CS 174B Database Systems Design and Implementation
Winter 2018
Course Project
Due: Thursday, March 8, 2018 at 11:59PM PT

---

**Note:**

- Make sure to re-read the "Policy on Academic Integrity". Copying any part of the project will automatically result in a failed grade. **Note:** We will be using sophisticated software to check for plagiarism.
- The course webpage announcements will contain any updates or corrections. Please check occasionally: http://www.cs.ucsb.edu/~xyan/classes/CS174B-2018winter/
- This is a two student project. Each student must write each other's user id at the top of their pdf report submission.
- Contact the course TA if you have any questions: Yi Ding yding@cs.ucsb.edu

---

# 1 Project Overview

In this project, you will implement a small database. The database consists of a console, a disk-based B+ tree for text data, and several operations for using your database. Your database will support storage operations associated with manipulation of an index file. The data input is meant to simulate what occurs in a storage system using a B+ tree. You will need to make use of the page layout, buffer management, rollback and concurrency algorithms that we cover in class to accomplish this task

This project should be implemented using C/C++. You will need to create a makefile that can build your project. Your project should build by executing `make` in the project root directory. You are not allowed to use any third party frameworks or non-native libraries for your implementation. You can use standard and native libraries that are installed on CSIL for every user (the definition of standard or native libraries includes any piece of code that does not require installation or downloading). Also keep in mind that your submission can't exceed 10MBs.

# 2 Data Input

We have preprocessed Yelp reviews for this assignment. Please download and extract the dataset `gunzip yelp_reviews.gz` from the project directory on the course website. There should be one big text file where each line represents a Yelp review. It should look something like this:

```
DOC0 super simple place but amazing no....
DOC1 small unassuming place that changes their...
DOC2 lester 's is located in a beautiful...
...
```

Your task is to build a database system that supports some common operations on this file. Note that you do not need to store the contents of the file in your index, your index only needs to maintain the mapping between the document names and words that appear in the document. For example if I search for the word `super` then `DOC0` would be returned in the above small example. Other documents that contain the word `super` should also be returned.

# 3 Index File

You will first need to create an application to generate the index file for this data. You should implement support for the following commands:

`> build_index [text_file] [index_file] -p [page_size]`

`text_file`: Name of all document to be indexed. 1 document per line.

`index_file`: The name of the index file to be stored on disk

`page_size`: The page or block size, default is 8096 bytes.

Your index must store the mapping between a word and the document name(s) that the given word maps to.

# 4 Supported Features

You will also need to implement console support for accessing your database. Whenever your console is started, you will be given a parameter `-b number_of_pages`. This parameter will set the size of the buffer in main memory. You **cannot** load more than this amount of. The default size is 8,096 pages. That is, you cannot load more than 8,096 pages from the index in main memory unless otherwise specified. You need a simple file management routine and a buffer manage routine to manage pages in your index file and the buffer.

Your console should support the following operations:

`> load [index_file]`: Loads a B+ index.

`> merge [index_file]`: Merge the current index file with the second index file, and update the index file on the disk.

`> quit`: close the console.

`> insert [document_name]`: Insert the word:document_name pair into the index.

`> delete [document_name]`: Removes a document with the given name from the index.

`> count [keyword]`: Counts a keyword by printing the number of documents that contain this keyword

`> search [keyword`$_1$`, keyword`$_2$`, ...]`: Search multiple key words. Prints the name(s) of documents that contain all of these keywords

`> printpath [keyword]`: Prints the search path (the id of pages that are accessed when we search the keyword in the B+ tree)

`> page [i]`: Displays the contents of the $i^{th}$ page

`> range [keyword`$_1$`, keyword`$_2$`]`: Range query. Print all of the keywords between keyword$_1$ and keyword$_2$ where keyword$_1$ < keyword$_2$

In addition to any output, you should print 1) the time it takes to perform the operation, 2) number of pages accessed, including pages in buffer, 3) number of pages read from or written to disk.

# 5 Rollback & Concurrency

Rollback and concurrency are difficult to implement, so be very careful here and make sure to thoroughly test your code.

`> rollback [n]:` Rollback n insertion/deletion operations. That is, restore the previous index file by discarding the recent n insertion and deletion operations, we do not consider the merge operation here. The rollback function should also be able to rollback operations that happened in previous sessions. A session includes all the operations that take place between the start and the close of a console.

**Concurrency** Suppose there are two open consoles. When two insertion operations happen simultaneously to the same index file, how is the file consistency maintained? This functionality will be tested by simultaneously running two merge operations and examining the resulting index file.

# 6 Project Report

You will submit a 4-page `report.pdf` together with the source code. Please put yours and your partners user id at the top of the report. The report should be no more than 4 pages which clearly explains the design of your system. Your report should discuss the following topics with an emphasis on algorithms:

**Architecture** Overview of your database architecture

**Algorithms** Describe your design and implementation choices for Page Layout, Buffer Management, Rollback and Concurrency

**Challenges** Difficulties encountered during implementation and how they were solved

**Improvements** Where you think your database is lacking and what improvements you can make

Clarity is important. Communicate in a way so that your design and implementation choices are easy to understand. You are welcome to use figures to aid in your discussion.

# 7 Submission Instructions

Use the CSIL turnin command to submit the necessary files and directories. Be sure to include your `report.pdf` and `README` files. Note that all directory and file names are case sensitive. For this assignment you need to issue the following command:

```
> turnin final@cs174b final
```

Once you are finished developing your code, copy all necessary source files (including header files, source code files, the pdf report, makefile, etc.) into a directory named `final`. The grader will compile and execute your programs. You do not need to include any binary or data files. **Your code must compile without referencing any external files or libraries not included in your submission**. Standard header files, such as iostream.h, or other native libraries and modules are fine to use as long as they are available on CSIL.

You must use C/C++ for this project. You may use any OS/compiler version for development, but your final code must compile and run on the CSIL machines. **Make sure to check that your code compiles and runs on the CSIL machines before submission**

# 8 Grading

A majority of your grade relies on a correct implementation of your algorithm. We will test for correctness and for execution speed.

**30%** Detailed project report on design and implementation of algorithms.

**10%** Project compiles and runs with given directions.

**50%** Correctness of implementation.

**10%** Performance