Database System Implementation CSCI 421 Group Project Phase 4

v1.0

1 Phase Description

This is the forth, and final, phase of the semester long group project; other than the individual group evaluations.

In this phase you will be implementing indexing.

There are a few basic rules when implementing this phase:

- You must create and use an instance of the storage manager you created in phase 1.
- You must create and use an instance of the DDL/DML parsers you created in prior phases.
- You can assume the database location path exists and is accessible.
- You must use the catalog structure implemented in prior phases.
- When updating/removing/altering/inserting an attribute that has an index, you must update the index to reflect that change.
- Depending on your implementation you most likely will need to make changes to much of the functionality for the prior phases.

2 Phase Layout

In this you will be implementing B+ tree based indexing.

How you implement this into your project is up to you.

3 `create index` DDL command

A user can add an index at anytime. This functionality should be added to the `DDLParser`'s `parseDDLStatement` function.

The format of the command:

```
create index <i_name> on <tableName>( <attr> )
```

The parts of the command:

- `create index`: All DDL statements that start with this will be considered to be trying to create an index on a table.
- <`i_name`>: the name of the index to create. Index names are unique.
- <`tableName`>: the name of the table to create the index on. Table names are unique.
- <`attr`>: the name of the attribute in the table to create the index on. Attribute names are unique within a table.

Examples:

```
create index myFooIndex on foo( bar );
create index myOtherIndex on foo( bax );
```

An attribute can only have one index. Primary keys automatically get a B+ tree index when the table is created.

This functionality will be added to the `DDLParser`'s `parseDDLStatement`.

4 B+ Tree Index

You will implement a B+ Tree based index; based on lecture and book algorithms.

- Each node in the tree will be a page on hardware.
- Each node will have to track if it is an internal node or a leaf node.
- Internal nodes should contain a pointer to other nodes (pages)
- Leaf nodes should contain pointers to buckets (special pages).
- A root node can be either an internal node or a leaf node.
- A search-key value can only be one attribute value.
- The number of pointers in the node, the N of the tree, will be the max page size divided by the max size of a search-key, page pointer pair size. This will then be floored and minus one from the floor. The size of the search-key is the maximum size of any value the search-key can be.
  Example:

```
Page size is 4096 bytes
Data Type is varchar(10) --> max size is 10 * 2 bytes = 20 bytes
Page pointer is an integer --> size 4 bytes
Pair size = 20 + 4 = 24 bytes
4096 / 44 = 93.09 pairs
floor(170.6667) - 1 = 169 --> N value of the B+ Tree
```

- Buckets are special pages that contain `<page_number, index>` pairs; values that make up `RecordPointer`s. The page number references a page containing an entry with that search key value. The index references the index on that page containing an entry with that search key value. The size of bucket page will be the `floor( page_size / ((size(integer) + size(integer))`. The last entry in the page will be reserved as a special pointer to the next page of the bucket; it will be empty on the last page. Buckets in B+ Trees cannot contain pairs with different search key values.
- You cannot just store the `RecordPointer` objects.

5 Using an Index

When accessing data in the database you must you index if one exists on the search-key attribute combination. You must also update the index as you insert/update/delete data in the database.

5.1 Inserting Data using a Primary Index

When inserting/deleting/updating data in the database you must use the primary key index to find the page of the entry.

When inserting you will find the location in the index that the new item belongs and look at the page number of the entry just before it. Then try to insert it into that page.

For example, if inserting an row with the primary key of 5. Navigate to the node in the tree that the entry would belong in. If there is a search key with that value this is an errors (this is the primary key index). If there is no search key of that value insert the search key value; lets say between search keys 4 and 8. Get the `<page_number, index>` of pair of the search key 4 just before this new entry. This will give you an idea of where to insert the new entry, because 5 will belong right after 4.

Insert the new record data into the table page right after the `<page_number, index>` of the prior search key value. You must follow the rules of the storage manger; such as splitting, ordering, etc.

Then insert the new search key and record pointer into the B+Tree; updating the tree as needed.

You may need to update other search key's pointers if the insertion of the new data caused the storage manager to move rows to a new page when splitting pages.

This will require changes to your Storage Manager.

5.2 Deleting Data using a Primary Index

When deleting a value from the B+Tree, you will find the search key in thew tree; if it does not exist there is nothing to delete.

If the search key exists there should only be one record pointer at that search key value. Get the record pointer data and delete that record from that `<page_number, index>` location. Finally, delete the search key from the tree.

This will require changes to your Storage Manager.

5.3 Updating Data using a Primary Index

Updating a search key will involve inserting a new search key an deleting a the old one if the primary key or location changed.

This will require changes to your Storage Manager.

5.4 Using a Non-primary Key Index

When inserting new row there may be multiple non-primary key indices. These mainly will not involve updating anything in the storage manager. When inserting/updating/deleting

the storage manger will use the primary key index to insert, delete, or update the actual data.

The non-primary key indices will need to have `RecordPointer`s added to the search key value for the attribute the index is on. It SHOULD NOT need to make any changes to the table data.

5.5 Using Indices for Querying

When query data and index can be used in the `where` clause to find the data needed for the query.

For instance:

```
select * from foo where bar = 5;
```

Assuming there is an index on `bar`, the index can be used to find all tuples with the value 5 without reading the entire table.

Given:

```
select * from foo,bar where foo.b = 5 and bar.a = 2;
```

Assuming indices exists on both attributes in the `where` clause, you can use the indices to limit the rows in each of the tables before doing Cartesian product.

Given:

```
select * from foo where bar < 5;
```

Assuming there is an index on `bar`, the index can be used to find all of the tuples where `bar` is less than 5 by starting at the lower left leaf node and iterating the leaf nodes until the search-key values are no longer less than 5.

Given:

```
select * from foo where bar > 5;
```

Assuming there is an index on `bar`, the index can be used to find all of the tuples where `bar` is greater than 5 by finding the leaf node that would contain the search-key value 5 and iterating the leaf nodes to the right.

Multi-attribute indices can be used for a single attribute if the single attribute is the first attribute in the index.

This does not cover all cases, review the lecture materials for using indices to assist with query processing.

6 IBPlusTree Interface

This section will outline the details of the variables and functions of the `IBPlusTree` interface. This will used to outline the required functionality of the B+ Tree indexing. You must implement each of the functions as outlined in this document. The implementation will be placed in `BPlusTree` class.

This is a provided file. **DO NOT CHANGE ANYTHING IN THIS FILE**

When an index is created it will need to be provided `<K>`. This is the data type of the search key; Integer, Double, Boolean, String.

6.1 `insertRecordPointer(RecordPointer rp, K searchKey)` Function

This function will insert a record pointer into the tree as outlined above.

Params:

- `rp` is the record pointer being inserted.
- `searchKey` is the search key of the record pointer being inserted

This function will return true if successful, and false if some error occurs. Errors should be reported to `System.err`.

6.2 `removeRecordPointer(RecordPointer rp, K searchKey)` Function

This function will remove a record pointer into the tree as outlined above.

Params:

- `rp` is the record pointer being removed.
- `searchKey` is the search key of the record pointer being removed

This function will return true if successful, and false if some error occurs. Errors should be reported to `System.err`.

6.3 `search(K searchKey)` Function

This function will return all of the record pointers for the provided search key. A search key can have one (primary index), multiple (non-primary index), or no record pointers (search key does not exist).

Having no pointers is different than having an error.

It will return an ArrayList of record pointers.

6.4 `searchRange(K searchKey, boolean lessThan, boolean equalTo)` Function

This function will return all of the record pointers for the provided search key range. A search key range can have multiple, or no record pointers.

Params:

- `seachKey` the search key to start the range search at. Note this exact search key may or may not exist.
- `lessThan` if true this is a less than search; otherwise a greaterThan search.
- `equalTo` if true include the search key value; otherwise do not include it.

Less than will be more difficult than greater than. You will have to start at the front of the index leaf nodes an iterate up to the search key value.

Greater than you will find the search key value and iterate up from there.

Having no pointers is different than having an error.

It will return an ArrayList of record pointers.

7 `ITable` Interface

You will have to implement the `addIndex` function in this interface in this phase. It will take in an attribute name and add an index to it. When adding an index the index must be created and populated with any current data in the table.

An attribute can only have one index. Adding another index will result in an error.

8 Project Constraints

This section outlines details about any project constraints or limitations.

Constraints/Limitations:

- You must follow the rules of B+ Trees as outlined in lecture.
- Everything, except for the values in Strings (char and varchar), is case-insensitive; like SQL. Anything in double quotes is to be considered a String. String literals will be quoted and can contain spaces. Example: `"foo bar"` is a string literal. `foo` is a name; not a string literal. The quotes do not get stored in the database. They must be added when printing the data.
- You must use a Java and the requirements provided.
- Your project must run and compile on the CS Linux machines.
- Submit only your source files in the required file structure. Do not submit and IDE directories or projects.
- Your code must run with any provided tests cases/code. Failure to do so will result heavy penalties.
- Any Java errors, such as file reading/writing errors, should be handled with a useful error message printed to `System.err`. The user can determine how to handle the error based on the return of the functions.
- Words such as `Integer`, `create`, `table`, `drop`, etc are considered key words and cannot be used in any attribute or table name.
- Data must be checked for validity. Examples:
  - type matching
  - not nulls
  - primary keys
  - foreign keys
- Any changes to data in the database must keep the database consistent. This means that all not null, foreign keys, and primary keys must be observed.

9 Grading

Your implementation will be grading according to the following:

- (10%) DDL parsing for index creation.
- (40%) Indices used in queries, insertions, etc.
- (40%) B+Tree based indices.
- (10%) Indices added for primary keys.

Penalties:

- (-50% of points earned) Data is written as text not binary data.
- (-25% of points earned) Does not use storage manager to handle hardware access. Catalog reading/writing to hardware does not need to be handled by the storage manager.
- (up to -50% of points earned) Does not work with any provided testing files.
- (up to -100% of points earned) Does not compile on CS machines.

Penalties will be based on severity and fix-ability. The longer it takes the grader to fix the issues the higher the penalty.

Examples:

- Code does not compile due to missing semicolon: -5%
- Code does not compile/run with testers due to missing functions or un-stubbed functions: -10%
- Multiple syntax errors causing issues compiling issues and takes over 30 mins to fix: -100%
- Multiple Crashes with provided testers that take an extended time to fix: -50%

These are just examples.The basic idea is "Longer to fix, more points lost. Eventually give up, assign a zero."

10 Submission

Zip any code that your group wrote in a file called `phase3.zip`. Maintain any required package structure. Do NOT submit any provided code. The grader will use their own versions.

Submit the zip file to the Phase 1 Assignment box on myCourses. No emailed submissions will be accepted. If it does not make it in the proper box it will not be graded.

The last submission will be graded.

There will be a 48 hour late window. During this late window no questions will be answered by the instructor. No submissions will be accepted after this late window.