Project 2

Part A Due Sunday, 11/08/2015 by 11:00PM
Part B Due Sunday, 11/15/2015 by 11:00PM
Part C Due Sunday, 11/22/2015 by 11:00PM
Part D Due Sunday, 11/29/2015 by 11:00PM

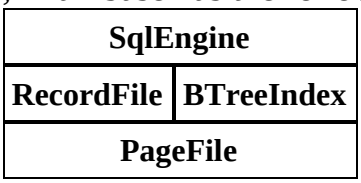
In this project, you will have to implement B+tree indexes for Bruinbase and use them to handle SELECT queries. Towards this goal, you will first download, build, and play with Bruinbase to get familiar with it. You will then go over its overall system architecture and the source code to learn its internals. Finally, you will implement the code for the B+tree index and modify its SQL engine to make Bruinbase use a B+tree for query processing.

Part 0: Download and Understand Bruinbase

First, read the Bruinbase overview page to download the Bruinbase code and play with it.

Understand Bruinbase Architecture

Now that you are familiar with the functionality of Bruinbase, it is time to learn its internal architecture. Internally, Bruinbase has the following four core modules:



PageFile: The `PageFile` class (implemented in `PageFile.h` and `PageFile.cc`) at the bottom of the above diagram provides page-level access to the underlying Unix file system. All file read/write is done in the unit of a page (whose size is set by the `PageFile::PAGE_SIZE` constant, which is 1024). That is, even if you want to read/write a few bytes in a file, you have to read the full page that contains the bytes. The `PageFile` module uses the LRU policy to cache the most-recently-used 10 pages in the main memory to reduce disk IOs.

The following diagram shows the conceptual structure of a PageFile that has 7 pages:

PageId: 0 1 2 3 4 5 6

In the above diagram, each rectangle corresponds to a page in the file. Every page in a `PageFie` is identified by its `PageId` which is an integer value starting at 0. `PageFile`supports the following file access API (the complete list of Bruinbase classes and their methods is available as part of the Bruinbase API documentation):

- `open()`: This function opens a file in the read or write mode. When you open a non-existing file in the write mode, a file with the given name is automatically created.
- `close()`: This function closes the file.
- `read()`: This functions allows you to read a page in the file into the main memory. For example, if you want to read the third (grey) page in the above diagram, you will have to call `read()` with PageId=2 with a pointer to a 1024-byte main memory buffer where the page content will be loaded.
- `endPid()`: This function returns the ID of the page immediately after the last page in the file. For example, in the above diagram, the call to `endPid()` will return 7 because the last PageId of the file is 6.
  Therefore, `endPid()==0` indicates that the file is empty and was just created. You can scan an entire PageFile by reading pages from PageId=0 up to immediately before endPid().
- `write()`: This function allows you to write the content in main memory to a page in the file. As its input parameters, you have to provide a pointer to 1024-byte main memory buffer and a PageId.
  - If you write beyond the last PageId of a `PageFile`, the file is automatically expanded to include the page with the given ID. Therefore, if you want to allocate a new page from PageFile, you can call endPid() to obtain the first unallocated PageId and write to that page. This way, a new page will be automatically added at the end of the file.

RecordFile: The `RecordFile` class (implemented in `RecordFile.h` and `RecordFile.cc`) provides record-level access to a file. A record in Bruinbase is an integer key and a string value (of length up to 99) pair. Internally, `RecordFile` "splits" each 1024-byte page in `PageFile` into multiple *slots* and stores a record in one of the slots. The following diagram shows the conceptual structure of a RecordFile with a number of (key, value) pairs stored inside:

PageId: 0 1 2 ... 11

SlotId: 0	key,value	key,value	key,value		key,value
1	key,value	key,value	key,value	...	
...	...	...	...		
	key,value	key,value	key,value		

When a record is stored in `RecordFile`, its location is identified by its (PageId, SlotId) pair, which is represented by the `RecordId` struct. For example, in the above diagram, the`RecordId` of the red record (the first record in the second page) is (pid=1, sid=0) meaning that its PageId is 1 and SlotId is 0. `RecordFile` supports the following file access API:

- `open()`: This function opens a file in the read or write mode. When you open a non-existing file in the write mode, a file with the given name is automatically created.
- `close()`: This function closes the file.
- `read()`: This function allows you to read the record at `RecordId` from the file. For example, if you want to read the the red record in the above diagram, you will have to call `read()` with `RecordId` of pid=1 and sid=0.
- `append()`: This function is used to insert a new record at the end of the file. For the above example, if you call `append()` with a new record, it will be stored at `RecordId`pid=11 and sid=1. The location of the stored record is returned in the third parameter of this function. Note that `RecordFile` does not support updating or deleting an existing record. You can only append a new record at the end.
- `endRid()`: This function returns the RecordId immediately after the last record in the file. For instance, a call to `endRid()` in the above example will return a `RecordId`with pid=11 and sid=1. When `endRid()` returns {pid=0, sid=0}, it indicates that the RecordFile is empty. You can scan an entire RecordFile by reading from RecordId with pid=0 and sid=0 through immediately before endRid().

In Bruinbase, `RecordFile` is used to store tuples in a table. (Internally, `RecordFile` uses its private member variable `pf`, which is an instance of `PageFile`, to store tuples in a page.)

SqlEngine: The `SqlEngine` class (implemented in `SqlEngine.h` and `SqlEngine.cc`) takes user commands and executes them.

- `run()`: This function is called when Bruinbase starts. This function waits for user commands, parses them, and calls `load()` or `select()` depending on the user command. When the user issues the QUIT command, the control is returned from this function.

- `load()`: This function is called when the user issues the `LOAD` command. In Part A, you will have to implement this function to support loading tuples into a table from a load file.
- `select()`: When the user issues the `SELECT` command, this function is called. The provided implementation scans all tuples in the table to compute the answer. Later in Part C, you will have to extend this function to use an index for more efficient query processing.

BTreeIndex: The `BTreeIndex` class implements the B+tree index. The provided source code only contains its API definition, so you will have to implement this class as Part B of this project.

Make sure that you clearly understand the basic architecture of Bruinbase and its API before you proceed to the next step.

Part A: Implement LOAD Command

In the first part of Project 2, you will be implementing the bulk load command, similar to the `LOAD DATA` command you used in MySQL for Project 1B. For Bruinbase, the syntax to load data into a table is:

```
LOAD tablename FROM 'filename' [ WITH INDEX ]
```

This command creates a table named `tablename` and loads the (key, value) pairs from the file `filename`. If the option `WITH INDEX` is specified, Bruinbase also creates the index on the key column of the table. The format for the input file must be a single key and value pair per line, separated by a comma. The key must be an integer, and the value (a string) should be enclosed in double quotes, such as:

```
1,"value 1"
2,"value 2"
...
```

Note that when the user issues a LOAD command, Bruinbase invokes the `SqlEngine::load(table, loadfile, index)` function with the user-provided table name and the load file name as its parameters. The third parameter is set to true if `WITH INDEX` option is specified. For example, if the user issues the command "LOAD movie FROM 'movieData.del'", the first parameter `table` will be "movie", the second parameter `loadfile` will be "movieData.del", and the third parameter `index` will be false. Implement the `SqlEngine::load()` function to load tuples into the table from the load file. Since you have not implemented any indexes yet, you do not need to worry

about the index creation part yet. For now, assume that the third parameter `index` will be always false and implement the load function.

For reading the loadfile, you may use any standard C/C++ file I/O functions, like `fstream`, `fgets`, etc. For table storage, however, you must use the provided `RecordFile` class. The created `Recordfile` should be named as tablename + ".tbl". For example, when the user issues the command "`LOAD movie FROM 'movieData.del'`", you should create a RecordFile named `movie.tbl` (in the current working directory) and store all tuples in the file. If the file already exists, the `LOAD` command should append all records in the load file to the end of the table. Roughly, your implementation of the `load` function should open the input loadfile and the RecordFile, parse each line of the loadfile to read a tuple (possibly using SqlEngine::parseLoadLine()) and insert the tuple to the RecordFile. Note that for this part of the project, you can modify only the `load()` function of `SqlEngine`, not any other parts of the Bruinbase code.

In bruinbase.zip, we have provided a sample data file `movie.del`, as well as a RecordFile `movie.tbl` loaded from the sample data file. As you will be using the `RecordFile` class for table storage, the table file created by your code from `movie.del` should be roughly the same as the one provided. You can verify if your load command is successful by comparing some query results between the two tables.

Remember that, as you modify your code, you'll need to run `make` and rerun Bruinbase to witness any changes you've made. As you implement the `SqlEngine::load` function, you may find it helpful to use debugging tools such as GDB.

The primary goal of Part A is to make sure that you (1) understand the Bruinbase code thoroughly and (2) feel comfortable with modifying the code as needed. Since our focus is not string parsing, you do not have to worry about possible variations in the input load file. During the testing of your code, we will use only "well-formed" load files that follow the spec, so as long as your code can correctly parse and load well-formed load files, you won't lose any points for this part of the project. The provided`SqlEngine::parseLoadLine()` is likely to be sufficient for your parsing need of the input load file.

Notes on CR/LF issue: Again, if your host OS is Windows, you need to pay particular attention to how each line of a text file ends. If you get some unexpected errors from make, flex, bison, or g++ during your build, make sure that this is not because of the CR/LF issue. Run `dos2unix` from the guest OS on all files modified by a Windows program.

Submit your work of Part A by the deadline. See the submission instructions at the bottom of the page to submit Part A. We emphasize that even though only Part A is due after one week, **you are STRONGLY encouraged to continue with Part B in the first week, because the rest of this project will be SIGNIFICANTLY more work than earlier projects.** We expect that Parts B and C will take at least 20 hours to complete even for the students with much C++ programming experience and significantly more for those with less experience. We have made Part A due in one week to ensure that everyone starts with this project early on, not to delay your work on Part B to the second week of this project.

We also emphasize that while Project 2 is split into 4 submissions, the project is really a single project and will be graded *once* after all submissions are over. We have split the project into multiple submissions to encourage an early start, not because each part is a separate project that will be graded independently. In particular, the final grading will be done by running a test script that requires that **your code compiles, loads tuples, and executes queries**. To get partial credit, it is of paramount importance that your submitted code works at least for some test cases provided at the end of this page. Even if you implement up to Part C perfectly, if you fail to submit a working version of Part D, we won't be able to give you partial credit because our script wouldn't work. Carefully plan ahead your schedule, so that you will be able to submit a **WORKING VERSION FOR EVERY PART** even if each part may not work for all cases.

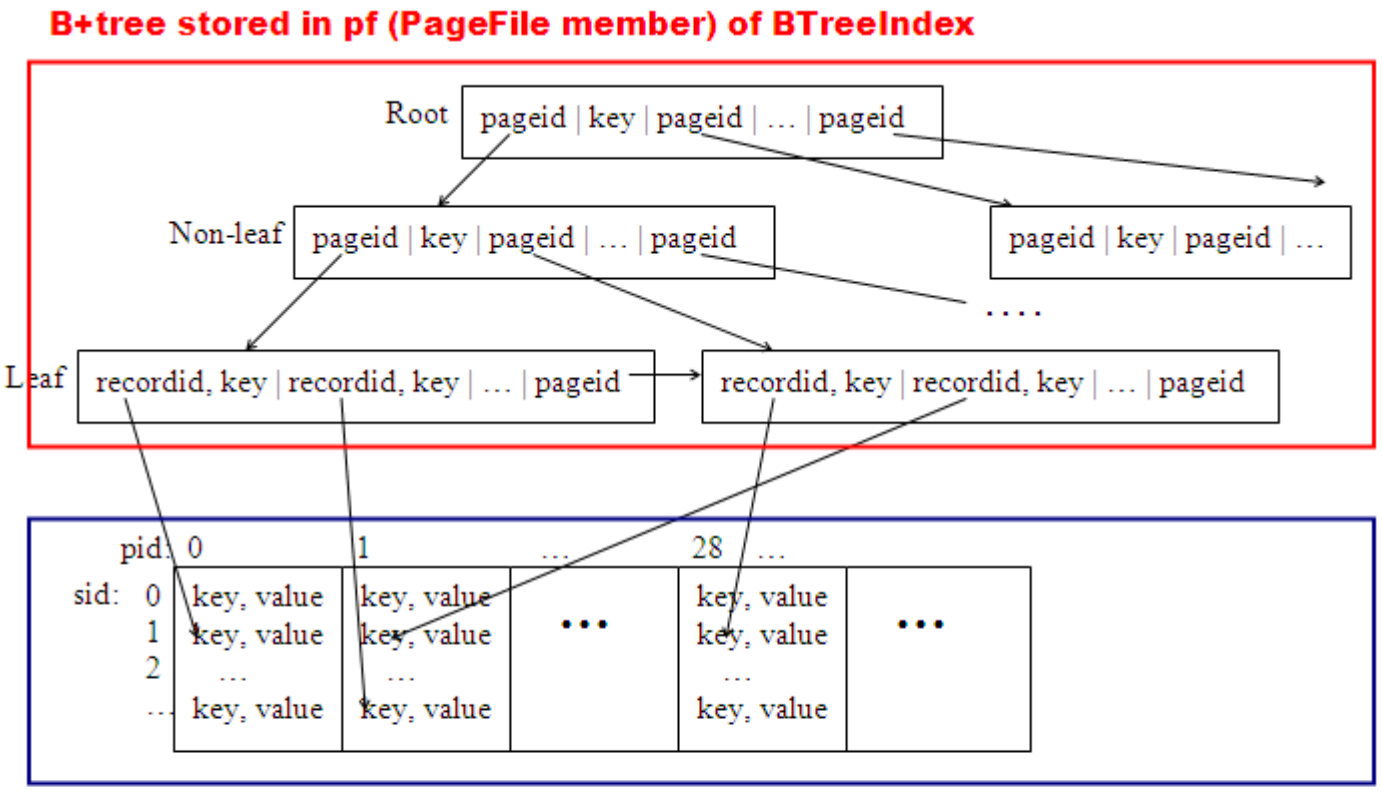
Part B: Implement B+tree (part 1)

For Parts B and C of Project 2, you will be adding the B+tree indexing mechanism to Bruinbase.

Overview of the Structure of B+tree in Bruinbase

If you are still not sure about how B+tree works, please go over the lecture slides on index and Chapter 12.3 of the textbook to understand the search and insertion algorithms for B+tree. Remember that Bruinbase only supports load and select, which means your index must support search and insertion, but deletion support is not required.

Take a few minutes to read through the `BTreeIndex` interface defined in `BTreeIndex.h` to get a sense of which member functions will handle the various tasks for record insertion and key lookup. Here is an abstract diagram of the overall structure of B+tree that you will have to implement for Parts B and C.



- In this diagram, one black rectangle corresponds to one page in a PageFile
- In B+tree, a pageid stored in a non-leaf node works as the "pointer" to a child node
  - The last pageid in a leaf node of B+tree works as the pointer to its next sibling node
- In B+tree, a recordid stored in a node works as a "pointer" to a record in RecordFile

- In this diagram, one black rectangle corresponds to one disk page. In particular, every node in B+tree corresponds to a page in `PageFile`.
- In B+tree, a PageId stored in a non-leaf node works as the "pointer" to a child node.
  - The last PageId in a leaf node of B+tree works as the pointer to its next sibling node.
- In B+tree, a RecordId stored in a leaf node works as a pointer to a record in a RecordFile.

The above diagram shows the minimum information that should be stored in the nodes, and you are welcome to add more information as necessary. As you see in the above diagram, `PageId` can be effectively used as a "pointer" to a page in a PageFile (and thus, to a node in B+tree) and `RecordId` can be used as a "pointer" to a record in a`RecordFile`. Note that all information of a B+tree is eventually stored in a `PageFile`, which is a member variable of `BTreeIndex`.

The three most important methods in `BTreeIndex` are the following:

```
  /**
   * Insert (key, RecordId) pair to the index.
   * @param key[IN] the key for the value inserted into the index
   * @param rid[IN] the RecordId for the record being inserted into the index
   * @return error code. 0 if no error
   */
  RC insert(int key, const RecordId& rid);

  /**
   * Run the standard B+Tree key search algorithm and identify the
   * leaf node where searchKey may exist. If an index entry with
   * searchKey exists in the leaf node, set IndexCursor to its location
   * (i.e., IndexCursor.pid = PageId of the leaf node, and
   * IndexCursor.eid = the searchKey index entry number.) and return 0.
   * If not, set IndexCursor.pid = PageId of the leaf node and
   * IndexCursor.eid = the index entry immediately after the largest
   * index key that is smaller than searchKey, and return the error
   * code RC_NO_SUCH_RECORD.
   * Using the returned "IndexCursor", you will have to call readForward()
   * to retrieve the actual (key, rid) pair from the index.
   * @param key[IN] the key to find
   * @param cursor[OUT] the cursor pointing to the index entry with
   *                    searchKey or immediately behind the largest key
   *                    smaller than searchKey.
   * @return 0 if searchKey is found. Othewise, an error code
   */
  RC locate(int searchKey, IndexCursor& cursor);

  /**
   * Read the (key, rid) pair at the location specified by the IndexCursor,
   * and move foward the cursor to the next entry.
   * @param cursor[IN/OUT] the cursor pointing to an leaf-node index entry in
the b+tree
   * @param key[OUT] the key stored at the index cursor location
   * @param rid[OUT] the RecordId stored at the index cursor location
   * @return error code. 0 if no error
   */
  RC readForward(IndexCursor& cursor, int& key, RecordId& rid);
```

Perhaps the best way to understand what each method of `BTreeIndex` should do is to trace what `SqlEngine` will have to do when a new tuple (10, 'good') is inserted. As you implemented in Part A, `SqlEngine` stores the tuple in `RecordFile` by

calling `RecordFile::append()` with (10, 'good'). When append() finishes, it returns the location of the inserted tuple as the third parameter `RecordId`. Let us say the returned `RecordId` is [3,5] (i.e., pid=3 and sid=5). Now that your tuple is stored in `RecordFile` at [3,5], `SqlEngine`needs to insert the tuple's key and the "pointer" to the tuple (i.e., its `RecordId` [3,5]) into the B+tree. That is, `SqlEngine` will have to call `BTreeIndex::insert()` with the parameter (10, [3,5]), where 10 is the key of the inserted tuple and [3,5] is the tuple's location in `RecordFile`. Given this input, `BTreeIndex::insert()` should traverse the current B+tree, insert the (10, [3,5]) pair into a leaf node and update its parent node(s) if necessary.

Later, when `SqlEngine` wants to retrieve the tuple with key=10 (let us say, the user issued the query `SELECT * FROM table WHERE key=10`), it will call `BTreeIndex::locate()` with the key value 10. Then your B+tree implementation will have to traverse the tree and return the location of the appropriate leaf-node index entry as `IndexCursor`. Using the returned `IndexCursor`, `SqlEngine` then calls `BTreeIndex::readForward()` to retrieve (10, [3,5]). Finally using the returned `RecordId` [3,5], `SqlEngine` calls `RecordFile::read()`and retrieves the tuple (10, 'good') stored at [3,5].

As you implement the index, remember that you may **NOT** use standard file I/O. Instead, you **MUST** use `PageFile` to store and access the index, where each node in the B+tree corresponds to one page in `PageFile`.

Implement B+tree node insertion and search algorithms

As the first step to implementing B+tree, you have to write the code for managing individual nodes of the B+tree (both leaf and non-leaf nodes). To help you identify what functionalities B+tree nodes should support, we have included a sample interface definition for `BTLeafNode` and `BTNonLeafNode` classes in the `BTreeNode.h` header file.

`BTLeafNode` is the C++ class that supports insert, split, search, and retrieval of index entries from a leaf node of a B+tree:

```
/**
 * BTLeafNode: The class representing a B+tree leaf node.
 */
class BTLeafNode {
  public:
   /**
    * Read the content of the node from the page pid in the PageFile pf.
    * @param pid[IN] the PageId to read
    * @param pf[IN] PageFile to read from
```

```
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC read(PageId pid, const PageFile& pf);

   /**
    * Write the content of the node to the page pid in the PageFile pf.
    * @param [IN] the PageId to write to
    * @param [IN] PageFile to write to
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC write(PageId pid, PageFile& pf);

   /**
    * Insert the (key, rid) pair to the node.
    * Remember that all keys inside a B+tree node should be kept sorted.
    * @param key[IN] the key to insert
    * @param rid[IN] the RecordId to insert
    * @return 0 if successful. Return an error code if the node is full.
    */
    RC insert(int key, const RecordId& rid);

   /**
    * Insert the (key, rid) pair to the node
    * and split the node half and half with sibling.
    * The first key of the sibling node is returned in siblingKey.
    * Remember that all keys inside a B+tree node should be kept sorted.
    * @param key[IN] the key to insert.
    * @param rid[IN] the RecordId to insert.
    * @param sibling[IN] the sibling node to split with.
    *         This node MUST be EMPTY when this function is called.
    * @param siblingKey[OUT] the first key in the sibling node after split.
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC insertAndSplit(int key, const RecordId& rid, BTLeafNode& sibling, int&
siblingKey);

   /**
    * If searchKey exists in the node, set eid to the index entry
    * with searchKey and return 0. If not, set eid to the index entry
    * immediately after the largest index key that is smaller than searchKey,
    * and return the error code RC_NO_SUCH_RECORD.
    * Remember that keys inside a B+tree node are always kept sorted.
    * @param searchKey[IN] the key to search for.
    * @param eid[OUT] the index entry number with searchKey or immediately
```

```
                        behind the largest key smaller than searchKey.
     * @return 0 if searchKey is found. If not, RC_NO_SEARCH_RECORD.
     */
     RC locate(int searchKey, int& eid);

    /**
     * Read the (key, rid) pair from the eid entry.
     * @param eid[IN] the entry number to read the (key, rid) pair from
     * @param key[OUT] the key from the entry
     * @param rid[OUT] the RecordId from the entry
     * @return 0 if successful. Return an error code if there is an error.
     */
     RC readEntry(int eid, int& key, RecordId& rid);

    /**
     * Return the PageId of the next sibling node.
     * @return PageId of the next sibling node
     */
     PageId getNextNodePtr();

    /**
     * Set the PageId of the next sibling node.
     * @param pid[IN] the PageId of the sibling node.
     * @return 0 if successful. Return an error code if there is an error.
     */
     RC setNextNodePtr(PageId pid);

    /**
     * Return the number of keys stored in the node.
     * @return the number of keys in the node
     */
     int getKeyCount();

 private:
    /**
     * The main memory buffer for loading the content of the disk page
     * that contains the node.
     */
     char buffer[PageFile::PAGE_SIZE];
};
```

Remember that each node in a B+tree has to be eventually written to and read from the disk as a page in `PageFile`. When you access a B+tree for traversal or insertion, you will first have to read the corresponding disk pages into main memory, since you

cannot manipulate the node directly inside the disk. Therefore, for each B+tree node that you access, you will need 1024-byte main memory to "load" the content of the node from the disk. The 1024-byte-array member variable, `buffer`, of `BTLeafNode` can be used for this, but you are welcome to take a different approach and modify the class definition.

Once the page for a node is read into the main memory (of size 1024 bytes), your code will have to read and store keys, RecordIds, and PageIds inside the main memory using pointers and typecasting operators. If you are not comfortable with arrays and pointers in C++, you may find it useful to review the C++ Pointer Tutorial. Pay particular attention to the sections on arrays and pointer arithmetic.

Since you will be doing lots of pointer-based memory access, you are likely to encounter memory-related bugs. These bugs are known to be very difficult to fix, since the place that you introduce the bug is often different from where you observe an unexpected behavior. There is an excellent memory-related debugging tool called `valgrind`. If you run your program in valgrind, it gives you a warning immediately where an out-of-bound or uninitialized memory access occurs. It gives you warning for memory leaks as well. You can run your program in valgrind by typing "`valgrind` *your_program_name*". See Valgrind Quick Start Guide to learn how to use it.

The leaf nodes of your index have to store (`key`, `RecordId`) pairs that are provided as the input parameters of `BTLeafNode::insert()`. Since these pairs will essentially be provided by `SqlEngine`, you do not have to worry about where these values come from for this part of the project. For now, all you have to do is to store the (`key`, `RecordId`) pair at the node, and when the user tries to `locate` the `searchKey` from the node, return the corresponding entry number. (More precisely, the first entry whose key is larger than or equal to searchKey.) You may also assume that there will be **no duplicate key value** inserted into a node. Finally, in your implementation, **each leaf node MUST be able to store at least 70 keys**.

We emphasize that we provide the the above class definition just as a hint on what functionalities your B+tree leaf nodes should support. You are welcome to modify the class definition if a different interface is more suitable for your coding style. Feel free to add extra member functions and variables to the class as necessary.

`BTNonLeafNode` is the C++ class that supports insert, split, and search mechanisms for non-leaf nodes of B+tree.

```
/*
 * BTNonLeafNode: The class representing a B+tree nonleaf node.
```

```
 */
class BTNonLeafNode {
  public:
   /**
    * Read the content of the node from the page pid in the PageFile pf.
    * @param pid[IN] the PageId to read
    * @param pf[IN] PageFile to read from
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC read(PageId pid, const PageFile& pf);

   /**
    * Write the content of the node to the page pid in the PageFile pf.
    * @param [IN] the PageId to write to
    * @param [IN] PageFile to write to
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC write(PageId pid, PageFile& pf);

   /**
    * Insert the (key, pid) pair to the node.
    * Remember that all keys inside a B+tree node should be kept sorted.
    * @param [IN] the key to insert
    * @param [IN] the PageId to insert
    * @return 0 if successful. Return an error code if the node is full.
    */
    RC insert(int key, PageId pid);

   /**
    * Insert the (key, pid) pair to the node
    * and split the node half and half with sibling.
    * The middle key after the split is returned in midKey.
    * Remember that all keys inside a B+tree node should be kept sorted.
    * @param key[IN] the key to insert
    * @param pid[IN] the PageId to insert
    * @param sibling[IN] the sibling node to split with. This node MUST be empty
when this function is called.
    * @param midKey[OUT] the key in the middle after the split. This key should
be inserted the parent node.
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC insertAndSplit(int key, PageId pid, BTNonLeafNode& sibling, int& midKey);

   /**
    * Given the searchKey, find the child-node pointer to follow and
```

```
    * output it in pid.
    * @param searchKey[IN] the searchKey that is being looked up.
    * @param pid[OUT] the pointer to the child node to follow.
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC locateChildPtr(int searchKey, PageId& pid);

   /**
    * Initialize the root node with (pid1, key, pid2).
    * @param pid1[IN] the first PageId to insert
    * @param key[IN] the key that should be inserted between the two PageIds
    * @param pid2[IN] the PageId to insert behind the key
    * @return 0 if successful. Return an error code if there is an error.
    */
    RC initializeRoot(PageId pid1, int key, PageId pid2);

   /**
    * Return the number of keys stored in the node.
    * @return the number of keys in the node
    */
    int getKeyCount();

 private:
   /**
    * The main memory buffer for loading the content of the disk page
    * that contains the node.
    */
    char buffer[PageFile::PAGE_SIZE];
};
```

Note that in principle the structure of a non-leaf node is different from that of a leaf node. In particular, you need to maintain (key, RecordId) pairs for a leaf node, but it is sufficient to store only (key, PageId) pairs for a non-leaf node. Despite this difference, you may decide to use the same structure for both leaf and non-leaf nodes and ignore the sid field of RecordId for non-leaf nodes. This way, you may simplify your implementation and share some code between the implementation of the two node types. (While this implementation is a reasonable choice, it will reduce the branching factor of your tree and make your B+tree less efficient.) Whatever you decide, **each non-leaf node MUST be able to store at least 70 keys**.

Again, you are welcome to change the class definitions for B+tree non-leaf nodes if you find it helpful; you can change function names and their input parameters and add or drop some functions as necessary. It is OK to even merge the two class

defintions, `BTLeafNode` and `BTNonLeafNode`, into one class, say `BTreeNode`, so that you can share code more easily between leaf and nonleaf nodes. Another option is to derive the leaf and nonleaf node classes from a common parent class using inheritance. Whatever you decide, keep in mind that only `BTreeNode.h` and `BTreeNode.cc` files can be modified for Part B.

Submitting correct implementation for Part B will be essential to get reasonable grade for Project 2, so try to thoroughly test your code from Part B in isolation before moving on to Part C. (You will have a chance to resubmit your code for `BTreeNode` later if you want to, but you may get penalty if there are substantial changes.) You may find it useful to implement helper or debugging functions, such as a print function to show the contents of a B+tree node. Such functions are not required, but you may find them helpful to visualize and verify the node structure as you are developing. During your testing, you can use our Makefile and simply execute the "make" command to build your testing program, which will help you avoid any dependency error. Keep in mind the incremental development practices and unit testing you learned in CS31 and CS32.

Part C: Implement B+tree (part 2)

Now that you finished your implementation of individual B+tree nodes, it is time to implement the B+tree insert and search mechanisms. Go over the comments in`BTreeIndex.h`, understand what each method should do, and implement them. To simplify your code, you may assume that **no duplicate key values** will be inserted into the tree.

We strongly recommend you implement the three functions `insert()`, `locate()`, and `readForward()` of `BTreeIndex` as they are defined in `BTreeIndex.h`. You are welcome to add additional helper functions and private member variables to `BTreeIndex`, but if you need to change the definitions of the three functions, contact the TA and get an explicit approval. Regardless, only `BTreeIndex.h` and `BTreeIndex.cc` files can be modified for Part C. If you need to define additoinal classes, structures and functions, add your code to one of these two files.

Hints on B+tree implementation:

- As you traverse down the tree for index lookup, you need to know whether you have reached the leaf level of the tree. There are a number of ways to do this. One possible option is to "save" the height of the tree somewhere, keep track of how many levels of the tree you have traversed down during index lookup, and compare these two numbers to see whether you have reached the leaf level.

Another possibility is to add a special "flag" to each node to indicate whether the node is a leaf node or not.

- When you insert a new entry into a leaf-node, the node may overflow and a new (key, pointer) pair needs to be inserted into its parent. Since B+tree does not maintain pointers to parent nodes, your code essentially has to "keep track of" the sequence of the nodes that you visit as you traverse down the tree, so that you can get the parent node when needed. This can be done either using recursive algorithm or by explicitly storing the page ids of the visited nodes somewhere.
- We strongly suggest that you implement your B+ tree operations using recursive algorithms. Although you may find it somewhat difficult to understand these algorithms at first, using a recursive approach will greatly simplify your coding task.
- Pay attention to what is stored in the main memory and what is stored in the disk. For example, any member variable of the `BTreeIndex` class (e.g., `rootPid`) is **NOT**stored in the disk and the value will be lost when you restart Bruinbase. If you want to store some information "permanently", you have to make sure that it is stored in the disk (possibly as part of your index file).

Again, we suggest implementing helper or debugging functions, such as a print function to show the contents of a B+tree tree, even though they are not required. You will find them helpful to visualize and verify the B+tree structure as you are developing. Apply the incremental development practices and unit testing approach. Thoroughly test your code from Parts B and C in isolation before moving on to Part D. Remember that `gdb` and `valgrind` are your friends if you encounter unexpected errors from your code.

Submit your implementation of `BTreeIndex` before the deadline for Part C. For Part C submission, you may find it necessary to change your code for `BTreeNode` due to minor bugs or changes in your design. It is OK to resubmit your new code for `BTreeNode` as part of this submission, but the changes to `BTreeNode` should be minimal. In particular, the resubmitted code should share at least 50% of the lines with your earlier submission (when measured by `diff`). Changes to more than 50% of the lines will lead up to 5% penalty for Part B.

Part D: Modify SqlEngine

In the final part of the project, you will be modifying the main Bruinbase SQL engine to make use of your indexing code. This will involve augmenting the `SqlEngine::load()` function to generate an index for the table, and the `SqlEngine::select()` function to make use of your index at query time.

- **SqlEngine::load(const string& table, const string& loadfile, bool index)**
  You will need to change the load function you implemented in Part A so that if the third parameter `index` is set to true, Bruinbase creates the corresponding B+tree index on the key column of the table. The index file should be named 'tblname.idx' where tblname is the name of the table, and created in the current working directory. Essentially, you have to change the load function, such that if `index` is `true`, for every tuple inserted into the table, you obtain `RecordId` of the inserted tuple, and insert a corresponding (`key`, `RecordId`) pair to the B+tree of the table.
- **SqlEngine::select (int attr, const string &table, const vector<SelCond> &conds)**
  The `select()` function is called when the user issues the SELECT command. The attribute in the SELECT clause is passed as the first input parameter `attr` (`attr=1`means "key" attribute, `attr=2` means "value" attribute, `attr=3` means "*", and `attr=4` means "COUNT(*)"). The table name in the FROM clause is passed as the second input parameter `table`. The conditions listed in the WHERE clause are passed as the input parameter `conds`, which is a vector of `SelCond`, whose definition is as follows:

```
struct SelCond {
  int attr;       // 1 means "key" attribute. 2 means "value" attribute.
  enum Comparator { EQ, NE, LT, GT, LE, GE } comp;
  char* value;    // the value to compare
};
```

  For example, for a condition like "key > 10", `SqlEngine` will pass a `SelCond` with "attr = 1, comp = GT, and value = '10'".

  The current implementation of `select()` performs a scan across the entire table to answer the query. Your last task will be to modify this behavior of `select()` to meet the following requirements:

  - If a SELECT query has one or more conditions on the key column and if the table has a B+tree, use the B+tree to help answer the query as follows:
    - If there exists an equality condition on key, you should always use the equality condition in looking up the index.
    - Queries which specify a range (like key >= 10 and key < 100) must use the index. `SqlEngine` should try to avoid retrieving the tuples outside of this range from the underlying table by using the index.

        - You should not to use an inequality condition <> on key in looking up the B+tree.
        - You should avoid unnecessary page reads of getting information about the "value" column(s); for example, if ONLY "key" column(s) exist in a SELECT statement, or if travesing leaf nodes on B+tree returns the count(*).
        - As a rule of thumb, **you should avoid unecessary page reads if queries can be answered throgh the information stored in the nodes of a B+ Tree**.
    - Query result sets using the index must be equivalent to when using no index. Any ordering of output tuples is OK.
    - Your implementation should print out only the result tuples (and the performance numbers). No other (debugging) information should be printed either to the standard output or to the standard error.

While doing this part of this project, please remember that **YOU SHOULD NEVER CHANGE THE DEFINITIONS OF SqlEngine::load() AND SqlEngine::select()**. Our testing and grading script assumes the current API of these two functions, so if you change them, you are likely to get zero score for the entire project 2! You are allowed to change only their implementation, not their interface.

Adding indexing support to the query engine should noticeably improve performance, particularly for queries specifying very few tuples (e.g. SELECT * FROM Movie WHERE key = 2997). For small data sets, the running time may not differ significantly, but the number of pages read should give an indication of how much improvement your index is providing. Make sure you adequately test your implementation by issuing load and several different types of select statements. Compare the time and number of pages read for queries both with and without indexing and make sure that there is noticeable difference.

To help you test your code and ensure that you will get at least some partial credit for your work, here are a few sample tests for your bruinbase code. Extract with

```
$ unzip project2-test.zip
```

Included are 5 sample data files, a shell script, an sql script, and the expected output. The test files are created such that they test incrementally more complex cases of B+tree. In particular, below is what case they should be testing (these are assuming you used a reasonably large n value; in your implementation, you should assign a reasonably large n):

- xsmall: 8 tuples (single node b+tree, single-page table)

- small: 50 tuples (single node b+tree, multiple-page table)
- medium: 100 tuples (two-level b+tree, only one split at leaf)
- large: 1000 tuples (two-level b+tree, multiple splits at leaf)
- xlarge: 12000+ tuples (three-level b+tree)

You will get partial credit if your submission handles at least one of the test cases correctly. For example, even if your code does not handle node overflow and split cases correctly, your code should work at least for the `xsmall` and `small` datasets. Please note that these are just a sample, and by no means exhaustive testing. You will need to test several other queries to make sure you're performance is improving over the non-indexed select where expected. Passing these tests is a good sign, but does not guarantee a perfect score during the actual grading. Note that your submission will be graded primarily based on correctness. You will get full credit if your code produces correct results for all queries, as long as your code shows reasonable performance improvement from full table scan. The performance numbers in the sample output from our test cases are just examples, so your code does not have to match or exceed them.

Submit your implementation of `SqlEngine` before the final deadline. If you have to change your code for `BTreeIndex` or `BTreeNode` for Part D submission, you are allowed to resubmit them, but the changes should be minimal. If there is more than 50% change in your earlier code, you will get up to 5% penalty for the resubmitted parts.

Special Reminders:

- Nodes in B+tree should be never too empty. For example, the minimum key in non-leaf nodes should be "ceiling(n/2)-1", where (here) n is the max # of pointers in a node.
- Note that we are mainly testing your "B+Tree" index implementation. Thus, even though your work might return the correct results of a testing query, you may still get no point for that testing case if your implementation of B+Tree is wrong. For example, your results come from directly scanning the entire record file, when the query should be handled through the help of the index.
- Third-party libraries:
    - You can **NOT** ask TA to "install" any third-party library for you to run your code in your README. You work should be runnable without the premise of installing any additional library.
    - It is okay to use STL but you have to be careful about how these will be actually stored in disks. In general we discourage you from using other libraries because they are often not needed.

- You may use another library that is pre-installed in the virtual machine, but you have to justify your cause. Please email TA your justifications in advance to get an approval. If you believe some library is needed for your implementation, please justify (1) "why" those usage are necessary (or better choices) and (2) where (roughly) you plan to use them.
- Please be aware that the B+Tree itself is a low-level data structure. And you have to implement requirements (in particular, B+Tree algorithms) specified in this page on your own, including but not limited to node insertion and splitting. You **cannot** use any existing/third-party library to offload these tasks.

Submission Instruction

Preparing Your Submission

Please create **a folder named as your UID**. Put all your source code files (including Makefile) and your README file into the UID folder. Compress your UID folder into a single zip file called "**P2.zip**". In summary, your zip file should have the following structure.

```
P2.zip
 |
 +- Folder named with Your UID, like "904200000" (without quotes)
     |
     +- readme.txt (your README file)
     |
     +- team.txt (your teaminfo file)
     |
     +- Makefile (project Makefile)
     |
     +- Bruinbase.h, main.cc, SqlParse.y, ... (all your project source files)
```

What to Submit

For each of the four submission deadlines, your **P2.zip** file should have the `Makefile`, all source files, and `readme.txt` file. Your submitted code should compile without any error. You can change the following files in each submission:

- **For part A**, only `SqlEngine.cc` should have been modified from our original files in your submission.
- **For part B**, only `SqlEngine.cc`, `BTreeNode.h` and `BTreeNode.cc` should have been modified from our original files in your submission.

- **For part C**,
  only `SqlEngine.cc`, `BTreeNode.h`, `BTreeNode.cc`, `BTreeIndex.h` and `BTreeIndex.cc` should have been modified from our original files in your submission.
- **For part D**, all source files that are neceesary for compilation and execution of your code.
- **`team.txt`**: A plain-text file (no word or PDF, please) that contains the UID(s) of every member of your team. If you work alone, just write your UID (e.g. 904200000).
  If you work with a partner, write both UIDs separated by a comma (e.g. 904200000, 904200001). **Do not include any other content in this file!**
- A README file that includes:
  - apart from things in our spec, if you decide to made an improvement to reduce(optimize) the number of page reads, please provide a list of your optimizations. In general, for every item in the list, a brief statement plus an example is sufficient. If things are debatable, please provide your justification(s). (*You could skip this part if you do ONLY things asked in this spec.)
  - your email and your partner's name and email.
  - any information you think is useful.

Testing of Your Submission

Grading is a difficult and time-consuming process, and file naming and packaging convention is very important to test your submission without any error. In order to help you ensure the correct packaging of your submission, you can test your packaging by downloading this test script. In essence, this script unzips your submission to a temporary directory and tests whether or not you have included all your submission files. The script for Project 2 can be executed like:

```
cs143@cs143:~$ ./p2_test <Your UID>
```

(Put your P2.zip file in the same directory with this test script, you may need to use "chmod +x p2_test" if there is a permission error).

You **MUST** test your submission using the script before your final submission to minimize the chance of an unexpected error during grading. Again, significant points may be deducted if the grader encounters an error during grading.When evertything runs properly, you will see an output similar to the following from this script:

```
Everything seems OK. Please upload your P2.zip file to CCLE.
```

Submitting Your Zip File

Visit the Project 2 Submission page and submit your "P2.zip" file electronically by the deadline of each part. Note that if you accidentally submit a wrong file, you can always fix it by resubmiting your zip file before the deadline.

In order to accomodate the last minite snafu during submission, you will have 30-minute window after the deadline to finish your submission process. That is, as long as you start your submission before the deadline and complete within 30 minutes after the deadline, we won't deduct your grade period without any penalty.

While Project 2 is split into 4 submissions, the project will be graded *once* after all submissions are over. The final grading will be done by running a number of test cases to evaluate the correctness of your implementation. Since our testing script requires that your code compiles without any error and can load tuple and execute select queries, it is of paramount importance that you submit a working version of your code to get non-zero score for the project. Carefully plan ahead your schedule, so that you will be able to submit a working version for each part. To get partial credit, it is better to submit *all parts* that work in some test cases than to submit only a subset of the parts that are supposed to work in all cases. Make sure that the files you submit compile, build, and work fine without any changes to any of the other original files.

Late Submission Policy

To accommodate the emergencies that students may encounter, each team has 4-day grace period for late submission. The grace period can be used for any part of the project in the unit of one day. For example, a student may use 1-day grace period for Project 1A and 2-day grace period for Project 2B. Note that even if a team submits a project 12 hours late, they would need to use a full day grace period to avoid late penalty. If your project is submitted late, we will automatically use the available days in your grace period unless you specifically mention otherwise in the README file.