Low-Overhead Concurrency Control Using State-Based Transaction Scheduling

Andrew Crotty
Boston College

Abstract

NewSQL RDBMSs specifically target OLTP applications, attempting to combine the high performance of NoSQL systems with the ACID guarantees of traditional architectures. Of late, these RDBMSs have eschewed the standard design practices of their predecessors in favor of more streamlined and specialized techniques. One such innovation in vogue at present is the serial execution model, in which data is divided into optimally sized partitions that each own a single worker thread for processing. Research suggests that, in current main-memory systems, this approach actually outperforms customary concurrency mechanisms by providing maximal CPU utilization at minimal cost. However, the serial execution model makes stringent assumptions regarding the partitionability of datasets, and many common workloads do not possess such characteristics.

In order to better address these workloads, we present a low-overhead concurrency control model that does not rely on strict expectations about partitionability. Our design uses a novel state-based transaction scheduler to efficiently organize the concurrent execution of non-conflicting operations. We demonstrate that our model achieves optimal CPU utilization for the aforementioned workloads and outperforms other concurrency control strategies.

1 Introduction

The so-called NewSQL movement has ushered in a class of relational database management systems (RDBMSs) that seeks to provide the scalability and availability offered by NoSQL's looser consistency model while still preserving all of the ACID guarantees of more conventional architectures [11]. These cutting-edge RDBMSs are specifically geared towards online transaction processing (OLTP) workloads consisting of repetetive, short-lived transactions that touch only a small subset of data [9]. With the constantly decreasing price of increasingly large main-memory storage devices, many OLTP applications have the unique benefit of a persistently memory-resident active dataset. Consequently, NewSQL systems make radical design choices intended specifically to leverage these transaction properties and data characteristics.

In particular, a multitude of research has suggested that traditional concurrency control mechanisms present a major bottleneck for main-memory RDBMSs [1, 5]. Performance benchmarking conducted on the Shore DBMS, for instance, identified the locking and latching associated with concurrency control as consuming around 30% of transaction execution time [6]. In order to avoid this significant timesink, a number of systems have proposed a serial execution model, wherein each data partition possesses a single thread used for processing transactions in a sequential fashion. By intelligently partitioning the data into optimal segments, this scheme allows for highly efficient parallelism, because transactions operating on distinct partitions do not require any coordination. Furthermore, with all of the data residing in main-memory and transactions based solely upon stored procedures, the system experiences no disk or user stalls, permitting short transactions to execute uninterrupted and thus offering near-perfect CPU utilization.

H-Store emerged as an early pioneer of the serial execution model, vastly outperforming legacy OLTP systems [7, 12]. The architecture eventually grew to incorportate Horticulture, an automatic database design tool, in order to further optimize transaction execution and yield additional performance improvements [9]. Horticulture strives to lay out the database in such a way as to maximize the number of low-overhead single-partition transactions while minimizing the number of multi-partition transactions, which necessitate expensive distributed concurrency control mechanisms. Provided that these multi-partition transactions occur infrequently, the required costs remain manageable; however, as their frequency increases, the associated overhead quickly becomes prohibitive. In order to make use of the CPU while waiting for a multi-partition transaction to complete, H-Store permits the speculative execution of local, single-partition transactions [10]. This scheme minimizes blocking and mitigates some of the requisite costs, increasing throughput as long as the number of multi-partition transactions remains relatively low.

However, many common workloads possess traits that make them resistent to effective partitioning, therefore necessitating a high percentage of multi-partition transactions. The additional overhead incurred by distributed concurrency control protocols severely degrades performance and transaction throughput, suggesting that the overall efficacy of the serial execution model requires highly specific dataset partitionability characteristics. In particular, this model fails to adequately handle datasets that are 1) nonpartitionable, 2) imperfectly partitionable, 3) maximally partitioned, or 4) subject to rapidly transitioning hotspots.

We present a low-overhead concurrency control model that specifically caters to workloads with these nontraditional partitionability characteristics. Unlike the data-oriented partitioning methods employed by the serial execution scheme, the state-based approach that we propose focuses instead on the operations that a transaction seeks to perform. Thus, by examining and analyzing transactional intention, our model enables efficient, conflict-free concurrent execution while introducing little additional overhead when compared to a serial model.

2 Motivating Example

Social networking applications represent the canonical example of a nontraditionally partitionable workload. The many-to-many relationships between users render common partitioning techniques, including range, round-robin, and hash-based partitioning, ineffective, as multi-partition transactions become increasingly necessary with the growth of user interconnectedness. Other work has suggested using graph partitioning and machine learning algorithms to minimize these multi-partition transactions; this approach may apply to applications where users form distinct, tight-knit communities, but does not effectively manage frameworks typified by more diverse and wider-reaching user connections. The popular microblogging website Twitter, for instance, allows users to connect based upon common interests, resulting in an interwoven mesh of relationships.

In addition to its nonpartitionable dataset, the Twitter platform in particular represents a workload eminently well suited to our concurrency scheme. Specifically, the vast majority of client transactions consist exclusively of either viewing or posting Tweets, thus constituting a workload comprised primarily of read-append operations. Since our model allows for non-blocking read and insert operations, the Twitter workload will realize even greater performance gains.

3 Concurrency Model

Our concurrency model is comprised of three primary components: a multiversioning scheme, a transaction scheduler, and consistency mechanisms. We address each of these components individually.

3.1 Multiversioning Scheme

The multiversioning scheme uses timestamps to preserve *historical versions* of data. Historical versions are the overwritten attribute values in the case of a modify operation, or the tuple as a whole in the case of a delete operation. An historical version is always *stable*, meaning that the data is committed and guaranteed not to change. The *current version* encompasses the as yet uncommitted changes to the latest historical version, and *prospective versions* correspond to proposed historical versions that will become available as of some specified future time. Current and prospective versions are considered *volatile*, meaning that the data is uncommitted and subject to change, until the transactions explicitly commit their modifications. A committed current version becomes the latest historical version, whereas a committed prospective version remains as a pending historical version.

Within this framework, every write transaction is assigned a timestamp from a shared strictly monotonically increasing counter, starting with a value of 0. All tuples possess an insert timestamp i, which is set to the timestamp of the inserting transaction, and a delete timestamp d, which is set initially to a value of -1 to signify current existence. Similarly, all attribute values possess a modify timestamp m, which is also set to the timestamp of the inserting transaction. Each attribute stores the historical versions of its values as a *version chain.* A version chain is a singly linked list of values, with the head of the list containing the newest version and the tail containing the oldest. In order to modify a tuple, a transaction provides a set V' of mappings from attributes to new values. For each attribute a in V', the new value v' is prepended to that attribute's version chain with an m set to the timestamp of the modifying transaction. In order to delete a tuple, d is simply set to the timestamp of the deleting transaction, and the tuple is left in place as part of an historical version of the dataset. We summarize the operation of write transactions in Algorithm 1.

Algorithm 1 Write pseudocode.

```
procedure INSERT(tuple $r$, values $V'$, transaction $t$)
    $r.i \leftarrow t$
    $r.d \leftarrow -1$
    for $a \mapsto v' \in V'$ do
        prepend $v'$ to $r.a.VersionChain$
        $v'.m \leftarrow t$
    end for
end procedure
procedure MODIFY(tuple $r$, values $V'$, transaction $t$)
    for $a \mapsto v' \in V$ do
        prepend $v'$ to $r.a.VersionChain$
        $v'.m \leftarrow t$
    end for
end procedure
procedure DELETE(tuple $r$, transaction $t$)
    $r.d \leftarrow t$
end procedure
```

Read transactions are assigned the timestamp of the most recently *completed writer* (MRCW), thereby providing them with the latest historical version of the data. A completed writer is a write transaction that has either committed or aborted. Read transactions utilize the tuple and attribute value timestamps set by write transactions in order to determine which historical versions are appropriate to read. First, a transaction t must decide whether to read a particular tuple. In the cases where $t \geq i$, and $d < 0$ or $t < d$, then t may read the tuple. Otherwise, the tuple was either inserted later or deleted earlier, respectively, and should not be read by t. Next, t must decide which versions of the tuple's attribute values to read. For each attribute, t iterates through that attribute's version chain, stopping when it finds a value such that $t \geq m$. We summarize the operation of read transactions in Algorithm 2.

Algorithm 2 Read pseudocode.

```
procedure READ(tuple $r$, transaction $t$)
    if $t \geq r.i$ and $(r.d < 0$ or $t < r.d)$ then
        set of values $V$
        for $a \in r.Attributes$ do
            for $v \in a.VersionChain$ do
                if $t \geq v.m$ then
                    $V[a] \leftarrow v$
                    break;
                end if
            end for
        end for
        read $r$ as $V$
    else
        do not read $r$
    end if
end procedure
```

3.2 Transaction Scheduler

Given this multiversioning scheme, the transaction scheduler utilizes a state machine, pictured in Figure 1, to generate conflict-serializable schedules with low overhead. Each state expresses the current *operational mode* of a transaction. An operational mode describes the permissions granted to a transaction, namely *read-only*, *insert-only*, or *read-write*. Read-only transactions can perform read operations on some historical version of the data. Insert-only operations can perform insert operations on some prospective version of the data. Read-write transactions can perform read, insert, modify, or delete operations on the current version of the data. In order to model these operational modes, the state machine requires the following five states: Read, Write, Block, Commit, and Abort.

Read New read-only transactions enter the Read state with the timestamp of the MRCW, thus guaranteeing that they will always see a consistent snapshot of the data based upon the latest historical version. Read-only transactions can then perform non-blocking read operations without worrying about conflicting concurrent write operations. To transition from the Read to the

Write state, a transaction t must request a new timestamp t' that satisfies the requirement $t' = MRCW + 1$. If t' does not satisfy this requirement, then t must abort.

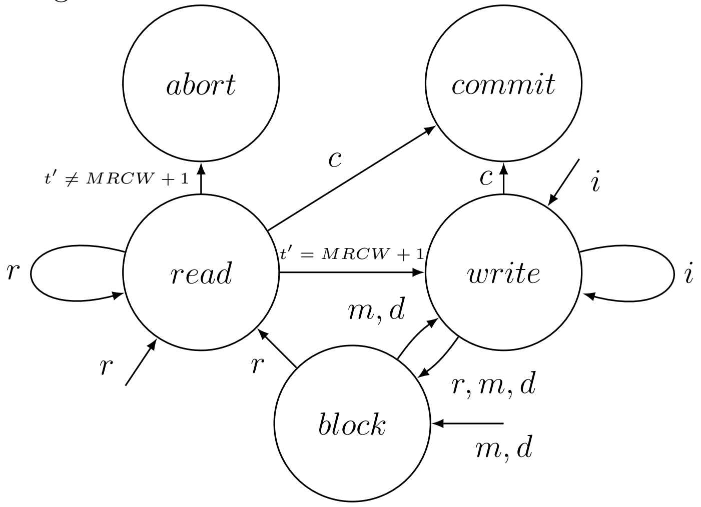
Write New insert-only transactions enter the Write state with a timestamp drawn from a counter common to all write transactions that provides strictly monotonically increasing values. Since insert-only operations do not interact with (i.e. read, modify, or delete) any data, they can execute in a non-blocking fashion such that the tuples they insert belong to some prospective version not visible to older concurrent transactions.

Block New read-write transactions enter the Block state with a timestamp drawn from the same counter used by insert-only transactions. In order to avoid conflicts, they must execute serially, with a queue enforcing this serial behavior. Upon reaching the head of the queue, a transaction is considered the active writer (AW) and may transition to the Write state to begin executing insert, modify, or delete operations. Additionally, the AW may transition freely between the Read and Write states, also without introducing conflicts, because it will always satisfy the requirement $t = MRCW + 1$.

Commit A transaction in the Read or Write state may transition freely to the Commit state by issuing a commit command. A committing write transaction t will be identified as the new MRCW only if it satisfies the requirement $t = MRCW + 1$; otherwise, it will be placed in a priority queue until it satisfies this requirement.

Abort A transaction t can only reach the Abort state in three ways: 1) by issuing an explicit abort command while in the Read, Write, or Block state; 2) by attempting to transition from the Read to the Write state with a new timestamp t' that does not satisfy the requirement $t' = MRCW + 1$; or 3) by violating an integrity constraint. An aborting write transaction t will be identified as the new MRCW only if it satisfies the requirement $t = MRCW + 1$; otherwise, it will be placed in a priority queue until it satisfies this requirement.

Figure 1: The state-based transaction scheduler.



3.3 Consistency Mechanisms

While the multiversioning scheme ensures that read transactions receive a consistent snapshot of the data and the transaction scheduler prevents write transactions from introducing conflicts, concurrent write transactions can still leave the database in an invalid state by collectively violating integrity constraints. A combination of consistency mechanisms serves to avert these anomolies.

Serial Writers The simplest method of enforcing integrity constraints is to mandate that all write transactions execute using only a read-write operational mode, effectively imposing a serial execution order. Then, transactions would always perform integrity constraint checks sequentially against the current version of the database, eliminating the possibility that concurrent changes could combine to violate a constraint. Despite severely reducing the possibility for parallelism, this option may suffice for read-heavy workloads, since only write transactions need to execute serially.

Deferred Commits Another simple option is to allow write transactions to execute in parallel but defer their commits until all older write transactions have already committed. Similar to the serial writer scheme, this approach still ensures that integrity constraint checks occur in sequential order using the database's current version but also permits the concurrent execution of transactional work, substantially improving throughput for workloads with a high volume of insert-only transactions. The risk arises that some younger write transactions may block while waiting for their turn to commit, only to discover that they actually need to abort because they have violated a constraint, but abort rates would remain relatively low in workloads where transactions frequently interact with disjoint data subsets.

Commit Queues The serial writers and deferred commit methods both enforce a strict chronological commit ordering such that consistency checks are always performed on the current version of the data. However, transactions need not commit in chronological order provided that their combined changes do not collectively violate any particular integrity constraint. By mandating that all transactions pass through a *commit queue*, transactions can perform their integrity checks out of order as long as they are performed serially.

Entity Sequences Entity integrity states that each tuple in a relation must have a primary key that is both non-null and unique. While null values can be screened easily by checking an operation's parameters, uniqueness must be resolved in a thread-safe manner in order to prevent concurrent transactions from inserting duplicate primary key values. One solution is an atomic *putIfAbsent* operation for primary indexes, allowing the insertion if the value is not already present but denying it otherwise. However, in applications where surrogate primary keys are feasible, or even preferred, we advocate the use of an even less expensive *entity sequence*, a shared counter that implements an atomic *getAndIncrement* operation, thereby guaranteeing the use of unique values. In situations where sequential primary keys are undesirable, such as the assignment of SSNs, values can be drawn randomly without replacement from a pre-allocated pool using an atomic *getAndRemove* operation.

Reference Counts Referential integrity stipulates that every value of the specified attribute must exist as a primary key in some relation. Each primary key can store an atomic counter to keep track of its references, preventing removal of keys that are currently in use.

4 Correctness

We now prove that all schedules generated by our algorithm are conflict-serializable. We proceed by first enumerating the circumstances that would violate conflict-serializability and then showing that these circumstances can never arise. Finally, we use precedence graphs to demonstrate the universal conflict-serializability of all schedules that our model produces.

4.1 Conflicts

Conflicts consist of pairs of noncommutative operations. Namely, the three conflict cases that violate conflict-serializability are read-write conflicts, write-read conflicts, and write-write conflicts.

Read-Write Conflicts Read-write conflicts involve a transaction over-writing a value that was previously read by a different transaction, causing an unrepeatable read. Consider two transactions $t1$ and $t2$ operating on some datum x with an initial value 'A':

1. $t1$ reads the value 'A' from x.
2. $t2$ writes the new value 'B' to x, overwriting the old value 'A', and then commits.
3. $t1$ reads the new value 'B' from x; $t1$ cannot repeat its original read, resulting in a conflict.

Since our multiversioning scheme preserves all historical versions of x, this conflict can never occur. Rather than overwriting the value of x, $t2$ would create a new version x' to store the new value 'B':

1. $t1$ reads the value 'A' from x.
2. $t2$ writes the new value 'B' to x', preserving the old value 'A', and then commits.
3. $t1$ ignores the new value 'B' from x' and reads the old value 'A' from x; $t1$ can now repeat its read, avoiding the conflict.

The phantom read problem is a specific example of a read-write conflict. Imagine now that x represents a relation. Any transaction that changes the cardinality of x, i.e. by performing an insert or delete operation, effectively creates a new version of the relation x'. Thus, an older reader can avoid phantoms by continuing to reference the historical version x.

Write-Read Conflicts Write-read conflicts involve a transaction reading a value written by a different, uncommitted transaction, causing a dirty read. Consider two transactions $t1$ and $t2$ operating on some datum x with an initial value 'A':

1. $t1$ writes the new value 'B' to x, overwriting the old value 'A'.
2. $t2$ reads the new value 'B' from x and then commits.
3. $t1$ aborts, resetting the value of x to the old value 'A'; the new value 'B' read by $t2$ is now invalid, resulting in a conflict.

Since our multiversioning scheme assigns read transactions the timestamp of the MRCW, this conflict can never occur. Rather than reading from the new uncommitted version x', $t2$ would read from the old committed version x:

1. $t1$ writes the new value 'B' to x', preserving the old value 'A'.
2. $t2$ ignores the new value 'B' from x' and reads the old value 'A' from x, and then commits.
3. $t1$ aborts, removing the new version x'; the old value 'A' read by $t2$ is still valid, avoiding the conflict.

Write-Write Conflicts Write-write conflicts involve a transaction overwriting a value written by a different, uncomitted transaction, causing a lost update. Consider two transactions $t1$ and $t2$ operating on some data x and y both with an initial value 'A':

1. $t1$ writes the new value 'B' to x, overwriting the old value 'A'.
2. $t2$ writes the new value 'C' to y, overwriting the old value 'A'.

3. $t1$ writes the new value 'B' to y, overwriting the old value 'C', and then commits.

4. $t2$ writes the new value 'C' to x, overwriting the old value 'B', and then commits; x and y are left in an imposible state with the values 'C' from $t2$ and 'B' from $t1$, respectively, resulting in a conflict.

Despite our multiversioning scheme, this interleaving would still result in an impossible state, with the newest version of x having a value of 'C' from $t2$ and the newest version of y having a value of 'B' from $t1$. However, since our transaction scheduler restricts the interleaving of conflicting transactions, this conflict can never occur. Rather than interleaving $t1$ and $t2$, the scheduler would force serial execution:

1. $t1$ writes the new value 'B' to x, overwriting the old value 'A'.

2. $t1$ writes the new value 'B' to y, overwriting the old value 'A', and then commits.

3. $t2$ writes the new value 'C' to y, overwriting the old value 'B'.

4. $t2$ writes the new value 'C' to x, overwriting the old value 'B', and then commits; x and y are left in a correct state both with the value 'C' from $t2$, avoiding the conflict.

4.2 Precedence Graphs

The conflict-serializability of a particular schedule can be tested by constructing a precedence graph representing all of the conflicting operations. If the graph contains no cycles, then the schedule is conflict-serializable. The rules for constructing a precedence graph from some schedule s are as follows:

1. For each transaction t specified in s, create a new node in the graph.

2. *Read-Write:* For each write operation on some datum x by transaction t_j that occurs after a read operation on x by transaction t_i, add a new edge $t_i \rightarrow t_j$ to the graph.

3. *Write-Read:* For each read operation on some datum x by transaction t_j that occurs after a write operation on x by transaction t_i, add a new edge $t_i \rightarrow t_j$ to the graph.

4. *Write-Write:* For each write operation on some datum x by transaction t_j that occurs after a write operation on x by transaction t_i, add a new edge $t_i \rightarrow t_j$ to the graph.

Only rules 2-4 specify cases when a new edge should be added to the precedence graph. Since we have already shown that each of these cases can never occur, all schedules produced by the transaction scheduler will result in the construction of edgeless graphs. All edgeless graphs are necessarily acyclic graphs, and thus all schedules produced by the transaction scheduler must be conflict-serializable.

5 Discussion

The concept of data multiversioning as a means of concurrency control has been studied extensively over the past several decades [2, 3, 8, 4]. Unlike update-in-place models, data multiversioning introduces the unique problem of garbage accumulation, such that useless historical versions begin to clutter the storage engine. Over time, this garbage can grow to consume a substantial amount of space, which represents a particular concern for main-memory systems that have only a limited storage capacity. Additionally, garbage can negatively impact performance, since transactions may waste time by unnecessarily examining extraneous historical versions. While these complications do not manifest in read-append workloads similar to Twitter, the applicability of our scheme to more general application models predicates an efficient garbage collection algorithm. We leave the evaluation of garbage collection techniques to future work.

In terms of database consistency, we presume that the overwhelming majority of transactions are *well-behaved*, meaning that they will not intentionally attempt to violate integrity constraints. Under typical circumstances, individuals will not knowingly overdraw their checking accounts, and customers will not place orders that exceed the current inventory. The primary concern for integrity constraint violation comes from concurrent write transactions whose changes are valid when considered individually but cause problems when considered together. If constraint checks occur simultaneously, then transactions may miss concurrent changes introduced by other transactions, providing them with a false sense of the current state of the database. Thus, some serialization point must exist to ensure that constraint checks occur in a sequential fashion, such that no concurrent changes are ever missed.

6 Future Work

As previously mentioned, garbage collection represents the most pressing area for immediate future work. Without an efficient method for reclaiming storage space occupied by useless historical versions, our model cannot be applied to more generalized workloads that incorporate modify or delete operations. Relevant data structures, then, become especially important, and care needs to be taken not to introduce any overhead in the form of locks or latches. The design of version chains represents a related concern, since the associated storage overhead negatively impacts tuples that occupy only a small amount of space. Relations with these attribute properties would benefit from the implementation of modify operations that create completely new versions of entire tuples rather than individual attribute values, possibly achieved by functionally combining a delete and insert operation. We propose that static analysis of tuple size be used to automatically select the optimal multiversioning configuration for relations on an individual basis.

The transaction scheduler also warrants further investigation. Presently, read-only and insert-only transactions execute in a nonblocking fashion, while read-write transactions must execute serially. This pessimistic strategy ensures that no conflicts can occur but also unnecessarily prevents the parallel execution of read-write transactions that otherwise touch completely disjoint subsets of data. More optimistic strategies, such as lightweight dynamic conflict detection, would offer greater concurrency and, accordingly, greater throughput at the price of an increased risk for aborts. A different possibility is to partition a single serial execution queue into several subqueues by hashing the parameters from the stored procedures, therefore permitting greater concurrency with virtually no added overhead. The same concept can be applied to commit queues for checking integrity constraints.

Finally, since our model offers the greatest performance gains to workloads comprised primarily of read-append operations, another area of interest is the development of a tool to automatically and transparently convert traditional application designs to more advantageous implementations. Then, developers could migrate existing applications to our system with no redesign prerequisites, and create new applications without explicitly having to tailor the logic to reap any benefits.

7 Conclusion

Overall, we maintain that the current state of the art does not sufficiently address the unique needs of workloads with nontraditional partitionability characteristics. These workloads require a specialized class of system specifically tailored to their operational requirements. Our model represents a stark divergence from the customary data-centric approach and instead moves toward an operation-centric paradigm, focusing solely on the actions that transactions seek to perform. With this reliance on transactional intention, our model can match, and perhaps even surpass, the performance of current NewSQL systems without any associated workload partitionability requirements.

References

[1] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, S. Haldar, S. Joshi, A. Khivesera, H. Korth, P. Mcilroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, S. Seshadri, A. Silberschatz, S. Sudarshan, M. Wilder, and C. Wei. DataBlitz Storage Manager: Main-Memory Database Performance for Critical Applications. In *SIGMOD*, pages 519–520, 1999.

[2] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. In *CSUR*, pages 185–221, 1981.

[3] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control—Theory and Algorithms. In *TODS*, pages 465–483, 1983.

[4] M. J. Carey and W. A. Muhanna. The Performance of Multiversion Concurrency Control Algorithms. In *TOCS*, pages 338–378, 1986.

[5] V. Gottemukkala and T. J. Lehman. Locking and Latching in a Memory-Resident Database System. In *VLDB*, pages 533–544, 1992.

[6] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008.

[7] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB*, pages 1496–1499, 2008.

[8] C. H. Papadimitriou and P. C. Kanellakis. On Concurrency Control by Multiple Versions. In *TODS*, pages 89–99, 1984.

[9] A. Pavlo, C. Curino, and S. Zdonik. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*, pages 61–72, 2012.

[10] A. Pavlo, E. P. C. Jones, and S. Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. In *VLDB*, pages 85–96, 2011.

[11] M. Stonebraker. New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps, 2011.

[12] M. Stonebraker, S. 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*, pages 1150–1160, 2007.