Understanding the Internals of a Data Page

Introduction

The data SQL Server logically stores in tables is physically stored on 8 KB data pages. While there are several types of pages in SQL Server, such as Boot page, Page Free Space (PFS), Global Allocation Map (GAM), Index page, etc., data pages are the core of storage structures such as heaps and indexes.

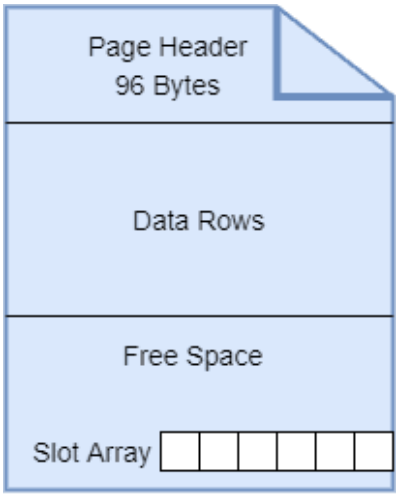
There are three types of data pages in SQL Server: in-row, row-overflow, and LOB data pages. Knowing the internals of a data page is critical to understanding what happens internally when we change data, how data rows consume space on data pages, how fixed-length columns differ from variable-length columns in terms of space requirements. This knowledge also helps us understand advanced concepts, such as page-splits and index fragmentation, which we will cover in separate articles.

In this article, we focus mainly on the structure and types of data pages. Later in the article, we look at the data storage structures (heaps, b-trees, columnstore) used by data pages and how SQL Server physically organizes data in those structures.

Note: This is the first 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:

Structure of a data page

SQL Server numbers the data pages consecutively, starting with zero. Each page is uniquely identified by its page number combined with the database file ID. An 8 KB data page starts with the page header of size 96 bytes, followed by data rows and a row offset array at the end.



Structure of a Data Page

Page header

The page header consumes 96 bytes out of the total 8192 bytes, leaving 8096 bytes for the data rows and offset array. It comprises information including, but not limited to, the following.

1. Page ID, which includes the file number and page number in the database.

2. Object ID of the object to which the page belongs.
3. Log Sequence Number or LSN, which corresponds to the last log record that modified the page.
4. Free bytes on the page.
5. Torn bits string for storing checksum information, which detects whether or not the page is corrupt.

Data rows

For a table with fixed-length columns, a single row must fit on a single page and have a maximum size of 8060 bytes of in-row data. Fixed-length columns are those with fixed-length data types, such as int and char, that consume the same amount of space regardless of the actual value stored or whether the column contains nulls. For rows with variable-length and LOB columns, SQL Server stores the data on row-overflow and LOB data pages if in-row storage is not possible.

Variable-length columns are those with variable-length data types, such as varchar and varbinary, which consume space that is actually required to store the data and a few additional bytes of overhead. If a table has all fixed-length columns, then the number of rows that we can store on each page is constant, and so is the total length of each row. However, if there are variable-length columns in a table, then the number of rows per page varies with the data's actual length in the variable-length columns. The data rows are followed by free space on the page to store new rows.

Row offset array

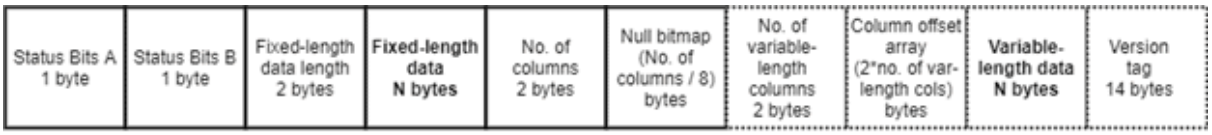
The last portion of the data page is the row offset array, which provides each row's position on the page. The offset array is a block of two-byte entries where each entry dictates its corresponding row's address where the row begins on the page. In other words, the slot array holds two bytes for every row on the page. Those two bytes store the address of the corresponding row on the page. The array defines the logical ordering for the rows on the data page, meaning SQL Server looks at the slot array to determine the order in which to read the rows on the page.

For example, in a clustered index, the leaf pages contain data ordered as per the clustering key column. The slot array on those leaf data pages provides the logical ordering of rows wherein slot 0 stores the lowest key value offset, followed by slot 1 with the next lowest key value offset, and so on, in the index key order. The actual physical rows may be scattered in a disorderly fashion on the page; therefore, the row offset array defines the rows' logical order.

For example, say a clustered index key column is ID with an Identity data type, and there are ten rows with ID 1, 2, 3...10 on a data page. The slot array on that page is populated based on the index key order. That is, the rightmost two-byte entry in the slot array (slot 0) corresponds to the address for the row with the lowest key column, i.e., ID = 1, next to its left is the two-byte entry (slot 1) pointing to the row with the second-lowest key value (ID = 2), and so on. The two bytes per row in the slot array is additional overhead on top of the row size.

Structure of a data row

This next section explains how SQL Server physically stores a row on a data page. SQL Server stores an uncompressed row on the data page such that all fixed-length columns have their data stored first, followed by the variable-length columns. The image below shows the general structure.



Structure of a Data Row

The two bytes that appear first in the row are Status Bits A and B, respectively. The eight bits in A (0 to 7) hold information such as whether the record is a primary record, forwarded record, forwarding stub, index record, row-overflow record, ghost data, or index record. The Status Bit A also dictates whether variable-length columns exist in the row, whether the row has a version tag.

Status Bits in B contain information on whether the record is a ghost forwarded record.

The following two bytes hold the fixed-length part of the row, followed by the fixed-length data.

The following two bytes after fixed-length data are for the number of columns. After that, there is a null bitmap with 1 bit for each column, whether or not columns are nullable.

The portion after the null bitmap is for the variable-length columns. It starts with two bytes showing the number of variable-length columns followed by a column offset array which holds a two-byte offset value for each variable-length column, even for the ones with nulls. The column offset array size is (*2* no. of variable-length columns) bytes.*

Then comes the actual variable-length data. Finally, there is a 14-byte version tag at the row end. SQL Server populates this tag for transactions running under optimistic isolation level such as snapshot and other operations such as online index rebuild.

Place your nullable variable-length columns at the end of the table to save space

The column offset array shows the position within the row where the variable-length column ends. If a row has any variable-length columns, the column offset array comprises two-bytes for each variable-length column. If there are nulls in the variable-length columns, they don't take any space in the variable-length data portion of the row, but the column offset array still consumes two-bytes for each variable-length column.

Therefore, while variable-length columns with nulls don't consume space in the variable-length data portion of the rows, they consume two-bytes per column in the column offset array. However, there is a particular case where the variable-length columns with null values don't consume the two-bytes in the column offset array. That is when the variable-length columns (with null values) are stored at the end of the list of the variable-length columns in the *create table* statement. For example, say we have four variable-length columns in the order var1, var2, var3, and var4. If a row has a non-null value in var1 but nulls in the rest of the columns that follow var1, except for var1, there would be no two-byte entries in the column offset array for the three null columns, therefore saving six bytes on that row. However, if var4 is has a non-null value and the remaining columns are all nulls, because the trailing column var4 is non-null, all four columns will have two-byte entries each in the column offset array even though columns var1 - var3 have all null values. Therefore, the column offset array in this case will consume eight bytes.

We can reduce the size of the data rows by simply putting the variable-length columns, which contain null values, last in the *create table* statement. That way, the data rows with those variable-length columns as nulls don't have the two-byte entries in the column offset array, thus reducing the overall row size.

Tip: Always have the variable-length columns, which you suspect to have lots of nulls, at the end of the *create table* statement. This order of the columns when creating a table can save space in the long run as the table size increases. On the other hand, fixed-length columns always have the same length whether or not they contain nulls.

In-row, row-overflow, LOB pages

Data pages in SQL Server come in three forms: in-row, row-overflow, and LOB pages. Let us examine each one of these.

In-row page

SQL Server always attempts to store a data row on an in-row page as long as it can fit on a single page. An in-row data page can accommodate a single row of maximum size equal to 8060 bytes, including the overhead bytes. All fixed-length columns along with internal overhead bytes of a row must fit on a single in-row page.

If there are variable-length columns in the row and the page does not have enough space to accommodate the entire row, SQL Server allocates row-overflow pages and stores the variable-length columns on them. If SQL Server must allocate row-overflow pages, it adds a 24-byte pointer per row on the in-row page, pointing to the variable-length column's location on the row-overflow page, creating a link from in-row to row-overflow pages. This 24-byte is included in the maximum limit of 8060 bytes an in-row data row can have.

Let us see how SQL Server stores a record on an in-row data page. In the following example, we create a new database called TestDB and a new table called InRowTbl. We enter a row of size enough to fit on a single page. We are looking to understand what SQL Server's first choice of the data page is. That is, whether it stores the record on an in-row, row-overflow, or a LOB page.

Using the undocumented command DBCC IND, we will see all the pages allocated to the new table. Last, we run another undocumented command, DBCC PAGE, to examine the page contents.

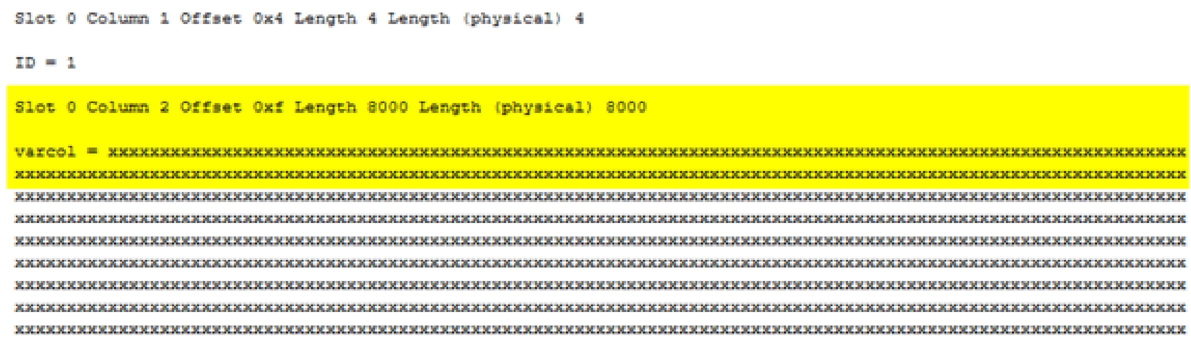
```
CREATE DATABASE TestDB
GO
--create a new table and insert a record of size enough to fit in a single page
DROP TABLE IF EXISTS InRowTbl
CREATE TABLE dbo.InRowTbl (
    ID INT NOT NULL
   ,varcol VARCHAR(8000) NULL
)

INSERT INTO dbo.InRowTbl (
        ID
        ,varcol
        )
VALUES (
        1
        ,replicate('x', 8000)
        )
--using undocumented dbcc ind, check all the pages allocated for the table
DBCC IND('TestDB' --database name
               , 'dbo.InRowTbl' --table name
               , -1 --print info for all pages
        )
--examine the page contents using undocumented dbcc page
DBCC TRACEON (3604);--print the output to console
DBCC PAGE('TestDB' --database name
               , 1 -- file ID
               , 352 --Page ID
               , 3 -- Print mode = 3 displays header and row information
        )
```

DBCC IND gave the following output. The page with page ID 322, which shows the page type as 10, is an Index Allocation Map or IAM page tied to the data page with page ID 352. Page 352 shows 1 as the page

type, which means it's an in-row data page, as shown in the iam_chain_type column. Therefore, SQL Server stored the one row on an in-row page.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	322	NULL	NULL	581577110	0	1	72057594043170816	In-row data	10
1	352	1	322	581577110	0	1	72057594043170816	In-row data	1

Moving on, let's see what is on page 352 by running DBCC PAGE against it. The following screenshot shows the partial portion of the slot array.



Sure enough, our record is entirely stored on page 352. The slot 0 contains two columns: ID = 1, and varcol = xxxx (a total of 8000 x values).

Row-overflow page

If SQL Server cannot accommodate a row on a single page with variable-length columns, SQL Server allocates row-overflow pages to store them. Row-overflow pages apply only to variable-length columns, which are no larger than 8000 bytes each. Fixed-length columns, on the other hand, must all add up to 8060 bytes. A single row can span multiple row-overflow pages if a table contains many large variable-length columns with no more than 8000 bytes per column.

Let us see a row-overflow situation in action. We repeat the steps we performed in the in-row section above, except instead of a record that would fit in a single page, we insert a record so large that SQL needs to allocate multiple pages to accommodate it. Notice that each variable-length column does not exceed the 8000-byte limit, one of the required conditions for a variable-length column to be stored off-row on a row-overflow page.

```
--create a new table and insert a record of size large so that it spans multiple data pages
DROP TABLE IF EXISTS RowOverflowTbl;
CREATE TABLE dbo.RowOverflowTbl
(   ID      INT           NOT NULL
  , varcol1 VARCHAR(8000) NULL
  , varcol2 VARCHAR(8000) NULL
  , varcol3 VARCHAR(8000) NULL);
INSERT INTO dbo.RowOverflowTbl
    (ID, varcol1, varcol2, varcol3)
VALUES
    (1, REPLICATE ('x', 8000), REPLICATE ('y', 8000), REPLICATE ('z', 8000));
--using undocumented dbcc ind, check all the pages allocated for the table
DBCC IND('TestDB' --database name
, 'dbo.RowOverflowTbl' --table name
,-1 --print info for all pages
);
--examine the page contents using undocumented dbcc page
DBCC TRACEON(3604);
--print the output to console
DBCC PAGE('TestDB' --database name
, 1 -- file ID
, 368 --Page ID
```

```
, 3 -- Print mode = 3 displays header and row information
);
```

DBCC IND output shows that SQL Server allocated 6 pages, of which two are IAM (PageType 10), four are data pages, of which page 368 is in-row and 360, 361, 362 are all row-overflow data pages. SQL Server started with in-row first and then allocated three row-overflow pages to accommodate the three variable-length columns.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	324	NULL	NULL	597577167	0	1	72057594043236352	In-row data	10
1	368	1	324	597577167	0	1	72057594043236352	In-row data	1
1	323	NULL	NULL	597577167	0	1	72057594043236352	Row-overflow data	10
1	360	1	323	597577167	0	1	72057594043236352	Row-overflow data	3
1	361	1	323	597577167	0	1	72057594043236352	Row-overflow data	3
1	362	1	323	597577167	0	1	72057594043236352	Row-overflow data	3

Let us examine the in-row page 368 using DBCC PAGE and see what it contains.

```
Slot 0 Column 1 Offset 0x4 Length 4 Length (physical) 4

ID = 1

varcol1 = [BLOB Inline Root] Slot 0 Column 2 Offset 0x13 Length 24 Length (physical) 24

Level = 0                    Unused = 0                    UpdateSeq = 1
TimeStamp = 938934272        Type = 2
Link 0

Size = 8000                  RowId = (1:360:0)

varcol2 = [BLOB Inline Root] Slot 0 Column 3 Offset 0x2b Length 24 Length (physical) 24

Level = 0                    Unused = 0                    UpdateSeq = 1
TimeStamp = 1894842368       Type = 2
Link 0

Size = 8000                  RowId = (1:361:0)

varcol3 = [BLOB Inline Root] Slot 0 Column 4 Offset 0x43 Length 24 Length (physical) 24

Level = 0                    Unused = 0                    UpdateSeq = 1
TimeStamp = 1103364096       Type = 2
Link 0

Size = 8000                  RowId = (1:362:0)
```

The in-row page 368 has the first column value (ID = 1) and pointers to the three row-overflow pages 360, 361, and 362. The pointers are the row ids containing the file ID, page number, and slot number.

Let us pick the row-overflow page 360 and see what it has. Since it is associated with varcol1, we expect to see the value x, 8000 times.

```
Blob row at: Page (1:360) Slot 0 Length: 8014 Type: 3 (DATA)

Blob Id:938934272


000000F6A23F806E:  78787878  78787878  78787878  78787878 xxxxxxxxxxxxxxxx

000000F6A23F807E:  78787878  78787878  78787878  78787878 xxxxxxxxxxxxxxxx

000000F6A23F808E:  78787878  78787878  78787878  78787878 xxxxxxxxxxxxxxxx

000000F6A23F809E:  78787878  78787878  78787878  78787878 xxxxxxxxxxxxxxxx
```

And yes, page 360 contains the column varcol1 at slot 0, as shown in the partial DBCC PAGE output above. Page type 3 represents a row-overflow data page. Similarly, the row-overflow pages 361 and 362 contain varcol2 and varcol3 data, respectively.

LOB (Large OBject) page

If a column needs to be larger than 8000 bytes, then consider using LOB data types such as varchar (max), nvarchar (max), varbinary (max). For these max columns, SQL Server may choose among in-row, row-overflow, and LOB pages depending on the column size. It starts with in-row, and if it is not an option and the column size is not larger than 8000 bytes, it allocates a row-overflow page. If the column exceeds 8000 bytes, SQL Server stores it on LOB pages.

The deprecated LOB types such as text, ntext, and image columns are all stored on LOB pages by default.

Let us see a LOB page in action. Again, we repeat the same steps, except we insert a massive record into the table. Besides the two variable-length columns, varcol1 and varcol2, we insert a 32,000 byte-sized value in a deprecated LOB *text* column. Our expectation here is to get SQL Server to allocate LOB pages for the text column since we surpass the 8 KB byte limit to store on row-overflow pages.

```
--create a new table and insert a record of size very large to make it allocate a LOB page
DROP TABLE IF EXISTS LobTbl;
CREATE TABLE dbo.LobTbl
(   ID      INT           NOT NULL
  , varcol1 VARCHAR(8000) NULL
  , varcol2 VARCHAR(8000) NULL
  , lobcol  TEXT          NULL);
INSERT INTO dbo.LobTbl
    (ID, varcol1, varcol2, lobcol)
VALUES
    (1, REPLICATE ('x', 8000), REPLICATE ('y', 8000), REPLICATE (CONVERT (VARCHAR(MAX), 'z'), 32000));
--using undocumented dbcc ind, check all the pages allocated for the table
DBCC IND('TestDB' --database name
, 'dbo.LobTbl' --table name
,-1 --print info for all pages
);
--examine the page contents using undocumented dbcc page
DBCC TRACEON(3604);
--print the output to console
DBCC PAGE('TestDB' --database name
, 1 -- file ID
, 400 --Page ID
, 3 -- Print mode = 3 displays header and row information
);
```

Below is the output of DBCC IND.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	327	NULL	NULL	613577224	0	1	72057594043301888	In-row data	10
1	400	1	327	613577224	0	1	72057594043301888	In-row data	1
1	326	NULL	NULL	613577224	0	1	72057594043301888	Row-overflow data	10
1	392	1	326	613577224	0	1	72057594043301888	Row-overflow data	3
1	325	NULL	NULL	613577224	0	1	72057594043301888	LOB data	10
1	384	1	325	613577224	0	1	72057594043301888	LOB data	3
1	385	1	325	613577224	0	1	72057594043301888	LOB data	3
1	386	1	325	613577224	0	1	72057594043301888	LOB data	3
1	387	1	325	613577224	0	1	72057594043301888	LOB data	3
1	388	1	325	613577224	0	1	72057594043301888	LOB data	3

SQL Server allocated ten pages: 3 IAM pages, 1 in-row page (page 400), 1 row-overflow page (page 392), 1 LOB root page (page 388, in green), 4 child LOB data pages (pages 384 - 387, in red).

Let us examine the in-row page 400 using DBCC PAGE.

```
Slot 0 Column 1 Offset 0x4 Length 4 Length (physical) 4

ID = 1

Slot 0 Column 2 Offset 0x13 Length 8000 Length (physical) 8000

varcol1 = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

As expected, SQL Server stored the first column ID (=1) and the 8000-byte varcol1 column on the in-row data page 400.

We now need to know where varcol2 and lobcol columns are stored. The same in-row page 400 also contains pointers to the row-overflow page 392 and the root LOB page 388. The root LOB page doesn't store data itself, but it has pointers to the child LOB pages which contain the data. So in this example, the LOB root page 388 points to the child LOB pages.

```
varcol2 = [BLOB Inline Root] Slot 0 Column 3 Offset 0x1f53 Length 24 Length (physical) 24

Level = 0                       Unused = 0                      UpdateSeq = 1
TimeStamp = 83755008            Type = 2
Link 0

Size = 8000                     RowId = (1:392:0)

lobcol = [Textpointer] Slot 0 Column 4 Offset 0x1f6b Length 16 Length (physical) 16

TextTimeStamp = 131137536       RowId = (1:388:0)
```

Notice the 24-byte pointer on the in-row page pointing to the row-overflow address is reduced to a 16-byte pointer for storing the LOB page address. From the screenshot, varcol2 is stored on the row-overflow page 392. The lobcol column points to page 388. Let's examine page 388.

```
Blob row at: Page (1:388) Slot 0 Length: 84 Type: 5 (LARGE_ROOT_YUKON)

Blob Id: 131137536 Level: 0 MaxLinks: 5 CurLinks: 4

    Child 0 at Page (1:385) Slot 0 Size: 8040 Offset: 8040

    Child 1 at Page (1:386) Slot 0 Size: 8040 Offset: 16080

    Child 2 at Page (1:387) Slot 0 Size: 8040 Offset: 24120

    Child 3 at Page (1:384) Slot 0 Size: 7880 Offset: 32000
```

The DBCC PAGE against page 388 reveals it is a root LOB page that has pointers to the child LOB pages 384 - 387. Each child page stored 8000 bytes, and since our lobcol column is 32,000 bytes, each of the four child LOB pages stored 8000 bytes of data.

It is important to note that the larger the LOB column gets, the more the LOB structure levels. That is, if the LOB column value is not more than 32KB and can be accommodated across five data pages, the root LOB page has direct pointers to the child LOB pages, as we saw in the above example. However, suppose it takes more than five pages to store the large value. In that case, SQL Server allocates intermediate levels between the root LOB page and the child pages like the B-tree structure, thus potentially affecting performance due to long links.

Deprecated LOB data types vs. MAX data types

In deprecated LOB columns such as text, SQL Server stores the value on LOB pages by default even when the size of the value can easily fit on an in-row data page. On the other hand, the newer max columns, such as varchar (max), store on in-row pages as long as the data can fit. Otherwise, it stores the data on LOB pages.

Let us quickly demonstrate this. We insert a 4000-byte value into the LOB column lobcol of the text data type.

```
--text vs. varchar(max)
DROP TABLE IF EXISTS [text]
CREATE TABLE dbo.[text] (
ID INT NOT NULL
,varcol1 VARCHAR(8000) NULL
,varcol2 VARCHAR(8000) NULL
,lobcol TEXT NULL
);
INSERT INTO dbo.[text] (
ID
,varcol1
,varcol2
,lobcol
)
VALUES (
1
,replicate('x', 8000)
,replicate('y', 8000)
,replicate('z', 4000)
);
DBCC IND('TestDB' --database name
, 'dbo.text' --table name
, -1 --print info for all pages
```

Let us see what pages SQL Server allocated for the table.

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	429	NULL	NULL	741577680	0	1	72057594043826176	In-row data	10
1	456	1	429	741577680	0	1	72057594043826176	In-row data	1
1	428	NULL	NULL	741577680	0	1	72057594043826176	Row-overflow data	10
1	448	1	428	741577680	0	1	72057594043826176	Row-overflow data	3
1	427	NULL	NULL	741577680	0	1	72057594043826176	LOB data	10
1	440	1	427	741577680	0	1	72057594043826176	LOB data	3

SQL Server stored the column ID and varcol1 on the in-row page 456, varcol2 on row-overflow page 448, and lobcol on LOB page 440, even though it used only 4000 bytes.

Let us change the text data type to varchar(max) and run the example.

```
DROP TABLE IF EXISTS [varcharmax]
CREATE TABLE dbo.[varcharmax] (
ID INT NOT NULL
,varcol1 VARCHAR(8000) NULL
,varcol2 VARCHAR(8000) NULL
,lobcol VARCHAR(max) NULL
);
INSERT INTO dbo.[varcharmax] (
ID
,varcol1
,varcol2
,lobcol
)
VALUES (
1
,replicate('x', 8000)
,replicate('y', 8000)
,replicate('z', 4000)
);
DBCC IND('TestDB' --database name
, 'dbo.varcharmax' --table name
, -1 --print info for all pages
)
DBCC TRACEON (3604);--print the output to console
DBCC PAGE('TestDB' --database name
, 1 -- file ID
, 472 --Page ID
, 3 -- Print mode = 3 displays header and row information
)
```

PageFID	PagePID	IAMFID	IAMPID	ObjectID	IndexID	PartitionNumber	PartitionID	iam_chain_type	PageType
1	431	NULL	NULL	757577737	0	1	72057594043891712	In-row data	10
1	472	1	431	757577737	0	1	72057594043891712	In-row data	1
1	430	NULL	NULL	757577737	0	1	72057594043891712	Row-overflow data	10
1	464	1	430	757577737	0	1	72057594043891712	Row-overflow data	3
1	465	1	430	757577737	0	1	72057594043891712	Row-overflow data	3

Excluding the two IAM pages, SQL Server allocated one in-row (page 472), two row-overflow pages (464 and 465). Examining the in-row page 472 reveals that SQL Server stored the ID column and LOB column lobcol values on it. The two variable-length columns, varcol1 and varcol2, are stored on the row-overflow pages 464 and 465, respectively. Therefore, even though we had a LOB data type for lobcol column, SQL Server still stored it on an in-row page since the data size could accommodate it.

```
Slot 0 Column 1 Offset 0x4 Length 4 Length (physical) 4

ID = 1

varcol1 = [BLOB Inline Root] Slot 0 Column 2 Offset 0x13 Length 24 Length (physical) 24

Level = 0                    Unused = 0                    UpdateSeq = 1
TimeStamp = 2686976          Type = 2
Link 0

Size = 8000                  RowId = (1:464:0)

varcol2 = [BLOB Inline Root] Slot 0 Column 3 Offset 0x2b Length 24 Length (physical) 24

Level = 0                    Unused = 0                    UpdateSeq = 1
TimeStamp = 1210253312       Type = 2
Link 0

Size = 8000                  RowId = (1:465:0)

lobcol = [BLOB Inline Data] Slot 0 Column 4 Offset 0x43 Length 4000 Length (physical) 4000

000000F6AB96E3B0:   7a7a7a7a 7a7a7a7a 7a7a7a7a 7a7a7a7a 7a7a7a7a  zzzzzzzzzzzzzzzzzz
000000F6AB96E3C4:   7a7a7a7a 7a7a7a7a 7a7a7a7a 7a7a7a7a 7a7a7a7a  zzzzzzzzzzzzzzzzzz
```

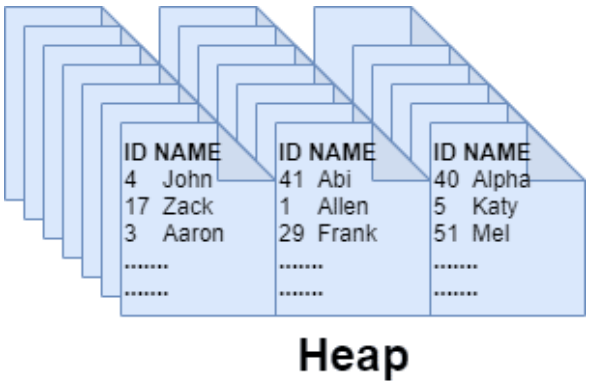
Note that while SQL Server provides the functionality to store large-size variable-length columns in a table using the concept of row-overflow and LOB pages, there is a performance trade-off. With both row-overflow and LOB pages, a single row could span multiple pages, which means many pages to scan when reading a single data row leading to poor performance.

Heap, B-Tree, and Columnstore

There are typically three ways SQL Server organizes data: Heap, B-Tree (Balanced-Tree), and Columnstore. Let's examine each of these below.

Heap

The smallest unit of storage in SQL Server is an 8 KB page. A heap is a table without a clustered index. There is no logical ordering of rows, as there is no key column on which to order. Just like the word's English definition, a heap in SQL Server is a stack of 8 KB data pages stored in a disorderly manner. When a query runs to retrieve a record from a heap, SQL Server has to do the hard work of scanning through all data pages until it finds the record, resulting in poor performance due to the full table scan. Although records are scattered across pages, there is still a mechanism that makes searching for records easier — IAM pages.

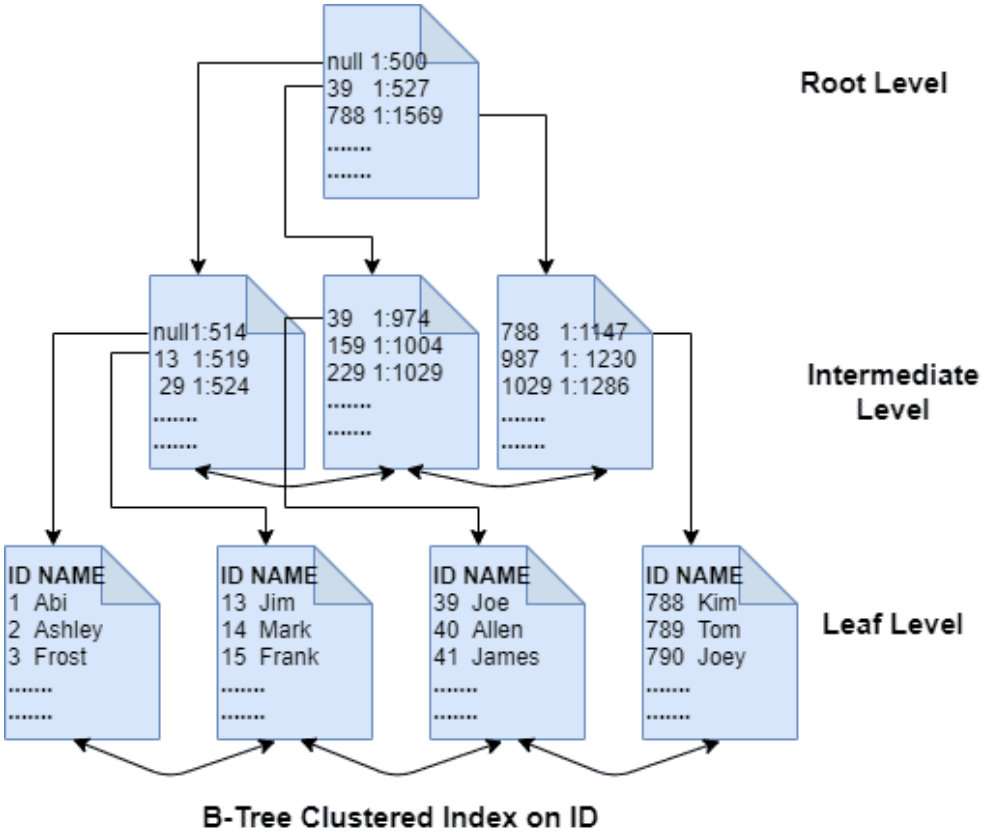


An Index Allocation Map or IAM page tracks which extents and pages belong to a specific object. Each IAM page has a range of a Global Allocation Map (GAM) interval, i.e., 4 GB or 64k extents. It links the extents and pages to their corresponding objects. In the above DBCC IND output screenshots, notice that the data pages are tied to IAM pages, as seen in the column IAMPID of the corresponding data page.

An IAM page is what SQL Server uses to track the data pages (or index pages) and extents of a table to scan and perform a search. There can be multiple IAM pages for a table. A heap can have non-clustered indexes, but when a query needs to perform RID lookups against the base heap, there could be poor performance unless a non-clustered index has all the columns to satisfy the query, and SQL Server does not have to perform lookups against the base heap table.

B-Tree

Both clustered and non-clustered indexes adhere to this structure. SQL Server sorts the data pages in the order of the index key column (s), and there are double links between pages to allow for scanning the pages sequentially and back and forth. Each index page contains the page addresses of the previous and the next page in its header, thus linking pages with each other. The structure of a B-tree comprises multiple levels. At the top level is a root page with rows pointing to the pages below it. One level down from the root is an intermediate level comprising index pages with rows pointing to the pages further below it at the next intermediate level.



Depending on the table size, there may be multiple intermediate levels. The level down from the last intermediate level is the leaf level. This level is comprised of leaf pages, which could be data pages if it's a clustered index, or index pages if it's a non-clustered index. The latter contains pointers to the base table, which may be a heap or a clustered index.

Each row on the index pages on the non-leaf levels (i.e., root and the intermediate levels) contain two critical details: the minimum key value on the page they reference and its physical address. Each row maps to a page in the next level, making it a 1 row:1 page mapping. If the index key column is narrow, there may not be a need to have multiple intermediate levels. A single intermediate level could suffice even for a table

with millions of records. Unlike heaps where data is merely scattered with no proper order, B-trees allow traversing the index tree sequentially with the help of the links between pages, making the search process much more straightforward and optimized.

Columnstore

In this structure, instead of the traditional row-based storage, SQL Server stores data in a column-wise manner. Instead of rows with all columns stored on a page, the storage structure stores values of the same column grouped. Columnstore indexes store data on a per-column as opposed to a per-row basis. Data is stored in a column-based manner across multiple row groups where each row group comprises up to 1,048,576 rows. SQL Server encodes and compresses each row group and stores the data as a LOB allocation unit, just like the LOB pages discussed above. SQL Server stores the columnstore index as LOB_DATA.

We realize the benefit of this structure in analytical workloads, where we read a vast number of rows with a few columns. Even if you retrieve only select columns, heaps and B-trees still read all the columns from the disk into memory, thus eating up memory unnecessarily. Column-based storage and batch mode execution go hand-in-hand. Batch-mode processing reduces the CPU load to a large extent. Instead of processing the data one row at a time, SQL Server processes at a batch level. Therefore, we can replace row-level operations such as aggregations with batch-level for better performance. An in-depth explanation of columnstore indexes will be covered in a future article.

Conclusion

Data pages are the most popular type of pages in SQL Server. The rows that SQL Server returns when you run a select statement are physically stored in slots on the data pages. SQL Server always prefers in-row pages first, and if that is not possible, it stores the rows on row-overflow pages as long as the column value is no larger than 8000 bytes, and if it is, SQL Server stores the data on LOB pages. Storing the nullable variable-length columns at the end of your table in the *create table* command can save 2 bytes (in the column offset array of that data row) per null value for those trailing variable-length columns. Heaps, b-trees, and columnstore are the three ways SQL Server organizes data.

Now that we understand how data is physically stored, in the following article, we will see how SQL Server processes insert, update, and delete statements against heaps and what happens internally on the data pages.