B+ Tree Access Method

Yanming Cao and Celena Tanguay

May 13, 1995

INTRODUCTION

Our group was responsible for implementing the B+ Tree access method for the multi-user minirel project. Our component allows a user to create (and destroy) a B-Tree index on a specified database file and key. Once this index is built, a user can scan for records having a specified key, or for all records having a key within a specified range of keys. The user can delete the current record during a scan, and they can insert and delete records at any time.

The B-Tree component interacts with components from the following groups: buffer management, log management, lock management, recovery management, and OS infrastructure. These interactions are required for allocating and freeing pages, pinning and unpinning pages, writing log records, and locking pages.

EXTERNAL INTERFACE

Our external interface consists of one file, **btfile.h**. The following is a list of classes and the functions they provide:

BTreeFile

The **BTreeFile** class is a type of **IndexFile**, which is a virtual base class for all of the access methods. The following features are public:

BTreeFile There are two types of constructors. The first will simply open the specified file, as long as it exists, and simply requires a file name. The other will create a B-Tree file, given a name, the key type, and the key size. The key size should be consistent with the type, i.e., if key type is integer, the key size should be sizeof(int). Also the header page is IS-locked when the index is open.

BTreeFile The destructor function unlocks the lock on the header page.

destroyFile This function will destroy the entire index file (i.e. delete all internal and leaf pages and the header page).

insert This function, given a ⟨**key, recid**⟩ pair, will insert it into the B-Tree. A user will be allowed to insert up to an entire page of records with duplicate keys. However, once that page is full, any further inserts of records having that key will be rejected. Also, duplicate ⟨**key, recid**⟩ pairs, will not be inserted into the tree.

Delete This function, given a ⟨**key, recid**⟩ pair, will delete it from the B-Tree. The **recid** must be specified, as duplicate keys are supported.

new_scan This function creates a new index file scan on the B-Tree. A low and a high key must be provided, although either or both can be null. The following types of scans are supported:

- **entire index:** lo_key=NULL & hi_key=NULL
- **min to hi_key:** lo_key=NULL & hi_key!=NULL
- **lo_key to max:** lo_key!=NULL & hi_key=NULL
- **exact match:** lo_key!=NULL & hi_key!=NULL & lo_key=hi_key
- **range:** lo_key!=NULL & hi_key!=NULL & lo_key<hi_key

BTreeFileScan

The **BTreeFile** class is a type of **IndexFileScan**, which is a virtual base class for all of the Access Methods' scan operations. The following features are public:

BTreeFileScan The constructor, called by **new_scan**, simply creates the index file scan specified.

BTreeFileScan The destructor function of scan.

get_next This function returns the **recid** of the next record in the scan. If it is the first time it has been called since opening the scan, it will locate the first record of the scan by searching the tree from the root page, and return its **recid**. Otherwise, it simply traverses the leaf pages until the scan is complete.

delete_current This function deletes the current record of the scan.

INTERNAL DESIGN

Tree Design

A B-Tree index file has a header page, which contains the following information of this index:

- the name of this index file
- the page no of the root of the tree

- the key type on which the index is built
- the size of the key for the index

A header page is created when the B-Tree index is created. From the root page, we can simply follow pointers to the other pages in the B-Tree.

We have two types of pages in the tree itself: internal and leaf pages. Both of these consist of the following information:

- node type (leaf of internal)
- leftmost child
- right sibling
- left sibling
- parent

The global page size is 1024. However, the **lsn**, which the B-Tree will not need to access, is stored in the last 4 bytes of each page. Thus, the space for a B-Tree page is 1020. We do not support variable length records, so the number of records that can fit on a page is fixed. Thus, we can simply use the size of a B-Tree page (1020) minus the space required for the page overhead listed above to compute the following values when the B-tree is initially created:

- the size of a record on an internal page
- the size of a record on a leaf page
- the capacity of an internal page
- the capacity of a leaf page

These four are all members of a BTreeFile object.

Immediately following the page header information is a slot table. This simply contains the offsets for each record in ascending order. The array is initialized to the negative of the offsets, which specifies that all slots are empty. When a slot becomes empty after a deletion, we simply set the offset in that position of the slot array to the negative of its value and compact the slot table, moving the newly freed slot to the end of the array. Since the slot table is compacted on deletions, traversing the array until a negative value is reached will give us the keys on the page in ascending order. The remainder of the page consists of the actual ⟨**key, ptr**⟩ records.

As we traverse the tree, we update the parent pointer of each page we come to. This is to ensure that the value is always correct, since it may have become obsolete during a previous page split. This value is then used when splitting to insert an entry into the page's correct parent. Thus, to insure integrity of the parent information, we must start from the root and traverse to the correct leaf when we want to insert (i.e. a user cannot insert during a scan).

The left sibling pointer is required for merging, which we have not implemented in our final version. It could also be used for descending scans of the tree, if they were desired for some reason. Although we have no direct use for it in our final code, we chose to leave it in to simplify merge implementation.

Duplicate keys are supported by our B-Tree implementation. We ensure that all records containing the same key are stored on one leaf page. However, once a page is full (with the same key) further insertions of a record with that key are rejected. When inserting a record with a key already in the tree, we check that it is not a duplicate record. If so, we do not insert the duplicate.

The leftmost child pointer is not needed for the leaf pages, and is thus null. However, having a common page type is worth the cost of storing this extra pointer for the leaf pages. All of our page structures can be cast uniformally as Page, thus simplifying the interface with the buffer management, logging and recovery code. For example, the header page of a B-Tree is different from an actual B-Tree page. However, we read them in the same manner:

```
BufMgr→pinPage(headpageno,(Page*&)headerpage);
BufMgr→pinPage(btpageno,(Page*&)btpage);
```

Scanning

We support five types of scanning, as listed above in the external interface section. These include: AllScan, MinRangeScan, MaxRangeScan, MinMaxRangeScan, and ExactMatch. Exact match will return all records containing the specified key. Also, as stated above, we do support the delete-current operation.

AllScan This scans the entire B-Tree index.

MinRangeScan This scan only requires that a low key be specified. It will find the first record containing the low key and continue the search until there are no records remaining.

MaxRangeScan This scan only requires that a high key be specified. It will go to the first record in the B-Tree and continue the search until it reaches the first record containing the high key, or a value greater than the high key.

ExactMatch This scan occurs when the same value is specified for the low and high keys. It will find the first occurrence of that value in the B-Tree, if it exists, and continue the scan until it finds a key value greater than that key. Thus, it will return all records containing the specified key.

MinMaxRangeScan This scan occurs when both the high and low keys are specified, but are not equal. It will traverse the B-Tree to find the low key and then continue through the leaf nodes until it reaches the high key.

Logging

Logging is handled for two separate cases: when a record is deleted or inserted without causing a split, and when an insert causes a split. The first case is fairly simple. If no split occurs, there are two ranges of the page that are affected: the slot table and the actual slot where the record has been inserted or deleted. Thus, we enter a WriteUpdateLog record for both of these ranges of the page. However, when a split has occurred, things become slightly more complicated. We have now affected information in the header information, the entire slot table (since it is compacted on deletion), and various record slots. It would be a tedious task to locate all of these affected regions and enter a WriteUpdateLog record for each of them. Thus, we simply enter the log record, including the before and after images of the entire page. We need to do this for three pages: the page that was split, the parent of the split page, and the new page created. (The new page created simply gets logged with a new image, since it did not exist prior to this.) We have defined a global variable, **BTREE_LOG**, which allows us to alternately activate and deactivate logging procedures. When **BTREE_LOG** is set to 1, all logging code will be executed. When it is set to 0, all logging code will be ignored.

Locking

When a transaction requires access to a B+ tree index, an IS lock is required on the header page. This lock is not release until the transaction commits. Different locking protocols are used for insert and delete. When inserting a record,

1. X-lock the root
2. get the root and make it current
3. while current != leaf do
    4. X-lock desired child
    5. get this child and make it current
    6. if current is safe then release all locks on the ancestors of current
7. insert the record to current and propagate changes up the tree
8. unlock when done

Safe means that current node is not full.

Since delete is much simplified in our implementation, the locking protocol for delete is very simple.

1. S-lock the root
2. get the root and make it current
3. while current != leaf do
    4. S-lock desired child
    5. unlock the current
    6. get the child and make it current

7. look for the record to be deleted. If not found, unlock current and done. If found, upgrade R-lock to X-lock and delete the record, unlock when done.

For scans, the locking protocol is similar to that of delete. Only upgrade the R-lock to X-lock of the current node when one wants to do delete_current() and the upgraded locks are not released until commit time.

We have defined a global variable, **BTREE_LOCK**, which allows us to alternate between single- and multi-user mode. When **BTREE_LOCK** is set to 1, all locking code will be executed. When it is set to 0, all locking code will be ignored.

Algorithms

All of the major functions are described above. These include:

- Creating a B-Tree index
- Deleting a B-Tree index
- Inserting a ⟨**key, recid**⟩ pair into a B-Tree
- Deleting a ⟨**key, recid**⟩ pair from a B-Tree
- Scanning a B-Tree (any of the five types)
- Deleting the current record in a scan

Another important function used by our component is the page-splitting function. This function ensures that all records with the same key reside on the same page after the split is completed. We have also written numerous simple functions to make it easier to deal with the B-Tree. These include functions to compute the record size of a page, determine the capacity of a page, get the number of records actually on a page, determine if a page is full, retrieve the record at the specified slot on a page, etc. (These are all commented and easy to find in the code.) The insertion and deletion functions compact the slot table after each execution.

Some other functions used soley by the B-Tree code include:

compare This is a general function for comparing two keys of any of the three basic types (int, float, string). It simply returns a -1 if key1<key2, 0 if key1=key2, and 1 if key1>key2. This makes scanning and insertion much simpler, as we do not have to worry about the key type while traversing the tree.

num_last This function returns the number of records that have the same key as the last record in a B-Tree page. It is used when splitting to help ensure that records with the same key value are all located on the same page after the split. It also helps ensure that we do not get more than one full page of duplicate keys.

PrintPage This function was written for debugging and testing purposes, but can be used so that a user may arbitrarily view information about the tree. It prints the page header information and a list of the keys and their associated B-Tree page pointers (if an internal page) or record ids (if a leaf page).

get_first This function is used during scanning and is called when the user executes **get_next** for the first time. Given the low key, it will find the leaf page and position on that page of the first occurrence of the key value. After this is found, there is no need to traverse the tree, other than simply stepping through the leaf pages.

ACCOMPLISHMENTS

The first thing we had to do was to set up the pages for the B-Tree. We simply use the global page type, and add a few features of our own. We needed to store the page header information, and to write functions to access and change this information easily (especially the slot table).

The main part of the project was the actual B-Tree construction. We had to write functions to insert into and delete from the tree, including splitting full pages on insertion. Once this was done, the scanning procedures were fairly straight-forward.

We had originally planned on merging pages once they became less than half full, which is why we have the left sibling pointers on each page. However, we ran into problems with this. The major problem came up while deleting the current record in a scan. When doing a range scan, we only traverse the tree to get to the first leaf page in the scan. From there, we simply follow the right sibling pointers until we are at the end of the range for the scan (or until there are no more records in the tree). A user can arbitrarily delete a record at any point during the scan. Thus, we do not necessarily know the parent of the current node. (We do maintain parent pointers but, as mentioned above, we can only guarantee their correctness when traversing from the root to the leaf.) We could get around this problem by ensuring that all parent pointers that may be affected by a split are updated during the split. However, we do not feel that it is reasonable for this project, given the overhead of these checks during every split.

For the same reasons as with merging pages, we also ran into problems when attempting to delete empty pages. Thus, the B-Tree may have some empty pages after enough deletes are performed.

TESTING

As we had not finished the multi-user version of B+ tree as of Thursday May 11, we only tested the single-user version. Two types of keys are tested, namely, integer and character string. There are four tests:

Test1 In test1, a B+ tree index is created and 4000 <key, rid> pairs are inserted. The key type is integer and keys are generated randomly from 0 to 3999 (hence there are some duplicate keys). After insert, some randomly chosen records are deleted. Then all 5 types of scans are tested: AllScan, MinRangeScan, MaxRangeScan, ExactMatch and MinMaxRangeScan. During the MinMaxRangeScan, the record that is currently scanned is deleted (testing delete_current()). For every scan test, the number of records scanned is shown. At any stage of the test, one can print out the header page for the B+ tree, the root page content or content of any page given the page no.

Test2 In test2, the B+ tree index created in test1 is opened and various unusual scans are tested, e.g., in MinMaxRangeScan the low key value is greater than the high key value etc. Lastly, destroyFile() is tested. The list of page numbers that are freed is shown.

Test3 In test3, a B+ tree index is created and 2590 <key, rid> pairs are inserted. The key type is string. The key input file is got from the user names in this department and keys are in ascendant order. The delete, all kind of scans and destroy are tested.

Test4 Test4 is the same as test3 except that the keys are inserted descendantly.

RETROSPECTIVE

There seem to be two areas of our component that could be improved upon. As of now, our code will only allow the tree to have up to one full page of records with duplicate keys. This could be modified so that, when we do have a full page of duplicates, some alternative method (such as page chaining) is provided to enable a user to insert additional records with the same key. Also, as stated above, merging of sparse pages and deletion of empty pages in the tree have not been implemented due to some problems we ran into. Adding these features would significantly improve space usage.

As for the design of the entire project, it seems to have run fairly smoothly. The main problem we encountered was in keeping track of all the changes in the other components upon which ours relies. However, this is expected of a project of this type and size. One thing that perhaps would have to helped is an overview meeting early in the project involving all of the groups, in addition to our weekly group meetings. This would have given us a better understanding early on as to how the other groups (i.e. buffer manager, log manager, etc.) were designing their components. Although many of these designs changed significantly from the original plans, it perhaps would have helped us get started on our code a little sooner. (But Bill and Peter did a nice job of keeping everyone informed of major changes.)

Also we better fix the global variables and constants at an earlier stage. For instance, the error class is changed in the middle of the project so everyone has to change their codes to reflect that. It would be not very hard if all groups gather together to discuss these global things. Since memory is shared, globals variables are better to be pointers. To keep track where the error occurs at the first stage, a not_so_simple error class should be provided, etc. Constants such as MAXNAMELEN, PAGESIZE are to be determined early and be broadcast early. Types such as PageId, attrType are provided to avoid the situation that many people typedef their own types which are essentially same.

APPENDIX

Copies of the files **btfile.h** and **btfile.c** are attached.