msdn

Data Compression: Strategy, Capacity Planning and Best Practices

Microsoft® SQL Server 2008

SQL Server Technical Article

Writer: Sanjay Mishra

Contributors: Marcel van der Holst, Peter Carlin, Sunil Agarwal

Technical Reviewer: Stuart Ozer, Lindsey Allen, Juergen Thomas, Thomas Kejser, Burzin Patel, Prem Mehra, Joseph Sack, Jimmy May, Cameron Gardiner, Mike Ruthruff, Glenn Berry (SQL Server MVP), Paul S Randal (SQLskills.com), David P Smith (ServiceU Corporation)

Published: May 2009

Applies to: SQL Server 2008

Summary: The data compression feature in SQL Server 2008 helps compress the data inside a database, and it can help reduce the size of the database. Apart from the space savings, data compression provides another benefit: Because compressed data is stored in fewer pages, queries need to read fewer pages from the disk, thereby improving the performance of I/O intensive workloads. However, extra CPU resources are required on the database server to compress and decompress the data, while data is exchanged with the application. Therefore, it is important to understand the workload characteristics when deciding which tables to compress.

Introduction

The data compression [http://technet.microsoft.com/en-us/library/cc280449.aspx] feature in the Microsoft® SQL Server® 2008 database software can help reduce the size of the database as well as improve the performance of I/O intensive workloads. However, extra CPU resources are required on the database server to compress and decompress the data, while data is exchanged with the application. Therefore, it is important to understand the workload characteristics when deciding which tables to compress. This white paper provides guidance on the following:

- How to decide which tables and indexes to compress
- How to estimate the resources required to compress a table
- How to reclaim space released by data compression
- The performance impacts of data compression on typical workloads

About Data Compression

SQL Server 2008 provides two levels of data compression – row compression and page compression. Row compression [http://technet.microsoft.com/en-us/library/cc280576.aspx] helps store data more efficiently in a row by storing fixed-length data types in variable-length storage format. A compressed row uses 4 bits per compressed column to store the length of the data in the column. NULL and 0 values across all data types take no additional space other than these 4 bits.

Page compression [http://technet.microsoft.com/en-us/library/cc280464.aspx] is a superset of row compression. In addition to storing data efficiently inside a row, page compression optimizes storage of multiple rows in a page, by minimizing the data redundancy. Page compression uses prefix compression and dictionary compression. Prefix compression [http://blogs.msdn.com/sqlserverstorageengine/archive/2008/01/18/what-is-page-compression.aspx] looks for common patterns in the beginning of the column values on a given column across all rows on each page. Dictionary compression [http://blogs.msdn.com/sqlserverstorageengine/archive/2008/01/18/details-on-page-compression-page-dictionary.aspx] looks for exact value matches across all columns and rows on each page. Both dictionary and prefix are type-agnostic and see every column value as a bag of bytes.

For more information about the data compression feature, see SQL Server Books Online [http://technet.microsoft.com/en-us/library/cc280449.aspx] . The SQL Server Storage Engine blog [http://blogs.msdn.com/sqlserverstorageengine/archive/tags/Data%20Compression/default.aspx] is also a great resource for the internals of data compression.

The data compression feature is available in the Enterprise and Developer editions of SQL Server 2008. Databases with

compressed tables or indexes cannot be restored, attached, or in any way used on other editions. To determine whether a database is using compression, query the dynamic management view (DMV)sys.dm_db_persisted_sku_features. To determine what is compressed, and how (row or page), query the data_compression_desc column in the catalog view sys.partitions.

Deciding What to Compress

SQL Server 2008 provides great flexibility in how data compression is used. Row and page compression can be configured at the table, index, indexed view, or partition level. Some examples of the flexibility in applying data compression are to:

- Row-compress some tables, page-compress some others, and don't compress the rest.
- Page-compress a heap or clustered index, but have no compression on its nonclustered indexes.
- Row-compress one index, and have no compression on another index.
- Row-compress some partitions of a table, page-compress some others, and don't compress the rest.

With this flexibility comes the challenge in deciding what to compress. This section provides some guidelines to assist in deciding what to compress. Some of the factors that influence this decision are:

- Estimated space savings
- Application workload

Estimated Space Savings

The stored procedure sp_estimate_data_compression_savings [http://msdn.microsoft.com/en-us/library /cc280574.aspx] estimates the amount of space saved by compressing a table and its indexes. It functions by taking a sample of the data and then compressing it in **tempdb**. Estimate the space savings for the largest tables and indexes in a database, and consider compressing only those tables and indexes that yield significant space savings.

The stored procedure sp_estimate_data_compression_savings [http://msdn.microsoft.com/en-us/library /cc280574.aspx] estimates the space savings one table at a time. This can be wrapped in a script to estimate the space savings for all the tables and indexes in a database – as shown in these two blogs: Whole Database - Data Compression Procs [http://sqlblog.com/blogs/paul_nielsen/archive/2008/03/13/whole-database-data-compression-procs.aspx] and Procedure used for applying Database Compression to Microsoft SAP ERP system [http://blogs.msdn.com/saponsqlserver/archive/2008/11/24/procedure-used-for-applying-database-compression-to-microsoft-sap-erp-system.aspx] . Be aware that estimating data compression savings on an entire database may take a long time in a database with several thousand tables and indexes, such as an SAP ERP database.

Data and Data Types

The amount of space saved by compressing a table depends on the "data" the table contains (after all, it is called "data" compression!). Some data compresses significantly, while some other doesn't. Tables that contain the following patterns of data compress very well:

- Columns with numeric or fixed-length character data types where most values don't require all the allocated bytes: For example, integers where most values are less than 1000
- Nullable columns where a significant number of the rows have a NULL value for the column
- Significant amounts of repeating data values or repeating prefix values in the data

Some patterns of data that do not benefit much from compression are:

- Columns with numeric or fixed-length character data types where most values require all the bytes allocated for the specific data type
- Not much repeating data
- Repeating data with non-repeating prefixes
- Data stored out of the row
- FILESTREAM data

A table or a partition can have three allocation units - IN_ROW_DATA, LOB_DATA, and ROW_OVERFLOW_DATA [http://msdn.microsoft.com/en-us/library/ms189051.aspx] . The data stored in LOB_DATA and ROW_OVERFLOW_DATA allocation units is not compressed. Only the data that is stored in the IN_ROW_DATA allocation unit is compressed. Use Appendix B to understand how much data is stored in each of these three allocation units.

FILESTREAM [http://msdn.microsoft.com/en-us/library/cc949109.aspx] data is stored outside the database in a FILESTREAM data container on an NTFS volume. This data is not compressed.

Application Workload

Compressed pages are persisted as compressed on disk and stay compressed when read into memory. Data is decompressed (not the entire page, but only the data values of interest) when it meets one of the following conditions:

- It is read for filtering, sorting, joining, as part of a query response.
- It is updated by an application.

There is no in-memory, decompressed copy of the compressed page. Decompressing data consumes CPU. However, because compressed data uses fewer data pages, it also saves:

- Physical I/O: Because physical I/O is expensive from a workload perspective, reduced physical I/O often results in a bigger saving than the additional CPU cost to compress and decompress the data. Note that physical I/O is saved both because a smaller volume of data is read from or written to disk, *and* because more data can remain cached in buffer pool memory.
- Logical I/O (if data is in memory): Because logical I/O consumes CPU, reduced logical I/O can sometimes compensate for the CPU cost to compress and decompress the data.

The savings in logical and physical I/O is largest when tables or indexes are scanned. When singleton lookups (for read or write) are performed, I/O savings from compression are smaller - they only occur if compression causes more requests to target the same page, and this leads to reduced physical I/O.

The CPU overhead of row compression is usually minimal (generally less than or equal to 10 percent in our experience). If row compression results in space savings and the system can accommodate a 10 percent increase in CPU usage, all data should be row-compressed. For example, SAP ERP NetWeaver 7.00 Business Suite 7 [https://www.sdn.sap.com /irj/sdn/go/portal/prtroot/docs/library/uuid/60a236a2-8104-2b10-5ebe-8fef61cc82fd] and above use row compression on all tables.

The CPU overhead of page compression can be higher than row compression, and therefore deciding what to page-compress is more difficult. Some general guidelines for page compression are:

- Start with less frequently used tables and indexes to ensure the understanding of the system behavior is accurate.
- If CPU headroom is not available, do not use page compression without thorough testing.
- Query operations such as filtering, joins, aggregates, and sorting see decompressed data, and hence the cost of these operations are not affected by data compression. A query, whose cost is dominated by complex processing operations (for example, a query involving multiple table joins or complex aggregate operations) is much less likely to see any significant change in performance or CPU utilization, due to data compression. These complex queries usually occur in data warehousing applications, but they can occur in other applications as well. If CPU time for an application consists primarily of complex queries, page compression may not impact CPU measurably. In such scenarios, space savings may be the prime driver for data compression.
- Most large-scale data warehousing workloads are scan-intensive, and storage is typically a premium as well. In a data warehouse or large-scale data mart, if there is CPU headroom available, we recommend page-compressing all objects in the database, rather than evaluating object-by-object as outlined below.

A more detailed approach to deciding what to compress involves analyzing the workload characteristics for each table and index. It is based on the following two metrics:

- U: The percentage of update operations on a specific table, index, or partition, relative to total operations on that object. The lower the value of U (that is, the table, index, or partition is infrequently updated), the better candidate it is for page compression.
- S: The percentage of scan operations on a table, index, or partition, relative to total operations on that object. The higher the value of S (that is, the table, index, or partition is mostly scanned), the better candidate it is for page compression.

U: Percent of Update Operations on the Object

To compute U, use the statistics in the DMV sys.dm_db_index_operational_stats [http://technet.microsoft.com/en-us /library/ms174281.aspx] . U is the ratio (expressed in percent) of updates performed on a table or index to the sum of all operations (scans + DMLs + lookups) on that table or index. The following query reports U for each table and index in the database.

Transact-SQL   Copy Code

```
[illegible code block]
```

```
▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊
▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊
▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊
▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊ ▊▊▊▊▊▊ ▊▊▊▊▊▊
▊ ▊ ▊ ▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊
▊ ▊ ▊ ▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊
▊ ▊ ▊ ▊ ▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊
```

S: Percent of Scan Operations on the Object

To compute S, use the statistics in the DMV sys.dm_db_index_operational_stats [http://technet.microsoft.com/en-us/library/ms174281.aspx] . S is the ratio (expressed in percent) of scans performed on a table or index to the sum of all operations (scans + DMLs + lookups) on that table or index. In other words, S represents how heavily the table or index is scanned. The following query reports S for each table, index, and partition in the database.

Transact-SQL	Copy Code

```
▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊ ▊ ▊ ▊ ▊▊
▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊
▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊▊
▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊
▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊ ▊▊▊▊▊▊ ▊▊▊▊▊▊
▊ ▊ ▊ ▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊
▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊
▊ ▊ ▊ ▊▊▊▊▊▊▊▊▊▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊
▊ ▊ ▊ ▊ ▊ ▊▊▊▊▊▊▊▊▊▊▊▊▊ ▊▊▊
```

Be aware that the counters in the DMV sys.dm_db_index_operational_statsreflect the operational statistics of the tables and indexes in the metadata cache. Carefully consider the timeframe of the statistics used. If you prefer to err on the side of too little compression, use performance statistics during the period where CPU usage is the highest (for example, month-end processing).

For active tables and indexes, the statistics reflected are cumulative since the time the SQL Server service was restarted, or since the last time the database was opened. Less active objects may not have operational statistics in this DMV; however, the less active objects may be good candidates for page compression, provided they are of significant size and the estimated space saving is significant.

Inserts to append-only (inserted at the end of the table) tables that are rarely used, do not have much overhead for page compression. Such tables may perform well with page compression even if S is low. Examples of such tables are logging or audit tables, which are written once and rarely read. These are good candidates for page compression.

Example

Here is an example how a customer used these measurements to decide which tables to page-compress. The customer had an OLTP database running on a server with average CPU utilization of approximately 20 percent. The large amount of available CPU, the significant amount of planned database growth, and the expense of storage provided motivation for data compression. The customer computed space savings, and the U and S measurements for the largest tables in the database, and targeted tables with S greater than 75 percent, U less than 20 percent for page compression. Table 1 shows the estimated row and page savings, the values of S and U, and the decision as to whether to row or page compress.

Table	Savings ROW %	Savings PAGE %	S	U	Decision	Notes
T1	80%	90%	3.80%	57.27%	ROW	Low S, very high U. ROW savings close to PAGE
T2	15%	89%	92.46%	0%	PAGE	Very high S
T3	30%	81%	27.14%	4.17%	ROW	Low S

T4	38%	83%	89.16%	10.54%	ROW	High U
T5	21%	87%	0.00%	0%	PAGE	Append ONLY table
T6	28%	87%	87.54%	0%	PAGE	High S, low U
T7	29%	88%	0.50%	0%	PAGE	99% appends
T8	30%	90%	11.44%	0.06%	PAGE	85% appends
T9	84%	92%	0.02%	0.00%	ROW	ROW savings ~= PAGE
T10	15%	89%	100.00%	0.00%	PAGE	Read ONLY table

Table 1: Deciding what to compress

Based on the metrics shown in Table 1, the customer decided to page-compress tables T2, T5, T6, T7, T8, and T10. All other tables in the database were row-compressed. Following this plan, the customer achieved 50 percent space savings, and approximately 10 percent increase in CPU utilization.

Planning Compression: Estimating Workspace, CPU, I/O

Tables and indexes are compressed using the ALTER TABLE… REBUILD [http://technet.microsoft.com/en-us/library /ms190273.aspx] and ALTER INDEX … REBUILD [http://technet.microsoft.com/en-us/library/ms188388.aspx] statements respectively. Compressing a table or an index requires workspace, CPU, and I/O. Compressing a table or index uses the same mechanism as rebuilding an index. This section provides estimated resource requirements for compressing a clustered index, alongside the resource requirements for rebuilding the same uncompressed index, for comparison. The resource requirements depend upon:

- Whether you are compressing a heap, a clustered index, or a nonclustered index
- Whether you set the SORT_IN_TEMPDB option to ON
- Whether you are performing the compression operation with the ONLINE option set to ON
- Whether you are using the simple, bulk logged, or full recovery model

Workspace

Free workspace is required in the following:

- User database
- Transaction log
- **tempdb**

Estimate the workspace requirements before starting the compression and have enough free space to avoid potentially expensive autogrow of the database files or failed compression due to lack of disk space. Use the script in Appendix C to determine the available free space in each filegroup.

Workspace Required in the User Database

In the user database, free workspace is required for the following:

- The compressed table or index
- The mapping index [http://technet.microsoft.com/en-us/library/ms191261(sql.90).aspx] if you are compressing a heap or a clustered index with the ONLINE option set to ON and the SORT_IN_TEMPDB option set to OFF. (The recommendation is to set SORT_IN_TEMPDB to ON. The workspace requirement for the mapping index is discussed in the **tempdb** section.).

While a table is being compressed, both the uncompressed table and the compressed table exist together until the compression is successful and committed. After the table or the index is compressed, the uncompressed table is dropped, and the space is released to the filegroup. To estimate the size of the compressed table, use the output of sp_estimate_data_compression_savings.

Transaction Log Space

The amount of transaction log space needed depends on whether ONLINE is set to ON or OFF, and the recovery model used (full, bulk-logged, or simple [http://technet.microsoft.com/en-us/magazine/2009.02.logging.aspx]).

Workspace Required in tempdb

In the **tempdb** database, free workspace is required if ONLINE is set to ON:

- For the mapping index [http://technet.microsoft.com/en-us/library/ms191261(sql.90).aspx] , an internal structure used to map old bookmarks to new bookmarks, enabling concurrent DML transactions. This is stored in **tempdb** if SORT_IN_TEMPDB is set to ON.
- For the version store. This is only used if there are concurrent DML operations. Size depends upon the volume of ongoing modifications and duration of long running DML transactions.

I/O

I/O is generally proportional to the workspace used.

CPU

On average, row compression takes 1.5 times the CPU time used for rebuilding an index, whereas page compression takes 2 to 5 times the CPU time used for rebuilding an index. As an example data point, 41 CPU seconds per GB for index rebuild, 48 CPU seconds per GB for row compression, and 182 CPU seconds per GB for page compression were observed with ONLINE set to OFF, on the hardware listed in Appendix A. ONLINE operations require more CPU. These numbers are meant to give a general ballpark, not to be precise, because the performance varies depending upon the characteristics of the data and the hardware.

Rebuilding and compression can be parallelized and take advantage of multiple CPUs. An example of using the MAXDOP option is shown in this blog [http://sqlblog.com/blogs/linchi_shea/archive/2008/05/05/sql-server-2008-page-compression-using-multiple-processors.aspx] . There are two caveats:

- SQL Server uses statistics on the leading column to distribute work amongst multiple CPUs, thus multiple CPUs are not beneficial when creating, rebuilding, or compressing an index where the leading column of the index has relatively few unique values or when the data is heavily skewed to just a small number of leading key values – only limited effective parallelism will be achieved in this case.
- Compressing or rebuilding a heap with ONLINE set to ON uses a single CPU for compression or rebuild. However, SQL Server first needs to scan the table—the scan is parallelized, and after the table scan is complete, the rest of the compression processing of the heap is single-threaded.

Summary of Resource Requirements for Data Compression

Table 2 shows a summary of workspace, CPU, and I/O requirements for compressing a clustered index as compared to rebuilding the same uncompressed index. Measurements used:

- X = number of pages before compression (or rebuild)
- P = number of pages after compression ($P < X$)
- Y = number of new or updated pages (by a concurrent application, applies only to the ONLINE case)
- M = size of the mapping index (estimate based on guidelines in the TEMPDB Capacity Planning white paper)
- C = the CPU time taken to rebuild the uncompressed index

	Workspace			I/O			CPU
	TEMPDB	**UserDB**	**UserDB Tran Log**	**TEMPDB**	**UserDB**	**UserDB Tran Log**	
OFFLINE with BULK_LOGGED or SIMPLE Recovery Model							
Rebuild	0	X	~0	0	X+2X	~0	C
Compress	0	P	~0	0	X+2P	~0	1.5C to 5C
OFFLINE with FULL Recovery Model							

Rebuild	0	X	X	0	X+X	X	~C
Compress	0	P	P	0	X+P	P	1.5C to 5C
ONLINE with FULL Recovery Model							
Rebuild	M+Y	X+Y	2X+Y	M+4Y	X+X+Y	2X+Y	~2C
Compress	M+Y	P+Y	2P+Y	M+4Y	X+P+Y	2P+Y	3C to 10C

Table 2: Workspace, CPU, and I/O summary for compressing a clustered index

Some important notes from Table 2:

- Comparing to rebuild without compression, compression uses less workspace and I/O, but more CPU. The reduced workspace/I/O is due to the smaller size of the resulting structure.
- The lowest resource utilization is when running OFFLINE with the bulk-logged or simple recovery model. Offline, bulk operations involve reading the existing data, and writing new, compressed data. Minimal log is written—only the allocations are logged.
- If it is not possible to use the BULK_LOGGED or SIMPLE recovery model (but still using OFFLINE), additional I/O is needed to fully log the compressed data, with the full recovery model. No additional workspace (compared to bulk-logged) is allocated, because log space is reserved even in the bulk-logged and simple recovery models.
- Many production systems cannot run in bulk-logged or simple mode, and many systems cannot afford downtime for OFFLINE compression. ONLINE operations require about twice as much CPU as OFFLINE operations.

How and When to Compress

The key decisions to be made on when and how to compress are:

- Online vs. Offline: Whether to set the value of ONLINE to ON or OFF depends upon what else is running on the database at the same time. OFF is faster and requires less resources than ON, but the table is locked for the duration of the compression operation. Be aware of the restrictions of online operations as discussed in SQL Server Books Online [http://msdn.microsoft.com/en-us/library/ms190981.aspx] .
- One table, index, or partition at a time vs. many concurrently: Compressing one table, index, or partition at a time is recommended in most cases. When doing multiple simultaneous compressions, workspace, I/O, and CPU requirements outlined earlier must be available for all the compressions together. Be mindful of the risk of insufficient free space and the need to expand data files, as well as the likelihood of large amount of unused space in the data files at the end of compression. If you have sufficient resources (workspace, I/O, and CPU), and have an efficient mechanism to reclaim the unused space (refer to section 6 later in this white paper), executing multiple compressions simultaneously may be acceptable.
- Order of compressing the tables: After you have decided on the list of tables and indexes to compress, compress the objects starting with the smallest in this list. Compressing smaller objects requires less workspace, and it releases space to the data files that can be used as workspace for compressing the larger objects subsequently. This approach minimizes the need for additional disk space during the compression process.
- Setting SORT_IN_TEMPDB to ON or OFF: ON is recommended. This makes use of **tempdb** space for the mapping index, and therefore, it requires less workspace in the user database.

Side Effects of Compressing a Table or Index

When you compress a table or an index, you should be aware of two side effects:

- Compression includes a rebuild, thus removing fragmentation from the table or index.
- When a heap is compressed, if there are any nonclustered indexes on the heap, they are rebuilt as follows:

o   With ONLINE set to OFF, the nonclustered indexes are rebuilt one by one.

o   With ONLINE set to ON, all the nonclustered indexes are rebuilt simultaneously.

You must account for the workspace required to rebuild the nonclustered indexes, because the space for the uncompressed heap is not released until the rebuild of the nonclustered indexes is complete.

Manipulating Compressed Data

This section explains what happens when compressed data is changed: new rows inserted, rows deleted or updated.

Newly-Inserted Rows

With row compression, newly inserted rows are row-compressed. With page compression, a newly inserted row is row-compressed or page-compressed, depending upon the following:

- The table organization: heap or clustered index
- How and where the new row is inserted

Table 3 summarizes the compression state of the newly inserted rows into a compressed table.

Table organization	Table compression setting	
	ROW	PAGE
Heap	The newly inserted row is row-compressed.	The newly inserted row is page-compressed: · if new row goes to an existing page with page compression · if the new row is inserted through BULK INSERT with TABLOCK · if the new row is inserted through INSERT INTO ... (TABLOCK) SELECT ... FROM Otherwise, the row is row-compressed.*
Clustered index	The newly inserted row is row-compressed.	The newly inserted row is page-compressed if new row goes to an existing page with page compression Otherwise, it is row compressed until the page fills up. Page compression is attempted before a page split.**

Table 3: Compression of the newly inserted rows into a compressed heap or clustered index

* The resulting row-compressed pages can be page-compressed by running a heap rebuild with page compression.

** With page compression, all the pages in the table may not actually be page-compressed. A page is page-compressed only if the space savings on that page exceeds an internally defined threshold.

Updating or Deleting Compressed Rows

All updates to the rows in a row-compressed table or partition will maintain the rows in row-compressed format. Not every update to the rows in a page-compressed table or partition will cause the column prefix and page dictionary to be recomputed. When the number of changes on a given page-compressed page exceeds an internally defined threshold, the column prefix and page dictionary are recomputed.

What Happens to the Supporting Data Structures

When an application manipulates (INSERT, UPDATE, DELETE, CREATE/REBUILD INDEX, and so on) data in a table, some supporting data structures may be created by SQL Server, which may temporarily hold a subset of the data. Some of these data structures are:

- Transaction log
- Mapping index
- Version store
- Sort pages

Whether or not these supporting structures are compressed if the data in a compressed table is manipulated depends upon the type of the data structure and the type of data compression used on the table. Table 4 summarizes the compression characteristics of the supporting data structures when a compressed table is manipulated.

Table compression	Transaction log	Mapping index for rebuilding the clustered index	Sort pages for queries	Version store (with SI or RCSI isolation level)
ROW	ROW	NONE	NONE	ROW
PAGE	ROW	NONE	NONE	ROW

Table 4: Compression characteristics of supporting data structures when data in a compressed table is modified

On a Page-Compressed Index, Nonleaf Pages are Row-Compressed

All updates to the rows in a row-compressed table or partition will maintain the rows in row-compressed format. Not every update to the rows in a page-compressed table or partition will cause the column prefix and page dictionary to be recomputed. When the number of changes on a given page-compressed page exceeds an internally defined threshold, the column prefix and page dictionary are recomputed.

On a page-compressed index (clustered or nonclustered), the leaf level pages are page-compressed; but the nonleaf pages are row-compressed, not page-compressed (see Appendix D). This is for efficiency reasons.

- The number of nonleaf pages in an index are relatively small, and the space saving that would result from page-compressing these pages is relatively insignificant.
- The nonleaf pages are accessed much more frequently, and having them row-compressed (instead of page-compressed) reduces the cost of decompressing them on each access.

Reclaiming the Space Released by Data Compression

After data compression has completed, the space saved is released to the respective data file(s). However, the space is not released to the filesystem, because the file size doesn't reduce automatically as part of data compression. There are several options to release the space to the filesystem by reducing the size of the file(s):

Option 1: You can decide not to reclaim the released space, keeping the free space in the filegroup for the future data growth. This is a simple option for databases where the data volume within the existing filegroups is expected to grow in the future. It is not an option for partitioned tables where each partition is allocated on a different filegroup, and you want to compress the older read-only partitions to save disk space.

Option 2: DBCC SHRINKFILE (or DBCC SHRINKDATABASE) is an option, but shrinking a database file severely fragments its contents. Also be aware that DBCC SHRINKFILE is single-threaded and may take a long time to complete. In a test, after a clustered index was page-compressed, a filegroup had about 68 percent free space. After DBCC SHRINKFILE, the fragmentation became 100 percent.
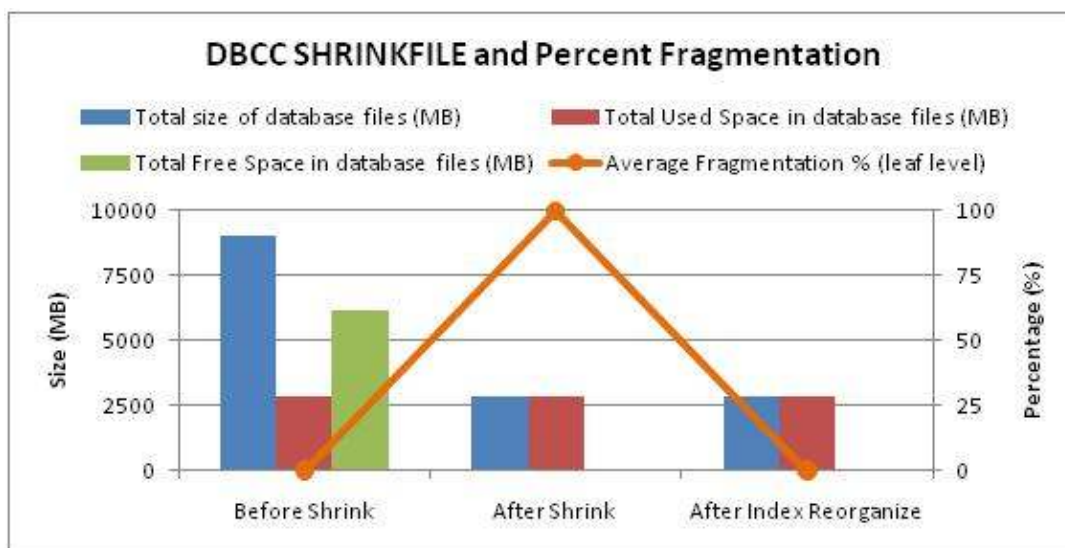
Figure 1: Database free space and percent fragmentation after DBCC SHRINKFILE

After ALTER INDEX … REORGANIZE, the fragmentation reduced to 0. Use REORGANIZE (which does not require additional data file space), and not REBUILD (which creates a new index and then drops the existing index, requiring more workspace in the filegroup and extending the data file again).

Option 3: If compressing all the tables in a filegroup:

 - Create a new filegroup.

 - Move tables and indexes to the new filegroup while compressing.

　- You need to create (or re-create, if one already exists) a clustered index (to move a heap, refer to Option 4) on the table and move it to the new filegroup in one operation. Use the DROP_EXISTING option of CREATE CLUSTERED INDEX command, if you already have a clustered index. For example:

Transact-SQL　　Copy Code

```
&5($7[ B1,48( K/867(5('1,1'(;B3.B75$'(@
21075$'(B%8/.0D7B.'6&1
: ,7+ II'$7$B&2035(66,21 3$*[ II'523B(;67,1* 21I6257B1B7(03'% 21I21D)*B'0018
```

 - After all the tables and indexes from the old filegroup have been compressed and moved to the new filegroup, the old filegroup and its file(s) can be removed to release space to the filesystem.

 - Be aware of a caveat in this method. If the table has a LOB_DATA allocation unit in the same filegroup, then this method will not move the LOB_DATA to the new filegroup (only the IN_ROW_DATA and ROW_OVERFLOW_DATA allocation units are moved when a clustered index is re-created in a different filegroup). So the old filegroup will not be completely empty, and hence cannot be dropped.

Option 4: There is another solution if you are compressing all the tables in a filegroup. Create an empty table in the new filegroup, compress it, and then copy the data over to the new table using INSERT … SELECT.

Transact-SQL　　Copy Code

```
III& 1HDWHD(1HZ HP SW\H0E6(1Q H7H1QHZ IDDHJURXS
$/7(5 I'$7$%$6( D7HWV %00 2 ' ,)<1)/(*52831)*B&203I' ()$8/7I
6(/(&7II,172 D7DE1@) 520 D7DE@: +(5( III III
III& RP SI1HVWHD(1HZ 0.(UHDWHG HP SW\H0E11
$/7(5 I7$%/( D7DE111$(%8/' I7 ,7+ I'$7$B&2035(66,211 I3$*(11
III& 1HDWHHD(1$SSURSUIDHLQGH[HVRQ 1H1H011
III& RS\1HH1DO01RYHU01H1HQHZ 101E11
,16(571,172 D7DE1@: ,7+11$B/2& .1 I6(/(&7111 520 D7DE@
III7KHQHQFRP IQJ1DN01Q HEHFRP SHVVHD DVH1H4JHWWQVHLUG
II17$B/2& . DI'HIXXLHG1HIH H0UHM1D0E1E/B HPS
III7UDFHHDJ1H I III P D\EHS HQDE1E I1QP DDUJJQJ
III7KHI/2B%D'$7$ DIFD7HUQ1QQM11 011DYRE1HIRSD1@
III' 1RS11HH101E1HDQGH1HQDP HHH1HQHZ 10E1HDVRE1H01E11
III& 1UDD1111H1HE1M1DH111FRSHG10H1HKII/1LH P R1H1HIR11111H1HJURXS
```

Application Performance with Data Compression

As discussed earlier, data compression reduces logical and physical I/O, but it increases CPU consumed. In this section we discuss some workloads and their performance behavior under data compression.

Workload 1: An OLTP Application with High Volumes of DML Operations

Figure 2 shows the performance achieved by a customer on an OLTP application with high volumes of DML (INSERT, UPDATE, and DELETE) operations. The average response time of four different types of business transactions were measured with NONE, ROW, and PAGE compression.
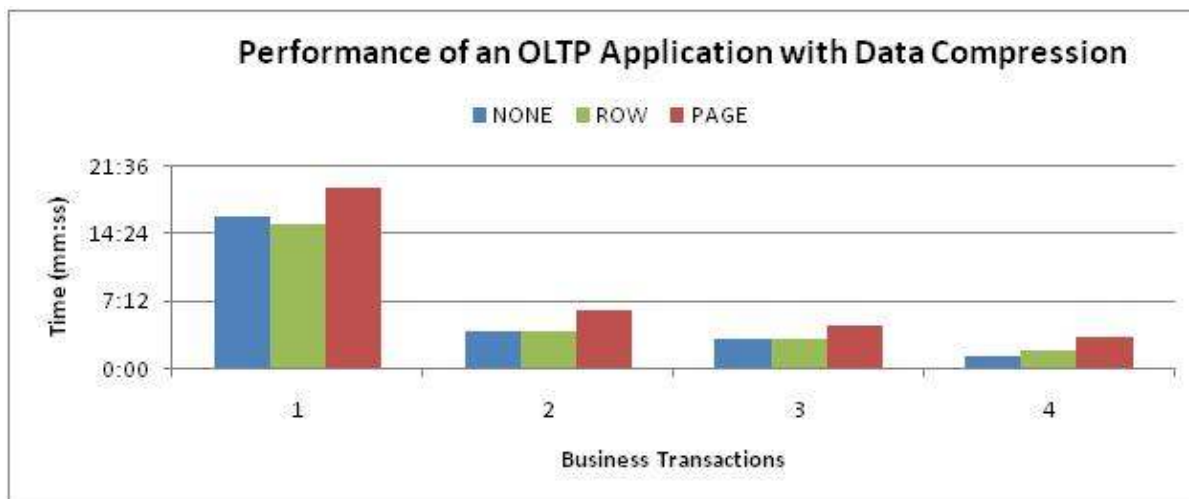
Figure 2: A Customer OLTP workload performance with data compression

As illustrated in Figure 2, this workload achieved similar or better performance with row compression as compared to no compression, for all business transactions, except the fourth one. However, with page compression, all the business transactions took longer to complete as compared to no compression or row compression. Based on these tests, this customer decided to use row compression with all their tables and indexes in this application.

Workload 2: A Reporting Application with Large Queries

Figure 3 shows the performance achieved by a customer on a reporting application. The response times of eight queries were measured with no compression and page-compressed data in the customer's test environment.
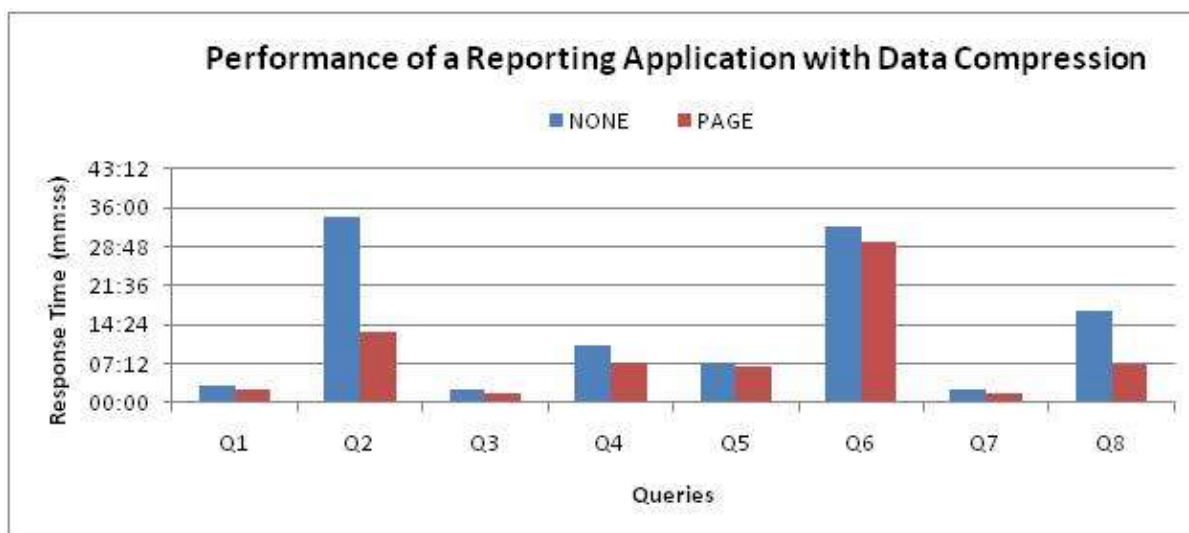
Figure 3: A Customer reporting workload performance with data compression

As illustrated in Figure 3, this workload achieved better performance for all queries with page compression, as compared to no compression. For some queries the response time was cut in half or better. Based on these tests, this customer is planning to deploy page compression in production for this application.

Rebuilding Compressed Indexes

Rebuilding compressed indexes takes longer compared to rebuilding corresponding uncompressed indexes. Note that if the table is compressed, but the index (nonclustered) is not compressed, rebuilding that index will not require additional processing time.

Rebuilding a compressed index involves decompressing the index pages and then rebuilding the index with compression. This requires longer time and more CPU resources. Figures 4 and 5 show the CPU usage and time taken to rebuild a clustered and a nonclustered index, respectively.
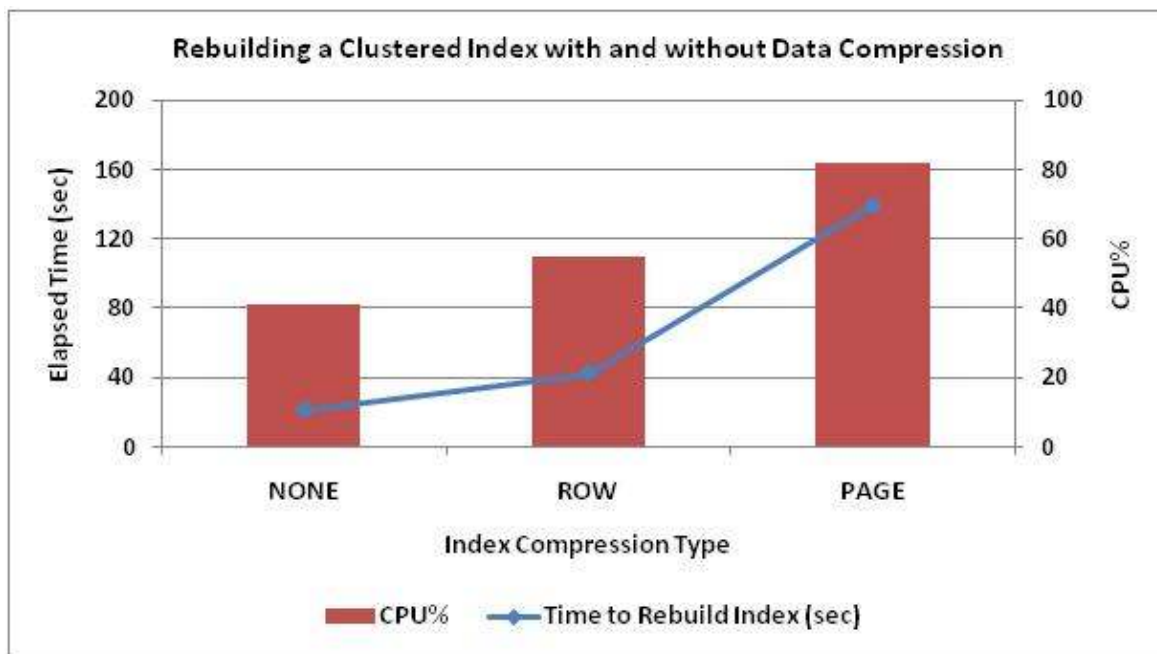
Figure 4: CPU usage and time taken to build a clustered index
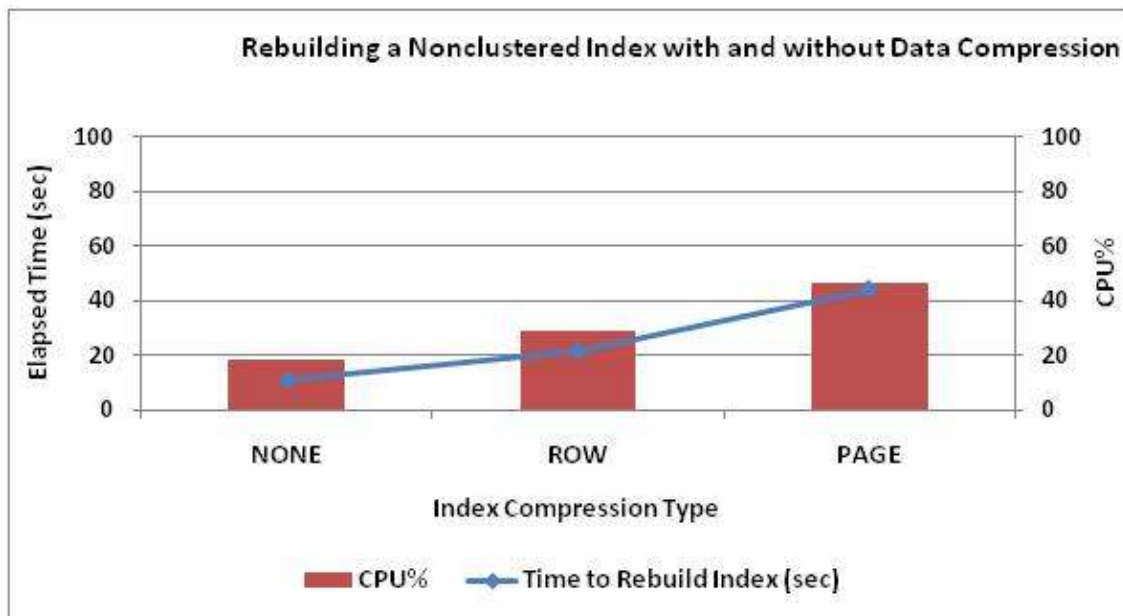
Figure 5: CPU usage and time taken to build a nonclustered index

Note the significant increase in the CPU usage for rebuilding the page-compressed index. The reason for the high CPU usage is that when you rebuild a page-compressed index, the compression information (prefix and dictionary) on each (leaf-level) page is recomputed.

BULK INSERT with Data Compression

Bulk loading data into a compressed table involves compressing the data while performing the load. Therefore, BULK INSERT takes longer on a compressed table. Figure 6 shows the performance of BULK INSERT on a heap with and without data compression. The first three data points display BULK INSERT performance without the TABLOCK option. Notice that without the TABLOCK option, the size of the compressed heap table is the same with both row and page compression. If the TABLOCK option is not used while performing BULK INSERT on a page-compressed heap, the newly inserted pages will be row-compressed, instead of being page-compressed (refer to Table 3 in section 5.1). Use the query in Appendix E to confirm.
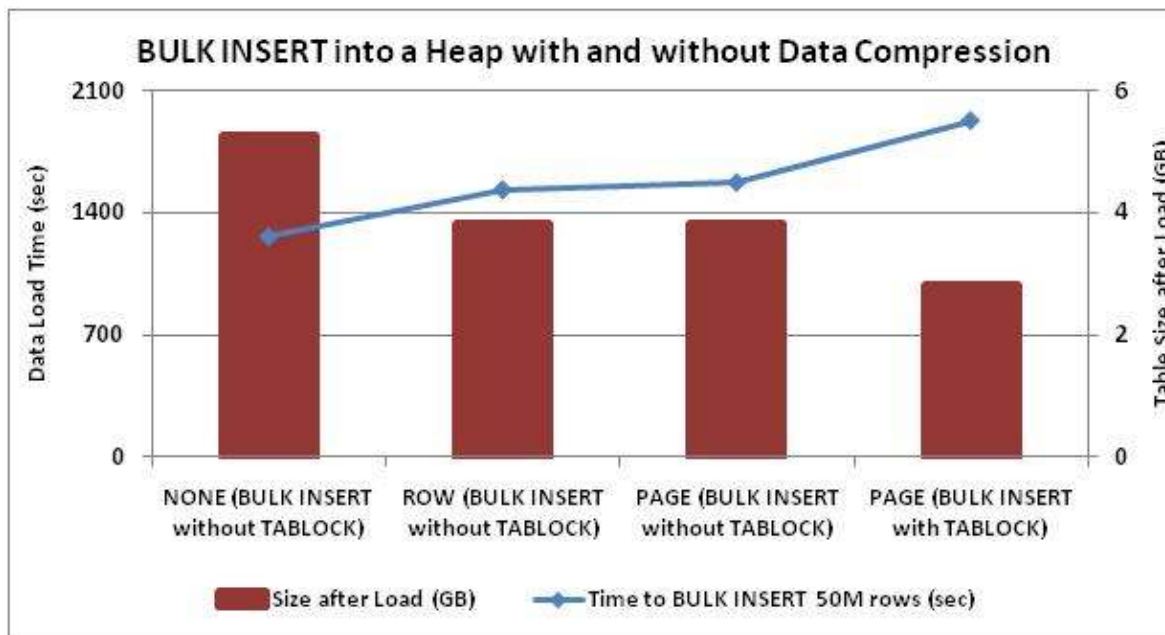
Figure 6: BULK INSERT into a heap with and without data compression

The last data point in Figure 6 shows the BULK INSERT performance (on a page-compressed heap) with the TABLOCK option. Note that the TABLOCK option results in a smaller table size, because almost all the pages are page-compressed during the load. Therefore, remember to use the TABLOCK hint if you are loading data into a page-compressed heap.

The same considerations apply if you are performing an INSERT … SELECT operation into a page compressed heap. Be sure to use the TABLOCK hint to ensure that new pages are compressed.

BULK INSERT Followed by CREATE CLUSTERED INDEX

In many bulk loading scenarios, loading data is typically followed by creating a clustered index. In a sliding window scenario, usually new data is loaded into an empty staging table, and then a clustered index (and other appropriate indexes and constraints) is created on the staging table, to make it ready for switching into an empty partition in a partitioned table. If you are loading data into an empty table and then creating a clustered index, and the data needs to be compressed, there are multiple options:

- Option 1: BULK INSERT into uncompressed heap, followed by CREATE CLUSTERED INDEX WITH (DATA_COMPRESSION = PAGE). Because the data is loaded into an uncompressed heap, it allows for faster loading compared to the other two options. This option allows compressing the data at the same time as creating the clustered index, thereby reducing the total time. However, more free space is needed in the user database compared to option 3, because, the uncompressed heap and the compressed clustered index need to reside in the user database simultaneously while the index is being created.
- Option 2: BULK INSERT into a page-compressed heap, followed by CREATE CLUSTERED INDEX. Because the data is loaded into a heap, the loading is faster compared to option 3; however, because the heap is compressed, loading takes longer compared to option 1 (the data is compressed while being loaded). And, because the heap and the clustered index need to reside in the user database while the index is created, more free space is needed than option 3; but less than option 1, because both the heap and the clustered index are compressed.
- Option 3: BULK INSERT into a page-compressed clustered index. This option takes longer, because the data is loaded into a clustered index, and data is compressed during loading, but all the tasks (loading, compressing, creating clustered index) are completed together. Because there is no post-processing involved in the form of creating the clustered index or compressing the data, no extra free space is required in the user database.

Figure 7 illustrates the time required for bulk loading data, creating a clustered index, and compressing the data. Figure 8 illustrates the workspace required for these tasks.
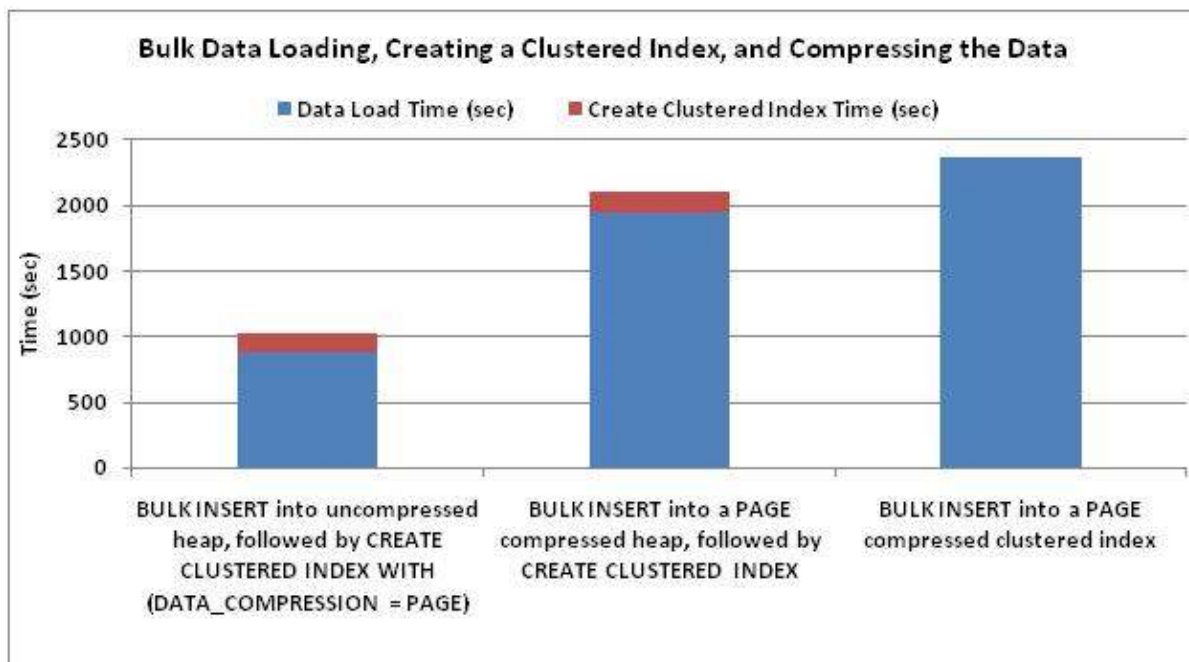
Figure 7: Time required for bulk data loading, creating a clustered index, and compressing the data (simple recovery model, ONLINE = OFF, SORT_IN_TEMPDB = ON)
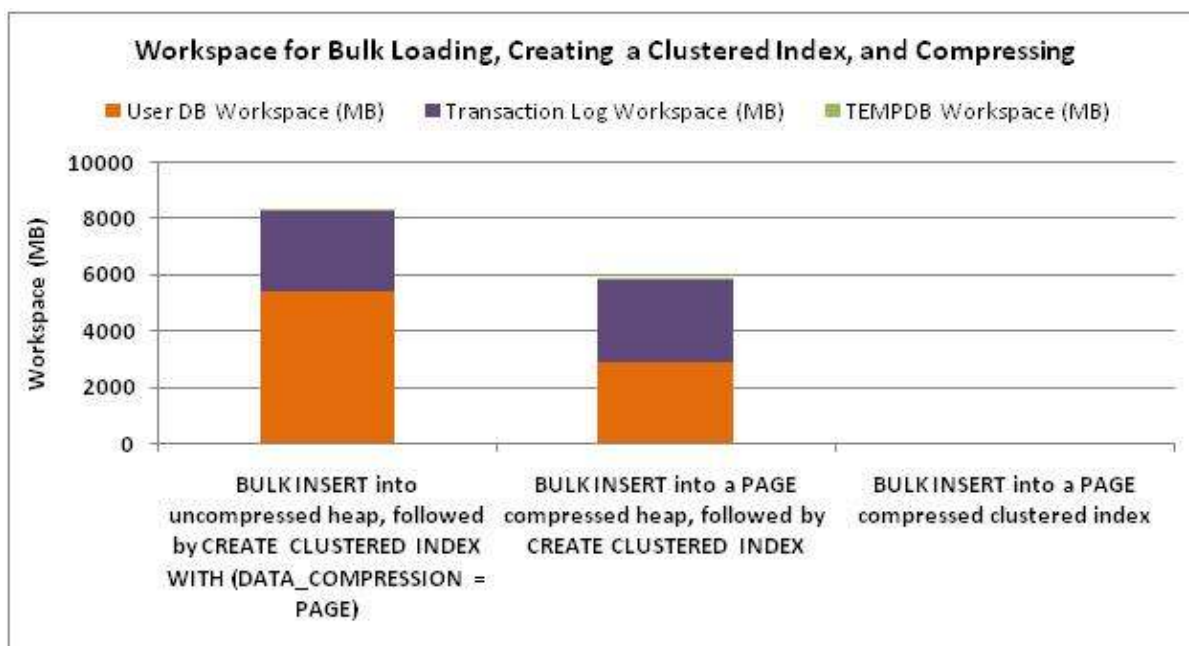
Figure 8: Workspace required for bulk data loading, creating a clustered index, and compressing the data (simple recovery model, ONLINE = OFF, SORT_IN_TEMPDB = ON)

Data Compression and Partition Manipulation

Partitioning provides flexibility for compressing data selectively. Each partition in a table can have a different compression setting. Be aware of the compression settings on partition manipulation operations, such as switch, split, and merge.

Switch

Switching a partition requires that the source and the target have the same compression setting. The target of the switch partition operation is always an empty partition (or table). Changing the compression setting of an empty partition or table is very quick, because this is a metadata-only operation. Therefore, check and change the compression setting of the target, if needed, prior to executing the switch command.

Split

New partition inherits the data compression property of the partition being split.

Merge

Merging two partitions essentially removes the boundary between the two partitions, thereby dropping one partition and moving all the data in that partition to the other partition. Therefore merging partitions involves a source partition (the partition being dropped) and a destination partition (the partition into which the data from the dropped partition is moving). The destination partition retains its compression property.

If the destination partition's compression setting is NONE, the data coming in from the source partition will be decompressed during the merge operation. If the destination partition's compression setting is ROW, the data coming in from the source partition will be row-compressed during the merge operation. If the destination partition's compression setting is PAGE, the data coming in from the source partition will be either row-compressed (if the table is a heap) or page-compressed (if the table is a clustered index), as shown in Table 5.

Compression setting of the destination partition	What happens to the data moving in from the source to the destination partition
NONE	The incoming data is decompressed during merge
ROW	The incoming data is row-compressed during merge
PAGE	- Heap: The incoming data is row-compressed during merge - Clustered index: The incoming data is page-compressed during merge

Table 5: Partition merge and data compression

During a partition merge operation, the source partition and the destination partition are identified based on the partition function used. The example in Figure 9 illustrates the compression behavior of partition merge for LEFT and RIGHT partition functions.

```
This color represents a compressed partition

This color represents an uncompressed partition

-- Create a partition table with a LEFT partition function
CREATE PARTITION FUNCTION PF_LEFT(INT) AS RANGE LEFT FOR VALUES (1,2)
```

< = 1	> 1, < = 2	> 2

```
-- Merge two partitions
ALTER PARTITION FUNCTION PF_LEFT() MERGE RANGE (1)
```

< = 2	> 2

The resulting partition is uncompressed. The partition boundary RANGE (1) belongs to the left partition, and because we are merging RANGE (1), we are effectively removing that boundary. The data is moving from the left partition (source) to the right partition (destination), and hence inheriting the compression characteristics of the destination (right) partition; and thereby getting decompressed.

```
-- Create a partition table with a RIGHT partition function
CREATE PARTITION FUNCTION PF_RIGHT(INT) AS RANGE RIGHT FOR VALUES(1,2)
```

< 1	>= 1, < 2	> = 2

```
-- Merge two partitions
ALTER PARTITION FUNCTION PF_RIGHT() MERGE RANGE (1)
```

< 2	>= 2

In this case, the resultant partition is compressed. Because the partition function is defined with RIGHT, the partition boundary RANGE (1) belongs to the right partition. When we are merging RANGE (1), (and effectively removing that boundary), data is moving from the right partition (source) to the left partition (destination), and hence inheriting the compression characteristics of the destination (left) partition; and thereby getting compressed. Note that if the table is a heap, the incoming data from the source partition will be effectively row-compressed.

Figure 9: Data compression and partition manipulation

Some partitioning scenarios in large data warehouses dedicate an entire separate filegroup to each partition, so that individual partitions can be marked as read-only to minimize backup requirements. In this scenario, to avoid using DBCC SHRINKFILE, it is important that a partition is initially loaded as compressed, as discussed in Section 7. For heaps, this requires that the partition is bulk loaded using the TABLOCK hint, and for clustered indexes, it means that the compressed clustered index should be in place as the data is loaded, rather than created later, leaving unused empty space in the filegroup.

Data Compression and Transparent Data Encryption

Transparent data encryption (TDE) is another very useful feature in SQL Server 2008. TDE provides encryption of data in a database at the storage level without requiring any application changes. A common question related to this is "How does data compression perform against an encrypted database?"

TDE is transparent to data compression. Figure 10 displays the amount of time it takes to page-compress a clustered index, and the amount of space savings obtained by page compression, with and without TDE. As illustrated in Figure 10, TDE has negligible, if any, impact on data compression in either compression time or space saved.
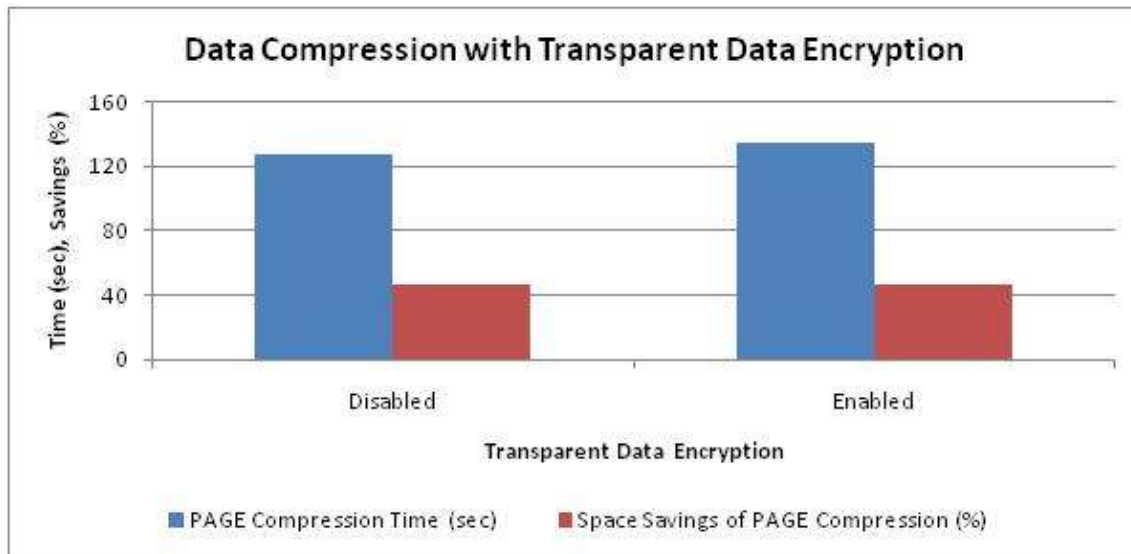
Figure 10: Data compression with transparent data encryption

TDE encrypts the pages when they are written to disk and decrypts them when they are read from disk into memory. Because data compression (as well as decompression) is performed on in-memory pages, data compression always sees unencrypted data, and hence the effectiveness and efficiency of data compression is not impacted by TDE.

Conclusion

Data compression provides multiple benefits. It saves disk space, and it can help improve the performance of certain workloads. The benefits of data compression come at the cost of higher CPU usage for compressing and decompressing the data. Therefore, it is important to understand the workload characteristics on a table before deciding on a compression strategy. Data compression provides flexibility in terms of levels of compression (row or page) and the objects you can compress (table, index, partition). This enables fine-tuning the compression based on the characteristics of data and the workload.

Another important advantage of data compression is that it works transparently to the application, and it works well with other SQL Server features, such as TDE and backup compression.

The results shown in this white paper are based on the data and the hardware used in our tests. Your results will vary based on your own data, workload and hardware. Perform thorough testing when deciding what tables and indexes to compress.

For more information:

http://www.microsoft.com/sqlserver/ [http://www.microsoft.com/sqlserver/default.aspx] : SQL Server 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:%20Data%20Compression:%20Strategy,%20Capacity%20Planning,%2
] .

Appendix A: Test Hardware and Software

All the tests (except those in Figure 2 and Figure 3, which were run in customer test environments) were performed on the following hardware and software environment.

Server

DELL R805 with:

- 2 socket quad core
- AMD Opteron Processor 2354 @2.20 GHz
- 32 GB RAM

Storage

EqualLogic iSCSI Storage

3 Disk shelves – Each Containing 14 SAS drives. Each drive had:

- 300 GB
- 15K rpm
- RAID 1+0

Software

- The 64-bit edition of the Windows Server 2008 Enterprise operating system
- The 64-bit edition of SQL Server 2008 Enterprise

Appendix B: Data Volume in Each Allocation Unit Type

A table or a partition can have three allocation units - IN_ROW_DATA, LOB_DATA and ROW_OVERFLOW_DATA [http://msdn.microsoft.com/en-us/library/ms189051.aspx] . Usually, most of the data in the table is stored in the IN_ROW_DATA [http://technet.microsoft.com/en-us/library/ms189087.aspx] allocation unit. Depending upon the row size and the table options, some data can be stored outside the row in ROW_OVERFLOW_DATA [http://technet.microsoft.com/en-us/library/ms186981.aspx] or LOB_DATA allocation units. Use the following script to determine how much data is stored in each of the three allocation units.

Transact-SQL	Copy Code

```
[illegible code]
```

Appendix C: Free Space in Database Files

Transact-SQL	Copy Code

```
[illegible code]
```

Appendix D: On a Page-Compressed Index, Verifying That Nonleaf Pages

are Row-Compressed

Use the following query on a page-compressed index (clustered or nonclustered) to verify that the leaf-level pages are page-compressed; but the nonleaf pages are row-compressed, not page-compressed.

Transact-SQL   Copy Code

```
(garbled code block)
```

The output of this query on a page-compressed clustered index looks like the following.

Name	index_type_desc	partition_number	data_compression_desc	index_level	page_count	compressed_p
TRADE_BULK	CLUSTERED INDEX	1	PAGE	0	370508	370502
TRADE_BULK	CLUSTERED INDEX	1	PAGE	1	830	0
TRADE_BULK	CLUSTERED INDEX	1	PAGE	2	5	0
TRADE_BULK	CLUSTERED INDEX	1	PAGE	3	1	0

Table 6

Observe the columns index_level and compressed_page_count in the output. Note that almost all the leaf pages (index_level = 0) are page-compressed; but none of the nonleaf pages (index_level >=1) are page-compressed (compressed_page_count column is 0 for nonleaf level pages).

Appendix E: Test Hardware and Software

The following query can be used to determine how many pages in a table, index, or partition are page-compressed.

Transact-SQL   Copy Code

```
(garbled code block)
```

After performing BULK INSERT on a page-compressed heap, the output of the above query will look similar to the following.

name	index_type_desc	partition_number	data_compression_desc	page_count	compressed_page_count
TRADE_BULK	HEAP	1	PAGE	501574	0

Table 7

Note the columns data_Compression_desc and compressed_page_count. The column data_compression_desc in sys.partitions shows the data compression setting (metadata) for a given table, index, or partition. To get the actual number of page-compressed pages, use the column compressed_page_count in the dynamic management function (DMF) sys.dm_db_index_physical_stats. Note that in this output, even though the data compression setting of the heap is PAGE, the newly loaded pages are not page-compressed.

After BULK INSERT is performed with TABLOCK on a page-compressed heap, the output of this query will look similar to the following.

name	index_type_desc	partition_number	data_compression_desc	page_count	compressed_page_count
TRADE_BULK	HEAP	1	PAGE	370658	370656

Table 8

Note that almost all the pages are page-compressed. Remember to use the TABLOCK hint if you are loading data into a page-compressed heap, or if you are performing an INSERT … SELECT operation into a page-compressed heap.

Note: The compressed_page_count column is displayed only when the DETAILED mode is used with the sys.dm_db_index_physical_stats DMF. When the LIMITED mode (the default) is used, the output of this column is NULL. Be careful when using the DETAILED mode, because it scans all the pages in the table, and it takes a significant amount of time on large tables. When you use the DETAILED mode, we recommended that you specify a value for object_id; do not specify NULL for this parameter. Specifying object_id as NULL (an invalid or nonexistent object reference to object_id translates to NULL) will scan all the pages in all the tables in the database, and it may significantly impact performance.

Appendix F: Additional References

SQL Server Storage Engine Team Blog [http://blogs.msdn.com/sqlserverstorageengine.aspx]

SAP on SQL Server blog on data compression [http://blogs.msdn.com/saponsqlserver/archive/2008/11/24/procedure-used-for-applying-database-compression-to-microsoft-sap-erp-system.aspx]

SQL Server 2008 Technologies for SAP Solutions [https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/60a236a2-8104-2b10-5ebe-8fef61cc82fd]

HP white paper on SQL Server data compression [http://h71028.www7.hp.com/ERC/downloads/4AA1-8766ENW.pdf]

Unisys white paper on SQL Server data compression [http://www.unisys.com/eprise/main/admin/corporate/doc/services/Unisys_SQL_Compression_white-paper_082008.pdf]

NetApp white paper on SQL Server data compression [http://www.netapp.com/us/library/technical-reports/tr-3719.html]

Linchi Shea's blog on data compression [http://sqlblog.com/blogs/linchi_shea/archive/2008/05/05/sql-server-2008-page-compression-using-multiple-processors.aspx]

Paul Nielsen's blog on data compression [http://sqlblog.com/blogs/paul_nielsen/archive/2008/03/13/whole-database-data-compression-procs.aspx]

Kalen Delaney's blog on data compression [http://sqlblog.com/blogs/kalen_delaney/archive/tags/compression/default.aspx]

Microsoft Case Study on Data Compression [https://www.microsoft.com/casestudies/Case_Study_Detail.aspx?casestudyid=4000003962]