B+ Trees – How SQL Server Indexes are Stored on Disk

Introduction

SQL Server organizes indexes in a structure known as B+Tree. Many think, B+Trees are binary trees. However, that is not correct. A binary tree is a hierarchical structure organizing nodes (table rows) in a manner that allows searches to be executed extremely efficiently. On the flipside, the binary tree structure is very volatile when it comes to updates, often requiring the entire structure to be rebuilt when a single data point was changed. That makes binary tree a very poor choice when it comes to persisting data on disk.

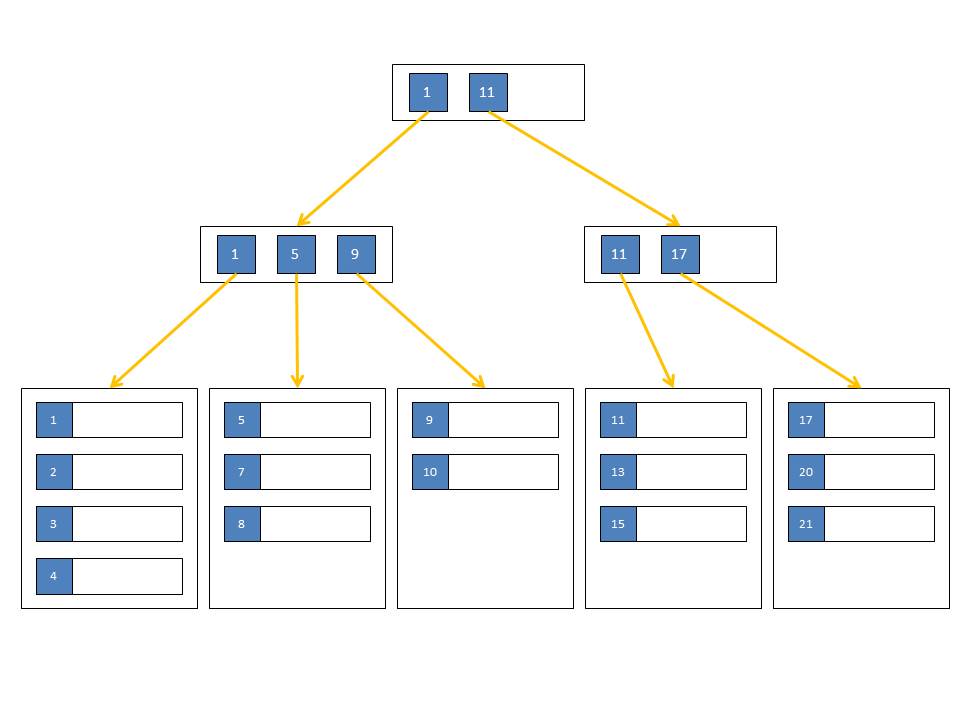
The B-Tree

In 1972, Rudolf Bayer and Ed McCreight, both working at Boeing at the time, were looking for a way to overcome some of the shortfalls of the binary tree. Their solution was the B-Tree. The main difference between a binary tree and a B-Tree is that the latter allows for more than one data point (table row) per node. B-Trees are also balanced, which means that the time it takes to execute a search within this structure is mostly independent of the value to be found.

For a long time it was unclear what the "B" in the name represented. Candidates discussed in public where "Boeing", "Bayer", "Balanced", "Bushy" and others. In 2013, the B-Tree had just turned 40, Ed McCreight revealed in an interview, that they intentionally never published an answer to this question. They were thinking about many of these options themselves at the time and decided to just leave it an open question.

The B+Tree

B-Trees are a lot more efficient than binary trees when it comes to updates, but some operations can still turn out expensive, depending on where the node that will hold the new or updated data lives in the tree. Therefore, another optimization was made to B-Trees to help with this problem. Instead of treating all nodes equal, the new structure has two types of nodes. The lowest level nodes, also called leaf nodes, hold the actual data. All other nodes including the root node only hold the key values and pointers to the next nodes. This type of tree is called a B+Tree and you can see an example below:



There are no limitations on the number of key-pointer-pairs or data rows within a node. The only limitation is that all leaf nodes have the same distance from the root node. That means that the work to seek for a particular data point is always the same, no matter what the key value is. It also keeps updates very localized in the tree. I might have to move a few rows to a new node during an update but then I probably only need to change a single parent node to integrate that new node. It is however possible for a single change to affect every level of the tree, but those changes are rare.

B+Trees in SQL Server

SQL Server stores its indexes in B+Tree format. There are a few exceptions - for example temporary hash indexes, created during a hash join operation, or column store indexes, which are not really indexes at all. However, all key-based clustered and non-clustered persisted SQL Server indexes are organized and stored as B+Trees.

Each node in such a tree is a page in SQL Server terms and you will find two page types in each index. The first type are the data pages (pages of type 1). Each leaf level node in a SQL Server B+Tree index is a single data page. The second type are intermediate index pages (pages of type 2). Each node in an index B+Tree that is not a leaf level node is a single page of type 2. Those pages contain rows just like the data pages. But in addition they contain a pointer for each row that identifies the next child page. That child page can be either of type 1 or of type 2, depending on the location in the B+Tree.

Clustered Index Structures

In SQL Server, indexes are organized as B-trees. Each page in an index B-tree is called an index node. The top node of the B-tree is called the root node. The bottom level of nodes in the index is called the leaf nodes. Any index levels between the root and the leaf nodes are collectively known as intermediate levels. In a clustered index, the leaf nodes contain the data pages of the underlying table. The root and intermediate level nodes contain index pages holding index rows. Each index row contains a key value and a pointer to either an intermediate level page in the B-tree, or a data row in the leaf level of the index. The pages in each level of the index are linked in a doubly-linked list.

Clustered indexes have one row in sys.partitions, with **index\_id** = 1 for each partition used by the index. By default, a clustered index has a single partition. When a clustered index has multiple partitions, each partition has a B-tree structure that contains the data for that specific partition. For example, if a clustered index has four partitions, there are four B-tree structures; one in each partition.

Depending on the data types in the clustered index, each clustered index structure will have one or more allocation units in which to store and manage the data for a specific partition. At a minimum, each clustered index will have one IN\_ROW\_DATA allocation unit per partition. The clustered index will also have one LOB\_DATA allocation unit per partition if it contains large object (LOB) columns. It will also have one ROW\_OVERFLOW\_DATA allocation unit per partition if it contains variable length columns that exceed the 8,060 byte row size limit.

The pages in the data chain and the rows in them are ordered on the value of the clustered index key. All inserts are made at the point where the key value in the inserted row fits in the ordering sequence among existing rows. The page collections for the B-tree are anchored by page pointers in the **sys.system\_internals\_allocation\_units** system view.

For a clustered index, the **root\_page** column in **sys.system\_internals\_allocation\_units** points to the top of the clustered index for a specific partition. SQL Server moves down the index to find the row corresponding to a clustered index key. To find a range of keys, SQL Server moves through the index to find the starting key value in the range and then scans through the data pages using the previous or next pointers. To find the first page in the chain of data pages, SQL Server follows the leftmost pointers from the root node of the index.

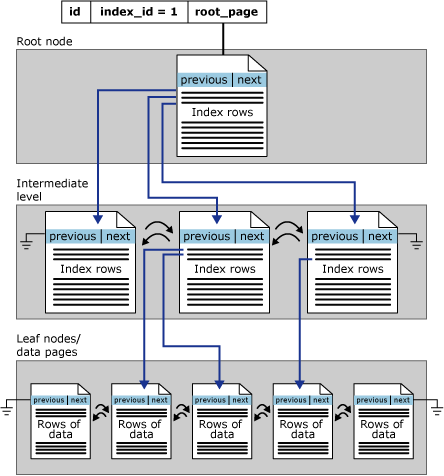


Figure shows the structure of a clustered index in a single partition.

Non clustered Index Structures

Nonclustered indexes have the same B-tree structure as clustered indexes, except for the following significant differences:

* The data rows of the underlying table are not sorted and stored in order based on their nonclustered keys.
* The leaf layer of a nonclustered index is made up of index pages instead of data pages.

Nonclustered indexes can be defined on a table or view with a clustered index or a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value.

The row locators in nonclustered index rows are either a pointer to a row or are a clustered index key for a row, as described in the following:

* If the table is a heap, which means it does not have a clustered index, the row locator is a pointer to the row. The pointer is built from the file identifier (ID), page number, and number of the row on the page. The whole pointer is known as a Row ID (RID).
* If the table has a clustered index, or the index is on an indexed view, the row locator is the clustered index key for the row. If the clustered index is not a unique index, SQL Server makes any duplicate keys unique by adding an internally generated value called a **uniqueifier**. This four-byte value is not visible to users. It is only added when required to make the clustered key unique for use in nonclustered indexes. SQL Server retrieves the data row by searching the clustered index using the clustered index key stored in the leaf row of the nonclustered index.

Nonclustered indexes have one row in sys.partitions with **index\_id**>0 for each partition used by the index. By default, a nonclustered index has a single partition. When a nonclustered index has multiple partitions, each partition has a B-tree structure that contains the index rows for that specific partition. For example, if a nonclustered index has four partitions, there are four B-tree structures, with one in each partition.

Depending on the data types in the nonclustered index, each nonclustered index structure will have one or more allocation units in which to store and manage the data for a specific partition. At a minimum, each nonclustered index will have one IN\_ROW\_DATA allocation unit per partition that stores the index B-tree pages. The nonclustered index will also have one LOB\_DATA allocation unit per partition if it contains large object (LOB) columns. The page collections for the B-tree are anchored by **root\_page** pointers in the **sys.system\_internals\_allocation\_units** system view.

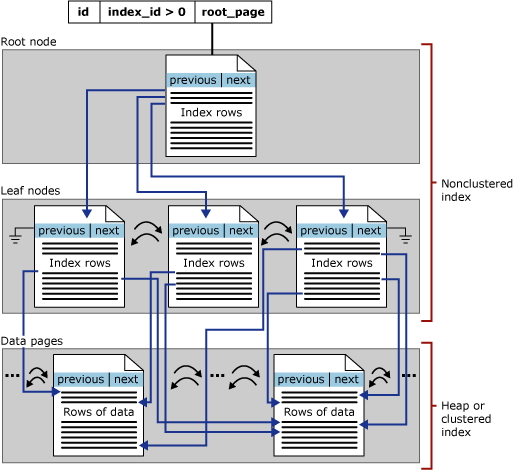


Figure shows the structure of a nonclustered index in a single partition.

**Index Fragmentation in SQL Server**

Index fragmentation is a phenomenon where the index contents are scattered. Normally the contents are in contiguous fashion which helps in fast retrieval of the underlying data. When the indexes are fragmented the data access becomes time consuming because of the scattered data that needs to be searched and read.

Fragmentation occurs as data is modified. The following are the two types of Index fragmentation:

1. Internal fragmentation
2. External fragmentation

**Internal fragmentation:**

This happens when space is available within your index page i.e. when the index pages have not been filled as full as possible. Due to internal fragmentation the index is taking up more space than it needs to. Thus when scanning the index it results in more read operations. Internal fragmentation also happens due to specifying a low value of fill factor (which determines the % of space to be filled in a leaf level page).

This is also caused by rows that are removed by DELETE statements or when pages are split and only filled to about half. Empty space on pages means there are less rows per page, which in turn means more page reads.

**External Fragmentation:**

External fragmentation occurs when the pages are not contiguous on the index. If the pages in a book are NOT ordered in a logical way (page 1, then page 2, then page 3 and so on) causing you to go back and forward to compound the information and make sense of the reading. External fragmentation happens when there are frequent UPDATES and INSERTS in a table having small amount of free space in the index page.

Since the page is already full or only has less free space left and if it is not able to accommodate the new row inserted or updated, as a result Page split happens in order to allocate the new row. Due to page split, original page will be split such that half the rows are left on the original page and the other half is moved to the new page. Mostly the new page is not contiguous to the page being split. Page split is an expensive operation and should always be avoided.

**How to determine fragmentation?**

The following query will give the fragmentation information of a particular table named person.address in adventureworks database. Please modify the query to replace the database name and table name according to your requirements.

SELECT CAST(DB\_NAME(database\_id) AS varchar(20)) AS [Database Name],  
CAST(OBJECT\_NAME(object\_id) AS varchar(20)) AS [TABLE NAME], Index\_id, Index\_type\_desc, Avg\_fragmentation\_in\_percent, Avg\_page\_space\_used\_in\_percent  
FROM sys.dm\_db\_index\_physical\_stats(DB\_ID('AdventureWorks'),OBJECT\_ID('person.address'),NULL,NULL,'Detailed')

If you wish to identify the fragmentation information for the tables in a particular database please use the below query. I am using it to find the fragmentation in Adventureworks database.

SELECT CAST(DB\_NAME(database\_id) AS varchar(20)) AS [Database Name],  
CAST(OBJECT\_NAME(object\_id) AS varchar(20)) AS [TABLE NAME], Index\_id, Index\_type\_desc, Avg\_fragmentation\_in\_percent, Avg\_page\_space\_used\_in\_percent  
FROM sys.dm\_db\_index\_physical\_stats(DB\_ID('AdventureWorks'),NULL,NULL,NULL,'Detailed')

SELECT CAST(DB\_NAME(database\_id) AS varchar(20)) AS [Database Name],

CAST(OBJECT\_NAME(object\_id) AS varchar(20)) AS [TABLE NAME], Index\_id, Index\_type\_desc, Avg\_fragmentation\_in\_percent, Avg\_page\_space\_used\_in\_percent

FROM sys.dm\_db\_index\_physical\_stats(DB\_ID('AdventureWorks'),NULL,NULL,NULL,'Detailed')

Take a look at the output of the following columns in the above query:

**Avg\_fragmentation\_in\_percent:**

If the value is >5 and <30 you need to REORGANIZE the index using ALTER index REORGANIZE command. If the value is >30 you need to REBUILD the indexes using ALTER index REBUILD command.

**Avg\_page\_space\_used\_in\_percent:**

This value represents the amount of page space used in an index. If the value is <75% and >60% we need to REORGANIZE the indexes else REBUILD the indexes.

**Defragmenting the Indexes:**

We need to either use ALTER INDEX REBUILD or ALTER INDEX REORGANIZE commands to remove the fragmentation from the indexes. Generally its advisable to schedule a job to do this operations in OFF-Production hours as they consume lots of resources.

**Best Practices on Indexing**

* Periodically, run the Index Wizard or Database Engine Tuning Advisor against current Profiler traces to identify potentially missing indexes.
* Remove indexes that are never used.
* Don’t accidentally create redundant indexes.
* As a rule of thumb, every table should have at least a clustered index. Generally, but not always, the clustered index should be on a column that monotonically increases — such as an identity column, or some other column where the value is increasing — and is unique. In many cases, the primary key is the ideal column for a clustered index.
* Since you can only create one clustered index per table, take extra time to carefully consider how it will be used. Consider the type of queries that will be used against the table, and make aneducated guess as to which query (the most common one run against the table, perhaps) is the most critical, and if this query will benefit from having a clustered index.
* If a column in a table is not at least 95% unique, then most likely the query optimizer will not use a non-clustered index based on that column. Because of this, you generally don’t want to addnon-clustered indexes to columns that aren’t at least 95% unique.
* Keep the “width” of your indexes as narrow as possible. This reduces the size of the index and reduces the number of disk I/O reads required to read the index, boosting performance.
* If possible, avoid adding a clustered index to a GUID column (uniqueidentifier data type). GUIDs take up 16-bytes of storage, more than an Identify column,which makes the index larger, whichincreases I/O reads, which can hurt performance.
* Indexes should be considered on all columns that are frequently accessed by the JOIN, WHERE, ORDER BY, GROUP BY, TOP, and DISTINCT clauses.
* Don’t automatically add indexes on a table because it seems like the right thing to do. Only add indexes if you know that they will be used by the queries run against the table.
* When creating indexes, try to make them unique indexes if at all possible. SQL Server can often search through a unique index faster than a non-unique index because in a unique index, each
* Row is unique, and once the needed record is found, SQL Server doesn’t have to look any further.
* If you perform regular joins between two or more tables in your queries, performance will be optimized if each of the joined columns has appropriate indexes.
* Don’t automatically accept the default value of 100 for the fill factor for your indexes. It may or may not best meet your needs. A high fill factor is good for seldom changed data, but highly
* Modified data needs a lower fill factor to reduce page splitting.
* Don’t over index your OLTP tables, as every index you add increases the time it takes to perform INSERTS, UPDATES, and DELETES. There is a fine line between having the ideal number of
* Indexes (for SELECTs) and the ideal number to minimize the overhead that occurs with indexes during data modifications.
* If you know that your application will be performing the same query over and over on the same table, consider creating a non-clustered covering index on the table. A covering index, which is a form of a composite index, includes all of the columns referenced in SELECT, JOIN, and WHERE clauses of a query. Because of this, the index contains the data you are looking for and SQL Server doesn’t have to look up the actual data in the table, reducing logical and/or physical I/O, and boosting performance.