DBMS Design Report

Concurrency and transaction atomicity are two essential parts of any database management system. In this simplified DBMS project, we use ReadWriteLock to handle concurrency errors and tempfile to guarantee transaction atomicity.

Our team members prefer file safety to performance, so we chose serializable as this DB's isolation level and ReadWriteLock to deal with concurrency errors. ReadWriteLock has a great advantage than synchronize, that is it has two separate locks for writing and reading. It allows multiple users to read a certain file, at a time. In other words, multiple threads can read from a shared file without causing concurrency errors. This will improve reading performance. The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently. Since we lock the file when writing, no one else can access the file that guarantee situations like dirty read and phantom read would never happen.

Furthermore, we create two file officialFile named cs542.db and tempFile named tempFile.db so that we can make sure the original file keeps untouched when DB write data to file. Every time when new data has been added or old data has been removed, the updated data will always be pushed to tempFile first. Then we rename the tempfile cs542.db. If machine reboots or system is down when updating data, data in tempfile may loss but data in cs542 will keep untouched since we do not write data to cs542 directly. This design guarantees any unsuccessful transaction rolls back. In addition, tempFile can also be used to measure the length of file in case of oversize. What's more, because of 5MB limitation of the file's size, overwritting the whole data is not a big problem, while if the file can be much larger, say 10TB, then overwritting will definitely be a nightmare.

Besides this two file options, we have other choices to update database: record all start points and end points for each key-value pair. When add new data, check whether the file has enough room for appending new data. If space is sufficient, append the new data. If not, check whether combination of all available room is enough for new data or not. If yes, push all available interspace down to generate a new bigger room for new data. If not, denied the put request. When removing data, we have three ways to achieve that. The first solution is pushing all following data up so that there is no fragmentation between key-value pairs. Another solution is to drag the last key-value pair to fill in the available room if it fits. The third choice is to mark the being deleted data instead of erasing it. When we need to put new data, find those marked data and replace it by new data. All these solution is just one file solution with O(n) time complexity. These solutions use less space than two file option but they have a big disadvantage than two file option. That is writing data directly to CS542 file. If machine reboots or some inappropriate requests are being called, data will be dirty and it's hard to roll back especially portion of new data is already being written into file.

Thus, considering safety of file, transaction atomicity and size of the file, two file option is much better than one file option.

Before we can call any methods, we must initialize HW1 class first. Then we create a HashMap to store data in cs542 and create a tempFile, a ReadLock and a WriteLock.

Put function:

Lock the HW1 object by writeLock then put the new key- value pair into HashMap. Here is the rule for putting new data that if there is any key in hashtable same as the adding key, previous value will be replaced by the new value. After putting new key-value pair into hashtable, we copy the whole hashMap to tempFile and judge tempFile by length. If the tempfile meets the length requirements, we rename tempFile as cs542.db. If not, return "The file does not have enough room for data". Finally release the lock.

Get function:

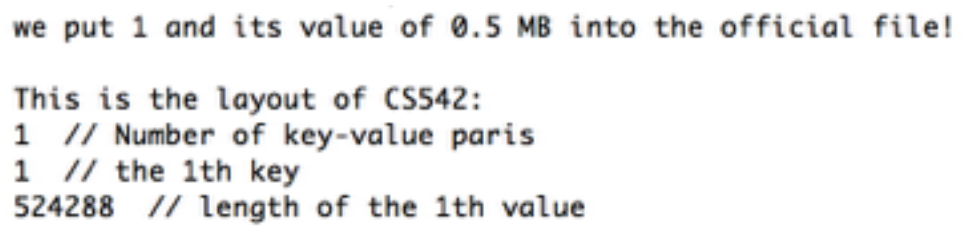
Lock the HW1 object by readLock then retrieve the data for the given key. If we can find that key in the hashtable, then return the length of data for that key. If not, return "We did not find the key". After that, release the lock.

Remove function:

Lock the HW1 object by writeLock then remove the data for the given key. If we can find that key in the hashtable, return "Successes". If not, return "did not find". After that, write new hashMap to tempFile and rename tempFile cs542 meanwhile update the length of 542.db. Then release the lock.

Test result:

We first put one data pair with key = 1 and length = 0.5M into file as the original officialFile cs542.db and do tests.

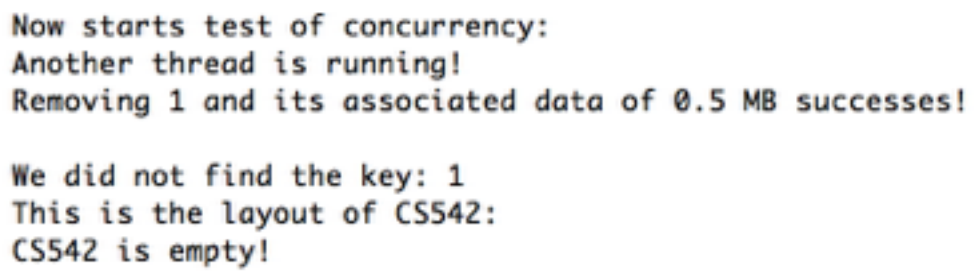


```
we put 1 and its value of 0.5 MB into the official file!

This is the layout of CS542:
1  // Number of key-value paris
1  // the 1th key
524288  // length of the 1th value
```

Concurrency test:

We do the following test: when one caller does a Remove() and another caller does a Get() with the same key a millisecond later. In order to achieve that, we create a new thread named t to run get method and make main thread to run the remove function. Result shows as following: We run this concurrency test more than ten times. For each time, we always got the same result: we cannot find the key. If concurrency errors happen, then the result may be different because we can’t predict which thread is running. In that case, we got a big chance to find the key. Since this situation did not happen, we conclude that we pass the concurrency test.

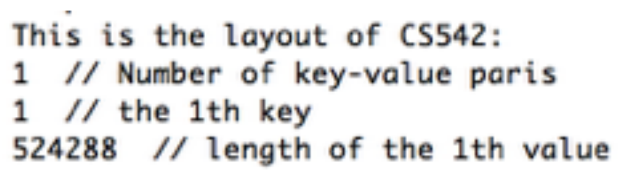


```
Now starts test of concurrency:
Another thread is running!
Removing 1 and its associated data of 0.5 MB successes!

We did not find the key: 1
This is the layout of CS542:
CS542 is empty!
```

Durability test:

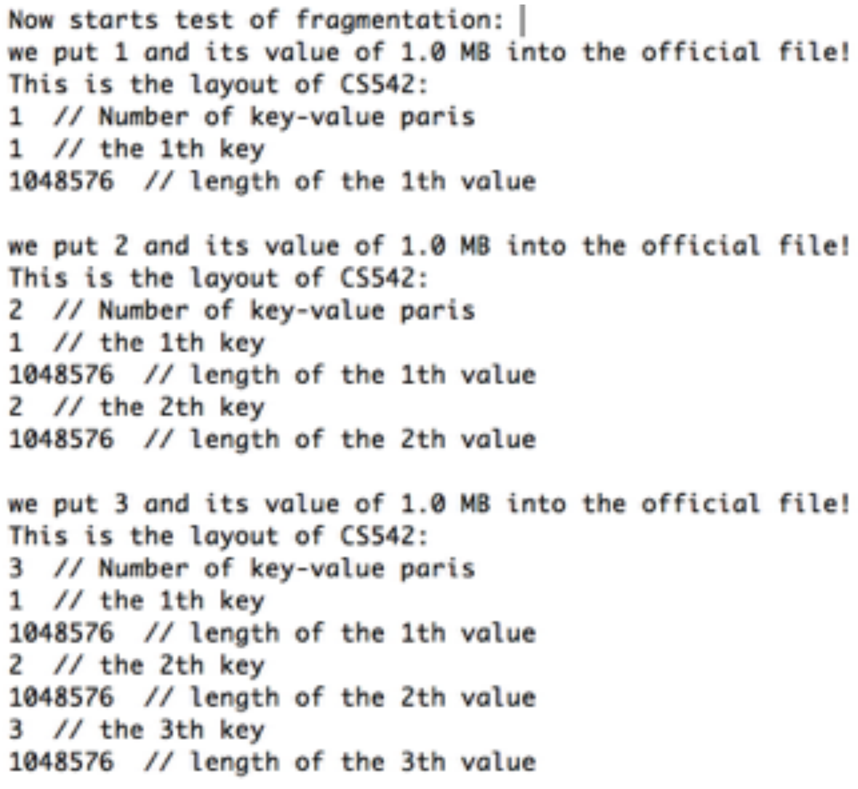
When we put new key-value pair into CS542, we rebooted the machine and found the file before rebooting and file after rebooting were the same.



```
This is the layout of CS542:
1  // Number of key-value paris
1  // the 1th key
524288  // length of the 1th value
```

Fragmentation test:

We do the test: Put() 4 values, byte arrays of 1 MB each, with keys A, B, C and D. Remove key B.Put() ½ MB in size for key E. Validate that a Put() 1 MB in size for key F fails. Remove C and now validate that a Put() 1 MB in size for key G succeeds. Remove E and try Put() 1 MB in size for key H.
The resulting is showing as following, we do every process successfully.



```
Now starts test of fragmentation:
we put 1 and its value of 1.0 MB into the official file!
This is the layout of CS542:
1  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value

we put 2 and its value of 1.0 MB into the official file!
This is the layout of CS542:
2  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
2  // the 2th key
1048576  // length of the 2th value

we put 3 and its value of 1.0 MB into the official file!
This is the layout of CS542:
3  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
2  // the 2th key
1048576  // length of the 2th value
3  // the 3th key
1048576  // length of the 3th value
```

```
we put 4 and its value of 1.0 MB into the official file!
This is the layout of CS542:
4  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
2  // the 2th key
1048576  // length of the 2th value
3  // the 3th key
1048576  // length of the 3th value
4  // the 4th key
1048576  // length of the 4th value

Removing 2 and its associated data of 1.0 MB successes!
This is the layout of CS542:
3  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
3  // the 2th key
1048576  // length of the 2th value
4  // the 3th key
1048576  // length of the 3th value

we put 5 and its value of 0.5 MB into the official file!
This is the layout of CS542:
4  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
3  // the 2th key
1048576  // length of the 2th value
4  // the 3th key
1048576  // length of the 3th value
5  // the 4th key
524288  // length of the 4th value

The file does not have enough room for data of which key is: 6
This is the layout of CS542:
4  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
3  // the 2th key
1048576  // length of the 2th value
4  // the 3th key
1048576  // length of the 3th value
5  // the 4th key
524288  // length of the 4th value

Removing 3 and its associated data of 1.0 MB successes!
This is the layout of CS542:
3  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
4  // the 2th key
1048576  // length of the 2th value
5  // the 3th key
524288  // length of the 3th value
```

```
we put 7 and its value of 1.0 MB into the official file!
This is the layout of CS542:
4  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
4  // the 2th key
1048576  // length of the 2th value
5  // the 3th key
524288  // length of the 3th value
7  // the 4th key
1048576  // length of the 4th value

Removing 5 and its associated data of 0.5 MB successes!
This is the layout of CS542:
3  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
4  // the 2th key
1048576  // length of the 2th value
7  // the 3th key
1048576  // length of the 3th value

we put 8 and its value of 1.0 MB into the official file!
This is the layout of CS542:
4  // Number of key-value paris
1  // the 1th key
1048576  // length of the 1th value
4  // the 2th key
1048576  // length of the 2th value
7  // the 3th key
1048576  // length of the 3th value
8  // the 4th key
1048576  // length of the 4th value
```