Assignment 6: Write-Ahead Logging

In this assignment you will complete the implementation of a write-ahead logging system for NanoDB that will allow the database to provide transaction atomicity and durability. The specific components to implement are:

- Enable transaction support in NanoDB, and disable the "flush after command" flag
- Update heap tuple files to log state changes to the write-ahead log
- Implement an atomic "force WAL" operation
- Implement code that enforces the Write-Ahead Logging Rule when the Buffer Manager evicts data pages to disk
- Implement transaction rollback using the write-ahead log
- Implement both the redo and undo phases of recovery processing

Toward the end of the assignment, you will again have to work with some low-level file structures. You are encouraged to use logging statements (i.e. `logger.debug(...)`) generously as you write your code; it will make debugging much easier for you in the long run.

Overview

NanoDB includes a very basic implementation of transaction processing and write-ahead logging that captures some of the basic concepts of the ARIES write-ahead logging mechanism. Transaction management is provided by the code in the `edu.caltech.nanodb.transactions` package, and write-ahead logging is provided by the code in the `edu.caltech.nanodb.storage.writeahead` package.

Transaction Demarcation

The transaction manager leverages the same event-handling framework that index management uses, but it uses the before/after command events rather than the before/after row events. Transaction management is straightforward:

The user can issue a "`BEGIN [ WORK ]`" or "`START TRANSACTION`" command to manually start a transaction. Subsequent DML statements will be part of the same transaction until either a "`COMMIT [ WORK ]`" or "`ROLLBACK [ WORK ]`" command is issued, ending the transaction.

- The user can also issue DDL/DML commands without using the above operations, and the transaction manager will automatically enclose each command within its own transaction.

The entry points for starting, committing, and rolling back transactions are the following methods on the `TransactionManager` class:

- `startTransaction()` – starts a transaction
- `commitTransaction()` – commits the current transaction
- `rollbackTransaction()` – rolls back the current transaction

The `TransactionStateUpdater` class implements the logic described above for starting and ending transactions, and the `[ Begin | Commit | Rollback ] TransactionCommand` classes in the `commands` package also call the above methods on the transaction manager.

Write-Ahead Logging

The write-ahead logger is managed as a component of the transaction manager; no code outside of the transaction manager really should need to interact with it. The main class to be aware of is the

`WALManager` class. It exposes `writeXXXX()` operations to write various log records to the WAL, a `rollbackTransaction()` method that uses the WAL to rollback the current transaction, and a `doRecovery()` entry-point that performs recovery processing when NanoDB starts.

The write-ahead logger implements an *extremely* simplified version of the ARIES logging and recovery mechanism. The logger does not support checkpointing, although it does support all the record types discussed in class: begin/commit/rollback, update, and redo-only update. Other relevant details are as follows:

- The write-ahead log is stored in one or more files in the `datafiles` directory, the files being named `wal-`*`nnnnn`*`.log`. (In this assignment, and in the source code, the term "write-ahead log" refers to the entire aggregate log, not an individual file.) The file number *nnnnn* starts at 0 and increments to 65535, at which point it wraps around back to 0. Each file is constrained to be no more than ~10MB in size, but the end of the file will never be a partial record; the last record in the file will always be complete.

- Each log record is identified by a `LogSequenceNumber` (LSN) value, consisting of the WAL file number, and the offset from the start of the file that the record is stored at. Note that the LSN does not include a page number; the page number can be computed from the file-offset and the WAL file's page size, when it is needed.

- The write-ahead log is loaded and saved through the Storage Manager, just as with all data files. This means that WAL pages will often be cached within the buffer manager to minimize IO costs. This also means that care must be taken to write out the WAL at appropriate points during the transaction lifecycle. The operation of ensuring that the WAL is written out to disk at least as far a particular LSN is called forcing the WAL to that LSN. This operation is provided on the Transaction Manager. We will discuss this operation in more detail momentarily.

- As with ARIES logging, every `DBPage` object in memory has a `pageLSN` value associated with it, which is the most recent LSN that applies to the page. Unlike ARIES, this value is not recorded to disk! Rather, the Buffer Manager uses this value to enforce the write-ahead logging rule: when dirty pages must be evicted from the buffer manager, it forces the WAL out to the largest `pageLSN` of the dirty pages being evicted, and then it writes the dirty pages back to disk. This ensures that all changes written by the database are properly reflected in the WAL first.

  The `WALManager` class updates the `pageLSN` value on `DBPage` objects when the `writeUpdatePageRecord()` and `writeRedoOnlyUpdatePageRecord()` methods are called. You shouldn't have to manage these values yourself.

  Note that some dirty `DBPage` objects will not have a `pageLSN` value! Specifically, pages in files that do not support transactions will not have this value set, and WAL pages in particular will not specify this value. In general, data pages relating to the write-ahead log are not transacted. (It wouldn't make sense to log changes to WAL pages in the write-ahead log.)

Dirty Page Management

Managing data pages in the context of write-ahead logging is somewhat involved. For example, when a page is modified, the original version of the page must be kept so that the write-ahead logger can generate the "old value" and "new value" to store in the log. These kinds of capabilities are provided on the `DBPage` class – you will notice that when a clean page is made dirty (by `DBPage.setDirty(true)`), it also makes a copy of the clean version of the page.

Similarly, when a change to a dirty data page has been logged to the WAL, it is no longer necessary to record the original version of the page because the WAL now reflects the changes. Thus, the WAL Manager calls `DBPage.syncOldPageData()` when it writes an "update" record, so that the "old value" of the page matches the "new value" of the page. To see why this is important, imagine the following sequence of events against a specific page:

- Tuple 1 in the page is updated. An "update" record is written to the WAL, recording the portion of the page that contains the old page-data for tuple 1, and the new page-data for tuple 1.
- Next, tuple 2 in the same page is updated, in the same transaction. Again, an "update" record is written to the WAL, but it should only contain the old and new page-data for tuple 2, not the data for tuple 1.

This is why the logger calls `DBPage.syncOldPageData()` after writing an "update" record, so that each time an update is recorded, it only reflects the changes since the previous "update" record.

Logging Writes to Table Files

Logging the changes to data pages is very dependent on the kind of data file being managed. For example, in heap files, adding or removing a tuple involves changing the page's slot array, as well as changing the tuple-data region of the page. Thus, any code modifying a data page page must inform the write-ahead logger when it is time to log changes to the WAL. This is done via the `DBPage.logDBPageWrite()` method. (If the page is not dirty, this is a no-op.)

Currently, the heap tuple file implementation doesn't include any of these calls, so changes will not appear in the write-ahead log; therefore, they will not be durable or atomic. This will be one of the first tasks to complete, calling `logDBPageWrite()` at appropriate times so that operations against tables will be properly logged in the WAL and governed by transactions.

Page Debugging

Unfortunately, debugging the write-ahead logger tends to be difficult. You will need to look at logging output as the database performs transaction rollback, redo processing, undo processing, and so forth. Be prepared.

Sometimes you will need to see the contents of data pages that have been modified, etc. There are a few helper functions on `DBPage` to help you with this:

- The `toFormattedString()` method will return a formatted string containing all data in the `DBPage`. Be warned; it is verbose. However, sometimes it is the easiest way to identify issues.
- The `getChangesAsString()` method is similar in that it will print out the contents of a `DBPage`, but it will only show the differences between the old and new versions of the page. This can be very helpful to see exactly what changes were made on a particular data page.

The CRASH Command

In order to simplify basic testing, NanoDB has a `CRASH` command that will cause the database to terminate immediately without saving any buffered data. It is very easy to use:

```
CMD> CRASH;
Goodbye, cruel world!  I'm taking your data with me!!!
$ [your shell prompt]
```

This command is useful for basic testing, but it will not allow exhaustive testing of your transaction processing system. The reason is simple: databases can also crash at far more inopportune times

than just at the start of a command. We may put together something more advanced (e.g. sending a `kill -9` to the JVM), but be aware of the limitations of this command when testing your code.

To get us a little closer to that, you can also give an argument to the `CRASH` command, the number of seconds to wait before crashing:

```
CMD> CRASH 5;
CMD> INSERT INTO t VALUES (1, 'a', 1.1);
CMD> INSERT INTO t VALU
Goodbye, cruel world!  I'm taking your data with me!!!
$ [your shell prompt]
```

This version of the command starts a background thread that will crash the database after the specified number of seconds has passed. This allows a script to go on to other things, so that the database crash can interrupt more serious operations, like writing data out to disk, etc. There is still an element of chance in it, so it'll be like playing Russian Roulette, at least until you have transaction atomicity and durability working properly. After that, crashes shouldn't matter.

The FLUSH Command

NanoDB also has a `FLUSH` command that will cause the database to flush any buffered data in the Buffer Manager to disk.

```
CMD> FLUSH;
Flushing all unwritten data to disk.
CMD>
```

You can use this command to make sure that the buffer manager and write-ahead logger work together properly. This can be particularly useful when testing crashes, since only commits force the WAL out to disk. For example, you can rollback a transaction (or just leave it incomplete), flush the database to disk, and then crash the database, like this:

```
CMD> ROLLBACK;
CMD> FLUSH;
Flushing all unwritten data to disk.
CMD> CRASH;
Goodbye, cruel world!  I'm taking your data with me!!!
```

Next, you can try to restart NanoDB and see if it does the proper thing during recovery processing.

PLEASE READ: Important Limitations!

Currently, NanoDB does not support transacted DDL, for a variety of reasons. Don't even try it. ☺ The same goes for indexes; indexes are not transacted.

The write-ahead logger supports logs that span multiple files, but this functionality has not been heavily tested. You are expected to support multiple logs, but if you run into strange bugs that don't seem to be in your code, please talk to Donnie and/or the TAs right away.

Step 1: Enable Transaction Processing in NanoDB

Before you can work on the write-ahead logger implementation, you need to enable transaction processing in NanoDB. Do this by editing the script that starts the database. If you are on UNIX or Mac OS X, you will edit the `nanodb` file and add the underlined text:

```
# To set the page-size to use, add "-Dnanodb.pagesize=2048" to JAVA_OPTS.
# To enable transaction processing, add "-Dnanodb.txns=on" to JAVA_OPTS.
JAVA_OPTS="-Dlog4j.configuration=logging.conf -Dnanodb.txns=on"
```

If you are on Windows, you need to edit the `nanodb.bat` file:

```
rem To set the page-size to use, add "-Dnanodb.pagesize=2048" to JAVA_OPTS.
rem To enable transaction processing, add "-Dnanodb.txns=on" to JAVA_OPTS.
set JAVA_OPTS=-Dlog4j.configuration=logging.conf -Dnanodb.txns=on
```

NOTE: After doing this, you should delete the contents of your `datafiles` directory, since this change does affect what files are created and managed in this directory.

Next, you need to turn off a flag in the `edu.caltech.nanodb.server.NanoDBServer` class. This flag causes the database to flush the buffer manager after every command, but this won't exercise the write-ahead logger as effectively. Find this line and change the flag from true to false:

```
static final boolean FLUSH_DATA_AFTER_CMD = false;
```

This change will become active the next time you recompile NanoDB.

Step 2: Add Logging to Heap Tuple Files

As stated earlier, heap tuple files currently don't log any of their changes. Fortunately, this is not terribly difficult to add, especially since the number of pages that change for any given operation will tend to be small.

You need to update the heap tuple file code to properly log all changes to the write-ahead log. Go through both the `HeapTupleFile` and `HeapTupleFileManager` classes, looking for any place that you write to a `DBPage`. For example, adding/updating/removing a tuple, storing statistics, updating non-full-page lists, would all be places where changes are made to the tuple file. Wherever you finish modifying the contents of a `DBPage`, add in a call to the Storage Manager's `logDBPageWrite(DBPage)` method as the last step of that operation. This will ensure that the changes to the page are written to the write-ahead log.

Try to avoid calling `logDBPageWrite()` on a page multiple times for a single operation. For example, it would be wasteful to call `logDBPageWrite()` once after updating a page's slots, and then again after updating the page's tuple data. Rather, it should be called once, after all changes have been made to the page.

Step 3: Implement an Atomic Force-WAL Operation

The `TransactionManager` class provides a `forceWAL(LogSequenceNumber)` method that is critical for atomic and durable transaction processing. This method must ensure that the write-ahead log is written out to the specified LSN, and sync'd to disk, before it returns. This method is called every time a transaction is committed to ensure that the committed transaction is reflected in the WAL, and it is also called anytime dirty pages must be written to disk during a transaction.

The only problem is, our write-ahead log can span multiple files. And, there is a second issue that the buffer manager could output WAL pages in any order it wants. We have no guarantees that the database won't crash during this process, which could leave our WAL completely corrupted. (*You could say it was "WALloped," ha ha!*) Therefore, careful thought must be given to how the WAL is written to disk, to ensure it is updated atomically.

The transaction manager also manages a separate file called `txnstate.dat`, which it uses to record various critical values related to transaction management:

- The "next transaction ID" value to use the next time the database is started.

- The "first LSN" value for the write-ahead log. We will discuss this value shortly; it is critical to recovery processing.

- The "next LSN" value for the write-ahead log. This is the location where the next WAL record would be stored in the WAL, as reflected on disk. In other words, all WAL records between `firstLSN` (inclusive) and `nextLSN` (exclusive) are considered to be valid.

  Note that the `WALManager` also maintains a `nextLSN` value in memory, but the `WALManager`'s value will be larger than this value if there are unsaved WAL records/pages in the buffer manager. The `txnstate.dat` file's "next LSN" value should always reflect the end of the write-ahead log that is stored on disk.

This file is intentionally small; it should *always* be atomically writable. This means that it should fit within one disk sector (as small as 512 bytes, but possibly up to 4KiB in size with modern disks), so that we don't have any *write-tearing* occur (i.e. when a buffer that spans an atomic-write boundary is written or sync'd to the disk, and the system crashes as that boundary is being crossed).

Implement the transaction manager's `forceWAL(LogSequenceNumber)` method to atomically and durably write the WAL, as far as the specified LSN. Specifically, this method must ensure that buffered WAL pages containing records up to the specified value are written out of the buffer manager, and that the involved WAL files are also sync'd to disk. This method will need to look at the current transaction state to tell what range of WAL records must be written out, and will use the buffer manager to ensure that pages containing those records are indeed written and sync'd to disk. It will also need to modify the current transaction state.

Of course, you will also need to make sure that you update the WAL in an atomic and durable way, particularly in the context of the issues outlined earlier, namely having multiple WAL files and buffer pages being written out in no particular order.

Here are additional requirements and suggestions:

- **Make sure to explain in comments exactly why your implementation is atomic and durable! Failure to do so will lose points.** (You will also include this explanation in your design document.)
- You should be aware that LSNs point to the start of a WAL record, but this method must guarantee that the entire record is written out. The `LogSequenceNumber` class has a `getRecordSize()` method you will find useful for this.
- There will definitely be times when this operation will be a no-op! Make sure to detect when the force operation does not need to be performed, and return immediately in those cases. (Hint: LSNs are comparable; you can use `lsn1.compareTo(lsn2)` to determine if one LSN comes before or after another LSN.)
- Similarly, only write the pages of the WAL that must be written; if you write additional pages beyond what is specified by the argument to `forceWAL()`, you will be penalized. (Remember, the idea is to avoid unnecessary IOs, because they are slow.)
- The `TransactionManager` provides a `storeTxnStateToFile()` method, which will write and sync the `txnstate.dat` file to disk. It records the LSN value stored in `txnStateNextLSN`.
- The `BufferManager` class also provides several very helpful operations for you to use:

  You can call `BufferManager.getFile()` to see if a file is currently open. NanoDB always syncs files when it closes them, so syncing should only be required if a file is currently open.

The `BufferManager` also provides several flavors of `writeDBFile()` to allow some or all dirty pages to be written to disk, followed by an optional sync. Note that the sync is always performed if requested, because dirty pages may have been evicted from the buffer manager but still cached within the OS.

- You can use the `WALManager.getWALFileName(int)` method to get the filename of a WAL file based on its file number.

Once this operation is available, you can use it to make sure that the database will follow the write-ahead logging rule. This is the focus of our next task…

Step 4: Enforce the Write-Ahead Logging Rule

When the Buffer Manager needs more memory, it evicts some of the pages loaded into memory, taking care to write dirty pages back to disk. This is all fine and good, except that the Buffer Manager doesn't know anything about transactions or write-ahead logging. If the Buffer Manager's page-writes are not coordinated with the Transaction Manager's updates to the write-ahead log, then there is no way we can enforce durability and atomicity in the database.

The solution to this problem is that the Buffer Manager can notify other components when dirty pages are about to be written to disk. The Transaction Manager registers a handler on the Buffer Manager to receive notifications when dirty pages are about to be evicted; this is where we can ensure that the write-ahead logging rule is enforced. The method that the Buffer Manager will call is `TransactionManager.beforeWriteDirtyPages()`.

Implement this method on the Transaction Manager so that the database will follow the WAL rule when any dirty pages are written back to disk. This is a pretty straightforward operation to complete, especially after the `forceWAL()` method from the last step has been implemented. In fact, this method will call `forceWAL()`; the only complexity is figuring out what LSN to pass to the method. There are more details in the comments for `beforeWriteDirtyPages()`.

Once you complete this operation, the database will follow the write-ahead logging rule. Additionally, you should see the database updating the write-ahead log with details of committed transactions, which would make them durable.

Except for the fact that we don't have any recovery processing yet.

Not only that, but if we end up needing to roll back a transaction, NanoDB still can't do that.

Step 5: Implement Transaction Rollback

For the final two tasks, you will need to become familiar with the format of the write-ahead log. The Javadocs for the `edu.caltech.nanodb.storage.writeahead` package describe the log format in detail; you should refer to this documentation as you complete the last two steps.

The `WALManager` class provides a `rollbackTransaction()` method used during normal operation to rollback the current transaction a client is participating in. **You will need to complete the implementation of this method.** Note that this method is not used during recovery processing; transactions are rolled back in a different way during recovery.

The `rollbackTransaction()` method operates as described in class. NanoDB manages session state for each client, and part of that session state is the LSN of the last WAL record issued for the current transaction. This is where rollback begins. Additionally, all WAL records except for the

"start transaction" record include a "previous LSN" value, allowing the sequence of operations performed in the transaction to be rolled back in reverse order.

Of course, this is a relatively straightforward task, because rollback only cares about two records: the "start" record which means rollback is done, and the "update page" record which specifies a change to roll back. Any other record encountered during rollback would indicate a serious error, worthy of throwing a `WALFileException` to report a corrupt write-ahead log.

The `WALManager` provides a helper method `getWALFileReader(LogSequenceNumber)`, that opens (or retrieves from the buffer manager) the WAL file corresponding to the specified LSN, and seeks to the file-offset specified by the LSN. The returned object is a `DBFileReader`, which allows our page-based database files to be read and written in a more stream-like format. You should always use this `getWALFileReader()` method to get a reader for a given LSN, to avoid unnecessary complexity in your implementation. (It's not the most efficient thing to do, but we can live with it for now.)

As the transaction is rolled back, new records must be written to the write-ahead log. Use the `writeTxnRecord(WALRecordType)` and `writeRedoOnlyUpdatePageRecord(DBPage,int, byte[])` methods to write these records as you rollback. Note that there are two versions of these methods; use the versions that do not require transaction info to be explicitly specified! Otherwise, the client's transaction state will not be updated properly during rollback.

The last important detail is how to undo the changes to a particular data page. The WAL record will specify the file and page number that was modified; load this page using the storage manager and then use the `applyUndoAndGenRedoOnlyData()` helper to undo the changes on the page and generate the redo-only data at the same time. Note that this method has very specific requirements on the `DBFileReader` position when the method is invoked; read the method's Javadoc for details!

Once you have completed the `rollbackTransaction()` method, you should be able to successfully start and rollback transactions in your database. Give it a try:

1. Create a simple table and add a few rows to it.
2. Then, start a new transaction with `BEGIN`, and try either changing some rows and/or adding new rows to the table. Query the table to verify that your changes are all visible!
3. Finally, rollback the transaction by typing `ROLLBACK`. If your code is working properly, you should be able to query the table after rollback and see the original contents. Pretty cool!

If you have any issues during your testing, you will probably want to delete all files in your `datafiles` directory. Corrupt or invalid write-ahead logs will leave NanoDB *very* confused.

Once this step is finished, you will have atomic transactions in NanoDB. However, they are still not durable until recovery processing is completed. Onward!

Step 6: Implement Redo and Undo Processing

This is probably the most complicated part of the write-ahead logging implementation, because it involves moving forward and backward through a log containing variable-size records. Just like with the index implementation, this low-level stuff can become a little grungy. Such is life.

The `WALManager` provides an entry-point for recovery processing called `doRecovery()`. The `TransactionManager` invokes this method at database startup using the values loaded from the

`txnstate.dat` file. We already discussed the `nextLSN` value stored in this file; it is just past the last known-good record in the WAL. The `firstLSN` value is important for recovery: it is the lower bound on the set of records necessary for recovery processing. The `firstLSN` value provides the following guarantees:

- The `firstLSN` value specifies a point in time where all data files reflect all changes that are recorded in the write-ahead log before that LSN. Thus, redo processing can start from this point. (Without checkpointing, this is the best we can do.)
- Additionally, there are no incomplete transactions at the `firstLSN` point in the log. This is important because when undo processing is traversing the log backwards, we know it won't have to look earlier than `firstLSN` in the log.

Unfortunately, without checkpoints, we really cannot move `firstLSN` forward until the completion of recovery processing. Thus, this is the only time that NanoDB moves this value forward.

Redo Processing

Redo processing is implemented in `performRedo()`. This method takes a `RecoveryInfo` object, which keeps track of what transactions are incomplete, along with the last LSN seen for each transaction. This second detail is important: when incomplete transactions are rolled back, we must properly chain new WAL records into the earlier records for each transaction.

You must complete the implementation of this method. The implementation should be mostly straightforward; update the `RecoveryInfo` object as appropriate for the WAL records you encounter, and be aware that for this method you must make sure the `walReader` is always moved past each WAL record.

In situations where a change must be redone, you can use the `applyRedo()`method to perform the task. As with the `applyUndoAndGenRedoOnlyData()` method, the `walReader` must be positioned just after the segment-count when the method is invoked. (In this case, `applyRedo()` could read the number of segments itself, but it seems beneficial to make both methods behave similarly.)

No WAL records should be emitted during redo processing!

Undo Processing

Undo processing is implemented in `performUndo()`. This method takes the same `RecoveryInfo` object populated by `performRedo()`, since now the set of incomplete transactions should be known. This method behaves similarly to the `rollbackTransaction()` method, except that it is rolling back all incomplete transactions at the same time. As before, it only cares about "start" records and "update" records; in particular, "redo-only update" records are ignored during this phase. (Of course, a "commit" or "rollback" record should never be seen for an incomplete transaction…)

You must complete the implementation of this method as well. Note that the part of this method that you have to implement is relatively small compared to the overall size of the method; the bulk of the code is devoted to navigating backward through the write-ahead logs, a rather tedious task that doesn't lend itself to factoring into another method.

As stated before, you will find this implementation very similar to the `rollbackTransaction()` implementation, but there will be some important changes. You will need to emit additional WAL records as you rollback incomplete transactions, but this time you must use the information in the `RecoveryInfo` object to manage each transaction's "previous LSN" value. You cannot use the `writeXXXX()` methods that pull these details from the session state; your logs will be corrupt!

Step 7: Testing

Obviously, with a system as complex as write-ahead logging, there can be a lot of subtle errors in your implementation. Therefore, you should test your code as extensively as you can to make sure it works. You should test from simple to complicated, so that you can isolate issues as quickly as possible. Also, putting generous logging statements in your code will give you a lot of help in identifying issues.

Finally, don't forget that you had to add logging to your heap-file implementation, so it would be a very good idea to test inserts, updates and deletes, involving one page and multiple pages, and so forth. That way you can ensure that your database logs all changes properly.

Here are some ideas for testing. In between tests, you should probably delete the contents of your `datafiles` directory so that corrupt files don't send you on a wild goose chase.

1. Start the database, then shut it down, then start it again. See if this works without error.
2. Start the database, create a table, insert a few rows (relying on autocommit to commit each operation separately). Shut it down.

   Start it again, and make sure the table is the same, and that recovery processing ran successfully.

   Start it again, and see if recovery processing is now a no-op, since the `firstLSN` value in `txnstate.dat` will have been moved forward by the previous startup.
3. Start the database, create a table. Explicitly start a transaction, and make multiple changes (e.g. insert multiple rows) within that single transaction. Then commit it.

   Try the same steps as in the previous test, to see if the data stays the same after restarting the database. Make sure recovery processing runs without errors.
4. Start the database, create a table, insert a few rows (relying on autocommit again). Explicitly start a transaction, perform a few changes (e.g. change values, insert rows), then rollback the transaction. Make sure the original table data is restored.

   Shut down the database, then restart it. Make sure recovery processing ran successfully. Verify that the data is unchanged.

   Shut down and restart the database again, and check the data again. (Remember, for the second restart, recovery processing should be a no-op.)
5. Repeat the same steps as for 4, but use `CRASH` instead of `ROLLBACK` to see if things will work properly.

   The previous test tends to work with no problems, because the WAL isn't forced out until a commit or a database shutdown. Therefore, you should also try issuing a `FLUSH` before the `CRASH`; this will make sure that your recovery processing can handle an incomplete transaction in the WAL.
6. Start the database, create a table. Perform some modifications and commit them. Then perform other modifications and roll them back. Finally, perform more modifications and commit them. Make sure the data is correct.

   Shut down and restart the database, and verify that recovery works correctly. Verify the data is valid.

   Shut down and restart the database again, and check the data again.

7. Start the database, create a table. Add some data to the table.

   Start a transaction, perform some operations within the transaction, then abort the transaction.

   Start another transaction, perform some more operations, then abort that transaction.

   Make sure the original table data is still there!

   Shut down (or flush and crash) the database, then restart it and make sure the contents of the table are still correct.

If you are reasonably rigorous about your testing, you should have pretty good confidence that your write-ahead logging is working correctly.

You will also find a file in `schemas/writeahead/basic-tests.sql`. **You can't run this file directly against NanoDB**; it requires manual steps that aren't yet scriptable. But, you can use it to do some of the testing specified above.

Extra Credit!

Automating the above tests would be extremely useful, and extra credit points will definitely be awarded for good tests that exercise the transaction processing capabilities of the database. Note that some kinds of testing will be easier than others; specifically, verifying the recovery-processing part of the system would be extremely difficult given the current mechanism for starting and stopping the database.

It would probably be best to break tests into categories based on complexity:

- Basic tests – against a file with only one data page
  - Start a transaction, insert/update/delete data, commit. Verify data is still present.
  - Start a transaction, insert/update/delete data, roll back. Verify that changes are gone.
  - Sequences of two transactions doing simple operations, e.g. commit/commit, commit/abort, abort/abort.
  - And so forth.
- More advanced tests – against a file with multiple data pages
  - Similar tests as for the basic tests, except with operations that will modify multiple data pages.

Finally, if you figure out a way to test recovery processing effectively in an automated way, this will get a lot of points!

Submitting Your Assignment

When you are finished with the coding part of the assignment, tag it with a `hw7` tag as usual, and then push all of your changes to your repository on the CMS cluster.