Multiversion Concurrency Control-Theory and Algorithms

Bernstein, Goodman (1983)

What kind of paper is this?

- Another framework paper.
- Present a formalism.
- Use it to analyze old algorithms.
- Use it to prove things about new algorithms.

Overall Flow

1. Define model.
2. Use model to define serializability over a non-multiversion database.
3. Extend model to multiversion database.
4. Use model to prove things about existing algorithms.
5. Present new algorithms.
6. Use model to prove things about new algorithms.

Basic Model

- Key element is the partial order "happens-before"
- Goes into a lot of detail about message exchange, Lamport clocks, and then says that we can ignore it. Curious to think about why this had to be written in terms of a distributed system and why one couldn't have left that out entirely.
- Transaction log is a partially ordered set on the reads and writes executed by a set of transactions. (Partially ordered, because it allows for parallelism both within and among transactions.)
- In the single version case, $r(w)_i[x]$ indicates that transaction i read (wrote) item x.
- No repeated reads.
- No reading of uninitialized data.
- Although a log is a partial order, if two operations conflict, then you must know the "happens-before" relation between them.
- Log Equivalence
  - Reads-from relation indicates information flow. Transaction T_j reads-x-from Transaction T_i iff
    1. $w_i[x]$ and $r_j[x]$ are in the log
    2. $w_i[x] < r_j[x]$ (happens-before)
    3. There does not exist a $w_k[x]$ such that $w_i[x] < w_k[x]$ and $w_k[x] < r_j[x]$ (i.e., no intervening operations).
  - Two logs are equivalent iff they have the same reads-from relationships.
  - Can strengthen equivalence by also requiring the same final state.
- Serializable Logs
  - A serial log is a totally ordered log such that all pairs of transactions have a "happens-before" relation.
  - A serial log represents no concurrency and is obviously correct.
  - Any log equivalent to a serial log is serializable (SR).
  - Define a serialization graph of a log be the graph formed whose vertices are transactions and whose directed edges represent the "happens-before" relation (i.e., an edge from T_i to T_j indicates that $T_i < T_j$).
  - If the serialization graph of a log is acyclic then the log is serializable.

Extending the Model

- We extend our model so that writes produce new versions of objects and reads can read from any version.
- $r_i[x_j]$ means that transaction i wrote the value of x written by transaction j.
- $w_i[x_i]$ is the result of transaction i writing to variable x (it creates a new version i).
- Define a mapping function h that maps a read(write) on an item x to a versioned read(write) on x.
- A multiversion log (MVL)is a partially ordered set such that
    1. Each operation in MVL is an appropriately mapped versioned operation.
    2. For all operations o and p in a particular transaction, if o < p, then h(o) < h(p) (no changing the order of operations within a transaction).
    3. If Transaction j read version i of x (i.e., T_j such that $r_j[x_i]$), then a write of x in transaction i happened-before the read of x in j (i.e., $w_i[x_i] < r_j[x_i]$). In English: you cannot read a version that has not yet been produced.
    4. The reads-from relation in the basic model translates directly. Note that the version read indicates exactly from which transaction a value is read.
    5. As in the basic model, two logs are equivalent if they share the same reads-from relation. But, since the reads-from relation is explicit, we can simply state that MV logs are equivalent if they have the same versioned operations.
    6. Note also that the serialization graph is trivial to produce: an edge exists from i to j for every reads of the form $r_j[x_i]$.
- One Copy Serializability
    - We initially described serialization graphs in the basic model in terms of "happens-before" using it interchangably with reads-from. In the multi-version model, this gets us into trouble, because we can express three transactions T_i, T_j, and T_k such that T_i happens-before T_j happens-before T_k, but T_k reads-from T_i, even if T_j wrote the value that T_k read. This is problematic and could never happen without multiple versions.
    - Define a log as 1-copy serial (1-serial) if for all i, u, and x, if T_j reads-from T_i, then either i = j or T_i is the last transaction preceding T_j that wrote any version of x. This restriction gets rid of the problematic case above.
    - Definition: A log is one-copy serializable (1-SR) if it is equivalent to a 1-serial log.
    - 1-SR is the correctness definition for a multi version log -- a log is equivalent to a non-MV serial log iff the log is 1-SR.
- The 1-SR Theorem
    - Define a version order is a total order over all the versions of x in log (L).
    - The version order relation << is the union of the version orders over all data items in L.
    - The multiversion serialization graph has vertices corresponding to all the T_i and edges from T_i to T_j the same as a regular serialization graph PLUS if $r_k[x_j]$ and $w_i[x_i]$ and k != i, then if $x_i << x_j$ include edge from T_i to T_j else include T_k to T_i. In English: if a transaction reads a version that came before the one you're reading, then you have to come before that transaction; if you're reading a version that came later, then you better come after the version that wrote it.
    - As before, an MV log is 1-SR iff there exists a version order such that the MVSG is acyclic.
    - Determining if an MV log is 1-Serializability is NP Complete (how many found this surprising?).

Multi-version Time Stamping

- Algorithmic Definition
    - A transaction T_i is assigned a timestamp TS(i) at begin.
    - The mapping function h works as follows:
        1. $h(r_i[x]) = r_i[x_k]$ where x_k is the verison of x with the largest timestamp <= TS(i). (Read the last version written before the current transaction's timestamp.)
        2. Reject a write that would render a previous read incorrect (upon encountering $w_i[x]$, if there exists $r_j[x_k]$ such that TS(k) < TS(i) < TS(j)).
        3. If condition 2 is not met, then accept the write $w_i[x_i]$.
- Log-based Definition

1. Every T_i has a timestamp TS(i) satisfying the uniqueness condition that TS(i) == TS(j) iff i == j.
2. All versioned reads and writes are related via the reads-from relation.
3. You can only read things that have already been written: A transaction k can only read a version j if TS(j) <= TS(k).
4. When you read, read the version with the max timestamp before yours (TS3).
5. Cannot overwrite a value that has already been read (TS4).

- Now, given this representation of the problem, we show that logs produced by this algorithm are 1-SR.
- Proof -- fairly straight forward presentation in the paper.

Multiversion Locking

- Transactions and versions are either certified (C) or uncertified (NC).
- Require that all versions read are C.
- Requires that all certified versions for the same element are well-ordered.
- Version order corresponds to certification order.
- Algorithm
    1. Map $r_i[x]$ into $r_i[x_k]$, where x_k is either the last certified version or any uncertified version.
    2. Map $w_i[x]$ into $w_i[x_i]$ (x_i is uncertified at this point).
    3. At transaction end, certify transaction and all written values.
        1. For each data item written, obtain the (blocking) certify lock. (Certify locks are exclusive.)
        2. Verify that each version written by a different transaction but read by the current transaction is certified.
        3. Verify that for every version written, for every already certified version, every transaction that read those versions have been certified.
        4. If the last two conditions are satisfied, the transaction is certified and all its writes are certified.
    4. Need to deal with two kinds of deadlocks! (Yuck)
    5. As above, we now cast the multiversion locking algorithm into statements on logs and show that all logs produced by MV locking are 1-SR.
    6. For details, check out the paper.

Multiversion Mixed Method

- This is actually the "standard" multiversion algorithm used today.
- Reads use multi-version timestamping.
- Writes use MV locking.