CS 2550 Project Phase 2 Writeup
Nate Ackerman, Bin Dong, Paul Elder, Shibo Xing

Scheduler:

Detailed description of the design and implementation of the concurrency control strategies under different execution modes, including the compatibility table, the locking mechanisms for each operation, and so on.

- The scheduler uses the lock manager to control when instructions are allowed to execute from any given transaction. The scheme is strict 2PL. The lock manager uses a custom non-blocking read-write lock. The locks allow the ability to first express interest in a lock, but if you are not able to get it, you are queued up but do not block so that the scheduler may move on to the next instruction from a different transaction. A lock being acquired as read is shared, but write is exclusive. A read lock can be upgraded to a write lock if no other transaction wants that lock. When an instruction is able to acquire all of the locks needed, then the instruction is allowed to run.
- Locks are implemented at the table and tuple level. Whether the lock is acquired as a read or write lock for a tuple is determined by whether the tuple is being written or read. However, for the table, we found that we only needed the write lock when deleting the table, as multiple transactions could write into the same table concurrently as long as they were writing tuples with different primary keys.
- If the sequence of instructions is a process and not a transaction then no locks are acquired and the instruction is always allowed to run when the scheduler deems it to be that process's turn.
- Both round robin and random scheduling is implemented. Round robin leads to deterministic runs while random will not produce the same result every time.
- When an instruction is deemed runnable, we first build the necessary before image for that instruction type and log it with the instruction. Only once this is done is the instruction actually run.
- When the scheduler sees an abort or commit operation, it will release the locks it acquired per the strict 2PL scheme. If the instruction is abort, the scheduler will check the logs and do the reverse operation defined per each operation type.

Detailed description of the design and implementation of the deadlock detection and handling mechanisms.

- For our deadlock detection, we simply use a graphing library to graph all the edges. If a cycle gets detected, then a deadlock has occurred. There are 4 conditions that we have to check. These four conditions are:
    1. If the requester is trying to acquire a READ LOCK while there may (or may not) be a READ LOCK on the data item
        a. We ignore it if it's a table lock or we are the lock owner
        b. If we want a tuple READ LOCK, and we are not the lock owner
    2. If the requester is trying to acquire a READ LOCK while there is a WRITE LOCK on the data item
        a. In this case, we issue an edge from the Transaction ID of the requester to the WRITE LOCK owner of the data item.

3. If the requester is trying to acquire a WRITE LOCK while there is a READ LOCK on the data item
    a. In this case, we issue an edge from the Transaction ID of the requester to EACH and every transaction that has a READ LOCK.
4. If the requester is trying to acquire a WRITE LOCK while there is a WRITE LOCK on the data item
    a. In this case, we issue an edge from the Transaction ID of the requester to the WRITE-LOCK owner of the data item.

- Each and every time a request to acquire a lock is queued up, we will run the deadlock detection algorithm. We use a queue-ing system to figure out who to give the lock to. The lock is given at a first come first come basis, therefore, if a transaction wants a lock and there are other transactions that want the same lock ahead of it, the transaction would have to wait until the completion of all the other transactions, thus being queued up. After the request is placed on the queue, we check if it will cause any deadlocks via the graphing library. If a cycle is detected, we kill the transaction that introduced the deadlock, solving the deadlock issue.

Detailed description of the design and implementation of the recovery mechanisms under transaction abortion.

- Our recovery mechanism for transaction abortion is Un-Dos. Transaction Abortion has many scenarios. Below, I will explain how we handle each scenario. The undos are done in reverse order of operations in the log that match the transaction ID that is being aborted or killed due to deadlock.
    - Undo-ing Delete Table
        - In the log we store a copy of the table as the before image, along with the metadata that the LSM mechanisms need to efficiently use the storage. The records and metadata are written back to disk when undo-ing the delete.
    - Undo-ing Reads
        - We ignore this because a read doesn't do harm our database.
    - Undo-ing New Record
        - The record is deleted when this operation needs undone. The delete is based on the primary key in the log entry.
    - Undo-ing Update Record
        - The record first has its values stored as a before image before the update is done. When undo-ing, we simply overwrite the record to the old values.
    - Undo-ing Delete Record
        - The record is stored as the before image in the log entry. When undo-ing, we simply write the record back.

Transaction Manager:

Detailed description of the design and implementation of the operation reading mecha-Nisms.

- The duty of the Transaction Manager has been somewhat delegated to all the other classes. It has been bundled into our Instruction class and our scheduler class. Each Transaction is wrapped around by the InstructionSequence class. Each InstructionSequence has an attribute that will tell us whether the current sequence is a Transaction or Process. We InstructionSequence Object is managed by the Scheduler. For further implementation details on how the Scheduler handles the Transactions, see Scheduler section up above.

Testing:

We have many test scripts, found in `test_scripts/phase2` in the project repository. The scripts that are in directories are meant to be run in parallel. We have tested them all and they produce the correct output. The output is confirmed in our output log (printed to stdout) to check if there is a deadlock, as well as in the storage saved on disk, confirmed with `hexdump -C storage/*/*/*/*`. For the normal tests, we will show here one sample execution and expected final storage state for each category of correctness.

For each normal test case, state clearly the transaction setup, the expected output with reasons, and the actual output. If the actual output is different from the expected output, analyze the reason.

Concurrency control:

We can observe concurrency control in action by seeing the difference between running the same two scripts in parallel as processes and as transactions, run in round robin.

Script 1:
B 0/1 (0 for process, 1 for transaction)
W X (2, Alice, 111-111-1111)
W X (2, Bob, 222-222-2222)
C

Script 2:
B 0/1
W X (2, Charlie, 333-333-3333)
C

If these two are run as processes, we expect the final state of the storage to be (2, Bob, 222-222-2222), since `W X (2, Bob, 222-222-2222)` is the final instruction to run. On the other hand, if these two are run as transactions, we expect the final state of the storage to be (2, Charlie, 333-333-3333), since `W X (2, Charlie, 333-333-3333)` will wait for transaction 1 to complete execution before being able to execute. No deadlock is expected. The result that we got matches the expected results. These scripts can be found in `test_scripts/phase2/simple_conc_trans` and `test_scripts/phase2/simple_conc_proc`

CS 2550 Project Phase 2 Writeup
Nate Ackerman, Bin Dong, Paul Elder, Shibo Xing

Deadlock Detection:

Script 1:
B 1
W X (1, Alice, 412-000-0000)
W X (2, Bob, 412-000-0001)
A

Script 2:
B 1
W X (2, Bob, 412-000-0001)
W X (1, Alice, 412-000-0000)
A

Expected deadlock (with round robin) or either deadlock or no deadlock (with random), and nothing in storage. Result is correct. These scripts can be found in `test_scripts/phase2/simple_both_abort_deadlock`

Recovery

Was not tested because restarts were not a requirement for this project.

Others

When a table delete instruction is executed, we gzip the entire table and save that as the before image. When this instruction is undone, the before image is gunzipped and restored. We have scripts in `test_scripts/phase2/delete_table` to test this.

Script 1:
B 1
W X (1, Alice, 111-111-1111)
C

Script 2:
B 1
D X
A

With round robin, record (1, Alice, 111-111-1111) will first be written to table X. Transaction 2 will then try to delete table X, but needs to wait on the write lock. Transaction 1 commits, then transaction 2 deletes table X, but aborts, so the deletion of table X is undone. The storage

should then contain the record (1, Alice, 111-111-1111). The results that we got match this expectation.

For each benchmark test case, state clearly the transaction setup, the execution result (e.g., if the program crashed), and the efficiency measures in throughput.

Concurrency control:

The concurrency control test had two types of scripts, and odd script and an even script. The odd script alternates between reads and writes, for a total of 100 operations, and commits. The even script alternates between writes and reads, for a total of 100 operations, and commits. We generated 50 odd and 50 even scripts (named trans1.txt, trans2.txt, and so on until trans100.txt). They are all transactions, and they all read and write to and from the same record and table. These can be found in `test_scripts/phase2/bench_conc`

The scripts were executed with the round robin scheduler, in alphanumerical order (meaning trans100.txt was first, then trans10.txt, trans11.txt and so on, with trans9.txt being the last). This execution completed in 1 minute 56 seconds, on a single core of an i9-9900K, with a CPU frequency governor on powersave.

Deadlock:

The deadlock control test had a loop of scripts with one transaction each. Each transaction n would do three writes on its own table, and then would do a write on the table of n+1 mod number of transactions. There were 100 transactions, and each transaction had a duplicate as well. What would happen then is that the duplicate transactions would wait on the main transactions and never run, and after the 100th main transaction tried to write to the table of the first transaction, the deadlock loop would complete, and the 100th transaction would be killed. These scripts can be found in `test_scripts/phase2/bench_deadlock`

The scripts were executed with the round robin scheduler, in alphanumeric order. This execution completed in 33 seconds, on the same i9-9900K as before.

Recovery

Was not tested because restarts were not a requirement for this project.

Others

We also ran a benchmark test for table deletion and undoing the table deletion, with many inserts in between. We have two scripts, with 100 transactions each. The first script does 100 unique inserts into a table, and commits. The second script deletes the table, and aborts. These scripts were executed with the round robin scheduler, with script 1 first and script 2 second. Thus the 100 inserts would run first, as the delete table is waiting for transaction 1 to release the lock. Then after transaction 1 completes, transaction 2 deletes the table and aborts,

causing the table to be restored. These scripts can be found in `test_scripts/phase2/bench_delete_table`

These scripts were executed on the same i9-9900K as before, and the execution completed in 4 minutes and 30 seconds.

Others:

Detailed description of the contribution of each team member based on the implemented components. The granularity of the components should be as fine as possible.

- Everybody worked on all components equally via quad-pair-programming
- Most of the finer granularity split work was done in phase 1; phase 2 was very atomic and hard to split