql05vb048.mspx?mfr=true [http://www.microsoft.com/technet/scriptcenter/scripts/sql/sql2005/waitstats/sql05vb048.mspx?mfr=true]).

Also, collecting data from **sys.dm_os_latch_stats** is useful, especially if you plan to run many bulk threads into a single table.

The following table lists some typical waits during bulk load, their causes, and the action you can take to investigate and eliminate the bottlenecks.

Wait type	**Typical cause**	**Investigate / resolve**
LCK_<X>	One process blocking another	Are you using non overlapping input streams? Correct TABLOCK used? Find the top blocker.
PAGEIOLATCH_<X>	Disk system too slow	Add more disk or tune I/O. See "Optimizing I/O and File Layout".
IMPROV_IO	Text file data drive too slow	Optimize I/O on drive used for input files.
PAGELATCH_UP	Contention on PFS Pages	Make sure disk system is fast enough. See "PFS Contention" Run with –E flag.
ASYNC_NETWORK_IO	Network cannot keep up	See "Performance Counters Optimizing Network".
WRITELOG	Transaction log cannot keep up	Verify that you are using minimally logged operations. Make sure the transaction log is on fast disk.
OLEDB	Input data too slow	Optimize speed of input data source.
SOS_SCHEDULER_YIELD	Scheduler contention	See "Scheduler Contention".
ALLOC_FREESPACE_CACHE	Heap allocation contention (only found in **sys.dm_os_latch_stats**)	Too many threads are inserting into a heap at the same time. Consider partitioning table to get more heaps as insert targets.
PREEMPTY_COM_<X>	Nothing	These waits are normal and expected. Ignore them.

Table 4: Typical wait types for bulk operations

Performance Counters

The following performance counters are useful to track while bulk loading.

Performance Object	**Counter**	**Usage**
Logical Disk	Disk Write Bytes Bytes / sec	Measures the effective write speed to the drives.

Logical Disk	Disk Read Bytes / sec	Measure how fast data is being read from the source.
Logical Disk	Avg Disk Bytes / Write	Describes how large the disk block sizes are. See “Optimizing I/O and File Layout”.
Processor	% Processor time – Total	Expect one CPU at 100% for each bulk load task if not bound by bottlenecks.
MSSQL::Databases	Bulk copy rows / sec	Measure the number of rows coming into the database. Optimization yields a higher number.
Network Interface	Bytes Total / sec	Measures the bandwidth you are getting from the NICs in the server.
SQL:Databases	Log bytes Flushed / Sec	Measures throughput to transaction log.

Table 5: Performance Counters Optimizing Network

If you use Integration Services or BCP to bulk load data over the network, having the correct throughput configured is crucial.

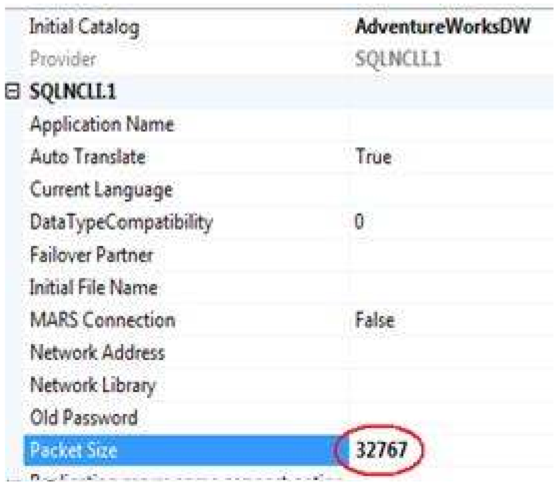
Work with your network department to understand the infrastructure and configuration of the network. Also, make sure you are configured with as little latency as possible. Here are some settings you can optimize:

- Use fast NIC and switches.
- Have the latest certified drivers for your NIC.
- Enable full duplex.
- Enable support for Jumbo frames.
- Use TCP Chimney Offload.
- Use Receive Side Scaling (RSS).

The following links provide an overview of key NIC abilities and how they interact with the Windows® operating system:

- High Performance Network Adapters and Drivers [http://www.microsoft.com/whdc/device/network/NetAdapters-Drvs.mspx]
- Scalable Networking Pack: Frequently Asked Questions [http://www.microsoft.com/technet/network/snp/faq.mspx]
- Introduction to Receive-Side Scaling [http://msdn.microsoft.com/en-us/library/ms795616.aspx]

From the bulk load client itself, you can optimize network throughput by changing the network packet size parameter of the SQL Server connection. In Integration Services, this is done in properties of the connection manager as illustrated here.



Initial Catalog	AdventureWorksDW
Provider	SQLNCLI.1
⊟ SQLNCLI.1	
Application Name	
Auto Translate	True
Current Language	
DataTypeCompatibility	0
Failover Partner	
Initial File Name	
MARS Connection	False
Network Address	
Network Library	
Old Password	
Packet Size	32767

Figure 10: Configuring Network Packet Size

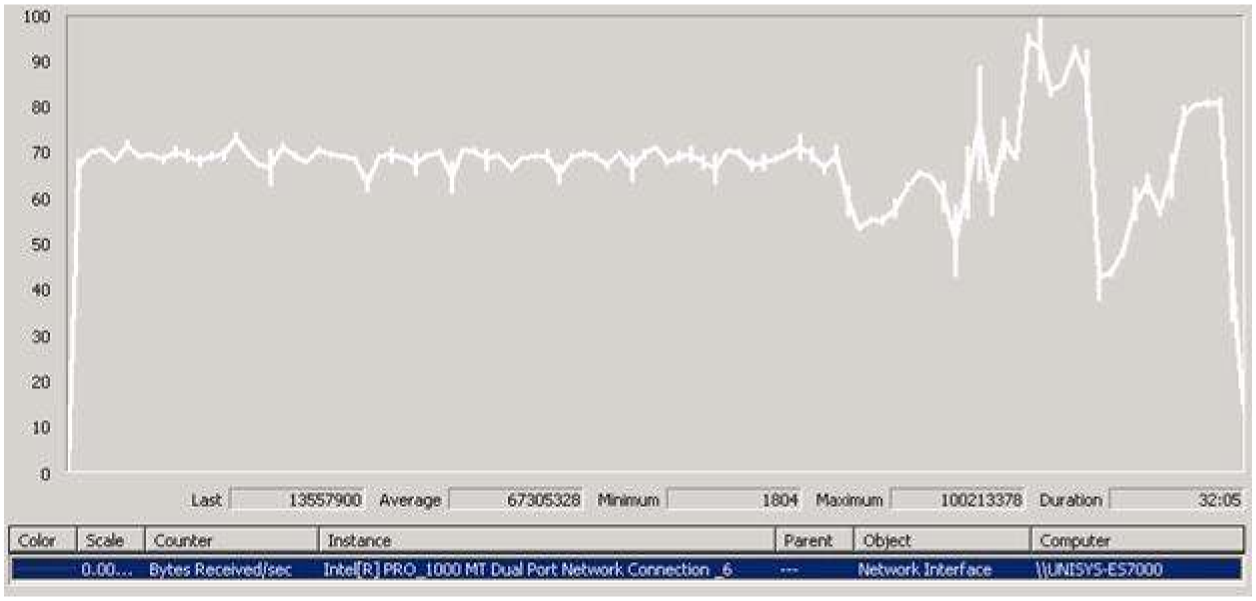
If you are using BCP, the **-a** switch allows you to specify the packet size. Tuning the network packet size brings down the number of read and write operations required by the SQL Server network stack. Because each operation has a CPU overhead required to set up, doing fewer I/O operations increase performance. The following table lists the optimization done with a single stream on the ETL World Record.

Network Packet Size	# Network Reads on SQL Server	# Disk Writes on SQL Server	# Disk reads from Integration Services	# Network Writes on Integration Services	Run Time
4096 (default)	465K	8K (256 KB)	14K (128 KB)	465K	2 min 56 sec
32K	58K	8K (256 KB)	14K (128 KB)	58K	2 min 36 sec

Table 6: Effect of network packet size

If you are using multiple network cards to achieve the proper bandwidth, you might also benefit from configuring Windows for interrupt affinity. For more information, see Scaling Heavy Network Traffic with Windows [http://blogs.msdn.com/sqlcat/archive/2008/09/18/scaling-heavy-network-traffic-with-windows.aspx] (http://blogs.msdn.com/sqlcat/archive/2008/09/18/scaling-heavy-network-traffic-with-windows.aspx).

With a 1 gigabit NIC, you can achieve around 70-100 megabytes (MB) per second depending on the data being bulk loaded. Measure the **Network Interface** performance counters to diagnose this.



```
Last    13557900   Average   67305328   Minimum   1804   Maximum   100213378   Duration   32:05
```

Color	Scale	Counter	Instance	Parent	Object	Computer
	0.00...	Bytes Received/sec	Intel[R] PRO_1000 MT Dual Port Network Connection _6	---	Network Interface	\\UNISYS-ES7000

Figure 11: Example of network throughput monitoring

Scheduler Contention

Scheduler contention happens when two, concurrent bulk load operations end up on the same SQLOS scheduler where they compete for the same CPU cycles. This typically occurs on computers that have many CPU cores. When it happens, you will see waiting for **SOS_SCHEDULER_YIELD** even if you have only as many load threads as CPUs.

One way to solve this is to configure your SQL Server for software nonuniform memory access (soft-NUMA), configuring one soft-NUMA node per CPU. After this, you assign individual TCP/IP ports for each soft NUMA node. This allows you to choose which CPU you run on, by specifying the port as part of your connection to SQL Server. Note that changing the soft NUMA settings requires a restart of SQL Server.

The following diagram illustrates this principle.

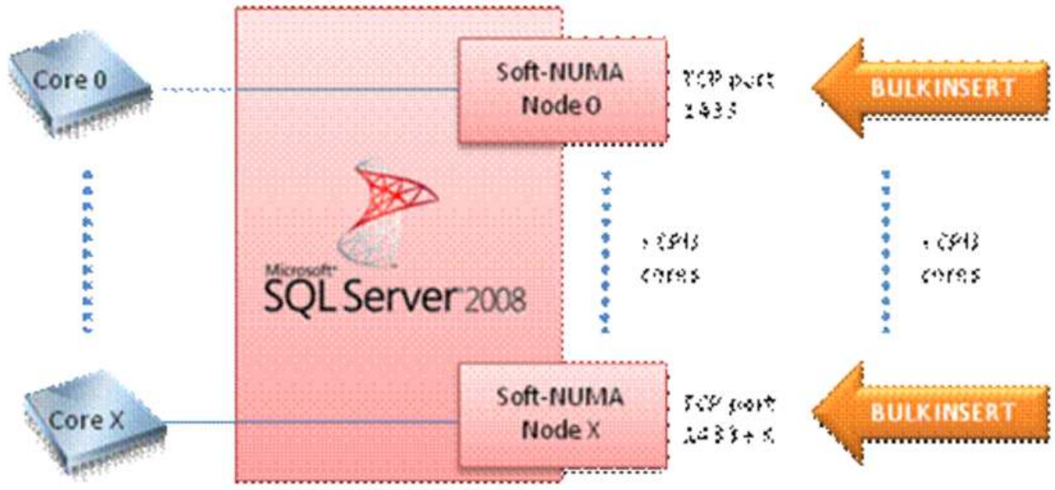
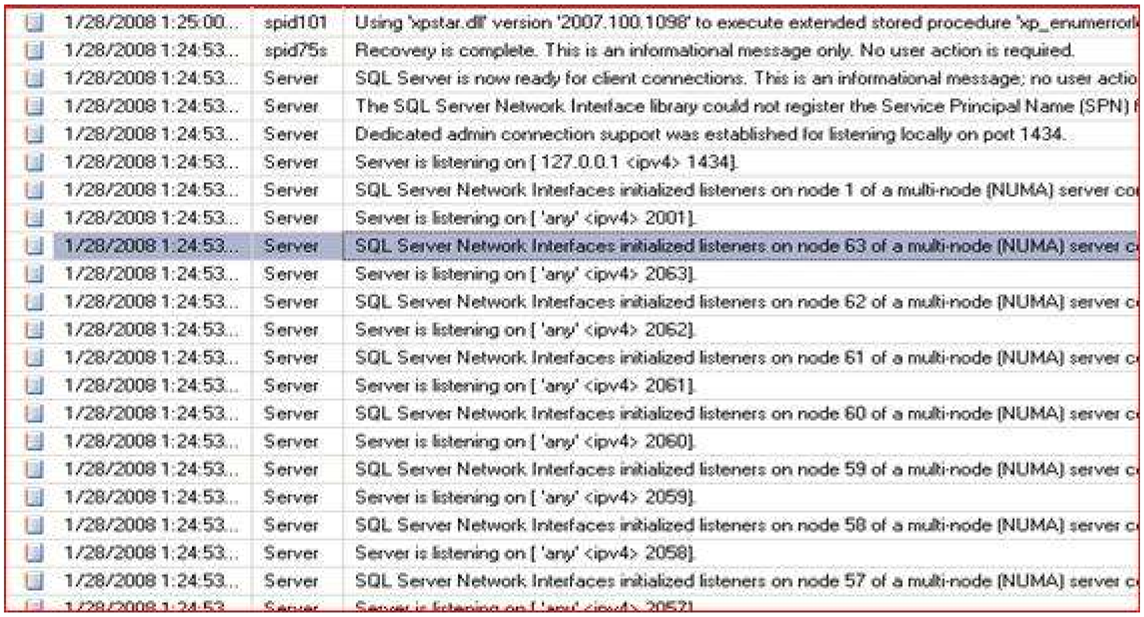


Figure 12: soft-NUMA and bulk connections

For information about configuring soft-NUMA for this type of configuration, see [How to: Configure SQL Server to Use Soft-NUMA](http://msdn.microsoft.com/en-us/library/ms345357.aspx) [http://msdn.microsoft.com/en-us/library/ms345357.aspx] (http://msdn.microsoft.com/en-us/library/ms345357.aspx) and [How to: Map TCP/IP Ports to NUMA Nodes](http://msdn.microsoft.com/en-us/library/ms345346.aspx) [http://msdn.microsoft.com/en-us/library/ms345346.aspx] (http://msdn.microsoft.com/en-us/library/ms345346.aspx).

When ports are configured for individual nodes, the SQL Server log file will show the TCP ports.



```
1/28/2008 1:25:00...  spid101  Using 'xpstar.dll' version '2007.100.1098' to execute extended stored procedure 'xp_enumerror
1/28/2008 1:24:53...  spid75s  Recovery is complete. This is an informational message only. No user action is required.
1/28/2008 1:24:53...  Server   SQL Server is now ready for client connections. This is an informational message; no user actio
1/28/2008 1:24:53...  Server   The SQL Server Network Interface library could not register the Service Principal Name (SPN) f
1/28/2008 1:24:53...  Server   Dedicated admin connection support was established for listening locally on port 1434.
1/28/2008 1:24:53...  Server   Server is listening on [ 127.0.0.1 <ipv4> 1434].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 1 of a multi-node (NUMA) server co
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2001].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 63 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2063].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 62 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2062].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 61 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2061].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 60 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2060].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 59 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2059].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 58 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2058].
1/28/2008 1:24:53...  Server   SQL Server Network Interfaces initialized listeners on node 57 of a multi-node (NUMA) server c
1/28/2008 1:24:53...  Server   Server is listening on [ 'any' <ipv4> 2057].
```

Figure 13: Inspecting soft-NUMA configuration in the SQL Server log

You can also simultaneously maintain another port that is mapped to *all* CPUs for regular connections that are not participating in the synchronized Bulk Insert workload.

It is possible to avoid scheduler contention using a different technique – called terminate and reconnect. This technique is documented in this technical note: Resolving scheduler contention for concurrent BULK INSERT [http://sqlcat.com/technicalnotes/archive/2008/04/09/resolving-scheduler-contention-for-concurrent-bulk-insert.aspx] (http://sqlcat.com/technicalnotes/archive/2008/04/09/resolving-scheduler-contention-for-concurrent-bulk-insert.aspx).

PFS Contention

Even though very little information must be logged under bulk load, some allocation bottlenecks can still occur. SQL Server uses a special type of page, the Page Free Space (PFS), to keep track of the used pages inside a data file.

Because bulk operations need to allocate new pages very fast, accessing the PFS pages to allocate new space, may become a bottleneck. If this bottleneck occurs, you will see waits for PAGELATCH_UP. You should query **sys.dm_os_buffer_descriptors** to make sure that the page you are waiting for is a PFS page. The PAGELATCH_UP bottleneck can be removed by adding more data files to the affected filegroup. Note that having many data files in a filegroup can create a less efficient I/O pattern, as described in "The SQL Server Bulk Target" below.

For more information about PFS pages and how they may form a bottleneck, see the technical note How many files should a database have? -Part 1: OLAP workloads [http://sqlcat.com/technicalnotes/archive/2008/03/07/How-many-files-should-a-database-have-part-1-olap-workloads.aspx] (http://sqlcat.com/technicalnotes/archive/2008/03/07/How-many-files-should-a-database-have-part-1-olap-workloads.aspx) and Managing Extent Allocations and Free Space [http://msdn.microsoft.com/en-us/library/ms175195.aspx] (http://msdn.microsoft.com/en-us/library/ms175195.aspx) in SQL Server Books Online.

Optimizing I/O and File Layout

When bulk loading many streams in parallel, you should carefully consider the layout of the I/O subsystem. As you increase read and write speeds, disk latency becomes a focal point of optimization.

The bulk load process must both concurrently read the data from the source and bulk load them into the Database Engine. This means that the I/O pattern issued is highly sequential in nature. Significant performance gains can be had by taking advantage of this. As an example, the ETL World Records test saw a factor-of-two improvement in I/O by changing the disk layout to support the bulk load.

Often, you can benefit from placing the source data on a different set of disks than the destination. By isolating the read activity from the writes, you are splitting the two different sequential streams to their own dedicated spindles.

From a Windows perspective, you should make sure that the partitions you use are formatted at 64 KB NTFS cluster sizes and properly sector aligned.

The source and destination can themselves be optimized, which will be discussed in the following sections.

Data Source

For the highest bulk load performance, you should be issuing multiple, concurrent reads from the data source, one or more for each bulk load operation.

If the source is partitioned, and you are performing sequential read, you can optimize the source by placing each source data partition on its own dedicated set of spindles. However, if you have many source partitions, this can quickly become unmanageable.

As a useful example, let us assume that you use text files as the source data and that you are executing 64 concurrent copies of the BULK INSERT command on these text files.

Let us look at the two extremes for disk configuration:

- Just a Bunch Of Disks (JBOD) – in this configuration, a single disk (or perhaps a RAID 1 or RAID 10 LUN) allocated per sequential stream
- Stripe And Mirror Everything (SAME) – in this configuration, a single large LUN is allocated

Just a Bunch Of Disks (JBOD)

From a performance perspective, the JBOD I/O layout is typically the fastest. In this case, you have one disk allocated per text file. Such a configuration is illustrated below.

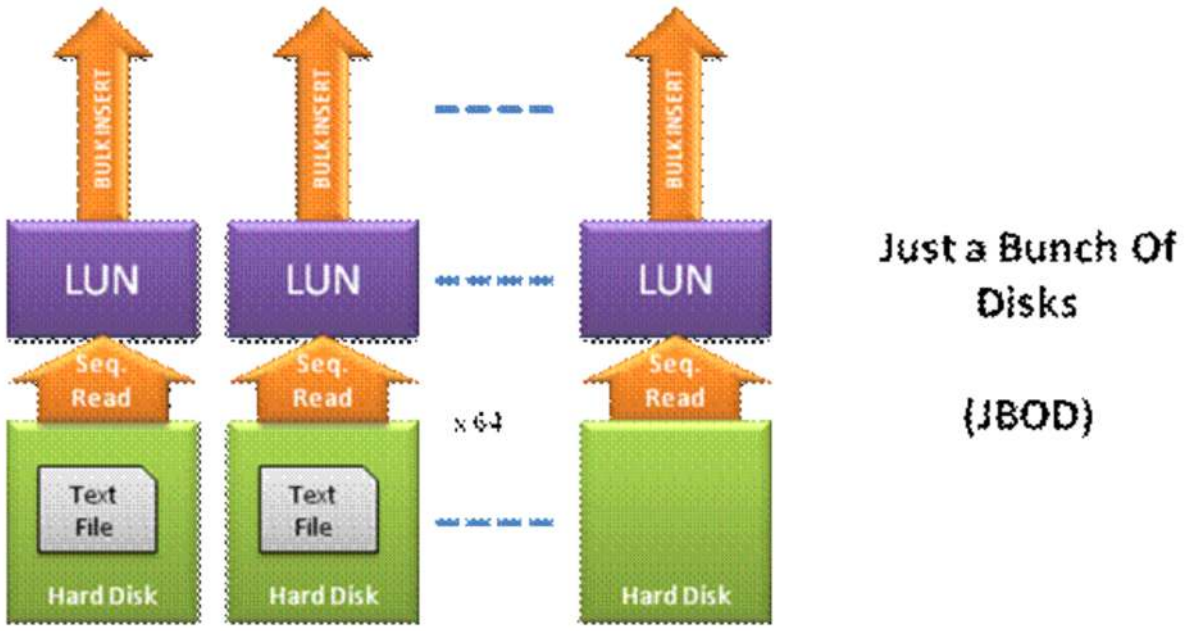


Figure 14: JBOD disk layout

Because BULK INSERT will sequentially read the file, this disk will see a purely sequential I/O pattern with very little latency on disk seek operations. A modern hard drive can easily deliver over 100 MB/second at sequential read, more than enough to feed the BULK INSERT. However, consider the impact of this layout on manageability. You would have to expose 32 LUNs to the operating system, just to hold input files. Each of these LUNs will need to be configured at the SAN configuration/SCSI controller level. Add the partitioning, file system management, and backup procedures to this burden and you are suddenly talking a lot of work. Unless this setup is automated and you have worked out an advanced file placement and retrieval mechanism, the room for human error is large.

Stripe And Mirror Everything (SAME)

At the other end of the spectrum, you could go with a highly manageable configuration. In such a configuration, you allocate a single, large LUN, with many disks in a RAID configuration for all data files.

This scenario is commonly used because it is easy to manage and provides a fast way to expose large amounts of storage to a server.

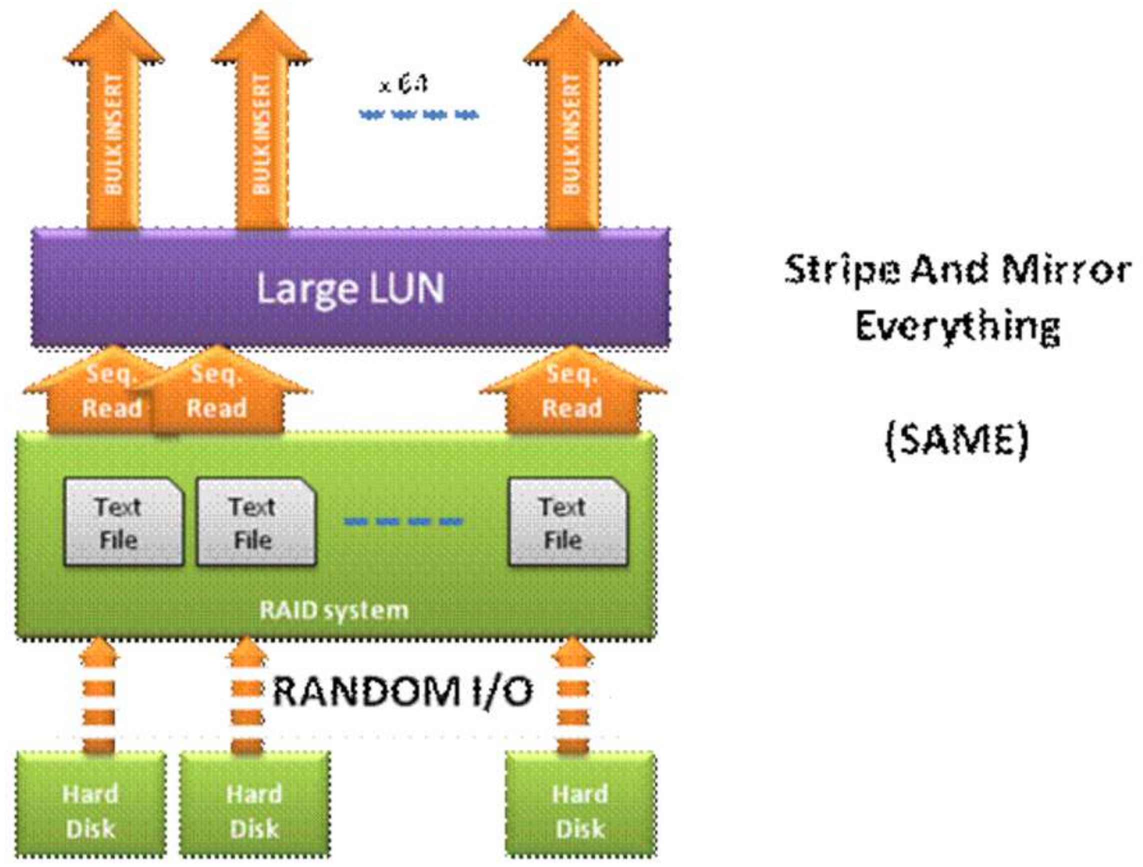


Figure 15: SAME disk layout

However, multiple, concurrent sequential reads on the same spindles create what is known as I/O weaving. Even though the workload is sequential in nature (from the perspective of the SQL Server), it is seen as random from the perspective of the disk. Because you are now doing random I/O and incurring the overhead of disk seeking, you need an order of magnitude more spindles to get the throughput required.

Some SAN systems and SCSI controllers are able to use their cache mechanism to transform the random I/O pattern generated by the multiple streams back into a sequential pattern. You should work with your storage vendor to test the limits of the storage system with this I/O pattern.

The best configuration for your environment may lie somewhere between the JBOD and SAME strategies. You should carefully balance the performance of JBOD with manageability of SAME. Also, note that the JBOD strategy utilizes the fact that bulk load is highly sequential in nature. If you are tuning for a mixed workload, the SAME strategy may provide better overall throughput.

The SQL Server Bulk Target

As was the case with the source data, the bulk load target table can also benefit from I/O optimization. When you are bulk loading into a table, the I/O write pattern is highly sequential. This can be used to your advantage.

First, understand how SQL Server writes to files. Consider a filegroup with 2 files.

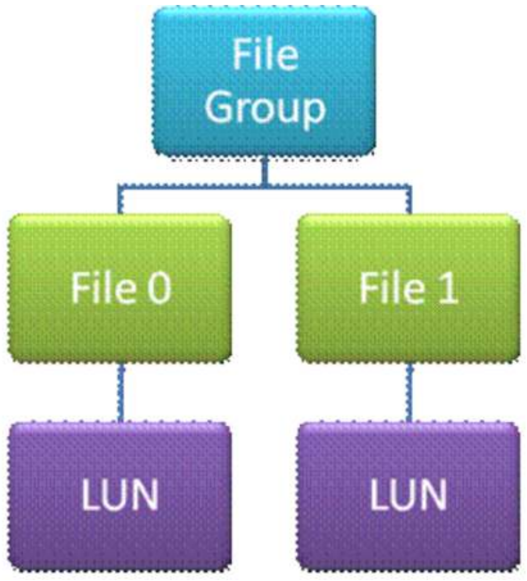


Figure 16: Filegroups and files

When SQL Server bulks load into a table allocated to this filegroup, extents are allocated in a round robin fashion. In the default configuration, SQL Server allocates one extent to each file before moving to the next in the chain. It is possible to increase this to 64 extents using the –E startup flag.

File	**Default extend allocation**	**With –E startup flag**
File 0	Extent 1	Extent 1-64
File 1	Extent 2	Extent 65-128
File 0	Extent 3	Extent 129-192
File 1	Extent 4	Extent 192-256
…etc..	… etc…	… etc…

Table 7: The effect of the -E flag

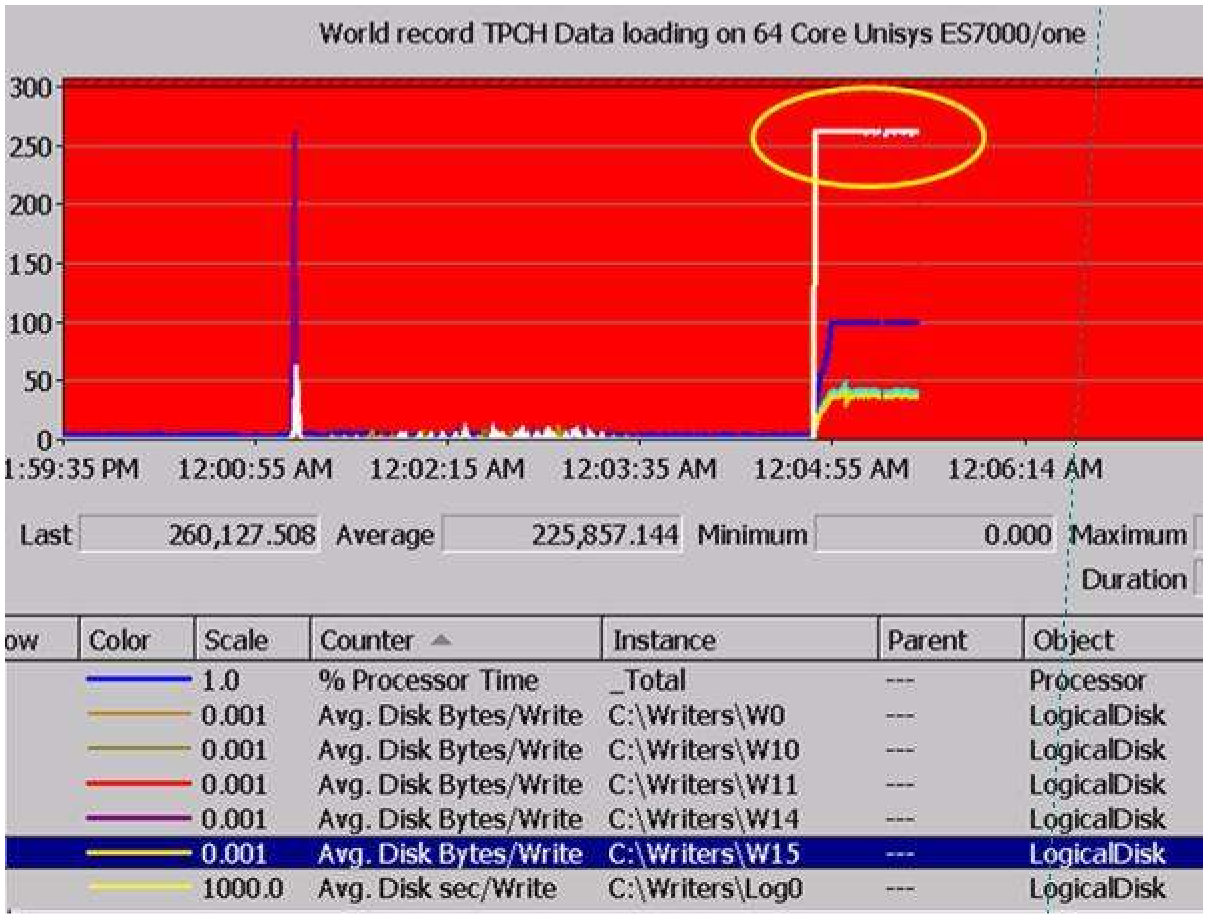
Because each extent is 64 KB, this is the minimum write size into a file when bulk loading. At first glance, we would therefore expect to see a sequential stream of 64 KB blocks written to each LUN. However, the picture is actually a bit better than this.

SQL Server uses the Windows scatter/gather I/O optimization. With this optimization, several I/O requests can be grouped together to form a single, larger block request. For example, four 64 KB I/O requests may be gathered together into a single 256 KB request. Because issuing a single, large I/O request is faster than more, smaller requests, this scatter/gather improves the speed of write operations. At a high bulk throughput, this works to your advantage, and you will typically see larger block writes – up to the maximum of 256 KB per I/O request.

The more files you have in a filegroup, the longer it takes for SQL Server to "return" to the first file for writes. Because of this, the scatter/gather optimization may choose to flush the I/O buffer before SQL Server is able to fill the I/O buffer to its 256 KB limit. By measuring **the Logical Disk / Avg. Disk Bytes / Write** performance counter, you can determine how well the scatter/gather is filling up the write buffer.

The fewer files you have in a filegroup, the more likely it is that the scatter/gather will create large I/O requests. Running SQL Server with the –E flag will further contribute to grouping write block sizes.

For the ETL World Record, the number of files per filegroup was kept low, which allowed us to get a pattern of pure, 256 KB block sizes.



Last 260,127.508 Average 225,857.144 Minimum 0.000 Maximum
Duration

ow	Color	Scale	Counter	Instance	Parent	Object
	🟦	1.0	% Processor Time	_Total	---	Processor
	🟧	0.001	Avg. Disk Bytes/Write	C:\Writers\W0	---	LogicalDisk
	🟫	0.001	Avg. Disk Bytes/Write	C:\Writers\W10	---	LogicalDisk
	🟥	0.001	Avg. Disk Bytes/Write	C:\Writers\W11	---	LogicalDisk
	🟪	0.001	Avg. Disk Bytes/Write	C:\Writers\W14	---	LogicalDisk
	🟨	0.001	Avg. Disk Bytes/Write	C:\Writers\W15	---	LogicalDisk
	🟨	1000.0	Avg. Disk sec/Write	C:\Writers\Log0	---	LogicalDisk

Figure 17: Measuring block sizes

However, be aware that too few files may cause PFS contention, as described earlier.

Optimizing the Relationship Between Files and LUNs

During bulk load, the individual database files will receive a sequential stream of write I/O, with block sizes anywhere between 64 KB and 256 KB.

As we discussed in the "Data Source" section, optimizing the I/O system for a highly sequential workload pattern can yield significant benefits. Again, you should consider the two extremes: JBOD vs. SAME – should you have one big LUN or several, smaller LUNs?

As a rule of thumb, allocating one database file per LUN create the highest performing configuration. Each LUN can then be built out of several disks in a RAID configuration. Unlike the read optimization, where each bulk input file needed its own LUN, you rely on SQL Server to stream multiple bulk destination tables sequentially into the same file. Because SQL Server, during bulk load, writes sequentially to the files, you do *not* have to allocate one LUN per bulk stream for optimal performance. This means that you can create fewer, large LUN on nonshared spindles. Each LUN should service one database file – that is, you should keep the ratio between LUN and database close to 1:1. Of course, if the SAN or controller cache can handle the I/O weaving, you can get away with more than one file per LUN. You should test before you deploy to understand how multiple, sequential write streams to the same LUN affect your I/O system.

The above considerations work in favor of the SAME configuration – one big LUN for each filegroup in the database. However, as was mentioned earlier in "PFS Contention", too few files (and therefore LUNs, because you want to keep the ratio as 1:1) may cause bottlenecks of PFS pages. Therefore, you may want to allocate more than one database file/LUN per filegroup. For example, in the case of the ETL World Record, we used one database file/LUN for every 4 cores in the destination computer.

Case Study: ETL World Record

During the ETL World Record, 56 bulk insert tasks were run in parallel with an average speed of 600 MB per second and peaks up to 850 MB per second.

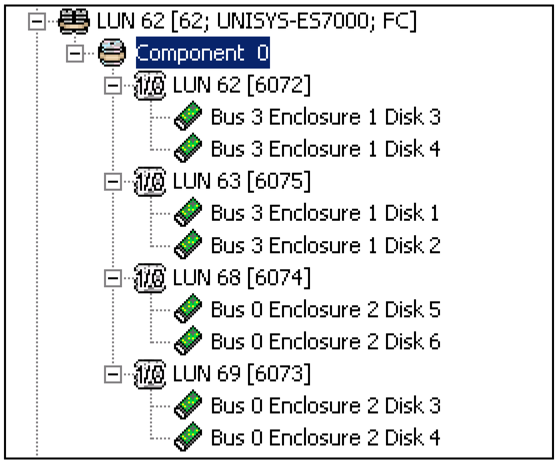
The SQL Server 2008 database server:

- Unisys ES7000/one Enterprise Server with 32 Dual core 3Ghz Intel 7140M CPUs
    - Total of 64 cores

- 8 x 4Gbit Emulex HBAs
- 8 x 1Gbit Intel Pro /1000MT Network cards
- 256 GB RAM (but only 30GB used for bulk insert operations)

The optimal SQL Server file layout under these conditions was:

- SQL Server Data files located on EMC Clariion CX3-80 SAN with a total of 165 spindles.
    - 16 Meta-LUNs
        - Each Meta-LUN is made built out a total of 8 spindles as a



LUN 62 [62; UNISYS-ES7000; FC]
Component 0
LUN 62 [6072]
Bus 3 Enclosure 1 Disk 3
Bus 3 Enclosure 1 Disk 4
LUN 63 [6075]
Bus 3 Enclosure 1 Disk 1
Bus 3 Enclosure 1 Disk 2
LUN 68 [6074]
Bus 0 Enclosure 2 Disk 5
Bus 0 Enclosure 2 Disk 6
LUN 69 [6073]
Bus 0 Enclosure 2 Disk 3
Bus 0 Enclosure 2 Disk 4

- Figure 18: a Meta-LUN
    - 133GB /4Gbit , 15K rpm spindles
    - 56 data files, with up to 4 data files on each LUN
        - SQL Server Log file

    - A single4x(1+1R1) Meta-LUN
    -

The file layout for reading:

- 2 x EMC Clariion CX600 SAN
    - 10 LUNs in RAID5, 5 on each CX600
    - 7 x 36GB , 15K rpm spindles per LUN
    - Each LUN holds the text files for 5 or 6 of the input files

For more information, see the following:

- ETL World Records Announcement [http://blogs.msdn.com/sqlperf/archive/2008/02/27/etl-world-record.aspx]
- Unisys ES7000 Enterprise servers and BI solutions [http://www.unisys.com/BI.aspx]

Additional References on I/O

For more information about I/O, refer to these articles:

- Storage Top 10 Best Practices [http://www.microsoft.com/technet/prodtechnol/sql/bestpractice /storage-top-10.mspx]
- SQL IO download page [http://www.microsoft.com/downloads/details.aspx?familyid=9a8b005b-84e4-4f24-8d65-cb53442d9e19&displaylang=en]
- SQLIOSim tool KB article [http://support.microsoft.com/kb/231619.aspx]
- SQL Server Storage Engine Team Blog [http://blogs.msdn.com/sqlserverstorageengine/]
- SQL Server I/O Basics [http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain /sqlIObasics.mspx] (http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain

/sqlIObasics.mspx)

- Predeployment I/O Best Practices [http://www.microsoft.com/technet/prodtechnol/sql/bestpractice /pdpliobp.mspx]
- Disk Subsystem Performance Analysis for Windows [http://www.microsoft.com/whdc/device/storage /subsys_perf.mspx]

Conclusion

If you are running out of your processing batch window to load all your data, or if you simply want to achieve maximum throughput when loading a large amount of data, this paper provides you with the guidance to investigate and optimize performance.

We have provided you with information how to choose the bulk load method that best fits your scenario. Some typical scenarios occur when bulk loading. Solutions are described and sample scripts to help you get started are provided.

As with all workloads that drive heavy throughput, removing bottlenecks are key to providing the best possible performance. In the "Optimizing Bulk Load" section, we have provided guidelines on how to optimize the configuration of SQL Server for fast bulk load. Where appropriate, links to additional reading are provided.

The combination of the SQL Server and Integration Services allow you to load your data very fast. During the ETL World Record we used the guidance described in this paper. By implementing the optimization described here, we tripled the throughput compared with an "out of the box" installation. Furthermore, we achieved close to linear scalability all the way from 1 to 64 concurrent bulk load operation on a 64 core Unisys ES7000/one system running Windows Server® 2008 Datacenter Edition.

For more information:

http://www.microsoft.com/sqlserver/ [http://www.microsoft.com/sqlserver/default.aspx] : SQL Server Web site

http://sqlcat.com [http://sqlcat.com/default.aspx] : SQL Customer Advisory Team Web site

http://technet.microsoft.com/en-us/sqlserver/ [http://technet.microsoft.com/en-us/sqlserver/default.aspx] : SQL Server TechCenter

http://msdn.microsoft.com/en-us/sqlserver/ [http://msdn.microsoft.com/en-us/sqlserver/default.aspx] : SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?

Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback [mailto://microsoft.com:25 /default.aspx?subject=White%20Paper%20Feedback:%20The%20Data%20Loading%20Performance%20Guide] .