

Project 05: B+ Tree Indexes

Project Overview

In this project, you will complete an implementation of the B+ tree index structure to support faster equality scans. Optionally, you can “close the loop” and incorporate your index structure in your join optimizer.

Background

In the `edu.caltech.nanodb.storage.btreefile` package, you will find a basic, unfinished implementation of a B+ tree tuple file. This implementation follows the description given in class, where the B+ tree is a table file that also supports indexes. This allows us to more easily perform file-scans over indexes without significant changes to the file-scan code paths.

NanoDB’s `CREATE TABLE` command allows specifying the storage format to use for each table:

```
CREATE TABLE bt (  
  a INTEGER,  
  b VARCHAR(20)  
) PROPERTIES (storage = 'btree');  
  
INSERT INTO bt VALUES (1, 'abc');  
INSERT INTO bt VALUES (2, 'xyz');  
INSERT INTO bt VALUES (1, 'def');
```

Since the B+ tree format keeps its records in sequential order, when we “`SELECT * FROM bt;`”, the records (should) come back sorted on all columns. (Right now, the implementation is unfinished, so this won’t work.)

Given this design, an index is simply a table that is built against another table, containing a subset of that table’s columns, and including an extra column that holds file-pointers to tuples in the referenced table. Rows in the index are populated from the referenced table. Finally, our indexes are B+ tree files to support fast equality-based/range-based lookups.

We can create a table and index as follows:

```
CREATE TABLE t (  
  a INTEGER,  
  b VARCHAR(30),  
  c FLOAT  
);  
CREATE INDEX i ON t (a);
```

Under the hood (see the `datafiles` directory), we have multiple files. In addition to the table file “`t.tbl`”, we have an index file “`t_i.idx`” with the schema (`t.a: INTEGER, t.#TUPLE_PTR: file-pointer`), with one row for every row in `t`. (Index names can be repeated across tables, so the file name needs to include both.)

Implementation Quirks

Some additional notes on NanoDB's B+ tree implementation:

- “Fullness” of a node is not determined by the number of entries/pointers, but instead by the number of bytes used in the node.
 - Leaf nodes and internal nodes can store different numbers of entries/pointers, since their structures are slightly different.
 - When a node is split, to avoid a full-page scan, the number of entries/pointers is divided in half, and one half is moved to a sibling node. This means that we will *usually* satisfy the “at least half full” rule, but some nodes may not. (This is fine since they should be close, though.)
- When relocating entries/pointers from a node to a sibling, only enough entries are relocated to allow the new entry to be added to either node. There is no attempt to “balance” the number of entries between the pair of nodes.
 - We try to make space for the new entry to be added to either node because when relocating or splitting, we don't necessarily know which sibling the new entry will end up in.
- Other details are as discussed in class, including:
 - Leaf nodes form a singly linked list.
 - Inner nodes only reference other nodes in the tree.
 - No additional structure is maintained beyond this; in particular: nodes do not reference parents, and leaves do not reference previous siblings.

Implementation Classes

The structure is somewhat similar to the heap file you worked with at the beginning of term, with a few natural differences:

- The [BTreeTupleFile](#) class provides most of the operations for accessing or modifying tuples in a B+ tree file. It performs high-level operations like searching for a particular leaf.
- File-manipulation tasks are delegated to two classes, [InnerPageOperations](#) and [LeafPageOperations](#). A third class, [FileOperations](#) is responsible for generic file operations, like finding new empty pages in the file.
- The [InnerPage](#) and [LeafPage](#) wrapper classes wrap a [DBPage](#) to more easily manipulate node contents.
- The [BTreeTupleFileManager](#) class provides file-level operations, such as creating a new B+ tree file, storing metadata, etc.

Keys and Indexes

NanoDB can be configured to automatically create indexes in certain situations. For example, if a table is declared with a primary key or UNIQUE constraints, the database can create indexes on those keys. Foreign keys behave similarly. However, this functionality is initially turned off, since the B+ tree implementation isn't yet functional.

Once you have it working, you can enable this by modifying the `PropertyRegistry.initProperties()` package:

- Set “`nanodb.enableIndexes`” to true to enable indexes in the first place.
- Set “`nanodb.createIndexesOnKeys`” to true to automatically create indexes on primary/unique/foreign keys.

You may want to erase your `datafiles` directory to ensure newly created files use these settings.

B+ Tree Operations

While a lot of the low-level munging is implemented for you, you will have to implement some higher-level operations to complete the B+ tree implementation. Unfortunately, B+ trees have a lot of nitty implementation details. Especially in this project, read the documentation to make sure you know what helper functions are available, and what they do. Additionally, you should feel free to add additional logging statements.

Navigation

All B+ tree operations (adding, removing, searching) require the tree structure to be navigated from root to leaf. This operation is *partially* implemented in `navigateToLeafPage()` in [BTreeTupleFile](#).

Task 1: `BTreeTupleFile#navigateToLeafPage()`

Complete the implementation of this method. Some notes:

- This method only navigates to a leaf page, and returns that leaf page for the caller to use (potentially along with its path).
- A search may be conducted without a uniquifier, and more generally, with any prefix of the index tuple. To account for this, compare tuples using [TupleComparator#comparePartialTuples\(\)](#). Because we use uniquifiers, use `SHORTER_IS_LESS` for the comparison mode.
- Refer to the [InnerPage](#) class for processing the data stored inside internal pages for navigation.

At this point, you can create a table with the B+ tree storage format, insert records into it, and see that the contents of the table always appear in order. However, this table has a maximum size, because NanoDB doesn't yet know how to split a leaf page.

Splitting Leaves

To support B+ tree files with more than one leaf page, the implementation must be able to split a leaf and update the parent node with the newly created pointer. This operation is handled by the private `splitLeafAndAddTuple()` method of the [LeafPageOperations](#) class.

Task 2: `LeafPageOperations#splitLeafAndAddTuple()`

Complete the implementation of this method. The most complicated part will likely be updating the parent of the leaf, but as usual, there are helper functions that make some parts less onerous. Some notes:

- There are helper functions in [LeafPage](#), [LeafPageOperations](#), and [InnerPageOperations](#) that will be useful.
- When calling helpers, the `pagePath` argument must always be the path to the specific page being manipulated. Thus, when handling the parent (inner) page, remember to remove the last element from the `pagePath` list.
- Make sure you handle the very first split of the root node that occurs. There won't be an existing parent node if it's the only node in the tree.

At this point, you can create somewhat larger B+ tree files, with many leaf pages. However, this still has a maximum size, because NanoDB doesn't know how to handle multiple inner pages.

Reorganizing Internal Nodes

Each of the B+ tree operations involving an internal node (redistributing, splitting, and coalescing) involves moving pointers and tuples from a node to a sibling node. Two nodes are siblings if their pointers are separated by one tuple in the same parent node, which may not exist yet. This functionality is provided by the `movePointersLeft()` and `movePointersRight()` methods of the `InnerPage` class.

Recall that every tuple must be sandwiched between two pointers; that is, an inner page contains N pointers and $N - 1$ tuples. If you move M pointers (and the $M - 1$ tuples between them), this will expose a tuple without a pointer on one side. Additionally, the sibling node receiving the pointers and tuples will already have pointers on both sides of all its tuples.

This is where you must figure out how to incorporate the parent node's tuple, if exists. In lecture, we discussed what happens when a single pointer is moved, but this method can move M pointers. You will need to figure out where to store the parent's old tuple, if provided, and what to return as the parent's new tuple.

Task 3: Moving Pointers in InnerPage

Complete the implementation of `movePointersLeft()` and `movePointersRight()` in the `InnerPage` class.

Some notes:

- You can use [PageTuple.storeTuple\(\)](#) to write a key to a `DBPage`.
- You will always return a new tuple in your implementation. You may not receive an old tuple if the root node is being split, since there will not yet be a parent.
- When moving M pointers to the left sibling, the pointers are taken from the start of the node's sequence, and inserted at the end. When moving pointers to the right sibling, they are taken from the end of the node's sequence, and inserted at the start. You'll find [DBPage#moveDataRange\(\)](#) helpful for these sorts of operations.
- **Never** write to the `DBPage`'s internal byte array directly! Always use methods like `setDataRange()`, `write()`, or `writeShort()` so that it can appropriately track whether the page is dirty. (It may be helpful to *read* the byte array directly, however.)

At this point, your B+ tree implementation should be complete.

Using Indexes

Although you have B+ tree files working, the index files still need to be kept in sync with their corresponding tables. NanoDB uses “events” to handle actions that need to occur in response to other actions (command execution and row inserts/updates/deletes). This is in the [EventDispatcher](#) class. Components can implement the [CommandEventListener](#) or the [RowEventListener](#) interface to receive events.

At the end of `StorageManager.initialize()`, the storage manager registers a row event listener called [IndexUpdater](#), which handles all index updates. Any time a table is modified, the index updater applies the appropriate updates to all the table’s indexes.

The `IndexUpdater` handles adding and removing tuples on a table’s indexes. Updates are currently modeled as a deletion followed by an insertion, which is not optimal but is good enough. There are two methods you will complete on this class:

`IndexUpdater#addRowToIndexes()`

Called when a row is inserted or updated. Iterates through the table’s indexes, constructs a suitable tuple, and adds it to the tuple-file for each index.

`IndexUpdater#removeRowFromIndexes()`

Called when a row is updated or deleted. Iterates through the table’s indexes, removing the corresponding tuple from each index. Throws an `IllegalStateException` if the index doesn’t actually contain the tuple being removed.

Task 4: Index Updates

Implement these two methods. [IndexUtils](#) has some methods that will be helpful:

- [makeTableSearchKey\(\)](#), to convert a table tuple to an index tuple. Note the argument `findExactTuple`.
- [findTupleInIndex\(\)](#), to attempt to search for a tuple in the index file with particular values.

Extra Credit

Closing the Loop

Unfortunately, we didn't have enough time this term to make "actually using the indexes in planning and execution" part of the projects. So, we'll leave the high-level steps here as some additional tasks for you to try your hand at and see how far you get.

We've added a new kind of plan node: the `IndexScanNode`. This works with other code in the `indexes` package to allow a query to start at a specific point within an index, and look up tuples in the indexed table based on the query's predicate. You can incorporate this functionality into the `SimplePlanner` or `CostBasedJoinPlanner` to use indexes where available. This involves a few distinct tasks:

- The `ANALYZE` command doesn't analyze indexes associated with the table being analyzed. Additionally, there is no analysis functionality provided for B+ tree files.
- The `IndexScanNode` includes no stats or plan costing model.
- A planner needs to check the available indexes to see if a predicate can be evaluated using them.

Extra Credit: B+ Tree Storage Optimization

Include indexes in your planner. The above tasks will need to be completed approximately in order.

Don't feel like you need to finish *all* of the above tasks to receive any extra credit. We'll give some for varying degrees of completion. (Though we'd be excited if you get all of it working, because it's really cool to go back and actually make NanoDB use B+ trees, especially for primary keys!).

Design Document

Design Doc

Answer the questions in the file `doc/05-b-plus-trees.md` in the document itself.

Submitting

Once you are finished, make sure your code is pushed to GitLab. Then, submit the commit hash you would like to be reviewed on Gradescope.

For more details, see the Project Submission instructions on the course website.