CS 121: Relational Databases — Winter 2026

Project 01: Heap Files

Project Overview

In this project, you will complete and optimize the storage layer for NanoDB. While the first task is small in scope and is intended to help familiarize you with the codebase, the second task involves some more complex design and optimization decisions.

Setup

Register for the assignment by going to the Airtable link on the course website, which will create you a repository on GitLab with the starter code. You will be working in this same repository for every NanoDB assignment, though we will distribute occasional patches.

Then follow the NanoDB setup guide to install necessary tools and software.

NanoDB Tuples

NanoDB is a very large codebase, because database engines are very large applications. We'll call out specific areas you should familiarize yourself with, but there are many portions that are handled for you already (such as query parsing and file resolution).

NanoDB is divided into various packages, each dealing with some primary component of a database. In this project, we will be working within the `storage` package, which manages the database's interaction with disk; and the `storage.heapfile` package, which implements heap tuple files (discussed in class).

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

  The **`HeaderPage`** class provides some lower-level operations to work with fields inside this page.

- All other pages in the table file are data pages, using the slotted-page data structure described in class. The **`DataPage`** class again provides lower-level operations to work with these.

- The **`HeapTupleFileManager`** class provides functionality to create, open, and delete tuple files using heap-file organization.

- The **`HeapTupleFile`** class provides functionality to interact with a specific tuple file. This 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. Much of the page tuple abstraction is in the **`PageTuple`** class, so that we can share tuple storage code across multiple file formats.

Task 0

Familiarize yourself with the purpose of each of the files mentioned above.

Modifying Tuples

Tuple Deletion

When a tuple is to be deleted, the **`HeapTupleFile#deleteTuple()`** method is called, which determines the slot-index of the tuple and calls **`DataPage#deleteTuple()`**.

`DataPage#deleteTupleDataRange(DBPage dbPage, int off, int len)`

The `deleteTupleData` function removes a sequence of bytes from the current tuple data in the page, sliding tuple data below the offset forward to fill the gap. Remember to modify slots if necessary!

Reading `insertTupleDataRange` may be helpful.

`DataPage#deleteTuple(DBPage dbPage, int slot)`

The `deleteTuple` function follows these constraints as discussed in class:

- Reclaims space using `deleteTupleDataRange()`.
- Updates the slot value to `EMPTY_SLOT`.
- Reclaims trailing empty slots.

Task 1.1

Implement the above two functions to support tuple deletion (`DELETE`) in NanoDB.

Tuple Updates

When a tuple is updated, **`PageTuple#setColumnValue()`** will call one of two functions, depending on whether the new value is `null`:

`PageTuple#setNullColumnValue()`

Updates the bit in the tuple's null-bitmask, and removes the space the old value used to occupy.

`PageTuple#setNonNullColumnValue()`

Replaces the existing value with a new value, shifting data if necessary.

Don't worry about the case when the new value of the tuple overflows the page.

Some additional `PageTuple` functions you will find useful:

- `getColumnValueSize()`
  - Use `schema.getColumnInfo(iCol).getType()` to get a `ColumnType`.
- `isNullValue()` and `setNullFlag()`
- `writeNonNullValue()`

Task 1.2

Implement the above two functions to support tuple updates (`UPDATE`) in NanoDB.

Both functions should update the `valueOffsets` member by changing as few values as possible. (This means you shouldn't call `PageTuple#computeValueOffsets`, but it will be helpful to read it.)

All of Task 1 is tested together. You can run the tests for this task by using any of the following methods:

- Terminal: Run `make clean` then `make build` to build the target. Then run `make hw1`.
  - The outcome of the tests will be printed to the terminal and also available under the `target` directory in a file `target/surefire-reports/index.html`.
- IntelliJ: Select the `hw1` run configuration from the toolbar and click Run.

Expected Behavior: All 121 tests in the `hw1` group should pass.

Buffer Manager

The **`BufferManager`** is the database's page cache, and needs to decide which pages to keep in memory or evict. This decision is complicated by the fact that queries may be accessing many different tuples, and the page containing that tuple data must be kept in memory until the query is finished using it.

A simple and common approach to solve this problem is to "*pin*" disk pages in memory while they are being used; the `BufferManager` cannot evict a page that is currently pinned. When a page is no longer in use, it is "*unpinned*," allowing the `BufferManager` to evict the page. This is a simple reference-counting mechanism, but it works very well to let the `BufferManager` know what it may safely evict.

In NanoDB, both `DBPage` and `Tuple` objects may be pinned and unpinned. The `BufferManager` works with pages, and queries work with tuples. When a `DBPage` is retrieved from the `BufferManager`, its pin-count is always incremented so that it won't immediately be evicted out from under the requester. Similarly, when a new tuple is retrieved, the tuple pins its backing `DBPage` to prevent the page from disappearing out from under it. When a tuple is unpinned, it will automatically unpin its backing `DBPage`. These objects may also be re-pinned by various components; however, they must eventually be unpinned, or else the `BufferManager` will not be able to reclaim the corresponding buffer space.

Your implementation should properly unpin tuples (and disk pages, if necessary) when they are no longer needed. We can simplify this task by recognizing that we only need to unpin a tuple when it is no longer being used by the database engine.

As you update your tuple insertion, you should concentrate on the following areas:

- The `HeapTupleFile` implementation may need to pin and/or unpin pages as they are traversed. As an example, the **`HeapTupleFile#getFirstTuple()`** method already manages page-pinning and unpinning properly.
- The `HeapTupleFileManager` may also need to pin and unpin the header page.

The NanoDB Buffer Manager maintains a strict upper bound on the size of the buffer pool. The size is exposed as the "`nanodb.pagecache.size`" property, and can be read or written during operation. While you are implementing the proper pinning/unpinning steps in your database, you may want to leave the pool size very large (its max size is 1GiB), but once you have pinning/unpinning working properly then you should be able to shrink the buffer-pool size down to very small sizes (the minimum is 64KiB, the largest size of a single data page) and still see correct operation.

Optimizing Tuple Insertion

Initially, 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 uses a first-fit strategy to search pages in order. If there's no page with enough space, it creates a new page and stores the tuple there. This approach is very slow ($O(N^2)$ page reads) for adding a large number of tuples, but it does use space reasonably effectively. This approach is implemented in **`HeapTupleFile#addTuple()`**, and is similar to an implicit list allocator.

Some notes about performance:

- Key constraints cannot be enforced efficiently without indexes, which we will implement later in the term. At this point, though, NanoDB includes code to enforce them inefficiently, so you will want to avoid using tables with primary or foreign keys in your testing.
- The `BufferManager` is configured to unpin and flush all in-memory pages after every command. This is not a good strategy in normal operation, because if one command accesses a data page, it's likely that a subsequent command will access the same page.

  In this project, though, we care about the performance of the storage layer, not the cache – so we flush the `BufferManager` after every command to make issues more apparent by enabling the property `nanodb.flushAfterCmd`.

  However, it will also nag you about this if too many pages are still pinned at the end of your queries. Releasing pages when they are no longer necessary means your database will operate more efficiently.

We can compare the performance of heap file implementations with two metrics, which can be viewed with the `SHOW 'STORAGE' STATS;` command:

- `storage.pagesRead` and `storage.pagesWritten`: the number of disk pages read and written since the server was started. (Note that this is independent of the actual size of the pages.)
- `storage.fileDistanceTraveled`: an approximation of the absolute distance traveled within files as pages are accessed, in units of 512-byte sectors. In a typical scenario using an HDD, this would very roughly approximate the amount of distance that the disk head would have to move.

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

We will be benchmarking against some large files, provided in the `schemas/insert-perf` directory:

- `ins20k.sql` – 20,000 rows of data. With the initial implementation, this file occupies 2.7MB of space, reads ~3,000,000 pages, and has a "distance traveled" value of ~100,000,000.
- `ins50k.sql` – 50,000 rows of data. With the initial implementation, this file occupies 6.7MB of space, reads ~20,000,000 pages, and has a "distance traveled" value of ~1,000,000,000.

Task 2

Modify the storage format to improve the insert performance, by using the page directory approach discussed in class. Some constraints:

- Your implementation should not rely on a specific page size – they can vary from 512B up to 64K.
- Your implementation **must** support a large number of pages (up to ~64K).
- Your implementation **must not** use table statistics.
- Your data file should be within 3-5% of the initial implementation's file size. Excessively large data files will not receive credit.
- You must actually modify the storage file format; do not just modify the in-memory data.
- You should make sure to include necessary pin/unpin calls.

Some suggestions:

- When you need to add a new page to a table file, you should be aware of the method `DBFile#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.
- You will likely need to modify the `HeapTupleFile`, `HeaderPage`, and `DataPage` classes.
- You are free to add an additional class(es), such as to represent an internal page directory, though it's not necessary to do so.
- `HeapTupleFile` contains several functions to work with tuples contained inside the heap file (scan, insert, update, delete) – make sure that each of these functions is updated to work with your new storage format.

Verifying Your Work: We have provided a benchmark script to measure your improvements.

- Terminal: Run `make hw1-optimizations`.
- Intellij: Select the `hw1-optimizations` run configuration from the toolbar and click Run.

Expected Behavior: There are no direct tests for Task 2. Compare your results against the baseline numbers listed above, the numbers should decrease dramatically. Your database should continue to work correctly as well.

Design Document

Design Doc

Answer the questions in the file `doc/heapfile-design.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.