SQL ServerCentral.com
SQL Provision NEW Provision databases efficiently and keep data safe TRY FREE
a community service from redgate
Try SQL Clone

Keep up to date - daily newsletter:
your@email.com
Sign up

Write For Us
Find out more

redgate | Hub
Insights
Learning
Forums
Ideas
Discover the new Redgate Hub ›

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Thank this author by sharing: G+ 0
Rate this | Join the discussion | Add to briefcase

Insert, Update, and Delete Indexes: Stairway to SQL Server Indexes Level 13

By David Durant, 2012/02/29

The Series

This article is part of the Stairway Series: Stairway to SQL Server Indexes

Indexes are fundamental to database design, and tell the developer using the database a great deal about the intentions of the designer. Unfortunately indexes are too often added as an afterthought when performance issues appear. Here at last is a simple series of articles that should bring any database professional rapidly 'up to speed' with them

In Levels 10 through 12, we looked at the internal structure of indexes and the impact of change on that structure. In this level we continue that theme by examining the impact of `INSERT`, `DELETE`, `UPDATE` and `MERGE` statements. First, we look at the four commands individually, and then we cover a subject that is applicable to all three: *per row* data modification versus *per index* data modifications.

INSERT

We “jumped the gun” on `INSERT` statements by introducing the subject in Level 11 – *Index Fragmentation*. A summary of what we said there is repeated here. For a fuller coverage, including code samples, refer to Level 11.

When a row is inserted into a table, regardless of whether that table is a heap or a clustered index, an entry must be inserted into each of the table’s indexes; with the possible exception of filtered indexes. When doing so, SQL Server uses the index key value from the row to traverse the index from root page to leaf level page. When it arrives at the leaf level page, it checks the page for available space. If there is sufficient empty space on the page, the new entry is inserted at its proper location.

Eventually SQL Server may attempt to insert an entry into a page that is already full. When this happens, SQL Server will search its allocation structures to find an empty page. Once it has found an empty page, it will do one of three things, each dependent on the sequence in which new index keys are being inserted:

Random Sequence: Normally, SQL Server will move half the entries (those of higher key value) from the full page to

Related Articles
FORUM index entry
2008/10/24
index entry
FORUM Insert / Deletes
2008/08/20
Insert / Deletes
ARTICLE Clustered Columnstore Index Gives "Unable to find index entry" Error
2014/08/28
BLOG
FORUM

Stay up to date:
Daily newsletters with brand new articles, scripts, editorials and a Question of the Day help you keep on top of SQL Server.
your@email.com
Sign up
No thanks

Tags
indexing
stairway series

Write For Us
Find out more

Copyright © 2002-2018 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

the empty page, and then insert the new entry into the appropriate page, thus producing two pages that are half full. If your application continues to insert, but not delete, rows, these two pages will eventually grow from half full to full, where upon they will split into pages that are half full. Over time, every page cycles from half full to full, again and again, resulting in an average page fullness of approximately 75%.

Ascending Sequence: If, however, SQL Server notes that the new entry would be the last entry on the full page, it will assume that rows are being inserted into the table in the same sequence as the index key. Therefore, it places the new row, and only the new row, in the empty page. If SQL Server was correct in its assumption that rows are arriving in index key sequence, subsequent rows will belong in this same page. This new page will fill, causing another new page to be allocated. Once a page becomes full, it stays full, resulting in little or no internal fragmentation.

Descending Sequence: Conversely, if SQL Server notes that the new entry would be the first entry on the full page, it will assume that rows are being inserted into the table in descending sequence. Once again, it places the new row, and only the new row, in the empty page. The internal fragmentation is the same as the ascending sequence scenario, at or near 100%.

Once the new entry is placed into a page, some cleanup must be done. Four next-page / previous-page pointers, spread over three pages, must be updated; and an entry pointing to the new page must be inserted at the next higher level of the index, which, in turn, might result in page split at that index level.

When a row is deleted from a table, the corresponding index entries must be removed from the table's indexes. Again, for each index, SQL Server navigates from the root page to leaf level page and finds the entry. Once SQL Server has found the entry, it will do one of two things: It will either remove the entry immediately, or it will set a bit in the row header, marking the entry as a *ghost* record in the index. This technique is referred to as ghosting the entry. When it becomes appropriate and convenient to do so, ghost records will be removed, as described later in this Level.

Ghost records are ignored during all subsequent querying of the table. Physically, they still exist, but logically they do not. The number of ghost records in an index can be obtained from the *sys.dm_db_index_physical_stats* system function.

It is for performance and concurrency management reasons that SQL Server marks the entry for subsequent removal rather than removing it immediately. Not only the performance of the `DELETE` itself, but also the performance of a subsequent transaction rollback, should one be needed, benefits from ghosting the rows. It is much easier to rollback a `DELETE` operation by un-marking the entry, rather than by recreating that entry from transaction log records.

The decision to ghost the entry, and the length of time that will elapse before the ghost record is actually removed, is influenced by a variety of factors, many of which are beyond the scope of this stairway. This variety of factors makes it difficult to predict

whether a DELETE statement will remove the entries, ghost the entries, or some combination of both. Which, in turn, makes it difficult to predict the immediate impact of a DELETE statement on index fragmentation.

A partial list of the factors that influence the DELETE process appears below:

- If *row level locking* is in effect, deleted index entries will be ghosted.
- If 5000 row locks have been acquired during the execution of a statement, locking normally escalates to *table level locking*.
- The use of *row versioning*, rather than locking, as the concurrency mechanism, may cause the ghosting of deleted records.
- Ghost records will never be removed until their transaction has finished.
- SQL Server's background *ghost-cleanup thread* removes ghost records; however, exactly when it will do so is unpredictable. The DELETE operation itself does not alert the ghost cleanup thread to the presence of the ghost record; rather, it is any subsequent scan of the page that adds the page to a list of pages containing ghost records, a list that is periodically processed by the cleanup thread.
- The *ghost-cleanup thread* awakens approximately once every five seconds. It will clean up to ten pages each time it executes. These numbers are subject to change in a future release.
- You can force the cleanup of ghost records by executing either the *sp_clean_db_free_space* or the *sp_clean_db_file_free_space* stored procedure, which will remove *all* eligible ghost records from the entire database or from a specified database file.

In other words, when you delete rows, they are, logically speaking, gone. If they are not removed immediately, they will be removed as soon as it is safe and practical for SQL Server to do so.

To illustrate ghost records, we load a table that has a nonclustered index with 20000 rows using the sequential insert pattern mentioned above, producing a table whose pages are completely full. Then, within an uncommitted transaction we delete half the table's rows and examine the result. Using the *sys.dm_db_index_physical_stats* system view, we will see that some of the deleted rows have been removed, while some have been ghosted. Lastly, we commit the transaction and reexamine the table, perhaps several times, waiting until the ghost records have been removed.

We do two variations of this demonstration: one in which we delete every other entry in the index, thus deleting half the rows on all the pages; and one in which we delete the first half of the entries, thus deleting all the rows on half the pages.

To check for the presence of ghost records, and for fragmentation, after each DELETE, we use a query similar to one we used in *Level 11 – Index Fragmentation*, only this time the

rightmost column will tell us the number of ghost records in the index. Since we will use this query to access an index that is being modified by a still open transaction, we must run it on the same connection that the transaction is using. To simplify, we create a view named *dbo.viewTestIndexInfo*, shown in Listing 1, and select from this view within our test scripts.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT *
       FROM sys.objects
      WHERE name = 'viewTestIndexInfo' and t
BEGIN
  DROP VIEW dbo.viewTestIndexInfo
END
GO
CREATE VIEW dbo.viewTestIndexInfo
AS
SELECT IX.name as 'Name'
   , PS.index_level as 'Level'
   , PS.page_count as 'Pages'
   , PS.avg_page_space_used_in_percent as 'F
   , PS.ghost_record_count as 'Ghost Records
 FROM sys.dm_db_index_physical_stats( db_id(
                   , default, default
                   , 'DETAILED') PS
 JOIN sys.indexes IX
  ON IX.object_id = PS.object_id AND IX.inde
 WHERE IX.name = 'PK_FragTest_PKCol';
GO
```

Listing 1: The fragmentation report view

Listing 2 shows the code used to create our table and load it, in index sequence, with 20000 rows. It also is similar to one that we used in Level 11.

```
USE AdventureWorks;
GO
IF EXISTS (SELECT *
       FROM sys.objects
      WHERE name = 'viewTestIndexInfo' and t
BEGIN
  DROP VIEW dbo.viewTestIndexInfo
END
GO
CREATE VIEW dbo.viewTestIndexInfo
AS
SELECT IX.name as 'Name'
   , PS.index_level as 'Level'
   , PS.page_count as 'Pages'
   , PS.avg_page_space_used_in_percent as 'F
   , PS.ghost_record_count as 'Ghost Records
 FROM sys.dm_db_index_physical_stats( db_id(
                   , default, default
                   , 'DETAILED') PS
 JOIN sys.indexes IX
  ON IX.object_id = PS.object_id AND IX.inde
```

```
 WHERE IX.name = 'PK_FragTest_PKCol';
GO
```

Listing 2: Loading a table / index with full pages

Once the table has been loaded, executing `SELECT * FROM dbo.viewTestIndexInfo` reveals the size and page fullness of its nonclustered index, as shown in Figure 1.

	Name	Level	Pages	Page Fullness (%)	Ghost Records
1	PK_FragTest_PKCol	0	38	97.5134297010131	0
2	PK_FragTest_PKCol	1	1	6.07857672349889	0

Zoom in | Open in new window

Figure 1: The Index with Full Pages

Running the code shown in Listing 3 begins a transaction, deletes every other row from the table, and then selects from the view.

```
BEGIN TRANSACTION

DELETE DBO.FragTest
 WHERE PKCol % 2 = 0;

SELECT *
 FROM dbo.viewTestIndexInfo;
GO
```

Listing 3: Deleting every other row

The results are shown in Figure 2.

Results | Messages

	Name	Level	Pages	Page Fullness (%)	Ghost Records
1	PK_FragTest_PKCol	0	38	72.9923400049419	4972
2	PK_FragTest_PKCol	1	1	6.07857672349889	0

Zoom in | Open in new window

Figure 2: The Index with Ghost Records

When the DELETE statement began execution, row level locking was in place and deleted index entries where ghosted. After 4972 rows had been deleted, generating 4972 ghost records, so many row locks existed that locking escalated to table locks. All subsequent deletes, 10000 – 4972 = 5028 rows, resulted in the index entry being removed rather than ghosted. This removal of 5028 index entries reduced the average page fullness to ~73%.

Committing the transaction makes the deleted rows eligible for ghost cleanup. At some point thereafter, running the Listing 1 view reveals the page fragmentation shown in Figure 3. The number of pages remains the same, but the pages are now half full, verifying that the ghost records have been removed from the index.

	Name	Level	Pages	Page Fullness (%)	Ghost Records
1	PK_FragTest_PKCol	0	38	48.7443661971831	0
2	PK_FragTest_PKCol	1	1	6.07857672349889	0

Zoom in | Open in new window

Figure 3: The Index after the transaction has committed

As ghost records are removed from a leaf level page, that page may become completely empty. When this happens, the page is released and its corresponding pointer in the lowest non-leaf level, Level 1, is deleted. We examine this behavior with a second demonstration.

For this variation, we reload the table by re-running the code in Listing 2. Selecting from the Listing 1 view before deleting any rows again produces the results shown in Figure 1.

Now we delete the first half of the rows, using the code shown in Listing 4. Once again, the delete is done inside an open transaction.

```
BEGIN TRANSACTION

DELETE DBO.FragTest
 WHERE PKCol <= 20000 / 2;
```

Listing 4: Deleting the first half of the rows

Selecting from the Listing 1 view after the first half of the rows have been deleted reveals the picture shown in Figure 4.

	Name	Level	Pages	Page Fullness (%)	Ghost Records
1	PK_FragTest_PKCol	0	29	94.879565109958	4851
2	PK_FragTest_PKCol	1	1	4.63306152705708	0

Zoom in | Open in new window

Figure 4: The index after the first half of its entries have been deleted

Again, approximately 5000 index entries where ghosted, the rest were removed. Since the removed entries were contiguous, whole pages became empty and were released. The number of pages went down while the average page fullness remained high.

When we commit the transaction, and select from the Listing 1 view, we obtain the results shown in Figure 5.

	Name	Level	Pages	Page Fullness (%)	Ghost Records
1	PK_FragTest_PKCol	0	21	88.2329132690882	1
2	PK_FragTest_PKCol	1	1	3.34815913021992	0

Zoom in | Open in new window

Figure 5: The index after the transaction has committed

Those pages at the start of the index, the ones that had all their entries deleted, have been freed and are no longer part of the index. One page, that had been the middle page of the index and contained the deleted row of highest key value, had some of its rows removed. The other pages had no deleted entries and maintained their fullness.

Why is there one ghost record remaining? Because the address of first page of the leaf level, like the address of the root page, is stored in system metadata. Therefore, once allocated, these two pages are never deallocated. The ghost-cleanup thread took care to leave one ghost record on the first page, ensuring that page would not be deallocated.

Non-leaf level entries are always removed immediately rather than being ghosted. If deleting the non-leaf entry causes its page to become empty, then that page is released and its pointer, at the next higher level, is deleted.

The root page, as was just mentioned, is an exception to this pattern: it is never deleted. Even if the index is completely empty, the root page will remain. For every index there must be a top level page, and that page is the root page.

When a row in a table is updated, index entries may need to be modified. For each index entry, SQL Server will perform this modification as either an in-place update, or as a `DELETE` followed by an `INSERT`. SQL Server will use an in-place `UPDATE`

whenever possible. However, there are some situations in which an in-place UPDATE cannot be performed, and SQL Server will have to execute the update as a DELETE-followed-by-INSERT. Here is a list of the most common reasons:

- The update will modify a key column, causing the entry to be relocated within the index.
- The update will modify a variable width column and cause the entry to no longer fit on the page.
- There is a DML trigger defined on the table.

If the column being modified is part of the index's key, the entry's position within the index will change. The entry must be removed from its old location and inserted into the index its new location based on index key sequence. In most cases, the update will be done as a DELETE followed by an INSERT. If the new location and the old location are on the same page, the update might be done in-place. SQL Server will walk from the root page to the leaf level twice; once to find the current version of the entry and once to determine the correct location for the new version.

If the column being modified is part of a clustered index's key, all nonclustered indexes will need to be updated because their bookmarks are comprised of the clustered index's key.

If the column being modified is a non-key column of the index, the entry's position with the index will remain the same; however the size of the entry might not remain the same. If there is insufficient room on the page for the new version of the entry, the update will be done as a DELETE followed by an INSERT.

The MERGE operation was introduced in SQL Server 2008, and has great power, flexibility, and benefit. Under the covers, MERGE generates INSERT, UPDATE and DELETE statements. The impact on your indexes of executing a MERGE statement is the same as it would be had you written and submitted the INSERT / UPDATE / DELETE statements yourself. For this reason, MERGE requires no special coverage in this Stairway.

When a data modification statement inserts, deletes, or updates a single row, SQL Server must make the modification to the row and then modify each index to reflect the change. But when a data modification statement inserts, deletes, or updates multiple rows, SQL Server has two choices:

- For each row, it can apply the change to the row and modify each index.
- Or:
- For each row, it can apply the change to the row and then, for each index, add the change information to a list of pending changes, rather than actually making the change. Once all the rows have been processed, the pending changes for each index are sorted into index sequence and applied to the index.

This second technique is called *index-at-a-time updating* or *wide updating*, and is an option for INSERT, UPDATE and DELETE operations.