Multi-versioning in Main-memory Databases: Limitations and Opportunities

Jose M. Faleiro
Yale University

Multi-version database systems preserve the explicit history of values taken by each database record. They do so by maintaining *versions*; if the value of a record needs to be updated, the system creates a new version, while keeping the previous value untouched. In contrast, a single-version database system updates records in place. As a consequence, in his seminal paper describing the "transaction concept", Jim Gray wrote that multi-versioning represents good practice because transactions leave an audit trail of changes [10]. Unfortunately, because they maintain multiple copies of the same record, multi-version databases have a greater memory footprint than their single-version counterparts. Thus, single-version systems can more easily fit entirely in main-memory. The majority of recent research on main-memory on-line transaction processing databases has therefore been focussed on single-version systems [13, 18, 20, 21]. As the cost of main-memory decreases, however, it will become increasingly practical to keep fully multi-versioned databases main-memory resident. Indeed, several database vendors, such as MemSQL, Microsoft SQL Server, and SAP, have bet on multi-versioning, and already offer main-memory resident multi-version database products [1–3].

Much of the motivation for multi-version databases has been due to its perceived advantages over single-version systems. Chief among these is the fact that multi-version databases have the potential to reduce the number of read-write conflicts among transactions. Because multi-version databases do not update data in place, conflicting reads and writes can be allowed to occur in parallel; reads can be directed to pre-existing versions of records, while writes create new versions. State-of-the-art multi-version databases exploit this greater read-write concurrency under *weak* isolation levels such as snapshot isolation and read committed [5, 6]. However, the picture is not so rosy for *serializable* protocols.

Today's state-of-the-art serializable multi-version concurrency control protocols suffer from two major shortcomings:

- Serializable protocols cannot effectively exploit multi-versioning to reduce read-write conflicts among transactions. In fact, several published serializable multi-version protocols offer the same logical concurrency as single-version protocols [14, 16]. For example, Microsoft's Hekaton [14] uses a protocol that is nearly identical to that of Silo, a recently proposed single-version optimistic system [21]. While their inability to reduce read-write conflicts is not a problem in and of itself, multi-version databases suffer from version maintenance overheads that single-verion databases do not. For instance, multi-versioning increases the cache-footprint of transactions, since a record update requires memory corresponding to the prior version *and* the new version of a record to be brought into cache. In contrast, a single-version system writes the same memory words that it reads (in order to perform an update). The lack of additional logical concurrency, and higher overheads per transaction render serializable multi-version systems uncompetitive with single-version systems.
- Serializable multi-version concurrency control protocols employ unscalable synchronization (anti-)patterns. For instance, despite overwhelming evidence that suggests frequent synchronization on contended memory words is a death knell for scalability [8, 21], several multi-version concurrency control protocols continue to use shared global-counters to obtain transaction timestamps [14, 16]. Furthermore, even protocols that do not utilize monotonically increasing timestamps, such as timestamp ordering [7, 15], associate shared meta-data with database records. Since this meta-data is shared among multiple threads, the database must use latches or other forms of synchronization to read or update it. This synchronization in turn limits performance under contention [12].

In order to address these issues, we designed a new multi-version concurrency control protocol, BOHM [9]. BOHM's design is influenced by the following insight: the cost and complexity of obtaining serializable orderings of transactions can be eliminated by decoupling concurrency control from transaction execution. BOHM processes transactions as follows. First, it totally orders transactions as they enter the system. Second, it relaxes this total order into a partial order based on the conflicts among transactions; the total order determines the serializable order of transactions, and the resulting partial order is equivalent to this serializable order, but relaxed in order to account for transactions that do not conflict. Third, it executes transactions according to the pre-determined partial order. The first and second steps correspond to concurrency control, while the third step corresponds to transaction execution.

The biggest difference between BOHM and conventional serializable multi-version protocols is that conventional protocols determine a serializable order *as transactions execute*. These conventional protocols use techniques such as locking, validation, or timestamping [7, 14, 15] to appropriately constrain the execution of transactions on-the-fly. In contrast, BOHM determines a serializable order *prior* to transaction execution (as in the first and second steps above).

While BOHM addresses several problems associated with conventional multi-version concurrency control, it does so by changing a transaction's execution model in two important ways. First, BOHM requires that a transaction's entire logic be submitted to the database prior to its execution in the form of stored-procedures. Second, BOHM requires that a transaction's write-set be declared or deduced prior to its execution. While these assumptions limit BOHM's general applicability, they do apply to a non-trivially-sized class of applications and workloads. For instance, several researchers have recently made similar assumptions [17, 18, 20], and have vindicated these assumptions in the real world [4, 19].

BOHM's design has three important implications on its performance and scalability:

- BOHM makes significantly stronger logical concurrency guarantees than conventional serializable multi-version concurrency control protocols. BOHM guarantees that reads *never* block writes,

while simultaneously ensuring that reads never abort. BOHM relies on the pre-determined partial order of transactions to determine which version of a record a transaction must be read; a transaction always reads the version of a record produced by the preceding writer. Note, however, that while reads never abort, they may have to wait for the appropriate version of a record to be written. BOHM also guarantees that write-write conflicts never lead to aborts; conflicting writes are ordered according to the pre-determined total order. In contrast, several multi-version databases abort transactions due to write-write conflicts [14,16]. Furthermore, write-write conflict induced aborts are not restricted to serializable protocols, even weaker isolation level implementations (such as Snapshot Isolation) abort transactions in the presence of write-write conflicts [14].

- BOHM eliminates the need for shared concurrency control metadata. Concurrency control is managed by a dedicated set of threads, and each thread is responsible for concurrency control on a mutually exclusive partition of the database. Concurrency control threads do not share data, and can therefore process transactions independently. BOHM therefore avoids the need for shared-memory synchronization. Furthermore, BOHM's concurrency control threads only determine the order in which transactions must execute; transactions's actual execution occurs in a shared-everything environment. By decoupling shared-nothing concurrency control from shared-everything transaction execution, BOHM avoids the deleterious performance consequences of both shared-nothing and shared-everything systems (multi-partition transactions, and shared-memory synchronization, respectively).
- BOHM uses a lightweight adaptive scheduling mechanism to execute transactions. The vast majority of database systems use the thread-to-transaction model of execution [11]. By binding transactions to a specific thread, database threads are often subject to excessive context-switching overheads, especially under contention. BOHM does not use the conventional thread-to-transaction model of execution. Instead, BOHM models a schedule of transactions as an explicit dependency graph, database threads process transactions by simply crawling this graph of transactions. We have found that dependency graph-based scheduling completely eliminates the cost of context-switching database threads, and automatically limits the number of active threads based on the degree of contention in a workload.

In this talk, I plan to discuss the shortcomings of state-of-the-art multi-version concurrency control protocols, and how BOHM's design addresses several of these limitations. I also plan to discuss the insights gained in designing and prototyping BOHM, which more generally applicable beyond serializable multi-version systems.

1. REFERENCES

[1] Hekaton Breaks Through. http://research.microsoft.com/en-us/news/features/hekaton-122012.aspx.

[2] MemSQL. http://memsql.com/.

[3] SAP HANA. http://hana.sap.com/.

[4] Voltdb. http://voltdb.com.

[5] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *ICDE*, 2000.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.

[7] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[8] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Linux OLS*, 2012.

[9] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[10] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, 1981.

[11] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. *Architecture of a database system*. Now Publishers, 2007.

[12] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *VLDBJ*, 23(1), 2014.

[13] A. Kemper and T. Neumann. Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[14] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4), 2011.

[15] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. *PVLDB*, 8(13), 2015.

[16] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, 2015.

[17] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2), 2010.

[18] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, 2007.

[19] A. Thomson and D. J. Abadi. Calvinfs: Consistent WAN replication and scalable metadata management for distributed file systems. In *FAST*, 2015.

[20] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[21] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *SOSP*, 2013.