Introduction to Database System - Assignment 4 Report

Team 22 - 108020017 黃珮綺, 108020021 賴柏翰

Phase 2 Report

1. Implementation Differences

> 💡 The highlighted files are the files we have modified.

- `org.vanilladb.core.storage.buffer`
  - Buffer
  - BufferPoolMgr
  - BufferMgr
- `org.vanilladb.core.storage.file`
  - BlockId
  - Page
  - FileMgr

  `org.vanilladb.core.storage.file.io.javanio`
  - JavaNioFileChannel

  `org.vanilladb.core.storage.file.io.jaydio`
  - JaydioDirectIoChannel
- `org.vanilladb.core.sql.storedprocedure`
  - StoredProcedureHelper (deleted)

2. Implementation Details

Buffer

In the solution, the integer variable `pins` is replaced with an atomic version, and an atomic Boolean variable `isRecentlyPinned` is added. Since the classes of `AtomicInteger` and `AtomicBoolean` provide atomic operations that can be executed safely among multiple threads, this change can ensure the thread safety of the class `Buffer`. The functions *flush()*, *pin()*, *unpin()*, *isPinned()*, and *assigneToBlock()* are modified as a result to fit the new variables.

The set `modifiedBy` is replaced with a single Boolean variable `isModified` since we only need to know whether the buffer is modified without recording who has done the changes. This way simplifies the code and reduces the amount of memory used by the class.

Finally, a content read-write lock and a swap lock are added to improve the concurrency. The synchronized blocks in the *getVal()*, *setVal()*, *lastLsn()*, *close()*, *block()*, and *getUnderlyingPage()* functions are removed and replaced with appropriate locks. With these locks, we can prevent the synchronization overhead.

BufferMgr

The coverages of the synchronized blocks in this file are reduced to improve performance.

For functions *available()*, *flushAllMyBuffers()*, and *flushAll()*, the synchronized blocks are simply removed, and for the functions *unpin()* and *unpinAlln()*, the synchronized blocks are preserved only for *bufferPool.notifyAll()*. Both our implementation and the solution have done this part.

Moreover, the solution has further modifications. For the functions *pinNew()* and *pin()*, the synchronized section is simply deleted. For the function *repin()*, the synchronized section is preserved for *bufferPool.wait(MAX_TIME)*. Therefore, the time for threads to spend waiting for locks is reduced, and the performance is improved accordingly.

Besides, an atomic Boolean variable `hasWaitingTx` was added to the class to guarantee thread safety. This variable is used to indicate whether there is a transaction waiting for an available buffer. The `bufferPool` can skip notifying threads in *unpin()* if this value is false. The local Boolean variable `waitOnce` which indicates whether the transaction has once waited for a buffer in *pinNew()* and *pin()* has the same effect.

BufferPoolMgr

Firstly, all usages of the `Logger` object have been removed from the class, which simplifies the code and improves performance.

A volatile integer `lastReplacedBuff` and an atomic integer `numAvailable` are added to the class and are used to track the last replaced buffer and the number of available buffers, respectively. The functions, such as *available(),* are modified to use the atomic variable as well.

The lock striping strategy, which locks on specific items instead of the entire instance of the class, is applied here to prevent lock contention among threads. The variables `stripSize` (=1009), `fileLocks` and `blockLocks` are introduced, and the methods *prepareFileLock()* and *prepareBlockLock()* were added to create locks for files and blocks, respectively. Then, the functions *pin()* and *pinNew()* use these locks to lock exactly on the block and the file they are accessing. With the help of these locks, the function *findExistingBuffer()* was rewritten that it directly returns the buffer with the given `blockId`.

Last but not least, when accessing blocks, the swap locks of the blocks are used to ensure thread safety.

In this file, we only removed the synchronized block of available().

BlockId

We use a fixed value `hashCode` which stores the hash code of the block to prevent re-calculation in each request to *hashCode()*. In the solution, the variable `myHashCode` has the same effect as in our implementation.

Nonetheless, the return value of the method *toString()* was also fixed by a string `myString` in the solution. This also avoids repeatedly computing the string value.

Page

We removed the synchronized block in the *read()*, *write()*, and *append()* methods since we assumed that there is no need of ensuring thread safety in `Page` for it only holds the contents of a single block. In addition, the size of synchronization blocks in the methods *getVal()* and *setVal()* are reduced and they only protect the sections that access the contents in a block.

Yet, we don't see any modification in this file in the solution.

FileMgr

In the solution, the class uses a concurrent hash map `fileNotEmptyCache` that caches nonempty files. The method *isFileEmpty()* can return whether the file is empty by simply checking the hash map without fetching the file.

An array named `anchors` of length 1009 is also added to manage the file access. The `FileMgr` will lock the objects in `anchors` whose corresponding file with the given file name is being modified, instead of locking the whole instance of `FileMgr`. The implementation can be seen in the methods *delete()* and *getFileChannel()*.

Furthermore, the synchronized blocks in the functions *read()*, *size()*, *isNew()*, *write()*, and *append()* have been removed. These operations should be ensured thread safety at the `IoBuffer` and `IoChannel` levels, which will be described later. Here, since we haven't handled the `IoBuffer` and `IoChannel` parts in our implementation, we only deleted the synchronized blocks in *read()*, *size()*, and *isNew()* considering that they won't be affected by the contention of locks among threads.

JavaNioFileChannel

In the solution, the class uses a `ReentrantReadWriteLock` to manage file access in *read()*, *write()*, *append()*, *size()*, and *close()* methods. It separates the locks for read and write operations and will provide thread-safe access to files.

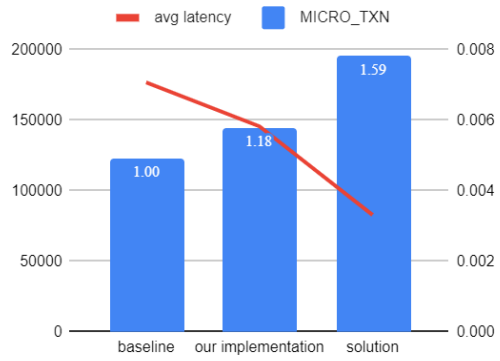
The variable `fileSize` is used to store the exact size of the table. It will be updated whenever the table is being *read()*, *write()*, or *append()*, and then returned in *size()* with a read lock. This prevents repeated computation when the file size remains unchanged.

JaydioDirectIoChannel

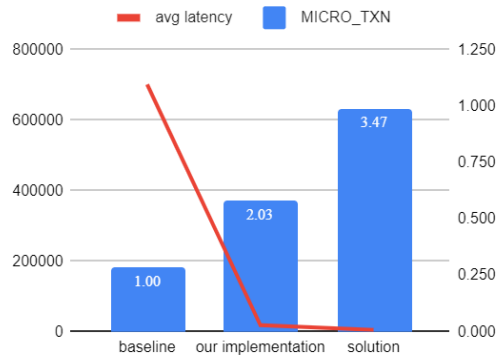
This class is modified in a similar way as done in `JavaNioFileChannel`. `ReentrantReadWriteLock` is used to manage file access in *read()*, *write()*, *append()*, *size()*, and *close()*, and thus avoids potential concurrency issues that could arise with shared file access. The variable `fileSize` is also introduced so that the method *size()* correctly reflects the size of the file being accessed.

3. Performance Comparisons

It is obvious that the solution has done more optimization for each module, but it's hard to evaluate exactly how much the difference in improvement between our implementation and the solution is. Thus, we perform experiments on the throughputs just following the ones in the phase 1 report. First, we'll use the default value, and then the optimized parameters. The results are shown below.

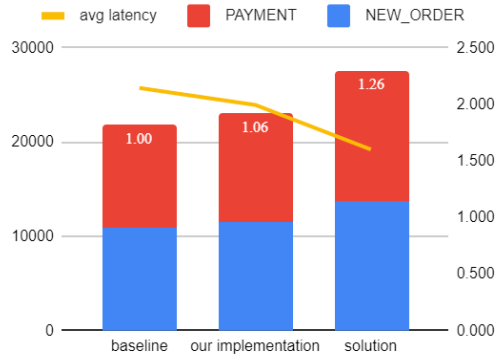


Improvement with default values

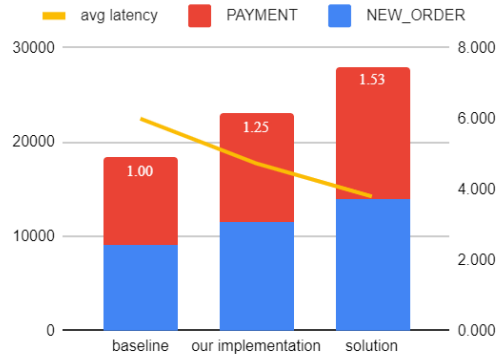


Improvement with optimized parameters

> The basic improvement rates of our implementation and the solution are 18% and 59%, respectively. With the optimized parameters, our implementation twice the throughputs as the baseline. Furthermore, the solution increases the throughput to three and a half times the original throughput.



Improvement with default values (NUM_RTE=1)



Improvement with NUM_RTES = 2

> In the phase 1 report, we use the default value of `NUM_RTES` (=1) to evaluate the performance. This time, we only detected a 6% improvement in the throughput, instead of 20% as reported in the previous report. Nevertheless, the solution reaches a 26% improvement in the throughput.
>
> Besides this, we noticed that the `NUM_RTES` in the solution is set to 2. Hence, we then tested the performance with two remote terminal executors to check if our improvement was underestimated. The rate of improvement is 25%, which is larger than it was asserted in the phase 1 report. Also, as it was expected, the solution gets a greater improvement, that is, 53% with the modification.

In summary, we can see that the solution outperforms our implementation with more thread safety and a higher rate of improvement.

4. Reference

Source Code

Appendix