DB assignment 5 phase 1 report

109062233 蘇裕恆
109062320 朱季葳
109062143 林聖哲

implementation

locktable.java

```java
//109062320 set public
public void sLock(Object obj, long txNum) {
    Object anchor = getAnchor(obj);
    txWaitMap.put(txNum, anchor);
    try {
        synchronized (anchor) {
            Lockers lks = prepareLockers(obj);

            if (hasSLock(lks, txNum))
                return;

```

We need to set numerous functions public since the heirachy of the conservative concurrent mgr are not the same as other mgr.
And if we want to use the function for locktable, we need to set those operations to be public.

ConservativeConcurrencyMgr.java

```java
public class ConservativeConcurrencyMgr
extends SerializableConcurrencyMgr
```

We inheritate it from `SerializableConcurrencyMgr` . Inside the function, in order to have implement the conservative policy, We create some functions.

lockAllItems()

```java
// 109062233 [mod] remove unnecessary critical section
public boolean lockAllItems(List<PrimaryKey> readKeys, List<PrimaryKey> writeKeys){
    for(PrimaryKey rKey : readKeys){
        /*if (logger.isLoggable(Level.INFO))
            logger.info("readKeys : " + rKey.getHashCode()  + " ");*/
        lockTbl.sLock(rKey, txNum);
    }
```

Upon the executation phase, we lock all items together by obtaining the `slock` or `xlock` for the object. Note that here we don't need the synchronize keyword since the we obtain the thread safe in `MicroTxnproc.java` .

ONTXCOMMIT() ONTXROLLBACK()

```
}
@Override
public void onTxCommit(Transaction tx) {
    // TODO releaseAll locks
    lockTbl.releaseAll(txNum, false);
}

@Override
public void onTxRollback(Transaction tx) {
    // TODO Auto-generated method stub
    lockTbl.releaseAll(txNum, false);

}
```

And after the transcation has finished or called the rollback , we release all the locks we've obtain previously.

PrimaryKey.java

In order to determine which record we need to lock , we need a primarykey as described in the hint.
The method of the construction a primary took reference on `SearchKey.java`

GENHASHCODE()

```
private void genHashCode() {
    //take reference on query.planner.opt.AccessPath
    //this.hashCode = preAp.hashCode() + newTp.hashCode();
    this.hashCode = tableName.hashCode() + keyEntryMap.hashCode();
}
```

Inside the `Primarykey()` , we have the `genhashcode()` , which generate the hashcode of the `primarykey` .

```
private int hashCode;
public PrimaryKey(String tableName, Map<String, Constant> keyEntryMap) {
    this.tableName = tableName;
    this.keyEntryMap = keyEntryMap;
    genHashCode();
}
```

The `ketEntryMap` and the `tableName` which tell the basic information fit the `primaryKey`

EQUAL()

```java
@Override
public boolean equals(Object obj) {

    if (obj == null)
        return false;

    if (this == obj)
        return true;

    if (!obj.getClass().equals(PrimaryKey.class))
        return false;
    PrimaryKey targetKey = (PrimaryKey) obj;
    // 109062233 [mod] rewrite the equals method
    if(!targetKey.getTableName().equals(this.tableName)){
        return false;
    }
    if(!targetKey.getKeyEntryMap().equals(this.keyEntryMap)){
        return false;
    }

    return true;        Wewe, last week • feat:finish todos
}
```

The equal function that is also required when checking whether the two primarykey are the same.

MicroTxnProc.java

EXECUTESQL()

```java
@Override
protected void executeSql() {
    MicroTxnProcParamHelper paramHelper = getParamHelper();
    Transaction tx = getTransaction();
    //need to be critical section
    Latch.lock();
    // 19062233 [fix] get the ConservativeConcurrencyMgr construct correctly
    ConservativeConcurrencyMgr ConservConcurMgr = new ConservativeConcurrencyMgr(tx.getTransac
    List<PrimaryKey> readKeys = getReadKeys();
    List<PrimaryKey> writKeys = getWriteKeys();
    //System.out.printf("read: " + readKeys.size() + " write: " + writKeys.size() + "\n");
    boolean finish = ConservConcurMgr.lockAllItems(readKeys, writKeys);
    Latch.unlock();
```

We overwrite the `executeSql()` function such that it first construct the `conserativeConcurrencyMgrm` and then get the read key and write key for that transcation.
Note that here we use the `ReentrantLock()` to lock the transaction to gurantee the deterministic execution.

GETREADKEY()

```java
}
//109062320 add getReadKeys()      Wewe, last week • feat:finish todos …
public List<PrimaryKey> getReadKeys(){
    MicroTxnProcParamHelper paramHelper = getParamHelper();
    List<PrimaryKey> ReadKeyList = new ArrayList<>(paramHelper.getReadCount());
    for(int idx = 0; idx < paramHelper.getReadCount(); idx++){
        Map<String, Constant> keyEntryMap = new HashMap<String, Constant>();
        keyEntryMap.put("i_id", new IntegerConstant(paramHelper.getReadItemId(idx)));
        ReadKeyList.add(new PrimaryKey("item", keyEntryMap));
    }

    return ReadKeyList;
}
```

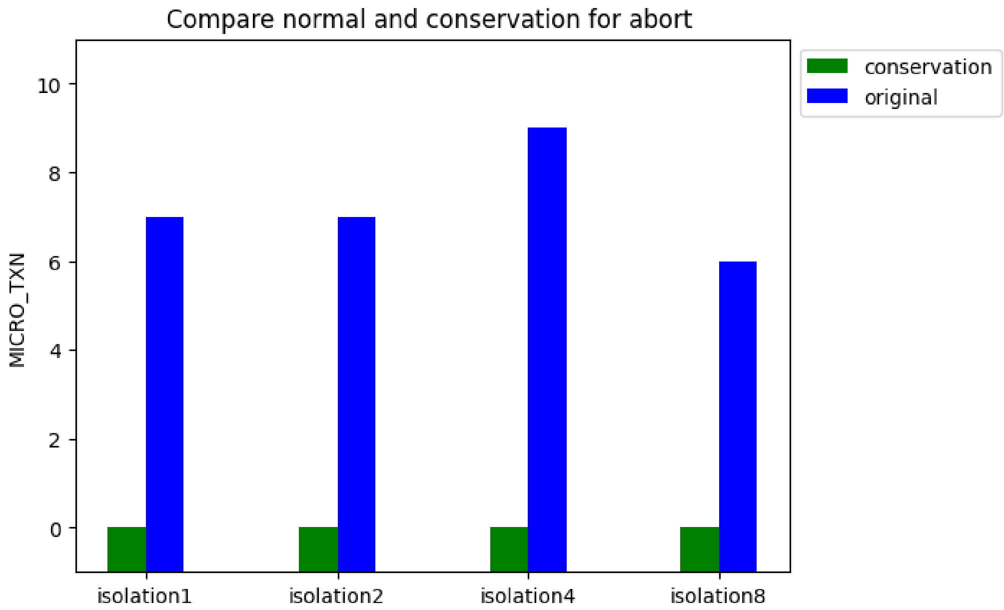
For the `getReadKey()` , we use a List of PrimaryKey to store the object that we want to lock.

challenge of implementing conservative locking for the TPC-C benchmark

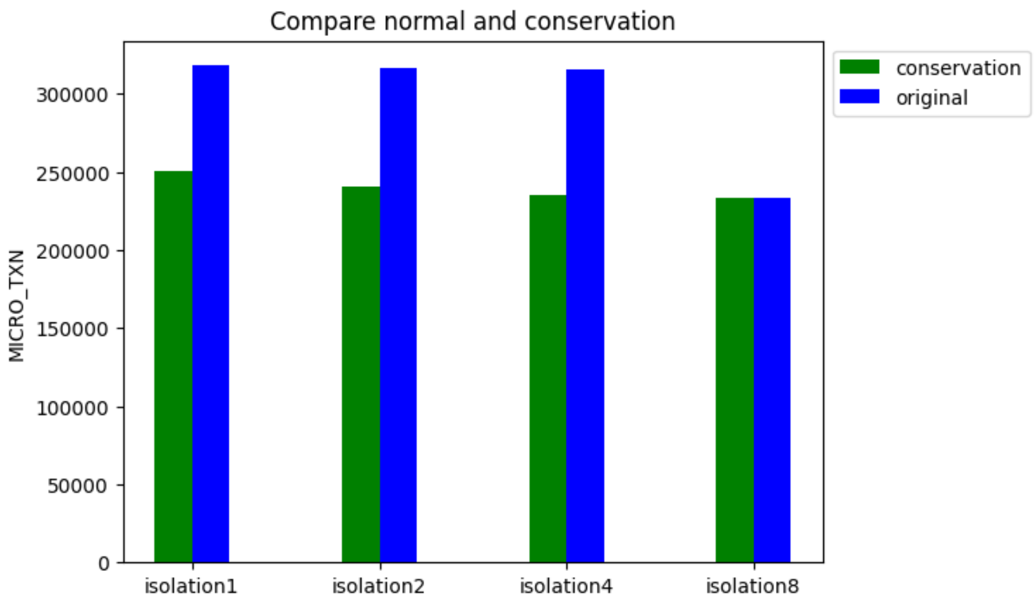
Since unlike the simple method of sp that only has
"SELECT i_name, i_price FROM item WHERE i_id = " + iid, tx
“UPDATE item SET i_price = " + newPrice + " WHERE i_id =” + iid,tx
TPC-C benchmark have more tables that we need to take into consideration. Moreover , the parameters we need to handle also increase such as cwid, cdid, cid/clast ... etc.
Which make it hard upon the paramhelper.
Last but not least , the tpc-c benchmark has inset command , which required us to rewrite some api calls instead of using the simple read / write as we used currently.

Experiment

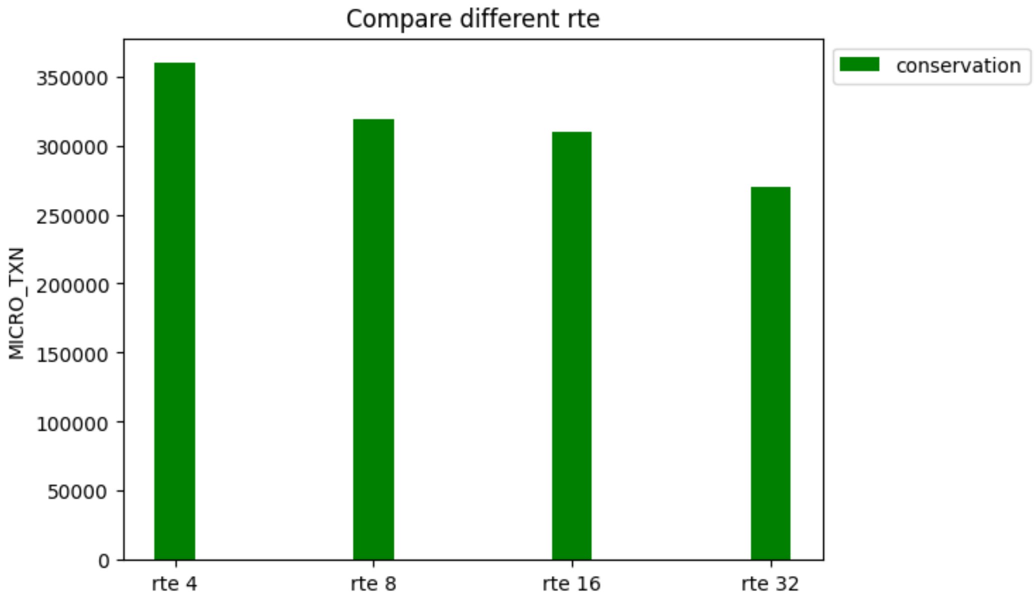
```
Environment
Intel® Core™ i5 CPU 4-Core @ 1.4 GHz
RAM : 8 GB 2133 MHz LPDDR3
MacOS Ventura 13.2.1
```



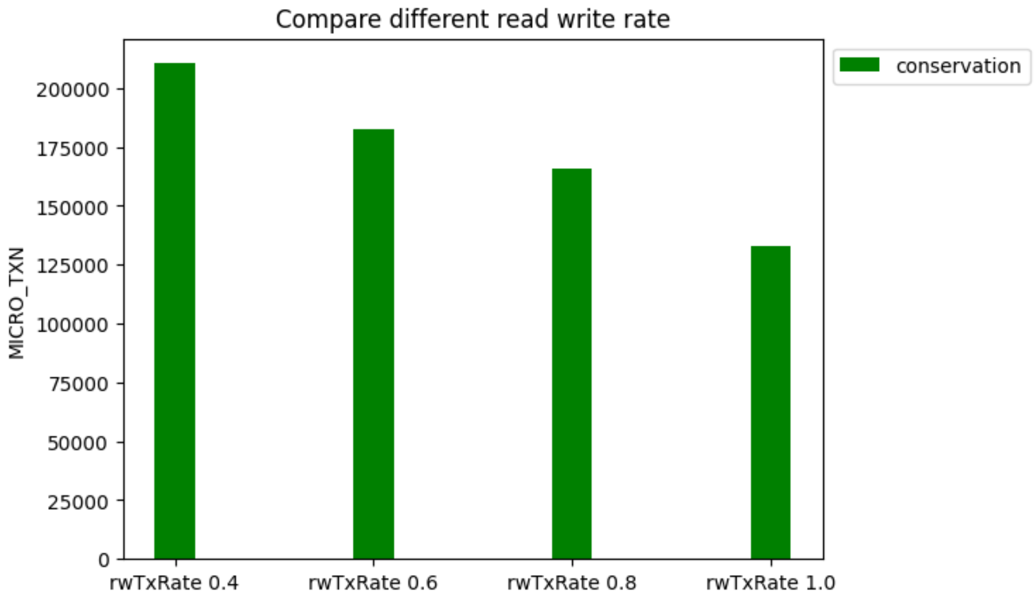
At the begining we first check if there exsist some abort after we implement conservative concurrency Mgr which should be no abort theorically. According the output report and the statistic chart, we can find we fit this requirement.



For this part the isolation 1 represent read uncommitted, isolation 2 represent read committed, isolation 4 represent repeatable read, isolation 8 represent serializable. For different islation in conservation, the TXN is almost the same since the conservation lack the resource for whole execution period. And for original, the TXNfall to almost the same for isolation 8 since it implement serializable.



For this part, since the remote terminal executors for benchmarking increasing, although there is no dead lock and cause abort, the remote terminal executors still need to wait for locking the read/write sets of a given transaction before execution, and thus when the remote terminal executors the TXN decrease.



For this part, since the write TX consume more time than the read Tx (the writeTx need to wait others to finish write but read didn't). According the infomation above, we can simply imaging that since when the rw rate increase, the waiting time will increase, and the TXN will fall obviously.