Lab 3: B+ Tree Index

布置日期: 2022/04/18

截止日期: 2022/05/08, 23:59:59

0. Introduction

In this lab you will implement a B+ tree index for efficient lookups and range scans. We supply you with all of the low-level code you will need to implement the tree structure. You will implement searching, splitting pages, redistributing tuples between pages, and merging pages.

As discussed in class, the internal nodes in B+ trees contain multiple entries, each consisting of a key value and a left and a right child pointer. Adjacent keys share a child pointer, so internal nodes containing *m* keys have *m*+1 child pointers. Leaf nodes can either contain data entries or pointers to data entries in other database files. For simplicity, we will implement a B+tree in which the leaf pages actually contain the data entries. Adjacent leaf pages are linked together with right and left sibling pointers, so range scans only require one initial search through the root and internal nodes to find the first leaf page. Subsequent leaf pages are found by following right (or left) sibling pointers.

1. Getting started

Lab3的作业请在Lab1 & 2的基础上完成。

2. Search

Take a look at `BTreeFile.java`. This is the core file for the implementation of the B+Tree and where you will write all your code for this lab. Unlike the HeapFile, the BTreeFile consists of four different kinds of pages. As you would expect, there are two different kinds of pages for the nodes of the tree: internal pages and leaf pages. Internal pages are implemented in `BTreeInternalPage.java`, and leaf pages are implemented in `BTreeLeafPage.java`. For convenience, we have created an abstract class in `BTreePage.java` which contains code that is common to both leaf and internal pages. In addition, header pages are implemented in `BTreeHeaderPage.java` and keep track of which pages in the file are in use. Lastly, there is one page at the beginning of every BTreeFile which points to the root page of the tree and the first header page. This singleton page is implemented in `BTreeRootPtrPage.java`. Familiarize yourself with the interfaces of these classes, especially `BTreePage`, `BTreeInternalPage` and `BTreeLeafPage`. You will need to use these classes in your implementation of the B+Tree.

Your first job is to implement the `findLeafPage()` function in `BTreeFile.java`. This function is used to find the appropriate leaf page given a particular key value, and is used for both searches and inserts. For example, suppose we have a B+Tree with two leaf pages (See Figure 1). The root node is an internal page with one entry containing one key (6, in this case) and two child pointers. Given a value of 1, this function should return the first leaf page. Likewise, given a value of 8, this function should return the second page. The less obvious case is if we are given a key value of 6. There may be duplicate keys, so there could be 6's on both leaf pages. In this case, the function should return the first (left) leaf page.

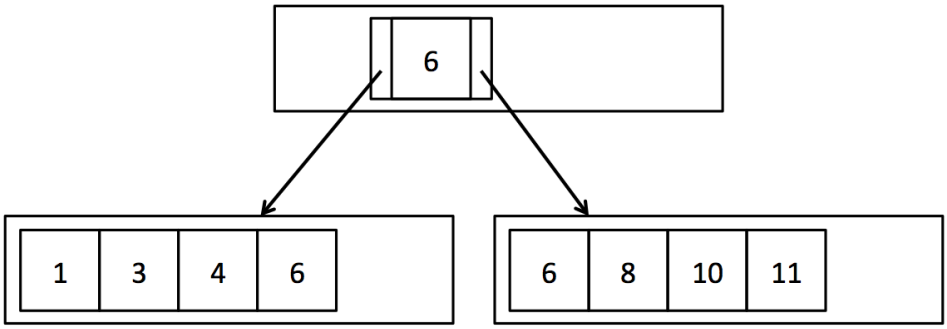


Figure 1: A simple B+ Tree with duplicate keys

Your `findLeafPage()` function should recursively search through internal nodes until it reaches the leaf page corresponding to the provided key value. In order to find the appropriate child page at each step, you should iterate through the entries in the internal page and compare the entry value to the provided key value. `BTreeInternalPage.iterator()` provides access to the entries in the internal page using the interface defined in `BTreeEntry.java`. This iterator allows you to iterate through the key values in the internal page and access the left and right child page ids for each key. The base case of your recursion happens when the passed-in BTreePageId has `pgcateg()` equal to `BTreePageId.LEAF`, indicating that it is a leaf page. In this case, you should just fetch the page from the buffer pool and return it. You do not need to confirm that it actually contains the provided key value f.

Your `findLeafPage()` code must also handle the case when the provided key value f is null. If the provided value is null, recurse on the left-most child every time in order to find the left-most leaf page. Finding the left-most leaf page is useful for scanning the entire file. Once the correct leaf page is found, you should return it. As mentioned above, you can check the type of page using the `pgcateg()` function in `BTreePageId.java`. You can assume that only leaf and internal pages will be passed to this function.

Instead of directly calling `BufferPool.getPage()` to get each internal page and leaf page, we recommend calling the wrapper function we have provided, `BTreeFile.getPage()`. It works exactly like `BufferPool.getPage()`, but takes an extra argument to track the list of dirty pages. This function will be important for the next two exercises in which you will actually update the data and therefore need to keep track of dirty pages.

Every internal (non-leaf) page your `findLeafPage()` implementation visits should be fetched with READ_ONLY permission, except the returned leaf page, which should be fetched with the permission provided as an argument to the function. These permission levels will not matter for this lab, but they will be important for the code to function correctly in future labs.

Exercise 1: BTreeFile.findLeafPage()

Implement `BTreeFile.findLeafPage()`.

After completing this exercise, you should be able to pass all the unit tests in `BTreeFileReadTest.java` and the system tests in `BTreeScanTest.java`.

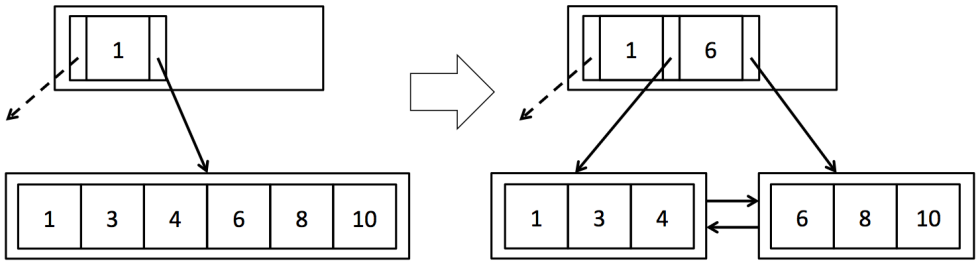
---

3. Insert

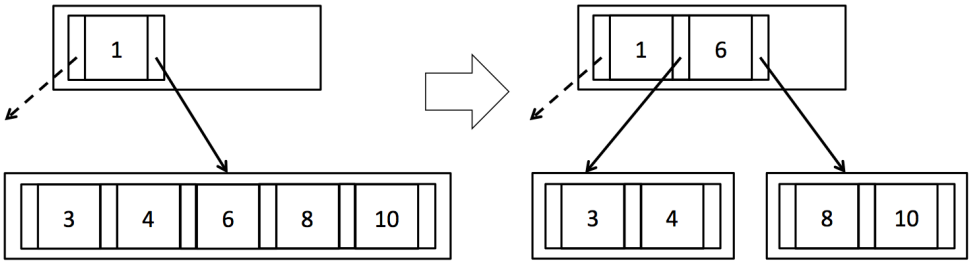
In order to keep the tuples of the B+Tree in sorted order and maintain the integrity of the tree, we must insert tuples into the leaf page with the enclosing key range. As was mentioned above, `findLeafPage()` can be used to find the correct leaf page into which we should insert the tuple. However, each page has a limited number of slots and we need to be able to insert tuples even if the corresponding leaf page is full.

As described in the textbook, attempting to insert a tuple into a full leaf page should cause that page to split so that the tuples are evenly distributed between the two new pages. Each time a leaf page splits, a new entry corresponding to the first tuple in the second page will need to be added to the parent node. Occasionally, the internal node may also be full and unable to accept new entries. In that case, the parent should split and add a new entry to its parent. This may cause recursive splits and ultimately the creation of a new root node.

In this exercise you will implement `splitLeafPage()` and `splitInternalPage()` in `BTreeFile.java`. If the page being split is the root page, you will need to create a new internal node to become the new root page, and update the BTreeRootPtrPage. Otherwise, you will need to fetch the parent page with READ_WRITE permissions, recursively split it if necessary, and add a new entry. You will find the function `getParentWithEmptySlots()` extremely useful for handling these different cases. In `splitLeafPage()` you should "copy" the key up to the parent page, while in `splitInternalPage()` you should "push" the key up to the parent page. See Figure 2 and review section 10.5 in the text book if this is confusing. Remember to update the parent pointers of the new pages as needed (for simplicity, we do not show parent pointers in the figures). When an internal node is split, you will need to update the parent pointers of all the children that were moved. You may find the function `updateParentPointers()` useful for this task. Additionally, remember to update the sibling pointers of any leaf pages that were split. Finally, return the page into which the new tuple or entry should be inserted, as indicated by the provided key field. (Hint: You do not need to worry about the fact that the provided key may actually fall in the exact center of the tuples/entries to be split. You should ignore the key during the split, and only use it to determine which of the two pages to return.)



Splitting a Leaf Page



Splitting an Internal Page

Figure 2: Splitting pages

Whenever you create a new page, either because of splitting a page or creating a new root page, call `getEmptyPage()` to get the new page. This function is an abstraction which will allow us to reuse pages that have been deleted due to merging (covered in the next section).

We expect that you will interact with leaf and internal pages using `BTreeLeafPage.iterator()` and `BTreeInternalPage.iterator()` to iterate through the tuples/entries in each page. For convenience, we have also provided reverse iterators for both types of pages: `BTreeLeafPage.reverseIterator()` and `BTreeInternalPage.reverseIterator()`. These reverse iterators will be especially useful for moving a subset of tuples/entries from a page to its right sibling.

As mentioned above, the internal page iterators use the interface defined in `BTreeEntry.java`, which has one key and two child pointers. It also has a recordId, which identifies the location of the key and child pointers on the underlying page. We think working with one entry at a time is a natural way to interact with internal pages, but it is important to keep in mind that the underlying page does not actually store a list of entries, but stores ordered lists of *m* keys and *m*+1 child pointers. Since the `BTreeEntry` is just an interface and not an object actually stored on the page, updating the fields of `BTreeEntry` will not modify the underlying page. In order to change the data on the page, you need to call `BTreeInternalPage.updateEntry()`. Furthermore, deleting an entry actually deletes only a key and a single child pointer, so we provide the funtions `BTreeInternalPage.deleteKeyAndLeftChild()` and `BTreeInternalPage.deleteKeyAndRightChild()` to make this explicit. The entry's recordId is used to find the key and child pointer to be deleted. Inserting an entry also only inserts a key and single child pointer (unless it's the first entry), so `BTreeInternalPage.insertEntry()` checks that one of the child pointers in the provided entry overlaps an existing child pointer on the page, and that inserting the entry at that location will keep the keys in sorted order.

In both `splitLeafPage()` and `splitInternalPage()`, you will need to update the set of `dirtypages` with any newly created pages as well as any pages modified due to new pointers or new data. This is where `BTreeFile.getPage()` will come in handy. Each time you fetch a page, `BTreeFile.getPage()` will check to see if the page is already stored in the local cache (`dirtypages`), and if it can't find the requested page there, it fetches it from the buffer pool. `BTreeFile.getPage()` also adds pages to the `dirtypages` cache if they are fetched with read-write permission, since presumably they will soon be dirtied. One advantage of this approach is that it prevents loss of updates if the same pages are accessed multiple times during a single tuple insertion or deletion.

Note that in a major departure from `HeapFile.insertTuple()`, `BTreeFile.insertTuple()` could return a large set of dirty pages, especially if any internal pages are split. As you may remember from previous labs, the set of dirty pages is returned to prevent the buffer pool from evicting dirty pages before they have been flushed.

Exercise 2: Splitting Pages

Implement `BTreeFile.splitLeafPage()` and `BTreeFile.splitInternalPage()`.

After completing this exercise, you should be able to pass the unit tests in `BTreeFileInsertTest.java`. You should also be able to pass the system tests in `systemtest/BTreeFileInsertTest.java`. Some of the system test cases may take a few seconds to complete. These files will test that your code inserts tuples and splits pages correcty, and also handles duplicate tuples.

---

4. Delete

In order to keep the tree balanced and not waste unnecessary space, deletions in a B+Tree may cause pages to merge (see Figure 3). You may find it useful to review section 10.6 in the textbook.

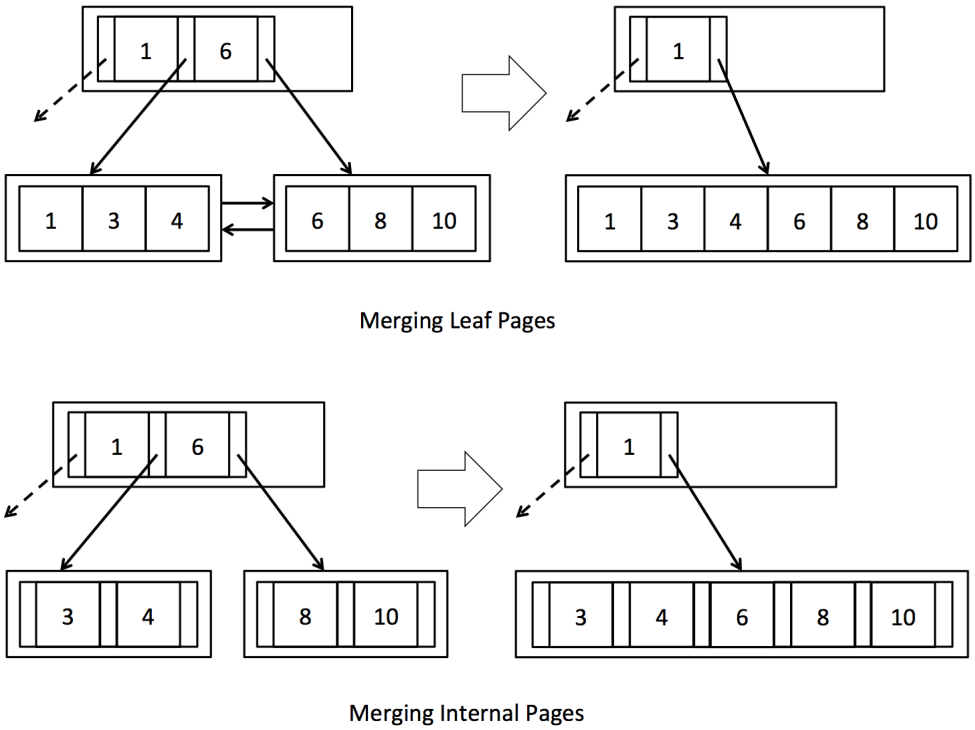
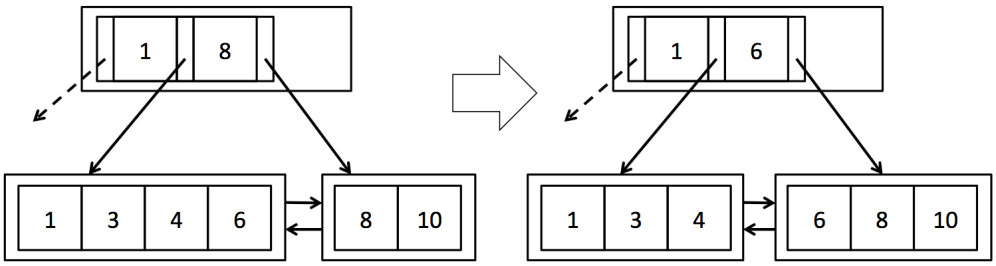
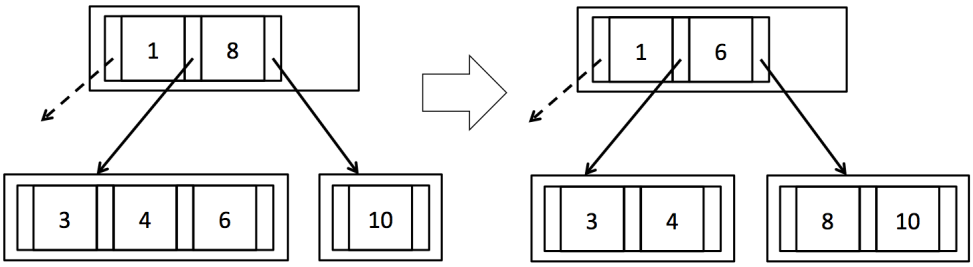


Figure 3: Merging pages

As described in the textbook, attempting to delete a tuple from a leaf page that is less than half full should cause that page to either steal tuples from one of its siblings or merge with one of its siblings. If one of the page's siblings has tuples to spare, the tuples should be evenly distributed between the two pages, and the parent's entry should be updated accordingly (see Figure 4). However, if the sibling is also at minimum occupancy, then the two pages should merge and the entry deleted from the parent. Occasionally, deleting an entry from the parent will cause the parent to become less than half full. In this case, the parent should steal entries from its siblings or merge with a sibling. This may cause recursive merges or deletion of the root node if the last entry is deleted from the root node.



Redistributing Leaf Pages



Redistributing Internal Pages

Figure 4: Redistributing pages

In this exercise you will implement `stealFromLeafPage()`, `stealFromLeftInternalPage()`, `stealFromRightInternalPage()`, `mergeLeafPages()` and `mergeInternalPages()` in `BTreeFile.java`. In the first three functions you will implement code to evenly redistribute tuples/entries if the siblings have tuples/entries to spare. Remember to update the corresponding key field in the parent (look carefully at how this is done in Figure 4 - keys are effectively "rotated" through the parent). In `stealFromLeftInternalPage()` / `stealFromRightInternalPage()`, you will also need to update the parent pointers of the children that were moved. You should be able to reuse the function `updateParentPointers()` for this purpose.

In `mergeLeafPages()` and `mergeInternalPages()` you will implement code to merge pages, effectively performing the inverse of `splitLeafPage()` and `splitInternalPage()`. You will find the function `deleteParentEntry()` extremely useful for handling all the different recursive cases. Be sure to call `setEmptyPage()` on deleted pages to make them available for reuse. As with the previous exercises, we recommend using `BTreeFile.getPage()` to encapsulate the process of fetching pages and keeping the list of dirty pages up to date.

Exercise 3: Redistributing and merging pages

Implement `BTreeFile.stealFromLeafPage()`, `BTreeFile.stealFromLeftInternalPage()`, `BTreeFile.stealFromRightInternalPage()`, `BTreeFile.mergeLeafPages()` and `BTreeFile.mergeInternalPages()`.

After completing this exercise, you should be able to pass the unit tests in `BTreeFileDeleteTest.java`. You should also be able to pass the system tests in `systemtest/BTreeFileDeleteTest.java`. The system tests may take several seconds to complete since they create a large B+ tree in order to fully test the system.

---

5. Extra Credit

Bonus Exercise 4: (10% extra credit)

Create and implement a class called `BTreeReverseScan` which scans the `BTreeFile` in reverse, given an optional `IndexPredicate`.

You can use `BTreeScan` as a starting point, but you will probably need to implement a reverse iterator in `BTreeFile`. You will also likely need to implement a separate version of `BTreeFile.findLeafPage()`. We have provided reverse iterators on `BTreeLeafPage` and `BTreeInternalPage` which you may find useful. You should also write code to test that your implementation works correctly. `BTreeScanTest.java` is a good place to look for ideas.

---

6. Logistics

6.1. 提交说明

提交要求和Lab1一致。注意源代码直接把项目打包并命名为“学号_姓名_lab3.zip”，报告导出 pdf 并命名为“学号_姓名_lab3.pdf”，发送到邮箱 xuhongyuan@dbis.nankai.edu.cn，邮件的主题为“数据库上机作业_学号_姓名_lab3”（此邮箱对主题包含“数据库上机作业”开启自动回复）

代码 push 到 GitLab 的截止时间为5月8日周日23:59:59，代码和报告发送到邮箱的截止时间为5月9日周一23:59:59。晚交的同学会酌情扣分。

6.2. 分数组成

在跑测试代码的时候，我们会从 GitLab 上拉取大家提交后的代码，并将 build.xml 测试代码替换成最初的版本。如果你对程序的 API 做了任何改动导致无法通过测试，一定要在作业报告和代码讲解的过程中主动提出来。

- 50% 源代码（通过测试，代码风格）
- 10% Git Commit History（截图放在报告中）
- 20% 作业报告（思考题，设计思路，重难点，改动部分）
- 20% 代码讲解

代码讲解有**期中**和期末两次，安排在lab2和 lab4 完成后的**一周**内，到时候会提前通知大家预约时间。