CS 4321/5321 Project 4 (Full Instructions)

Fall 2016

Due Tuesday, November 15, 11:59 pm

This project is out of 60 points and counts for 20% of your grade.

1 Goals and important points

In Project 4, you will extend your SQL interpreter to support B+-tree indexing.

- you will write code to build B+-tree indexes for your database. Since your database is static, you do not need to implement tree insert/delete functionality. Instead you will focus on:
  - implementing an algorithm to construct a tree via *bulk loading*
  - implementing functionality to serialize/deserialize a tree from a file
- you will use B+-tree indexes to help with relational algebra selections, and do some performance benchmarking to see whether using indexes speeds up your queries.

You will still support the same subset of SQL as in previous projects, and you should still follow the join order implied by the `FROM` clause when you build your plan.

If you want to reuse any of the 4320 B+-tree code for this project, **you may do so but you must give a proper acknowledgment**.

2 B+-trees

This section explains the B+-tree properties, algorithms and serialization format to be used in this Project.

2.1 Tree basics

Every tree has an *order*, which is an integer d. Every leaf node must have at least d and at most $2d$ data entries. Every index node must contain at least d keys and $d+1$ child pointers, and at most $2d$ keys and $2d+1$ child pointers.

You only need to support indexes on one attribute, and you will be building at most one index per relation.

Your tree should use Alternative (3), see textbook p. 276. Thus the leaf nodes contain data entries of the form $< key, list >$ where *key* is the (integer) search key for the index and *list* is a list of record ids (*rid*s). A *rid* is a tuple identifier and has the form $(pageid, tupleid)$ where *pageid* is the number of the page the tuple is on, and *tupleid* is the number of the tuple on the page numbered *pageid*. For the purpose of *rid*s, we number both pages and tuples within pages starting at 0. Thus the very first tuple in the file has *rid* of $(0, 0)$, the second tuple has rid $(0, 1)$ (unless we have such huge tuples that only one can fit on a page, in which case the rid of the second tuple would be $(1, 0)$), etc. The data entries for a given key should be sorted first by *pageid* then by *tupleid*.

2.2 Bulk loading

The bulk loading algorithm is a fast way to build a B+-tree without having to do repeated insertions and splitting. You should not use the bulk loading algorithm described in your textbook; rather, you should use the one described below.

Begin by scanning the relation file and generating all the data entries in the format described above, in sorted order by key.

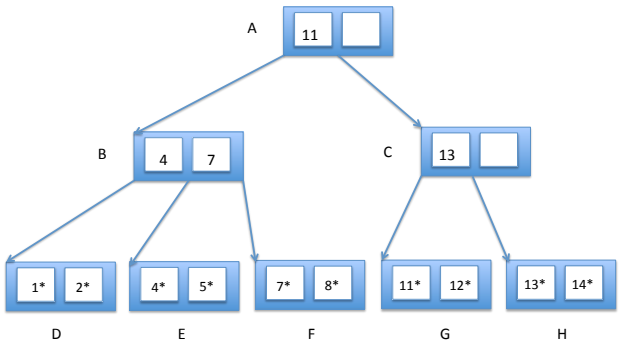
Next, build the leaf node layer. Because the data is static it is in our interest to fill the tree completely (to keep it as short as possible). Thus every leaf node gets $2d$ data entries. However, this may leave us with $< d$ data entries for the last leaf node. If this case happens, handle the second-to-last leaf node and the last leaf node specially, as follows. Assume you have two nodes left to construct and have k data entries, with $2d < k < 3d$. Then the second-to-last node gets $k/2$ entries, and the last node gets the remainder.

Next, build the layer of index nodes that sits directly above the leaf layer. Every index node gets $2d$ keys and $2d + 1$ children, except possibly the last two nodes to avoid an underfull situation as before. If you have two index nodes left to construct, and have a total of m children, with $2d + 1 < m < 3d + 2$, give the second-to-last node $m/2$ children (i.e. $m/2 - 1$ keys) and the remainder of the children to the last node.

When choosing an integer to serve as a key inside an index node, consider the subtree corresponding to the pointer *after* the key. Use the smallest search key found in the leftmost leaf of this subtree.

Continue construction of the next index layer(s) until you get to the root. The root may be underfull.

Here is an example of a tree built using the bulk loading algorithm above, with $d = 1$. (Ignore the labels on nodes like A, B, C for now, they will be used in the next section.)



You may assume we will never ask you to build an index on an empty relation. If the relation is not empty,

but so small that the tree has only one leaf node, you should create a two-node tree consisting of the leaf node and one index node. The index node contains no key, only a pointer to the leaf node.

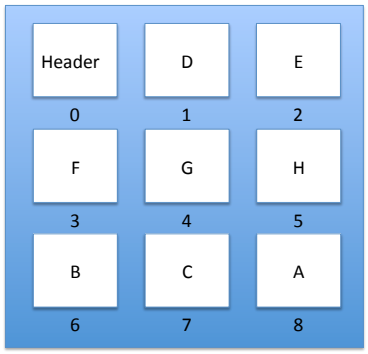
2.3 Serializing a tree to a file

Just like database relations, B+-trees must be stored in files. Here we explain the format for serializing a B+-tree to a file. In all the below, whenever there is empty (unused) space at the end of a page, it should be filled with zeroes.

The file is laid out in pages, with each page being 4096 bytes long. The first page in the file is a *header page*, and then every node in the tree is laid out on its own page. **You may assume every node will fit in a 4096-byte page.**

The pages are laid out as follows. After the header page, we serialize all leaf nodes in order left-to-right, then the layer immediately above the leaves in order left-to-right, and so on. The root node is serialized onto the last page in the file. For our example tree from before, the overall file layout is as shown below. Every white box represents a 4096-byte page.

Note that because the index is static and because the leaf pages are consecutive in the file, the leaf pages do not need to maintain pointers to next/previous leaves like in your textbook.



The *address* of a node is the number of the page it is serialized on. Thus the address of the root in our tree is 8, the address of node E is 2, and so on.

The *header page* contains just three integers:

- the address of the root, stored at offset 0 on the header page
- the number of leaves in the tree, at offset 4
- the order of the tree, at offset 8.

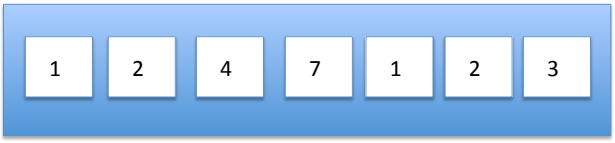
For our example tree, the header page would thus contain the integers $8, 5, 1$.

The way a node is serialized to a page depends on whether it is an index node or a leaf node.

For an *index node* the corresponding page in the file contains, in order:

- the integer 1 as a flag to indicate this is an index node (rather than a leaf node)
- the number of keys in the node
- the actual keys in the node, in order
- the addresses of all the children of the node, in order

An example serialized page for node B is shown below. Every white box is a four-byte integer "slot" on the page.



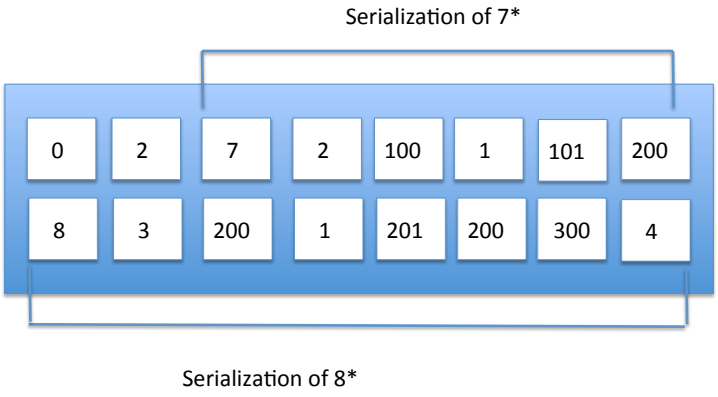
For a *leaf node* the corresponding page in the file contains, in order:

- the integer 0 as a flag to indicate this is a leaf node
- the number of data entries in the node
- the serialized representation of each data entry in the node, in order.

Recall that a data entry has the format $< k, [(p_1, t_1), (p_2, t_2), \cdots (p_k, t_k)] >$ where k is the key and the (p_i, t_i) pairs are record ids. To serialize a data entry, we write, in order:

- the value of k
- the number of *rid*s in the entry
- p_i and t_i for each *rid* in the entry

In our example tree, consider node F. Suppose the data entry 7^* is $< 7, [(100, 1), (101, 200)] >$ and the entry 8^* is $< 8, [(200, 1), (201, 200), (300, 4)] >$. A serialized representation of the node is as shown below.



3 Input and output formats

The file format for database relations is the same as in Project 3. The file format for indexes is as described above in Section 2.3. In this section we describe the top-level input and output formats for your overall program, with a focus on what has changed from Project 3.

3.1 Top-level functionality

For our grading (and your own testing), it should be possible to run your code in several different ways. There are three binary options you need to support:

1. We may want your code to run queries, or just to build indexes without running any queries.
2. If we want to run queries, we may want you to build indexes, or we may provide indexes of our own.
3. If we want to run queries and assuming indexes are available (either you built them or we provided them), we may want you to use indexes for selection, or to ignore the indexes and implement selection as in Project 3.

Each of the three binary options above is governed by a boolean flag set in an appropriate configuration file, as explained below.

3.2 Interpreter configuration file

The biggest difference from Project 3 is that we add a new *configuration file* for your interpreter. We will now run your code (on the command line, as a .jar) by passing in the path to the configuration file as a single command-line argument.

The interpreter configuration file will contain:

- on the first line, the input directory to be used,
- on the second line, the output directory to be used,
- on the third line, the temporary sort directory to be used,
- on the fourth line, a flag to indicate whether the interpreter should build indexes (0 = no, 1 = yes)
- on the fifth line, a flag to indicate whether the interpreter should actually evaluate the SQL queries (0 = no, 1 = yes).

The first three items above are the same as the three command-line arguments you used for Project 3; they have just been moved into a config file because we need more arguments now.

The fourth and fifth arguments govern two of the three binary choices described in Section 3.1. If we set these arguments to $1, 0$ your code should build indexes but not run queries, if we set them to $1, 1$ it should build indexes and run queries, and if we set them to $0, 1$ you should *not* build indexes (we will provide them), but only run queries. The option with flags $0, 0$ does not make much sense so we will not use it.

3.3 Directory structure

The output and temporary sort directories are as in Project 3. The input directory has a a few small differences to Project 3. Take a look at the samples provided to clarify what is going on.

At the top level, the input directory contains:

- a `queries.sql` file
- a `plan_builder_config.txt` file
- a `db` subdirectory

Most of these are as in Project 3. However, we expand the contents of `plan_builder_config.txt`. We add a third line to the file to specify whether the physical plan builder should use indexes for selection (where possible) or if it should ignore the indexes use only the full-scan-based implementation you have been using until now. This is the third binary option discussed in Section 3.1. To use indexes we set 1 on the third line of the config file, and to use the full-scan implementation we set 0 on the third line of the config file.

Within the `db` subdirectory, there is:

- a `data` directory which is exactly as in Project 3,
- a `schema.txt` file which is exactly as in Project 3,
- a `index_info.txt` file and a `indexes` directory which are new and described below.

The `index_info.txt` file specifies which indexes should be built. There is one line per desired index. Each line specifies the relation name, the attribute name, a flag 0 or 1 depending on whether the index is unclustered (0) or clustered (1), and the order of the tree. Thus an example file would be:

```
Sailors A 0 10
Boats E 1 20
```

This states there is an unclustered index on `Sailors.A` with order 10 and a clustered index on `Boats.E` with order 20.

This file will be used by your code in two different ways:

- if you need to build indexes, you will read the file to figure out which indexes to build
- if you don't need to build indexes (just evaluate queries), you will read the file to figure out what indexes are available.

The `indexes` directory contains the actual indexes, in the binary format described in Section 2.3. The index for a relation R and attribute A is stored in a file named $R.A$. You may assume the `indexes` directory always exists, though it may be empty.

4 Implementation instructions

4.1 Tree construction and serialization

Start by implementing functionality to build a B+-tree index on a single attribute of a given relation on the database, and serialize that index to a file.

We have provided some sample indexes; for each sample index, we have given you the binary serialized index file, and a human-readable deserialization of the whole tree.

Implement the bulk loading algorithm from Section 2.2. You can serialize the layers of nodes as you generate them, starting from the leaves. Note that the index nodes need to know the addresses of their child nodes in order to be serialized. Because of the file format used for serializing trees, you do not need to compute the address of every node right away; you may compute these "as you go" layer by layer. Once you have serialized the leaf nodes and know their addresses, you can store these addresses in the index nodes (i.e. the Java objects representing index nodes) for the next layer, and then you are ready to serialize these index nodes and obtain *their* addresses, etc.

The root is the last node to be serialized; it follows that the header page is the last page to be written since you need to know the address of the root. Of course you could instead compute the address of the root in advance since you know how many data entries your index has.

Add support for building both clustered and unclustered indexes. If the index is to be unclustered, build it as described above; if the index is to be clustered, start by sorting the relation on the desired attribute and replacing the old (unsorted) relation file with the new (sorted) relation file. Then build the index.

It is OK to use unbounded state during index construction; that is, you may keep the tree in memory during construction, and if you need to sort anything you may use in-memory sort.

Now integrate the index construction functionality into your top-level code, following the logic described in Section 3.2. That is, if your interpreter is called with the appropriate flag set, make sure it scans the `index_info` file and builds indexes in the `indexes` directory as required. Note that all indexes should be built before running any queries.

4.2 The index scan operator

Now, integrate indexes into query evaluation. The way to do this is to implement an *index scan operator*. This is a new physical operator. An index scan is somewhat like a file scan, except that it will only retrieve a range (subset) of tuples from a relation file, and it will use a B+-tree index to do so.

Every index scan operator needs to know the relation to scan, the index to use, whether the index is clustered or not, and two parameters that set the range of the scan: `lowkey` and `highkey`. If you want to use the index to retrieve tuples with key between 50 and 500, for example, you would create an index scan operator with `lowkey` 50 and `highkey` 500. One of `lowkey` and `highkey` can be set to null, to indicate the lack of a bound. E.g. to return all tuples with key > 50, you could use `lowkey` 50 and `highkey` null.

It is up to you to decide whether the interval between `lowkey` and `highkey` is open or closed; you can do it either way. That is, you can decide whether the index scan will return all tuples having `lowkey` $\leq$ key $\leq$ `highkey`, or `lowkey` $<$ key $<$ `highkey`.

Your index scan operator should implement `getNextTuple()` by retrieving tuples one at a time using the index. Somewhere early in the life cycle of your index scan operator – either upon construction or upon first call to `getNextTuple()` – you will need to access the index file, navigate root-to-leaf to find `lowkey` (or where `lowkey` would be if it were in the tree, since it may not be present) and grab the next data entry from the leaf. Note that this root-to-leaf descent will involve deserializing a number of nodes in the tree; **do not deserialize the whole tree**, deserialize only the pages that you need.

Once calls to `getNextTuple()` start arriving, the behavior of your index scan depends on whether the index is clustered or unclustered. If the index is unclustered, each call must examine the current data entry, find the next rid, resolve that rid to a page and a tuple within the data file, retrieve the tuple from the data file, and return it. If the index is clustered, you must scan the (sorted) data file itself sequentially rather than going through the index for each tuple. Thus you don't need to scan any index pages after the initial root-to-leaf descent that gives you the rid of the first matching tuple.

4.3 Using index scan to implement selection

Now you are ready to use the index scan operator for query evaluation. The idea is that in conversion from the logical to the physical query plan, certain logical selection operators can be translated to index scan operators instead of your old full-scan selection operators. Of course this only works if:

- the selection is on a base table, i.e. the child of the selection is a leaf rather than, say, a projection or a join, AND
- there is an appropriate index that can actually help with the selection.

Furthermore, depending on the selection condition, the index may not be able to handle all of it. For example if you have a selection on table `R` and the condition is `R.A < 5 AND R.B == 3`, and you have an index only on `R.A`, you cannot use it to help with the `R.B == 3` portion.

Your `PhysicalPlanBuilder` should proceed as follows. First, determine whether to attempt using indexes at all based on the configuration file. If no indexes are to be used, the plan builder works as in Project 3, translating every logical selection operator to the old full-scan-based physical selection operator.

Otherwise, the plan builder attempts to use an index to help with every logical selection operator. Indexes only help if the selection operator has a leaf/scan as its child; if you are constructing the join tree by pushing all selections as suggested in Project 2, this will be true for all selection operators in your logical plan. If you are not pushing selections, you need some way to determine whether the child is a leaf/scan.

Now you need to find out whether there is an index on the relation in the selection, and whether that index is clustered or not; your trusted database catalog class should be able to help with that. Remember, for simplicity we are assuming that each relation has at most one index built on it.

Next, you need to separate the selection into the part that can be handled via an index scan and the "remainder" that cannot. That is, your logical selection operator is potentially translated into *two* new operators: an index scan operator that handles a portion of the selection, and a full-scan physical selection operator that has the index scan as a child and handles the rest of the selection. It is possible that the entire selection condition can be handled via the index, in which case the plan builder should create only one (index scan) operator, or that none of it can be handled via the index, in which case the plan builder should create only one (full-scan) operator.

The main challenge will be dividing the selection condition into the portion that can be handled by the index and the portion that can't, and translating the first portion into a single `lowkey`/`highkey` pair for your index scan. This can be done using another class that implements `ExpressionVisitor`. Note that if your indexed attribute is `R.A`, conditions of the form `R.A < 42, R.A <= 42, R.A > 42, R.A >= 42` and `R.A = 42` can all be handled through the index, but any other conditions cannot, even if they involve `R.A`, e.g. `R.A != 42` or `R.A == R.B`. You should combine all the conditions for which the index is helpful into a single `lowkey`/`highkey` pair for your index scan. Of course you will also need to keep track of the remainder of the selection condition, which will be used in the full-scan physical selection operator.

You can assume we will not give you any pathological selection conditions that would evaluate to false on every tuple, such as `R.A > 5 AND R.A < 1`. You can also assume that the selection conditions will not contain any comparisons between two integers, such as `41 < 42`, which evaluate to either true or false trivially. Pathological conditions such as the ones above are conceptually easy to eliminate during a preprocessing pass, and we will not make you write the preprocessing pass.

4.4 Performance benchmarking

As in Project 3, we want you to do some benchmarking to see whether/when indexing speeds your query evaluations. Choose at least three queries that involve selection, and try to run each one using full-scan selection, using an unclustered index, and using a clustered index. If your queries involve a join, you may use any join algorithm you like.

Report the running times in a bar graph as in Project 3. As before, you will not be graded on the actual performance numbers you report, only on whether you have carried out the evaluation to our specifications.

You may make your own decisions on how to choose the queries, data and indexes, but as before you need to make those very clear to us for grading. It may be interesting to experiment with queries that have higher or lower selectivity; for instance if the value of attribute `A` is between 0 and 1000, indexes are likely to give a much bigger gain for queries with small ranges (`SELECT * FROM R WHERE R.A < 5`) than for large ranges (`SELECT * FROM R WHERE R.A < 500`).

You must create a pdf file called `Experiments.pdf`. This file must contain:

- the three queries you ran
- a description of the data you used - i.e the schema of each relation, how many tuples per relation, and how these tuples were generated (e.g. "each attribute value was chosen uniformly at random in the range 0 to 1000")
- the indexes used for each query
- as in Project 3, a bar graph comparing the running times.

5 Grading

As usual, you are free to make your own architectural decisions with the following exceptions:

- You need to implement an index scan operator as explained in Section 4.2. This operator must:

  - extend your physical operator abstract class
  - allow the setting of `lowkey` and `highkey` parameters for the scan, either in the constructor or somewhere else
  - separately handle the case of clustered vs unclustered indexes
  - not deserialize the entire tree on the initial root-to-leaf descent

- Your physical plan builder must translate every logical selection into an index scan and full-scan portion, unless the selection condition is such that only one of those is appropriate. Furthermore, it must handle the maximum possible portion of the selection condition via an index scan.
- As in Project 3, you need to provide a good-faith implementation of the functionality we are asking for, or tell us in the README if you have not been able to implement some of this functionality.

Next we give the grading breakdown.

5.1 Code style and comments (10 points)

As in previous projects, you must provide comments for every method you implement. At minimum, the comment must include one sentence about the purpose/logic of the method, and `@params`/`@return` annotations for every argument/return value respectively. In addition, every class must have a comment describing the class and the logic of any algorithm used in the class. As in previous projects, if you follow the above rules and write reasonably clean code that follows our overall architecture, you are likely to get the full 10 points for code style.

5.2 Benchmarking (10 points)

You will not be graded on the running times of your code; you will be graded on whether your `Experiments.pdf` file contains all the info we ask for in Section 4.4. So these should be an easy 10 points.

5.3 Automated tests (40 points)

We will run your code on our own queries and our own data. Our goal is to award you 20 points for index construction and 20 points for the use of indexes in selection queries. Correctly building the sample indexes we provided will count for 8 of the 20 points for index construction. Of the 20 points for selection, 10 will be awarded through testing with our own indexes, and the other 10 through testing with your indexes.

6 Submission instructions

Carefully reread information about the input/output formats to be sure your code will not break automation during testing.

Create a README text file containing the following information. **Submissions without a README will receive zero points.**

- a line stating which is the top-level class of your code (the interpreter/harness that reads the input and produces output).
- an explanation of the logic for your index scan operator, specifying clearly:
  - where the `lowkey` and `highkey` are set
  - where in your code the grader can see different handling of clustered vs. unclustered indexes
  - an explanation on how you perform the root-to-leaf tree descent and which nodes are deserialized
- an explanation of the logic in your physical plan builder for separating out the portion of the selection which can/cannot be handled via the index
- any other information you want the grader to know, such as known bugs.

Submit via CMS a .zip archive containing:

- your Eclipse project folder
- a .jar of your project that can be run on the command line; name this `cs4321_p4.jar`
- your README file
- the `Experiments.pdf` file showing the results of your benchmarking, as discussed in Section 4.4
- an acknowledgments file if you consulted any external sources, as required under the academic integrity policy. Remember that if you reused any of the 4320 B+-tree code for this project, **you must give a proper acknowledgment**.