Optimization:

Buffer:

在原本的Buffer.java裡，所有的method皆被synchronized包住，即是不管thread要read還是write，只要有其他thread在read/write皆會被blocked住。理想情況下，除非resource正在被write，否則不應該block住想要read的thread。為了滿足此條件，我們新增了read/write lock，如下圖：

```
private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
private final ReadLock rLock = lock.readLock();
private final WriteLock wLock = lock.writeLock();
```

接著，我們將各method的synchronized拿掉，替換成rLock/wLock，如下圖：

```
public Constant getVal(int offset, Type type) {
    rLock.lock();
    try {
        return contents.getVal(DATA_START_OFFSET + offset, type);
    } finally {
        rLock.unlock();
    }
}

void setVal(int offset, Constant val) {
    wLock.lock();
    try {
        contents.setVal(DATA_START_OFFSET + offset, val);
    } finally {
        wLock.unlock();
    }
}
```

如此一來，只有當有thread在write的時候才會block想要read的thread。

BufferMgr:

在BufferMgr中，所有method的critical section都幾乎包住了整個method，然而，有些地方其實根本不需要synchronized，所以我們重新安排了新的critical section，只有當需要synchronize時我們才將之設為critical section。

FileMgr:

在原本的FileMgr.java裡，read/write function都使用了synchronized，這讓所有的thread在一起使用這個filemgr時，filemgr一次只能讀/寫一個disk block，失去了multi-thread的意義。我們將filemgr的read/write改成首先會去計算要讀/寫的filename的hash，接著開了一個r/w lock的array，把剛剛計算的hash值mod 100後就是我們要拿的

r/w lock number，如此一來，不同的file若是計算出來的值不一樣就會拿到不同的r/w lock，並且可以同時做read/write，而當某個file被lock起來也能確保這個file不會被同時read和write。

BlockId：

在這裡我們讓BlockId的hashCode function在第一次計算後就將值存在這個BlockId的hash_value裡，如此一來，當下次再叫hashCode function時只需要回傳hash_value，不用再重新計算。

Experiment:

experiment environment:

Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz , 12GB RAM , 932GB HDD,Windows 10

experiment result:

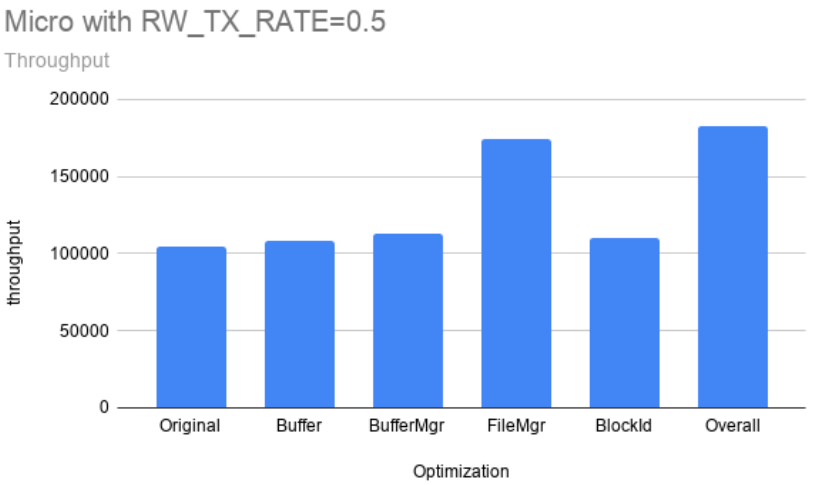
我們使用micro-benchmark對優化作實驗，其中，vanilladb的bufferpool size維持102400，而micro-benchmark我們則分別使用了RW_TX_RATE=0.5和RW_TX_RATE=0.9做實驗。下面的圖表是我們的實驗結果，original代表原本沒優化過的版本，buffer、buffermgr……則是分別優化了某個file的版本，最後的overall是全部優化的版本。

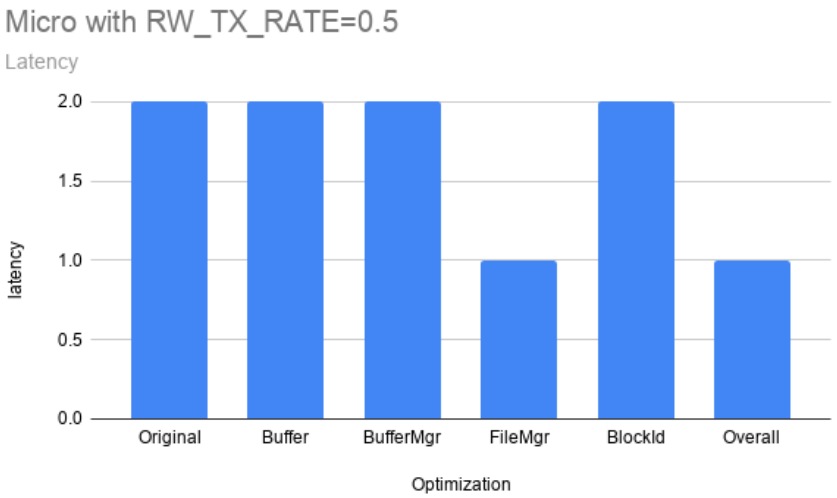
micro-benchmark's parameters：

```
# The number of items in the testing data set
org.vanilladb.bench.benchmarks.micro.MicrobenchConstants.NUM_ITEMS=100000
# The ratio of read-write transactions during benchmarking
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.RW_TX_RATE=0.9
# The ratio of long-read transactions during benchmarking
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.LONG_READ_TX_RATE=0.0
# The number of read records in a transaction
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.TOTAL_READ_COUNT=10
# The number of hot record in the read set of a transaction
```

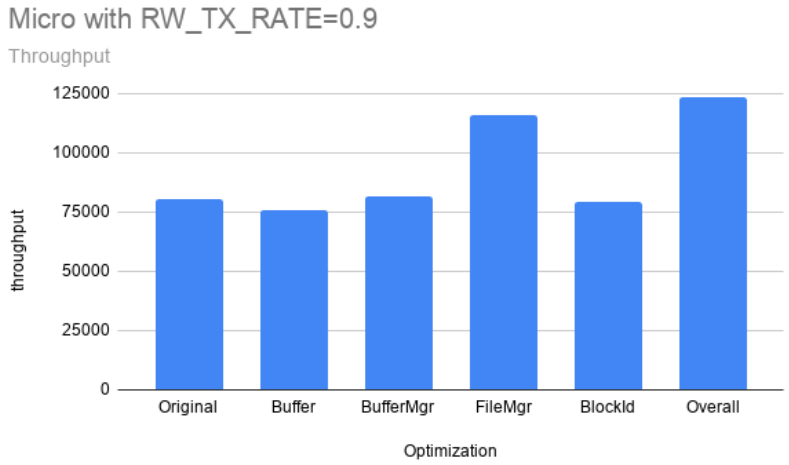
```
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.LOCAL_HOT_COUNT=1
# The ratio of writes to the total reads of a transaction
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.WRITE_RATIO_IN_RW_TX=0.5
# The conflict rate of a hot record
org.vanilladb.bench.benchmarks.micro.rte.MicrobenchmarkParamGen.HOT_CONFLICT_RATE=0.001
```

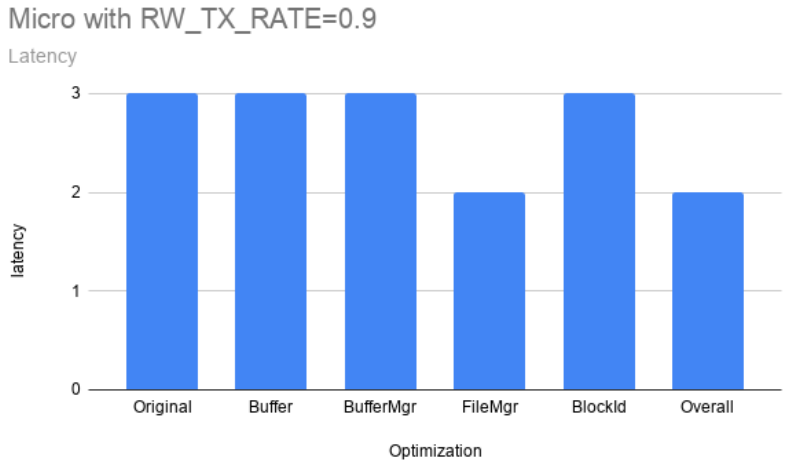
micro-benchmark(rw_rate=0.5):





micro-benchmark(rw_rate=0.9)





從以上的圖表我們發現不管是r/w rate=0.5或0.9，優化幅度最好的都是FileMgr，我們認為這是因為FileMgr只有一個，當所有thread同時透過FileMgr去read/write disk block時，從原本一次只能讀/寫一個block(相當於只有一個thread在做事)，變成可以同時有很多thread在read/write，因此優化幅度很大。而buffer相對於FileMgr，原本就有很多個，不同thread去read/write同一個buffer page的機率不會這麼大，而且buffer page的read/write是在memory中進行，相對於FileMgr是做disk IO，本來就進行得很快，因此把buffer改成r/w lock的優化效果不會這麼明顯。