Project 3: OMVCC (due Tuesday, May 24 at 23:55pm)

In this project, you are asked to implement a (main-memory) data store with optimistic multi-version concurrency control (OMVCC). The objects stored in this key-value store are <int, int> key-value pairs.

Having a good understanding of the standard MVCC (multi-version concurrency control) and OCC (optimistic concurrency control) algorithms is useful for this project. Please use the course materials (lectures and books) in order to learn about these topics.

OMVCC Specification

Lifecycle

The lifecycle of an OMVCC transaction starts by a call to "begin" where the transaction starts its execution. The execution phase of a transaction consists of zero or more calls to "read", "write" or "modquery" operations:

- read(key): returns the value associated with key
- modquery(k): returns the list of values where the value is divisible by k
- write(key, value): writes the value associated with key

Finally, the transaction either commits via an explicit call to "commit", or gets aborted via an explicit call to "rollback" or an exception thrown by "read", "write" or "commit" (which in this case, the transaction should be aborted implicitly, before throwing the exception).

Timestamps

When a transaction starts, two timestamps are assigned to it, "startTimestamp" and "transactionID". "startTimestamp" is used as a reference timestamp for reading data, and all the committed versions (i.e., the versions created by committed transactions) in the database with a lower commit timestamp are visible to the transaction. "transactionID" should be a number bigger than any start or commit timestamp in the lifetime of the system, which is unique among active transactions. This timestamp resembles a temporary commit timestamp and will be used for referencing the transaction during its lifetime, and all the versions created by a transaction should have this timestamp, so they will remain invisible to all other active transactions. After committing the transaction, this timestamp is replaced by the actual commit timestamp of the transaction.

Data Manipulation

Each key has a list of versions assigned to it. Each write operation for a given key creates a new version for its associated value. If the key did not exist before, it acts like an insert; otherwise it is an update operation.

When creating a new version for a given key, the operation should fail (and abort and throw an exception) if there exists already an uncommitted version or a

committed version for which its commit timestamp is newer than the startTimestamp of the current transaction. If a transaction writes more than once on a given key, it overwrites the existing version. Moreover, each transaction keeps track of the set of versions written by it, which we refer to it as "undoBuffer".

Query

The query operations (i.e., read and modquery) read the most recent committed versions as of the time that transaction began (identified by "startTimestamp"), or the versions written by the transaction itself if one already exists. Moreover, each query operation creates a "predicate" object, which encapsulates the condition for the query. For example, read(8) creates a predicate that checks whether the key is equal to 8, and modquery(7) creates a predicate that checks whether the value is divisible by 7. Each transaction keeps track of the set of predicates used in its query operations during its execution.

Validation and Commit

In order to guarantee serializability, OMVCC has a validation phase before successfully committing a transaction. Read-only transactions (i.e. transactions with an empty "undo buffer") do not require validation and skip this phase. For a non-read-only transaction T to be valid, OMVCC gathers all the transactions that committed after T.startTimestamp (from a list of recently committed transactions). Then, it checks whether any version in the "undoBuffer" of these transactions matches against any predicate in the set of predicates of T. For a version V created by a write operation that updates an existing key, the previous version of V should also be matched against any predicate in the set of predicates of T.
If no match is found, T is valid and can be committed, and a "commitTimestamp" is assigned to T from the same sequence generator that generates startTimestamp, and then the timestamp of all versions written by T should be updated to have its "commitTimestamp".

If an operation is to be refused by OMVCC, abort its transaction (what work does this take?) and throw an exception. Throw exceptions when necessary, such as when we try to execute an operation in a transaction that is not running; when we try to read/delete a nonexistent key, write into a key where it already has an uncommitted version, etc. You may, but do not need to, create different exceptions for operations that are refused and cause the transaction to be aborted. Keep it simple!

Remarks

- The key-value store is initially empty, and the only way to interact with your implementation is through OMVCC API.
- After handling the commit operation for a transaction, a successful commit cannot be revoked later (in order to have durability), and an unsuccessful commit should be automatically followed by an abort and throwing an exception.

- After a transaction is terminated, any further operations from that transaction should be ignored.
- Garbage collection of versions is optional.
- There is only a single thread of execution for testing your implementation, so taking care of concurrent threads is not needed for this project.
- You can implement OMVCC in either Java or Scala.
- The interface is given to you in OMVCC.java and OMVCC.scala. Please, keep the interface. We want to test automatically. Any changes to the given interface, might lead to missing the whole grade, as the automatic tests might fail.
- We've already created two sample test files and a shell script to run them. You can download OMVCC_Test_Java.zip or OMVCC_Test_Scala.zip depending on the language that you use for your implementation. These tests as well as some other similar ones will be used for grading your submitted implementations.
- OMVCC is a simplified version of the algorithm presented in [1], which has the same behavior with more optimizations. The complete specification of OMVCC is provided above and you do not have to refer to this paper unless you want to know about more details.

Deliverable:

- The modified and commented OMVCC.java or OMVCC.scala should be submitted. If you need other helper classes in your implementation, please keep them as private classes in the same java file.

Reference:

[1] Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems" by T. Neumann, T. Mühlbauer and A. Kemper in SIGMOD 2015 (http://dl.acm.org/citation.cfm?id=2749436)