

B-trees

A storage container for massive
key space

A balanced tree

- The b-tree data structure is container system somewhat like a binary tree (although the “b” stands for balanced and the tree is not binary).
- The API to a b-tree is usually similar to other key-to-data systems such as a hash table, dictionary, or associative array.
- B-trees are used, for example, for file system indexing and database indexing.
- The performance gains for a b-tree rely on having a block based memory system (such as disk sectors or cache lines) wherein the cost to get another block far outweighs the cost of doing multiple tests on data (keys) in the currently accessible block.

Example Containers

- All of these storage containers can be used for key to data lookup. Which you choose to use should depend on the systems and data involved.
 - Linked lists (single and double)
 - Hash table, Dictionary, Associative Array, Map (stl)
 - Binary tree, Red-Black tree
 - Vector (stl)
 - Queue, Stack, Deque
 - B-tree
 - Priority Queue
 - Trie

Features

- **Fast**
(relative to other containers) for large key sets
- **Self balancing**
(leaf nodes are always at the same depth (or level) within the tree)
- **There is less depth**
(search steps) relative to other tree containers
- **There is a negative: Nodes may be up to half empty**
(unless you make an exception for read-only b-trees)

Common API

- `Insert(key, data)`
- `Remove(key)`
- `Has(key)`
- `ForEach(key_data_function_ptr)` or iterator
- `Open(file_name)`
- `Close(file_name)`

Example Uses

- **Apple's HFS**

Their (Hierarchical File System) uses a b-tree for file and directory information

A second b-tree is used for file extents (fragmentation) information (past the first three extents).

- **MySQL offers a b-tree option for indexing (among other options).**
- **I wrote a game save file format that uses a b-tree that is used in several games.**

Performance

- **A bad use of a b-tree:**

When you can fit your entire key set into fast, homogeneous access time memory

- **A good use of a b-tree:**

When your key set is potentially large enough that it cannot fit entirely in your fastest memory and the performance difference between accessing nodes from slower storage outweighs the cost of doing multiple key comparisons.

It's a database

- A b-tree can be considered (and it is) a simple database.
 - Okay it's not a relational database right-off and doesn't in itself represent a transactional database.
 - It's rather straightforward to use a b-tree to build a hierarchical database.

Values (data)

- The common use of a b-tree is to store data that can be retrieved later by providing the b-tree with a (unique) key that matches a key already in the b-tree
- In some cases the key may also be considered the data.
- In most cases there is data that is separate from the key (aka a value in a “key to value” database).

Inline, leaf, or external data

- **Inline:**
data is stored in the same nodes as the keys (inline with the keys)
- **Leaf:**
Leaf data b-trees use “index nodes” to store only keys and “leaf nodes” (or data nodes) to store keys and data.
- **External:**
Another option is to store the data in a separate location and use references to the data as the value (which may be stored inline, for example).

Implementing

- **B-trees are node based**

The size of a node (or page) can have a big impact on the performance of your b-tree.

- **Header**

To be complete, a b-tree will usually include a header that contains things like:

Reference to the root node

Reference to the start of the free list

Nodes

- The size of the Nodes will play a part in the number of keys each node can store
- The number of downlinks from a node will be the number of keys stored there plus one (i.e. the order-M of the node)
- If the keys are fixed-length you can work out exactly how many keys will fit within a node
- Variable length keys are also possible, but you may want to adjust what you consider a half-full node
- In either case, being able to fit more than two keys into a node is rather fundamental for being a b-tree.
- There is no upper limit on the number of keys per node you can design your b-tree to use. Four keys to several hundred keys per node seems reasonable.

Node size

- **Page or sector boundaries:**

It can help if you choose a Node size that exactly matches the cluster size of the disk you are using or the page size of operating system

- **Alignment:**

Once you have chosen a good node size, it's also smart to align the nodes by multiples of the node size (e.g. if you have 4096 byte nodes then having the address of each node be an exact multiple of 4096 is a good idea).

Alignment

- **Start nodes at a multiple of the node size:**

Once you have chosen a good node size, it's also smart to align the nodes by multiples of the node size (e.g. if you have 4096 byte nodes then having the address of each node be an exact multiple of 4096 is a good idea).

- Files will already be allocated at a multiple of the cluster size.
- Ram alignment can be achieved with special allocators (if provided) or by over-allocating (a larger buffer) and then starting your b-tree at the next multiple of the node size
- **Header size tip:**

If start with an aligned piece of storage and you allocate space for an entire node (or multiple nodes) for your header then the rest of your nodes in the same buffer or file should remain aligned.

Insert

- New keys (and data) are inserted in sort order:

The API for a b-tree will generally not allow you to append, prepend, or insert at a specific location in the tree. You simply give it the key and data and allow it to store it for you.

Inserting to an empty b-tree

- A b-tree begins as an empty tree with no nodes, only a header
- When a key is inserted to an empty b-tree a node is created. This node is now the root.
- After more inserts, when the root is full and yet another insert comes in, the root is divided in half with the left half going into the left child node and the right half going in the right child node and the center key moving to a new root node.

Inserting to a full b-tree

- As more keys are inserted more nodes are allowed to fill up.
- Each time a key is added to a full node, it is divided in two and the center key is moved up to the parent.
- A b-tree grows in depth by adding a new root node to the top of the tree (not by adding leaf nodes to the bottom).
- By their nature, b-trees tend to grow very wide and not very tall (i.e. deep).

Getting data from a b-tree

- To retrieve data from a b-tree you need only provide the key.

This is where the b-tree is finally being really used: you know a key, but don't know the data – you give the key to the b-tree and it gives you the data (or lets you know the key was not found in the b-tree).

- Retrieving data from the tree does not normally change the tree in any way

(you might change the last access time on the file or you might change which nodes are cached, but I'm not considering those changes a change to the b-tree).

Removing keys from a b-tree

- Removing keys (and data) from a b-tree requires only the key to be provided.
- If the node containing the key is at least half full after the key is removed and the delete is finished.
- If the node is less than half full after removing the key then that node (unless it's the root node) will need to get a key from another node.
- The b-tree will first try to get a key from the left sibling and failing that it will go to the right sibling
- If the sibling node has a key to spare (and still be at least half full), the b-tree will try to arrange a trade between the sibling node, the current node, and the parent node.
- The nearest key from the sibling node will go to the parent and the parent key from the parent will go to the current node.

Removing keys from a b-tree (cont.)

- If both siblings don't have a key to spare (they're already at half full), then the current node will be combined with one of the parent's keys and the keys from the sibling (which is then added to the free node list). One of the end keys from that set will be replace the key the parent donated (so the parent has the same number of keys, but may have a different key).
- If there is no sibling to combine with the parent will give up a key for the child, wherein the parent may start the process themselves (if the parent is now less than half full).

Removing the last key from the root

- As a special case, when we remove the last key from the root node, we need to do something to get a valid root node again.
- Either the of the next lower or next higher keys would make a good root key.

For example, we can take one step the left node from the root and then step to each rightmost node until we get to a leaf node. The rightmost key from that node was (in sort order) logically next to the removed root key, so it can be moved from that leaf node to the root node as the new root key.

Ta da

- And such is the life and times of b-tree nodes.
- Now on to b-tree variations and examples

Variations

- Index and Data nodes
- How keys are stored within a node
- Keys and references with data stored separately
- References with inline storage of small data

Index and Data nodes

- Some systems store all of their keys and data in leaf nodes
i.e. in a long sorted list
- These systems use index nodes that store only keys and references to other nodes
- Advantages: If your values are not tiny compared to the keys, this moves all that data to the leaves which gets more keys-per-node in the index nodes (which is faster when searching a large key set).
- Disadvantages: If your keys are large compared to your values then this system will waste time and space redundantly storing keys.

HFS Example

- The HFS (file system) used by Apple uses an index/data node system.
- Another interesting thing is that they included back and forward links between the leaf nodes
Once you'd found, for example, the start of a directory, you could race along and list all the files in the directory by walking across the leaf nodes.
- HFS was hierarchical in that there were markers within the list of files that delimited where directories (folders) began and ended. Sub-directories within a directory had information on the key for the listing of that directory and so on.
- HFS was also interesting because any directory in the system can be accessed as if it were the root directory as long as you knew the key for that sub-directory.
So you could walk a file path once, keep the key to a directory, and then go directly to that directory quickly.
- Another fun thing about HFS, is that a directory could be moved within the file hierarchy and, as long as you had the directory key, you could still access that directory without knowing the path to it.

How keys are stored within a node

- Keys are often stored in some sort of order within the node
- The method of storage varies between different b-tree implementations
- Common options include
 - A sorted array
 - An unsorted pack of keys with a sorted reference array also stored within the node
 - A binary tree within the node

Keys and references with data stored separately

- The data need not be stored in the b-tree
- The data could be represented in another file or collection of files
- The “real” values in the b-tree would be references to the data stored elsewhere
- For example, it is rather common for a database to have its data stored in separate files from its indices (e.g. b-trees)

References with inline storage of small data

- A variation on external data storage is to inline small data items with the keys in the place that you would otherwise store the reference
- E.g. if the reference were normally eight bytes, you could store data of eight bytes or less inline.

If you want to know more

- Hmm, google?
- Or ask me.

Dave Schuyler