SUTD 2021 50.043 SimpleDB Project Part 3 Writeup Report Document

James Raphael Tiovalen / 1004555

Implementation Description

For part 3 of the project, I also simply implemented most of the skeleton methods/functions and classes specified by the handout. I also followed the implementation guide as laid out by the exercises closely, and hence, there are very few deviations from the intended path.

Several design decisions that I have made include:

- Implementing a `LockManager` class, as specified by the handout, which is used by the `BufferPool` class. A helper `RWLock` class was also defined.
- Implementing strict two-phase locking at page granularity. If we were to implement tuple-level locking, we can keep track of the mapping between tuples to their locks, as well as the tuples held by the transaction in `LockManager`, and calls to acquire locks would happen in the `insertTuple` and `deleteTuple` methods of `BufferPool` (instead of in `getPage` instead). Tuple-level locking might actually become much more costly (more overhead) since we need to manage more locks, even if it becomes less restrictive/prohibitive. In comparison, page-level locking is relatively faster (even though it would lead to more conflicts) and thus more suitable for multi-user database management systems.
- Implementing a NO STEAL/FORCE buffer management policy.
- Implementing a deadlock detection method by building and using a wait-for graph and then detecting cycles in said dependency graph. This wait-for graph mechanism can be quite costly in terms of time complexity, since we need to consult and hit the graph every time we acquire locks to check for potential presence of cycles, as well as space complexity, since we need to maintain this separate dependency graph object. Another method would be to maintain a global transaction ordering (based on the transaction ID). This way, we do not have to maintain a separate dependency graph object. However, this method also has its drawbacks since maintaining a global order can be difficult as the scale of the database goes up, and a global order might force a transaction to grab a lock earlier than it would like, tying up resources for too long.

Several challenges that I have faced include:

- Implementing certain methods in the `LockManager` class with the specific Java oddities/eccentricities. Older implementations of some methods exhibit weird behavior such as deadlocking if the method is `synchronized` (or not) for some reason. Race conditions and various `ConcurrentModificationException`-related errors had to be taken care of as well during development.
- To ensure correctness, it is insufficient to pass the `TransactionTest` system test only once. Due to the very inconsistent, fickle, and tricky nature of deadlocks and concurrency synchronization problems, multiple runs of the unit tests and system tests had to be done to ensure that my code implementation is indeed correct.
- Handling certain weird cases when my code passed the `DeadlockTest` unit test but failed the `TransactionTest` system test. It involved interacting with the way threads and processes are spawned, handled, and garbage collected in Java during repeated runs of the system tests, which was quite annoying. Thankfully, I managed to somehow (quite miraculously and magically, in fact) fix it. Synchronization, concurrency, and parallelism are always big sources of headache. :D

The original provided API of the project was not changed.

All of the requested elements/parts for part 3 are completed.