Database Systems

Instructor: Hao-Hua Chu

Fall Semester, 2005

Assignment 4: B+ Tree

Deadline: 09:00 December 5 (Monday), 2005

Cheating Policy: If you are caught cheating, your grade is 0.
Late Policy: You may hand in your late assignment before 23:59 on Wednesday (12/7/2005) for 80% of original grade, or before 23:59 on Friday (12/9/2005) for 70%. We will not accept any assignment submissions after Monday (12/9/2005).

1. Introduction

In this assignment, you will implement part of a B+ tree in which leaf level pages contain entries of the form [key, rid of a data record] (Alternative 2 for data entries, in terms of the textbook.) You must implement the full search and insert algorithm as discussed in class. Your insert routine must be capable of dealing with overflows (at any level of the tree) by splitting pages.

You will be given HFPage and SortedPage. SortedPage is derived from HFPage, and it augments the insertRecord method of HFPage by storing records on the HFPage in sorted order by a specified key value. The key value must be included as the initial part of each inserted record, to enable easy comparison of the key value of a new record with the key values of existing records on a page. The documentation available in the header files is sufficient to understand what operation each function performs.

There are two page-level classes, BTIndexPage and BTLeafPage, both of which are derived from SortedPage. These page classes are used to build the B+ tree index. In these classes, you can create, destroy, open and close a B+ tree index. They are also able to open a scan on the B+ tree, allowing its caller to iterate through all of the data entries (from the leaf pages) that satisfy some search criterion.

2. Getting Started

- Copy *Makefile* to your working directory and modify *MINIBASE*.
- Type *make setup*
- implement *btfile.C* and *btfile.h* file
- *main.C, btree_driver.C, keys*: B+ tree test driver program and the ascii key data that will used by the testing program.
- *results*: correct test output

You can find other useful include files btfilepart.h, btreefilescan.h, btindex_page.h , btleaf_page.h, *bt.h, hfpage.h, sorted_page.h, index.h, test_driver.h, btree driver.h, minirel.h* and *new_error.h* in /include.

3. Design Overview

You should begin by reading the chapter Tree Structured Indexing of the textbook to get an overview of the B+ tree layer.

3.1 A Note on Keys for this Assignment

You should note that key values are passed to functions using void * pointers (pointing to the key values). The contents of a key should be interpreted using the AttrType variable. The key can be either a string(attrString) or an integer(attrInteger), as per the definition of AttrType in minirel.h. We just implement these two kinds of keys in this assignment. If the key is a string, it has a fixed maximum length, MAX_KEY_SIZE1, defined in bt.h.

Although the specifications for some methods (e.g., the constructor of BTreeFile) suggest that keys can be of (the more general enumerated) type AttrType, you can return an error message if the keys are not of type attrString or attrInteger.

The SortedPage class, which augments the insertRecord method of HFPage by storing records on a page in sorted order according to a specified key value, assumes that the key value is included as the initial part of each record, to enable easy comparison of the key value of a new record with the key values of existing records on a page.

3.2 B+ Tree Page-Level Classes

There are four separate pages classes. HFPage is the base class (given), and from it is derived SortedPage. BTIndexPage and BTLeafPage are derived from SortedPage.

- *HFPage*: This is the base class, you can look at hfpage.h to get more details.
- *SortedPage*: This class is derived from the class HFPage. Its only function is to maintain records on a HFPage in a sorted order. Only the slot directory is rearranged. The data records remain in the same positions on the page. This exploits the fact that the rids of index entries are not important: index entries (unlike data records) are never `pointed to' directly, and are only accessed by searching the index page.
- *BTIndexPage*: This class is derived from SortedPage. It inserts records of the type [key, pageNo] on the SortedPage. The records are sorted by the key.
- *BTLeafPage*: This class is derived from SortedPage. It inserts records of the type [key, dataRid] on the SortedPage. dataRid is the rid of the data record. The records are sorted by the key. Further, leaf pages must be maintained in a doubly linked list.

The following two figures show examples of how a valid B+ Tree might look.

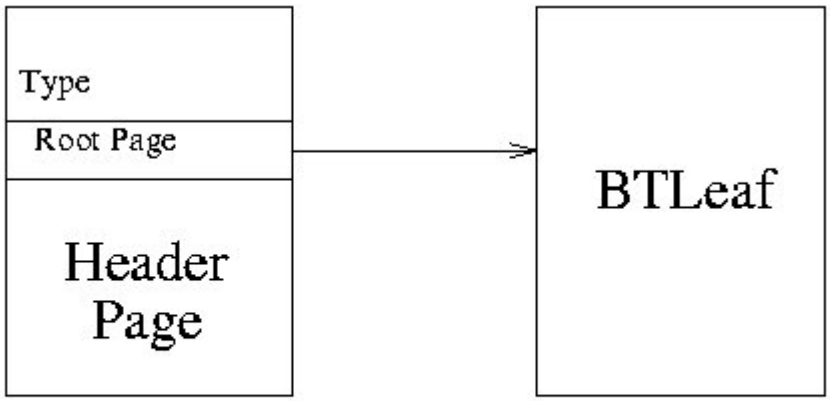

Figure 1: Layout of a BTree with one BTLeafPage

Figure 1 shows what a BTreeFile with only one BTLeafPage looks like; the single leaf page is also the root. Note that there is no BTIndexPage in this case.

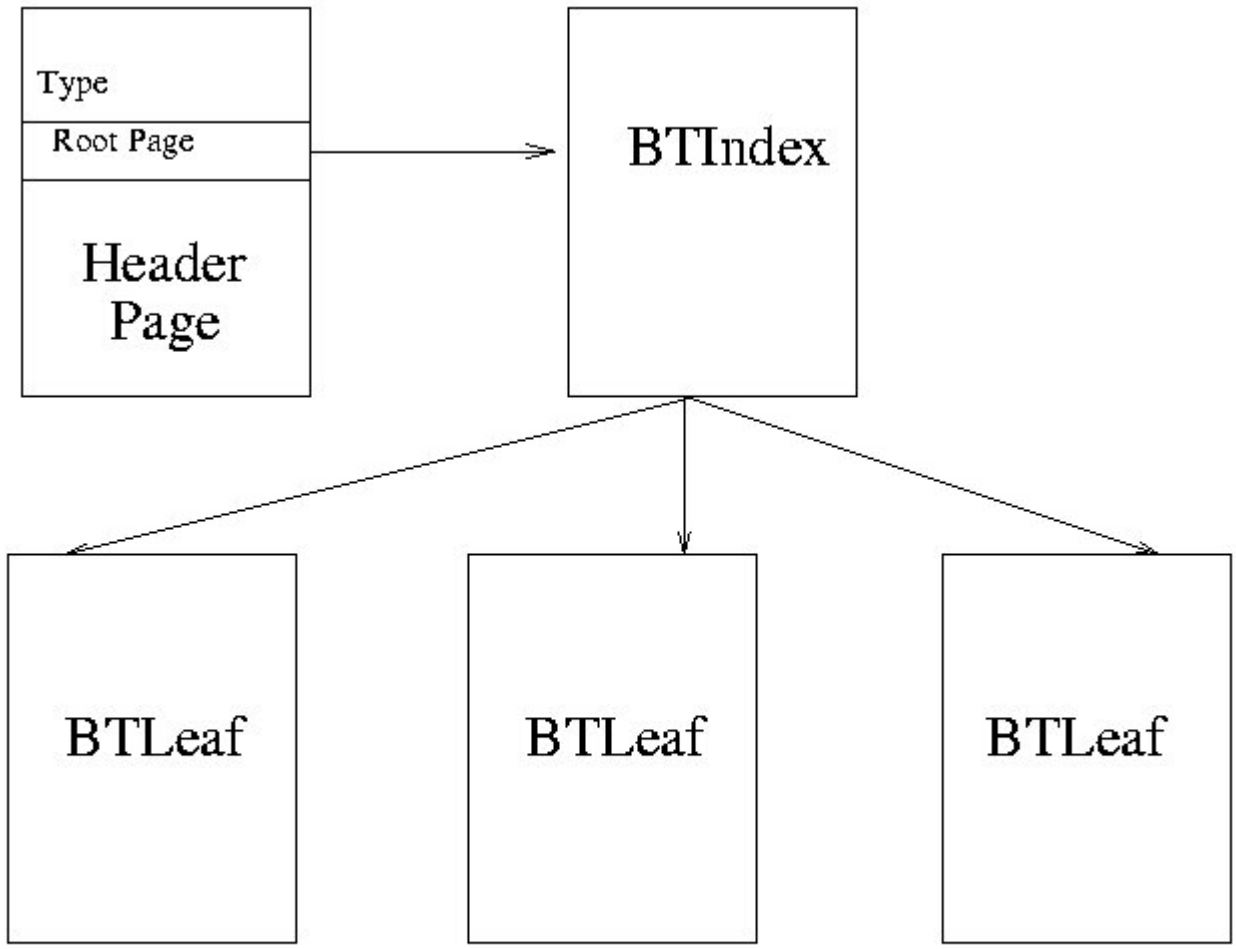

Figure 2: Layout of a BTree with more than one BTLeafPage

Figure 2 shows a tree with a few BTLeafPages, and this can easily be extended to contain multiple levels of BTIndexPages as well.

For further details about the individual methods in these classes, look at the header pages for the classes.

Remember to cast pointers to this struct as (Page *) pointers when making calls to functions such as *pinPage()*.

3.3 Other B+ Tree Classes

3.3.1 IndexFile and IndexFileScan

A BTree is one particular type of index. There are other types, for example a Hash index. However, all index types have some basic functionality in common. We've taken this basic index functionality and created a virtual base class called IndexFile. You won't write any code for IndexFile. However, any class derived from an IndexFile should support IndexFile(), Delete(), and insert(). (IndexFile and IndexFileScan are defined in /include/index.h).

Likewise, an IndexFileScan is a virtual base class that contains the basic functionality all index file scans should support.

3.3.2 BTreeFile

The main class to be implemented for this assignment is BTreeFile. BTreeFile is a derived class of the BTreeFilePart and BTreeFilePart is derived from IndexFile class, which means a BTreeFile is a kind of IndexFile. However, BTreeFilePart implements most functions of IndexFile, the only function you need to implement in BTreeFile is the insert and search algorithm:

IndexFileScan BTreeFile::*new_scan(const void *lo_key = NULL, const void *hi_key = NULL);

In this function, you should create a scan with given keys
Cases:

(1) lo_key = NULL, hi_key = NULL
    scan the whole index
(2) lo_key = NULL, hi_key!= NULL
    range scan from min to the hi_key
(3) lo_key!= NULL, hi_key = NULL
    range scan from the lo_key to max
(4) lo_key!= NULL, hi_key!= NULL, lo_key = hi_key
    exact match (might not unique)
(5) lo_key!= NULL, hi_key!= NULL, lo_key < hi_key
    range scan from lo_key to hi_key

Status BTreeFile::insert(const void *key, const RID rid);
The BTreeFile::insert method takes two arguments: (a pointer to) a *key* and the *rid* of a data record. The data entry to be inserted---i.e., a `record' in the leaf pages of the BTreeFile ---consists of the pair [key, rid of data record].
If a page overflows (i.e., no space for the new entry), you should split the page. You may have to insert additional entries of the form [key, id of child page] into the higher level index pages as part of a split. Note that this could recursively go all the way up to the root, possibly resulting in a split of the root node of the B+ tree.

3.3.3 BTreeFileScan

Scans return data entries from the leaf pages of the tree. You will create the scan through a member function of BTreeFile (*BtreeFile::new_scan(...)* as defined in *btfile.h*). The parameters passed to *new_scan()* specify the range of keys that will be scanned in the B+ tree. They are explained in detail in *btfile.h*.

3.4 Errors

In the Buffer Manager assignment, you learned how to use the Minibase error protocol. Reviewing it now would be a good idea. In that assignment, all the errors you returned belonged to one of the categories in new error.h, namely BUFMGR. In this assignment, you will need to use BTREE, BTLEAFPAGE, and BTINDEXPAGE.

4. Some Notes for BTreeFile::new_scan()

For those who don’t participate in Practicum in Database System, this section can help you to get idea about some useful functions which could be used in your implementation.

bth: bth is a member variable (BTHeader *bth;) which is inherited from base class (BTreeFilePart). It’s a pointer pointing to the root of B+ tree. The structure of BTHeader:

```
class BTHeader
{
   public:
```

```
    PageId root_pid;
    AttrType key_type;
    int key_length;
    int pagenum;
};
```

You can know the key type by **bth->key_type** and get root pageId by **bth->root_pid**.

MINIBASE_BM->pinPage(PageId PageId_in_a_DB, Page*& page);
MINIBASE_BM->unpinPage(PageId globalPageId_in_a_DB, int dirty);
When you want to access a page, call pinPage and it will return a pointer (page) pointing to the specified page. When you don't need that page anymore, you should call unpinPage. If the page had been modified, remember to set the dirty bit.

PageId BTIndexPage::getLeftLink(): Get the left-most pointer of the index page.
HFPage::firstRecord(RID& firstRid): Get the first record's rid of the page.

int keyCompare(const void *key1, const void *key2, AttrType t);
Compare two keys, key1 and key2
Return values:
- key1 < key2 : negative
- key1 == key2 : 0
- key1 > key2 : positive

In BTreeFile::new_scan(), you need to return an BTreeFileScan object. BTreeFileScan is used to get records of a range scan. Each time BTreeFileScan::get_next() is called to get next record in this range scan. Therefore, the main effort in new_scan() is to fill in the member variables of BTreeFileScan object:

RID startrid: rid of the first matched record.
int no_bound: if there is no up-bound for the scan, i.e. hi_key=null, set this variable to 1. Else, set it to 0 and copy the hi_key to the following variable.
void *endkey;
AttrType keytype;

5. Submission

Please e-mail your “*btfile.C*” and “*btfile.h*” to *r93922001@ntu.edu.tw*