Recitation 18 — Concurrency Control

Context

- Late 2000's, skyrocketing core counts. People speculated we would soon have machines with 1000's of cores. This never happened; thousands of cores are not practical for many software applications (as shown in this paper).

Databases

- Collections of tables of records
- Can be very large
- DBMS: software system to manage a database. In particular, DBMS can automatically handle locking so that the programmer doesn't have to think about it. The beauty of transactions!
- In particular, DBMS can do 2PL. Remember the problem from lecture with 2PL: it can result in deadlock. We have the ability to abort one of the transactions when this happens (great!), but we have a lot of choices on how to do that.

Handling deadlocks in 2PL

- DL DETECT - Build a graph of which transactions are waiting for each other, and use that graph to detect and resolve deadlocks
- NO_WAIT - Just abort a transaction as soon as it blocks attempting to acquire a lock
- WAIT_DIE - Abort a transaction T1 if it started after T2 and it blocks attempting to acquire a lock from T2

Other approaches

- Timestamp ordering - Keep track of transaction that did the latest read or write on each object. Abort transactions that perform a write on an object with a later write or read timestamp, or a read on an object with an earlier write timestamp.
- MVCC - A lot like timestamp ordering, but keep a history of timestamps instead of just one. Allows more concurrency than timestamp ordering but adds overhead.
- OCC - Checks what a transaction read/wrote against any transactions that executed concurrently with it but completed before it read/wrote. Writes don't go to the DB immediately.

Evaluation

- Uses YCSB ("Yahoo Cloud Service Benchmark"), a popular benchmark from this era.
- Under no contention: DL_DETECT, NO_WAIT do okay because they never block or abort. Other schemes suffer because they have to allocate timestamps.
- Medium contention - DL_DETECT falls off after 256 due to "lock thrashing" (transactions waiting for locks and then eventually deadlocking). MVCC does well enough. OCC does less well because transactions have to wait until they complete to abort.
- High contention - nothing does well. Being eager to kill transactions (NO_WAIT) is a win. Overheads of timestamp allocation matter less.

- H-Store comes out looking very good, but this is only a case with no skew; designed to make H-Store look very good.

Discussion

- Why might the authors want to make the H-Store approach look good?
- The 1000 core assumption never came to pass. What does that say about the legacy of this paper? Are the conclusions still valid?
- How much does being able to run 10M+ transactions really matter anyway? How many transactions is enough?
- What would happen with more complex workloads?