Understanding CRUD Operations on Tables with B-tree Indexes, Page-splits, and Fragmentation

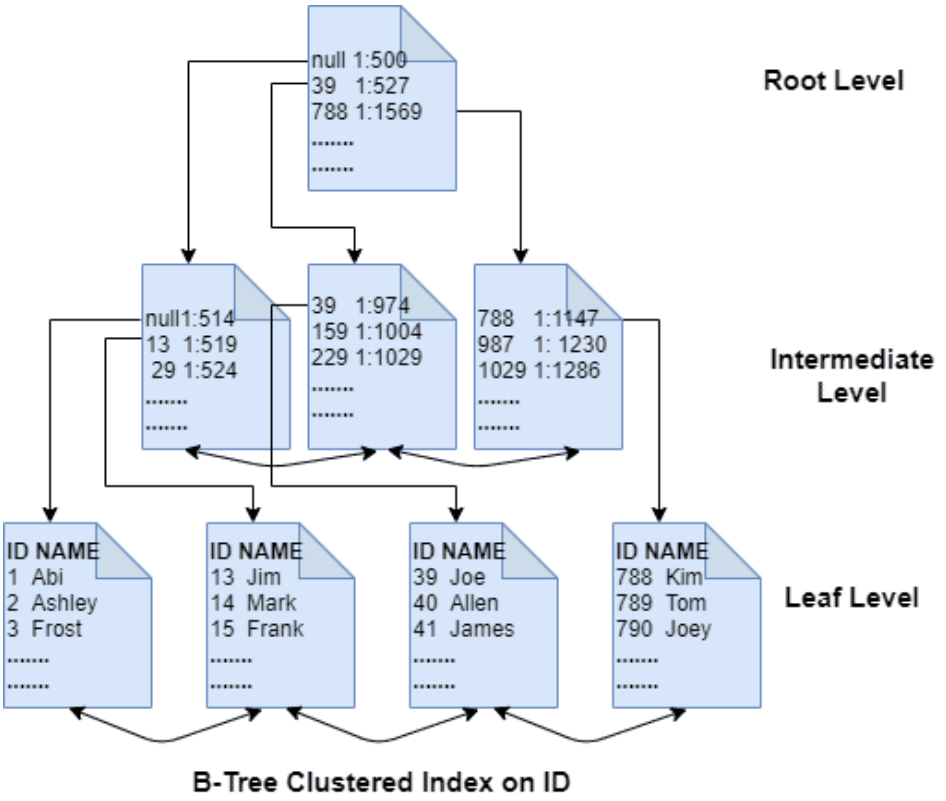
Introduction

Every DML transaction reads the data before it makes any changes. Not only during a SELECT query, but when you run any DML statement, insert, update, or delete, SQL Server first fetches a bunch of pages into the buffer pool locating the desired rows and changes them while synchronously writing to the transaction log file.

While indexes on tables help improve read performance, they add overhead during modifications to the data. Every insert, update or delete that affects the columns in the base table, a heap or a clustered index, also occurs on any non-clustered indexes that contain that column.

Note: This is the third article in a series. However, I wrote these articles in a self-contained way so you can read them a la carte. The other articles are:

Before we dive into the DML commands, let us first understand the basic B-tree structure.



In a b-tree structure, SQL Server sorts the data pages in the order of the index key column(s). The pages are connected as doubly linked lists to allow for scanning the pages sequentially either in the forward or reverse direction. Each index page header contains the page addresses of the previous and the next page. A b-tree index has three types of levels: root, intermediate, and leaf.

A root page sits at the top level with rows pointing to the pages below it. The next level down from the root is an intermediate level consisting of index pages with each row pointing to the page at the next level below it. There can be multiple intermediate levels if a table is large. The last intermediate level points to the leaf level consisting of leaf pages, which could be data pages if the b-tree in question is a clustered index and index pages if it's a non-clustered index. If the leaf pages are index pages, they contain pointers to the base table, which may be a heap or a clustered index.

Insert

When inserting a new row, SQL Server reads the data page on which it must insert it into the buffer pool unless the page doesn't already exist. In this case SQL Server allocates a new extent. Then it inserts the row on the page, synchronously writes the corresponding log record to the transaction log, and asynchronously flushes the dirty page to the data file on disk via either checkpoint or lazy writer.

SQL Server knows which page to store the row on when inserting a row into a table with a clustered index. For example, if there is a page with clustering key column values 1 to 10. When we insert a new row with the key column value 11, SQL Server tries to accommodate it on the same page so long as there is enough space. If there is none, SQL Server allocates a new page and links it to the previous page to maintain the order. Suppose it must insert a row on a page with no room, say, a row with column value 5 in our example. In that case, SQL Server splits the page, i.e., it allocates a new page and moves close to 50% of the rows from the previous page to the newly allocated page and performs the insert on either the old or new page depending on the index key value.

While this page-split process maintains the rows' logical ordering based on the clustering key column, it is costly. SQL Server has to unlink the old page, create a new link to the newly allocated page, and move the rows there. Also, an entry for the new page is added to the parent page above it. The operation is fully logged, resulting in increased transaction logging, which could have dire consequences if the database is part of a feature that relies on transaction log such as Availability Groups.

Page splitting also causes logical (or external) fragmentation. If the new page is not physically contiguous to the page being split in a page-split, which is almost always the case, it affects performance. Especially during read-ahead reads, a performance optimization mechanism where SQL Server, in anticipation, reads a bunch of extra pages into memory before they are even needed by issuing large I/O requests. By not having the neighboring pages physically contiguous because of page-splits, SQL Server has to move back and forth when reading them. The rows' logical order doesn't follow the physical order, thus causing logical fragmentation that is detrimental to performance. Page-splits can happen in the index root page, intermediate index pages, leaf-level data pages of a clustered index, or leaf-level index pages of a non-clustered index.

A page can undergo multiple splits if the new row cannot be accommodated after a single split. Also, a page split executes as an internal transaction, meaning if you ran an insert transaction and it caused a page to split and cancel it, while SQL Server rolls back the insert, it doesn't undo the split. Also, SQL Server exclusively locks the page being split when it splits between two pages, meaning the rows on the page are not accessible to sessions needing them.

Fragmentation in clustered indexes often occurs when the clustering key is not increasing, meaning if the key column values are randomly generated, there is no guarantee that new values will be before or after the existing values. For example, suppose a clustered index uses a uniqueidentifier data type with NEWID() function to generate values randomly. Although the values will be unique, they can fall before or after the existing values, leading to page-splits and cause fragmentation. The same explanation holds for non-clustered indexes.

Index Root Page-split

A splitting of the index root page is not common as indexes usually have a few levels, and for the root page to split, the table must be huge with bulk inserts. If the index root page must split to accommodate a new index row, SQL Server allocates two new index pages. It moves the rows from the root page onto the two new pages, followed by the new row's insert into one of the two new pages based on its logical order. SQL Server does not remove the root page. The original rows on the root page now exist on the two new pages, and the root page now has two new index rows, each pointing to one of the two new index pages. This splitting of the root page introduces a new intermediate level in the index, i.e., the two new index pages.

Intermediate Index Page-split

Suppose the intermediate-level index page must split to accommodate a new index row. In that case, SQL Server allocates a new index page, locates the middle of the index rows on the page-to-split, and moves the bottom 50%

of the rows into the new page followed by inserting the new row into either the page with top 50% or the new page with bottom 50% rows depending on its logical order and linking the new page appropriately. The corresponding parent page above has a new index row inserted with the physical address and the minimum key column value on the newly allocated page.

Leaf Page Split

By far, the most common page split is the one that occurs at the leaf level of either clustered or non-clustered index. Like in an intermediate page-split, SQL Server creates a blank new page and moves 50% of the rows from the old to the new page and inserts the new row into one of the two pages. While any non-clustered indexes on the table are not affected because the clustered index key values don't change in a page split, it affects key lookups' performance because SQL has to perform scattered reads if the clustered index leaf pages are physically noncontiguous.

Insert Demo

Let us use the same database we created in the heap demo article to see how inserts can cause fragmentation. The following code creates a new table called Insert_frag with a uniqueidentifier column using NEWID() function as the clustering key and populates it with a bunch of rows. We then insert some more records to cause random inserts, resulting in page-splits and fragmentation.

```
--Fragmentation because of insert
USE DML_DB
GO
IF NOT EXISTS (
SELECT *
FROM sysobjects
WHERE name = 'Insert_frag'
AND xtype = 'U'
)
BEGIN
CREATE TABLE dbo.Insert_frag (
ID UNIQUEIDENTIFIER CONSTRAINT DF_GUID DEFAULT NEWID()
,Name VARCHAR(7800)
);
END
--insert a bunch of rows
INSERT INTO Insert_frag (Name)
SELECT TEXT
FROM sys.messages
CREATE CLUSTERED INDEX CIX_Insert_frag ON dbo.Insert_frag (ID);
--check fragmentation
USE DML_DB
GO
SELECT index_type_desc
,page_count
,CAST(avg_fragmentation_in_percent AS DECIMAL(6, 2)) AS avg_frag_in_percent
,fragment_count AS frag_count
,avg_fragment_size_in_pages AS avg_frag_size_in_pages
,CAST(avg_page_space_used_in_percent AS DECIMAL(6, 2)) AS avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(DB_ID('DML_DB'), OBJECT_ID('dbo.Insert_frag'), NULL, NULL, 'DETAILED')
--Random inserts resulting in fragmentation
INSERT INTO Insert_frag (Name)
SELECT name
FROM sys.system_columns
--check fragmentation
SELECT index_type_desc
,page_count
,CAST(avg_fragmentation_in_percent AS DECIMAL(6, 2)) AS avg_frag_in_percent
,fragment_count AS frag_count
,avg_fragment_size_in_pages AS avg_frag_size_in_pages
,CAST(avg_page_space_used_in_percent AS DECIMAL(6, 2)) AS avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(DB_ID('DML_DB'), OBJECT_ID('dbo.Insert_frag'), NULL, NULL, 'DETAILED')
```

The top outcome is from before the random inserts. Notice that fragmentation was almost zero with high page density shown in column, avg_page_space_used_in_percent column. After random insert, the bottom result shows a lot of fragmentation and low page density (page fullness) because of page-splits. Also, notice that the page count increased after the second insert.

	index_type_desc	page_count	avg_frag_in_percent	frag_count	avg_frag_size_in_pages	avg_page_space_used_in_percent	forwarded_record_count
1	CLUSTERED INDEX	5594	0.05	168	33.2976190476191	98.76	NULL
2	CLUSTERED INDEX	22	18.18	8	2.75	87.94	NULL
3	CLUSTERED INDEX	1	0.00	1	1	7.59	NULL

	index_type_desc	page_count	avg_frag_in_percent	frag_count	avg_frag_size_in_pages	avg_page_space_used_in_percent	forwarded_record_count
1	CLUSTERED INDEX	8137	62.05	5171	1.57358344614195	68.57	NULL
2	CLUSTERED INDEX	41	100.00	22	1.86363636363636	68.63	NULL
3	CLUSTERED INDEX	1	0.00	1	1	14.16	NULL

Delete Operations

SQL Server performs deletes efficiently. As in any DML operation, SQL Server reads into memory the data pages from which to delete the data rows. SQL Server doesn't remove the rows from the slots of the page physically. Instead, it merely flags the deleted rows as ghosted records by changing the row headers' status bits on those affected pages. This makes the rollback easier, as it just has to revert the status bits on those pages, making no physical changes.

SQL Server does the actual removal of the ghost records in the background with a background thread, called ghost-cleanup. This thread removes the ghost records as long as any active uncommitted transaction does not require them. The thread also deallocates the entire page if we remove all the rows from it; however, heap tables have to be rebuilt to deallocate empty pages. When we delete a row from a table, SQL Server deletes the corresponding row from any non-clustered indexes since they contain a pointer to the deleted row.

Deletes result in gaps in the otherwise physically contiguous pages. As an example, say page 1 has rows with values 1 to 10, page 2 has values 11 to 20, and page 3 has values 21 to 30. If we delete all the rows from page 2, the page becomes blank and causes a gap between pages 1 and 3. Page 2 can either be used by future inserts, updates or deallocated by the background ghost-cleanup thread. So, while deletes do not cause page splits directly, they create gaps other indexes may use in the future, potentially resulting in fragmentation, but it isn't as severe as the one introduced by inserts and updates.

Delete Demo

The following code creates a new table called Delete_frag with an identity column as the clustered index and populates it with a bunch of rows. We then delete some random records to cause gaps resulting in small fragmentation.

```
--Fragmentation because of delete
USE DML_DB
GO
IF NOT EXISTS (
SELECT *
FROM sysobjects
WHERE name = 'Delete_frag'
AND xtype = 'U'
)
BEGIN
CREATE TABLE dbo.Delete_frag (
ID INT IDENTITY(1, 1)
,Name VARCHAR(3000)
);
END
--insert a bunch of rows
INSERT INTO dbo.Delete_frag (Name)
SELECT name + replicate('a', 2500)
FROM sys.all_views
CREATE CLUSTERED INDEX CIX_Delete_frag ON dbo.Delete_frag (ID);
```

```
SELECT index_type_desc
,page_count
,CAST(avg_fragmentation_in_percent AS DECIMAL(6, 2)) AS avg_frag_in_percent
,fragment_count AS frag_count
,avg_fragment_size_in_pages AS avg_frag_size_in_pages
,CAST(avg_page_space_used_in_percent AS DECIMAL(6, 2)) AS avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(DB_ID('DML_DB'), OBJECT_ID('dbo.Delete_frag'), NULL, NULL, 'DETAILED')
USE DML_DB
GO
--delete some random set of rows to create gaps
--deletes result in page gaps, so fragmentation is usually minimal
DELETE
FROM dbo.Delete_frag
WHERE ID BETWEEN 25
AND 75
DELETE
FROM dbo.Delete_frag
WHERE ID BETWEEN 85
AND 135
DELETE
FROM dbo.Delete_frag
WHERE ID BETWEEN 135
AND 235
waitfor delay '00:00:05'
SELECT index_type_desc
,page_count
,CAST(avg_fragmentation_in_percent AS DECIMAL(6, 2)) AS avg_frag_in_percent
,fragment_count AS frag_count
,avg_fragment_size_in_pages AS avg_frag_size_in_pages
,CAST(avg_page_space_used_in_percent AS DECIMAL(6, 2)) AS avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(DB_ID('DML_DB'), OBJECT_ID('dbo.Delete_frag'), NULL, NULL, 'DETAILED')
```

	index_type_desc	page_count	avg_frag_in_percent	frag_count	avg_frag_size_in_pages	avg_page_space_used_in_percent	forwarded_record_count
1	CLUSTERED INDEX	179	0.00	2	89.5	94.13	NULL
2	CLUSTERED INDEX	1	0.00	1	1	28.72	NULL

	index_type_desc	page_count	avg_frag_in_percent	frag_count	avg_frag_size_in_pages	avg_page_space_used_in_percent	forwarded_record_count
1	CLUSTERED INDEX	112	1.79	3	37.3333333333333	93.79	NULL
2	CLUSTERED INDEX	1	0.00	1	1	17.96	NULL

Update Operations

Unless the page containing the row to be updated is already in the buffer pool, SQL Server physically reads the page from disk, causing physical I/O. It then updates the row on the page and generates a new transaction log record, which holds enough information to redo the update if needed during recovery. It then synchronously writes the log record to the transaction log file on disk. The changed or 'dirty' page is flushed to the data file on disk at some point.

As long as the clustering key columns do not change, no non-clustered indexes are affected. However, if the clustering key changes because of an update command, the corresponding row in the non-clustered index is updated to reflect the new key value. If an update causes the row to be moved to a different physical location and the clustering key value remains intact, there is no change to the non-clustered index.

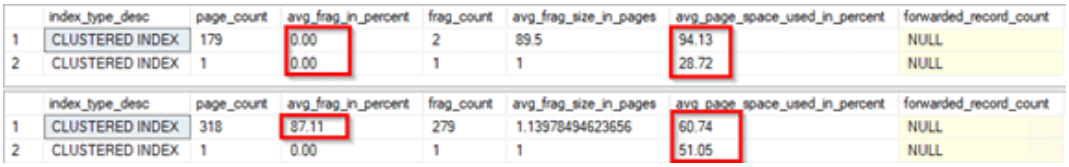
An update results in page splits and index fragmentation when we update a column to a larger value that can neither replace the old value within the same location nor move to another location on the same page, leaving page-split as the only option. Fragmentation can also result when the index key value changes. The new value location is on a different page with not enough space to accommodate the new row resulting in that page splitting.

Update demo

The following code creates a new table called Update_frag with an identity column as the clustered index and populates it with a bunch of rows. We then update some records to a larger value, causing fragmentation.

```
--Fragmentation because of update
USE DML_DB
GO
IF NOT EXISTS (
SELECT *
FROM sysobjects
WHERE name = 'Update_frag'
AND xtype = 'U'
)
BEGIN
CREATE TABLE dbo.Update_frag (
ID INT IDENTITY(1, 1)
,Name VARCHAR(3000)
);
END
--insert a bunch of rows
INSERT INTO dbo.Update_frag (Name)
SELECT name + replicate('a', 2500)
FROM sys.all_views
CREATE CLUSTERED INDEX CIX_Update_frag ON dbo.Update_frag (ID);
USE DML_DB
GO
SELECT index_type_desc
,page_count
,CAST(avg_fragmentation_in_percent AS DECIMAL(6, 2)) AS avg_frag_in_percent
,fragment_count AS frag_count
,avg_fragment_size_in_pages AS avg_frag_size_in_pages
,CAST(avg_page_space_used_in_percent AS DECIMAL(6, 2)) AS avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(DB_ID('DML_DB'), OBJECT_ID('dbo.Update_frag'), NULL, NULL, 'DETAILED')
USE DML_DB
GO
-- update a bunch of rows to a larger value to cause page-splits
UPDATE dbo.Update_frag
SET name = REPLICATE('b', 3000)
WHERE ID > 120
--notice there are no forwarding pointers because of clustered index
SELECT index_type_desc
,page_count
,CAST(avg_fragmentation_in_percent AS DECIMAL(6, 2)) AS avg_frag_in_percent
,fragment_count AS frag_count
,avg_fragment_size_in_pages AS avg_frag_size_in_pages
,CAST(avg_page_space_used_in_percent AS DECIMAL(6, 2)) AS avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(DB_ID('DML_DB'), OBJECT_ID('dbo.Update_frag'), NULL, NULL, 'DETAILED')
```

The top outcome is from before the update. Notice fragmentation was zero with high page density. The bottom result is after the update, which shows fragmentation and low page density because of page-splits.



	index_type_desc	page_count	avg_frag_in_percent	frag_count	avg_frag_size_in_pages	avg_page_space_used_in_percent	forwarded_record_count
1	CLUSTERED INDEX	179	0.00	2	89.5	94.13	NULL
2	CLUSTERED INDEX	1	0.00	1	1	28.72	NULL

	index_type_desc	page_count	avg_frag_in_percent	frag_count	avg_frag_size_in_pages	avg_page_space_used_in_percent	forwarded_record_count
1	CLUSTERED INDEX	318	87.11	279	1.13978494623656	60.74	NULL
2	CLUSTERED INDEX	1	0.00	1	1	51.05	NULL

Controlling Page-splits and Fragmentation

Page-splits result in updates to multiple pages, which are all fully logged in the transaction log. The next obvious question is, how do we avoid page splits? One way is to use an appropriate clustering key that does not insert data into random pages but does so at the end of the table, thus avoiding inserts into the earlier pages, which would otherwise cause page-splits.

The recommended characteristics for a clustering key are unique, static, narrow, and ever-increasing. GUIDs cause pages to split because of their random nature, so they may not be the best choice for a clustering key. Instead, you may use an ever-increasing IDENTITY or a DATETIME column. However, with an ever-increasing key, the last page might become a hot spot resulting in the *last page insert contention*, meaning there could be multiple concurrent inserts on the last page in memory, causing high PAGELATCH_xx waits for the threads. SQL Server 2019 introduced an index option called OPTIMIZE_FOR_SEQUENTIAL_KEY to improve throughput for high concurrent inserts into the index, thus lowering the last page insert contention.

Second, use the appropriate value for Fill Factor when creating or rebuilding an index. The Fill Factor determines how much free space is reserved for potential future inserts on an index's leaf pages. The correct value for Fill Factor is driven by the type of workload that runs on the instance. Suppose there is a lot of free space on the pages. In that case, it leads to low page-density (aka internal fragmentation) and wastage of memory as SQL Server reads the entire page into memory regardless of whether it is wholly or partially filled. Likewise, if you choose to fill the pages to their capacity, there is a high likelihood of page-splits and fragmentation. So there needs to be a careful evaluation of the workload.

A Fill Factor of 0 is the same as 100, meaning to fill the pages to their capacity. 0 is the default fill factor value when creating an index and may be changed to suit the workload. One way is to start with 100, monitor the fragmentation levels, and gradually adjust the value down until you find an optimal value.

Avoiding variable-length data types also minimizes fragmentation. Data in data types such as VARCHAR and NVARCHAR can change over time and potentially cause page splits. Row versioning can also introduce fragmentation. When row versioning is enabled, SQL Server stores the old version of the records in tempdb's version store and adds a 14-byte version tag to the rows in the database to link to the rows in the version store. This 14-byte increase in each row might force rows to move around, causing page-splits.

Shrinking a database also introduces fragmentation, which we can fix by rebuilding the index. Shrinking moves the pages from the end of the data file to the front and releases the free space at the end. SQL Server moves the pages to the front without following the physical contiguity, introducing fragmentation. Therefore, we should avoid shrinking unless necessary.

We must follow proper index reorganize and rebuild strategies. Microsoft recommends rebuilding an index if the fragmentation exceeds 30 percent and reorganize if it's between 5 and 30 percent. While index rebuild runs as a transaction that, when stopped rolls back completely, index reorganize can be interrupted without losing the work completed up to the point where it was stopped. SQL Server 2017 introduced a resumable online index rebuild feature that allows pausing an index rebuild either manually or when there is a failure and resuming it afterward.

Conclusion

Indexes must be maintained regularly to remove internal and external fragmentation. We must carefully evaluate the Fill Factor, as there is no universally accepted value for it. Choose a unique, static, narrow, and ever-increasing column for a clustered index to minimize page-splits and fragmentation. Non-clustered indexes should be narrow to increase page density leading to faster index maintenance, efficient buffer pool utilization, fewer I/Os, faster backups, and faster DBCC CHECKDB. Inserts, updates, and deletes can all lead to fragmentation. However, deletes technically cause gaps between pages which may eventually lead to fragmentation, but it's not as significant as the fragmentation caused by inserts and updates.