CSE 510: Database Mgmt Sys Implementn (Phase 1)

Title: MiniBase Database

Group members

1. Kunj Patel (khpatel8@asu.edu , 1213184152)
2. Khushal Modi (kcmodi@asu.edu, 1219446332)
3. Mishal Shah (mshah31@asu.edu, 1217195421)
4. Monil Nisar (mnisar2@asu.edu, 1217111805)
5. Samip Thakkar (sthakka2@asu.edu, 1217104967)
6. Saurabh Gaggar (sgaggar1@asu.edu, 1219666643)

Abstract

This phase builds up understanding of working of a relational database(MiniBase). We compile the database for our machine and run various tests to understand the implementation of the provided Database Management System.

1. Introduction:

1.1. Terminology:

- RDBMS: Relational DataBase Management System
- BufferManager: Component that is a specialized type of cache between Disk and CPU.
- DiskSpaceManager: Component that helps in communication with the disk.
- HeapFile: A type of storage system that helps in efficient retrieval.
- BTree: A indexing structure essentially used to store records in the proper manner so that they can be retrieved easily. MiniBase uses B+ tree for indexing.
- Join: An operation that combines two or more tables in a manner similar to cartesian product.

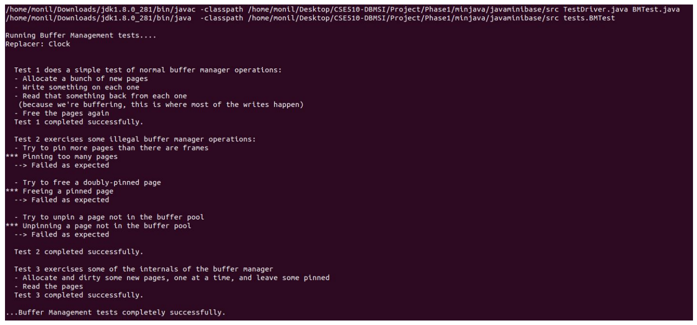
1.2. Goal Description:

The goal of this phase is to explore and understand the implementation of MiniBase. Understanding the bare minimum components of a RDBMS and how they communicate internally. We do this by studying and executing the Tests that are already defined.

2. Description of the tests:

2.1. BufferManager Test:

The BufferManager works as an intermediate cache between disk and CPU. It stores pages in main memory so the costly disk operations can be avoided for pages that are in the buffer.



```
/home/monil/Downloads/jdk1.8.0_281/bin/javac -classpath /home/monil/Desktop/CSE510-DBMSI/Project/Phase1/minjava/javaminibase/src TestDriver.java BMTest.java
/home/monil/Downloads/jdk1.8.0_281/bin/java  -classpath /home/monil/Desktop/CSE510-DBMSI/Project/Phase1/minjava/javaminibase/src tests.BMTest

Running Buffer Management tests....
Replacer: Clock


  Test 1 does a simple test of normal buffer manager operations:
  - Allocate a bunch of new pages
  - Write something on each one
  - Read that something back from each one
   (because we're buffering, this is where most of the writes happen)
  - Free the pages again
  Test 1 completed successfully.

  Test 2 exercises some illegal buffer manager operations:
  - Try to pin more pages than there are frames
*** Pinning too many pages
  --> Failed as expected

  - Try to free a doubly-pinned page
*** Freeing a pinned page
  --> Failed as expected

  - Try to unpin a page not in the buffer pool
*** Unpinning a page not in the buffer pool
  --> Failed as expected

  Test 2 completed successfully.

  Test 3 exercises some of the internals of the buffer manager
  - Allocate and dirty some new pages, one at a time, and leave some pinned
  - Read the pages
  Test 3 completed successfully.

...Buffer Management tests completely successfully.
```

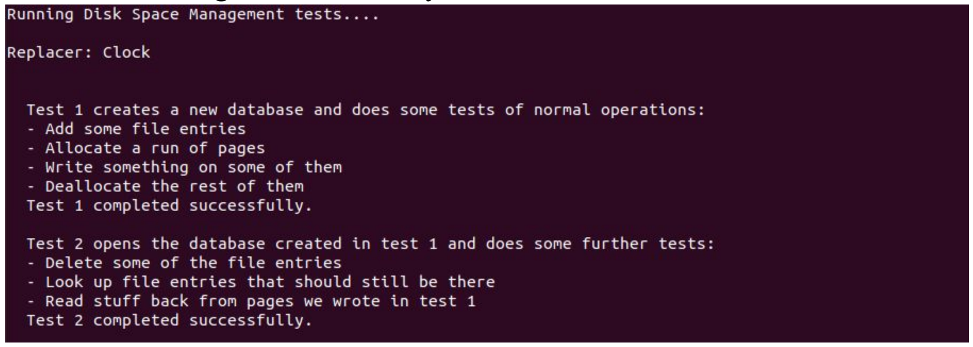
Test 1: This tests the regular operations of a buffer manager like creating pages, pinning them, writing and reading from those pages.

Test 2: This tests pinns more pages than available frame space and tries unpinning pages not in the memory pool. These tests are expected to fail as these operations are not valid.

Test 3: This test makes the page dirty by writing on it and then unpins some of them. Afterwards it reads the data from all pages. This test should be successful because the operations are logical and correct.

2.2. DiskSpaceManagement Test:

The DiskSpaceManager is a part that interacts directly with the disk and manages the pages in the disk. Operations involve creating and allocating pages, modifying them and deleting them efficiently.



```
Running Disk Space Management tests....

Replacer: Clock


  Test 1 creates a new database and does some tests of normal operations:
  - Add some file entries
  - Allocate a run of pages
  - Write something on some of them
  - Deallocate the rest of them
  Test 1 completed successfully.

  Test 2 opens the database created in test 1 and does some further tests:
  - Delete some of the file entries
  - Look up file entries that should still be there
  - Read stuff back from pages we wrote in test 1
  Test 2 completed successfully.
```

Test 1: This test creates and allocates 30 pages and tries to write on the first 20 of them. Then deletes the remaining pages. It also creates 6 files. This checks for valid disk operations as allocating, de-allocating and writing on disk pages. These tests run successfully as we have enough disk storage to accommodate them.

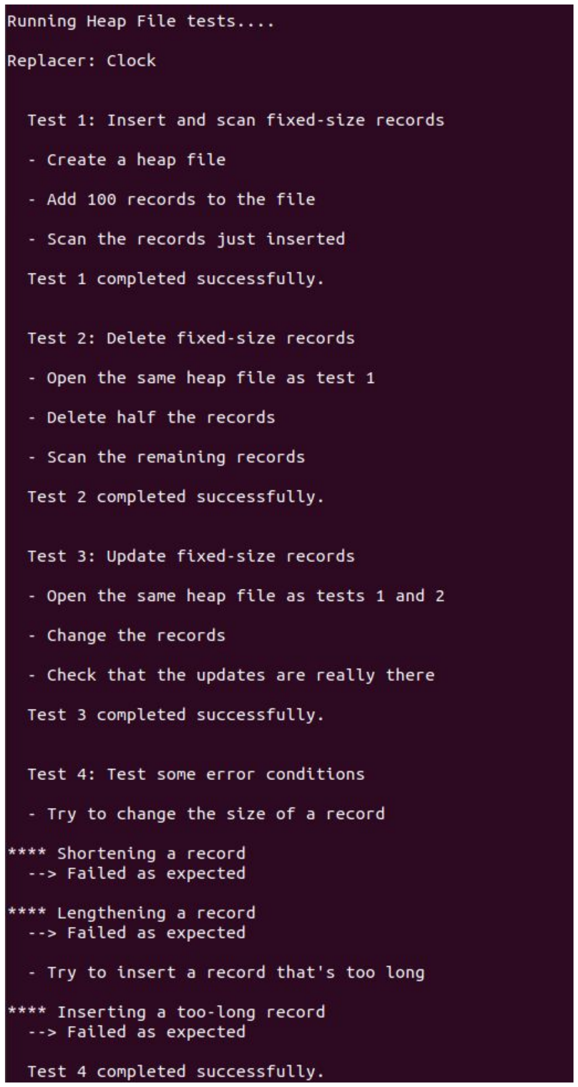
Task 2: Thie test deletes some file entries that we did in test1 and then tries to read the remaining files. Also it reads data that we wrote in test1, making sure that data has not changed. This test should be successful because it simply reads the pages we stored earlier.

Test 3: These tests are for checking that the system returns proper results for files that are not present. So we must fail when trying to access or modify a deleted file or a non-existing file or when trying to create a file with the same name or with an invalid name. We must also fail when we execute a run of pages with invalid values like negatives or big values.

Test 4: These tests are to check page allocation and deallocation in the space map. It checks for extreme cases when the database becomes full, tries to allocate more pages, fails as expected. It deallocates pages from between and reallocates them. This should work as we have space.

2.3. HeapFile Test:

HeapFiles are used to store records in an organised manner so that we can efficiently scan, insert and delete records.



```
Running Heap File tests....

Replacer: Clock


  Test 1: Insert and scan fixed-size records

  - Create a heap file

  - Add 100 records to the file

  - Scan the records just inserted

  Test 1 completed successfully.


  Test 2: Delete fixed-size records

  - Open the same heap file as test 1

  - Delete half the records

  - Scan the remaining records

  Test 2 completed successfully.


  Test 3: Update fixed-size records

  - Open the same heap file as tests 1 and 2

  - Change the records

  - Check that the updates are really there

  Test 3 completed successfully.


  Test 4: Test some error conditions

  - Try to change the size of a record

**** Shortening a record
  --> Failed as expected

**** Lengthening a record
  --> Failed as expected

  - Try to insert a record that's too long

**** Inserting a too-long record
  --> Failed as expected

  Test 4 completed successfully.
```

Test 1: This test checks for normal HeapFile methods like creating a file, inserting a record and then accessing them by scanning the file. Thes test is expected to pass, and this test tells us that file creation and insertion operations are valid.

Test 2: This test is based on test1, here we delete the odd records and then try to access the remaining even records. This is expected to pass and this tells us that the delete operation is valid.

Test 3: This test checks the update method. We modify the records by updating the value by multiplying by 7, and then we scan all records to check if everything went okay. This test is expected to pass and this will tell us about the implementation of delete functionality.

Test 4: This test is for some error/boundary conditions like inserting invalid records. This is expected to fail because the implementation forbids us to update the length of an existing record and not allowing us to insert a too long record.

2.4. BTree Test:

BTree is a type of index structure that helps to organize data by creating a balanced tree and updating the records such that the height of the tree is balanced. Thus, making the insert/update/delete operations efficient.

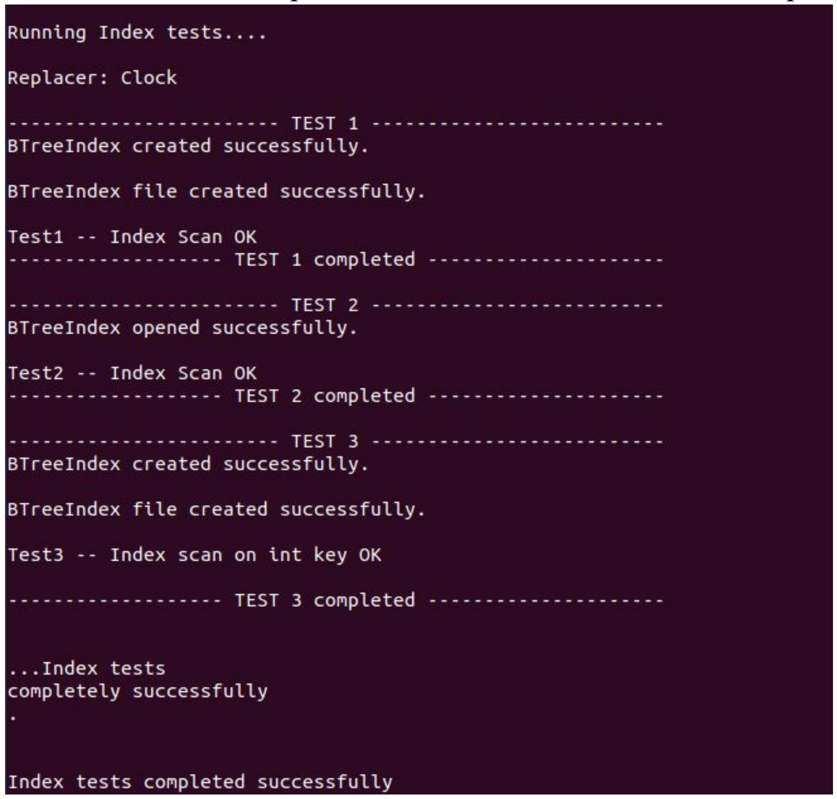
Test 1: This test checks the self adjusting property of B+ tree, that it expands and shrinks automatically as the number of entries change. We start by inserting 10240 records into the first file. We get the B-tree with 3 levels, then we delete many records. The B-Tree shrinks and it becomes 2 levels. This indicates the self adjusting property of the B-Tree. This is expected to run successfully as this is the correct implementation of B+Tree.

Test 2: This test is to check simultaneous updates on multiple index. We create 2 files with many records in it and then delete some records from them alternatively and check for that record. This is expected to pass and we should be able to manage multiple index structures.

Test 3: This test is to check the sequential property of the index structure. So we insert records randomly and then perform a sequential scan from start. This is expected to pass, as the index structure maintains the sequence of records.

2.5. Index Test:

This is part of "File and Access Methods", where the program creates an index structure and inserts, updates and deletes the records in a HeapFile.



```
Running Index tests....

Replacer: Clock

------------------------ TEST 1 -------------------------
BTreeIndex created successfully.

BTreeIndex file created successfully.

Test1 -- Index Scan OK
------------------- TEST 1 completed --------------------

------------------------ TEST 2 -------------------------
BTreeIndex opened successfully.

Test2 -- Index Scan OK
------------------- TEST 2 completed --------------------

------------------------ TEST 3 -------------------------
BTreeIndex created successfully.

BTreeIndex file created successfully.

Test3 -- Index scan on int key OK

------------------- TEST 3 completed --------------------


...Index tests
completely successfully
.


Index tests completed successfully
```

Test 1: Tests the insertion of records by creating unsorted files with data and creating an index structure to store the sorted index. This test is expected to pass, and it signifies that whenever we insert records, they can be accessed in sorted manner because of B+Tree index structure.

Test 2: This test is for searching a record based on index. It uses the HeapFile and Index structure we created in test1. We search the term "dsilva" in the records we had inserted in test1. This should pass as "dsilva" is included in the data we inserted in test1.

Test 3: This test is for inserting random data of different types (int, float & string) and then creating indexes on the integer field. Then it tries to get the records for which the field has value less than 900 and greater than 100. This is expected to pass and also the scan on the projected data will be in-order (sorted).

2.6. Join Test:

The JoinTest runs 6 different queries on a database consisting of 3 files (Sailor, Boat and Reserve). It uses a different approach to find the result. All the queries should produce expected results to show success in database creation and query execution.

Query 1: A select query with join operation on two tables. Performed without indexing, but a sort-merge join is used.

Query 2: A select query with join operation on three tables. Performed indexing on one of the tables with a nested loop join.

Query 3: A select query very similar to Query 1. Performed without indexing, but a sort-merge join.

Query 4: It is Query3 but with duplicate elimination.

Query 5: Similar to Query 3, but we have multiple selections.

Query 6: Runs like Query 5, but different constraints and nested-loop join instead of sort-merge.

2.7. Sort Test:

Sorting the records based on the value provided. This is mainly used when we perform join operations that would be very expensive otherwise. Also it helps when performing a search on index.

Test 1: This test checks the sorting functionality. It inserts some records into a database and sorts them in ascending. It is expected to pass and it represents sorting of string based tuples.

Test 2: This test is also similar to Test1, but it does so in descending order.

Test 3: This test is to check sorting capability of tuples, based on some particular field. It sorts in ascending using integer field and in descending suing float field. This should pass and represents that that system can handle tuples as well.

Test 4: This test checks for anomalies during sorting like different number of records or records not in order. This test verifies the integrity of the sorting mechanism.

2.8. Sort-Merge Test:

Similar to the above Sort Test.

Query 1: A select query with join operation on two tables. Performed without indexing, but a sort-merge join is used.

Query 2: A select query with join operation on three tables. Performed indexing on one of the tables with a nested loop join.

Query 3: A select query very similar to Query 1. Performed without indexing, but a sort-merge join.

Query 4: It is Query3 but with duplicate elimination.

Query 5: Similar to Query 3, but we have multiple selections.

3. Conclusions:

This phase provided an opportunity to understand the implementation of the MiniBase by referring to the tests. This also helped to understand how different components are implemented at the source code level.

4. Bibliography:

4.1. "The Minibase Home Page" : https://research.cs.wisc.edu/coral/minibase/minibase.html

4.2. "SystemDefs": https://www.eecs.yorku.ca/course_archive/2013-14/W/4411/proj/javadoc/global/SystemDefs.html

5. Appendix:

5.1. Typescript is present in the submission zip file.