berkeley-cs186 / fa19-moocbase

Fall 2019, MOOCbase skeleton code

2 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

es1024 Release (hw2) - 09-16-2019 Latest commit `8df4315` 6 hours ago

images	Release (hw2) - 09-16-2019	6 hours ago
src	Release (hw2) - 09-16-2019	6 hours ago
.gitignore	Initial Release (hw0)	20 days ago
README.md	Initial Release (hw0)	20 days ago
courses.csv	Initial Release (hw0)	20 days ago
enrollments.csv	Initial Release (hw0)	20 days ago
hw0-README.md	Initial Release (hw0)	20 days ago
hw2-README.md	Release (hw2) - 09-16-2019	6 hours ago
hw3-README.md	Initial Release (hw0)	20 days ago
hw4-README.md	Initial Release (hw0)	20 days ago
hw5-README.md	Initial Release (hw0)	20 days ago
intellij-test-setup.md	Initial Release (hw0)	20 days ago
pom.xml	Initial Release (hw0)	20 days ago
public.key	Release (hw2) - 09-16-2019	6 hours ago
students.csv	Initial Release (hw0)	20 days ago
turn_in.py	Release (hw2) - 09-16-2019	6 hours ago

README.md

MOOCbase

This repo contains a bare-bones database implementation, which supports executing simple transactions in series. In the homeworks of this class, you will be adding to this implementation, adding support for B+ tree indices, efficient join algorithms, query optimization, multigranularity locking to support concurrent execution of transactions, and database recovery.

As you will be working with this codebase for the rest of the semester, it is a good idea to get familiar with it.

Overview

In this document, we explain

- how to fetch the released code from GitHub
- how to fetch any updates to the released code
- how to setup a local development environment
- how to run tests inside the CS186 docker container and using IntelliJ
- how to submit your code to turn in homeworks
- how to reset your docker container
- the general architecture of the released code

Fetching the released code

The **first** time you work on this codebase, run the following command (this is the same command given in HW0; you do not need to run this again if you are working through HW0):

```
docker run --name cs186 -v "/path/to/cs186/directory/on/your/machine:/cs186" -it cs186/environment /bin/bash
```

This should start a bash shell (type `exit` to exit). You should only need to run this one time in this class.

To start the container, run:

```
docker start -ai cs186
```

After some notifications, you should get a prompt like this:

```
ubuntu@1891ee9ee645:/$
```

This indicates that you are inside the container; to exit the container, type `exit` or `ctrl+D` at the bash prompt.

While inside the container, navigate to the shared directory:

```
cd /cs186
```

Clone this repo:

```
git clone https://github.com/berkeley-cs186/fa19-moocbase.git
```

If you get an error like `Could not resolve host: github.com`, try restarting your docker machine (exit the container and run `docker-machine restart`) or restarting your computer.

Now, navigate into the newly created directory:

```
cd fa19-moocbase
```

To test that everything is working correctly, run:

```
mvn clean test -P system
```

There should not be any failures.

Fetching any updates to the released code

In a perfect world, we would never have to update the released code, because it would be perfectly free of bugs. Unfortunately, bugs do surface from time to time, and you may have to fetch updates. We will post on Piazza whenever fetching updates is necessary. The following instructions explain how to do so.

Inside the container, navigate to the cloned repo:

```
cd /cs186/fa19-moocbase
```

Commit any changes you have made so far:

```
git add --all .
git commit -m "commit message"
```

If you get the following error:

```
*** Please tell me who you are.

Run
```

```
  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: empty ident name (for <ubuntu@7623beeffa12.(none)>) not allowed
```

then run the two suggested commands and try again.

Pull the changes:

```
git pull --rebase
```

If there are any conflicts, resolve them and run `git rebase --continue`. If you need help resolving merge conflicts, please come to office hours.

Setting up your local development environment

You are free to use any text editor or IDE to complete the homeworks, but **we will build and test your code in the docker container with maven**.

We recommend setting up a more local development environment by installing Java 8 locally (the version our Docker container runs) and using an IDE such as IntelliJ.

Java 8 downloads

If you have another version of Java installed, it's probably fine to use it, as long as you do not use any features not in Java 8. You should run tests somewhat frequently inside the container to make sure that your code works with our setup.

To import the project into IntelliJ, make sure that you import as a maven project (select the pom.xml file when importing). Make sure that you can compile your code and run tests (it's ok if there are a lot of failed tests - you haven't begun implementing anything yet!). You should also make sure that you can run the debugger and step through code.

Running tests

The code in this repository comes with a set of tests that can be run. These test cases are not comprehensive, and we suggest writing your own to ensure that your code is correct.

To run all the tests for a particular homework, start the container, navigate to the cloned repo (`cd /cs186/fa19-moocbase`), and run the following command:

```
mvn clean test -DHW=n
```

but replace `n` with one of: `0`, `2`, `3Part1`, `3Part2`, `4Part1`, `4Part2`, `5`.

Before submitting any homeworks, be sure to run tests inside your container. **We will not be accepting "the test ran successfully in my IDE" as an excuse -- you are responsible for making sure the tests run successfully *in the docker container*.**

A (very) common problem that students in past semester ran into when they did not run tests in Docker before submitting was encountering an error similar to the following after submitting:

```
[INFO] -----------------------------------------------------------
[ERROR] COMPILATION ERROR
[INFO] -----------------------------------------------------------
[ERROR] /cs186/fa19-moocbase/src/main/java/edu/berkeley/cs186/database/index/BPlusTr
ee.java:[3, 45] package com.sun.internal.rngom.parse.host does not exist
[INFO] 1 error
[INFO] -----------------------------------------------------------
[INFO] ----------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ----------------------------------------------------------------------
```

When your IDE lists autocomplete options, there are sometimes classes/enums/etc. from non-standard libraries. If you select one by accident, your IDE may automatically add the import for it, which stays even after you delete the incorrectly autocompleted word.

As the import is (probably) unused, just go to the specified file and line and delete the import.

We will be running tests with the same Docker image as you, so running tests in Docker before submitting lets you check that everything will work as intended when we run tests to grade your submission.

Running tests in IntelliJ

If you are using IntelliJ, and wish to run the same tests that `mvn clean test -DHW=n` runs, follow the instructions in the following document:

IntelliJ setup

Submitting homeworks

To submit a homework, **start your container**, navigate to the cloned repo, and run:

```
python3 turn_in.py
```

This will generate a zip file. Upload the zip file to the Homework Submission assignment on edX.

Note that you are only allowed to modify certain files for each homework, and changes to other files you are not allowed to modify will be discarded when we run tests.

Resetting the Docker container

If things are not working in the Docker container, a first step for troubleshooting is to start a container from the image again. This will discard any changes made in the container's filesystem, but will not discard changes made inside a mounted folder (i.e. `/cs186`).

Outside of Docker, first delete the container:

```
docker container rm cs186
```

Then, create it again (this is the command you ran back in HW0):

```
docker run --name cs186 -v "<pathname-to-directory-on-your-machine>:/cs186" -it cs186/environment /bin/bash
```

The code

The code is located in the `src/main/java/edu/berkeley/cs186/database` directory, while the tests are located in the `src/test/java/edu/berkeley/cs186/database` directory.

common

The `common` directory contains bits of useful code and general interfaces that are not limited to any one part of the codebase.

concurrency

The `concurrency` directory contains a skeleton for adding multigranularity locking to the database. You will be implementing this in HW4.

databox

Our database has, like most DBMS's, a type system distinct from that of the programming language used to implement the DBMS. (Our DBMS doesn't quite provide SQL types either, but it's modeled on a simplified version of SQL types).

The `databox` directory contains classes which represents values stored in a database, as well as their types. The various `DataBox` classes represent values of certain types, whereas the `Type` class represents types used in the database.

An example:

```
DataBox x = new IntDataBox(42); // The integer value '42'.
Type t = Type.intType();        // The type 'int'.
Type xsType = x.type();         // Get x's type, which is Type.intType().
int y = x.getInt();             // Get x's value: 42.
String s = x.getString();       // An exception is thrown, since x is not a string.
```

index

The `index` directory contains a skeleton for implementing B+ tree indices. You will be implementing this in HW2.

memory

The `memory` directory contains classes for managing the loading of data into and out of memory (in other words, buffer management).

The `BufferFrame` class represents a single buffer frame (page in the buffer pool) and supports pinning/unpinning and reading/writing to the buffer frame. All reads and writes require the frame be pinned (which is often done via the `requireValidFrame` method, which reloads data from disk if necessary, and then returns a pinned frame for the page).

The `BufferManager` interface is the public interface for the buffer manager of our DBMS.

The `BufferManagerImpl` class implements a buffer manager using a write-back buffer cache with configurable eviction policy. It is responsible for fetching pages (via the disk space manager) into buffer frames, and returns Page objects to allow for manipulation of data in memory.

The `Page` class represents a single page. When data in the page is accessed or modified, it delegates reads/writes to the underlying buffer frame containing the page.

The `EvictionPolicy` interface defines a few methods that determine how the buffer manager evicts pages from memory when necessary. Implementations of these include the `LRUEvictionPolicy` (for LRU) and `ClockEvictionPolicy` (for clock).

io

The `io` directory contains classes for managing data on-disk (in other words, disk space management).

The `DiskSpaceManager` interface is the public interface for the disk space manager of our DBMS.

The `DiskSpaceMangerImpl` class is the implementation of the disk space manager, which maps groups of pages (partitions) to OS-level files, assigns each page a virtual page number, and loads/writes these pages from/to disk.

query

The `query` directory contains classes for managing and manipulating queries.

The various operator classes are query operators (pieces of a query), some of which you will be implementing in HW3.

The `QueryPlan` class represents a plan for executing a query (which we will be covering in more detail later in the semester). It currently executes the query as given (runs things in logical order, and performs joins in the order given), but you will be implementing a query optimizer in HW3 to run the query in a more efficient manner.

recovery

The `recovery` directory contains a skeleton for implementing database recovery a la ARIES. You will be implementing this in HW5.

table

The `table` directory contains classes representing entire tables and records.

The `Table` class is, as the name suggests, a table in our database. See the comments at the top of this class for information on how table data is layed out on pages.

The `Schema` class represents the *schema* of a table (a list of column names and their types).

The `Record` class represents a record of a table (a single row). Records are made up of multiple DataBoxes (one for each column of the table it belongs to).

The `RecordId` class identifies a single record in a table.

The `HeapFile` interface is the interface for a heap file that the `Table` class uses to find pages to write data to.

The `PageDirectory` class is an implementation of `HeapFile` that uses a page directory.

table/stats

The `table/stats` directory contains classes for keeping track of statistics of a table. These are used to compare the costs of different query plans, when you implement query optimization in HW4.

Transaction.java

The `Transaction` interface is the *public* interface of a transaction - it contains methods that users of the database use to query and manipulate data.

This interface is partially implemented by the `AbstractTransaction` abstract class, and fully implemented in the `Database.Transaction` inner class.

TransactionContext.java

The `TransactionContext` interface is the *internal* interface of a transaction - it contains methods tied to the current transaction that internal methods (such as a table record fetch) may utilize.

The current running transaction's transaction context is set at the beginning of a `Database.Transaction` call (and available through the static `getCurrentTransaction` method) and unset at the end of the call.

This interface is partially implemented by the `AbstractTransactionContext` abstract class, and fully implemented in the `Database.TransactionContext` inner class.

Database.java

The `Database` class represents the entire database. It is the public interface of our database - we do not parse SQL statements in our database, and instead, users of our database use it like a Java library.

All work is done in transactions, so to use the database, a user would start a transaction with `Database#beginTransaction`, then call some of `Transaction`'s numerous methods to perform selects, inserts, and updates.

For example:

```java
Database db = new Database("database-dir");

try (Transaction t1 = db.beginTransaction()) {
    Schema s = new Schema(
        Arrays.asList("id", "firstName", "lastName"),
        Arrays.asList(Type.intType(), Type.stringType(10), Type.stringType(10))
    );
    t1.createTable(s, "table1");
    t1.insert("table1", Arrays.asList(
        new IntDataBox(1),
        new StringDataBox("John", 10),
        new StringDataBox("Doe", 10)
    ));
    t1.insert("table1", Arrays.asList(
        new IntDataBox(2),
        new StringDataBox("Jane", 10),
        new StringDataBox("Doe", 10)
    ));
    t1.commit();
}

try (Transaction t2 = db.beginTransaction()) {
    // .query("table1") is how you run "SELECT * FROM table1"
    Iterator<Record> iter = t2.query("table1").execute();
    System.out.println(iter.next()); // prints [1, John, Doe]
    System.out.println(iter.next()); // prints [2, Jane, Doe]

    t2.commit();
}

db.close();
```

More complex queries can be found in `src/test/java/edu/berkeley/cs186/database/TestDatabase.java`.