Project 3 - Design and Implementation Summary

CS222: Principles of Data Management

Christopher A. Wood (woodc1 - 54168394)
Tamir Husain (thusain - 40206791)
March 03, 2014

Task

Write a report to briefly describe the design and implementation of your index component.

Summary

This document describes the new design and RBF refactor needed for the index component. We first discuss how the record-based management functionality was generalized to be used by both the RBF and IX components, and then present a detailed discussion of the page and record formats used to implement the required B+ tree functionality.

Record-Based Management Refactor

Given the similar interfaces shared by the IX and RBF components, and the correctly implemented code in the RBF component from parts 1 and 2 of the project, we decided to refactor our RBF layer by generalizing all common record-based manipulation to a new component - the RecordBasedCoreManager. The RecordBaseFileManager and IndexManager classes now inherit from this new class, where the only significant different (from the perspective of on-disk storage) is the contents of the page footers.

In particular, as will be described in the following section, the IndexManager page footer contains data that is absent from the page footer maintained by the RecordBasedFileManager. Therefore, we use a common “core” footer for each page that the RBFM and IX layer extend to add layer specific footer data. The CoreManager only needs to know the final size of the footer so it can properly allocate enough space and is able index into the footer to find the common data by casting to the core footer type.

Index Manager Component Design

The IndexManager stores the entries of each B+ Tree page as records, and stores extra bookkeeping data in the footer of the page. This bookkeeping data includes:

Leaf Flag: true if this page is a leaf, fast if not
First RID: Points to the slot of the smallest record on this page
Parent Page#: Points to the parent page#, used for insertions/deletions
Next Leaf#: Links together leaf pages, used for iteration

Left Child#: In non-leaf pages, the pointer to the child page that has values <= First key

As you can see, not all footer values are used in both leaf/non-leaf pages, but for simplicity they are left in as this allows us to use the same footer for both types of pages.

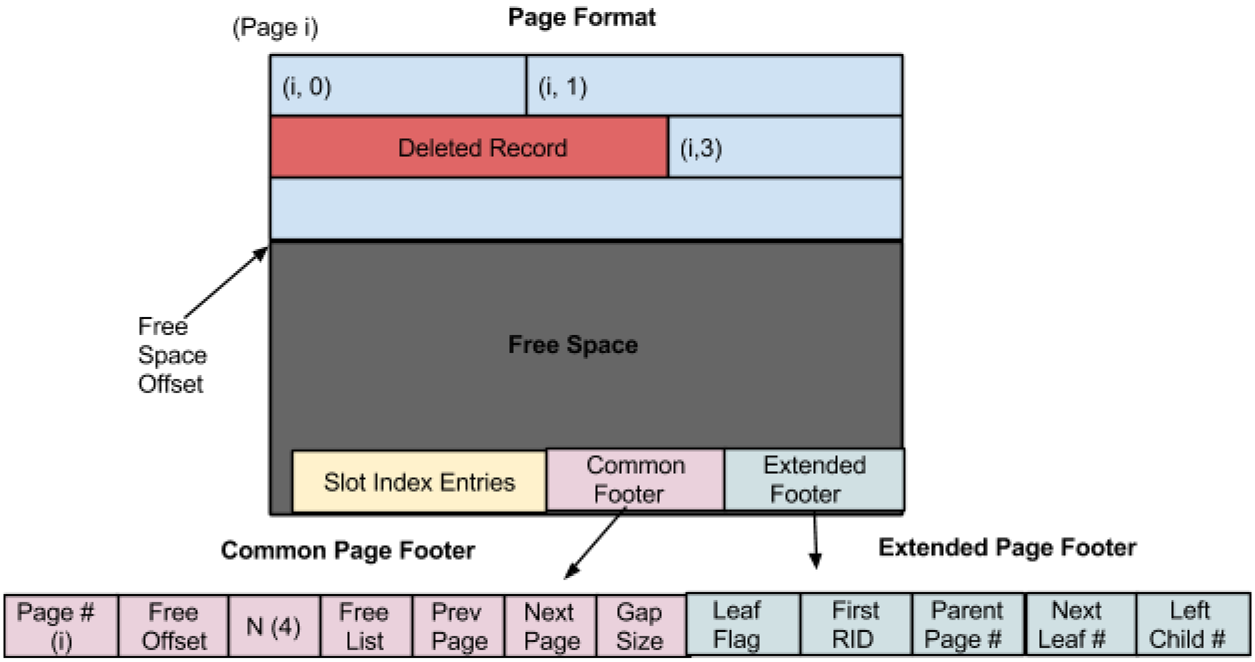

Figure 1. Index page format.

The leaf/non-leaf records themselves are also very similar. Both record types store a value that points to the next slot. This is to allow for out-of-order insertion of keys. The stored RID corresponds to the given data RID on leaf pages, and the >= page pointer on non-leaf pages. Finally, the key is the last item in the record. It will either be a simple integer, real, or a VarChar with corresponding size.

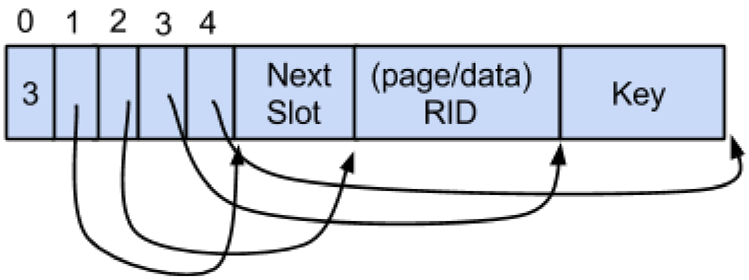

Figure 2. Index entry record format. The contents of the key entry vary based on the type of key (i.e., integer, real, VARCHAR).

The record format we use allows us to easily pair up key/pointer values. In a non-leaf page of a B+Tree we have one more pointer we have to account for, the pointer to values less than the first key. We store this in the footer as stated above, as the left child#.

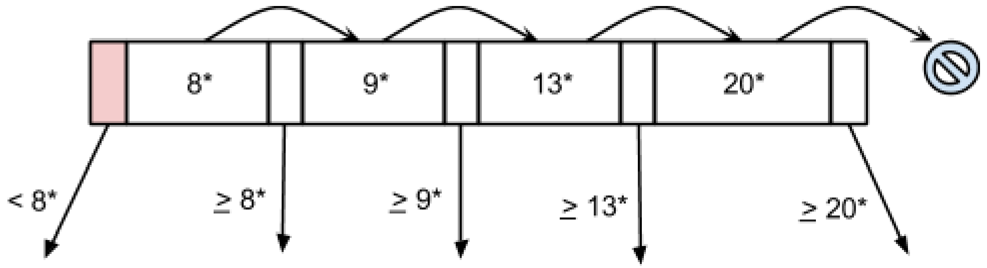

Figure 3. Sample non-leaf index entry (a subset of the contents of a non-leaf page). The red pointer is stored in the extended page header of the page and not in a record.

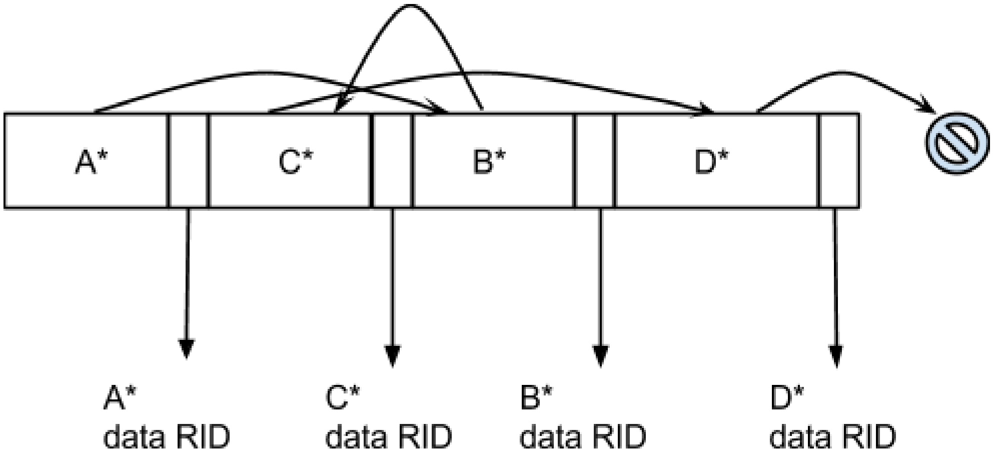

Figure 4. Sample leaf index entry (a subset of the contents of a non-leaf page). The insertion order may have been A*, C*, B*, D* and the nextSlot pointers show the correct ordering still.

Index Manager Implementation

IndexManager::createFile

After creating a reserved page 0 (for freespace bookkeeping data) we initialize page 1 to be our first leaf page, and set it to be the root. We store the page# of the root in our reserved page 0 for retrieval later. Page 0 is a hotspot, and the root page number could be cached and only written out as needed (when the root is split).

IndexManager::(destroyFile, openFile, ,closeFile)

These are simply thin wrappers around the CoreManager, which in turn calls the appropriate PageFileManager function.

IndexManager::insertEntry

After finding the root page, we begin searching for the leaf page where this entry should be placed. We do this by repeatedly iterating through non-leaf page records and stopping when our key is no longer greater than or equal to the key of the record we are comparing against. Once we have found the correct leaf page, if there is enough space on the page we simply do the same type of search and insert the key/data pair as a record in the appropriate location.

If there is not enough space, we must split the leaf page (described below in deletelessSplit). After splitting the page, we must insert the key for the new page into the parent non-leaf page. If the parent non-leaf page has enough free space this is done simply, otherwise the split will continue to cascade up the chain of non-leaf pages until it either no longer has to split or it splits the root.

IndexManager::deletelessSplit

This split method takes care of splitting both leaf and non-leaf pages. It does so by allocating two page buffers, one backed by the input page, and one backed by a newly allocated page. We iterate through the input page, copying records to the new left page until it is half full (by bytes). If we are working with a non-leaf page, we populate the left child pointer with the first element of the right page. We then iterate through the rest of the records and copy them to the new right page, updating the parent pointers of all of these records to point to this new right page. We update the nextPage pointers of leaf pages to ensure the newly split page is linked up for scan iterations.

IndexManager::deleteEntry

The deleteEntry function is very similar to the insertEntry function, it finds the record in the same way as insertEntry. Upon finding the record, we mark it as deleted, and update the previous record’s nextSlot value (if it existed) to point to the correct nextSlot after this deleted record.

We have implemented a lazy deletion method where pages that are less than half full are not merged with siblings or back up into their parents.

IndexManager::scan

In our scan initialization, we must first retrieve the record descriptor for our current index. This is based off of the passed in attribute type. We then will search for where the first record is. If the user provided no key, we will simply follow all left child pointers until we hit a leaf, at the leaf, the first record will be the smallest entry in our index which we will use as the -Infinity key. If the user gives a value, we will search down through the B+Tree (described above) to find the given value (or a value slightly larger if the exact match does not exist). A similar search is done with the high value.

After finding the start and end RIDs, we also iterate one RID forward and store that as our nextRID. We use the nextRID when the user deletes the currentRID during a scan in order to not lose our place.

IX_ScanIterator::getNextEntry

To get the next entry, we must first verify we have not gone out of our high or low bounds. If we have, then we will return immediately. If our high bound is inclusive and we have reached it, we store a simple flag to instruct us on the next getNextEntry() call that it should be our last. If we have a valid current key, we copy that key to the user's buffer, and then advance to the next slot, moving to the next leaf page if necessary.