Introduction to Database System - Assignment 4 Report

Team 22 - 108020017 黃珮綺, 108020021 賴柏翰

Phase 1 Report

1. Implementation Details

`org.vanilladb.core.storage.buffer`

BufferPoolMgr

The `BufferPoolMgr` handles the access to buffers and implements the block replacement strategy. It must ensure the thread safety within buffers to make sure the data accessed by multiple clients is correct. Therefore, we only remove one synchronized label of `available()` method because it doesn't modify blocks.

BufferMgr

The instance of `BufferMgr` manages the access to buffers for each single transaction. It will request `BufferPoolMgr`, who ensures the thread safety for blocks, to pin or unpin. Hence, we removed all the synchronized blocks in this file. Nonetheless, in `unpin()` and `unpinAll()` methods, we preserved the synchronized block for `bufferPool.notifyAll()` to make sure there is no thread which is accessing a block, especially unpinning a block or all blocks, at the same time.

`org.vanilladb.core.storage.file`

BlockId

The `BlockId` identifies a specific logical block by its own hash code. To avoid repeated calculation of hash code, we add a new class attribute `hashCode` to save the hash code. When asking for the hash code of an instance of `BlockId`, we can return the value of `hashCode` directly.

Page

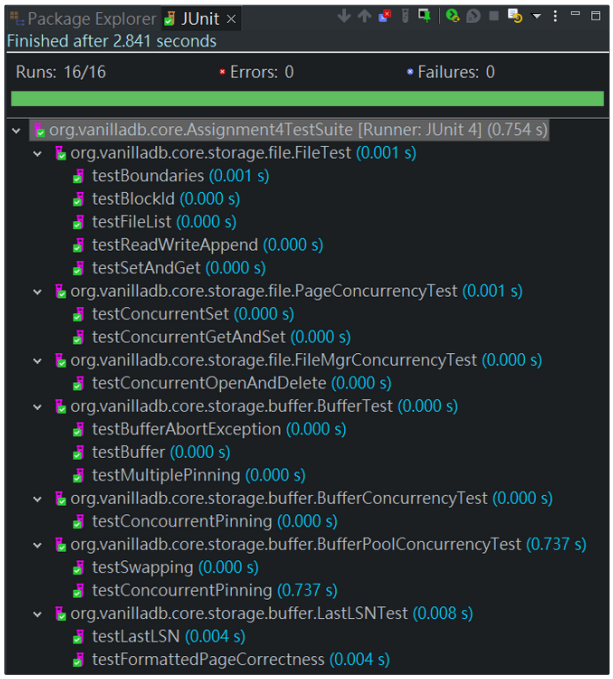
The instance of `Page` holds the contents of a single block, so there is no need of ensuring thread safety. Thus, we removed the synchronized block in `read()`, `write()` and `append()` methods. have optimized the synchronization of the `getVal()` and `setVal()` methods by implementing it only when accessing `contents` (contents in a block),

FileMgr

The `FileMgr` is responsible of file access, it will read, write or append an entire block. Accordingly, it should guarantee the thread safety of files.

We have removed the synchronized blocks of `read()`, `size()`, `delete()` and `isNew()` methods since synchronization is not needed, and leave the synchronized blocks of `write()` and `append()` unchanged.

2. JUnit Test



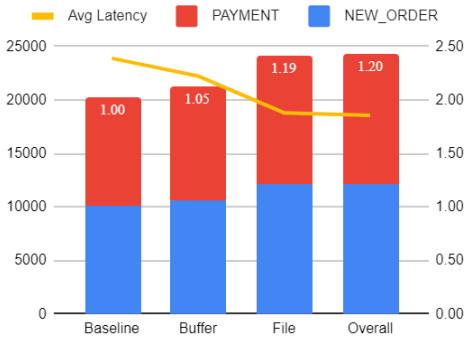
3. Experiments

Environment

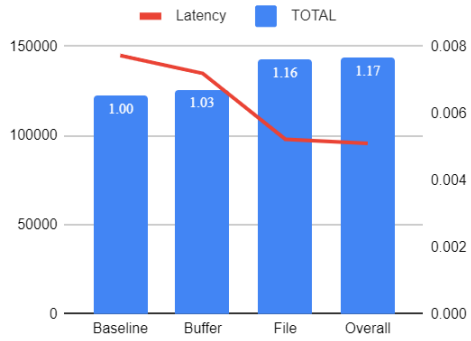
`Intel Core i5-8300H CPU @2.30GHz, Windows 10`

Basic Improvement

First, we use the default parameters to assess the improvement of our implementation. The improvement of each step is shown below, and the default value of the parameters as well as the data are provided in the Appendix.



TPC-C Benchmark Throughputs



Micro-Benchmark Throughputs

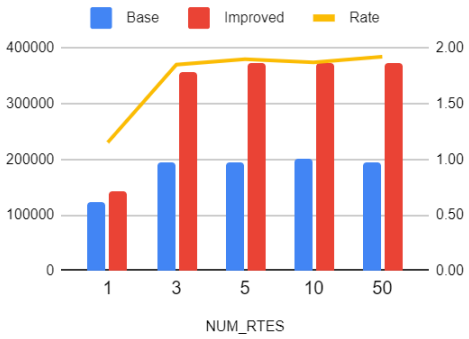
> The default parameters produces 12% improvement on micro-benchmark and 16% improvement on micro-benchmark. The average latencies of two benchmark also decreases.
>
> For each module, the modification on `org.vanilladb.core.storage.file` leads to a significant improvement compared to the modification on `org.vanilladb.core.storage.buffer`.

Adjusting Parameters

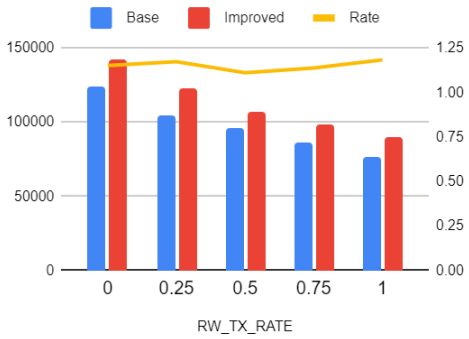
We have underwent some experiments on each parameters one by one to examine the improvement on performance of our implementation. We also eliminated experiments on some specific parameters, such as `TOTAL_READ_COUNT`, since we thought it won't affect the improvement on performance.

The original data are provided in the Appendix, and below are the analyses of the experiments.

- **Micro-Benchmark**

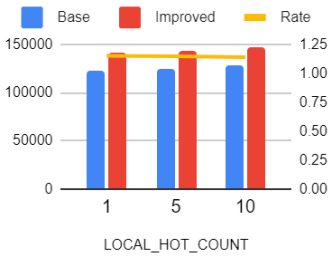


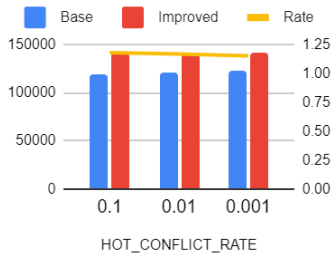
> **Experiment on `NUM_RTES` (default=1)**
>
> When `NUM_RTES` is at default value 1, there is slight difference between the baseline and the improved performance. After `NUM_RTES` is increased to 5 and over 5, the latter one outperforms the former one. However, the total throughput stops growing significantly as `NUM_RTES` is increased from 5 to 50. This is probably because of that the number of threads is limited between 1 to 5. Hence, the concurrency methods fairly improve the performance at `NUM_RTES` larger than 5.

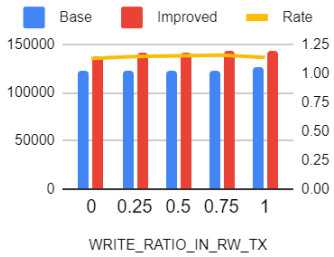


> **Experiment on `RW_TX_RATE` (default=0)**
>
> As `RW_RX_RATE` increases, both the throughputs of the baseline and the improved one decrease. The reason is that `WRITE` transactions require more steps to access data, which is already reasoned in HW2.
>
> The ratio of improvement done by modifying with respect to the baseline is at the range of 1.11 to 1.18. However, there is no trend can be observed. We won't change this value in the final test, since it has nothing to do with the improvement rate.

> **Experiment on `LOCAL_HOT_COUNT` (default=1)**
>
> Here, we use the default value of `TOTAL_READ_COUNT` (=10) to test `LOCAL_HOT_COUNT`.
>
> As the `LOCL_HOT_COUNT` increases, the throughputs slightly increases. Though, We didn't find any obvious changes between the rates of improvement with different values of `LOCAL_HOT_COUNT`.
>
> **Experiment on `HOT_CONFLICT_RATE` (default=0.001)**
>
> The baseline throughput decreases as the conflict rate increase, while the improved version is not influenced by the conflict rate. This brings a little increase in the rate of improvement compared to the baseline.
>
> **Experiment on `WRITE_RATIO_IN_RW_TX` (default=1)**
>
> The throughputs with different values pf `WRITE_RATIO_IN__RW_TX` are basically the same. Despite of this, when the improving rates with `WRITE_RATIO_IN_RW_TX` being larger than 0 are greater than that with `WRITE_RATIO_IN_RW_TX` being equal to 0.

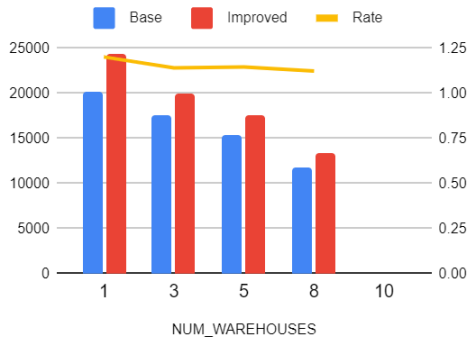






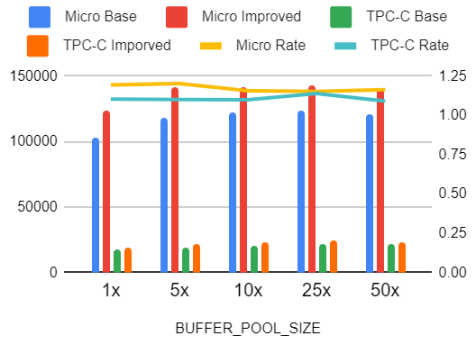
> Since we haven't modified the replacement strategy in this assignment, adjustment on these three parameters will not yield any significant changes in the performance improvement.

- **TPC-C Benchmark**



> **Experiment on `NUM_WAREHOUSE` (default=1)**
>
> Since the conflict rate is higher as `NUM_WAREHOUSE` increases, the throughput will then decrease. Nonetheless, there is no significant difference between the ratios of improvement, which is mainly affected by the replacement strategy in buffer pool.
>
> Besides, we have tested on numbers larger than 8, but we unfortunately met the storage limit during the loading procedure. Thus the throughputs and rates are not tested at `NUM_WAREHOUSE` larger than 8.

- **Configuration**



> **Experiment on `BUFFER_POOL_SIZE` (default=102400)**
>
> We have performed several tests on different `BUFFER_POOL_SIZE`, such as 1, 5, 10, 25 (default), and 50 time(s) of `BLOCK_SIZE` (=4096). When the `BUFFER_POOL_SIZE` is as large as `BLOCK_SIZE`, the throughputs is low, and as `BUFFER_POOL_SIZE` increases, the number of throughput increases. Yet, the improvement with respect to the baseline is fairly the same. Therefore, we consider that the default is already optimized.

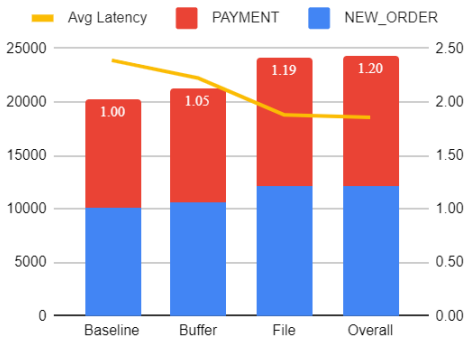
Maximum Improvement

We assume that the improvement on each parameter is additive and select the parameters that maximize the improvement. The parameters we use are listed below.

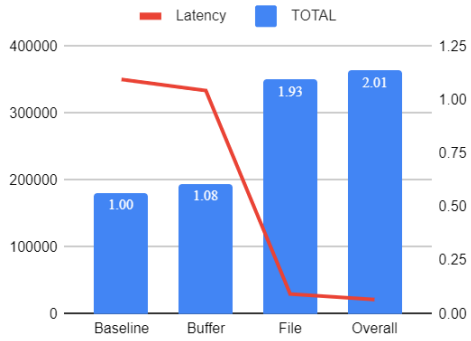
```
# For micro-benchmark
NUM_RTES = 5
READ_WRITE_TX_RATE = 0
LOCAL_HOT_COUNT = 1
HOT_CONFLICT_RATE = 0.1
WRITE_RATIO_IN_RW_TX = 1

# For TPC-C benchmark
NUM_WAREHOUSE = 1

# For VanillaDB configuration
BUFFER_POOL_SIZE = 102400
```



Final Improvement on TPC-C



Final Improvement on Micro-Benchmark

> The new parameters result in 101% overall improvement on micro-benchmark. Also, since we didn't change the parameters for TPC-C benchmark, the overall improvement remains 12%.

4. Reference

Source Code

Appendix