CSE 510: Database Management Systems Implementation

Project Phase 1

Report by Venkata Satya Sai Dheeraj Akula

Group 6

Venkata Satya Sai Dheeraj Akula
Riya Jignesh Brahmbhatt
Haard Mehta
Rutva Krishnakant Patel
Karthik Nishant Sekhar
Nithya Vardhan Reddy Veerati

Abstract

The goal of the project phase 1 is to understand the internal working details of Minibase database management system by running tests on it. This report is a simplified explanation of many such tests that check the functionality of minibase. These tests are run on various components of the minibase like Buffer manager, Disk space manager, heap files, B+ tree indices, sort and merge join functions .

Introduction

The source code for bare bone implementation of database management system called the minibase is provided in a tar file which had to be extracted on a linux machine. Then the make command had to be run to build the binaries from the source code. Then the make test command is used to run the tests. Many tests ran automatically without any interaction from the user. The B tree tests were interactive in the sense that it takes arguments from command line. All the results from the tests are

captured by the typescript file which has been added to the report at the end. I am hereby going to explain these tests.

Description of the tests

Buffer Management Tests

Test1

Test: In this test we are allocating a bunch of pages to our database. Since the granularity of the hard disk is limited the secondary memory is blocked in the form of pages.
Behavior: PASS
Why: We are allocating a healthy amount of pages to our database so no error is returned.

Test: In this test we are writing data on the pages. First we need to pin a page that is copy the page from secondary memory to primary memory so that the database can access it. This process of transferring memory is handled by the Buffer Manager. After we finished writing we are unpinning the page. We are also setting the dirty boolean of the page to ON which means that the page has been modified by the database and needs to get updated in the secondary memory.
Behavior: PASS
Why: We are pinning a healthy amount of pages so no error is returned.

Test: In this test we are reading the values that are written on each page. We need to again pin a page because we need to access it. Now we check if the values that we are reading are equal to the values that were written in the previous step. In this way we can ensure consistency of our operations.After we finished reading we are unpinning the page. We are also setting the dirty boolean of the page to ON.
Behavior: PASS
Why: We are pinning the same number of pages and accessing them as in the previous step so we get no errors.

Test: In the final step we are freeing the pages that got allocated. This step ensures that the pages in the buffer whose dirty boolean is set to ON is being written to the database.
Behavior: PASS
Why: freePage() function defined in the BufMgr.java source file works correctly so we get no errors.

Test2

Test: In this test we are trying to pin more pages than they are buffer frames. We are initially pinning pages such that there are no more unpinned buffers available. On top of that we are adding one more page after this.
Behavior: FAIL
Why: The test fails because we are pinning more pages that the available buffer frames. The buffer frames are limited in number because they have to be assigned on the main memory.

Test: In this test we are pinning the first page twice and then freeing this doubly pinned first page. There is a PinCount variable for every frame in the buffer which keeps track of the number of pins. The PinCount gets incremented if the frame is pinned and decremented if the frame is unpinned. Now the pin count variable gets incremented twice.
Behavior: FAIL
Why: We can not free a page that is not unpinned. To free a page its pin count should be zero but in our case the pincount is two. So the test will fail.

Test: In this test we are trying a unpin a page which is not in the buffer pool.
Behavior: FAIL
Why: We can only unpin a page that is pinned to the buffer pool. We cannot unpin a page that is not found in the buffer pool. So the test will fail.

Test3

Test: In this test we are creating some new pages and pinning them to the buffer. Then we dirty those pages by writing some values on to them. Then we began unpinning some of the pages and leaving the rest pinned.
Behavior: PASS
Why: Test3 should pass as it applies the same functions and internal logic of test1.

Test: Now we are pinning the pages again and verifying that the values are correct by checking them.
Behavior: PASS
Why: Test3 should pass as it applies the same functions and internal logic of test1.

Disk Space Management Tests

Test1

We are allocating some pages and file entries.

We are then allocating a run of pages. We are writing a string value on some of these pages. We are making sure we are assigning enough space so that we can fit in the string.

We are then de allocating the remaining pages that we did not write on.

Behavior: PASS
Why: We are performing some basic operations that are implemented in DB.java source code.

Test2

We are opening some of the files that are created in test1 and deleting them.

We check if the files that are not deleted are still existing.

We are going to read stuff from the files that are existing and check if the string values are equal to the values inserted in test1.

Behavior: PASS
Why: We are performing some basic operations that that are implemented in DB.java source code.

Test3

We are trying to lookup a deleted file entry. This returns us a NULL page id.
We are trying to delete a deleted file entry. This throws a `FileEntryNotFoundException` exception because the file was already deleted.

We are trying to delete a nonexistent file entry. This again throws a `FileEntryNotFoundException` exception because the file does not exist.

We are trying to lookup a nonexistent file entry with the name "blargle". This returns us a NULL page id because the file does not exist in our db.

We are trying to add a file entry thats already there. This throws a `DuplicateEntryException` exception because the file already exists.

We are trying to add a file entry whose name is longer than the maximum defined size for the name. This throws a `FileNameTooLongException` exception because the name exceeds the MAX_NAME size which is 50.

We are trying to allocate run of pages that are too long. We are assigning 9000 pages which gives us a `OutOfSpaceException` exception.

We are trying to allocate negative run of pages. Which gives us a `InvalidRunSizeException` exception.

We are trying to de allocate negative run of pages. Which gives us a `InvalidRunSizeException` exception.

Behavior: FAIL
Why: We are performing invalid operations to check if the program is failing as expected.

Test4

Test: When starting this test we are making sure that none of the pages are pinned. We are making this check by comparing the values of unpinned frames with the total number of frames.
Behavior: PASS
Why: Since no pages are used or scanned the number of pinned pages remain zero.

Test: We are allocating pages after accounting for the data base overhead. Database overhead is the memory that database uses to store meta data. So we are assigning the remaining memory to pages. So that no more additional memory is left for new pages.
Behavior: PASS
Why: Since we are accounting for the database overhead all the remaining memory can be used for allocation of pages.

Test: We are allocating one more page after the previous step.
Behavior: FAIL
Why: In the previous step we made sure that we are leaving no more space for extra pages. So when we are assigning one more page we get `OutOfSpaceException` exception.

Test: We are de allocating some of the pages namely pages 3-9 and 33-40.
Behavior: PASS
Why: We are performing valid operations using functions defined in DB.java source file.

Test: We are allocating some of the pages that got de allocated in the previous step namely the pages 33-40.
Behavior: PASS
Why: The pages which got freed can be allocated again.

Test: We are de allocating continued run of pages namely pages 11-17 and 18-28.
Behavior: PASS
Why: We are performing valid operations using functions defined in DB.java source file.

Test: We are re allocating the pages that got de allocated in the previous step namely pages 11-28.
Behavior: PASS
Why: We are performing valid operations using functions defined in DB.java source file.

Test: We are deleting some left over file entries. Now a significant number of file entries are added so that the directory surpasses a page.

Behavior: PASS
Why: We are performing valid operations

Test: We are de allocating the last two pages to test the boundary conditions of the space map.
Behavior: PASS
Why: The boundary conditions are valid because we are passing the correct page id by subtracting the database overhead from dbsize to the deallocate_page() function.

Heap File Tests:

Test 1

Test: We are trying to create a new heap file with the name "file_1".
Behaviour: PASS
Why: Heapfile() function from Heapfile.java works correctly

Test: We are checking if our newly created heap is not leaving pages pinned.
Behaviour: PASS
Why: Since we are creating the heap and not scanning anything there must be no pages pinned. So the condition
`SystemDefs.JavabaseBM.getNumUnpinnedBuffers() != SystemDefs.JavabaseBM.getNumBuffers()` which prints the error fails because getNumUnpinnedBuffers() and getNumBuffers() are equal because there is no pinned page.

Test: We are iterating through 100 entries in a for loop and inserting dummy records of fixed length 32 into our new heap.
Behaviour: PASS
Why: Heapfile.java works correctly

Test: We are checking if our newly created dummy records are not leaving pages pinned.
Behaviour: PASS
Why: Since we are just inserting records into the heap and not scanning anything there must be no pages pinned. getNumUnpinnedBuffers() and getNumBuffers() are equal because there is no pinned page.

Test: We are checking if we are able to scan on the heap file
Behaviour: PASS
Why: Scan.java works correctly

Test: We are checking if the scan on heap file is pinning a page.
Behaviour: PASS
Why: Since we started scanning the the heap file its page should be pinned to the buffer. getNumUnpinnedBuffers() and getNumBuffers() are not equal because there is a pinned page.

Test: We are checking if the length and values of the scanned dummy records are equal to the values inserted in a sequential order.
Behaviour: PASS
Why: If all the tests are passing up until this point then the inserted values and the scanned values must be equal.

Test: We are checking if after completing the scan the heap file has upinned its page from the buffer.
Behaviour: PASS
Why: Since scanning the the heap file is completed it upins its page. getNumUnpinnedBuffers() and getNumBuffers() are equal because there is no pinned page.

Test: Finally we are checking if the number of items we scanned and the items that we have inserted are equal.
Behaviour: PASS
Why: If all the tests are passing up until this point then the number of records inserted is equal to the number of records scanned.

Test 2

Test: We are opening the same heap file as in test 1
Behaviour: PASS
Why: We are passing the same file name "file_1" from test1 in which we stored our initial heap to Heapfile(). Heapfile() function from Heapfile.java is working correctly.

Test: We are first scanning the records then deleting half the records from file1. We are deleting the records whose index is odd. We are also making sure that our deletion is not leaving any pages pinned after the operation is completed by closing the scan
Behaviour: PASS
Why: deleteRecord() function from Heapfile.java is working correctly. Since we are making sure that scan is closed by using scan.closescan() there must be no pages pinned. So the condition `SystemDefs.JavabaseBM.getNumUnpinnedBuffers() != SystemDefs.JavabaseBM.getNumBuffers()` which prints the error fails because getNumUnpinnedBuffers() and getNumBuffers() are equal because there is no pinned page.

Test: We are now scanning the remaining records whose index is even and making sure that they have the correct values in them. We are skipping the odd records by adding 2 to the counter variable

Behaviour: PASS
Why: If all the tests are passing up until this point then the data that is entered will be the data that is going to be read.

Test 3

Test: Opening the same heap file that was used in test 1 and test2 by passing "file_1" as an argument to the Heapfile() function.
Behaviour: PASS
Why: Heapfile() function defined in Heapfile.java source code is working correctly.

Test: We are replacing the fval of every even record by scanning the record and then updating the record with a new tuple initialized with the new values. We are using the updateRecord() method from Heapfile.java. Remember that we are not updating the odd records because they were deleted in the previous test. We are also making sure that our update is not leaving any pages pinned after the operation is completed by closing the scan
Behaviour: PASS
Why: updateRecord() function defined in Heapfile.java source code is working correctly. Since we are making sure that scan is closed by using scan.closescan() there must be no pages pinned. So the condition `SystemDefs.JavabaseBM.getNumUnpinnedBuffers() != SystemDefs.JavabaseBM.getNumBuffers()` which causes the test to fail does not execute because getNumUnpinnedBuffers() and getNumBuffers() are equal because there is no pinned page.

Test: We are checking if the values we updated really got updated to the new value. We are using a new method getRecord() from Heapfile.java to read the data. Remember that we are chekcing only for even values as the odd records were deleted in the previous test.
Behaviour: PASS
Why: If all the tests are passing up until this point then the data that is updated will be the data that is going to be read.

Test 4

Test: We are trying to update the record with a tuple whose size is one lesser than the previous record.
Behaviour: FAIL
Why: We cant change size of the record which was already inserted and initialized to a different size.

Test: We are trying to update the record with a record whose size is one greater than the previous record.

Behaviour: FAIL
Why: We cant change size of the record which was already inserted and initialized to a different size.

Test: We are trying to insert a new record whose size is MINIBASE_PAGESIZE + 4 in bytes. We get heap.SpaceNotAvailableException exception.
Behaviour: FAIL
Why: We cant insert records which have size greater than the page size which is 1024 bytes.

Index tests

Test1

Test: We are creating a BTree with records of names. Initially we are opening the heapfile "test1.in" to populate the tuples. We are creating the tuples from data2[] array which contains a list of names. Then we are initializing the BTree making the string attribute as the desired field that gets indexed.
Behaviour: PASS
Why: Heapfile() function defined in Heapfile.java source code and BTreeFile() function defined in BTreeFile.java source code works correctly.

Test: Now we are copying the values from heapfile to the BTree that got initialized. We are using the insert() function defined in BTreeFile.java source code. We are inserting both the string name and rid in the BTreeIndex file.
Behaviour: PASS
Why: Insertion is done by insert() function defined in BTreeFile.java source code which works correctly.

Test: We are checking if the records that got inserted into BTreeIndex file are having the same number of records as the data2[] array from which we are populating the BTreeIndex file. We are also additionally checking if the values of strings are also equal to values that got inserted.
Behaviour: PASS
Why: If tests until this point works correctly the records that get inserted must be equal to the records that gets read.

Test2

Test: We are opening the BTree index file created in test1. We are using the same heap file from test1. So this heap file and BTree is already populated with values form test1
Behaviour: PASS
Why: Heapfile() function defined in Heapfile.java source code and BTreeFile() function defined in BTreeFile.java source code works correctly.

Test: We are making an index scan with the key value of "dsilva" on BTree to check if we can retrieve that value. To perform the index scan we are setting up a expression then passing it on to IndexScan() function defined in IndexScan.java.
Behaviour: PASS
Why: The scan return "dsilva" which is our key value. IndexScan() function defined in IndexScan.java works correctly.

Test: We are making a range scan from Key value of "dsilva" to Key value of "yuc" on BTree to check if we can retrieve those values in order. To perform the range scan we are setting up a expression then passing it on to IndexScan() function defined in IndexScan.java. We are also making sure that these strings are equal to the strings in the array from which the btree is constructed.
Behaviour: PASS
Why: The scan return key values {"dsilva", "dwiyono", "edwards", "evgueni", "feldmann", "flechtne", "frankief", "ginther", "gray", "guangshu", "gunawan", "hai", "handi", "harimin", "haris", "he", "heitzman", "honghu", "huxtable", "ireland", "jhowe", "joon", "josephin", "joyce","jsong", "juei-wen", "karsono", "keeler", "ketola", "kinc", "kurniawa","leela", "lukas", "mak", "marc", "markert", "meltz", "meyers", "mirwais", "muerle", "muthiah", "neuman", "newell", "peter", "raghu", "randal", "rathgebe", "robert", "savoy", "schiesl", "schleis", "scottc", "seo", "shi", "shun-kit", "siddiqui", "soma", "sonthi", "sungk", "susanc", "tak", "thiodore", "ulloa", "vharvey", "waic","wan", "wawrzon", "wenchao", "wlau", "xbao", "xiaoming", "xin", "yi-chun", "yiching", "yuc"} which are our required values. IndexScan() function defined in IndexScan.java works correctly.

Test3

Test: We are making a new heap file "test3.in" and adding tuples with string field, int field and float field. We are creating the BTree index on the integer field.
Behaviour: PASS
Why: Heapfile() function defined in Heapfile.java source code and BTreeFile() function defined in BTreeFile.java source code works correctly.

Test: Now we are copying the values from heapfile to the BTree that got initialized. We are using the insert() function defined in BTreeFile.java source code. We are inserting both the Integer value and rid in the BTreeIndex file.
Behaviour: PASS
Why: Insertion is done by insert() function defined in BTreeFile.java source code which works correctly.

Test: We are making a range scan from Key value of 100 to Key value of 900 on BTree to check if we can retrieve those values in order. To perform the range scan we are setting up a expression then passing it on to IndexScan() function defined in IndexScan.java. We are also making sure that these

values do not cross the lower limit and the upper limit.
Behaviour: PASS
Why: The scan returned values from {100..900} in order which are our required values.IndexScan() function defined in IndexScan.java works correctly.

B Tree Tests

B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.

The B+ tree is a B tree that stores the data only at the leaf nodes of the tree. The leaf nodes of the B+ tree also allows sequential access of data because they are joined by pointers in the form of a linked list.

These B Tree tests are interactive and requires the user to input an option ranging from 0-19.

Test 1: In this test, there is a prompt to the user to enter the number of records (N). On entering this number, a new file is created to store the BTree file. Records `(KEY, RID (pageno, i))` are created in a for loop which run N times.

Test 2: In this test, there is a prompt to the user to enter the number of records (N). On entering this number, a new file is created to store the BTree file. Records `(KEY, RID (pageno, N-i))` are created in a for loop which run N times. The N records are inserted in reverse order.

Test 3: Insert N records at random. The records `(KEY, RID (pageno, k))` are created in the BTree file in for loop which runs N times. Where the value of k is randomly generated.

Test 4: Insert N records then delete M records from the BTree. First M number of `(KEY, RID)` random value pairs are created and are checked if they are present in the BTree file. Delete the record if the b tree consists that pair.

Join and Sort-Merge Joint Tests

Test1

Query: Find the names of sailors who have reserved boat number 1. and print out the date of reservation.

SELECT S.sname, R.date
FROM Sailors S, Reserves R

WHERE S.sid = R.sid AND R.bid = 1

Sailor file and Reserve file are sorted based on respective SID as key.
Then sort-merge happens based on the inner join condition (S.sid=R.sid).
Records are filtered based on BID(=1).
The output consists of sailor names and reservation dates.

Test2

Query: Find the names of sailors who have reserved a red boat and return them in alphabetical order.

SELECT S.sname
FROM Sailors S, Boats B, Reserves R
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
ORDER BY S.sname

The BTree index is created on the Sailors SID key.
Then inner join is performed on Sailors and Reserves files based on join condition (S.sid=R.sid).
Then a nested loop join is performed on the above-retrieved records and the Boat files based on the join condition(R.bid=B.bid).
The resulting records are filtered based on the boat color(=Red).
The filtered records are sorted alphabetically(using ORDER BY). The output consists of sailor names.

Test3

Query: Find the names of sailors who have reserved a boat.

SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid

Sailor file and Reserve file are sorted based on respective SID as key.
Then sort-merge happens based on the inner join condition (S.sid=R.sid).
The output consists of sailor names.

Test4

Query: Find the names of sailors who have reserved a boat and print each name once.

SELECT DISTINCT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid

Sailor file and Reserve file are sorted based on respective SID as key.
Then sort-merge happens based on the implicit inner join condition (S.sid=R.sid).
Then the duplicate records are removed using the DISTINCT keyword.
The output consists of sailor names.

Test5

Query: Find the names of old sailors or sailors with a rating less than 7, who have reserved a boat, (perhaps to increase the amount they have to pay to make a reservation).

```
SELECT S.sname, S.rating, S.age
FROM Sailors S, Reserves R
WHERE S.sid = R.sid and (S.age > 40 || S.rating < 7)
```

Sailor file and Reserve file are sorted based on respective SID as key.
Then sort-merge happens based on the implicit inner join condition (S.sid=R.sid).
Resulting records are filtered based on the age(>40) or ratings(<7). The output consists of sailor names, sailor ratings and sailor age.

Test6

Query: Find the names of sailors with a rating greater than 7 who have reserved a red boat, and print them out in sorted order.

```
SELECT S.sname
FROM Sailors S, Boats B, Reserves R
WHERE S.sid = R.sid AND S.rating > 7 AND R.bid = B.bid AND B.color = 'red'
ORDER BY S.name
```

Inner join is performed on Sailors and Reserves files based on join condition (S.sid=R.sid).
The resulting records are filtered based on sailors' ratings (>7).
Nested loop join is performed on the above-retrieved records and the Boat files based on the join condition(R.bid=B.bid).
The resulting records are filtered based on the boat color(=Red).
The output consists of sailor names sorted alphabetically.

Sort Tests

Test1

Test: We are creating a heap file with the name "test1.in". We are then initializing some tuples with string values from data1[] array . Then we are copying these tuples into the heap. Then we sort the values of the heap in ascending order after performing a file scan on the heap file. The sorting is done on the string attribute. We check if the values are sorted correctly by comparing them to the contents of data2[] array. Note: data1[] array consists of unsorted names whereas data2[] array consists of ascending order sorted names in the data1[] array.
Behaviour: PASS
Why: Sort() function from the Sort.java source file works correctly. Pass int 0 as an additional argument to sort in Ascending order.

Test2

Test: We are creating a heap file with the name "test2.in". We are then initializing some tuples with string values from data1[] array . Then we are copying these tuples into the heap. Then we sort the values of the heap in descending order after performing a file scan on the heap file. The sorting is done using the string attribute. We check if the values are sorted correctly by comparing them to the contents of data2[] array.This time we are going to reverse iterate on the data2[] array to get the values in descending order. Note: data1[] array consists of unsorted names whereas data2[] array consists of ascending order sorted names in the data1[] array.
Behaviour: PASS
Why: Sort() function from the Sort.java source file works correctly. Pass int 1 as an additional argument to sort in descending order.

Test3

Test: We are creating a heap file with the name "test3.in". We are then initializing some tuples with string, int and float attributes. The int attribute is set to a random value. Then we are copying these tuples into the heap. Then we sort the values of the heap in ascending order after performing a file scan on the heap file. The sorting is done on the int attribute. We check if the values are sorted correctly by comparing them to the previous element. We also perform another sort on the FLOAT attribute which was also set to a random value. Now we perform a descending order sort after performing a file scan on the heap file. We check if the values are sorted correctly by comparing them to the previous element.
Behaviour: PASS
Why: Sort() function from the Sort.java source file works correctly.

Test4

Test: We are creating a 2 new heap files with the names "test4-1.in" and "test4-2.in". We are performing test1 and test2 simultaneously on them.

Behaviour: PASS
Why: Test1 and Test2 are passing.

Conclusions

The Tests ensured that all the moving parts of the project got compiled successfully.
All these tests showcased several features of minibase.
I was able to get familiar with the different components of minibase by performing these tests and reading the documentation from the javadoc folder.

Bibliography

“The Minibase Home Page.” Wisc.edu, 2021, research.cs.wisc.edu/coral/minibase/minibase.html. Accessed 28 Jan. 2021.

“Minibase Overview.” Wisc.edu, 2021, research.cs.wisc.edu/coral/minibase/intro/single_user.html. Accessed 29 Jan. 2021.

Dhanushka Madushan. “How Database B-Tree Indexing Works.” Dzone.com, DZone, 22 Nov. 2019, dzone.com/articles/database-btree-indexing-in-sqlite. Accessed 29 Jan. 2021.

Unknown. “DBMS Indexing in DBMS - Javatpoint.” Www.javatpoint.com, 2011, www.javatpoint.com/indexing-in-dbms. Accessed 29 Jan. 2021.

Unknown. “BUFFER MANAGEMENT.” Blogspot.com, Feb. 2021, dbmsfortech.blogspot.com/2016/05/buffer-management.html. Accessed 30 Jan. 2021.

Appendix

typescript file

```
Script started on 2021-01-25 18:27:29-07:00 [TERM="xterm-256color" TTY="/dev/pts/0" COLUMNS="189" LINES="42"]
]0;machine7@machine7-VirtualBox: ~/Desktop/Database_Management_System_Implementation/minjava/javaminibase/src[
cd tests; make bmtest dbtest; whoami; make hftest bttest indextest jointest sorttest sortmerge
make[1]: Entering directory '/home/machine7/Desktop/Database_Management_System_Implementation/minjava/javaminibas
/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java BMTest.java
/usr/lib/jvm/default-java/bin/java  -classpath .:.. 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.

/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java DBTest.java
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.DBTest

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 3 tests for some error conditions:
  - Look up a deleted file entry
**** Looking up a deleted file entry
  --> Failed as expected

  - Try to delete a deleted entry again
**** Delete a deleted file entry again
  --> Failed as expected

  - Try to delete a nonexistent file entry
**** Deleting a nonexistent file entry
  --> Failed as expected

  - Look up a nonexistent file entry
**** Looking up a nonexistent file entry
  --> Failed as expected

  - Try to add a file entry that's already there
**** Adding a duplicate file entry
  --> Failed as expected

  - Try to add a file entry whose name is too long
**** Adding a file entry with too long a name
  --> Failed as expected

  - Try to allocate a run of pages that's too long
**** Allocating a run that's too long
  --> Failed as expected

  - Try to allocate a negative run of pages
**** Allocating a negative run
  --> Failed as expected

  - Try to deallocate a negative run of pages
**** Deallocating a negative run
  --> Failed as expected

  Test 3 completed successfully.

  Test 4 tests some boundary conditions.
```

```
    (These tests are very implementation-specific.)
  - Make sure no pages are pinned
  - Allocate all pages remaining after DB overhead is accounted for
  - Attempt to allocate one more page
**** Allocating one additional page
  --> Failed as expected

  - Free some of the allocated pages
  - Allocate some of the just-freed pages
  - Free two continued run of the allocated pages
  - Allocate back number of pages equal to the just freed pages

  - Add enough file entries that the directory must surpass a page
  - Make sure that the directory has taken up an extra page: try to
    allocate more pages than should be available
**** Allocating more pages than are now available
   --> Failed as expected

  - At this point, all pages should be claimed.  Try to allocateone more.
**** Allocating one more page than there is
   --> Failed as expected

  - Free the last two pages: this tests a boundary condition in the space map.
  Test 4 completed successfully.

...Disk Space Management tests completely successfully.

make[1]: Leaving directory '/home/machine7/Desktop/Database_Management_System_Implementation/minjava/javaminibase
machine7
make[1]: Entering directory '/home/machine7/Desktop/Database_Management_System_Implementation/minjava/javaminibas
/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java HFTest.java
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.HFTest

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.


...Heap File tests completely successfully.

/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java BTTest.java
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.BTTest
Replacer: Clock


Running  tests....

 ********** The file name is: AAA0  **********
------------------------- MENU -----------------
```

```
[0]   Naive delete (new file)
[1]   Full delete(Default) (new file)

[2]   Print the B+ Tree Structure
[3]   Print All Leaf Pages
[4]   Choose a Page to Print

          ---Integer Key (for choices [6]-[14]) ---

[5]   Insert a Record
[6]   Delete a Record
[7]   Test1 (new file): insert n records in order
[8]   Test2 (new file): insert n records in reverse order
[9]   Test3 (new file): insert n records in random order
[10]  Test4 (new file): insert n records in random order
      and delete m records randomly
[11]  Delete some records

[12]  Initialize a Scan
[13]  Scan the next Record
[14]  Delete the just-scanned record

          ---String Key (for choice [15]) ---

[15]  Test5 (new file): insert n records in random order
        and delete m records randomly.

[16]  Close the file
[17]  Open which file (input an integer for the file name):
[18]  Destroy which file (input an integer for the file name):

[19]  Quit!
Hi, make your choice :2
The Tree is Empty!!!
-------------------------- MENU -----------------



[0]   Naive delete (new file)
[1]   Full delete(Default) (new file)

[2]   Print the B+ Tree Structure
[3]   Print All Leaf Pages
[4]   Choose a Page to Print

          ---Integer Key (for choices [6]-[14]) ---

[5]   Insert a Record
[6]   Delete a Record
[7]   Test1 (new file): insert n records in order
[8]   Test2 (new file): insert n records in reverse order
[9]   Test3 (new file): insert n records in random order
```

```
[10]  Test4 (new file): insert n records in random order
      and delete m records randomly
[11]  Delete some records

[12]  Initialize a Scan
[13]  Scan the next Record
[14]  Delete the just-scanned record

           ---String Key (for choice [15]) ---

[15]  Test5 (new file): insert n records in random order
        and delete m records randomly.

[16]  Close the file
[17]  Open which file (input an integer for the file name):
[18]  Destroy which file (input an integer for the file name):

[19]  Quit!
Hi, make your choice :7
Please input the number of keys to insert:
5
 ********** The file name is: AAA1  **********
------------------------- MENU ------------------


[0]   Naive delete (new file)
[1]   Full delete(Default) (new file)

[2]   Print the B+ Tree Structure
[3]   Print All Leaf Pages
[4]   Choose a Page to Print

           ---Integer Key (for choices [6]-[14]) ---

[5]   Insert a Record
[6]   Delete a Record
[7]   Test1 (new file): insert n records in order
[8]   Test2 (new file): insert n records in reverse order
[9]   Test3 (new file): insert n records in random order
[10]  Test4 (new file): insert n records in random order
      and delete m records randomly
[11]  Delete some records

[12]  Initialize a Scan
[13]  Scan the next Record
[14]  Delete the just-scanned record

           ---String Key (for choice [15]) ---

[15]  Test5 (new file): insert n records in random order
        and delete m records randomly.
```

```
[16]  Close the file
[17]  Open which file (input an integer for the file name):
[18]  Destroy which file (input an integer for the file name):

[19]  Quit!
Hi, make your choice :2



---------------The B+ Tree Structure---------------
1     4
--------------- End ---------------


------------------------- MENU ------------------


[0]   Naive delete (new file)
[1]   Full delete(Default) (new file)

[2]   Print the B+ Tree Structure
[3]   Print All Leaf Pages
[4]   Choose a Page to Print

           ---Integer Key (for choices [6]-[14]) ---

[5]   Insert a Record
[6]   Delete a Record
[7]   Test1 (new file): insert n records in order
[8]   Test2 (new file): insert n records in reverse order
[9]   Test3 (new file): insert n records in random order
[10]  Test4 (new file): insert n records in random order
      and delete m records randomly
[11]  Delete some records

[12]  Initialize a Scan
[13]  Scan the next Record
[14]  Delete the just-scanned record

           ---String Key (for choice [15]) ---

[15]  Test5 (new file): insert n records in random order
        and delete m records randomly.

[16]  Close the file
[17]  Open which file (input an integer for the file name):
[18]  Destroy which file (input an integer for the file name):

[19]  Quit!
Hi, make your choice :3
```

```
---------------The B+ Tree Leaf Pages---------------

**************To Print an Leaf Page ********
Current Page ID: 4
Left Link      : -1
Right Link     : -1
0 (key, [pageNo, slotNo]):   (0,  [ 0 0 ] )
1 (key, [pageNo, slotNo]):   (1,  [ 1 1 ] )
2 (key, [pageNo, slotNo]):   (2,  [ 2 2 ] )
3 (key, [pageNo, slotNo]):   (3,  [ 3 3 ] )
4 (key, [pageNo, slotNo]):   (4,  [ 4 4 ] )
************** END ********



------------- All Leaf Pages Have Been Printed --------


-------------------------- MENU ------------------


[0]   Naive delete (new file)
[1]   Full delete(Default) (new file)

[2]   Print the B+ Tree Structure
[3]   Print All Leaf Pages
[4]   Choose a Page to Print

           ---Integer Key (for choices [6]-[14]) ---

[5]   Insert a Record
[6]   Delete a Record
[7]   Test1 (new file): insert n records in order
[8]   Test2 (new file): insert n records in reverse order
[9]   Test3 (new file): insert n records in random order
[10]  Test4 (new file): insert n records in random order
      and delete m records randomly
[11]  Delete some records

[12]  Initialize a Scan
[13]  Scan the next Record
[14]  Delete the just-scanned record

           ---String Key (for choice [15]) ---

[15]  Test5 (new file): insert n records in random order
        and delete m records randomly.
```

```
[16]  Close the file
[17]  Open which file (input an integer for the file name):
[18]  Destroy which file (input an integer for the file name):

[19]  Quit!
Hi, make your choice :19

... Finished .


/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java IndexTest.java
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.IndexTest

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
/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java JoinTest.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.JoinTest
```

```
Replacer: Clock



Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer  Sciences Department or the
developers...

**********Query1 strating **********
Query: Find the names of sailors who have reserved boat number 1.
       and print out the date of reservation.

  SELECT S.sname, R.date
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid AND R.bid = 1


(Tests FileScan, Projection, and Sort-Merge Join)
[Mike Carey, 05/10/95]
[David Dewitt, 05/11/95]
[Jeff Naughton, 05/12/95]

Query1 completed successfully!
**********Query1 finished!!!**********



**********Query2 strating **********
Query: Find the names of sailors who have reserved a red boat
       and return them in alphabetical order.

  SELECT   S.sname
  FROM     Sailors S, Boats B, Reserves R
  WHERE    S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
  ORDER BY S.sname
Plan used:
 Sort (Pi(sname) (Sigma(B.color='red')  |><|  Pi(sname, bid) (S  |><|  R)))

(Tests File scan, Index scan ,Projection,  index selection,
 sort and simple nested-loop join.)

After Building btree index on sailors.sid.

[David Dewitt]
[Mike Carey]
[Raghu Ramakrishnan]
[Yannis Ioannidis]

Query2 completed successfully!
**********Query2 finished!!!**********
```

**********Query3 strating **********
Query: Find the names of sailors who have reserved a boat.

```
  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid
```

(Tests FileScan, Projection, and SortMerge Join.)

[Mike Carey]
[Mike Carey]
[Mike Carey]
[David Dewitt]
[David Dewitt]
[Jeff Naughton]
[Miron Livny]
[Yannis Ioannidis]
[Raghu Ramakrishnan]
[Raghu Ramakrishnan]

Query3 completed successfully!
**********Query3 finished!!!**********

**********Query4 strating **********
Query: Find the names of sailors who have reserved a boat
and print each name once.

```
  SELECT DISTINCT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid
```

(Tests FileScan, Projection, Sort-Merge Join and Duplication elimination.)

[David Dewitt]
[Jeff Naughton]
[Mike Carey]
[Miron Livny]
[Raghu Ramakrishnan]
[Yannis Ioannidis]

Query4 completed successfully!
**********Query4 finished!!!**********

**********Query5 strating **********

```
Query: Find the names of old sailors or sailors with a rating less
       than 7, who have reserved a boat, (perhaps to increase the
       amount they have to pay to make a reservation).

  SELECT S.sname, S.rating, S.age
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid and (S.age > 40 || S.rating < 7)

(Tests FileScan, Multiple Selection, Projection, and Sort-Merge Join.)

[Mike Carey, 9, 40.3]
[Mike Carey, 9, 40.3]
[Mike Carey, 9, 40.3]
[David Dewitt, 10, 47.2]
[David Dewitt, 10, 47.2]
[Jeff Naughton, 5, 35.0]
[Yannis Ioannidis, 8, 40.2]

Query5 completed successfully!
******************Query5 finished!!!*****************



**********************Query6 strating *********************
Query: Find the names of sailors with a rating greater than 7
  who have reserved a red boat, and print them out in sorted order.

  SELECT   S.sname
  FROM     Sailors S, Boats B, Reserves R
  WHERE    S.sid = R.sid AND S.rating > 7 AND R.bid = B.bid
           AND B.color = 'red'
  ORDER BY S.name

Plan used:
 Sort(Pi(sname) (Sigma(B.color='red')  |><|  Pi(sname, bid) (Sigma(S.rating > 7)  |><|  R)))

(Tests FileScan, Multiple Selection, Projection,sort and nested-loop join.)

After nested loop join S.sid|><|R.sid.
After nested loop join R.bid|><|B.bid AND B.color=red.
After sorting the output tuples.
[David Dewitt]
[Mike Carey]
[Raghu Ramakrishnan]
[Yannis Ioannidis]

Query6 completed successfully!
******************Query6 finished!!!*****************
```

```
Finished joins testing
join tests completed successfully
/usr/lib/jvm/default-java/bin/javac -classpath .:.. TestDriver.java SortTest.java
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.SortTest

Running Sort tests....

Replacer: Clock

----------------------- TEST 1 -------------------------
Test1 -- Sorting OK
------------------- TEST 1 completed --------------------

----------------------- TEST 2 -------------------------
Test2 -- Sorting OK
------------------- TEST 2 completed --------------------

----------------------- TEST 3 -------------------------
 -- Sorting in ascending order on the int field --
Test3 -- Sorting of int field OK

 -- Sorting in descending order on the float field --
Test3 -- Sorting of float field OK

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

----------------------- TEST 4 -------------------------
Test4 -- Sorting OK
------------------- TEST 4 completed --------------------


...Sort tests
completely successfully
.


Sorting tests completed successfully
/usr/lib/jvm/default-java/bin/javac -classpath .:.. SM_JoinTest.java TestDriver.java
Note: SM_JoinTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
/usr/lib/jvm/default-java/bin/java  -classpath .:.. tests.SM_JoinTest
Replacer: Clock



Any resemblance of persons in this database to people living or dead
is purely coincidental. The contents of this database do not reflect
the views of the University, the Computer  Sciences Department or the
developers...

**********Query1 strating **********
```

```
Query: Find the names of sailors who have reserved boat number 1.
       and print out the date of reservation.

  SELECT S.sname, R.date
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid AND R.bid = 1


(Tests FileScan, Projection, and Sort-Merge Join)
[Mike Carey, 05/10/95]
[David Dewitt, 05/11/95]
[Jeff Naughton, 05/12/95]

Query1 completed successfully!
******************Query1 finished!!!*****************



**********************Query3 strating ********************
Query: Find the names of sailors who have reserved a boat.

  SELECT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid

(Tests FileScan, Projection, and SortMerge Join.)

[Mike Carey]
[Mike Carey]
[Mike Carey]
[David Dewitt]
[David Dewitt]
[Jeff Naughton]
[Miron Livny]
[Yannis Ioannidis]
[Raghu Ramakrishnan]
[Raghu Ramakrishnan]

Query3 completed successfully!
******************Query3 finished!!!*****************



**********************Query4 strating ********************
Query: Find the names of sailors who have reserved a boat
       and print each name once.

  SELECT DISTINCT S.sname
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid
```

(Tests FileScan, Projection, Sort-Merge Join and Duplication elimination.)

[David Dewitt]
[Jeff Naughton]
[Mike Carey]
[Miron Livny]
[Raghu Ramakrishnan]
[Yannis Ioannidis]

Query4 completed successfully!
******************Query4 finished!!!*****************

**********************Query5 strating *********************
Query: Find the names of old sailors or sailors with a rating less
       than 7, who have reserved a boat, (perhaps to increase the
       amount they have to pay to make a reservation).

```
  SELECT S.sname, S.rating, S.age
  FROM   Sailors S, Reserves R
  WHERE  S.sid = R.sid and (S.age > 40 || S.rating < 7)
```

(Tests FileScan, Multiple Selection, Projection, and Sort-Merge Join.)

[Mike Carey, 9, 40.3]
[Mike Carey, 9, 40.3]
[Mike Carey, 9, 40.3]
[David Dewitt, 10, 47.2]
[David Dewitt, 10, 47.2]
[Jeff Naughton, 5, 35.0]
[Yannis Ioannidis, 8, 40.2]

Query5 completed successfully!
******************Query5 finished!!!*****************

Finished joins testing
join tests completed successfully
make[1]: Leaving directory '/home/machine7/Desktop/Database_Management_System_Implementation/minjava/javaminibase
]0;machine7@machine7-VirtualBox: ~/Desktop/Database_Management_System_Implementation/minjava/javaminibase/src[

Script done on 2021-01-25 18:32:39-07:00 [COMMAND_EXIT_CODE="0"]