CS122 Assignment 1: NanoDB Set-Up and Storage Layer (100 points)

In this assignment you have these tasks to complete:

- Get your code repository and development environment set up
- Add support for tuple updates and deletion in NanoDB
- Improve the insert performance of the NanoDB heap file implementation, without substantially increasing the overall file size
- Submit your work along with a design document describing your efforts

The above tasks are described in detail, in the following sections.

Software Requirements

The NanoDB codebase requires the following tools and libraries to be present:

Java SE 8 Development Kit

- http://www.oracle.com/technetwork/java/javase/downloads/index.html
- The commands `javac`, `java`, and `javadoc` all need to be on your path. For example, typing "`javac -version`" should report 1.8 or higher as the version.
- You need the Java Development Kit (JDK), not just the Java Runtime Environment (JRE).

Apache Ant

- http://ant.apache.org/
- The `ant` command should be on your path. For example, typing "`which ant`" should show you where `ant` resides.

Additionally, you are encouraged to use a Java IDE (Integrated Development Environment) to work on NanoDB, because it is rather sizeable. The Community edition of IntelliJ IDEA is very nice, having some very powerful features; you can get it at http://www.jetbrains.com/idea/. However, if you prefer Eclipse or another IDE, or even just Emacs or Vim, feel free to use it!

NanoDB Repository Setup

The NanoDB codebase is substantial, and your team will be developing some complex features on the project. In situations like this, a version control system is absolutely essential; it allows you to make checkpoints of your work as you go, and it facilitates collaborative coding by multiple people. Additionally, you will submit your work using the version-control system so that we can retrieve and grade what you have done.

We will be using the Git distributed version control system for CS122. You can install Git onto your local machine from the following website: http://git-scm.com/. (It is already installed on the CMS cluster, a.k.a. "the CS cluster," if you decide to work there.) There are installers for all major OSes. Once Git is installed, you can follow the instructions below to get started on NanoDB.

At least one team member must also have a CMS cluster account for CS122. This is how you will retrieve the source code. If your repository is hosted on the CMS cluster, this is also how you will submit your homework. Make sure that someone on your team can log in to the CMS cluster as soon as possible.

Your team has a choice of using a Git respository hosted on the CMS cluster, or using a *private* Git repository on GitHub or Bitbucket.

- If you intend to use the CMS cluster for your Git repository, please email Donnie ASAP so that he can create an empty repository for you. If you use this approach, all teammates must have working cluster accounts.
- If you intend to use a Git repository hosted somewhere else, please make sure it is a private repository that is only accessible by the people who need to access it.

Steps:

1. First you will need to figure out a location to work. You can work on your local computer, on the CMS cluster, or wherever you feel comfortable doing so. (If you decide to change where you do your assignments, you will need to complete these setup instructions again.)

   I usually program within a `projects` directory on my laptop, but perhaps you will want to create a `cs122` directory for yourself to work in.

2. Once you have figured this out, you need to *clone* your team's repository into whatever location you have chosen. This means that you are making a complete copy of the original repository (which Git calls "`origin`") for yourself. You will be able to make whatever changes you want to this local repository without affecting the team's repository that you cloned.

   If you are using a 3rd-party Git service like GitHub or Bitbucket, you will probably need to read their instructions for cloning the team repository. However, these services will often tell you exactly what to type in order to clone your repository.

   If you are using the CMS cluster for your team's repository then you need to run the following command, replacing `username` with your CMS username, and `reponame` with your team's repository name. It will create a local copy of the repository, in a directory named `reponame`.

   ```
   git clone username@login.cms.caltech.edu:/cs/courses/cs122/teams/reponame
   ```

   (Note: If you are familiar with SSH, you can set this up to login for you automatically, but this is a bit beyond the scope of this document.)

   When you execute this command, you may see it say something like:

   ```
   Cloning into 'reponame'...
   warning: You appear to have cloned an empty repository.
   ```

   This is fine – you haven't put the NanoDB source code into your repository yet. You will do that in a few steps.

   Once you have cloned your team's repository, change into the directory that was just created.

3. Next, you must configure Git with your user information. (This is required so that we can tell what work each teammate has committed to the project.) The double-quotes in these commands are necessary if you have spaces in the values you specify.

   ```
   git config --global user.name "Your Name"

   git config --global user.email "your@email.tld"
   ```

   You will probably also find it helpful to turn on colorful output:

   ```
   git config --global color.ui true
   ```

4. **Now, <u>one teammate</u> needs to do the following:**

   (If multiple teammates do this, it will mess everything up! You have been warned…)

   In your local repository, run:

   ```
   git pull username@login.cms.caltech.edu:/cs/courses/cs122/nanodb
   ```

   This command will suck the entire NanoDB codebase down into your local repository.

   Next, make this code available in your team repository by running "`git push --all`". This will push your local repository's contents into your team's repository, so that your teammates can also access the NanoDB source.

   Once you have finished this, your teammates can also run "`git pull`" in their local repositories, to retrieve the NanoDB sources.

   Now you should be ready to experiment with NanoDB!

Optional Note: If you want to make it simpler to pull from the original NanoDB sources (e.g. when Donnie makes changes to the original sources, and you want to grab them), you can add the NanoDB repository location to your local Git configuration. Since it is highly likely that Donnie will make changes to the sources as the course progresses, you should probably do this.

If you run "`git remote -v`", you will see that your team's repository is now nicknamed "`origin`". This is the default remote that is used when you run "`git pull`" or "`git push`".

You can add other remotes, like this:

```
git remote add nanodb username@login.cms.caltech.edu:/cs/courses/cs122/nanodb
```

Now, when you run "`git remote -v`", you should see something like this:

```
nanodb  username@login.cms.caltech.edu:/cs/courses/cs122/nanodb (fetch)
nanodb  username@login.cms.caltech.edu:/cs/courses/cs122/nanodb (push)
origin  username@login.cms.caltech.edu:/cs/courses/cs122/teams/reponame (fetch)
origin  username@login.cms.caltech.edu:/cs/courses/cs122/teams/reponame (push)
```

(Your origin may be different, if you are hosting your team repository elsewhere. Also, the NanoDB repository will be read-only for you, so that you can't accidentally push your changes back to it.)

Once you have set things up this way, you can type "`git pull nanodb`" to get any changes to the original sources. You can use "`git pull`" to retrieve changes made by your teammates.

Git Repository Details

You should be aware that your local repository actually contains two components in one. First, you will see directories and files like `src`, `build.xml`, `test`, etc. These are actually not part of the Git repository itself; they are a *working copy* that you can edit separately. If you decide you don't like the changes you have made in your working copy, you can always revert back to the local repository's version with no problems.

When you are completely satisfied with your changes, then you can commit these changes to your repository. The repository itself is stored in a subdirectory named `.git`, which you can see if you type "`ls -al`". (Feel free to look in this directory, but don't modify anything in there unless you know exactly what you are doing.)

Compiling NanoDB with Ant

The first thing you should do is try to build NanoDB and generate the API documentation:

(in your local teamname directory)
```
ant compile javadoc
```

The `compile` task attempts to build all NanoDB sources, and the `javadoc` task generates the API documentation.

After this operation is completed, a new `build` directory will be created containing the results of the build process. **This directory should never be checked into Git.** To ensure that it is not, create a file named `.gitignore` in the directory, and add the line `build` to the file. Git will now ignore this file when it is examining your working copy. To be sure that Git remembers this, add the `.gitignore` file to your local repository by running the commands:

```
git add .gitignore

git commit
```

When you run the second command, Git will prompt you for a commit-log message, typically by starting `vim`. ***Make sure to always describe what you are doing when you commit a change to your repository, using the 50/72 format.***[1] This is critical in any software-development environment. If you fail to do this for every commit to your repository, you will lose substantial points on your assignments.

The message doesn't have to be too detailed; just be clear, complete and concise. A message like "Added .gitignore to ignore build directory" is perfect.

To view the API documentation for NanoDB, open the file `build/javadoc/index.html` in a web browser. The documentation is reasonably complete, although there are some gaps.

You can also test the NanoDB code by running `ant test`. The test results will be viewable in the file `build/results/index.html`. A lot of work is still needed on the test suite. Additionally, many tests will fail because you can't yet execute most SQL.

Other HTML output is generated by other build tasks; you can access all of this output by navigating to the file `build/index.html`. (I create a browser bookmark to this file on my local system.)

Finally, you can force a complete rebuild of the NanoDB code by typing `ant clean`. **This will delete the `build` directory.** For example, if you make substantial architectural changes, you would want to do `ant clean compile` to ensure that your changes didn't cause any unexpected breakages.

Running NanoDB

You might be tempted to try NanoDB out by writing a 15-line SQL reporting query with multiple subqueries, grouping, and all kinds of joins, but it isn't going to work. In fact, NanoDB is *extremely* limited – it can only run simple `SELECT` and `INSERT` statements at this point. You will be implementing these other features in the coming weeks.

[1] You can read about proper Git log formatting at http://chris.beams.io/posts/git-commit/

Also, you will notice many places in the NanoDB sources that say "TODO: IMPLEMENT" or "TODO:" followed by other notes. **You only have to implement the portions of NanoDB specified in each assignment.** Some of the TODOs will be addressed in future assignments, and others are just general work needed to improve NanoDB. You are only responsible for what each assignment says to complete.

You can start NanoDB by running:

```
./nanodb
Welcome to NanoDB.  Exit with EXIT or QUIT command.

CMD>
```

(Note: If you have the `rlwrap` utility on your computer, the `nanodb` script will automatically use it to start NanoDB, so that you can scroll through old commands, edit more easily, and so forth. You really should install this great utility!)

You can try creating a table:

```
CMD> create table t ( a integer );
Created table:  T

CMD> insert into t values ( 1 );
CMD> insert into t values ( 2 );
CMD> insert into t values ( 3 );
```

You can also perform basic selects:

```
CMD> select * from t;
+---+
| A |
+---+
| 1 |
| 2 |
| 3 |
+---+

SELECT took 0.003193 sec to evaluate.
Selected 3 rows.
CMD> select * from t where a < 3;
+---+
| A |
+---+
| 1 |
| 2 |
+---+

SELECT took 0.003350 sec to evaluate.
Selected 2 rows.
```

Finally, you can drop the table when you are finished. If you are able to do these things, congratulations, you are ready to start improving NanoDB.

You will notice that table files are created in the `./datafiles` directory. Similarly, the tests will create table files in the `./test_datafiles` directory. **Add both of these directories to your `.gitignore` file since they should never be checked into the repository.**

Debugging Your Work

NanoDB uses Apache Log4j pretty extensively, and the storage layer in particular will produce obscene amounts of logging output if so requested. Simply edit the `logging.conf` file, uncommenting the lines towards the end that set the log-levels of various components. If you set the log-level to DEBUG, you will be given reams of information about what is going on.

Of course, you can always add your own logging commands to the code. Just make sure that all of this is disabled before you submit your work; overly verbose output is likely to be penalized. **Adding simple print commands will be penalized; always use logging instead!**

NanoDB Tuple Updates and Deletion

As it stands, NanoDB does not yet support updating or deleting tuples! Your first tasks will be to implement the process of updating and deleting tuples in a table.

Note: Tables and indexes are both implemented as "tuple files" in NanoDB. This allows us to use the same storage implementation to manage both tables and indexes.

As noted in class, NanoDB tuple files have a simple structure. The major aspects are described here, along with the classes that provide the associated functionality. All of these classes are in the `edu.caltech.nanodb.storage.heapfile` package; you should go to this package and read through the classes mentioned here.

- Page 0 (or block 0; we use the terms interchangeably) is the header page of the table file, and stores details such as the table's schema. No tuples are stored in page 0.

  The `HeaderPage` class provides some lower-level operations for reading and writing fields within the header page; these operations are exposed as static methods.

- All other pages in the table file are data pages, implementing the slotted-page data structure as described in class. The `DataPage` class provides lower-level operations for manipulating data pages; again, these are exposed as static methods.

- The `HeapTupleFileManager` class provides the ability to create, open and delete tuple files that use the heap-file organization. When a table is opened for scanning, this class is used to open the table's corresponding tuple file.

- The `HeapTupleFile` class provides operations like scanning a tuple file, inserting a record into a tuple file, and so forth. The implementation of these operations relies heavily on the `HeaderPage` and `DataPage` classes.

- The `HeapFilePageTuple` class represents individual tuples whose data is backed by the file block that contains the tuple. You will see that the bulk of the page-tuple abstraction is implemented in the `edu.caltech.nanodb.storage.PageTuple` class, so that this tuple-storage code can be used in multiple file formats. (For example, if you were to implement a hash-file format, you could subclass `PageTuple` to allow reading and writing of tuples in your hash files.)

When a tuple is to be deleted, the `HeapTupleFile.deleteTuple()` method is called with the specific tuple to be deleted. This method determines the slot-index of the tuple, and then uses the `DataPage.deleteTuple()` method to delete the tuple at this slot.

Enable tuple deletion by completing the implementation of the `DataPage.deleteTuple()` method. Your implementation should conform to these guidelines:

- Reclaim the space that was previously occupied by the tuple being deleted by using the `deleteTupleDataRange()` helper function. Make sure you don't accidentally clobber adjacent tuples.
- Set the tuple's slot to the `EMPTY_SLOT` value.
- Finally, if there are empty slots at the end of the header, remove them so that this space can also be reclaimed. Remember that you cannot remove an empty slot if it is followed by one or more non-empty slots.

Enable tuple updating by completing the implementation of the `setNullColumnValue()` and `setNonNullColumnValue()` methods of the `PageTuple` class. These methods are somewhat tricky to get right. They are called by `PageTuple.setColumnValue()` to modify an existing tuple's column values.

- The `setNullColumnValue()` method sets the bit in the tuple's null-bitmask to indicate that the column's value is now NULL. Then it removes the space that the old value used to occupy. (Obviously, none of these steps are necessary if the old value for the column was NULL.)
- The `setNonNullColumnValue()` method must replace any existing value for the column with a new value.
- Some general implementation details are included as comments in these methods.

NanoDB Storage Performance

As discussed in class, NanoDB has a very simplistic approach to managing its heap files. In each heap file, NanoDB devotes the first page (page 0) to schema and other high-level details, and the remaining pages are devoted to tuple storage.

When NanoDB inserts a new tuple into a heap file, it follows a very simple process: Starting with page 1, it looks for enough space to store the tuple. If the page being considered has enough space then the tuple is added to that page.[2] If not, NanoDB goes on to page 2, and then page 3, and so forth. If NanoDB reaches the end of the file then it creates a new page and stores the tuple there.

This approach is very slow ($O(N^2)$) for adding a large number of tuples, but it does have the benefit of using space reasonably effectively. However, there is no reason why we shouldn't be able to have both benefits; fast inserts as well as efficient space usage.

The implementation of this strategy is in the `addTuple()` method of the `HeapTupleFile` class, in the `edu.caltech.nanodb.storage.heapfile` package.

Measuring Storage Performance

You can give NanoDB a try on this front by creating a simple table. After compiling NanoDB, start it with the `nanodb` script (or `nanodb.bat` on Windows), and create a simple table:

```
CREATE TABLE insert_perf (
    id INTEGER,
    str VARCHAR(200),
    num FLOAT
);
INSERT INTO insert_perf VALUES ( 1, 'hello', 3.14 );
```

[2] Note that this is a *first-fit* strategy, not a *best-fit* strategy.

```
    INSERT INTO insert_perf VALUES ( 2, 'goodbye', 6.28 );
    EXIT;
```

(A .sql file for this schema is in the schemas/insert-perf directory.)

You should now see a file named INSERT_PERF.tbl in the datafiles subdirectory. This is the table that we just created and then added the tuples to. It will be 16KB in size since the default page-size is 8KB.[3] If you want to get rid of this table, you can either delete the INSERT_PERF.tbl file from this directory, or you can use the command "DROP TABLE insert_perf;" in NanoDB.

How do we measure the actual performance? Wall-clock time (i.e. the actual time that the test takes to run) is not useful, because the test will be very dependent on computer hardware. Therefore, NanoDB includes a "SHOW STORAGE STATS" command, which will print out some statistics that will give us an approximation of the storage performance:

- storage.pagesRead and storage.pagesWritten are the number of disk pages that have been read and written, since the NanoDB server was started. (Note that this statistic is independent of the actual size of the pages.)
- storage.fileChanges is the number of times NanoDB accessed a different file from the previous file that was accessed. In a typical scenario using an HDD, a large disk seek would be performed every time a different file is accessed. For these tests, this value will be 1.
- storage.fileDistanceTraveled is an approximation of the absolute distance traveled within files as pages are accessed, in units of 512-byte sectors. For example, if sector 15 in the file is accessed, and then sector 29 is accessed, and then sector 3 is accessed, the distance traveled will be **abs**(29 – 15) + **abs**(3 – 29) = 40. In a typical scenario using an HDD, this would approximate the amount of distance that the disk head would have to move. It is an extremely rough approximation, though.

Ideally, as you improve your insert performance, you should see both the total number of accesses and the distance traveled decrease dramatically.

You can try some larger files, also provided in the schemas/insert-perf directory:

- ins20k.sql – 20,000 rows of data. With the default implementation, this file occupies 2.7MB of space, reads 3,306,267 pages, and has a "distance traveled" value of 105,144,752.
- ins50k.sql – 50,000 rows of data. With the default implementation, this file occupies 6.7MB of space, reads 20,645,580 pages, and has a "distance-traveled" value of 659,042,640.
- ins50k-del.sql – 50,000 rows of data, but also includes a smattering of DELETE statements that remove collections of rows from the table. Try this when you want to see how well you reclaim space! The default implementation's final result occupies 5MB of space.

You can try these operations simply by piping these files into the nanodb script, for example:

```
    ./nanodb < schemas/insert-perf/make-insperf.sql
    ./nanodb < schemas/insert-perf/ins20k.sql
```

Here are your tasks:

1. **Modify the storage format to improve the insert performance.** You can modify the HeapFileTableManager class in any way that you see fit. You can also modify the

[3] You can specify a different default page size by editing the nanodb script to pass -Dnanodb.pagesize=*n*, where *n* is a power of 2 between 2^9 and 2^{16}, inclusive. You can also specify a table's page-size in the CREATE TABLE command, like this: `create table t (a integer) properties (pagesize = 65536);`

`HeaderPage` and `DataPage` classes, which do the actual work of reading and writing table pages.

For example, you might want to add a linked-list of blocks with available space to the table-file, anchored from block 0. You might find it easiest to store such a list at the end of each block, but the design is entirely up to you. For example, you could subtract some space from the value that the `getTupleDataEnd()` method in the `DataPage` class returns, to open up some extra space for bookkeeping.

You could also create some kind of bitmap that records which blocks have available space. If you do this, your design should support the maximum number of blocks in NanoDB files, which is 65,536 (64K). Recall that page sizes may vary from 512B to 65,536B; your implementation should work properly regardless of a table's page size.

Don't forget to update your bookkeeping structures during tuple updates and deletes as well!

Do not use the table statistics as part of your implementation. This is not actually the intended use of table statistics. Table stats are updated infrequently by the database, and are therefore almost always out of date. Plus, stats-collection is not currently implemented; you will implement this yourself in a future assignment.

When you need to add a new page to a table file, you should be aware that the `DBFile` class provides a method `getNumPages()` that returns how many pages are currently in the file. Every `DBPage` has a reference to the `DBFile` object that it came from, so it should be very easy to tell where the new page should reside.

2. **Provide detailed documentation of your approach in the class-level comments for the `HeapTupleFile`.**

Additional requirements:

- Although you could easily improve performance by simply putting every new tuple in its own data page, you should endeavor to keep your data file to within 5% of the size of the original data file. Excessively large data files will be penalized.
- Additionally, you must actually modify the storage file format to improve this operation; do not simply modify the in-memory data tracked by the table manager.

Committing Work to Your Repository

As you work on your assignment, you may want to commit your changes as you get various parts of the project working. In fact, you are encouraged to do this! Nothing is more frustrating than finishing a complicated feature, then immediately mangling it as you start working on the next task. Commit your work anytime you finish anything you don't feel like doing again. At any point in your work, you can run the command "`git status`" to see what files have been modified in your working directory.

The command to commit changes to your repository is "`git commit`". However, it is important to understand Git's workflow for committing changes to the repository. Changes you make in your working directory will not immediately be included when you commit to your repository; rather, Git maintains a "staging area" of changes to be included in the next commit. In other words, you can make some changes that will be included in the commit, and other changes that will not be included in the commit. A file whose changes will be included in the next commit is described as being "staged" (i.e. its changes are in the staging area). A file whose changes will not be included in the next commit is "unstaged," or "modified but not staged."

To complicate this somewhat, files also fall into two categories: *tracked* files, which have been added to the repository and Git is managing them; and *untracked* files, which have not yet been added to the repository.

The upshot of all this is that if you want to add a new file to your repository, or you want to include changes of an existing file into your repository, you must run "`git add` *`filename`*" to include the file in the staging area. Then, these changes will be included in the next commit.

There is a simplification for when you haven't added any new files: you can run "`git commit -a`", which will perform the staging step as well as the commit step. However, if you create a brand new file, you still need to run "`git add` *`filename`*" on that new file before it will be committed.

Pushing Changes to the Team Repository

If you want to protect yourself from system crashes, you can also push your committed changes to your team's shared repository by running "`git push`" at any time. **This is strongly encouraged, since every year at least one or two students struggle with a crashed machine.** If you regularly push to the team repository, it will be relatively easy to get back online if your local system goes down in flames.

Submitting Your Work

When you are ready to submit your work, this is the process you should follow. **Only one member of your team should perform these steps.** If multiple people on your team do this, it will mess things up.

The first thing to do is tag your submission so that we can retrieve the correct version of your work:

```
git tag hw1
```

This will tag the current version of your repository with the name "`hw1`".

Once you have done this, you can push all of your changes to your CMS cluster repository:

```
git push --tags
```

The `--tags` argument will cause all tags in your local repository to be replicated into your CMS cluster repository. Once this is done, we will be able to retrieve the version of your work marked with this tag, so that we can grade it.

Note that if you decide to resubmit your work (e.g. because you found a bug that you needed to fix), you cannot reuse the same tag name again! Therefore, use a modified tag name that includes "`hw1`" in it, e.g. "`hw1-2`", so that we can still find your submission. Also, don't forget to push the new tag to your submission repository.

Finally, one team member should submit the completed Assignment 1 design document on the CS122 course Moodle.