Comparing Tables Organized with Clustered Indexes versus Heaps

SQL Server Best Practices Article

Published: May 25, 2007

Writers: Burzin Patel, Sanjay Mishra

Contributors: Kohei Ueda, Michael Thomassy

Technical Reviewers: Stuart Ozer, Prem Mehra, Sunil Agarwal, Artem Oaks, Tengiz Kharatishvili, Mike Ruthruff, Hermann Daeubler, Tom Davidson, Eric Jacobsen, Lindsey Allen

Applies To: SQL Server 2005 SP1

Summary: In SQL Server 2005, any table can have either clustered indexes or be organized as a heap (without a clustered index.) This white paper summarizes the advantages and disadvantages, the difference in performance characteristics, and other behaviors of tables that are ordered as lists (clustered indexes) or heaps. The performance for six distinct scenarios where DML operations are performed on these tables are measured and detailed observations presented. This white paper provides best practice recommendations on the merits of the two types of table organization, along with examples of when you might want to use one or the other.

On This Page

Introduction

In Microsoft® SQL Server™, any table may or may not have a clustered index, and may have zero or more nonclustered indexes. The data rows of a table that has a clustered index are organized as an ordered list in the order specified by the clustered index columns. The data rows of a table without a clustered index are not organized in any particular order and are commonly referred to as *heaps*.

As can be expected, there are both clear advantages and disadvantages of organizing tables as ordered lists or as heaps. This white paper summarizes the advantages and disadvantages, the differences in performance characteristics, and other behaviors of these tables. The performance in six distinct scenarios where DML operations are performed on these tables is measured and detailed observations presented. This white paper also provides best practice recommendations on the merits of the two types of table organization, along with examples of when you might want to use one or the other.

Top of page

Clustered Indexes and Heaps

Clustered indexes and heaps are two different ways to organize the data in tables. A clustered index consists of index pages as well as data pages. This means that, although the name *clustered index* suggests that this is an index, it is not just an index, but also contains the table data. A clustered index is organized as a B-tree, where the nonleaf nodes are index pages and the leaf nodes are data pages. The data in the clustered index is ordered with respect to the column(s) constituting the clustered index. Pages at any level (whether leaf or nonleaf) in the B-tree are linked to the previous and next pages at the same level. A table can only have one clustered index and the index is always identified by **index_id** = 1 in the catalog tables (such as **sys.indexes**, **sys.partitions**, and so on). For more information on the organization of clustered indexes, see Clustered Index Structures in SQL Server 2005 Books Online.

A heap consists only of data pages. Neither the data pages nor the physical placement of the pages are guaranteed to be in any particular order. A heap is always identified by **index_id** = 0 in the catalog tables. For more information on the organization of heaps, see Heap Structures in SQL Server 2005 Books Online.

Whether it is organized as a heap or a clustered index, a table can have zero or more nonclustered indexes. A nonclustered index is organized as a B-tree. Unlike a clustered index, a nonclustered index consists of only index pages. The leaf nodes in a nonclustered index are not data pages, but contain row locators for individual rows in the data pages. A nonclustered index is identified by an **index_id** that is greater than one in the catalog tables. For more information on the organization of nonclustered indexes, see Nonclustered Index Structures in SQL Server 2005 Books Online.

Top of page

Test Objectives

The objective of our testing was to characterize the performance and behavior of DML operations performed against the same set of table data organized:

- As a heap with a nonclustered index on a specified set of columns.
- With a clustered index on the same set of columns, and no other nonclustered index.

The behavior of a table without any indexes, or with two or more indexes is outside the scope of this paper.

The basic questions we wanted to answer were:

- Are clustered indexes necessary for all tables?
- What are the performance gains or losses for row-by-row INSERT, UPDATE, DELETE, and SELECT operations executed against a large table with a clustered index versus the same table without a clustered index (a heap) for a high-throughput workload?
- How does a range query perform on a table with a clustered index versus on a table with a corresponding nonclustered index?
- What are the effects of having the first column of an index be monotonically increasing? The purpose of this test was to measure performance and to quantify the maximum number of rows that can be concurrently inserted without bottlenecking on latch contention caused by 'hot-spot' effects at the insert location.
- What are the space utilization characteristics when rows are INSERTed and DELETEd from a table with a clustered index and from a table without a clustered index (a heap)?

Top of page

Test Methodology

Our goal was to conduct the tests described in the previous section against a workload that represented real-world scenarios as closely as possible. Another goal was to keep the test setup (server configuration, database settings, table schema, and so on) relatively constant across the tests so that we could compare and contrast performance between the different operations.

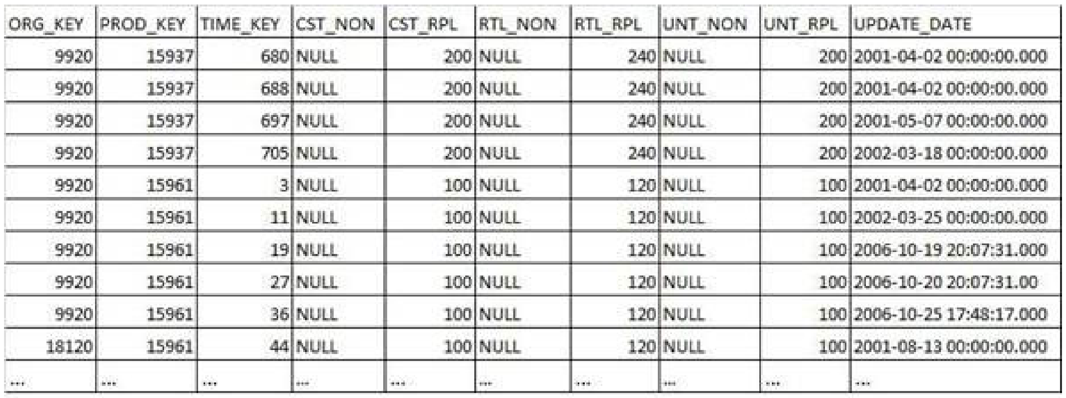
After some initial testing and analysis using a real-world workload, we quickly realized that, while the real-world database we had perfectly suited our needs, the associated workload did not. This was because the workload executed a nondeterministic set of operations, which was hard to quantify and keep constant across runs in a way that would make the results meaningful and applicable to a wide variety of other workloads.

Based on this finding, we decided to continue using the database but substitute the real-world workload with a set of individual tests that performed operations similar to the original workload, but in isolation. In this model, we created a light-weight client program that executed a set number of specific Transact-SQL operations in a loop (row-by-row operation). This test methodology was used for the tests explained in Test Results and Observations later in this paper. Our intent is that these individual tests will help you estimate the overall impact of the index choices for your particular application, based on the mix of Transact-SQL statements that your application executes.

All the tests (listed in Table 2), except Test 6, use the following table schema.

```
CREATE TABLE Tab1
(
    ORG_KEY BIGINT,
    PROD_KEY BIGINT,
    TIME_KEY BIGINT,
    CST_NON FLOAT,
    CST_RPL FLOAT,
    RTL_NON FLOAT,
    RTL_RPL FLOAT,
    UNT_NON FLOAT,
    UNT_RPL FLOAT,
    UPDATE_DATE DATETIME
);
```

At the start of each test, the table was restored from a backup and had 16.125 million rows. The sample data for the table is shown in Figure 1.



ORG_KEY	PROD_KEY	TIME_KEY	CST_NON	CST_RPL	RTL_NON	RTL_RPL	UNT_NON	UNT_RPL	UPDATE_DATE
9920	15937	680	NULL	200	NULL	240	NULL	200	2001-04-02 00:00:00.000
9920	15937	688	NULL	200	NULL	240	NULL	200	2001-04-02 00:00:00.000
9920	15937	697	NULL	200	NULL	240	NULL	200	2001-05-07 00:00:00.000
9920	15937	705	NULL	200	NULL	240	NULL	200	2002-03-18 00:00:00.000
9920	15961	3	NULL	100	NULL	120	NULL	100	2001-04-02 00:00:00.000
9920	15961	11	NULL	100	NULL	120	NULL	100	2002-03-25 00:00:00.000
9920	15961	19	NULL	100	NULL	120	NULL	100	2006-10-19 20:07:31.000
9920	15961	27	NULL	100	NULL	120	NULL	100	2006-10-20 20:07:31.00
9920	15961	36	NULL	100	NULL	120	NULL	100	2006-10-25 17:48:17.000
18120	15961	44	NULL	100	NULL	120	NULL	100	2001-08-13 00:00:00.000
...	...	...	...	...	...	...	...	...	...

Figure 1 Sample data in Tab1

For the tests on the table with a clustered index, the following clustered index was created.

```
    CREATE CLUSTERED INDEX c_idx1 ON Tab1 (ORG_KEY, PROD_KEY, TIME_KEY);
```

For the tests on the table without a clustered index (heap), the following nonclustered index was created.

```
    CREATE INDEX nc_idx1 ON Tab1 (ORG_KEY, PROD_KEY, TIME_KEY);
```

The physical index statistics of the tables with a clustered index and with a nonclustered index respectively (output of **sys.dm_db_index_physical_stats**) are shown in Table 1.

Table 1 Physical Statistics

Level	**Clustered index**	**Clustered index**	**Clustered index**	**Clustered index**	**Heap with nonclustered index**	**Heap with nonclustered index**	**Heap with nonclustered index**	**Heap with nonclustered index**
	Bytes /row	**Row count**	**Pages**	**MB**	**Bytes /row**	**Row count**	**Pages**	**MB**
Heap	N/A	N/A	N/A	N/A	88	16,125,000	181,182	1,415
3	88	16,125,000	181,182	1,415	36	16,125,000	75,708	591
2	34	181,182	810	6	42	75,708	412	3
1	34	810	4	0	42	412	4	0
0	34	4	1	0	42	4	1	0

Test Hardware and Software

We conducted all tests on commodity hardware that was configured with adequate storage. We used a HP DL385 with two dual-core processors and 8-GB memory as the database server, and a desktop class computer with two single-core processors as a client workload driver system. A gigabit network segment was used to connect the server to the client driver system. We conducted the tests on the SQL Server 2005 SP1 Enterprise Edition for x86 platform. Additional hardware and software details are provided in the Appendix.

Test Scenarios

We obtained results for the following test scenarios.

Table 2   Test Scenarios

Test	Test Description
1	**INSERT performance** 1. Measure the time taken to insert 1,000,000 rows of data into the table with the clustered index previously defined, by using individual (row-by-row) insert statements. 2. Measure the time taken to insert 1,000,000 rows of data into the table with the nonclustered index previously defined (no clustered index), by using individual (row-by-row) insert statements.
2	**UPDATE performance** 1. Measure the time taken to update 1,000,000 rows of data in the table with the clustered index previously defined, by using individual (row-by-row) update statements. 2. Measure the time taken to update 1,000,000 rows of data in the table with the nonclustered previously index defined (no clustered index), by using individual (row-by-row) update statements.
3	**DELETE performance** 1. Measure the time taken to delete 1,000,000 rows of data from the table with the clustered index previously defined, by using individual (row-by-row) delete statements. 2. Measure the time taken to delete 1,000,000 rows of data from the table with the nonclustered index previously defined (no clustered index), by using individual (row-by-row) delete statements.
4	**4a) Single-row SELECT performance** 1. Measure the time taken to select 1,000,000 rows of data from the table with the clustered index previously defined, by using individual (row-by-row) select statements. 2. Measure the time taken to select 1,000,000 rows of data from the table with the nonclustered previously index defined (no clustered index), by using individual (row-by-row) select statements. **4b) Range SELECT performance** 1. Measure the time taken to select 228 rows repeatedly in a loop of 1,000,000 iterations from the table with the clustered index previously defined. 2. Measure the time taken to select 228 rows repeatedly in a loop of 1,000,000 iterations from the table with the nonclustered index previously defined (no clustered index).
5	**Table space usage** 1. Measure the change in the size of the table with a clustered index when 2 million and 4 million rows of data are inserted into the table. 2. Measure the change in the size of the table with a nonclustered index (no clustered index) when 2 million and 4 million rows of data are inserted into the table. 3. Measure the change in the size of the table with a clustered index when 2 million and 4 million rows of data are deleted from the table. 4. Measure the change in the size of the table with a nonclustered index (no clustered index) when 2 million and 4 million rows of data are deleted from the table.
6	**Concurrent insert performance** 1. Measure the transaction throughput of insert operations from multiple concurrent processes (20, 30, 40 and 50) on a table with a clustered index. Measure the transaction throughput of insert operations from multiple concurrent processes (20, 30, 40

	2. and 50) on a table with a nonclustered index (no clustered index).

Test Procedure

The following steps were used to execute tests 1 through 4 described in Table 2.

1. The table (Tab1) was created and initialized to an initial state from a backup.
2. The clustered index (c_idx1) was created on the table (Tab1).
3. The particular test described in Table 2 was executed.
4. The table (Tab1) was dropped.
5. The table (Tab1) was again created and initialized to an initial state from the backup.
6. The nonclustered index (nc_idx1) was created on the table (Tab1).
7. The particular test described in Table 2 was executed.
8. The table (Tab1) was dropped.

Tests 5 and 6 are comprised of a somewhat different series of steps. They are discussed in detail later in this document.

Top of page

Test Results and Observations

This section describes each of the six individual tests in detail and presents the results measured. It also summarizes observations and includes general recommendations where appropriate.

Test 1: INSERT Performance

In this test, the rows of a table with a clustered index are organized based on the order specified by the clustered index columns (ORG_KEY, PROD_KEY, TIME_KEY), while the data rows in a table without a clustered index are not organized in any particular order; that is, they are a *heap*.

As can be expected, the type and amount of work that needs to be done by the database engine to insert rows into each of these tables is very different. For the table with the clustered index, a new data row must be inserted into the correct location as specified by the clustered index definition. If the database page where this data row is to be inserted does not have sufficient free space, the respective page must be split (that is, a new page must be inserted into the chain) to make room for the new row. A table without a clustered index requires only that a new entry for the data row be inserted into the B-tree at a particular location[1]; the new data row itself can be inserted into any page in which space is available, or on a new page at the end of the chain.

For the table with the clustered index, only a single write operation is required since the leaf nodes of the clustered index are data pages (as explained in the section Clustered Indexes and Heaps), whereas for the table with the nonclustered index, two write operations are required—one for the entry into the index B-tree and another for the insert of the data itself.

This test attempts to measure the performance difference when performing row-by-row inserts into a table with a clustered index versus one without a clustered index (heap).

Test Setup

The INSERT test was conducted using the table, clustered index, nonclustered index, and table data specified in the Test Methodology section, and the procedure outlined in Test Procedure section. For this test, steps 3 and 7 of the test procedure were as follows:

Steps 3 & 7: The time it took to insert 1 million rows by using 1 million individual row-by-row insert statements was measured.

The insert statements were of the form:

```
INSERT INTO Tab1(ORG_KEY, PROD_KEY, TIME_KEY, UPDATE_DATE) VALUES
(@P1, @P2, @P3, @P4);
```

The sample values for the parameters for one such statement were: @P1=9717, @P2=46273, @P3=206 and @P4='2007-02-01 00:00:00:000'.

Results and Observations

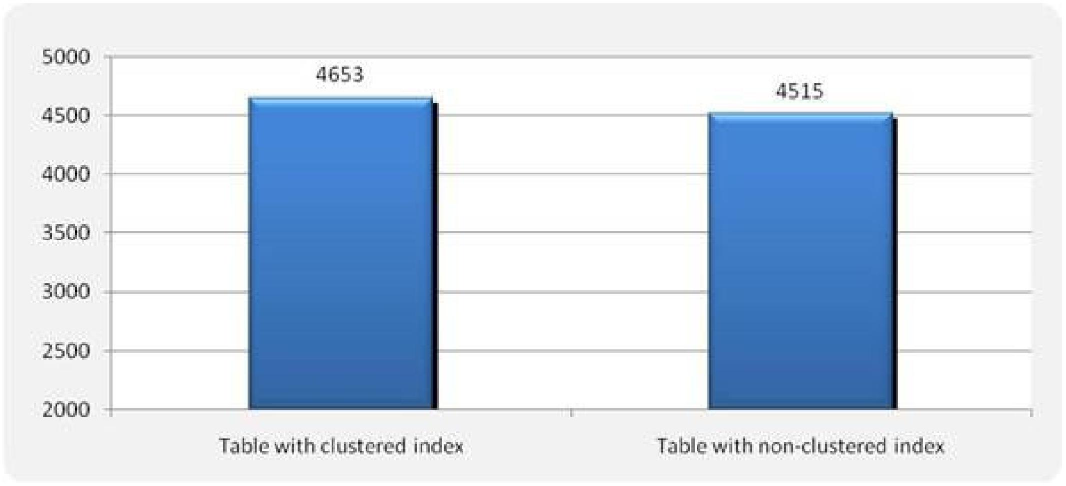


Figure 2 INSERT throughput (rows/sec)

Table 3 INSERT performance results

Metric	**Table with clustered index**	**Table with nonclustered index**	**Difference**
Time to INSERT 1 million rows (sec)	214.9	221.5	3.07%
Throughput (rows/sec)	4653	4515	-3.0%
Processor utilization	11.9	12.4	3.6%
Page splits/sec	52.2	19.4	-62.8%
Pages Allocated/sec	52.2	72.4	38.9%

As previously explained, there are two main differences when performing an insert into a table with a clustered index and a table with just a nonclustered index (heap). These are: 1) maintaining the order of the data rows in the case of the table with the clustered index, and 2) performing two writes for the table with the nonclustered index. From our tests, we observed that inserting data into a table with a clustered index was about 3% faster than inserting the same data into the table with just a nonclustered index. From this observation, we concluded that the overhead to perform two writes in the case of the table with the nonclustered index was higher than the overhead of maintaining the order of the clustered index.

From the System Monitor (perfmon) data, we observed there to be 62.8% fewer page splits/sec for the insert into the table with the nonclustered index as compared to the table with the clustered index. This is expected behavior, given that the index was created with the default fill-factor of zero (all pages are almost full). In addition, the probability of the page not having sufficient room to insert the data is higher than for the nonclustered index where only the index record needs to be inserted. Because of this, Pages Allocated/sec is equal to Page Splits/sec for the table with the clustered index, whereas for the table with the nonclustered index (heap) only 27% of the pages allocated resulted in page splits (72.4 pages allocated/sec, 19.4 page splits/sec). We also observed the processor utilization for the inserts into the table with the clustered index to be marginally (3.6%) less.

Based on these results, we concluded that the performance of inserting data into a table with a single clustered index is slightly better (3%) than inserting the same data into a table with a corresponding nonclustered index.

Test 2: UPDATE Performance

This test measures the performance of updating a nonindexed column (UPDATE_DATE) in the table. For each update operation, only a single row of data in the target table was updated. For the table with the clustered index, an update operation involves searching for the row to update by traversing the clustered index B-tree, and then changing the value of the column(s) to be updated. For a table with a nonclustered index, an update operation involves searching for the row to update by traversing the clustered index B-tree, followed by locating the page in the table that contains the corresponding row, and then changing the value of the column(s) to be updated.

Test Setup

The UPDATE test was conducted by using the table, clustered index, nonclustered index, and table data specified in Test Methodology, and the procedure outlined in the Test Procedure section. For this test, steps 3 and 7 of the test procedure were as follows:

Steps 3 & 7: The amount of time it took to update 1 million rows by using 1 million individual row-by-row update statements was measured.

The update statements were of the form:

```
UPDATE Tab1 SET UPDATE_DATE=@P1 WHERE ORG_KEY=@P2 AND PROD_KEY=@P3 AND
TIME_KEY=@P4;
```

The sample values for the parameters for one such statement were: @P1='2007-02-01 00:00:00:000', @P2=14517, @P3=19417, @P4=225.

Results and Observations

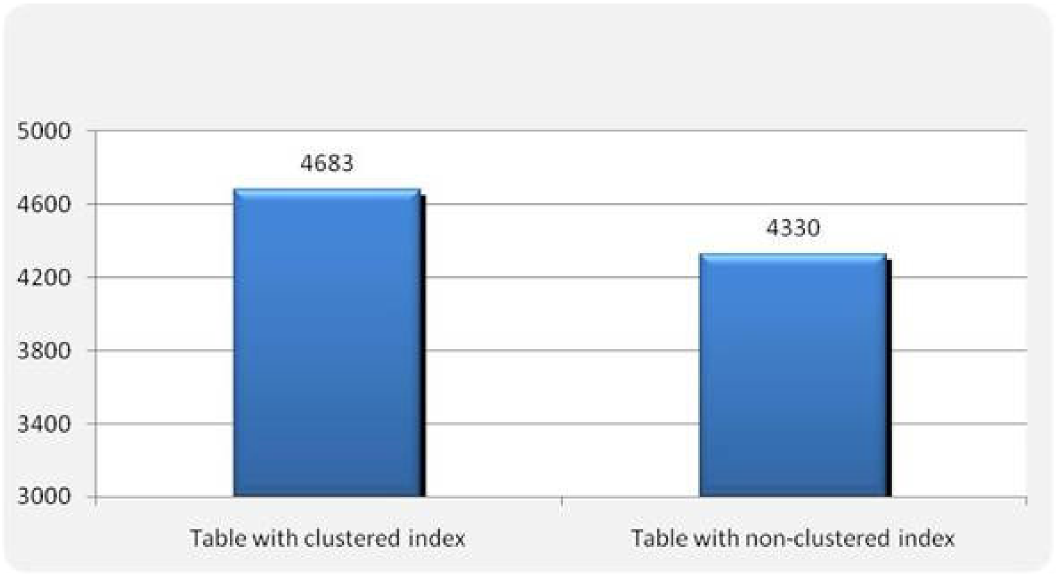


Figure 3 UPDATE throughput (rows/sec)

Table 4 UPDATE performance

Metric	**Table with clustered index**	**Table with nonclustered index**	**Difference**
Time to UPDATE 1 million rows (sec)	213.5	231.0	8.2%
Throughput (rows/sec)	4683	4330	-7.54%

For UPDATE operations on nonindexed columns, a table with a clustered index outperforms a table with nonclustered index by 8.2%. This is because the update operation on a table with a clustered index requires much less work to be performed than on a table with a nonclustered index.

As indicated in the execution plan in Figure 4, in the case of a clustered index, updating a row requires a single index seek operation followed by an update of the data row (leaf node of the index).

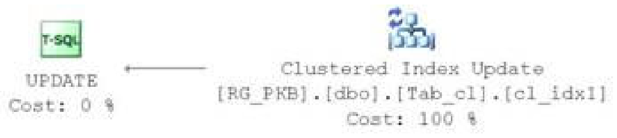


Figure 4 Execution plan for UPDATE operation on table with clustered index

The corresponding UPDATE operation on a table with a nonclustered index results in a seek operation using the nonclustered index, a dereference using the RID (row identifier) to locate the corresponding data row, followed by update of the data row, as indicated by the execution plan in Figure 5.

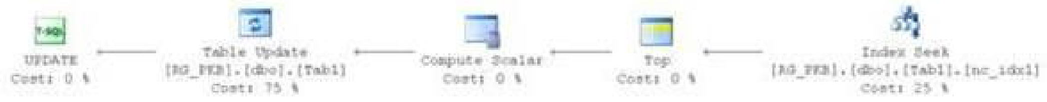


Figure 5: Execution plan for UPDATE operation on table with nonclustered index

The extra work required to update a row when using a nonclustered index as compared to using a clustered index explains the performance difference. Based on this test result, we concluded that for update operations it is advantageous to have a clustered index on a table.

Test 3: DELETE Performance

For a table with a clustered index, a delete operation, based on predicates on the indexed columns, results in a single index seek operation followed by deleting the data row. The corresponding delete operation in a table with a nonclustered index, results in a seek operation using the nonclustered index, a dereference using the RID (row identifier) to locate the corresponding data row, followed by the actual delete. In addition, the corresponding entry from the nonclustered index B-tree must be removed as well.

This test attempts to measure the performance difference in performing row-by-row deletes on a table with a clustered index and one with just a nonclustered index on the same columns (heap). For each delete operation, only a single row of data in the target table is deleted.

Test Setup

The DELETE test was conducted by using the table, clustered index, nonclustered index, and table data specified in Test Methodology, and the procedure outlined in the Test Procedure section. For this test, steps 3 and 7 of the test procedure were as follows:

Steps 3 & 7: The amount of time it took to delete 1 million rows by using 1 million individual row-by-row delete statements was measured.

The delete statements were of the form:

```
DELETE Tab1 WHERE ORG_KEY=@P1 AND PROD_KEY=@P2 AND TIME_KEY=@P3;
```

The sample values for the parameters for one such statement were: @P1=5718, @P2=18193, and @P3=52.

Results and Observations

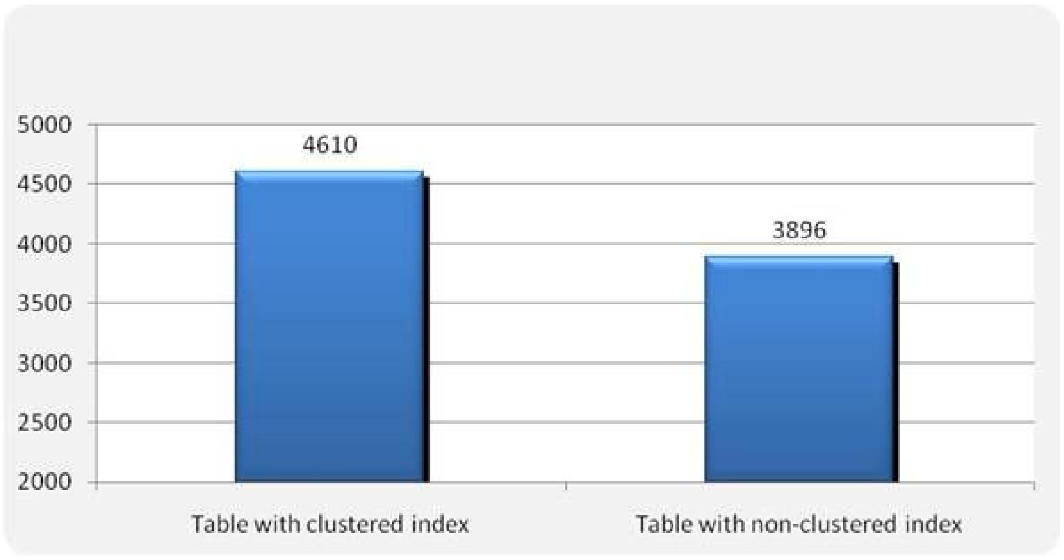


Figure 6   DELETE throughput (rows/sec).

A delete operation involves locating the set of data rows and consequently deleting the row. In the case of the table with the clustered index, the database engine SEEKed into the table and deleted the row by using a single 'Clustered Index Delete' operation. This is shown in the query execution plan in Figure 7.

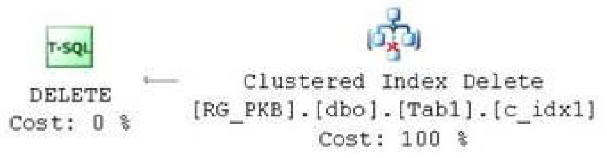


Figure 7   Query execution plan for DELETE operation on table with clustered index

For the table with the nonclustered index, the data row had to first be located by using an index seek operation followed by a Table Delete operation, which involves removing the actual row as well as the entry from the nonclustered index B-tree. This is shown in the query execution plan in Figure 8.

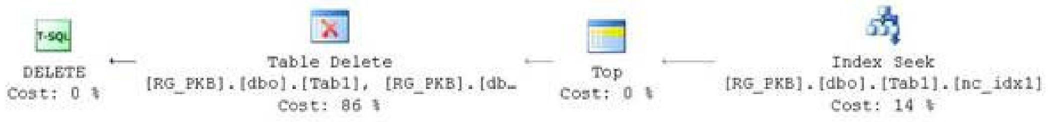


Figure 8   Query execution plan for DELETE operation on table without a clustered index

Given the additional operations that needed to be performed, we observed the performance of deleting values on a table with a clustered index to be 18.25% faster than performing the same operation on a table with a nonclustered index.

Table 5   DELETE performance counts

Metric	Table with clustered index	Table with nonclustered index	Difference
Time to DELETE 1 million rows (sec)	216.9	256.7	18.25%
Throughput (rows/sec)	4610	3896	-15.5%

Based on this test result, we concluded that for delete operations it is advantageous to have a clustered index on a table.

Test 4: SELECT Performance

The SELECT test measured the performance of two different types of SELECT operations:

Test 4a. Single-row select—in this test, each select operation retrieved a single row.

Test 4b. Range select—in this test, each select operation retrieved 228 rows.

Test 4a: Single-Row SELECT Performance

For the table with the clustered index, a select operation, based on the predicates on the indexed columns, results in an index seek operation, which then yields the data values requested. The corresponding select operation on a table without a clustered index results in a seek operation using the nonclustered index, followed by nested loop join with the table to extract the columns not in the index definition (assuming the index is not a covering index for the given query), which then yields the requested row.

This test measures the performance difference between performing single-row selects on a table with a clustered index and one with just a nonclustered index on the same columns (heap). Each select operation returns only a single row of data from the table.

Test Setup

The single-row SELECT test was conducted using the table, clustered index, nonclustered index, and table data specified in Test Methodology, and the procedure outlined in the Test Procedure section. For this test, steps 3 and 7 of the test procedure were as follows:

Steps 3 & 7: The amount of time to select 1 million rows using 1 million individual row-by-row select statements was measured.

The select statements were of the form:

```
SELECT * FROM Tab1 WHERE ORG_KEY=@P1 AND PROD_KEY=@P2 AND TIME_KEY=@P3;
```

The sample values for the parameters for one such statement were: @P1=2717, @P2=15385, and @P3=3.

Results and Observations

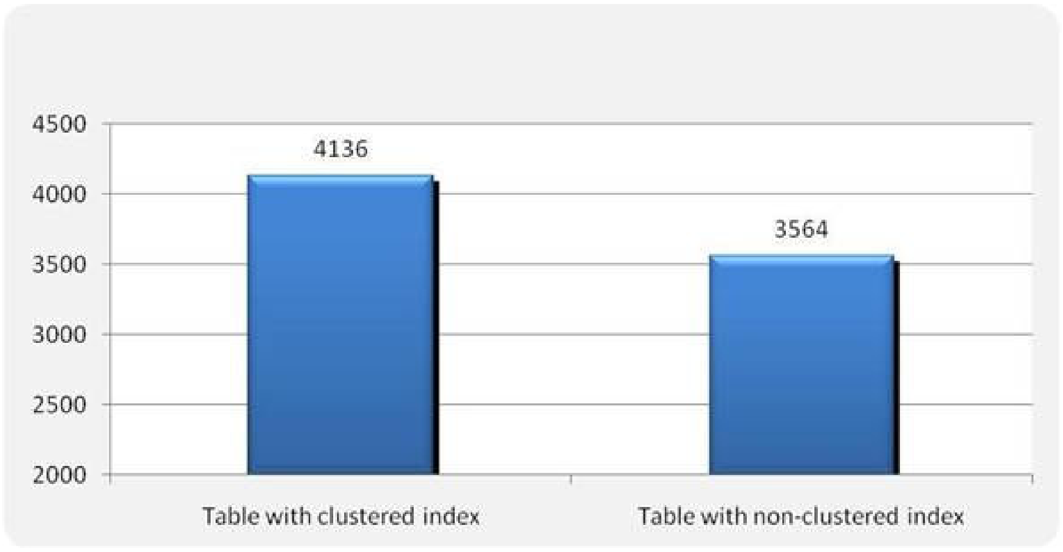


Figure 9   SELECT throughput (rows/sec)

A select operation involves the search for one or more rows from a table. In the table with a clustered index, the database engine performs a Clustered Index Seek operation into the table and yields the requested data row. This is shown in the query execution plan in Figure 10.

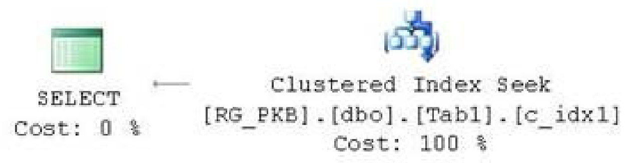


Figure 10 Query execution plan for SELECT operation on table with clustered index

For the table with the nonclustered index, the data row had to first be located by using an Index Seek operation with the nonclustered index, followed by Nested Loops (Inner Join) with a RID Lookup to extract the set of selected columns that were not a part of the nonclustered index. This is shown in the query execution plan in Figure 11.

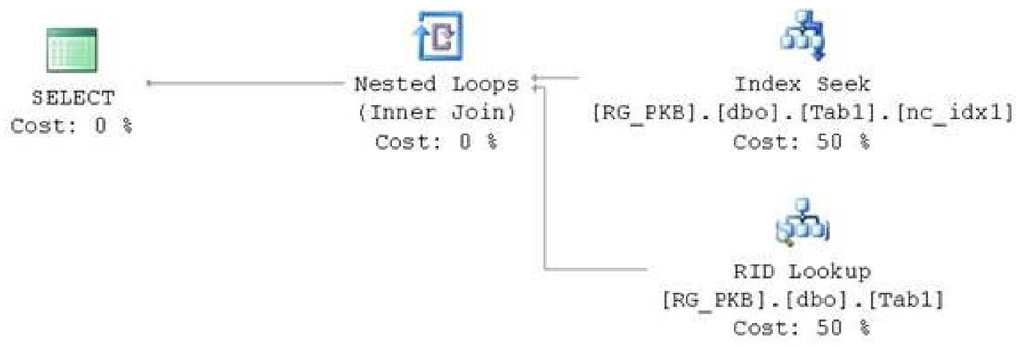


Figure 11 Query execution plan for SELECT operation on table with nonclustered index

Given the additional processing that needed to be done, we observed that the performance of selecting rows from a table with a clustered index was 13.8% faster than performing the same operation on a table with a nonclustered index.

Table 6: Single Row SELECT Performance

Metric	Table with clustered index	Table with non-clustered index	Difference
Time to SELECT 1 million rows	241.8	280.6	16.05%
Throughput (rows/sec)	4136	3564	-13.83%

Based on this result, we concluded that for single-row select operations it is advantageous to have a clustered index on a table.

Test 4b: Range SELECT Performance

The previous test (Test 4a) illustrated the performance behavior of selecting a single row from the table using an index (clustered and nonclustered respectively). Not all queries in an application return just one row. There are many queries that return a range of rows based on the indexed columns.

In the table with the clustered index, the rows are stored in the order of the indexed columns. Therefore, a query with a predicate that specifies a range of values for the leading column of the index returns rows that are stored on the same or consecutive pages. However, in a table with just a nonclustered index, while the nonclustered index itself is an ordered B-Tree, the data that the leaf nodes of the B-tree point to are not stored in any particular order. Therefore, a query with a predicate that specifies a range of values for the leading column of the index accesses rows that may be scattered throughout the table (heap).

This test attempts to measure the performance difference between a query with a predicate that specifies a range of values for the leading column of the index on a table with a clustered index and one with just a nonclustered index on the same columns (heap).

Test Setup

The range SELECT test was conducted by using the table, clustered index, nonclustered index, and table data specified in Test

Methodology, and the methodology outlined in the Test Procedure section. For this test, steps 3 and 7 of the test procedure were as follows:

Steps 3 & 7: The amount of time to select 228 rows was measured repeatedly in a loop of 1 million iterations.

The select statements were of the form:

```
SELECT * FROM Tab1
WHERE (ORG_KEY=@P1 OR ORG_KEY=@P2 OR ORG_KEY=@P3) AND PROD_KEY=@P4
```

The sample values for the parameters for one such statement were: @P1=117, @P2=118, @P3=119, and @P4=45505. The @P1, @P2, and @P3 value used in the test were always consecutive numbers; this resulted in 228 rows being returned for each select.

Results and Observations

Table 7   Range SELECT Performance

Metric	Table with clustered index	Table with nonclustered index	Difference
Time to execute 1 million SELECTs (seconds)	1703.9	2205.1	29.41%
Throughput (selects/sec)	587	454	-22.73%

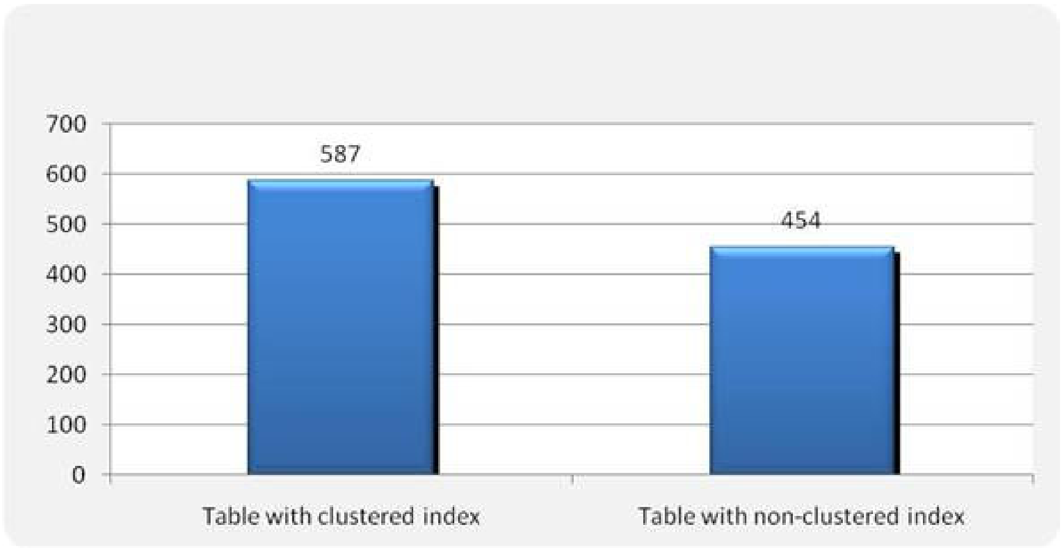


Figure 12   Range SELECT throughput (selects/sec)

The query execution plans for the range select test were identical to the ones for the single-row selects discussed in Test 4a.

We observed that the performance of selecting a range of rows from a table with a clustered index was 29.41% faster than performing the same operation on a table with a nonclustered index. SELECT statements that return a range of rows based on the indexed columns perform better on the table with the clustered index because the rows are already ordered; because of this, fewer pages must be read.

Based on this result, we concluded that for range select operations it is advantageous to have a clustered index on a table.

Test 5: Disk Space Usage

It is intuitive that inserting data into a table should cause the table to grow in size, while deleting data from a table should cause it to shrink. While this is generally accurate, the amount of growth and reduction is different for tables that have a clustered index and those that do not (heaps). This is because tables with a clustered index are susceptible to fragmentation when data is inserted, while tables without clustered indexes (heaps) are susceptible to the free space not being reclaimed when data is deleted, and therefore may not shrink in size in proportion to the data that is deleted.

Test Setup

All the tests were conducted by using the table, clustered and nonclustered indexes, and sample data specified in the Test Methodology section.

Test Procedure

Operations for INSERT test:

1. The table was created and initialized to an initial state from a backup.
2. The clustered index (c_idx1) was created.
3. The space utilized was measured by using the **sp_spaceused** system stored procedure.
4. Two million rows were inserted into the table using 2 million individual Transact-SQL statements. Example:

```
INSERT INTO Tab1(ORG_KEY, PROD_KEY, TIME_KEY, UPDATE_DATE)VALUES(@P1,  @P2, @P3, @P4)
```

   For the 2 million INSERT operations, a combination of ORG_KEY (@P1) and PROD_KEY (@P2) values were randomly selected from the range of possible valid values. This combination was used for 1,000 consecutive row-by-row insert statements. The value of TIME_KEY (@P3) was incremented by 1 starting from the max value of the initial table for each of the 1,000 insert statements. Example: 706, 707, 708, and so on. The value of the UPDATE_DATE (@P4) column was generated by using the date part of the **GetSystemTime()** function. For example, when 9521 and 17665 were selected as the first combination for ORG_KEY and PROD_KEY and 18918 and 47161 as the second combination, the sequence of INSERT statements was as follows:

   @P1 @P2   @P3 @P4

   9521, 17665, 706, '2006-12-15 00:00:00:000'

   9521, 17665, 707, '2006-12-15 00:00:00:000'

   9521, 17665, 708, '2006-12-15 00:00:00:000'

   :

   :

   9521, 17665, 1705, '2006-12-15 00:00:00:000'

   18918, 47161, 1706, '2006-12-15 00:00:00:000'

   18918, 47161, 1707, '2006-12-15 00:00:00:000'

   18918, 47161, 1708, '2006-12-15 00:00:00:000'

   :

5. The space used was measured again and the difference in size computed.
6. An additional 2 million rows were inserted by using a procedure similar to step 4.
7. The space used was measured again and the difference in size computed.
8. Steps 1 to 7 were repeated with the nonclustered index (nc_idx1) created on the table (Tab1).

Operations for DELETE test:

1. The table was initialized to an initial state from a backup.
2. The clustered index (c_idx1) was created.
3. The space utilized was measured by using the **sp_spaceused** system stored procedure.
4. Two million rows were deleted into the table by using the Transact-SQL command:

```
DELETE TOP (2000000) FROM Tab1
```

5. The space used was measured again and the difference in size computed
6. Another 2 million rows were deleted from the table by using the Transact-SQL command:

```
DELETE TOP (2000000) FROM Tab1
```

Note Both delete operations delete pseudo-random rows of data from the table.

7. The space used was measured again and the difference in size computed.
8. Steps 1 to 7 were repeated with the nonclustered index (nc_idx1) created on the table (Tab1).

Results and Observations

Table 8 INSERT operations - Disk space utilization

# of rows inserted	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**
	Table with clustered index	**Table with clustered index**	**Table with clustered index**	**Table with nonclustered index**	**Table with nonclustered index**	**Table with nonclustered index**
	Original size (MB)	Change (MB)	% change	Original size	change	% change
2 million	1456 MB	+ 206 MB	**+ 14.1 %**	2058 MB	+ 281 MB	**+ 13.6 %**
4 million	1456 MB	+ 393 MB	**+ 27.0 %**	2058 MB	+ 567 MB	**+ 27.6 %**

Table 9 DELETE operations - Disk space utilization

# of rows deleted	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**	**Disk space utilization**
	Table with clustered index	**Table with clustered index**	**Table with clustered index**	**Table with nonclustered index**	**Table with nonclustered index**	**Table with nonclustered index**
	Original size	change	% change	Original size	change	% change
2 million	1455 MB	- 181MB	**- 12.4 %**	2011 MB	- 69 MB	**- 3.4 %**
4 million	1455 MB	- 361 MB	**- 24.8 %**	2011 MB	- 139 MB	**- 6.9 %**

When 2 million and 4 million data rows were inserted into the table with a clustered index, the table grew almost linearly: 14.1% and 27%. Correspondingly, when 2 million and 4 million data rows were inserted into the table with a nonclustered index, the table also grew almost linearly: 13.6% and 27.6% respectively. This was expected behavior for disk space: the organization of the table data with a clustered index or a nonclustered index doesn’t make a difference because the data rows were small (80 bytes) compared to the data page size (8 KB) and didn’t result in much fragmentation. For tables with larger data rows, fragmentation could become a big issue and cause the table with the clustered index to grow significantly more than the table with the nonclustered index.

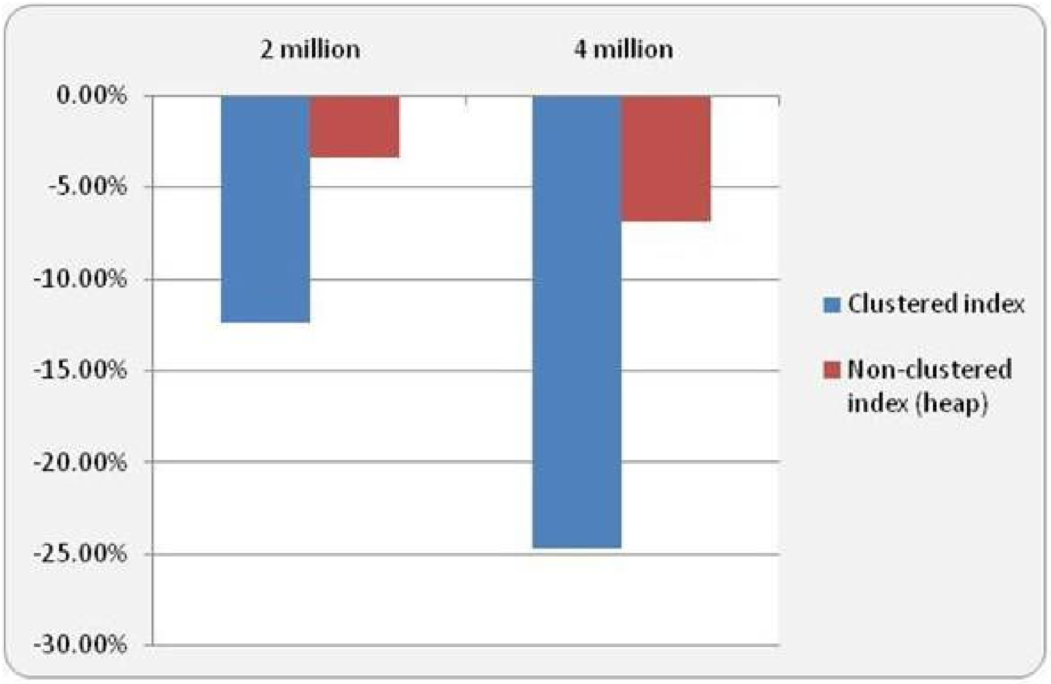


Figure 13   DELETE operations: disk space utilization change

When 2 million and 4 million data rows were deleted from the table with a clustered index, the size of the table shrunk almost linearly: 12.4% and 24.8%. Correspondingly, when 2 million and 4 million data rows were deleted from the table with the nonclustered index, the table also shrunk almost linearly: -3.4% and -6.9% respectively; however, not as much as it should have.

While there is little difference between the table with a clustered index and one with a nonclustered index (heap) for insert operations, there is a significant difference for the disk space utilization following delete operations. In the case of the table with the clustered index, the table shrunk almost the same amount as it had grown when an equal amount of data had been inserted into it. This is because for clustered indexes, empty extents are automatically deallocated from the object (Tab1). However, in the case of the table with the nonclustered index, the table shrunk only a fraction of the amount it had grown when an equal amount of data was inserted into it. This is because for heaps, extents are not deallocated automatically; rather, they are held on to for later reuse. This behavior often results in table size bloating and forces database administrators (DBAs) to perform additional index rebuild tasks to reclaim the free space.

Based on this result, we concluded that from a disk space utilization perspective, it is advantageous to have a clustered index on a table.

Test 6: Concurrent INSERT Performance

Concurrent insert operations performed into a table having a clustered index are often susceptible to encountering hot spots where the contention at the particular insert location in the table adversely affects performance. This test measures the effects of this behavior and compares it to the performance of similar insert operations into a table that has only a nonclustered index on the same columns.

Test Setup

The table schema used in this test is similar to the one used for the previous five tests. However, an additional monotonically increasing identity column was added to deterministically simulate the hot-spot effect.

```
CREATE TABLE Tab1
(
    ID BIGINT IDENTITY(1,1),
    ORG_KEY BIGINT,
    PROD_KEY BIGINT,
    TIME_KEY BIGINT,
    CST_NON FLOAT,
    CST_RPL FLOAT,
    RTL_NON FLOAT,
    RTL_RPL FLOAT,
    UNT_NON FLOAT,
    UNT_RPL FLOAT,
    UPDATE_DATE DATETIME
);
```

At the start of each test the table was restored from a backup and had 16.125 million rows. The sample data used for this test was similar that specified in the Test Methodology section.

For the tests on the table with a clustered index, the following clustered index was created.

```
    CREATE CLUSTERED INDEX c_idx1 ON Tab1 (ID);
```

For the tests on the table without a clustered index (heap), the following nonclustered index was created.

```
    CREATE INDEX nc_idx1 ON Tab1 (ID);
```

The index fill-factor was kept at the default of zero for all the tests, implying that the leaf level was filled nearly to capacity, but some space was left for at least one additional index row.

Test Procedure

1. The table was created and initialized to an initial state from a backup.
2. The clustered index (c_idx1) was created.
3. Two million rows were inserted into the table via 20 parallel processes, each inserting 100,000 rows, and the throughput (rows/sec) of the operations measured during steady state.
4. Steps 1 and 2 were performed again to reinitialize the table.
5. Three million rows were inserted into the table via 30 parallel processes, each inserting 100,000 rows, and the throughput (rows/sec) of the operations measured during steady state.
6. Steps 1 and 2 were performed again to reinitialize the table.
7. Four million rows were inserted into the table via 40 parallel processes, each inserting 100,000 rows, and the throughput (rows/sec) of the operations measured during steady state.
8. Steps 1 and 2 were performed again to reinitialize the table.
9. Five million rows were inserted into the table via 50 parallel processes, each inserting 100,000 rows, and the throughput (rows/sec) of the operations measured during steady state.
10. The table (Tab1) was deleted.
11. The table was created and initialized to an initial state from a backup.
12. The nonclustered index (nc_idx1) was created.
13. Steps 3-10 were repeated for the nonclustered index.

Results and Observations

When concurrently inserting data into the table, we observed that the amount of time per insert increased as the number of concurrent inserts increased for both the table with the clustered index as well as the table with the nonclustered index. The results for this are tabulated in Table 10.

Table 10 Concurrent insert performance

Number of processes	20	30	40	50
Clustered index	1.24 msec	2.22 msec	3.42 msec	5.16 msec
Nonclustered index (heap)	1.23 msec	2.07 msec	2.94 msec	3.96 msec

The amount of time per insert depends on many factors—the hot-spot effect previously explained is just one of them. The graph in Figure 14 plots the observed values, from which we can observe the following:

- As the number of processes that are concurrently inserting data into the table increases, the amount of time it takes per insert also increases.
- The increase is more significant in the case of the table with the clustered index as compared to the one with the nonclustered index.

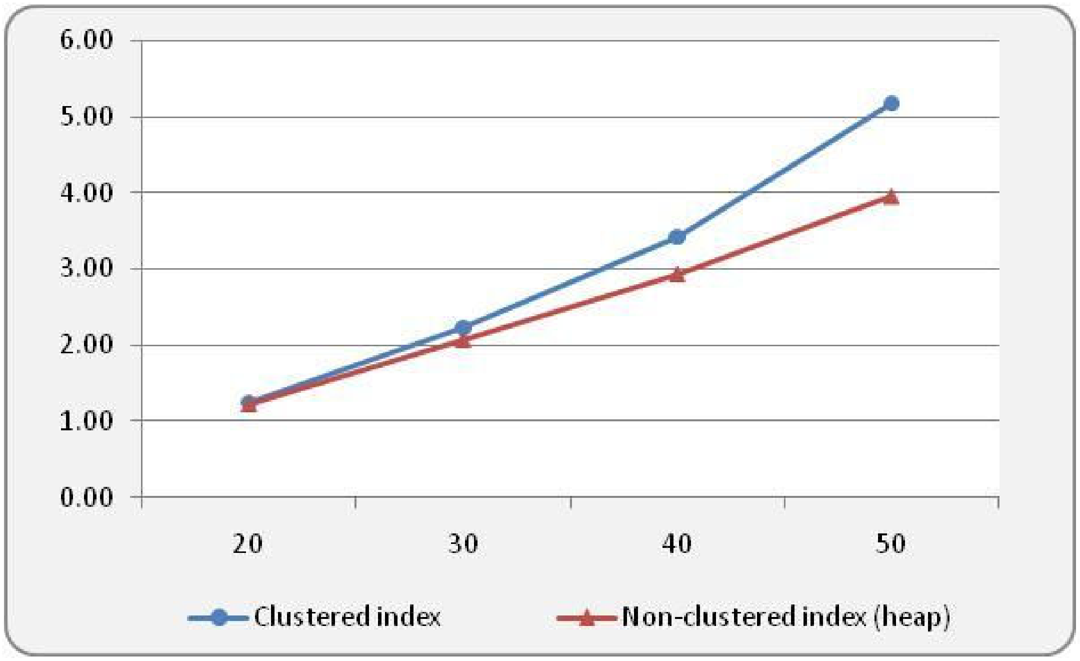


Figure 14 Milliseconds per insert versus number of concurrent processes

This observation is further corroborated by the 'Page latch waits started per second' counts shown in Table 11.

Table 11 Concurrent INSERT performance – Perfmon counters

# proc-esses	**Clustered index**	**Clustered index**	**Clustered index**	**Nonclustered index (heap)**	**Nonclustered index (heap)**	**Nonclustered index (heap)**
	Page Splits/sec	Pages Allocated/sec	Page latch waits started per second	Page Splits/sec	Pages Allocated/sec	Page latch waits started per second
20	1,355	201	94,468	323	251	84,191
30	2,947	173	127,095	718	222	94,915
40	4,216	163	140,622	999	211	95,568
50	5,623	156	150,524	1,280	197	93,602

When 20 processes are concurrently inserting data into the table, the number of 'page latch waits per second' is only 12% more for the table with the clustered index compared to the table with the nonclustered index (94,468 versus 84,191). However, when 50 processes are concurrently inserting data into the table, the number of 'page latch waits per second' is 61%

more for the table with the clustered index compared to the table with the nonclustered index (150,524 versus 93,602). This is a direct result of contention at the point of the insert with the clustered index. This difference in behavior is depicted in the graph in Figure 15.

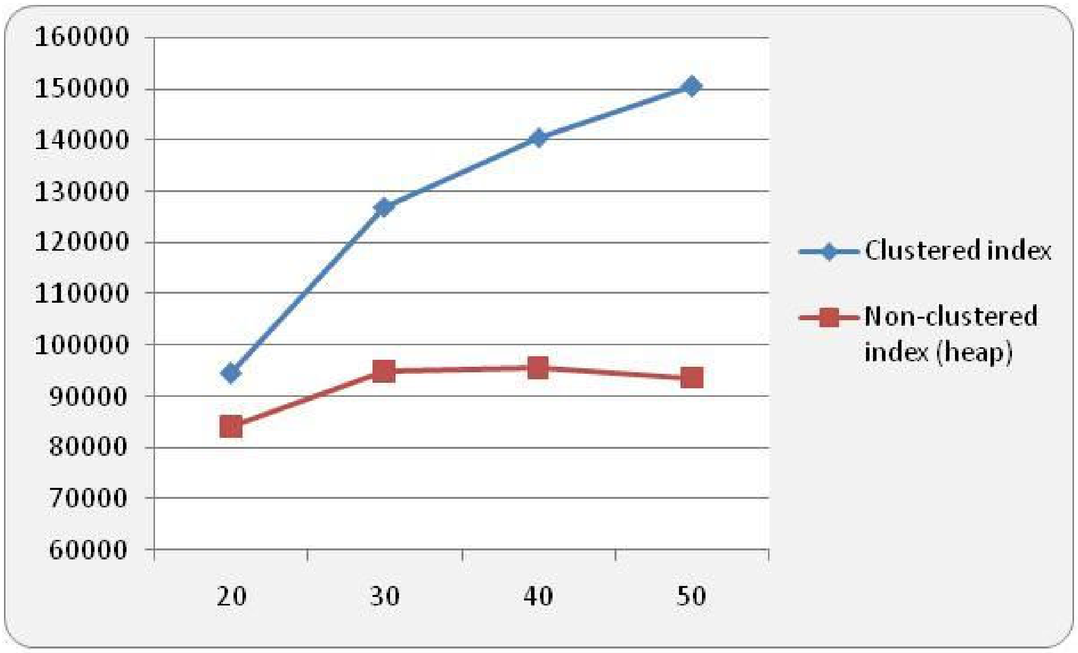


Figure 15 Page latch waits started per second versus number of concurrent processes

In SQL Server 2005, the Page Splits/sec count relates to the total number of page split operations attempted every second, whether they succeed or not. An unsuccessful page split could occur when two or more concurrent sessions attempt to split the same page. Only one session actually splits the page (succeeds); the other sessions no longer need to split the page since the task is already done. The actual number of page splits can be determined via the Pages Allocated/sec count.

For the table with the clustered index, the Page Splits/sec is about four times higher than the corresponding number of Page Splits/sec for the table with the nonclustered index. In addition, the ratio of Page Splits/sec to Pages Allocated/sec for the table with the clustered index is higher and increases more rapidly as the number of processes increases. This behavior is due to the fact that page allocations for a table without a clustered index (heap) never result in a page split (the data is just added to the heap), and there are fewer page splits for the nonclustered index pages because for the given table and index structure, almost twice as many rows can fit in a data page. The ratios of Page Splits/sec to Pages Allocated/sec are depicted in Figure 16.

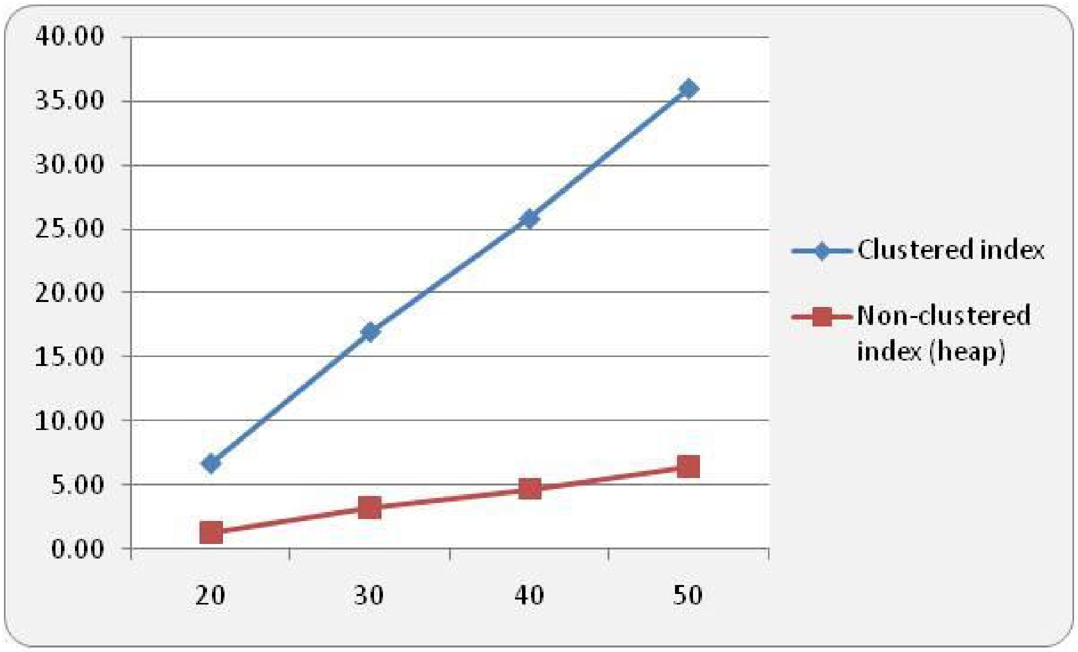


Figure 16 Page splits per second / pages allocated per second versus number of concurrent processes

Given all the analysis about why concurrent inserts on a table with a clustered index on a monotonically increasing column is slower than that of a nonclustered index, it must be noted that the performance difference is not practically significant. For the case with 50 concurrent sessions, the difference in the execution time per insert is only 1.20 milliseconds. For most applications, an overhead of 1.20 milliseconds is not significant. Fifty concurrent sessions inserting data into a table without any think time between consecutive statements represents a very significant workload. And even for such a high workload, the performance penalty for having a clustered index is negligible.

Top of page

Recommendations

After conducting these tests using our sample database table and sample data, we made the following general observations and recommendations. Please note that these are general recommendations that apply to the test used in this study. They may not be well suited for your particular application. Therefore, we encourage you to use them as base recommendations only and validate the applicability of the results to your particular scenario.

- In general, the performance benefits of having a clustered index on a table outweigh the negatives for our sample database table and simulated workload. For the case where the performance was lower (concurrent INSERT test, Test 6), the difference was insignificant. Given this, we recommend creating a clustered index on all SQL Server user tables.
- The disk space required by the table with a clustered index was almost 35% smaller than the table with the nonclustered index on the same set of columns (2.24 GB versus 1.46 GB). The additional space utilization was almost entirely attributed to the size of the nonclustered index. Given this, we recommend creating a clustered index on all SQL Server user tables.
- Deleting rows from a table that has a clustered index freed up more disk space when compared to a table organized as a heap. This residual disk space could be reclaimed by a DBA by using some maintenance tasks; however, this requires additional work and processing.

Top of page

Appendix: Test Environment

Database Server Computer

The following table contains technical specifications for the database server used in the above tests.

Feature	Description
Operating system	Microsoft® Windows® Server 2003, Enterprise Edition (R2)

Version	5.2.3790 Service Pack 1 Build 3790
System manufacturer	HP
System model	ProLiant DL385 G1
System type	x86-based PC
Processors	Two dual-core x86 Family 15 Model 33 Stepping 2 AuthenticAMD ~2405 Mhz
BIOS version/date	HP A05, 3/1/2006
Total physical memory	8 GB
Network adaptors	HP NC7782 Gigabit Server Adapter
SQL Server 2005	SQL Server 2005 Enterprise Edition for x86 platform, with SP1

Client Computer

The following table contains technical specifications for the application server used in the above tests:

Feature	Description
Operating system	Microsoft® Windows® Server 2003, Enterprise Edition
Version	5.2.3790 Service Pack 1 Build 3790
System manufacturer	MICRO-STAR INTERNATIONAL CO., LTD
System model	MS-7125
System type	X86-based PC
Processors	Two x86 Family 15 Model 43 Stepping 1 AuthenticAMD ~2010 Mhz
BIOS version/date	Phoenix Technologies, LTD 6.00 PG, 2005/09/09
Total physical memory	2 GB
Network adaptors	3Com 3C996B Gigabit Server NIC

Storage

Feature	Description
Model	HP StorageWorks Modular Smart Array 30 http://h50146.www5.hp.com/products/storage/diskarray/msa30/index.html
Cache	192 MB
Alignment offset	64 KB
Buses and enclosures	Bus 0, Enclosure 0: 14 146 GB SCSI 10K U320
LUNs/RAID groups	RAID Group 0: Eight disks organized as RAID-0; (D:\); Database data files RAID Group 1: Three disks organized as RAID-0; (E:\); Database Log files

[1] Inserting an entry into the B-tree can result in a page split if the page into which the entry is being inserted is full.

Top of page

Manage Your Profile