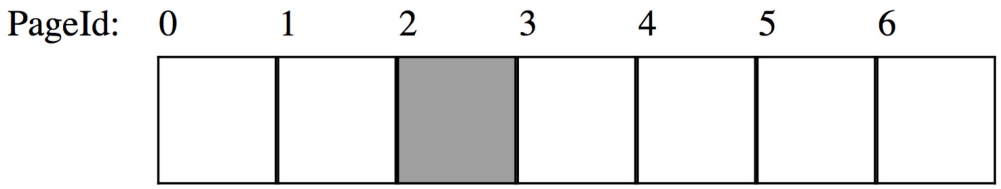
Bruinbase Architecture:

Bruinbase has the following four core modules:

SqlEngine	
RecordFile	BTreeIndex
PageFile	

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:

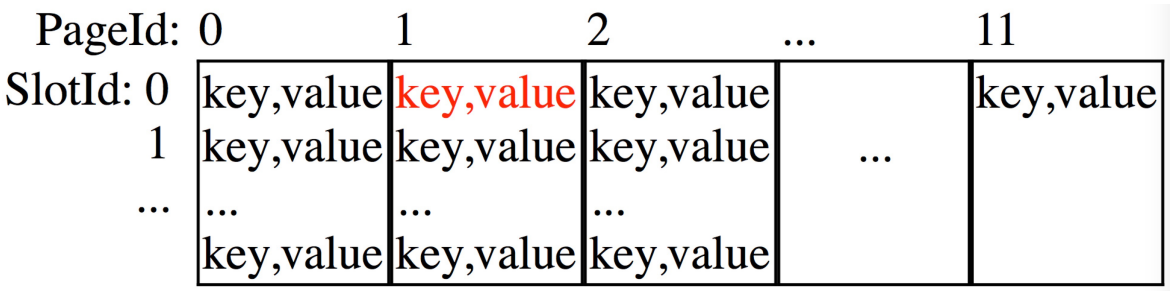


In the above diagram, each rectangle corresponds to a page in the file. Every page in a PageFie is identified by its PageId which is an integer value starting at 0. PageFile supports the following file access API:

• 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:



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.)

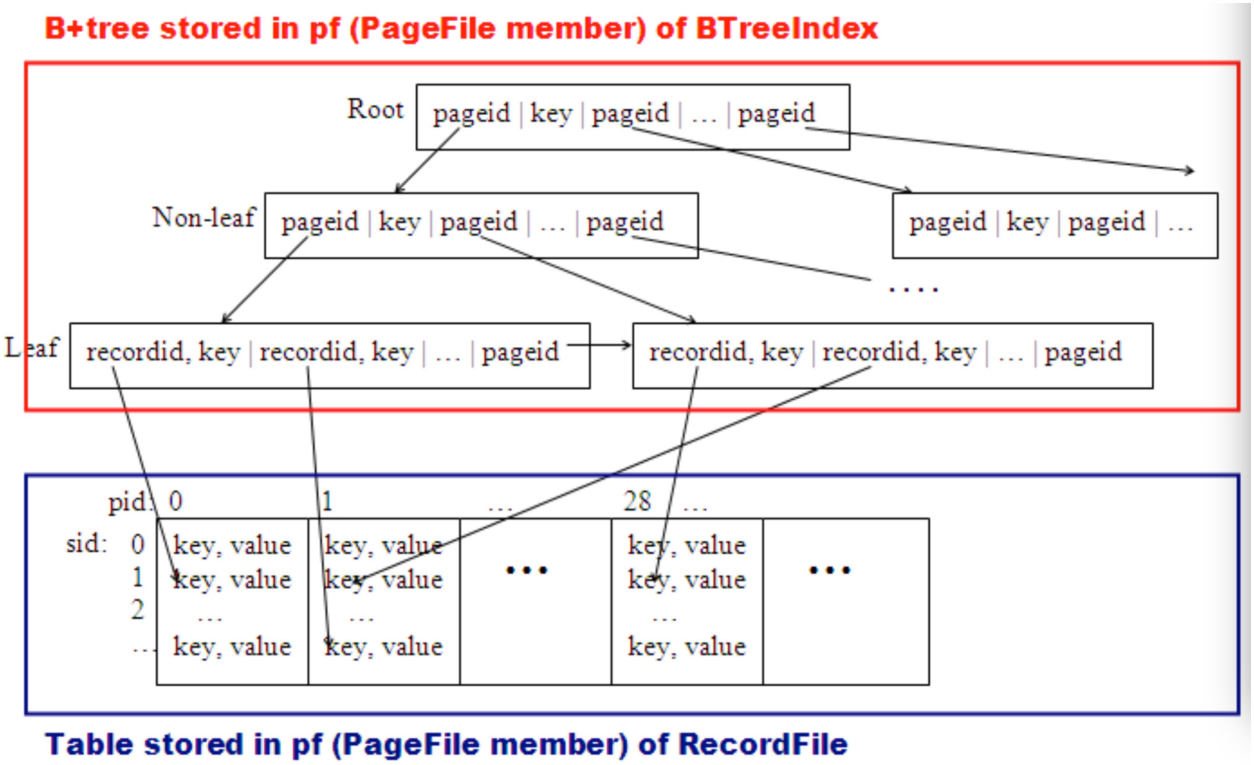
SqlEngine: The SqlEngine class (implemented in SqlEngine.h and SqlEngine.cc) takes user commands and executes them.

• 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.

Implement B+tree:

Overview of the Structure of B+tree in Bruinbase

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:



• 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.

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. 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].

Implement B+tree node insertion and search algorithms

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];
};
```

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];
};
```

Modify SqlEngine:

This will involve augmenting the SqlEngine::load() function to generate an index for the table, and the SqlEngine::select() function to make use of the index at query time.

- SqlEngine::load(const string& table, const string& loadfile, bool index)
  Load function needs to be changed 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, if index is true, for every tuple inserted into the table, RecordId of the inserted tuple is obtained, 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'".