CMPS 181, Spring 2016, Project 3

Due Friday, May 20, 11:55pm on eCommons

Course Project 3 Description

Project 3: Implementing an Index Manager

Introduction

In this project you will implement an Indexing (IX) component. The IX component provides classes and methods for managing persistent indexes over unordered data records stored in files. Each data file may have any number of (single-attribute) indexes associated with it. The indexes ultimately will be used to speed up processing of relational selections, joins, condition-based update, and delete operations. Like the data records themselves, the indexes are also stored in files. Hence, in implementing the IX component, you will use the file system, namely the PagedFileManager (PF) component that you implemented in Project 1, similar to the manner in which you used it for implementing the RecordBasedFileManager (RBF) in Projects 1-2. In the overall Project database system architecture, you can think of the IX component and the RBF component as sitting side-by-side above the PF layer and below the RelationManager (RM) component. As you did for the RBF component, you should pass attribute information in to the methods of the IX component. The RM layer will later be orchestrating both the RBF (for data files) and IX (for indices) layers when tuple-level operations happen, and the RM layer will also be managing the catalog information related to indexes at this level. However, in this part of the project, you will implement the IX component without worrying (yet) about the RM layer, just as you did for the RBF layer earlier. The RM layer and its behavior will be used in Project 4.

B+Trees

You will implement a basic B+ tree that has support for range predicates. Each B+ tree index can be stored in one file in the paged file system. Use the new **codebase.zi**p provided for Project 3 as your starting point, filled in with your code (or TA Coy Humphrey's code) from Project 2.

Note: It is possible to find pseudo-code and perhaps even software packages for B+ trees available publicly. You are welcome to use anything you happen to find as a reference, as long as you provide proper acknowledgment in your readme file when you turn in this part of the project. However, you are not welcome to reuse any code that you find elsewhere; the purpose of this project is for you to gain experience and understanding by actually writing the B+ tree code.

IX Interface

The IX interface that you are to implement consists of two classes: the IndexManager class and IX_ScanIterator class. In addition, there is an IX_PrintError routine for printing messages associated with nonzero return codes. As usual, all IX component public methods (except constructors and destructors) should return 0 if they complete normally and a nonzero return code otherwise. If you compare the RFB and IX components' interfaces, the pattern that you see here should be very familiar.

1. IndexManager Class

The IndexManager class handles the creation, deletion, opening, and closing of index files as well as the insert, delete, and scan initiation operations. Your program should create exactly one instance of the IndexManager class. All necessary initialization for the IX component should take place within the constructor for the IndexManager class. Symmetrically, any necessary clean-up in the IX component should take place within the destructor for the IndexManager class.

```
class IndexManager {

 public:
  static IndexManager* instance();

  // Create an index file
  RC createFile(const string &fileName);

  // Delete an index file
  RC destroyFile(const string &fileName);

  // Open an index and return an ixfileHandle
  RC openFile(const string &fileName, IXFileHandle &ixfileHandle);

  // Close an ixfileHandle for an index.
  RC closeFile(IXFileHandle &ixfileHandle);

  // Insert an entry into the given index that is indicated by the given ixfileHandle
  RC insertEntry(IXFileHandle &ixfileHandle, const Attribute &attribute, const void *key,
const RID &rid);

  // Delete an entry from the given index that is indicated by the given ixfileHandle
  RC deleteEntry(IXFileHandle &ixfileHandle, const Attribute &attribute, const void *key,
const RID &rid);

  // Initialize and IX_ScanIterator to support a range search
  RC scan(IXFileHandle &ixfileHandle,
      const Attribute &attribute,
      const void *lowKey,
      const void *highKey,
      bool lowKeyInclusive,
      bool highKeyInclusive,
      IX_ScanIterator &ix_ScanIterator);

  // Print the B+ tree in pre-order (in a JSON record format)
  void printBtree(IXFileHandle &ixfileHandle, const Attribute &attribute);

 protected:
  IndexManager();
  ~IndexManager();

 private:
  static IndexManager *_index_manager;
};
```

RC createFile(const string &fileName)

This method creates a B+tree index file with the given index name.

RC destroyFile(const string &fileName)

This method should delete the index file whose name is fileName.

RC openFile(const string &fileName, IXFileHandle &ixfileHandle)

This method will open the index file. If the open call is successful, an IXFileHandle object for the given index is returned. This ixfileHandle is passed to the insertEntry(), deleteEntry(), and scan() methods to insert/delete/scan entries in the index file.

RC closeFile(IXFileHandle &ixfileHandle)

This method closes the index file indicated by the ixfileHandle.

*RC insertEntry(IXFileHandle &ixfileHandle, const Attribute &attribute, const void *key, const RID &rid)*

This method should insert a new entry, a <key,rid> pair, into the index file. The second parameter is the attribute descriptor of the key, and the parameter rid identifies the record that will be paired with the key in the index file. (An index contains only the records' ids, not the records themselves.) The format for the passed-in key is the following: (1) For INT and REAL: use 4 bytes; (2) For VARCHAR: use 4 bytes for the length followed by the characters. Note that the given key value doesn't contain Null flags. In our system we won't store nulls in the index, so you can safely assume **all the key data is not null.** Note, though, that this rules out index-only query plans -- real systems that support them do enter ALL records, even the null ones, into their indexes for completeness.

An overflow can happen if we try to insert an entry into a fully occupied node. As we discussed in the B+ tree lectures, you are required to restructure the tree shape.

*RC deleteEntry (IXFileHandle &ixfileHandle, const Attribute &attribute, const void *key, const RID &rid)*

This method should delete the entry for the <key,rid> pair from the index. You should return an error code if this method is called with a non-existing entry. The format of the key is the same as the format that is given to insertEntry() function.

*RC scan(IXFileHandle &ixfileHandle, const Attribute &attribute, const void *lowKey, const void *highKey, bool lowKeyInclusive, bool highKeyInclusive, IX_ScanIterator &ix_ScanIterator);*

This method should initialize a condition-based scan over the entries in the open index. Once underway, by calls to IX_ScanIterator::getNextEntry(), the iterator should incrementally produce the entries for all records whose indexed attribute key falls into the range specified by the lowKey, highKey, and inclusive flags. If lowKey is a NULL pointer, it can be interpreted as -infinity. If highKey is NULL, it can be interpreted as +infinity. Otherwise the format of the parameter lowKey and highKey is the same as the format of the key in IndexManager::insertEntry().

void printBtree(IXFileHandle &ixfileHandle, const Attribute &attribute);

This method should print out the B+ tree structure in a pre-order fashion. We use it to "eyeball" the shape of your B+tree in some of our test cases. So, it is important to implement this function correctly.

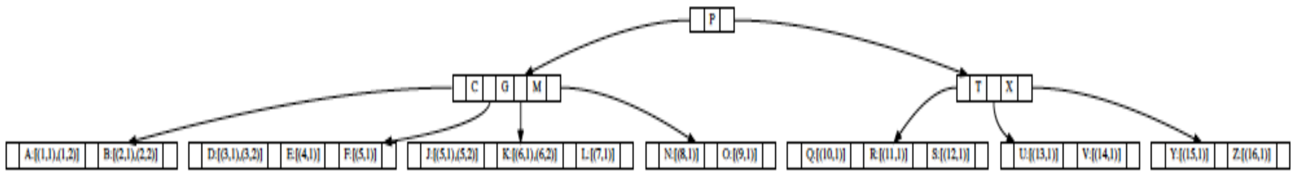
You should print your BTree as a valid JSON record to make visualization easier. To do so, you will need to traverse the B+tree nodes depth-first to print them in pre-order. You should print out all the keys in each node, and also its children, in JSON format. The whole tree should be serialized as one JSON object ({}), and each internal node should have two elements, "keys" and "children". The value of "keys" is an array of the actual indexed keys. The value of "child" is an array of child tree nodes. The leaf nodes will only have "keys", and the value of "keys" should include the associated RidLists. Here is an example of the desired output result:

```
{
"keys":["P"],
"children":[
    {"keys":["C","G","M"],
     "children": [
        {"keys": ["A:[(1,1),(1,2)]","B:[(2,1),(2,2)]"]},
        {"keys": ["D:[(3,1),(3,2)]","E:[(4,1)]","F:[(5,1)]"]},
        {"keys": ["J:[(5,1),(5,2)]","K:[(6,1),(6,2)]","L:[(7,1)]"]},
        {"keys": ["N:[(8,1)]","O:[(9,1)]"]}
    ]},
    {"keys":["T","X"],
     "children": [
        {"keys": ["Q:[(10,1)]","R:[(11,1)]","S:[(12,1)]"]},
        {"keys": ["U:[(13,1)]","V:[(14,1)]"]},
        {"keys": ["Y:[(15,1)]","Z:[(16,1)]"]}
    ]}
]
}
```

This is a simple JSON document, so you should be able to print out the tree this way in text format. (We do NOT suggest using an extra JSON library.) There are quite a few online JSON validators on the web; here's an example. You can copy and paste your printed results there to validate them. A useful feature of that validator is that if your tree gets too big to look at all at once, you can collapse part of the tree that you don't care about, which may help you focus on debugging the parts of the tree that aren't correct.

[Not required, but visualization could help you: If you want to visualize your BTree picture, you are welcome to try this tool. Please follow its instructions to convert your JSON file into a picture. (You need to install the python2.7 and the graphviz on your machine. Please file an issue on that Github repo if you encounter any problems.)

You can also copy and paste your JSON text or upload it to this website tool to generate and review your B+tree. A screen shot for the above tree will look like:



See tree3.svg, which is a file under Project3, for a clearer picture. Browsers can open svg files.]

2. IX_ScanIterator Class

The IX_ScanIterator class is used to perform condition-based scans over the entries of an index file. Note: as in the RM_ScanIterator and RBFM_ScanIterator, the IX_ScanIterator should not cache the scan result in memory. In fact, your code should be looking only at one (or a few) page(s) of data at a time when getNextEntry() is called. In this project, once again we let the OS do all the memory management work for you.

```
class IX_ScanIterator {
 public:
  IX_ScanIterator();
  ~IX_ScanIterator();

  // Get next matching entry
  RC getNextEntry(RID &rid, void *key);

  // Terminate index scan
  RC close();
};
```

RC getNextEntry(RID &rid, void *key)

This method should set its output parameters rid and key to be the RID and key, respectively, of the next record in the index scan. This method should return IX_EOF if there are no index entries left satisfying the scan condition. You may assume that IX component clients will not close the corresponding open index while a scan is underway. **All keys should be printed in ascending order.**

RC close()

This method should terminate the index scan.

3. IXFileHandle

This class hides (i.e., provides layered access to) the PF file handles of a given index. The IndexManager API uses an IXFileHandle to operate on an index, so users of the IX layer do not see or directly interact with PF FileHandle(s). One thing you will need to make sure is that an IXFileHandle provides a collectCounterValues() method that collects and aggregates performance data from the index's associated PF FileHandles.

```
class IXFileHandle {

public:

    // variables to keep counter for each operation
    unsigned ixReadPageCounter;
    unsigned ixWritePageCounter;
    unsigned ixAppendPageCounter;

    // Constructor
    IXFileHandle();

    // Destructor
    ~IXFileHandle();

    // Put the current counter values of associated PF FileHandles into variables
    RC collectCounterValues(unsigned &readPageCount, unsigned &writePageCount, unsigned
&appendPageCount);

}
```

Implementation Details

- The data types for the index attribute that must be supported by the Index Manager are: Int, Real, and VarChar. **You must handle space nicely for VarChar keys.**

- You are free to use alternative design ideas to those suggested here if you believe your ideas will improve the structure or performance of your code. The only thing that you must not alter is the interface itself, although you are free to extend it, if you want.
- The three fundamental B+ tree operations -- search (which extends to scan), insertion, and deletion -- vary quite a bit in their implementation complexity. We suggest that you get search and insertion running first, and then worry about deletion., which we're simplifying.
- Implementing a completely correct delete operation in B+ trees turns out to be quite difficult. As a simplification, we recommend that you implement lazy deletion. In this approach, when an entry is deleted, then even if it causes a leaf page to become less than half full, no redistribution or node merging takes place -- the underfilled page remains in the tree as is. Regardless of which approach you use, deletion must always work. That is, after an IX component client asks for an entry to be deleted, that deleted entry should not appear in a subsequent index search (scan).
- You do not have to support having arbitrary number of entries that share the same key and that span multiple pages. However, you should support an "arbitrary" number of entries that share the same key as long as they fit within a single page (e.g., an INT key with three different rids).
- Index scans will be used by higher-level components when executing selection, join, and delete operations, as well as when performing update operations on attributes other than the indexed attribute. Thus each index entry scanned will either be used to fetch (and possibly update) a record, or to delete a record. (The insert operation inserts one record at a time.) While making an index scan work correctly for selection, join, and non-index-key update operations is relatively straightforward, deletion operations are more complicated, even when using the simplified approach to deletion described above. You must ensure that it is possible to use an index scan to find and then delete all index entries satisfying a condition. That is, the following client code segment should work:

```
IX_ScanIterator ix_ScanIterator;
indexManager->scan(ixfileHandle, ..., ix_ScanIterator);
while ((rc = ix_ScanIterator.getNextEntry(rid, &key)) != IX_EOF)
{
   indexManager->deleteEntry(ixfileHandle, attribute, &key, rid);
}
```

You may assume that during a deletion scan, no other index entries will be inserted or deleted, and that no other retrieval scans will be underway.

Submission Instructions

The following are requirements on your submission. Points may be deducted if they are not followed.

- Write a report to briefly describe the design and implementation of your index module. Refer to the report template file (project3_report.txt)in the codebase for the contents that you need to include.
- You need to submit the source code under the "rbf", "rm", and "ix" folder. Make sure you do a "make clean" first, and do NOT include any useless files (such as binary files and data files). Your makefile should make sure the test files, ixtestXX.cc compile and run properly.
- Please organize your project in the following directory hierarchy: project3-**teamID** / codebase / {rbf, rm, ix, makefile.inc, readme.txt, project3-report.txt} where rbf, rm, and ix folders include your source code and makefiles.
- Compress project3-**teamID** into a SINGLE zip file. Each team only submits one file, with the name "project3-teamID.zip".
- Put the project3 test.sh and the zip file under the same directory. Run it to check whether your project can be properly unzipped and tested (use your own makefile.inc and the ixtest*.cc when you are testing the script). If the script doesn't work correctly, it's most likely that your folder

organization doesn't meet the requirement. Our grading will be automatically done by running the test script, which runs the ixtest*.cc tests. The usage of the script is:

```
./test.sh ''project3-teamID''
```

Q & A

- **Q:** Can we assume that each page (page) may hold a different number of entries if key size is variable?
  A: Yes. Please try to maximize the space utilization of each page as much as possible.

- **Q:** Can an index node span multiple pages? If we are storing the key in a node and the key is of type VarChar, are we guaranteed that all keys will fit into a single node?
  A: An index node should occupy exactly one page (including its free space). You can assume each entry (key and RID pair) fits into one node.

- **Q:** Tree pages are supposed to have between d and 2d entries on them, where d is the maximum number of entries that can appear on a page. What is the value of d for this project? Should we assume d = 2?
  A: The notion of d depends on the size of each page (currently 4KB) and the type/size of the key value; with variable-length data, there is no actual "constant" d; whatever fits, fits. If there’s no room for an insertion on a node, you need to handle that either by splitting or by sharing with a sibling (as described in class and in textbook). Remember that the tree must always have constant depth for all leaves.

  In a product implementation, you would also have to handle deletions properly (as described in class and textbook) if a node became less than half-full. But as we described above (“lazy deletion”), although you have to perform the deletion, you can allow nodes that become “less than half-full” to stay that way.

- **Q:** We're expected to deal with multiple entries having the same key. How should we deal with the fact that we may need to split a leaf node full of entries all with the same key? I don't see any easy solution to this problem (we would end up with two entries in the parent node having the same key). Should we just assume that this will not happen?
  A: Yes, assume that this won't happen, but please do detect it and throw an error if it does.

Testing

Please use the provided test files included in the codebase to test your code. Note that this file will be used to grade your project partially since we may also have our own private test cases. This is by no means an exhaustive test suite. Please feel free to add more cases to this, and test your code thoroughly. The test code includes provisional points for each test case. The points are subject to change if needed.