Cloud Databases Assignment 3

Group 2
102061105 Yun-Sheng, Chang
102061212 Jian-Hao, Huang

April 19, 2017

1 Implementation

1.1 BufferMgr

We know that every transaction has one *BufferMgr* and the entire system has one *BufferPoolMgr*. We leave the *BufferPoolMgr* unchanged and modify the *BufferMgr* according to the following rules:

1. shared resources among *BufferMgr* (e.g., *pinnedBuffers* and *waitingThreads*) should be protected by synchronization mechanisms (in fact, we're not sure whether the protection is required; we know that every transaction has one *BufferMgr* but we don't know if one transaction could be executed by multiple threads, if so, the protection is required; if not, the protection is redundant. Anyway, additional protection will not affect the correctness);
2. since the *BufferPoolMgr* remains unchanged, all the methods in *BufferPoolMgr* is thread-safe and they do not require protection;
3. make sure the *bufferPool.wait()* or *bufferPool.notifyAll()* is invoked in the *synchronized(bufferPool)* block.

1.2 FileMgr

FileMgr is a singleton among the system, and some methods in *FileMgr* (e.g., *read()* and *write()*) should not be defined as *synchronized* method when most of the operations are toward files rather than the entire system. Since one *IoChannel* associates with one file, we remove the prefix *synchronized* from some methods and protect each *IoChannel* in its own *synchronized* block. We show *read()* as an example:

```java
void read(BlockId blk, IoBuffer buffer) {
  try {
    IoChannel fileChannel = getFileChannel(blk.fileName());
    synchronized (fileChannel) {
      buffer.clear();
      fileChannel.read(buffer, blk.number() * BLOCK_SIZE);
    }
  }
  ...
}
```

FileMgr.java

However, the method *getFileChannel* is still a **global** operation which needs to be protected by some synchronization mechanisms. The following code shows the method without concurrency control:

```java
private IoChannel getFileChannel(String fileName) throws IOException {
  IoChannel fileChannel = openFiles.get(fileName);
  if (fileChannel == null) {
    File dbFile = fileName.equals(DEFAULT_LOG_FILE) ? new File(logDirectory, fileName)
        : new File(dbDirectory, fileName);
    fileChannel = IoAllocator.newIoChannel(dbFile);
    openFiles.put(fileName, fileChannel);
  }
  return fileChannel;
}
```

Original FileMgr.java

We illustrate a problematic condition when no concurrency control is done:

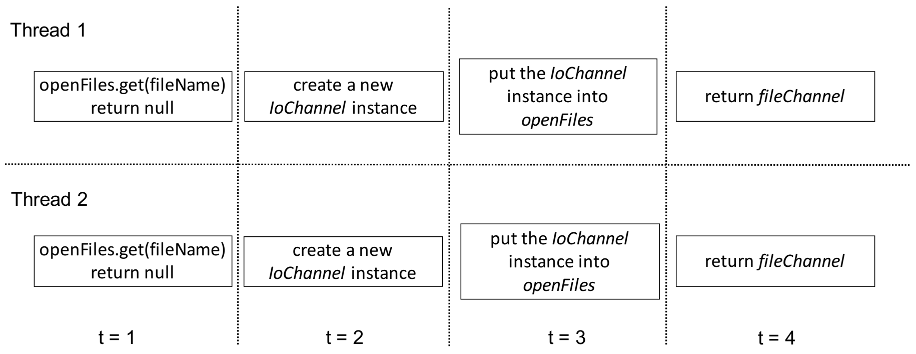
Thread 1
openFiles.get(fileName) return null
create a new IoChannel instance
put the IoChannel instance into openFiles
return fileChannel
Thread 2
openFiles.get(fileName) return null
create a new IoChannel instance
put the IoChannel instance into openFiles
return fileChannel
t = 1
t = 2
t = 3
t = 4

Figure 1: Without concurrency control

Assume that two threads try to enter *getFileChannel()* to get the *IoChannel* instance of a same file at the same time; since they both fail to get the *IoChannel* instance with *openFiles.get()*, they both create an *IoChannel* instance and put them into *openFiles*. As a result, there will be two *IoChannel* instances assosiate with the same file. The naive way to solve the problem is to make the entire method *synchronized*:

```java
private synchronized IoChannel getFileChannel(String fileName) throws IOException{
  ...
}
```

Inefficient Concurrency Control

But it's clearly not an efficient approach as the problem only happens at the very start when the *fileChannel* is not yet created. To guarantee correctness at the beginning and maintain efficiency afterward, we make the critical section only be executed when the *fileChannel* is not yet created. We modify the method to avoid unnecessary synchronization:

```java
private IoChannel getFileChannel(String fileName) throws IOException {
  IoChannel fileChannel = openFiles.get(fileName);
  if (fileChannel == null) {
    synchronized (openFiles) {
      fileChannel = openFiles.get(fileName);
      if (fileChannel != null)
        return fileChannel;
      File dbFile = fileName.equals(DEFAULT_LOG_FILE) ? new File(logDirectory, fileName)
          : new File(dbDirectory, fileName);
      fileChannel = IoAllocator.newIoChannel(dbFile);

      openFiles.put(fileName, fileChannel);
    }
  }
  return fileChannel;
}
```

FileMgr.java

Again, we explain why the code works with the same example:

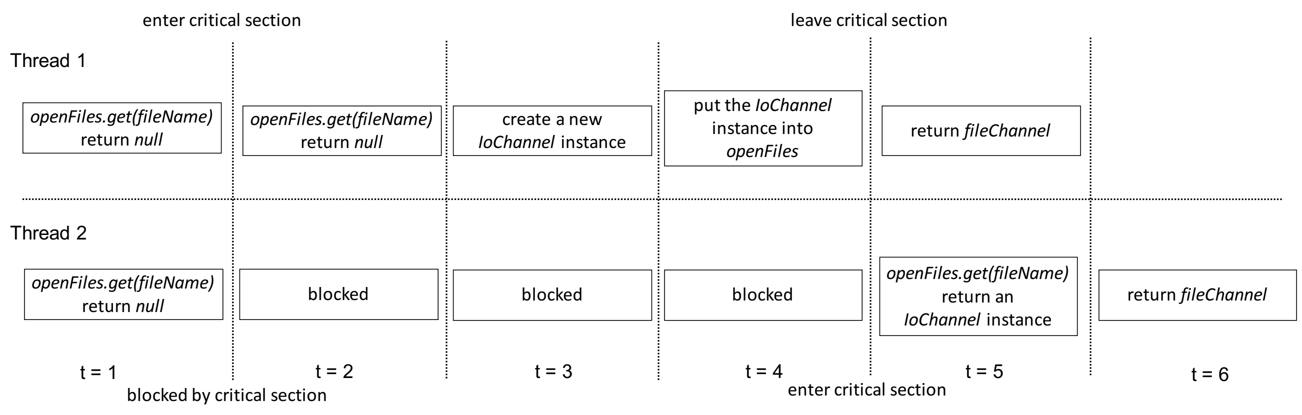
enter critical section
leave critical section
Thread 1
openFiles.get(fileName) return null
openFiles.get(fileName) return null
create a new IoChannel instance
put the IoChannel instance into openFiles
return fileChannel
Thread 2
openFiles.get(fileName) return null
blocked
blocked
blocked
openFiles.get(fileName) return an IoChannel instance
return fileChannel
t = 1
t = 2
t = 3
t = 4
t = 5
t = 6
blocked by critical section
enter critical section

Figure 2: With concurrency control

Likewise, two threads fail to obtain an *IoChannel* instance; this time, however, only one thread will be able to create a new instance. When thread 1 enters the critical section, it invokes *openFiles.get()* again to make sure there's no existing *IoChannel* instance before it creates a new instance; then, when thread 2 enters the critical section, it invokes *openFiles.get()* and finds out that the instance has already been created; thus, it simply returns the instance.

1.3 BlockId

While a *BlockId* has fixed *fileName* and *blkNum*, previous implementation computes the hash code every time the method *hashCode()* is invoked. To avoid this inefficiency of recomputation, we add an attribute *blkHashCode* to *BlockId* and initialize it in the constructor. Thus, when *hashCode()* is invoked, we can simply return *blkHashCode*.

```java
private int blkHashCode;

public BlockId(String fileName, long blkNum) {
  this.fileName = fileName;
  this.blkNum = blkNum;
  this.blkHashCode = this.toString().hashCode();
}

public int hashCode() {
  return blkHashCode;
}
```

BlockId.java

Also, we're now able to test the equality of two *BlockIds* faster with the help of the precomputed hash codes. If two *BlockIds* have different hash codes, we immediately know that they are different; if the two *BlockIds* have the same hash codes, it's very likely that they are the same, however, we should still compare the *fileName* and *blkNum* to guarantee the correctness.

```java
public boolean equals(Object obj) {
  ...
  BlockId blk = (BlockId) obj;
  if (this.blkHashCode != blk.blkHashCode)
    return false;
  return fileName.equals(blk.fileName) && blkNum == blk.blkNum;
}
```

BlockId.java

2 Experiments

2.1 Environment

Intel Core i7-4790 CPU @ 3.60 GHz
32 GB RAM
1TB HDD
Ubuntu Linux 16.04
Java Direct I/O (Jaydio)

2.2 Optimization Result

We first show the result after our optimization:

	Throughput (txs)	Average Latency (ms)
Basic	107245	5
Optimized	184296	3

The experiment is performed with **10 RTEs**, **conflict ratio = 0.001**, **write ratio = 0.5** and **buffer pool size = 102400**. The speed up is about **71.8%**.

For the following experiments, we fix the number of RTEs to **10** and vary the conflict ratio, the write ratio and the buffer pool size one at a time.

2.3 Vary Conflict Ratio

Fixed parameters: number of RTEs (**10**), write ratio (**0.5**), buffer pool size (**102400**).

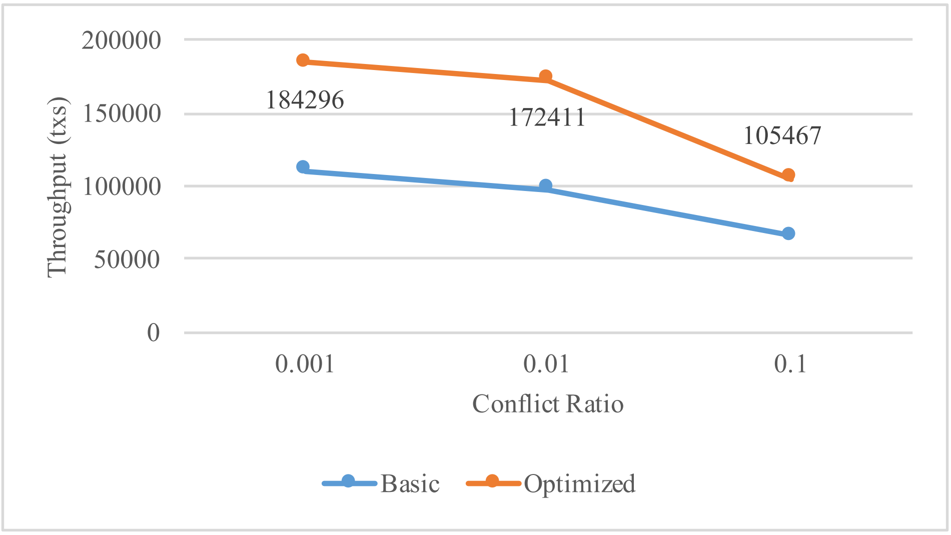

Figure 3: Throughput vs. Conflict Ratio

We can see that the higher the conflict ratio is, the lower the throughput is. We think it's because DBMS needs to guarantee the **atomicity** and **isolation** of transactions; when multiple transactions try to access the same data, some locks or serialization mechanisms are required and incur the performance penalties.

2.4 Vary Write Ratio

Fixed parameters: number of RTEs (**10**), conflict ratio (**0.001**), buffer pool size (**102400**).

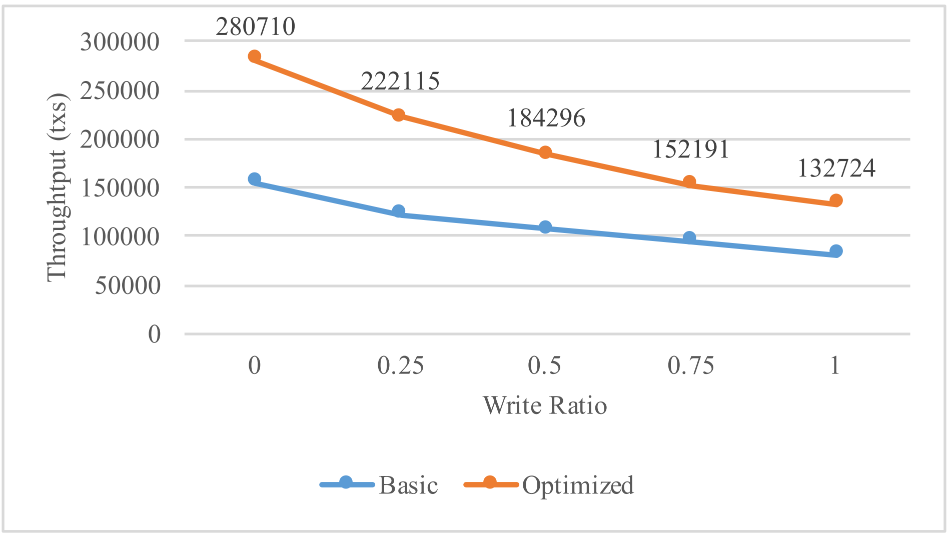

Figure 4: Throughput vs. Write Ratio

The write ratio represents the percentage of write transactions. We can see that the throughput decreases as the write ratio increases. It's trivial since the write transaction has more queries to do (10 more updates), and write operations natually cost higher than read operations.

2.5 Vary Buffer Pool Size

Fixed parameters: number of RTEs (**10**), conflict ratio (**0.001**), write ratio (**0.5**).

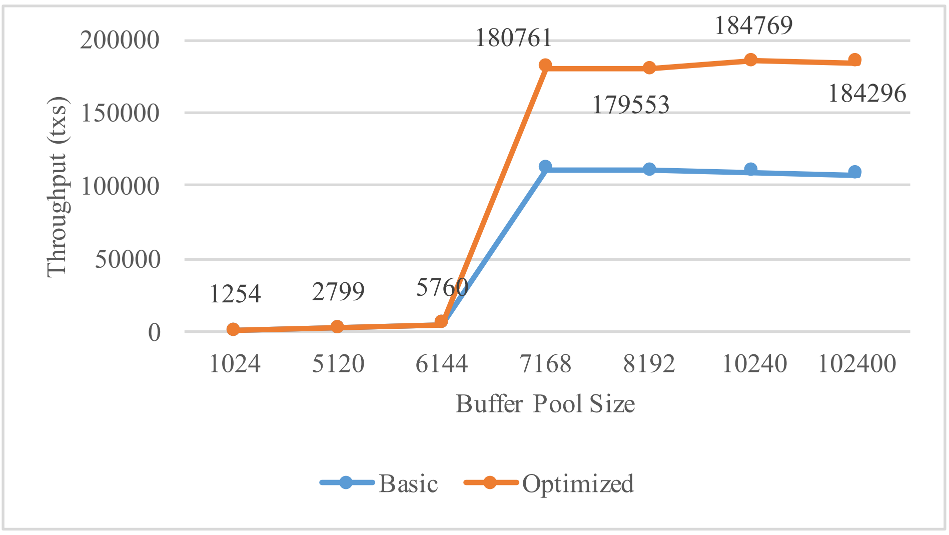

Figure 5: Throughput vs. Buffer Pool Size

Unlike previous results, the throughput does not **gradually** increases as the buffer pool size increases; instead, the throughput is **either very high or very low** depending on the buffer pool size. We think it's because that there exists a **match point** where the size of the buffer pool fit the size of the working set (in our experiment, the match point should lie somewhere in between 6144 and 7168). If the size of the working set is larger than the size of the buffer pool, then transactions would starve for buffers; on the other hand, if the size of the working set is smaller than the size of the buffer pool, then transactions would easily obtain buffers to store its data.