Introduction to Database System - Assignment 5 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 implemented.

- vanilladb.properties
- `org.vanilladb.core.sql`
  - PrimaryKey
- `org.vanilladb.core.storage.tx`
  - TransactionMgr
  - `org.vanilladb.core.storage.tx.concurrency.conservative`
    - ConservativeConcurrencyMgr
    - ConservativeLockTable
- `org.vanilladb.core.sql.storedprocedure`
  - StoredProcedure
- `org.vanilladb.bench.server.param.tpcc`
  - TpccSchemaBuilderProcParamHelper
- `org.vanilladb.bench.server.procedure`
  - StartProfilingProc, StopProfilingProc
  - `org.vanilladb.bench.server.procedure.micro`
    - MicroTxnProc
    - MicroCheckDatabaseProc, MicroTestbedLoaderProc
  - `org.vanilladb.bench.server.procedure.tpcc`
    - NewOrderProc, PaymentProc
    - TpccCheckDatabaseProc, TpccSchemaBuilderProc, TpccTestbedLoaderProc,

2. Implementation Details

PrimaryKey

Most of the functions in our implementations are the same as the solution, excluding the functions `equals()`, `genHashCode()`, and `toString()`.

For `equals()`, we only check if the given object has the identical hash code of the instance of `PrimaryKey`. Yet, the solution provides a more careful way that checks if the object equals the instance, if the object is null, or if the object has the same class name, table name, and key entry map.

For `genHashCode()`, the hash code in our implementation is the additives of the hash code of the table name and those of key entry map items, while, in the solution, it is simply the additives of the hash codes of the table name and the key entry map. The time taken for calculation may be longer in the former than in the latter for it needs to iterate through the entire key set of the key entry map. However, since the purpose of generating hash code is to identify each record by its primary keys uniquely, we just need to ensure that each primary key has a unique hash code no matter what the computation is. Considering this, both ways have met the requirement and there should be little difference in the performance.

In the solution, the function `toString()` is also implemented. It iterates through the entries of the entry map and prints out their field and value, respectively.

It is also worth mentioning that the class `PrimaryKey` is put under the same directory as `TransactionMgr` in our implementation but the file is put under `org.vanilladb.core.sql` in the solution for future use.

ConservativeConcurrencyMgr

We have implemented a simpler version of the conservative concurrency manager. That is, we collect the read/write objects in advance, then create the transaction, and get locks of these objects - the corresponding code can be seen in `MicroTxnProc`. This way, the section that creates transactions will be locked by a global lock until it gets the locks of its read/write objects, which can make sure that the transactions with lower `txNum`s acquire locks before transactions with higher `txNum`s. However, it is fairly slow because the transactions are able to be created only after the previous transaction gets the locks of its read/write objects.

To address this problem, we need an advanced version of the concurrency manager that implements a queue-version lock table and lets the transaction request the lock right after it was created (in the global lock) and get the lock later (outside the global lock). This version has been done in the solution.

In addition to `getLocks()` in our implementation - or `acquireBookedLocks()` in the solution - that actually locks on the read/write object, functions that request locks, say `bookReadKey()`, `bookReadKeys()`, `bookWriteKey()`, `bookWriteKeys()`, are introduced. The concurrency manager maintains data structures that store the read/write objects and booked objects. When requested locks, the concurrency manager will ask the lock table for locks on the requested objects if they haven't been booked. The read/write objects will be stored, and later used to acquire locks in `aquireBookedLocks()`. Lastly, the locks will be preserved until the transaction is committed or rolled back.

Moreover, in the solution, crabbing locking on the B-Tree index has also been implemented. Methods, such as `modifyLeafBlock()`, `readLeafBlock()`, `crabDownDirBlockForModification()`, and `crabDownDirBlockForRead()`, that book and acquire read/write locks specifically for B-tree index blocks are provided. The concurrency manager ensures the concurrent transactions will properly lock on B-tree indexes, which prevents conflicting access and maintains the integrity of the index structure. When these locks are no longer needed, methods such as `releaseIndexLock()`, `crabDownDirBlockForRead()`, `crabDownDirBlockForModification()` help release the acquired locks on the index blocks.

ConservativeLockTable

To implement the advanced version, the conservative version lock table is needed. The lock table serves as a supporting structure that provides reservations of the required locks for each transaction.

In the solution, the key component of `ConservativeLockTable` is the method `requestLock()`. It locates the lockers for `PrimaryKey` and puts the `tx` in the `requestQueue`. When locks are requested, the functions, `slock()`, `xLock()`, etc., will check if the current `tx` is at the front of the `requestQueue`. If not, the transaction will keep waiting until it becomes the head of the request queue. This way, the lock table ensures that the transactions always acquire locks in order of their transaction numbers.

TransactionMgr

We have modified the `TransactionMgr` to accommodate conservative concurrency. However, the solution simply alters the properties to use the conservative concurrency manager instead of the serializable one, so the `TransactionMgr` remains the same here.

StoredProcedure

The overall design pattern of our implementation closely aligns with the proposed solution but in different ways. For convenience, we separate the explanation into two parts. The first part is to prepare primary keys for locking and the second one is to serially execute transactions.

Firstly, we need to acquire the primary keys of the read/write objects for conservative locking. In the solution, an abstract function `prepareKeys()` is introduced. The affected classes, including `MicroTxnProc`, `PaymentProc`, and `NewOrderProc`, will be explained in the following sections. The other child classes will do nothing in the `prepareKeys()`. Regarding our implementation, the modification is similar to the proposed solution though it is implemented within the `MicroTxnProc`.

Secondly, to ensure that the transaction will get its read/write locks in order, we need a critical section that creates transactions and requests for locks. In the solution, this is done by `scheduleTransactionSerially()`. A `ReentrantLock` is introduced and wraps the transaction creation process. The transaction manager will create a new transaction, and `bookReadKeys()` and `bookWriteKeys()` were invoked to reserve read/write locks on primary keys. This thereby guarantees the order of each transaction. Later on, the reserved locks would be acquired before `executeSql()`. Also, this part was done only in the `MicroTxnProc` in our implementation.

MicroTxnProc

In the solution, the abstract function `prepareKeys()` is implemented. It constructs a set of `PrimaryKey` of read/write objects, where each `PrimaryKey` contains a key map that encapsulates a single field `i_id` along with the item id of the corresponding record. Here, our implementation - `getReadKeys()` and `getWriteKeys()` - is identical to the solution. We override the function `prepare()` to accommodate conservative concurrency. The read/write keys are obtained here before the transaction manager creates the transaction.

Furthermore, as mentioned above, we ensured the transitions are executed in a serial manner within the `MicroTxnProc`. The function `prepareTx()` will get a global lock before asking the transaction manager to create a transaction. It will release the global lock until the transaction is successfully created and get locks on its read/write objects. This works the same way as the `scheduleTransactionSerially()` does in the `StoredProcedure`.

Specifically, there's no difference in performance yielded by this part, since the underlying logic of both implementations is the same.

TpccSchemaBuilderProcParamHelper

The schema of the TPC-C benchmark has been slightly changed. For table DDL, a new field `h_id` of integer type is added to the table history. Besides, the TPC-C no longer supports index DDL on tables `history`, `orders`, `new_orders`, and `order_lines`.

PaymentProc

Despite the challenges of implementing conservative concurrency on TPC-C, the functions `prepareKeys()` are still assigned to `PaymentProc` and `NewOrderProc`.

The primary keys for read/write objects of each SQL command are created and added to the read/write key sets to wait for locks. However, we haven't addressed the problem that we can't lock on a record that is newly inserted within the same transaction. Considering this characteristic, we still cannot run the TPC-C benchmark.

In addition, a hard-code history ID is introduced and used to fit into the new schema of table `history`.

NewOrderProc

The function `prepareKeys()` will add the read/write keys to the read/write sets to acquire locks for the transaction. The next order ID of each district order ID is also hard coded.

3. References

Source Code