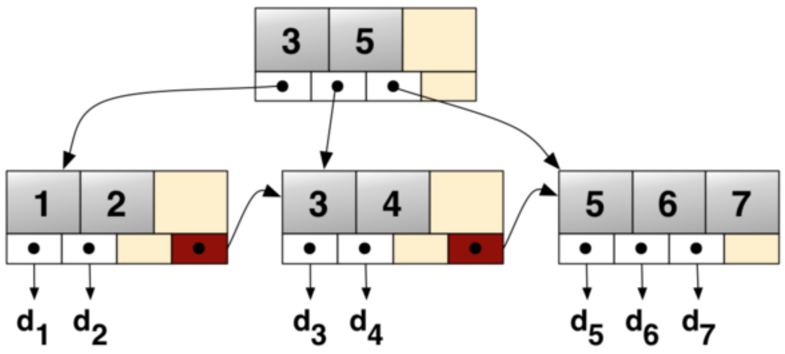
WIKIPEDIA

B+ tree

A **B+ tree** is an N-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves.[1] The root may be either a leaf or a node with two or more children.[2]



A simple B+ tree example linking the keys 1–7 to data values d_1-d_7. The linked list (red) allows rapid in-order traversal. This particular tree's branching factor is b=4.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node,[1] typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

The ReiserFS, NSS, XFS, JFS, ReFS, and BFS filesystems all use this type of tree for metadata indexing; BFS also uses B+ trees for storing directories. NTFS uses B+ trees for directory and security-related metadata indexing. EXT4 uses extent trees (a modified B+ tree data structure) for file extent indexing.[3] Relational database management systems such as IBM DB2,[4] Informix,[4] Microsoft SQL Server,[4] Oracle 8,[4] Sybase ASE,[4] and SQLite[5] support this type of tree for table indices. Key–value database management systems such as CouchDB[6] and Tokyo Cabinet[7] support this type of tree for data access.

Contents

Overview

The order, or branching factor, b of a B+ tree measures the capacity of nodes (i.e., the number of children nodes) for internal nodes in the tree. The actual number of children for a node, referred to here as m, is constrained for internal nodes so that $\lceil b/2 \rceil \leq m \leq b$. The root is an exception: it is allowed to have as few as two children.[1] For example, if the order of a B+ tree is 7, each internal node (except for the root) may have between 4 and 7 children; the root may have between 2 and 7. Leaf nodes have no children, but are constrained so that the number of keys must be at least $\lceil b/2 \rceil - 1$ and at most $b - 1$. In the situation where a B+ tree is nearly empty, it only contains one node, which is a leaf node. (The root is also the single leaf, in this case.) This node is permitted to have as little as one key if necessary and at most $b - 1$.

Node Type	Children Type	Min Number of Children	Max Number of Children	Example $b = 7$	Example $b = 100$
Root Node (when it is the only node in the tree)	Records	1	$b - 1$	1–6	1–99
Root Node	Internal Nodes or Leaf Nodes	2	b	2–7	2–100
Internal Node	Internal Nodes or Leaf Nodes	$\lceil b/2 \rceil$	b	4–7	50–100
Leaf Node	Records	$\lceil b/2 \rceil - 1$	$b - 1$	3–6	49–99

Algorithms

Search

The root of a B+ Tree represents the whole range of values in the tree, where every internal node is a subinterval.

We are looking for a value k in the B+ Tree. Starting from the root, we are looking for the leaf which may contain the value k . At each node, we figure out which internal pointer we should follow. An internal B+ Tree node has at most $d \le b$ children, where every one of them represents a different sub-interval. We select the corresponding node by searching on the key values of the node.

```
Function: search (k)
  return tree_search (k, root);

Function: tree_search (k, node)
  if node is a leaf then
    return node;
  switch k do
  case k < k_0
    return tree_search(k, p_0);
  case k_i ≤ k < k_{i+1}
    return tree_search(k, p_{i+1});
  case k_d ≤ k
    return tree_search(k, p_{d+1});
```

This pseudocode assumes that no duplicates are allowed.

Prefix key compression

- It is important to increase fan-out, as this allows to direct searches to the leaf level more efficiently.

- Index Entries are only to 'direct traffic', thus we can compress them.

Insertion

- Perform a search to determine what bucket the new record should go into.
- If the bucket is not full (at most $b-1$ entries after the insertion), add the record.
- Otherwise, *before* inserting the new record
  - split the bucket.
    - original node has $\lceil(L+1)/2\rceil$ items
    - new node has $\lfloor(L+1)/2\rfloor$ items
  - Move $\lceil(L+1)/2\rceil$-th key to the parent, and insert the new node to the parent.
  - Repeat until a parent is found that need not split.
- If the root splits, treat it as if it has an empty parent and split as outline above.

B-trees grow at the root and not at the leaves.[1]

Deletion

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If *L* is at least half-full, done!
  - If *L* has fewer entries than it should,
    - If sibling (adjacent node with same parent as *L*) is more than half-full, re-distribute, borrowing an entry from it.
    - Otherwise, sibling is exactly half-full, so we can merge *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Bulk-loading

Given a collection of data records, we want to create a B+ tree index on some key field. One approach is to insert each record into an empty tree. However, it is quite expensive, because each entry requires us to start from the root and go down to the appropriate leaf page. An efficient alternative is to use bulk-loading.

- The first step is to sort the data entries according to a search key in ascending order.
- We allocate an empty page to serve as the root, and insert a pointer to the first page of entries into it.
- When the root is full, we split the root, and create a new root page.
- Keep inserting entries to the right most index page just above the leaf level, until all entries are indexed.

Note :

- when the right-most index page above the leaf level fills up, it is split;
- this action may, in turn, cause a split of the right-most index page on step closer to the root;
- splits only occur on the right-most path from the root to the leaf level.

Characteristics

For a b-order B+ tree with h levels of index:

- The maximum number of records stored is $n_{\max} = b^h - b^{h-1}$

- The minimum number of records stored is $n_{\min} = 2\left\lceil \frac{b}{2} \right\rceil^{h-1} - 2\left\lceil \frac{b}{2} \right\rceil^{h-2}$
- The minimum number of keys is $n_{\mathrm{kmin}} = 2\left\lceil \frac{b}{2} \right\rceil^{h-1} - 1$
- The maximum number of keys is $n_{\mathrm{kmax}} = b^h - 1$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations
- Finding a record requires $O(\log_b n)$ operations
- Removing a (previously located) record requires $O(\log_b n)$ operations
- Performing a range query with *k* elements occurring within the range requires $O(\log_b n + k)$ operations

Implementation

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B+tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B+tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B+tree to actually provide an efficient structure for housing the data itself (this is described in[4]:238 as index structure "Alternative 1").

If a storage system has a block size of B bytes, and the keys to be stored have a size of k, arguably the most efficient B+ tree is one where $b = \frac{B}{k} - 1$. Although theoretically the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case a reasonable choice for block size would be the size of processor's cache line.

Space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers. For string keys, space can be saved by using the following technique: Normally the i-th entry of an internal block contains the first key of block $i + 1$. Instead of storing the full key, we could store the shortest prefix of the first key of block $i + 1$ that is strictly greater (in lexicographic order) than last key of block *i*. There is also a simple way to compress pointers: if we suppose that some consecutive blocks $i, i + 1, \ldots i + k$ are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of compression techniques is that the number of stored elements may vary considerably from a block to another depending on how well the elements are compressed inside each block.

History

The B tree was first described in the paper *Organization and Maintenance of Large Ordered Indices*. *Acta Informatica 1*: 173–189 (1972) by Rudolf Bayer and Edward M. McCreight. There is no single paper introducing the B+ tree concept. Instead, the notion of maintaining all data in leaf nodes is repeatedly brought up as an interesting variant. An early survey of B trees also covering B+ trees is Douglas Comer.[8] Comer notes that the B+ tree was used in IBM's VSAM data access software and he refers to an IBM published article from 1973.

See also

- Binary search tree
- B-tree
- Divide and conquer algorithm

References

1. Navathe, Ramez Elmasri, Shamkant B. (2010). *Fundamentals of database systems* (6th ed.). Upper Saddle River, N.J.: Pearson Education. pp. 652–660. ISBN 9780136086208.
2. http://www.seanster.com/BplusTree/BplusTree.html
3. Giampaolo, Dominic (1999). *Practical File System Design with the Be File System* (http://www.nobius.org/~dbg/practical-file-system-design.pdf) (PDF). Morgan Kaufmann. ISBN 1-55860-497-9.
4. Ramakrishnan Raghu, Gehrke Johannes – Database Management Systems, McGraw-Hill Higher Education (2000), 2nd edition (en) page 267
5. SQLite Version 3 Overview (http://sqlite.org/version3.html)
6. CouchDB Guide (see note after 3rd paragraph) (http://guide.couchdb.org/draft/btree.html)
7. Tokyo Cabinet reference (http://1978th.net/tokyocabinet/) Archived (https://web.archive.org/web/20090912082150/http://1978th.net/tokyocabinet/) September 12, 2009, at the Wayback Machine.
8. "The Ubiquitous B-Tree (http://doi.acm.org/10.1145/356770.356776)", ACM Computing Surveys 11(2): 121–137 (1979).

External links

- B+ tree in Python, used to implement a list (https://pypi.python.org/pypi/blist)
- Dr. Monge's B+ Tree index notes (http://www.cecs.csulb.edu/%7emonge/classes/share/B+TreeIndexes.html)
- Evaluating the performance of CSB+-trees on Mutithreaded Architectures (http://blogs.ubc.ca/lrashid/files/2011/01/CCECE07.pdf)
- Effect of node size on the performance of cache conscious B+-trees (http://www.cs.wisc.edu/~jignesh/publ/cci.pdf)
- Fractal Prefetching B+-trees (http://www.pittsburgh.intel-research.net/people/gibbons/papers/fpbptrees.pdf)
- Towards pB+-trees in the field: implementations Choices and performance (http://leo.saclay.inria.fr/events/EXPDB2006/PAPERS/Jonsson.pdf)
- Cache-Conscious Index Structures for Main-Memory Databases (https://oa.doria.fi/bitstream/handle/10024/2906/cachecon.pdf?sequence=1)
- Cache Oblivious B(+)-trees (http://supertech.csail.mit.edu/cacheObliviousBTree.html)
- The Power of B-Trees: CouchDB B+ Tree Implementation (http://books.couchdb.org/relax/appendix/btrees)
- B+ Tree Visualization (http://www.cs.usfca.edu/~galles/visualization/BPlusTree.html)
- B +-trees by Kerttu Pollari-Malmi (https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf)
- http://CSE 326: Data Structures B-Trees and B+ Trees (https://courses.cs.washington.edu/courses/cse326/08sp/lectures/11-b-trees.pdf)