Addendum to "High-Performance Concurrency Control Mechanisms for Main-Memory Databases"

Per-Ake Larson[1], Spyros Blanas[2], Cristian Diaconu[1], Craig Freedman[1], Jignesh M. Patel[2], Mike Zwilling[1]

[1]Microsoft, [2]University of Wisconsin – Madison

1. Single-version locking scheduler

Proving the single-version locking scheme correct is trivial, as the scheduler is a 2PL scheduler.

2. Multi-version pessimistic (locking) scheduler

The multi-version pessimistic (locking) scheme is in fact a MV2PL scheduler. Holding a certify (commit) lock on a data item in MV2PL is exactly like having the NoMoreReadLocks bit set in the latest version of the data item in our implementation (see Section 4.2.1). Section 5.5.2 of [WV02] describes MV2PL in detail and proves it only admits 1SR multi-version histories.

3. Multi-version optimistic scheduler

Let us now prove that the multi-version optimistic scheduler only admits 1SR multi-version histories. We use the notation and theorems from Section 5.2 of [BHG87]. The multi-version optimistic scheduler behaves like a MVTO scheduler, with the changes described below.

Let transaction Tx be a committed transaction with a Begin timestamp of TxBegin and an End timestamp of TxEnd.

Property 1: Timestamps are assigned in a monotonically increasing order, and each transaction has a unique begin and end timestamp, such that TxBegin < TxEnd.

Property 2: A given version is valid for the interval specified by the begin and end timestamps. There is a total order << of versions for a given datum, as determined by the timestamp order of the non-overlapping version validity intervals.

Property 3: The transaction Tx reads the latest committed version as of TxRead (where TxBegin <= TxRead < TxEnd) and validates (that is, repeats) the read of the latest committed version as of TxEnd. The transaction fails if the two reads return different versions.

Property 4: Updates or deletes to a version V first check the visibility of V. Checking the visibility of V is equivalent to reading V. Therefore, a write is always preceded by a read: if transaction Tx writes Vnew, then transaction Tx has first read Vold, where Vold << Vnew. Moreover, there exists no version V such that Vold << V << Vnew, otherwise Tx would have never committed: it would have failed during the Active phase when changing the end timestamp of Vold (see Section 3.1, paragraph "Update version")[1].

[1] Notice that all our concurrency control algorithms enforce a stronger property: they use the first-writer-wins rule to abort transactions that participate in a write-write conflict before it is determined whether the first writer will commit. The more relaxed property described here is sufficient to prove correctness.

Property 5: The transaction Tx logically writes at TxEnd, because the version is invisible to other transactions until TxEnd (see Section 2.5, tables 1, 2 on visibility of versions where TB or TE contains the ID of transactions in the Active phase).

The multi-version serialization graph MVSG(H, <<) is a graph defined on a multi-version history H and version total order <<. The MVSG has nodes for the committed transactions in H, and by definition, an edge Ti → Tj exists in the MVSG (where i, j, k are distinct), if and only if:

A) Ti writes Vi and Tj reads Vi
   or
B) Ti writes Vi and Tk reads Vj, where Vi << Vj
   or
C) Ti reads Vk and Tj writes Vj, where Vk << Vj

Let us prove that every edge in the MVSG is ordered with respect to the end timestamp order of the transactions involved. That is, we will prove that any directed edge Ti → Tj will always point from a transaction Ti to a transaction Tj such that TiEnd < TjEnd.

For (A), let Tj read Vi at TjRead < TjEnd. If TjRead < TiEnd, from Properties 3 and 5, it would have been impossible to read Vi, as it's uncommitted. Therefore TiEnd < TjRead < TjEnd, therefore the Ti → Tj edge is ordered with respect to the end timestamp, as TiEnd < TjEnd.

For (B), Tk reads Vj, therefore the read is preceded by Tj writing Vj and committing. Furthermore from Property 3, TjEnd < TkRead, or Vj wouldn't be visible at TkRead. From Proprety 4, since Tj wrote Vj, that means that Tj read Vi (or any later version) at TjRead, where TjRead < TjEnd. Therefore, from (A), TiEnd < TjRead < TjEnd, so the Ti → Tj edge is ordered with respect to the end timestamp, as TiEnd < TjEnd.

For (C), let Ti read Vk at TiRead, where TiRead < TiEnd. Tj writes Vj at TjEnd. There are three cases:

1) TjEnd < TiRead < TiEnd. This violates property 3, as Vj was a committed version at TiRead, therefore Ti would have read Vj, not Vk. (See Section 2.5, "Version Visibility".)
2) TiRead < TjEnd < TiEnd. This violates property 3, as Ti would repeat the read at TiEnd during validation, and would read Vj. However Ti read Vk at TiRead, therefore Ti would fail validation and would never participate in the MVSG. (See Section 3.2, "Preparation Phase").
3) TiRead < TiEnd < TjEnd. Ti would validate the read at TiEnd, read Vk again, and commit. Therefore the edge Ti → Tj is ordered with respect to the end timestamp, as TiEnd < TjEnd.

Therefore, from (A) and (B), all edges in the MVSG are ordered with respect to the transaction end timestamp order, and thus (from Property 1) they cannot be involved in a cycle. Hence, the MV histories that are accepted by our multi-version optimistic concurrency control scheduler are 1SR.

Bibliography

[BHG87] Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman: "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987, ISBN 0-201-10715-5.

[WV02] Gerhard Weikum, Gottfried Vossen: "Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery", Morgan Kaufmann, 2002, ISBN 1-55860-508-8.