Assignment 6: B$^+$ Tree Indexes

In this assignment you will get to explore an implementation of the B$^+$ tree index structure. Tasks to complete are as follows:

- Complete a B$^+$ tree tuple-file implementation.
- Use this functionality to manage indexes against table files.

Overview

In the `edu.caltech.nanodb.storage.btreefile` package you will find a basic implementation of a B$^+$ tree tuple file. This implementation follows the description given in class, with a few important differences. Possibly the most important one is that the B$^+$ tree implementation can be used to store tables as well as indexes. This is an important design choice, because it makes it very easy to perform file-scans over indexes without having to make any substantial changes to the file-scan code paths.

Given this design, an index becomes a table that is built against another table, using some subset of the referenced table's columns, and including an additional column that holds a file-pointer to tuples in the referenced table. Rows in the index are populated from the referenced table. Finally, the index uses some file organization that facilitates equality-based and/or range-based lookups against the index.

To illustrate, we can create a table and an index as follows:

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

CREATE INDEX i ON t (a);
```

Under the hood, we will end up with a tuple file "`T.tbl`" that has a schema (t.a : INTEGER, t.b : VARCHAR(30), t.c : FLOAT). Additionally, we will have an index file called "`T_I.idx`" that has the schema (t.a : INTEGER, t.#TUPLE_PTR : file-pointer), with one row for every row in table t. (Recall that index names are unique on a per-table basis, but two different tables can have indexes with the same name. Thus, we must use the table's name as part of the index's filename.)

Here are some additional notes on NanoDB's B$^+$ tree implementation:

- "Fullness" of a node is not determined by the number of pointers or entries in the node, but rather by the number of bytes used in the node. Leaf and inner nodes can store different numbers of entries and/or pointers, since their structures are slightly different. When a node is split, the implementation simply divides the number of pointers or entries in half, and moves half of the entries to a sibling node. This means that we will generally satisfy the "at least half-full" rule, but there will likely be nodes here and there that do not satisfy this constraint. (But, they will be close.)

- When relocating entries or 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 "even out" 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 than that, the implementation follows the description in class almost exactly.

- Each leaf node references the next leaf in the B$^+$ tree, forming a linear sequence of leaves.
- Inner nodes only reference other nodes in the tree (thus, they use an unsigned short for these page-pointers).
- No additional structure is maintained beyond that described in the lecture slides. Nodes do not reference their parents in the index structure. Leaves do not reference their previous sibling, only their next sibling. Inner nodes only reference nodes deeper in the structure. (The reasons for this will be explored in the design document.)

Important Implementation Classes

Here are the major components in this B$^+$ tree implementation. You will notice that it is mostly similar to the heap file implementation, with a few obvious differences due to the implementation details.

The `BTreeTupleFile` class provides most of the operations for accessing or modifying tuples in a B$^+$ tree file. It delegates many file-manipulation tasks to two classes, `InnerPageOperations` and `LeafPageOperations`, but it does perform some of the most basic operations such as looking up a leaf-entry in the file based on a search-key, or finding new empty pages in the file when more data needs to be stored.

The `InnerPageOperations` and `LeafPageOperations` classes handle larger-scale tasks like inserting entries into B$^+$ tree nodes, splitting nodes, and relocating entries between nodes. If you review this code, you will note that the implementations are very similar, but *just* different enough to force two separate implementations. (Oh well.)

These two classes also use the `InnerPage` and `LeafPage` wrapper-classes to manipulate individual B$^+$ tree pages. Each of these classes is used to wrap a `DBPage` object, allowing the contents of the node to be manipulated more easily.

Finally, the `BTreeTupleFileManager` class provides file-level operations such as creating a new B$^+$ tree file, storing the metadata, and so forth.

Keys and Indexes

NanoDB will automatically create indexes in certain situations. For example, if a table is declared with a primary key, or one or more `UNIQUE` constraints, the database will automatically create unique indexes on these keys. Therefore, you can also create indexes by issuing commands like:

```
CREATE TABLE t (
    a INTEGER PRIMARY KEY,
    b VARCHAR(30) UNIQUE,
    c FLOAT
);
```

(Foreign key constraints also cause NanoDB to create indexes, but we didn't have time to test and debug the implementation.)

Index Management Mechanism

Indexes must be checked and updated anytime a table is changed. To facilitate this, NanoDB fires events before and after commands are executed, and also before/after any row is inserted, updated, or deleted. The implementation for this mechanism is in the `edu.caltech.nanodb.server` package, in the `EventDispatcher` class. Components can implement the `CommandEventListener` interface to receive before/after command events, or the `RowEventListener` interface to receive before/after insert/update/delete events. Such listeners would then be registered on the `EventDispatcher` singleton to receive notifications when these events occur.

When the Storage Manager is initialized, it registers a row-event listener called the `IndexUpdater` (in package `edu.caltech.nanodb.indexes`), which takes care of index updates. Anytime a table is modified, the index-updater goes through the table's indexes, applying the appropriate updates.

Final Notes

The B+ tree and index code in NanoDB is still pretty new, so there are definitely still some bugs in it. While deleting a tuple from a B+ tree is supported, there are still a few lurking bugs, and the tests that exercise deletion from B+ trees are disabled for HW6. Similarly, although there are a number of index tests, most of these are disabled for HW6.

Also, if you are trying to see how to delegate tasks among your teammates: parts 1 and 2 can be completed in parallel, but at least the first step of part 1 must be working before you can start testing anything in part 2. Within each part, each step requires the previous steps to work before it will work, but you can still work on them in parallel. The analysis in part 3 can be started once the first few tasks in part 1 have been completed.

Part 1: Complete Missing B+ Tree Operations

The B+ tree implementation you have been given is missing several important pieces, which you must implement. Those pieces are outlined in this section.

To test your implementation, you can create tables using the "`btree`" storage format, which corresponds to the B+ tree implementation. For example, you can do this:

```
CREATE TABLE bt (
    a INTEGER
) PROPERTIES (storage = 'btree');

INSERT INTO bt VALUES (53);
INSERT INTO bt VALUES (21);
INSERT INTO bt VALUES (65);
...
```

As you get your implementation working, you should be able to "`SELECT * FROM bt`" and see the tuples always produced in order. Note that the tuples are sorted by all columns.

You can also use the `VERIFY` command to check your table for structural issues.

```
VERIFY bt;
```

This command will check the table for any issues, along with any indexes built against the table. All problems that are encountered will be printed out to the console.

1. All operations – adding a tuple, removing a tuple, or searching for tuples – require the B+ tree structure to be navigated from root to leaf. This operation is partially implemented in the `navigateToLeafPage()` method of the `BTreeTupleFile`. **You will need to complete this implementation.**

Note that this method only navigates the inner-page structure of the index until it reaches a leaf, and then the leaf page is returned to the caller. What happens after that depends on the specific operation being performed.

Also, all key-comparisons should be performed with the `comparePartialTuples()` method of the `TupleComparator` class (`edu.caltech.nanodb.expressions` package). This method allows tuples of different lengths to be compared, which allows us to search on any prefix of the tuple file's columns, not just the full set of columns. (It will also allow us to find tuples in indexes without specifying the file-pointer at the end of the search-key.)

Once this function is finished, you should be able to create a table like the one above, insert records into it, and see that the contents of the table always appear in order. However, if your table gets large enough to require two leaf pages, the implementation will fail. The reason is that NanoDB doesn't yet know how to split a leaf page into two leaves. Continue to the next step…

2. To support B$^+$ tree files larger than one leaf page, the implementation must be able to split a leaf into two leaves, and then update the parent of the leaf with the new leaf-pointer. This operation is handled by the `splitLeafAndAddTuple()` method of the `LeafPageOperations` class. **You will need to complete this implementation.**

   As always, there are many helper functions to help you with the implementation, on both the `LeafPage` and `InnerPage` classes. Probably the most complicated part will be updating the parent of the leaf properly, but you can use the `InnerPageOperations` class to help you with this task.

   Note that the `pagePath` argument must always be the path to the specific page being manipulated by a given function. Thus, when calling `InnerPageOperations` functions, you must remove the last element from the `pagePath` list. This is simple to do, and fast too, even though we are using an `ArrayList` for the collection: since we are removing the last element in the array-list, this will be a constant-time operation.

   Once you are done with this task, you should be able to create B$^+$ tree files with many leaf pages. There is one more problem, though – the index implementation still can't support multiple inner pages. To fix this issue, continue on to the final step.

3. The last functionality to complete for this index implementation is the code that allows inner-page pointers to be moved to a left- or right-sibling page. This is required for splitting an inner page into two, and also for relocating pointers between two sibling inner pages. This functionality is provided by the `movePointersLeft()` and `movePointersRight()` methods of the `InnerPage` class.

   These methods are a bit tricky to implement, mainly because of the requirement that every tuple in an inner page must be sandwiched between two pointers. Given an inner page containing N pointers and N-1 tuples, if you move M pointers (and the M-1 tuples between these pointers) from the node to its right sibling ($M < N$), this will expose a tuple in the node without a pointer on its right. Similarly, if you move M pointers from the node to its left sibling, this will expose a tuple without a pointer on its left.

   Additionally, the sibling node receiving the M pointers and M-1 tuples will already have pointers on both sides of all its tuples.

This is where you must figure out how the parent node's tuple fits into the puzzle. In the slides we discussed what happens when a single pointer is moved to a sibling inner-node, but in this implementation it is possible to move *M* pointers, not just one. You will have to figure out where to store the parent's old tuple, if provided, and what to return as the parent's new tuple.

(You will always return a new key in your implementation. You may not receive an old tuple if the top-level inner page is being split, since there will not yet be a parent of the node being split. The tuple you return will be used in initializing the new top-level inner page.)

The other complexity is that when moving *M* pointers to the left sibling, these pointers are taken from the start of the node's sequence, whereas when moving the pointers to the right sibling, they are taken from the end of the node's sequence. When moving pointers right, the implementation must make room in the target node for the new entries. When moving pointers left, the implementation must slide the remaining entries in the source node left. For these kinds of operations, the `DBPage.moveDataRange()` method will be very helpful.

You must never write to the `DBPage`'s internal byte-array directly! Doing this will break the `DBPage`'s ability to track whether the page is dirty. Always use the operations provided on the `DBPage` to write to its data. (You may find it helpful to read from the underlying byte-array, however, when moving data back and forth.)

Once you have successfully completed this task, your B$^+$ tree should be complete.

Part 2: Support for B$^+$ Tree Indexes

Once you have B$^+$ tree tuple files working, you can complete the mechanism that keeps indexes in sync with their corresponding tables. As explained earlier, there is a row-event handler that will update a table's indexes based on the changes made to the table. You will need to edit the `StorageManager` class to set its `ENABLE_INDEXES` flag to true; this will turn on index management.

The `IndexUpdater` class (in the `indexes` package) handles adding and removing tuples on a table's indexes. (Updates to a tuple are currently modeled as removing the old version and then adding the new version, which is not optimal, but it works.)

There are two methods that you must complete on this class:

1. The `addRowToIndexes()` method is called when a row is inserted or updated on a table. This method must iterate through the table's indexes, construct a suitable index-tuple for each index (based on the columns in the index), and then add this index-tuple to the index's tuple file.

   Also, some indexes are unique indexes while others are not; this method should also verify that the tuple being added will not violate any unique constraints.

2. The `removeRowFromIndexes()` method is called when a row is updated or deleted on a table. As before, this method must iterate through the table's indexes, removing the corresponding index-tuple from each index.

The `IndexUtils` class has a number of methods that will be helpful for completing these implementations:

- The `makeSearchKeyValue()` method takes a tuple from a table, and constructs a corresponding index tuple based on the index's definition. The last argument `findExactTuple` can be used to include or exclude the file pointer to the table-tuple, e.g. to either find or store the exact location of the table tuple into the index.

For example, when checking a unique index to see if a given tuple appears in the index, `false` should be passed for `findExactTuple`, because we don't want to consider the tuple's file-pointer when looking for another tuple with the same index-column values.

- The `findTupleInIndex()` method takes an index's tuple file and a search key, and attempts to locate a tuple in the index tuple-file with the same values as the specified key. This method handles the different interfaces of sequential and hashed tuple files, so that the `IndexUpdater` doesn't have to.

Part 3: Analysis of Implementation

The design document for this assignment has a number of questions for you to answer

Given NanoDB's B$^+$ tree implementation, consider a simple schema where an index is built against a single integer column:

```
CREATE TABLE t (
    -- An index is automatically built on the id column by NanoDB.
    id INTEGER PRIMARY KEY,
    value VARCHAR(20)
);
```

Answer the following questions. (Hint: You can turn on logging in the B$^+$ tree package to get some of these answers.) Recall that "pointers" to other pages in the B$^+$ tree structure are represented by the page-number of the referenced page, and therefore occupy two bytes. Also, when discussing B trees in general, the terms "node" and "page" are equivalent, because each node is one page in size.

Make sure to explain all of your answers; give your rationale, your calculations, and so forth. Don't simply state a value without any explanation, or you will receive point deductions.

1. What is the total size of the index's search-key for the primary-key index, in bytes? Break down this size into its individual components; be as detailed as possible. (You don't need to go lower than the byte-level in your answer, but you should show what each byte is a part of.)

2. What is the maximum number of search-keys that can be stored in leaf nodes of NanoDB's B$^+$ tree implementation? You should assume the default page-size of 8192 bytes.

3. What is the maximum number of keys that can be stored in inner nodes of this particular implementation? (Recall that every key must have a page-pointer on either side of the key.)

4. In this implementation, leaf nodes do not reference the previous leaf, only the next leaf. When splitting a leaf into two leaves, what is the maximum number of leaf nodes that must be read or written, in order to properly manage the next-leaf pointers?

   If leaves also contained a previous-leaf pointer, what would the answer be instead?

   Make sure to explain your answers.

5. In this implementation, nodes do not store a page-pointer to their parent node. This makes the update process somewhat complicated, as we must save the sequence of page-numbers we traverse as we navigate from root to leaf. If a node must be split, or if entries are to be relocated

from a node to its siblings, the node's parent-node must be retrieved, and the parent's contents must be scanned to determine the node's sibling(s).

Consider an alternate B+ tree implementation in which every node stores a page-pointer to the node's parent. In the case of splitting an inner node, what performance-related differences are there between this alternate representation and the given implementation, where nodes do not record their parents? Which one would you recommend? Justify your answer.

6. It should be obvious how indexes can be used to enforce primary keys, but what role might they play with foreign keys? For example, given this schema:

```
CREATE TABLE t1 (
    id INTEGER PRIMARY KEY
);

CREATE TABLE t2 (
    id INTEGER REFERENCES t1;
);
```

Why might we want to build an index on `t2.id`?

Part 4: Extra Credit

There are two main things you can focus on if you race through the rest of this assignment. The first option is definitely more appealing than the second one.

Use Indexes in Query Planning

Indexes can be used in a variety of ways during query planning. The two most obvious ones for NanoDB are in grouping/aggregation, and in using indexes for tuple lookups:

- If a grouping/aggregation operation's columns are all contained in an index, we can optionally perform a "file scan" over the index to compute the query.[1]
- Even better, if the index is ordered by the grouping columns, we can use a sort-based grouping/aggregate operation instead of a hash-based grouping/aggregate.

Similarly:

- If a query's predicate contains conjuncts that can be used with an index to look up tuples more efficiently, we can use the index to retrieve the tuples instead of scanning through the tuple file.

NanoDB's `FileScanNode` has been updated to be able to scan through indexes as well, so that you can perform the grouping/aggregate optimization. However, you will have to implement the code that determines when the index can be used for grouping and aggregation.

Similarly, a new `IndexScanNode` has been added, which can either perform an equality-based lookup of tuples, or a range-based lookup of tuples, using a specific index on a table. This implementation is incomplete; you must add plan-costing details to allow it to be useful with the cost-based planner/optimizer. Similarly, you would need to update the planner/optimizer to consider available indexes, and to utilize an `IndexScanNode` instead of a `FileScanNode`, when using an index would be cheaper.

[1] Note that we use the term "index scan" to mean performing an equality-based or range-based lookup on an index. Similarly, the term "file scan" means to scan through all tuples in a tuple file. Thus, we can perform a "file scan" over an index. (Microsoft SQLServer has clearer terminology; an "index scan" is a file-scan over an index, and an "index seek" is an equality- or range-based lookup against an index.)

Finally, you might note that the plan costs have no way of estimating seeks, so you will have to find a way to represent this in your plan costs. (Recall that index scans can generate huge numbers of seeks…)

Use Indexes for Constraint Enforcement

The indexes package also contains a class named `DatabaseConstraintEnforcer`, which endeavors to enforce various constraints, including `NOT NULL` constraints, primary/candidate key constraints, and foreign key constraints. Unfortunately, this class needs to be largely rewritten and cleaned up; it chooses some very hacky ways to implement this functionality. (It was written by a student, and I have not yet had time to clean it up.)

If you are a glutton for punishment, you should rewrite this and get constraint enforcement working properly in NanoDB.

You will need to enable the database constraint enforcer by registering it as a row-event listener at the end of the `StorageManager.initialize()` function.

Submitting Your Assignment

When you are finished with the coding part of the assignment, tag it with a `hw6` tag as usual, and then push all of your changes to your repository on the CMS cluster.