Understanding CRUD Operations Against Heaps and Forwarding Pointers

Introduction

Data Manipulation Language (DML) transactions are the core of any workload that runs on SQL Server. A solid understanding of what happens under the covers when you run a DML statement is crucial to writing better-performing code. In this article, we dive into the internals of the three main DML statements against heaps: insert, update, and delete. We start by understanding what happens internally when each of the three DML statements runs against a heap. Then we understand what forwarding pointers are and what causes them. After finishing the article, you should understand why SQL Server behaves the way it does under specific DML workload patterns against heaps, what causes forwarding pointers in heaps, and how to control them.

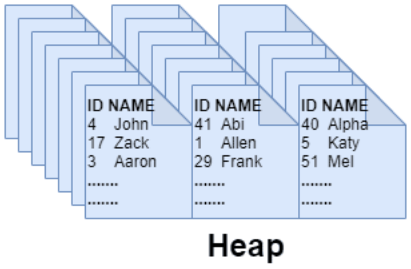
One thing to understand is that SQL Server never performs DML directly against the pages on disk. It brings them into a portion of memory called buffer pool, makes the updates in there, synchronously writes to the transaction log, and asynchronously saves the 'dirty' pages to the data file(s) on disk.

Note: This is the second article in a series, *A Deep Dive into DML internals in SQL Server*. However, I wrote these articles in a self-contained way so you can read them a la carte. The articles are:

DML on Heaps

A heap is nothing but a table without a clustered index. It doesn't follow a logical ordering of rows since there is no key column to order the pages. SQL Server uses the Index Allocation Map (IAM) pages to track and scan the data pages allocated by the table.

As shown in the diagram below, a heap is a cluttered stack of 8 KB data pages. SQL Server looks through all the data pages to find records matching the query statement's filter criteria, resulting in poor performance due to full scans.



Inserting rows into a heap

To insert a row into a table, SQL Server needs to locate a place to store the row. If the table is a heap, SQL Server inserts it wherever enough space is available, since unlike a b-tree index, there is no logical order in heaps. SQL Server uses the Index Allocation Map or IAM pages to find the extents that belong to the table in question, and from there, it uses PFS pages to track the free space on the pages and stuffs the row on a page with enough room to accommodate it. If there is no space on existing pages, SQL Server then tries to find the unallocated pages within the same extent that belongs to the table. If there is none, it allocates a new extent.

During inserts, SQL Server has to do the hard work of finding space for the row among the data pages, unlike a clustered index where a row has a definitive location based on the clustering key, which determines where it goes.

Deleting rows from a heap

When we delete a row from a heap table, SQL Server doesn't release the space automatically, nor does it move data on the page or reorganize it. SQL Server doesn't physically remove the rows from the page. It merely sets the row offset for the deleted row to 0, showing that space can insert a new row. Not only can we not reorganize the space on a page after a delete, but the empty pages also are not deallocated. If we delete all the rows from a page, SQL Server doesn't deallocate the empty page automatically. The heap continues to use the page until we rebuild the heap table.

A common myth among SQL Server users is that the space on the page held by the deleted rows never gets reused. While it's true that the page is not deallocated until we rebuild the heap table, it is available for any future inserts. If we must mark the blank page deallocated, we must either rebuild the heap table using the *alter table….rebuild* statement or create a clustered index to establish a logical order for all the rows based on the clustering key. In both cases, if there are any non-clustered indexes on the table, they are rebuilt as the heap's row-ids change post rebuild or when we create a clustered index on it wherein the row-id is the clustering key.

The non-clustered indexes are required to use the new row-ids to link the rows from non-clustered indexes to their corresponding rows in the base table. The row-ids may either be the physical address (RID) of the rows in the heap or clustering key if there is a clustered index.

Updating rows in a heap table

Updates to rows on a page may either occur in place or not in place. When an update changes the row to a new value without moving the row either on the same page or to a different page, it is in place. The changed row stays in the same location on the same page, and this is true regardless of whether the table is a heap or one with a clustered index. If the clustering key changes such that the row doesn't move from its place and stays on the same page in the same location, we consider it an in-place update with a clustered index.

Suppose an update causes the row to be moved to another page or a different location on the same page. In that case, we consider it an update not in place and occurs as a delete followed by an insert. If the not-in-place update occurs on a heap table, it results in a *forwarding pointer* problem which is detrimental to performance.

Forwarding pointers

In heap tables, a not-in-place update results in moving the row to another location and leaving a 16-byte row in the original row location with an address pointing to the new row location. This 16-byte row is called a *forwarding pointer*, and the new row that got forwarded is called a *forwarded row*. With this, any non-clustered indexes on the heap table do not need to be updated at all. If SQL Server must perform an index seek on the non-clustered index and use RID lookup against the heap table, it first goes to the row's old location and uses the 16-byte forwarding pointer to go to the location where the row now lives. Suppose the forwarded row is updated again in a way that the row moves again. In that case, SQL Server updates the forwarding pointer in the original location to reflect the new location, thus avoiding the situation of forwarding pointer pointing to another forwarding pointer.

Another benefit of forwarding pointers is that they minimize duplicate reads wherein a single row is read multiple times during a table scan within the same transaction. As an example, say SQL Server scans the data pages from page 1 to 10. We read a row on page 5, and SQL Server is currently reading page 7. An update to that row caused it to move to page 9, which has yet to be read. Since SQL Server is going in a serial left-to-right fashion, you would imagine it would re-read the row when it reaches page 9; however, that doesn't happen with the presence of forwarding pointers. SQL Server identifies the bit in the Status Bits A byte of the forwarded row and ignores re-reading it on page 9, thus avoiding duplicated reads.

Forwarding pointers introduce multiple physical reads during a single RID lookup, resulting in poor performance. We realize the real performance impact of forwarding pointers when SQL Server does heavy RID lookups, and there are many forwarding pointers. Forwarding pointers mainly occur in heaps when a variable-length column is updated to hold values larger than can fit in the same location, resulting in a not-in-place update and moving the row to another location.

Fixing forwarding pointers

It might tempt you to think heaps offer better performance than tables with a clustered index. Because unlike key lookups against a clustered index wherein SQL Server has to traverse through the index root page and then intermediate pages to get to the leaf pages for reading data, RID lookups directly go to the actual physical location of the row in the base heap table thus better performance. That is true to some extent. However, the root and intermediate index pages are most likely memory residents (buffer pool). Thus the affect of traversing through non-leaf index pages is very minimal.

The performance impact of logical reads against non-leaf clustered index pages during key lookups is minimal compared to the physical I/O introduced by the RID lookups against heaps. That said, heaps commonly offer better performance when large amounts of data must be inserted quickly, but in the overall realm of things, the benefits of having a clustered index far outweigh those of heaps.

If the forwarded row, which resulted from the column's increased length, shrinks to fit in the original location, the row may be moved back, and SQL Server removes the forwarding pointer. The same happens when the database shrinks. In a database shrink operation, the forwarded rows are moved back to their original locations as long as they can fit, and SQL Server removes the blank pages to reclaim space. However, the ideal method to get rid of forwarding pointers is to create a clustered index on the table that defines the sorted order in which SQL Server stores the rows in the index.

We can remove forwarding pointers by rebuilding the heap table with the *alter table…rebuild* statement, but the non-clustered indexes on the heap table are rebuilt when we rebuild the heap.

Demo

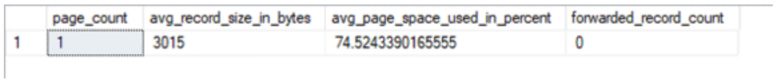
Let us see a forwarding pointer in action.

Step 1: Let us create a new database called DML_DB and create a table in it. Let us also insert two rows, about 3000 bytes each. Our intention here is to leave about 2000 bytes free on the page.

```
--create a new database called DML_DB
IF NOT EXISTS (
SELECT *
FROM sys.databases
WHERE name = 'DML_DB'
)
BEGIN
CREATE DATABASE DML_DB
END
GO
USE DML_DB
GO
--Create a table called Forwarding_Pointer
IF NOT EXISTS (
SELECT *
FROM sysobjects
WHERE name = 'Forwarding_Pointer'
AND xtype = 'U'
)
BEGIN
CREATE TABLE dbo.Forwarding_Pointer (
ID INT NOT NULL
,NAME VARCHAR(8000) NULL
)
END
--Insert 2 rows ~3000 bytes each to leave ~2000 bytes free on the page
INSERT INTO Forwarding_Pointer
VALUES (
1
,REPLICATE('a', 3000)
);
INSERT INTO Forwarding_Pointer
VALUES (
2
,REPLICATE('b', 3000)
);
GO
```

Step 2: Using the dynamic management function (DMF), sys.dm_db_index_physical_stats, we can see that there is only one page with no forwarded record. SQL Server accommodated both rows on the same page.

```
--Check the page count for the table
SELECT page_count
,avg_record_size_in_bytes
,avg_page_space_used_in_percent
,forwarded_record_count
FROM sys.dm_db_index_physical_stats(db_id('DML_DB'), object_id(N'dbo.Forwarding_Pointer'), 0, NULL, 'DETAILED');
GO
```

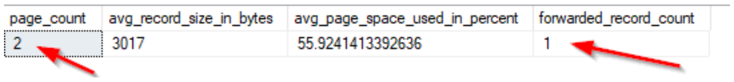


	page_count	avg_record_size_in_bytes	avg_page_space_used_in_percent	forwarded_record_count
1	1	3015	74.5243390165555	0

Step 3: Let us run three update statements against the row with ID = 1 one by one and notice which of the three updates result in a forwarding pointer. The comments for each statement explain what is happening.

```
--Update one of the two records. There is no forwarding pointer since the row size is not changing. Therefore, it is an in-place update.
UPDATE dbo.Forwarding_Pointer
SET Name = replicate('c', 3000)
WHERE ID = 1;
--Although the row size is increasing, it can still accommodate the new size on the same page, therefore still no forwarding pointer.
UPDATE dbo.Forwarding_Pointer
SET Name = replicate('a', 5000)
WHERE ID = 1;
--This update increases the row size to more than can fit on the page; therefore, a new page is allocated, resulting in forwarding pointer since a not
UPDATE dbo.Forwarding_Pointer
SET Name = replicate('a', 6000)
WHERE ID = 1;
```

By running the same dynamic management function after the third update, we can see that an increase in the row value resulting from an update caused it to move to a new page, and a forwarding pointer is created in the old row location.

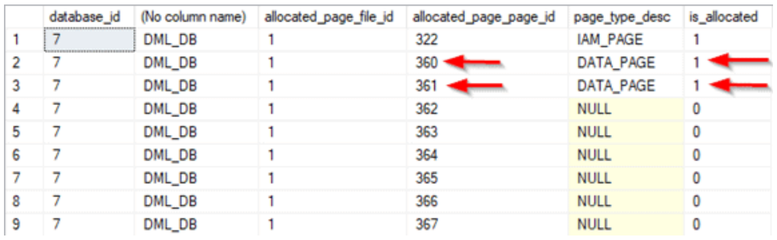


page_count	avg_record_size_in_bytes	avg_page_space_used_in_percent	forwarded_record_count
2	3017	55.9241413392636	1

Step 4: Using the dynamic management function sys.dm_db_database_page_allocations, examine the pages allocated and retrieve their page and file ids.

```
--Retrieve the page allocation details along with the page and file ids.
SELECT database_id
,db_name(database_id)
,allocated_page_file_id
,allocated_page_page_id
,page_type_desc
,is_allocated
FROM sys.dm_db_database_page_allocations(db_id('DML_DB'), object_id('Forwarding_Pointer'), NULL, NULL, 'DETAILED');
```

The eight-page extent SQL Server allocated comprises two data pages with page ids 360 and 361 and file id 1.



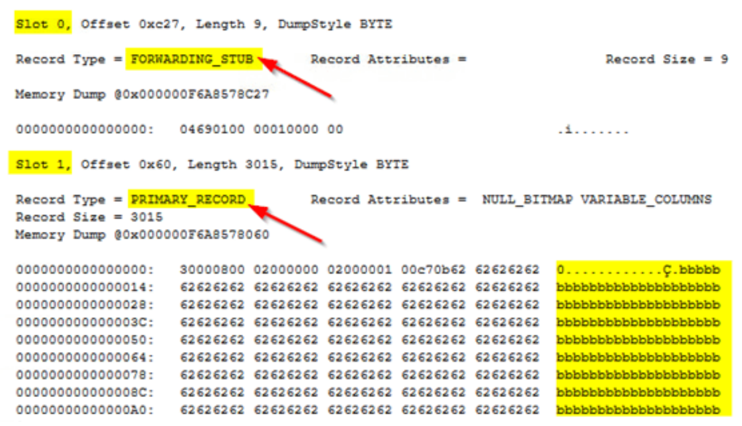
	database_id	(No column name)	allocated_page_file_id	allocated_page_page_id	page_type_desc	is_allocated
1	7	DML_DB	1	322	IAM_PAGE	1
2	7	DML_DB	1	360	DATA_PAGE	1
3	7	DML_DB	1	361	DATA_PAGE	1
4	7	DML_DB	1	362	NULL	0
5	7	DML_DB	1	363	NULL	0
6	7	DML_DB	1	364	NULL	0
7	7	DML_DB	1	365	NULL	0
8	7	DML_DB	1	366	NULL	0
9	7	DML_DB	1	367	NULL	0

Step 5: Let us use the undocumented DBCC PAGE to examine the two data pages. The parameters in DBCC PAGE are: database name, file id, page id, printopt (0 - 3, 1 prints page header and slot array dump for each row. Use trace 3604 to display the output in SSMS.

```
DBCC TRACEON (3604);
GO
DBCC PAGE(DML_DB, 1, 360, 1) —examining page 360
GO
```

I show the partial output below. Slot 0 shows the record type as FORWARDING_STUB, meaning this slot had the row with ID = 1, Name = aaaa… 6000 times.

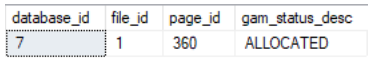
And because we updated the row to a larger value than the page could accommodate, it moved it to page 361 and replaced the original row on page 360 with a forwarding pointer. The next row is on slot 1 with PRIMARY_RECORD as the record type, showing that it is a normal record and value bbbb… 3000 times. The column *Name* value is highlighted on the right.



```
Slot 0, Offset 0xc27, Length 9, DumpStyle BYTE

Record Type = FORWARDING_STUB       Record Attributes =                 Record Size = 9

Memory Dump @0x000000F6A8578C27

0000000000000000:   04690100 00010000 00                          .i.......

Slot 1, Offset 0x60, Length 3015, DumpStyle BYTE

Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 3015
Memory Dump @0x000000F6A8578060

0000000000000000:   30000800 02000000 02000001 00c70b62 62626262  0...........Ç.bbbbb
0000000000000014:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
0000000000000028:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
000000000000003C:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
0000000000000050:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
0000000000000064:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
0000000000000078:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
000000000000008C:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
00000000000000A0:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
```

SQL Server 2019 introduced a new dynamic management function sys.dm_db_page_info, which can replace DBCC PAGE sometimes.

```
--SQL 2019 introduced a new dynamic management function which returns a similar result as DBCC PAGE
--sys.dm_db_page_info (DatabaseId, FileId, PageId, Mode)
SELECT database_id, file_id, page_id, gam_status_desc
FROM sys.dm_db_page_info(7, 1, 360, 'detailed')
```

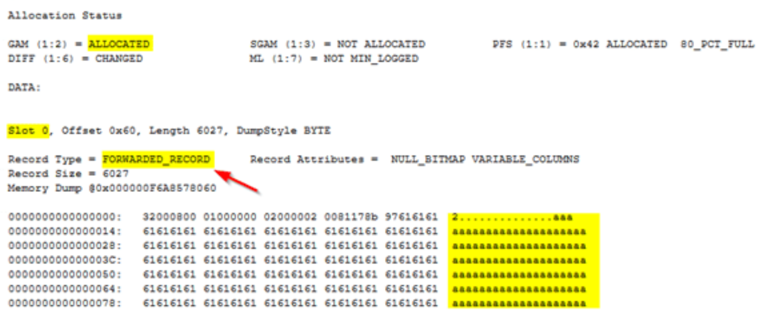


database_id	file_id	page_id	gam_status_desc
7	1	360	ALLOCATED

Step 6: Let's examine page 361 and verify if the forwarded row exists on it.

```
DBCC PAGE(DML_DB, 1, 361, 1)
GO
```

There it is! The only record that's on page 361 is the one with ID = 1. Notice the record type shows as FORWARDED_RECORD, confirming that the record is a forwarded record.

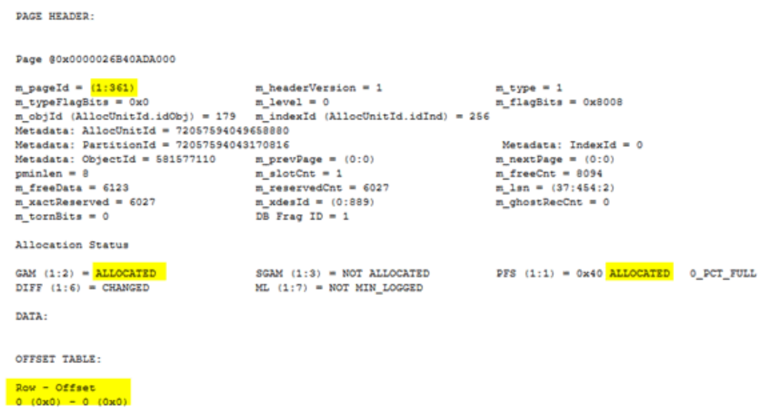


```
Allocation Status

GAM (1:2) = ALLOCATED              SGAM (1:3) = NOT ALLOCATED          PFS (1:1) = 0x42 ALLOCATED   80_PCT_FULL
DIFF (1:6) = CHANGED               ML (1:7) = NOT MIN_LOGGED

DATA:


Slot 0, Offset 0x60, Length 6027, DumpStyle BYTE

Record Type = FORWARDED_RECORD     Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 6027
Memory Dump @0x000000F6A8578060

0000000000000000:   32000800 01000000 02000002 0081178b 97616161  2...............aaa
0000000000000014:   61616161 61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaaaaaa
0000000000000028:   61616161 61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaaaaaa
000000000000003C:   61616161 61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaaaaaa
0000000000000050:   61616161 61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaaaaaa
0000000000000064:   61616161 61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaaaaaa
0000000000000078:   61616161 61616161 61616161 61616161 61616161  aaaaaaaaaaaaaaaaaaaa
```

Step 7: Let's remove the only row from page 361 so it becomes empty and examine the page.

```
--Delete the only row on the page
DELETE
FROM Forwarding_Pointer
WHERE id = 1
--Even after no row exists on the page, it still shows allocated and the space isn't available for other objects
DBCC PAGE(DML_DB, 1, 361, 1)
GO
```

Although the page is blank, the ghost cleanup thread has not deallocated it because it is a heap table, so it remains allocated. Also, note that since we used printopt 1 in our dbcc page command, the deleted record doesn't show up, but it still physically exists on the page. Using printopt option 2 shows the deleted record and the bytes it used.



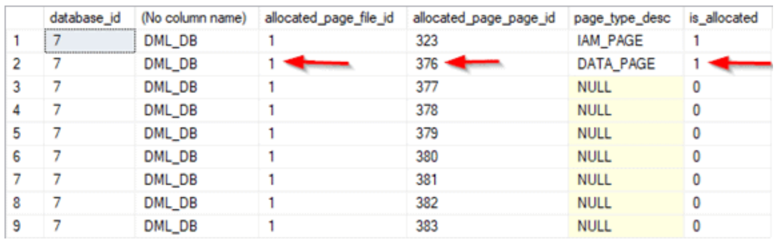
```
PAGE HEADER:


Page @0x0000026B40ADA000

m_pageId = (1:361)                      m_headerVersion = 1                   m_type = 1
m_typeFlagBits = 0x0                    m_level = 0                           m_flagBits = 0x8008
m_objId (AllocUnitId.idObj) = 179       m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594049658880
Metadata: PartitionId = 72057594043170816                                      Metadata: IndexId = 0
Metadata: ObjectId = 581577110          m_prevPage = (0:0)                    m_nextPage = (0:0)
pminlen = 8                             m_slotCnt = 1                         m_freeCnt = 8094
m_freeData = 6123                       m_reservedCnt = 6027                  m_lsn = (37:454:2)
m_xactReserved = 6027                   m_xdesId = (0:889)                    m_ghostRecCnt = 0
m_tornBits = 0                          DB Frag ID = 1

Allocation Status

GAM (1:2) = ALLOCATED                   SGAM (1:3) = NOT ALLOCATED            PFS (1:1) = 0x40 ALLOCATED   0_PCT_FULL
DIFF (1:6) = CHANGED                    ML (1:7) = NOT MIN_LOGGED

DATA:


OFFSET TABLE:

Row - Offset
0 (0x0) - 0 (0x0)
```

Step 8: Let's rebuild the table and find what new pages SQL allocates because of the rebuild.

```
--Let's rebuild the heap to fix the forwarding pointer
ALTER TABLE Forwarding_Pointer rebuild
--All new pages are allocated. The page with deleted row is deallocated
```

```
SELECT database_id
,db_name(database_id)
,allocated_page_file_id
,allocated_page_page_id
,page_type_desc
,is_allocated
FROM sys.dm_db_database_page_allocations(db_id('DML_DB'), object_id('Forwarding_Pointer'), NULL, NULL, 'DETAILED');
```

The table's rebuild resulted in the only row (with ID = 2) being moved to a new page 376. The table has only one row at this point. As you might recall, we deleted the row with ID = 1 earlier.

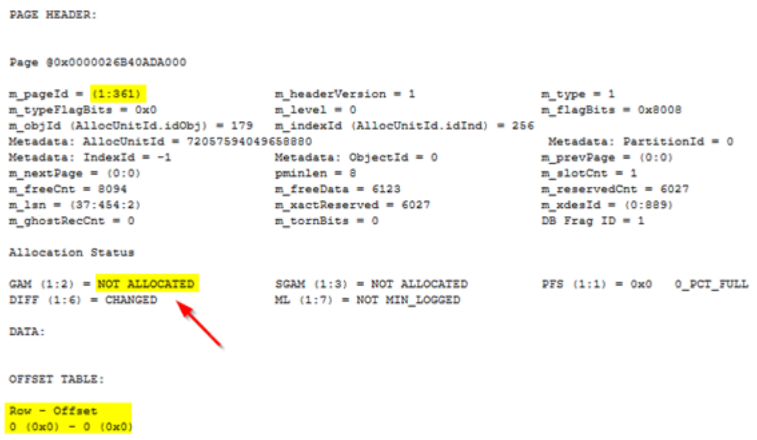


	database_id	(No column name)	allocated_page_file_id	allocated_page_page_id	page_type_desc	is_allocated
1	7	DML_DB	1	323	IAM_PAGE	1
2	7	DML_DB	1	376	DATA_PAGE	1
3	7	DML_DB	1	377	NULL	0
4	7	DML_DB	1	378	NULL	0
5	7	DML_DB	1	379	NULL	0
6	7	DML_DB	1	380	NULL	0
7	7	DML_DB	1	381	NULL	0
8	7	DML_DB	1	382	NULL	0
9	7	DML_DB	1	383	NULL	0

Let's check the old pages that previously had the two rows, i.e., page 361 and 360.

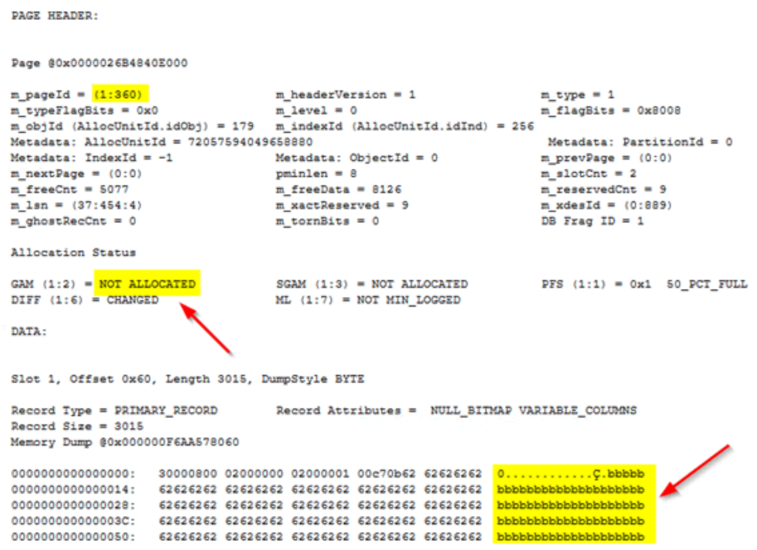
```
--check what the old 2 pages hold now that we have rebuilt the table
--both show not allocated, meaning any new inserts can use them
--however, notice the page 360 that previously had data still has old rows which any queries cannot see but still exist on the page physically
DBCC PAGE(DML_DB, 1, 361, 1)
GO
DBCC PAGE(DML_DB, 1, 360, 1)
GO
```

Post heap rebuild, page 361, which was otherwise empty, now shows not allocated and is available for any new inserts into this table.



```
PAGE HEADER:


Page @0x0000026B40ADA000

m_pageId = (1:361)                  m_headerVersion = 1                 m_type = 1
m_typeFlagBits = 0x0                m_level = 0                         m_flagBits = 0x8008
m_objId (AllocUnitId.idObj) = 179   m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594049658880                                Metadata: PartitionId = 0
Metadata: IndexId = -1              Metadata: ObjectId = 0              m_prevPage = (0:0)
m_nextPage = (0:0)                  pminlen = 8                         m_slotCnt = 1
m_freeCnt = 8094                    m_freeData = 6123                   m_reservedCnt = 6027
m_lsn = (37:454:2)                  m_xactReserved = 6027               m_xdesId = (0:889)
m_ghostRecCnt = 0                   m_tornBits = 0                      DB Frag ID = 1

Allocation Status

GAM (1:2) = NOT ALLOCATED           SGAM (1:3) = NOT ALLOCATED          PFS (1:1) = 0x0   0_PCT_FULL
DIFF (1:6) = CHANGED                ML (1:7) = NOT MIN_LOGGED

DATA:


OFFSET TABLE:

Row - Offset
0 (0x0) - 0 (0x0)
```

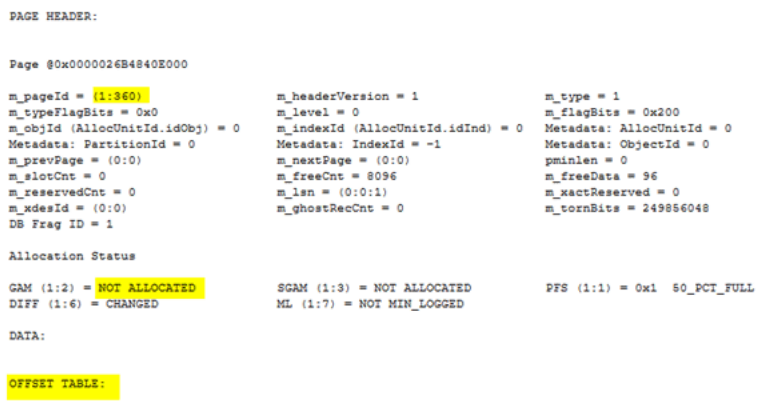
Page 360 initially had the record with ID = 2 (the record SQL Server moved to page 376 upon rebuild) shows not allocated and is available for future inserts. However, SQL did not physically remove the records from the page. The same holds for page 361 (use printopt 2 of dbcc page).



```
PAGE HEADER:


Page @0x0000026B4840E000

m_pageId = (1:360)                  m_headerVersion = 1                 m_type = 1
m_typeFlagBits = 0x0                m_level = 0                         m_flagBits = 0x8008
m_objId (AllocUnitId.idObj) = 179   m_indexId (AllocUnitId.idInd) = 256
Metadata: AllocUnitId = 72057594049658880                                Metadata: PartitionId = 0
Metadata: IndexId = -1              Metadata: ObjectId = 0              m_prevPage = (0:0)
m_nextPage = (0:0)                  pminlen = 8                         m_slotCnt = 2
m_freeCnt = 5077                    m_freeData = 8126                   m_reservedCnt = 9
m_lsn = (37:454:4)                  m_xactReserved = 9                  m_xdesId = (0:889)
m_ghostRecCnt = 0                   m_tornBits = 0                      DB Frag ID = 1

Allocation Status

GAM (1:2) = NOT ALLOCATED           SGAM (1:3) = NOT ALLOCATED          PFS (1:1) = 0x1   50_PCT_FULL
DIFF (1:6) = CHANGED                ML (1:7) = NOT MIN_LOGGED

DATA:


Slot 1, Offset 0x60, Length 3015, DumpStyle BYTE

Record Type = PRIMARY_RECORD        Record Attributes =  NULL_BITMAP VARIABLE_COLUMNS
Record Size = 3015
Memory Dump @0x000000F6AA578060

0000000000000000:   30000800 02000000 02000001 00c70b62 62626262  0............Ç.bbbbb
0000000000000014:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
0000000000000028:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
000000000000003C:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
0000000000000050:   62626262 62626262 62626262 62626262 62626262  bbbbbbbbbbbbbbbbbbbb
```

We removed the ghosted rows by manually running sp_clean_db_free_space.

```
--clean up the ghosted rows across the database. This will wipe out ghosted rows on page 360
EXEC sp_clean_db_free_space 'dml_db'
--see if the ghosted records are gone
DBCC PAGE(DML_DB, 1, 360, 1)
GO
```



```
PAGE HEADER:


Page @0x0000026B4840E000

m_pageId = (1:360)                  m_headerVersion = 1                 m_type = 1
m_typeFlagBits = 0x0                m_level = 0                         m_flagBits = 0x200
m_objId (AllocUnitId.idObj) = 0     m_indexId (AllocUnitId.idInd) = 0   Metadata: AllocUnitId = 0
Metadata: PartitionId = 0           Metadata: IndexId = -1              Metadata: ObjectId = 0
m_prevPage = (0:0)                  m_nextPage = (0:0)                  pminlen = 0
m_slotCnt = 0                       m_freeCnt = 8096                    m_freeData = 96
m_reservedCnt = 0                   m_lsn = (0:0:1)                     m_xactReserved = 0
m_xdesId = (0:0)                    m_ghostRecCnt = 0                   m_tornBits = 249856048
DB Frag ID = 1

Allocation Status

GAM (1:2) = NOT ALLOCATED           SGAM (1:3) = NOT ALLOCATED          PFS (1:1) = 0x1  50_PCT_FULL
DIFF (1:6) = CHANGED                ML (1:7) = NOT MIN_LOGGED

DATA:


OFFSET TABLE:
```

Conclusion

As a rule of thumb, always have a clustered index on tables to avoid forwarding pointers unless it's a staging environment and speed of insert is of the essence, in which case it may be acceptable to have heaps. Forwarding pointers potentially introduce extra physical reads where SQL Server has to reroute from one page with *forwarding_stubs* to the other with *forwarded_records* in a single read operation, affecting performance. Forwarding pointers can be removed by rebuilding the heap table, but that causes the non-clustered indexes that have links to the heap via physical row-ids to be rebuilt. Last, SQL Server doesn't deallocate empty pages in heaps via ghost cleanup thread, and to get around the problem, either have a clustered index or rebuild the table.

Now that we understand DML on heaps, in the following article, we will see what happens during insert, update, and delete against b-tree indexes and what causes pages to split and result in index fragmentation.