Bringing Modular Concurrency Control to the Next Level

Chunzhi Su[†] Natacha Crooks[†] Cong Ding[‡]
Lorenzo Alvisi[†‡] Chao Xie[†*]
[†]The University of Texas at Austin [‡]Cornell University

ABSTRACT

This paper presents Tebaldi, a distributed key-value store that explores new ways to harness the performance opportunity of combining different specialized concurrency control mechanisms (CCs) within the same database. Tebaldi partitions conflicts at a fine granularity and matches them to specialized CCs within a hierarchical framework that is modular, extensible, and able to support a wide variety of concurrency control techniques, from single-version to multiversion and from lock-based to timestamp-based. When running the TPC-C benchmark, Tebaldi yields more than $20\times$ the throughput of the basic two-phase locking protocol, and over $3.7\times$ the throughput of Callas, a recent system that, like Tebaldi, aims to combine different CCs.

1. INTRODUCTION

This paper introduces Tebaldi, a transactional key-value store that takes significant steps towards harnessing the performance opportunities offered by a *federation* of optimized concurrency controls by allowing these mechanisms to be composed *hierarchically*.

The rationale behind federating CC mechanisms [17, 34, 42, 45, 56] is straightforward: any single CC technique is bound to make trade-offs or rely on assumptions that cause it to perform extremely well in some cases but poorly in others. For instance, pessimistic techniques such as two-phase locking [10] perform well in highly-contended workloads, but may lead to write transactions unnecessarily stalling read transactions, while multiversioned CC algorithms improve read performance, but introduce non-serializable behaviors that are difficult to detect [24, 35]. Since concurrent transactions interact in fundamentally different ways across these scenarios, these trade-offs appear unavoidable. A promising approach is instead to *federate* different CC mechanisms within the same database, applying each given CC only to the transactions or workloads where it shines, while maintaining the overall correctness of the database.

In practice, however, realizing the performance potential of a federated solution is challenging. Such a solution should be modular: it should allow developers to reason about the correctness of any given CC mechanism in isolation, without being aware of other coexisting CC mechanisms. The solution should also be general: it should be capable of federating a large set of diverse techniques—optimistic and pessimistic, single-version as well as multiversion.

Prior approaches go some way towards achieving these goals as they enable different concurrency controls to execute on disjoint subsets of either data [42, 51], or transactions [13, 56]. However, some are restricted to specific CC combinations [13, 17, 34, 45, 51], while others, though more general, assume that the database/application can be partitioned such that conflicts across partitions are rare and inconsequential to the end-to-end performance of the system [42, 56]. As a result, most solutions simply handle every cross-partition conflict using a single, undifferentiated mechanism.

We find that the assumption behind this conclusion is flawed: cross-partition conflicts can in fact throttle the performance benefits of federation (§2) as a perfect partitioning of conflicts is, in general, unfeasible. In practice, there is often an inescapable tension between minimizing cross-partition conflicts and supporting aggressive CC mechanisms within each partition.

Tebaldi, the new transactional key-value store that this paper introduces, seeks to resolve this tension with a simple, but powerful, insight: the mechanism by which different concurrency controls are federated should *itself* be a federation. This flexibility, applied hierarchically at the level of cross-partition CC mechanisms, is key to realizing the performance potential of federation. Tebaldi's starting point is Modular Concurrency Control (MCC) [56], perhaps the most recent expression of the federated-CC approach. MCC proposes to *partition transactions* in groups, giving each group the flexibility to run its own private concurrency control mechanism. Charged with regulating concurrency only for the transactions within their own groups, these mechanisms can be very aggressive and yield more concurrency while still upholding safety. MCC imposes no restrictions on the kind of transactions or CCs that it can handle, and emphasizes modularity: as long as any given isolation property holds within each group, MCC guarantees that it will also hold among transactions in different groups. The question at the core of this paper is then simply: how should this guarantee be enforced?

Callas [56], the distributed database that represents the current embodiment of the MCC vision, offers a conservative answer: an inflexible mechanism based on two-phase locking. Tebaldi [1] aims

[*]Chao Xie is currently working at Google.

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064031

[1]Renata Tebaldi was Maria Callas's main rival on the opera stage.

to take MCC to a new level—both figuratively and, indeed, literally. Instead of handling cross-group conflicts through a single mechanism, Tebaldi regulates them by applying MCC recursively, adding additional levels to its tree-like structure of federated CC mechanisms. Refining the management of cross-group conflicts in this way increases flexibility in how conflicts are handled, which, in turn, improves performance. Tebaldi can, for example, combine the benefits of multiversioning [9, 12] with aggressive single version techniques such as runtime pipelining [56] at the cross-group layer.

Realizing this vision in Tebaldi presents two main technical hurdles. First, directly applying CCs hierarchically does not guarantee serializability (§4). We derive a sufficient condition—*consistent ordering*—that CCs must enforce to ensure correctness, and highlight how this property can be achieved in practice. Second, federating different CC mechanisms requires seamlessly managing the different expectations upon which their correctness depends (in terms of protocol, storage, failure recovery, etc.): Tebaldi allows CCs to independently implement the execution logic (including maintaining the necessary metadata) for making ordering decisions, but provides a general framework for composing the CCs' execution hierarchically and determining the appropriate version of data to read or write.

We show that Tebaldi's hierarchical MCC yields significant performance gains: running the TPC-C benchmark under serializability shows that Tebaldi yields more than $20\times$ throughput improvement over the basic two-phase locking protocol, and a $3.7\times$ throughput increase over Callas [56], the state-of-the-art federated system.

In summary, this paper makes the following contributions:

- It introduces a hierarchical approach to federating concurrency controls that allows conflicts to be handled more efficiently, while preserving modularity.
- It identifies a condition for the correct composition of concurrency controls within Tebaldi's hierarchy and shows that several existing concurrency controls can be modified to enforce it.
- It presents the design and evaluation of Tebaldi, a transactional key-value store that implements hierarchical MCC. Tebaldi currently supports four distinct concurrency controls and exposes an API that allows developers to add new CCs.

In the rest of this paper, we highlight the limitations of existing federating systems (§2) and introduce our hierarchical approach to concurrency control (§3), along with its theoretical foundations (§4). We then introduce the design (§5) and implementation (§7) of Tebaldi, the first transactional key-value store to federate CCs hierarchically for performance, highlighting its benefits and limitations. Finally, we quantify the benefits of our approach (§8), summarize related work (§9), and conclude (§10).

2. BACKGROUND AND MOTIVATION

Federating concurrency controls is an appealing solution for improving concurrency in distributed databases. In this section, we summarize its benefits (§2.1) along with the limitations of current approaches (§2.2).

2.1 The benefits of federation

The drive towards federating concurrency controls stems from a fundamental tension between a CC algorithm's generality and its

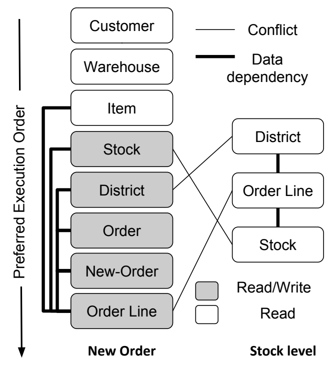


Figure 1: TPC-C `new order`/`stock level` transactions.

ability to aggressively handle conflicts. Mechanisms like two-phase locking [10] or optimistic concurrency control [27] make few assumptions about the application or the system, but are often overly pessimistic in dealing with conflicts. Most optimizations, however, rely on properties that are unlikely to hold in general, such as full knowledge of the read and write set [23, 47], lack of SC cycles [44], the ability to statically determine a total order of tables [56], or access locality [25, 29]. By federating these mechanisms, one can restrict the scope of these optimizations only to the portions of the application for which their assumptions hold, allowing for higher performance without sacrificing generality. Consider, for example, runtime pipelining (RP) [56] and deterministic concurrency control (DCC) [23, 47]. *Runtime pipelining* efficiently pipelines transactions by allowing operations to observe the result of uncommitted writes, maintaining correctness through a combination of static analysis and runtime techniques. RP is most effective when transactions generate few circular dependencies when accessing tables. As the number of transactions grows, however, such dependencies are increasingly likely. RP is, for instance, of limited use when applied to the full TPC-C (Figure 1), as there exists a circular dependency in the `new order` and `stock level` transactions between the `stock`, `order line`, and (the preferred execution order of) `district` tables. This cycle prevents RP from pipelining these operations; if its scope were instead restricted to regulating multiple concurrent instances of `new order`, RP could choose a finer grained pipeline, improving performance [56]. Similarly, *deterministic concurrency control* (DCC) [23,47] shines when the complete access set of transactions is known at begin time, as then DCC can pre-order transactions according to their data-access, removing the overhead of runtime conflict detection. Otherwise, DCC's benefit is limited by its need for expensive pre-transaction queries to construct the access set.

2.2 The limits of federation

Several prior systems [13, 17, 34, 42, 45, 51, 56] have tried to tap the performance benefits of federating CC mechanisms, variedly grouping conflicts along boundaries defined by the timing of conflicts [17, 34], the data they involve [42, 45, 51], or the code that triggers them [56]. Some are restricted to combining specific CCs [13, 17, 34, 45, 51]. Others are more general, but assume that the application/data can be cleanly partitioned such that coordination across these partitions is simple, with limited impact on performance [42, 56]. Their prescription has been to complement their partition-specific CC mechanisms with a single, catch-all mechanism suited to their partition strategy, e.g. two-phase locking [56].

Our findings, however, tell a different story. We find that combining aggressive in-partition optimizations with a single, conservative cross-partition mechanism exposes existing federated systems to a dilemma. On the one hand, in-partition mechanisms, to be effective, must handle a very specific, and hence narrow, subset of conflicts. On the other, pushing the remaining conflicts to the cross-partition layer can cripple its conservative CC, and in turn the performance of the whole system.

We highlight this dilemma in the Callas [56] system with our previous TPC-C example. Results are shown in Table 1. The first column shows the throughput of running `stock level` and `new order` in the same group (with RP in-group). As we discussed, this arrangement creates circular dependencies that void much of the potential benefit of RP. Perhaps surprisingly, placing these transactions in separate groups (using 2PL as cross-group CC) yields no benefit: throughput actually *drops* by an order of magnitude, as RP's preferred ordering of read-write accesses (§2.1) creates deadlocks at the cross-group level. Removing these deadlocks by reordering `new order`'s access to the `district` and `stock` table (third column) improves performance somewhat, but it is still only marginally better than placing the transactions in the same group. To get a sense for the role that the cross-group mechanism plays in determining these results, we show in the last row the throughput of a best-case scenario for partitioning. We place `stock level` and `new order` in separate groups, and artificially make them access different warehouses to eliminate all cross-group read-write conflicts: performance soars by almost an order of magnitude over that reported in the third column.

These results suggest three observations. First, *cross-group conflicts matter*. The performance bottleneck in our example is the 2PL cross-group CC mechanism, which cannot efficiently handle the read-write conflicts on the `district` table between the two transactions. Second, *the cross-group CC mechanism matters*. These read-write conflicts would have been better handled using a multiversion CC. Third, *no single cross-group CC mechanism can effectively address all conflicts.* This same multiversion CC would fare poorly under write-write cross-group conflicts.

Tebaldi's design responds to these observations starting from the simple premise that the key to performance is, once again, a federation of CC mechanisms, this time deployed to resolve *cross-group* conflicts. To realize this vision, we need to clear several technical hurdles. First, we need to determine the inner structure of the new CC federations that Tebaldi enables: our goal is to ensure that these new degrees of freedom do not come at the expense of modularity. Second, we need to identify the conditions that ensure the correctness of Tebaldi's more general federations. Finally, we need to develop the system-level support needed to bring this vision to fruition. We address these issues in the following sections.

3. HIERARCHICAL MCC

Tebaldi seeks to maximize flexibility in CC federations while preserving modularity. The benefits of *flexibility* are clear: finer control in determining the mapping between sets of conflicts and CC mechanisms enables greater concurrency and higher performance.

Same group	Separate - Deadl.	Separate - No Deadl.	Separate - No Conflict
3207 ± 1	158 ± 9	3598 ± 14	23834 ± 5

Table 1: Impact of grouping on throughput (txn/sec).

Unhinged flexibility can, however, come at the cost of *modularity*, defined as the ability of individual CCs to order conflicts independently while guaranteeing isolation. Consider, for instance, a set of three transactions, T_1, T_2, and T_3, and assume that, to maximize flexibility, conflicts between each pair of transactions are governed by a separate concurrency control. Suppose $CC_{1,2}$ orders T_2 after T_1, and that $CC_{2,3}$ orders T_3 after T_2. $CC_{1,3}$ is then left with only one correct choice (i.e., ordering T_3 after T_1) and it needs to become aware of that. In general, a CC mechanism may have to learn the ordering of all other CCs to guarantee correctness. Tebaldi balances these competing concerns by applying the theory of MCC [56] recursively, organizing CC mechanisms as nodes in a multi-level tree. Each node n is responsible for regulating conflicts among a set of transactions $\mathcal{T}$. In turn, n can choose to *delegate* some of this responsibility by assigning disjoint subsets of $\mathcal{T}$ to children nodes better suited to handle their conflicts, while only retaining responsibility for regulating conflicts across children.

Tebaldi's hierarchical refinement of MCC yields two key benefits. On the one hand, it enables greater flexibility: applying MCC recursively largely removes the concern that using aggressive in-group mechanisms may unnecessarily push conflicts to the conservative cross-group CC (§2.2). Instead, these conflicts are themselves further partitioned and mapped to efficient CCs—and so on recursively, until one reaches the root of the tree. On the other hand, Tebaldi's multi-level tree preserves a high-degree of modularity by retaining a key feature of MCC: the mapping from sets of conflicts to CCs is derived by subdividing a given set of transactions into mutually disjoint subsets. This makes it impossible for sibling nodes on Tebaldi's tree to make conflicting ordering decisions (they regulate disjoint portions of the serialization graph [3]). Instead, CCs need only communicate their ordering choices to (and, in turn, have their ordering choices constrained by) the CC mechanism of their parent node. As we will see in §4, this structural property is instrumental to guaranteeing that no two concurrency controls can make conflicting decisions on how to order a pair of transactions.

4. ENSURING CORRECTNESS

Tebaldi builds upon Adya's [3] general theory for expressing isolation. Adya associates with every execution a *direct serialization graph* whose nodes consist of committed transactions and whose edges mark the dependencies (read-write, write-write, or write-read) that exist between them. An execution satisfies a given isolation level if it disallows three properties: *aborted reads*, *intermediate reads*, and *circularity*. The first two conditions prevent a transaction T_1, respectively, from reading the value of an aborted transaction T_2 and from reading a version of an object *x* written by a transaction T_2 that T_2 subsequently overrides. The third condition is more complex: circularity prevents a cycle in the serialization graph. The specific edges that compose the cycle, however, are dependent on the particular isolation level.

In the spirit of modularity, existing MCC implementations [56] articulate these global requirements separately for in-group and cross-group mechanisms. In-group mechanisms must prevent circularity, aborted reads, and intermediate reads that solely involve the subset of transactions that they are responsible for. Cross-group CCs must prevent cycles, as well as aborted and intermediate reads, involving transactions from different groups.

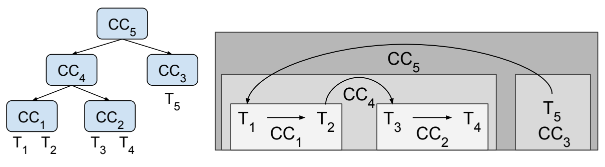
Tebaldi blurs the distinction between cross-group and in-group mechanisms: every concurrency control in the CC tree acts as an in-group mechanism in the eyes of its parent, and as a cross-group mechanism in the eyes of its children. Correctness can then simply be defined as follows:

Definition 1 *A CC tree is correct if every concurrency control in the tree prevents aborted reads, intermediate reads, and circularity for the committed transactions in its group.*

As it is, this definition says little about how CCs can be composed in a modular yet flexible fashion. Recall that the key to greater flexibility in Tebaldi is delegation: a parent concurrency control delegates conflicts that its child is better suited to handle. The parent's only responsibility is then to ensure that subsequent ordering decisions will be *consistent* with those of its children. Formally:

Consistent Ordering *For committed transactions T_1 and T_2, if a concurrency control at level $i-1$ in the tree (CC_{i-1}) creates a directed path from T_1 to T_2, then its parent CC at level i (CC_i) must never create a path from T_2 to T_1.*

In effect, each subtree in the CC tree defines a partial ordering on a subset of transactions. Outer concurrency controls merge different subtrees, extending the partial order and ensuring that the resulting transaction ordering does not violate circularity.



(a) A 3-layer CC tree. (b) Per-CC Ordering decisions.

Figure 2: Per-CC ordering responsibilities in the CC tree.

We illustrate this process in Figure 2. CC_1 orders transactions T_1 and T_2 and CC_2 transactions T_3 and T_4. CC_4's only responsibility is to ensure that T_1/T_2 and T_3/T_4 are ordered consistently—in this case, by ordering T_3 after T_2. Similarly, CC_5 orders T_5 before T_1.

Most off-the-shelf concurrency control mechanisms, however, are not quite as diligent as CC_4 and CC_5. In general, the parent CC can constrain the ordering decisions between transactions assigned to one of its children both directly (e.g., by timestamping transactions at their start time) and indirectly (by making cross-group ordering decisions that limit, in the service of correctness, a child CC's discretion in deciding how to order its own transactions). Ignoring these effects can lead to violations of consistent ordering.

For example, consider a non-leaf node CC_n, running 2PL. Assume CC_n has two children CCs: CC_1, in charge of ordering transactions T_1 and T_2, and CC_2, in charge of T_3. Suppose CC_1 serializes T_1 before T_2, and that T_2 commits first, releasing all its 2PL locks at CC_n. Nothing now prevents CC_n from letting T_3 read a data object written by T_2, thus forming a cross-group order $T_2 \rightarrow T_3$. Similarly, nothing forbids T_3 from writing some data object o and committing (thus releasing all locks at CC_n). Suppose T_1 now reads o, causing a write-read cross-group conflict at CC_n between T_3 and T_1. If CC_n orders T_1 after T_3, its two ordering decisions ($T_2 \rightarrow T_3$ and $T_3 \rightarrow T_1$) create a path from T_2 to T_1, violating consistent ordering [56].

Preserving consistent ordering We identify three general strategies by which nodes at adjacent levels of the CC tree can coordinate their ordering decisions and preserve consistent ordering. When a parent CC is faced with an ordering decision, it can either straightforwardly *adopt* the decision of one of its children CCs, take actions that *constrain* the future ordering decisions of its children, or *procrastinate*, leaving more time for its children to propose an ordering. The choice among these strategies largely depends on the timing of the decision and on the specifics of the parent's CC mechanism; indeed, mechanisms that take multiple steps to reach their final ordering decision may use a combination of these strategies.

When the parent CC's ordering decision comes *after* a child CC has decided how to order the transactions in its group, the simplest strategy to ensure consistent ordering is to fully embrace Tebaldi's emphasis on delegation and adopt the child's ordering decisions. This strategy proves particularly useful when the parent CC orders transactions at `commit` time, since by then it is likely to know its children's ordering decisions. Consider again the above example where CC_n, the parent CC, runs 2PL. CC_n, once it learns of its children's ordering decisions (e.g., $T_1 \rightarrow T_2$), must simply respect that ordering when committing transactions (so that the locks held by T_2 are only released after T_1 commits).

When instead the parent CC's ordering decision comes *before* the child has had time to decide, two strategies remain: constraining the child's decisions, or procrastinating until the child decides (and then adopting its decisions).

The constraints imposed by the parent CC can vary from subtle to overbearing. At the subtle end of the spectrum, how a parent resolves a cross-group conflict (for instance, by blocking conflicting transactions across groups in 2PL and RP) can often indirectly limit how the children CCs can serialize these transactions. At the other end of the spectrum, a parent CC could simply dictate the order of transactions to its children (for example, Timestamp Ordering [9] assigns each transaction a unique timestamp at begin time). This degree of micromanagement, of course, would run counter to Tebaldi's design, which leverages delegation as the key to greater performance. Nonetheless, such CCs can serve effectively as inner nodes in Tebaldi's hierarchy by selectively *procrastinating* ordering decisions until their children make theirs. For example, rather that labeling each transaction instance with a unique timestamp, the parent CC could assign the *same* timestamp to a batch of transactions from the same group. By waiving its chance to order these transactions, the parent would in effect delegate their ordering to the child CC responsible for that group, while still constraining the child by preventing it from ordering the transactions in batch $i+1$ before those in batch i.

In Section 6, we will discuss in detail how Tebaldi uses adoption, constraining, and procrastination to integrate several widely used CCs in its hierarchical architecture.

5. TEBALDI'S DESIGN

Having sketched out the correctness requirements of hierarchical MCC, we describe next how Tebaldi, our new transactional key-value

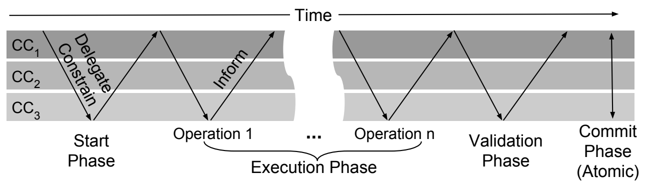


Figure 3: Execution Protocol.

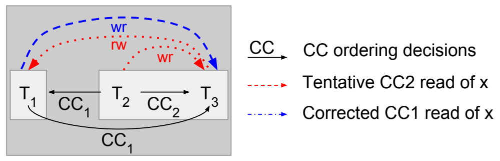


Figure 4: Read logic in execution phase.

store, enforces these requirements. Similarly to several distributed, disk-based, commercial systems [2, 4, 15], Tebaldi separates its concurrency control logic from its storage management and keeps metadata associated with CC protocols (like timestamps and version lists in snapshot isolation, and locks in 2PL) as transient state in the concurrency control module.

The *concurrency control module* coordinates how the diverse CC protocols in Tebaldi's hierarchy collectively determine the order of transactions. Tebaldi's framework for CC coordination leverages the observation that, despite their diversity, the steps that most CC protocols take in determining the ordering of a transaction T can be grouped into four distinct phases: a *start phase*, an *execution phase*, a *validation phase*, and a *commit phase*. Tebaldi executes each phase in two passes (Figure 3). The first pass, top-down, gives parent nodes the opportunity to constrain how their children's ordering decisions affect the ordering of T; the second pass, bottom-up, lets children inform their parent of T's current *dependency set*, i.e., the list of transactions in its group on which T depends. This structure gives Tebaldi its generality: Tebaldi can support a maximum of CC combinations by giving every concurrency control, in each phase, the opportunity to constrain or delegate to its child, while the child can in turn inform its parent as soon as dependencies become known. This generality does not come at the cost of modularity: the implementation of each concurrency control remains independent from that of its parents (or children) as they communicate only via well-defined communication channels. Further, Tebaldi is extensible: it provides a blueprint for adding a new CC to an existing CC hierarchy tree: all that is required is to identify and integrate the new CC's four phases in the tree.

The *storage module* stores and retrieves the appropriate version of a data object according to the CCs' ordering decisions. To support both single version and multiversion concurrency control protocols, this module is implemented as a multiversion storage that keeps all the committed and uncommitted writes on each object.

Naturally, there might be exceptions: some specialized concurrency controls may not fit well in Tebaldi's four-phase model or may expect a specific storage layout. Likewise, there may be inner CCs whose ordering decisions are not known at commit time. We discuss the limitations of our approach in §5.2.

5.1 Protocol

Executing a transaction T in Tebaldi requires coordinating the actions of all the CCs handling T. These CCs form a path π in the CC tree (starting at the root and ending in CC_n) as each CC delegates some of T's conflicts to a child. Tebaldi executes T as follows:

Start phase In the top-down pass, each CC on π allocates the specific metadata that it requires. A CC may, for example, initialize data-structures that either uniquely identify the transaction or order it relative to concurrently running transactions (e.g., start timestamps in serializable snapshot isolation and timestamp ordering, or transaction id in lock-based protocols). More complex protocols, like Calvin [47], may also use the start phase to batch transactions, preordering transactions within a batch. At the end of the phase, the bottom-up pass, starting from CC_n, lets children inform their parent of T's new dependency set.

Execution phase In this phase, each CC along π runs its execution phase for each read and write operation in T. In doing so, CCs refine T's position in the overall transaction schedule: choosing to read from a version created by a transaction T_i orders T after T_i, while inserting a new object version before T_j orders T before T_j.

In the top-down pass, each CC executes its own CC-specific logic and appropriately constrains the ordering decisions of its children by blocking (or aborting) operations. Lock-based systems, for instance, delay operations until conflicting locks have been released, before placing a lock on the chosen object and executing their children's execution phase. The process is similar for multiversioned systems: though these techniques do not require blocking, they may decide to abort T on write-write conflicts.

The bottom-up pass has two components. First, as in the start phase, a child can forward T's dependency set to its parent. Second, the CCs on π collaboratively identify the appropriate version to return on a read operation. Specifically, a child CC proposes on a read operation a "candidate" version to return. Its ancestors can then amend the child CC's proposal based on transactions that may have written to that same object in sibling groups.

To illustrate, consider the CC tree in Figure 4. T_1 is in one group, while T_2 and T_3 are in another (controlled by CC_2); the interaction between the two groups is regulated by the cross-group concurrency control CC_1. Suppose transactions T_1 and T_2 both write object x (producing, respectively x_1 and x_2) and that CC_1 chooses to order T_1 after T_2 but before T_3 (solid edges). Now consider a read of x by T_3: as T_1 and T_3 are in different groups, CC_2 is unaware that T_1 wrote x_1. It thus proposes T_2's write x_2 as a candidate read value (red dashed edges). Returning this value would create a cycle in the final transaction schedule that consists of an anti-dependency edge from T_3 to T_1 (as T_3 misses T_1's write) and an ordering edge from T_1 to T_3. CC_1 thus "corrects" CC_2 and instead returns x_1, removing the cycle (blue dotted edges). Importantly, CC_2 never becomes aware of the existence of x_1. Restricting CC_2 to this partial view is necessary to preserve Tebaldi's modularity, which hinges on concurrency controls making ordering decisions solely for transactions in their group.

Validation phase This phase makes the ultimate decision [10] on whether T can commit and on its position in the final transaction schedule. In the top-down pass, each CC along π determines whether T is committable and, if desired, constrains its children's ordering decisions by delaying their validation phase. In the bottom-up pass, starting from the last CC on π, each CC forwards to its parent either T's dependency set or its decision to abort T. The parent CC can in turn use that information to determine whether it can commit T in an order consistent with its child's decision.

The process of deciding whether T can commit varies widely across CCs. Validation is trivial in lock-based systems, as having acquired all locks is sufficient to ensure commit. Optimistic systems must instead verify whether the objects read by T are still current, or otherwise abort T. Likewise, the ease with which the full set of dependent transactions can be reported also varies. In most single-version systems, T always knows its dependency set by the end of the validation phase. In contrast, that information is not available in multiversion systems like SSI until all transactions that were concurrent with T have committed.

Commit phase Tebaldi guarantees that T is committed atomically across all CCs by ensuring that the chained commit phases execute uninterrupted, starting from the leaf CC on π.

5.2 Limitations

Tebaldi's framework strives to be general, modular, and extensible. These benefits, however, come at the cost of some efficiency, as, unlike systems designed to work solely with a fixed subset of CCs, Tebaldi cannot co-design components. Single-versioned concurrency controls, for instance, do not need to keep version histories, whereas Tebaldi's storage module must store them to support multiversioned CCs. Similarly, while many specialized systems can benefit from co-locating concurrency controls' metadata (such as locks and timestamps) with the actual data, Tebaldi's generality requires their separation. Finally, some CCs, such as 2PL, do not need a validation phase.

Moreover, Tebaldi's ability to incorporate a given CC is based on the assumption that the CC can be expressed using its protocol's four phases and modified to guarantee consistent ordering (§ 4). These assumptions, though valid for common concurrency controls, may not hold universally; and, even when they do, may reduce efficiency. For example, batching increases the probability of write-write conflicts in snapshot isolation (§6, §8.4) and may reduce the scheduling flexibility of time-travelling concurrency controls like TicToc [57].

6. USE CASES

This section sketches the different concurrency control protocols currently supported by Tebaldi. This initial selection achieves a dual purpose. First, it illustrates how one can guarantee consistent ordering for real, well-known concurrency controls and how these CCs can be implemented in Tebaldi. Second, it speaks to the generality of our approach: Tebaldi supports two lock-based, single-versioned protocols (traditional two-phase locking [7, 22] and the recently proposed runtime pipelining [56]), and two multiversioned protocols (serializable snapshot isolation [12, 24, 36], and multiversioned timestamp ordering [39]). We describe each mechanism in turn.

Two-Phase Locking (2PL) Our implementation directly follows that of the seminal algorithm [7, 22]: transactions acquire shared read locks when executing a read operation and exclusive write locks when executing write operations. Every transaction holds these locks until commit, so as to guarantee serializability. Any deadlocks are handled by timing out transactions.

Implementing 2PL as a non-leaf CC requires only two small changes [56] to the algorithm. First, 2PL delegates in-group concurrency control to its children CCs by marking all locks acquired by transactions from the same group as non-conflicting. Second, 2PL ensures consistent ordering by delaying a transaction's commit until all its in-group dependencies have also committed.

2PL takes no action in the start phase. All necessary locks are acquired in the top-down pass of the execution phase. The bottom-up pass of the execution phase decides the appropriate read version to return: 2PL accepts the child's proposal if it is an uncommitted value from its group or else returns the latest committed value. The validation phase then gathers the committing transaction's dependency set from the child CC and delays commit until all transactions in that set have committed. Finally, it releases the locks in the commit phase.

Runtime Pipelining (RP) Runtime pipelining [56] handles data conflicts more efficiently than 2PL through a combination of static analysis and runtime constraints. RP first statically constructs a directed graph of tables, with edges representing transactional data / control-flow dependencies, and topologically sorts each strongly connected set of tables. Transactions are correspondingly reordered and split into steps, with step i accessing tables in set i. A runtime pipeline ensures isolation: once T_2 becomes dependent on T_1, T_2 can execute step i only once either T_1 has terminated, or T_1 is executing a step larger than i. Operations within a step are isolated using 2PL.

RP's start phase initializes the step counter and dependency set. In the execution phase, RP delegates concurrency control within each group to the child CC by allowing transactions from the same group to execute the same step concurrently. Upon starting a new step i, RP first "commits" the previous step by releasing the step-level lock (after in-group dependencies have also step-committed). It then waits both for all cross-group dependencies to *finish* executing step i, and for all in-group dependencies to *start* executing step i, before acquiring the step-level lock. The bottom-up pass of the execution phase computes in-group dependencies and decides the appropriate read version to return: RP accepts the child's proposal if it is a write from its group that has not step-committed. Otherwise, it returns the latest step-committed value. The validation phase delays a transaction's commit until transactions in its dependency set have committed.

Serializable Snapshot Isolation (SSI) Tebaldi supports a distributed implementation of serializable snapshot isolation [12, 24, 36], a multiversioned protocol that rarely blocks readers. As in snapshot isolation, transaction order is decided using start/commit timestamps: transactions read from a snapshot at the start timestamp, while writes become visible at the commit timestamp. SSI ensures serializability by detecting (and preventing) "pivot" transactions that have both incoming and outgoing anti-dependency edges.

Enforcing consistent ordering in SSI requires care. Firstly, unlike 2PL and RP, SSI partially decides transaction ordering through start timestamp assignment. Consider for example two transactions T_1 and T_2 from the same group, with T_1 having a smaller start timestamp. Consistent ordering can be violated if the child CC orders T_2 before T_1, as T_2 may observe a write from another group that T_1 cannot see. To address this, Tebaldi uses *batching*. Instances of transactions from the same group are placed in a batch and assigned the *same* start timestamp, delaying their relative ordering until commit. A child CC is then free to order batched transactions without violating consistent ordering. Though transactions in a batch share a start timestamp, they can commit individually with different commit timestamps (once all their in-group dependencies have already committed). Introducing grouping and batching means that SSI must detect and prevent *pivot batches*, with both incoming and outgoing anti-dependencies.

The start phase assigns the batch's start timestamp to the transaction (determined by a centralized timestamp server). During the execution phase, SSI tracks pivot batches by asynchronously querying a *group manager* that keeps track of batches' anti-dependencies. Cross-group write-conflicting transactions are aborted. In the bottom-up pass of the execution phase, SSI decides on the appropriate read version: SSI accepts the child's proposal if it is a value from its own batch, and otherwise returns the latest committed version whose commit timestamp is smaller than the transaction's start timestamp. Finally, the validation phase waits for the asynchronous pivot-check replies, and for dependent transactions to commit, before acquiring the final commit timestamp and reporting the transaction's dependency set to the parent CC. Doing so may require additional waiting: the full set of anti-dependencies is not known until all transactions with a smaller start timestamp finish executing.

The aforementioned protocol can be further optimized under certain assumptions. For instance, if SSI is used as the CC at the root of the CC tree to separate read-only transactions from update transactions (as is often the case), the protocol can be optimized as follows. First, SSI does not need to wait for concurrent transactions to finish executing, as root CC does not need to report a transaction's dependency set. Second, in the presence of a single update child group (which can be further partitioned), batching is no longer necessary. Indeed, as transactions in the update group will never observe values from the read-only group, it is not necessary to assign them start timestamps. Consistent ordering can be achieved simply by committing transactions in the update group according to their in-group order. Finally, checking for pivot batches is not necessary, as a pivot batch must involve at least two update groups.

Multiversioned Timestamp Ordering (TSO) Multiversioned timestamp ordering [9, 39] minimizes snapshot isolation's high abort rates under heavy write-write conflicts. TSO decides the serialization order by assigning a timestamp to every transaction at their start time. A writer creates a new object version marked with its timestamp, unless a reader with a larger timestamp has read the prior version (i.e., has missed this write), in which case the writer is aborted. A read returns the latest version with a timestamp smaller than the reader's. To prevent aborted reads, a transaction logs the write-read dependencies, and only commits after all these dependencies have committed. Tebaldi, inspired by Faleiro et al. [23] implements an optimization: promises. Transactions can optionally specify at start time object keys that they will write during their execution. Tebaldi then delays any transactions that attempt to read those values until the corresponding write occurs (instead of eventually having to abort the write transaction).

To enforce consistent ordering in TSO, Tebaldi once again uses batching. The start phase creates a batch of transactions for each child group and assigns the same timestamp to all transactions in the same batch. As in SSI, batching delays TSO's ordering decisions for a batch until commit time, giving children CCs complete freedom in how they order transactions.

In the execution phase, the write logic remains identical. For reads, TSO accepts the child's proposal if it is a write from its own batch. Otherwise, it returns the latest version of that object with a smaller timestamp. In the validation phase, TSO uses the in-group dependency reports to decide on the order of transactions within a batch, and commits a transaction only after all its in-group dependencies have committed. As TSO exposes uncommitted values across groups (unlike SSI), the protocol must additionally verify that later batches read the latest write from previous batches: consistent ordering can be violated if the final order of writes differs from their execution order. Suppose, for instance, that two transactions T_1 and T_2 in the same batch B write the same object. If T_1 *executes* the write before T_2, a reader from another group, ordered after B, will read T_2 instead of T_1, even if the child CC eventually orders T_1 after T_2. To prevent this, TSO delays committing a transaction T until all batches with lower timestamps have committed. It aborts T if there exists a later object version that has the same timestamp as the version observed by T. Finally, TSO can conservatively report a transaction's dependency set to its parent to include all transactions with a smaller timestamp.

7. IMPLEMENTATION

The current prototype of Tebaldi provides support for tables, variable sized columns, and read-modify-write operations. It does not however currently support range operations, durability or replication. Our implementation consists of 21K lines of C++ code.

Cluster architecture A Tebaldi cluster consists of two major types of nodes. The *transaction coordinators* (TC) manage transactions' states, while *data servers* (DS) hold partitions of the data and handle data access requests from TCs.

The implementation of the four phases discussed in §5.1 is split between TC and DS nodes and follows a common pattern: for each phase, the TC issues request(s) to the appropriate DS and waits for the reply. In certain phases, some CCs may omit contacting the DS, while others may require additional communication. For example, in runtime pipelining the TC for transaction T contacts the TCs for the transactions in T's dependency set to determine when it is safe for T to begin executing a new step. The DS, meanwhile, maintains a lock table and manages timeouts.

Phase optimizations We previously introduced each of the four protocol phases as two-pass procedures: a top-down pass where the parent CC constrains the execution of its children, followed by a bottom-up pass where the child CC informs the parent of its ordering decisions. In our experience however, few CCs leverage the bottom-up pass in the start phase or the top-down pass in the validation phase. Our current implementation removes them for efficiency.

Latency reduction Sequentially executing the respective phases of every CC in a CC tree of height h could result in up to $O(h)$ network round trips, as each CC's logic may involve communication between the TC and DS. To side-step this issue, Tebaldi, for each phase, first executes the TC component of every CC, batching communication to the DS. The DS in turn executes the DS part of every CC, and batches replies to the TC. This reduces the framework's latency to a single round-trip per phase, in line with prior non-hierarchical approaches.

Garbage collection Tebaldi implements a garbage collection service that prunes stale versions from multiversioned storage. Logically, a write can be GCed when all CCs agree that it will never be read again. For efficiency, Tebaldi processes records in batch within a *GC epoch*: Tebaldi assigns a GC epoch id to every transaction, periodically incrementing the epoch. When all transactions in an epoch finish, Tebaldi asks all CCs to confirm that they will never order ongoing or future transactions before a transaction in this epoch. Once all CCs have confirmed, all stale writes in the epoch can be GCed.

8. EVALUATION

Tebaldi seeks to unlock the full potential of federating concurrency controls by applying MCC hierarchically. To quantify its benefits and limitations, we ask the following questions:

- How does Tebaldi perform compared to monolithic concurrency controls and two-layered MCC systems? (§8.1 and §8.2)
- Is Tebaldi's framework conducive to adapting CC trees to changes in the workload? (§8.3)
- How do Tebaldi's different features contribute to increasing concurrency? (§8.4)
- What is the overhead of running multiple CCs? (§8.5)

Experimental setup We configure Tebaldi to run on a CloudLab [1] cluster of Dell PowerEdge C8220 machines (20 cores, 256GB memory) connected via 10Gb Ethernet. The ping time between machines ranges from 0.08ms to 0.16ms. The cluster contains 20 machines for the TPC-C and SEATS experiments, and ten machines for microbenchmarks; each machine runs ten data server nodes and ten transaction coordinators (with one additional machine for timestamp assignment and batch management under SSI).

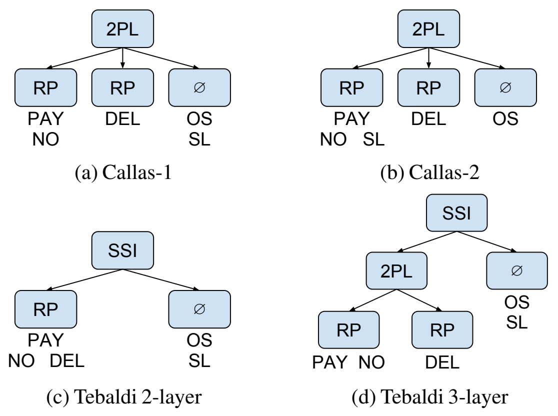


Figure 5: CC trees used in TPC-C. Leaf nodes are labeled with transactions: `payment` (PAY), `new order` (NO), `delivery` (DEL), `order status` (OS), and `stock level` (SL).

Benchmarks We evaluate the performance of our system using several microbenchmarks, TPC-C [16], and SEATS [19]. TPC-C models the business logic of a wholesale supplier and is the de-facto standard for OLTP workloads, while SEATS simulates an airplane ticket selling service.

We adapt the TPC-C benchmark to our transactional key-value store interface: we remove the scan over a customer's last name in the `payment` and `order status` transactions. We additionally use a separate table as a secondary index on the order table to locate a customer's latest order in the `order status` transaction. Our current implementation does not contain the 1% of aborted `new order` transactions. In line with prior work that focuses on contention-heavy workloads [33, 55, 56, 59], we run TPC-C test clients in a closed-loop and populate ten warehouses.

We also adapt the SEATS benchmark: we keep its application logic but reduce the number of available flights to demonstrate the benefits of hierarchical grouping under high-contention, and significantly increase the number of seats in each "flight" to run the benchmark for sufficiently long. The configuration we ultimately adopt, though unrealistic for airlines, may model seating assignments for a small number of sporting events, each with a large number of seats. Specifically, we remove the scan over the customer name (in `delete reservation` and `update customer`) and use separate tables as secondary indices on the `reservation` table to locate the reservation id based on the flight id and seat / customer id. Further, we reduce the number of available flights to 50, increase the number of seats available per flight to 30,000, and reduce the number of seats accessed in `find open seats` to 30.

Optimal grouping for both benchmarks is obtained by recursively identifying highly contended transactions and pairing them with concurrency controls well-suited to the transactions' inherent structure. We give specific details for each benchmark in §8.1 and §8.2.

8.1 Tebaldi's performance on TPC-C

Baselines We compare Tebaldi against two monolithic concurrency controls (2PL and SSI) and the federated system Callas [56]. These systems are implemented within the Tebaldi framework, and hence make use of the same network and storage stack. In SSI, we allow aborted transactions to backoff for 5 milliseconds before retrying to reduce the resource consumption.

Grouping To configure the Callas system, we start with the grouping strategy proposed in the Callas paper. This initial grouping (Callas-1, Figure 5a) partitions transactions into three groups. The first group contains `new order` and `payment`, whose conflicts can be aggressively optimized by runtime pipelining. The second group uses another instance of runtime pipelining for the `delivery` transaction. Finally, the two read-only transactions are in the third group. This grouping, when running under serializability (Callas originally runs under read-committed), introduces a large number of cross-group read-write conflicts between the `stock level` and `new order` / `payment` transactions. Because of the fixed structure of Callas, these conflicts must be handled by 2PL. To mitigate this problem, we modify Callas-1 by moving `stock level` in the first group, where it can be pipelined with `new order` (Callas-2, Figure 5b).

Thanks to the flexibility of hierarchical MCC, Tebaldi supports a wider variety of grouping strategies than its cousin Callas. We propose two such groupings, in Figure 5c and Figure 5d respectively. The first grouping strategy (Tebaldi 2-layer) leverages Tebaldi's ability to support cross-group protocols other than 2PL by selecting a multiversion cross-group protocol, SSI. It then partitions transactions into two groups: a read-only group containing transactions `order status` and `stock level` that requires no in-group concurrency control; and an update transaction group that uses runtime pipelining to optimize the `new order`, `payment`, and `delivery` transactions.

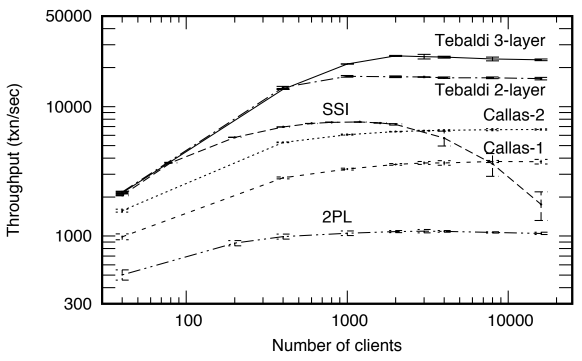


Figure 6: Performance of TPC-C benchmark.

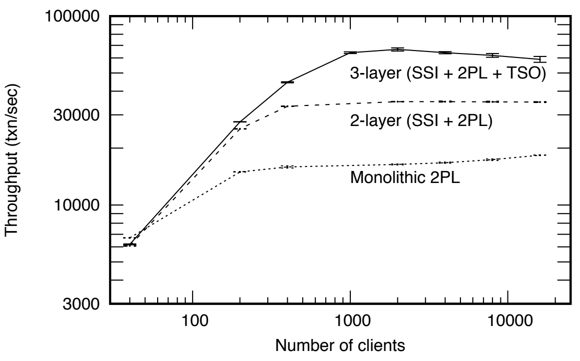


Figure 7: Performance of SEATS benchmark.

The second grouping strategy (Tebaldi 3-layer) instead leverages Tebaldi's hierarchical model by creating a concurrency control tree of depth three. This approach partitions transactions into the same leaf-level groups as the original Callas grouping (Callas-1). However, it then uses two distinct cross-group mechanisms to handle the remaining conflicts: 2PL for conflicts between `new order / payment` group and the `delivery` group, and SSI for conflicts between the read-only group and all other groups.

Results Figure 6 compares the performance of Tebaldi's grouping strategies against those of Callas, and against two monolithic concurrency control protocols: two-phase locking (2PL) and serializable snapshot isolation (SSI).

Consider first the performance of the two monolithic concurrency controls: the peak throughput of SSI is 7× higher than that of 2PL, because of the high read-write conflict ratio between `new order` and `payment` (the transactions in fact read and write different columns, but Tebaldi, like most other systems, takes row-level locks). As contention increases (by increasing the number of clients), the performance of SSI drops steeply as the high write-write conflict rate causes SSI to repeatedly abort transactions.

When contention is high, the performance of SSI and 2PL is lower than that of Callas and Tebaldi. Callas's initial grouping strategy (Callas-1) is bottlenecked by the heavy read-write conflicts between `stock level` and `new order` / `payment`. Revising this partitioning (Callas-2) yields a 77% throughput increase, but decreases the efficiency of RP (by moving `stock order` and `new order` to the same group): `new order`'s writes to `order`, `new order` and `order line` tables, which could never conflict in the previous grouping (as order ids are unique), can now read-write conflict with `stock order`'s reads (creating additional synchronization in RP). Additionally, combining `stock level` and `new order` creates circular table dependencies, resulting in a coarser-grained pipeline. There is once again a tension between the potential inefficiency of a monolithic cross-group mechanism, and the in-group mechanism's desire to handle few transactions.

To side-step this tension, Tebaldi has two options: select a more suitable cross-group mechanism (Tebaldi 2-layer), or create a deeper grouping hierarchy (Tebaldi 3-layer). The former yields a $2.6\times$ improvement over the best grouping strategy in Callas: SSI, as a cross-group mechanism, can efficiently handle the read-write conflict between `stock level` and `new order`, while the three update transactions can be pipelined fairly efficiently. This pipelining remains suboptimal however, as the potential conflict between `new order` and `delivery` voids `new order`'s unique access to tables, creating additional synchronization in `new order`'s execution. The latter grouping strategy (Tebaldi 3-layer) addresses this issue: the small and carefully selected scope of each group gives every in-group concurrency control the opportunity to perform well. The rare conflicts between the `new order` and `delivery` transactions are regulated by 2PL, while the common read-write conflicts are resolved using SSI. This careful tailoring of cross-group CCs to cross-group conflicts allows the three leaf groups to remain small. This narrow scope results in optimal pipelines for the two groups running RP, while the read-only group can operate without any concurrency control, leading to a further 44% improvement.

8.2 Tebaldi's performance on SEATS

Grouping We compare three grouping strategies. The baseline is a monolithic 2PL system. To optimize the read-write conflicts, our second grouping uses SSI to separate the read-only transactions (`find flights` and `find open seats`) from the update transactions, and uses 2PL to regulate the remaining update transactions. The third grouping further optimizes the conflicts among the update transactions. Unlike TPC-C however, these highly contended read-write transactions (`create`, `update` and `delete` of reservation) cannot be efficiently pipelined using RP because of the circular dependency between tables (`flight`, `customer` and `reservation`). Nonetheless, TSO can still pipeline these transactions by preordering them at runtime using timestamps. In doing so, however, it creates many spurious dependencies between non-conflicting transactions. To alleviate this concern, we observe that transactions that access different flights rarely conflict. We leverage Tebaldi's flexible grouping to create not one, but multiple TSO instances, one for each flight, and assign transactions to their group at start time according to their input, using 2PL as cross-group mechanism. This approach efficiently pipelines the (likely) conflicts for transactions that access the same flight but does not unnecessarily order those that don't (in the rare cases when conflicts between transactions accessing different flights arise, they are still handled by 2PL).

Results Figure 7 compares the performance of Tebaldi's three-layer hierarchy against those of two-phase locking (2PL) and the two-layer hierarchy (SSI+2PL). We find that, unsurprisingly, the

```
// input: w_id, d_id
begin transaction
    o_id = district[w_id, d_id].next_order_id;
    ol_num = order[w_id, d_id, o_id].ol_num;
    for (i = 0; i < ol_num; ++i) {
        item_id = order_line
                [w_id, d_id, o_id, i].ol_i_id;
        item_stats[item_id]++;
    }
commit
```

Figure 8: Pseudocode of `hot item`.

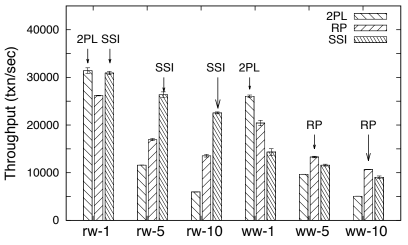


Figure 9: Cross-group CCs' performance.

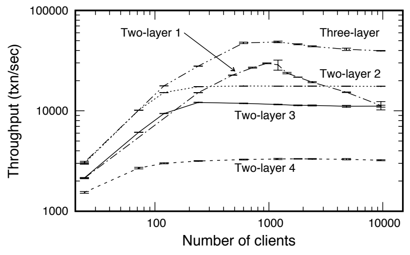


Figure 10: Two-layer vs three-layer.

peak throughput of the two-layer setting is $1.9\times$ higher than that of 2PL as it minimizes the effect of the read-write conflicts introduced by the long-running read-only transactions `find flights` and `find open seats`. As contention increases, however, `new reservation` transactions spend a prohibitive time waiting to acquire exclusive locks, hampering performance. Pipelining transactions that access the same flights using TSO yields a further about $2\times$ speedup: TSO can pipeline operations and expose uncommitted writes to subsequent transactions without delays. Tebaldi's flexibility enables a "hybrid" type of grouping: first by transaction type, and then by data (input type), once one knows whether two transaction instances will actually conflict. Thus, Tebaldi gets the best of both worlds: the efficiency of 2PL when transactions rarely conflict, and the localized performance gains of TSO's pipeline when transactions do.

8.3 Extensibility

For moderate changes in the workload, Tebaldi's modular and flexible design makes it possible to handle the new conflict patterns by adding new concurrency controls to the existing CC trees. To showcase this benefit, we add a new transaction `hot item` to TPC-C and sketch how the prior Tebaldi 3-layer hierarchy can be refined to account for the additional conflicts.

This new transaction (shown in Figure 8) computes popular or "hot" items by randomly sampling recent orders in the database, and aggregating the per-item sale count over all warehouses. We set the new TPC-C workload distribution to be the following: 41.8% of transactions are `new order` transactions, 41.8% are `payment`, while the remaining transactions all run 4.1% of the time.

The `hot item` and `new order` transactions read-write conflict heavily: neither of the current cross-group 2PL or SSI are thus good choices for regulating their behavior (the batching in SSI will periodically promote write-write conflicts between `new order` instances to cross-group conflicts, causing aborts). Instead, we have two solutions: we can keep the same three-layer hierarchy, placing the new transaction in the same group as `new order` and `payment` at the cost of a less efficient pipeline. Alternatively, we can leverage Tebaldi's flexibility to place the transaction in a separate group and use RP as the cross-group mechanism to regulate conflicts with the `new order`/`payment` group.

The experiment shows that the three-layer approach has a throughput of 16417 ± 192 txn/sec, while the four-layer approach gives 23232 ± 111 txn/sec. Placing `new order` and `hot item` in the same group reduces the pipeline's efficiency as `new order`'s accesses to tables are no longer guaranteed to be non-conflicting. In the four-layer solution, we side-step this issue by placing `hot item` and `new order` in their own group, yielding a 42% throughput increase.

8.4 Impact of flexibility

We next investigate in more detail how Tebaldi's higher flexibility enhances concurrency. Tebaldi increases flexibility over prior federated systems in two ways: by supporting multiple cross-group CCs, and by enabling finer partitioning of conflicts.

Support for different cross-group protocols We first quantify the potential gains associated with Tebaldi's support for different cross-group concurrency controls. To do so, we compare the performance of different cross-group CCs for different conflict patterns. We use a two-layer hierarchy with two groups; we fix the in-group CCs and set the cross-group CC to be either 2PL, SSI, or RP. In the first three workloads, each group contains an update transaction consisting of seven write operations. The first operation writes to a shared table consisting of n rows, so the conflict rate for both in-group and cross-group is $1/n$. The second operation writes to a group-local table of ten rows, adding only in-group conflicts. The remaining operations within the group conflict with low probability $(1/10000)$. Both groups use RP to handle in-group conflicts. By tuning n, we vary the cross-group conflict ratio in each workload (for benchmarks *ww-1*, *ww-5*, and *ww-10*, respectively 1%, 5%, and 10% of write-write conflicts). In the three remaining workloads, we replace one of the write-only groups with a read-only group. As read-only transactions never conflict with each other, we use an empty in-group concurrency control protocol. As above, we vary the cross-group conflict rate, this time of read-write conflicts, in each benchmark (for benchmarks *rw-1*,*rw-5*, and *rw-10*, respectively 1%, 5%, and 10%).

Figure 9 summarizes the throughput for each workload. We report the results in transactions per second as the average of three runs. We find that no single cross-group protocol outperforms the others in all cases, underscoring the practical importance of selecting the cross-group mechanism most suitable for a given workload. Specifically, we find that, SSI, unsurprisingly, performs best when handling read-write conflicts across groups, as readers and writers never block each other. In contrast, SSI performs worse in the presence of write-write conflicts because of repeated aborts. Its poor performance is exacerbated by the need for batching, as write-write conflicts cannot be resolved until the next batch change (in *ww-1* transactions already retry on average more than 2.5 times). Aborts in this benchmark are relatively cheap (they mostly happen on the first operation). Costlier aborts would likely cause SSI's performance to drop. Runtime pipelining, in contrast, performs best in scenarios with medium to high amounts of write-write contention (*ww-10*,*ww-5*). When write-write conflicts are rare, however, the overhead of maintaining the pipeline outweighs its benefit: the more conservative but simple 2PL performs best (*ww-1*).

Hierarchical application of MCC The previous microbenchmark was restricted to two-layer grouping strategies with a single cross-group mechanism per configuration. We next quantify the benefits of using deeper hierarchies in which we can combine multiple cross-group protocols. To do so, we focus on a scenario in which no single cross-group mechanism can efficiently handle the pairwise interactions of all transaction groups. This represents a best-case scenario for Tebaldi; we quantify potential overheads associated with deeper hierarchies in §8.5.

The microbenchmark consists of one read-only transaction, T_1 and two update transactions, T_2, and T_3. T_1 read-write conflicts heavily with T_2 and T_3, while T_2 and T_3 cannot be efficiently handled within a group. Table A suffers from heavy contention as it contains only ten rows, while all other tables (B to E) contain $10{,}000$ rows and rarely contend. Transaction T_1 reads a single row in A, and ten rows from the remaining tables. Transaction T_2 first writes a row in A, and subsequently writes a random key from every table B to E. Transaction T_3 does not access table A. Instead, it reads a random key from tables B to E, and subsequently writes back to B. Considering the previous in-group mechanism, runtime pipelining: RP can handle T_2 efficiently, but not T_2 and T_3 (or T_1).

Constructing a three-layer CC tree in Tebaldi side-steps this issue. The read-write conflict between T_1 and T_2/T_3 can be handled efficiently through selecting SSI as root CC, placing T_1 and T_2/T_3 in separate groups. Next, as T_2 and T_3 rarely conflict with each other, they can be placed in separate groups with 2PL as cross-group CC. Finally, conflicts between different instances of T_2 can be efficiently pipelined with RP. As contention is low for T_3, we simply use 2PL as its in-group mechanism.

We compare our solution to the four most promising two-layer hierarchies. The first two hierarchies use SSI as cross-group mechanism to optimize the read-write conflict between T_1 and T_2, but differ in how they handle T_2 and T_3: the first grouping strategy (Two-layer 1) puts T_2 and T_3 in separate groups. It allows T_2 to be efficiently pipelined, but requires SSI to run with batching enabled, which can periodically promote in-group conflicts to cross-group conflicts. The second baseline (Two-layer 2) places T_2 and T_3 in the same group. This gives SSI better performance at the cost of a less efficient in-group pipeline. The third grouping strategy (Two-layer 3) has T_1 and T_2 in one group running RP, and T_3 in another group running 2PL. 2PL is used across these two groups. This avoids the issues associated with having SSI across groups, yet still optimizes the conflict between T_1 and T_2 at the cost of a less efficient pipeline for T_2. The last baseline (Two-layer 4) runs all three transactions in separate groups (2PL cross-group). It prioritizes T_2 by using the optimal pipeline. None of these four solutions is perfect: while T_1, T_2 and T_3 would all benefit from being in a single group, no single concurrency control is well-suited to handle conflicts between T_1/T_2 and T_2/T_3.

Figure 10 confirms our intuition. The peak throughput achieved by the three-layer hierarchy is 63% higher than the best performing two-layered grouping strategy. The fourth (Two-layer 4) grouping strategy performs worst as two-phase locking cannot efficiently handle the frequent read-write conflicts between T_1 and T_2. The first baseline (Two-layer 1) performs best under moderate number of concurrent clients but suffers from a high abort rate due to SSI's sensitivity to the write-write conflicts: its performance drops as the number of clients increases. Finally, the performance of both the second and third grouping options (Two-layer 2 and 3) is hampered by a sub-optimal runtime pipeline.

Setting	Latency (ms)	Throughput (K txn/sec)
stand-alone RP	2.969 ± 0.004	490.0 ± 1.7
2PL - RP	3.068 ± 0.004	386.1 ± 0.4
SSI - RP	3.259 ± 0.006	368.4 ± 1.7
RP - RP	4.047 ± 0.004	291.9 ± 0.8

Table 2: Latency and resource cost of adding additional layers.

8.5 Overhead

Tebaldi attempts to improve the performance of applications that are bottlenecked on data conflicts. It does so by enhancing the concurrency of these applications, at the cost of more complex control logic: each transaction needs to percolate through every concurrency control in its path on the CC tree. This additional complexity can negatively impact the application in two ways: it can increase the latency of transactions, and can increase the application's resource consumption. We quantify these potential drawbacks in this section. To do so, we run a microbenchmark with grouping strategies that do not yield any additional concurrency, and measure the resulting cost increase. The benchmark consists of a single transaction type with seven write operations, and ensures concurrent transactions never conflict with each other. We use a stand-alone runtime pipelining protocol as the baseline, and add either 2PL, RP, or SSI cross-group layers to the hierarchy.

Latency overhead To measure the impact of the hierarchy's depth on latency, we run the benchmark with a small number of clients (20) to ensure that the resource (CPU / network) consumption is low. Our results are shown in the second column of Table 2 and denote the transaction's average latency over ten 60 seconds runs. We find that the relative latency increase of adding an additional layer in the hierarchy depends heavily on the cross-group concurrency control being added. Adding a 2PL cross-group layer yields a small 3.3% increase in latency. This increase is exclusively due to computational overhead: the number of round-trips in Tebaldi is independent of the hierarchy depth (§7). Any necessary additional round trips is thus a property of the concurrency control itself: 2PL requires no additional round trips, while SSI requires an additional round trip to contact the timestamp server, and RP requires an additional round-trip per operation. These additional network trips are reflected in our result: adding an SSI cross-group layer increases latency by 9.8%, while adding an RP cross-group layer increases latency by 36.3%.

Computation resources overhead Under high system load, the computational overhead of adding CCs could become prohibitive, bottlenecking the system on CPU or network resources. To quantify this overhead, we increase the workload to measure the peak throughput of the microbenchmark, ensuring that the CPU is the bottleneck each time. Our results are summarized in the third column of Table 2. Adding a 2PL layer over an RP in-group mechanism leads to a 21% decrease in throughput while adding an SSI layer leads to 25% drop. The overhead is relatively small, as 2PL and SSI remain fairly light-weighted when compared to RP. The overhead of adding an RP layer is more significant: throughput drops by 40%, as RP is fairly complex. Note that, even when adding an RP layer over the RP in-group mechanism, the throughput does not simply halve, as many components of the framework are independent of the hierarchy depth.

9. RELATED WORK

Tebaldi generalizes the concept of modular concurrency control introduced in Callas [56]. Callas is part of a large body of work that seeks to improve database performance by composing multiple concurrency controls. We summarize it here.

Partitioning by transactions or conflicts Many systems seek to improve performance by tailoring concurrency controls to specific transactions or conflicts. Mixed concurrency control [8], for instance, uses different CC mechanisms to handle write-write and write-read conflicts. Likewise, multiversion two-phase locking [9, 13, 14, 21, 50] partitions transactions into read-only and update transactions, regulating update transactions using 2PL but allowing read-only transactions to read, without blocking, from a consistent snapshot by using a multiversioned protocol. Similarly, H-Store [46] optimizes transactions that can execute on a single shard by removing unnecessary network communication, just as Granola [17] optimizes *independent* distributed transactions, i.e., transactions where each site can reach the same decision even without communication, by eliminating the two-phase commit protocol.

Though these techniques increase concurrency, they cannot flexibly combine CCs as they simply partition transactions in two types, with all conflicts of the same type handled by the same CC. In contrast, Tebaldi does not specify the partitioning a-priori and can combine diverse CC mechanisms.

Partitioning by data Other systems instead partition the database into multiple disjoint components, allowing each component to execute its own concurrency control while preserving consistency globally. Federated databases [11, 37, 38, 41] for instance, support serializable *global transactions* that touch multiple local databases running different CCs. Some systems [32] even compose federated DBs into a hierarchical structure. Unlike Tebaldi, federated DBs are motivated primarily by functionality requirements (i.e., the ability to execute transactions across different databases) rather than performance: indeed, they can perform *worse* than local DBs. Nonetheless, many of their techniques are relevant: Tebaldi's guidelines for modifying CCs to guarantee consistent ordering draw heavily from the federated databases' notion of serialization points [11, 37].

Alternatively, some systems partition data such that correctness is directly guaranteed if transactions are per-partition serializable, removing the need for cross-partition CCs. For instance, the work on local atomicity properties [51] explores the required properties of per-object CCs needed to guarantee database-level serializability. Likewise, Sha et al.[42] partitions the database into atomic datasets, each with its own CC, such that no consistency invariant spans multiple datasets. Database-level consistency then directly follows from per-dataset serializability. These approaches suffer from two main limitations: first, they place stringent constraints on how they allow data to be partitioned and CCs to be combined. Second, they require all conflicts on the same data partition to be handled by the same CC.

Partitioning by time Certain systems choose to partition the database into distinct phases during which different CCs can execute. Granola [17] executes one-shot and "independent" transactions (i.e., transactions whose commit/abort decision is deterministic) in *timestamp mode*, using an efficient lock-less timestamp-based protocol. Transactions that require coordination are instead constrained to execute in a less efficient *locking mode* that relies on traditional 2PL. Likewise, Doppel [34] distinguishes between joined/split and reconciliation phases. Transactions in the joined phase use traditional optimistic concurrency control. Transactions in the split/reconciliation phase, assuming their operations commute, are guaranteed never to conflict: instead, they modify split per-core state that is subsequently merged in the reconciliation phase.

Hierarchical decomposition Multi-level serializability [6, 40, 52, 53, 54] observes that transactions can be hierarchically decomposed such that each layer in a tree captures operations at a different level of abstraction: seemingly atomic operations at layer $i+1$ may in fact be implemented as composite elements at layer i, with different concurrency controls used at different layers. A sufficient condition to ensure serializability in this context is level-by-level serializability [5]: assuming conflicting operations at level $i+1$ generate at least one conflicting operation at level i, serializability is guaranteed if the serialization graph between levels i and $i+1$ is acyclic. Intuitively, this means that if a level orders conflicting operations, the corresponding operations at higher level should be ordered consistently. In the context of multi-level serializability, individual CCs regulate the interactions of *all* transactions at a given level. In contrast, CCs in Tebaldi are responsible only for a subset of transactions.

Specialized concurrency controls Some concurrency controls achieve better performance by targeting specific workloads, or assuming specific properties on transactions [23, 46, 47, 59]. These could be incorporated into Tebaldi within a group.

Pot-pourri Alternative approaches to improving database performance include relaxing consistency at the cost of additional programming complexity [18, 28, 30, 31, 43], relying on new hardware [20, 23, 26, 48, 49, 57], and co-designing the replication protocol with the concurrency control mechanism [58]. These techniques are orthogonal to hierarchical MCC and, if combined, would likely be mutually beneficial.

10. CONCLUSION

By allowing more flexibility in how concurrency control mechanisms can be assigned to manage conflicts, Tebaldi explores new ways to harness the performance opportunity of combining different specialized concurrency controls within the same database. Its underlying ethos is the hierarchical application of MCC: Tebaldi partitions conflicts at a fine granularity and matches them to a wide variety of CCs, within a framework that is modular and extensible.

Acknowledgments We thank the anonymous reviewers for their insightful comments, and Srinath Setty, Emin Gün Sirer, Vivek Shah, and Irene Y. Zhang for their valuable feedback on earlier drafts of this paper. We are deeply grateful to the amazing team at Cloudlab [1]. This work was supported by a Google Faculty Research award and by NSF grant CNS-1409555. Natacha Crooks was partially supported by a Google Fellowship in Distributed Computing.

References

[1] Cloud Lab. https://www.cloudlab.us/.

[2] MySQL. https://www.mysql.com.

[3] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering*, pages 67–78. IEEE, 2000.

[4] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[5] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A Model for Concurrency in Nested Transactions Systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[6] Catriel Beeri, Hans-Jörg Schek, and Gerhard Weikum. Multi-Level Transaction Management, Theoretical Art or Practical Need? In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '88, pages 134–154, London, UK, 1988. Springer-Verlag.

[7] P. A. Bernstein, W. S. Wong, and D. W. Shipman. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5:203–216, 1979.

[8] Philip A Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[9] Philip A Bernstein and Nathan Goodman. Multiversion Concurrency Control-Theory and Algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[10] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[11] Yuri Breitbart, Hector Garcia-Molina, and Avi Silberschatz. Overview of Multidatabase Transaction Management. In *Proceedings of the 1992 Conference of the Centre for Advanced Studies on Collaborative Research*, volume 2, pages 23–56. IBM Press, 1992.

[12] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, Vancouver, Canada, 2008. ACM.

[13] Arvola Chan, Stephen Fox, Wen-Te K Lin, Anil Nori, and Daniel R Ries. The Implementation of an Integrated Concurrency Control and Recovery Scheme. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 184–191. ACM, 1982.

[14] Arvola Chan and Robert Gray. Implementing Distributed Read-Only Transactions. *IEEE Transactions on Software Engineering*, (2):205–212, 1985.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[16] Transaction Processing Performance Council. TPC benchmark C, Standard Specification Version 5.11, 2010.

[17] James Cowling and Barbara Liskov. Granola: Low-overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[18] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1615–1628, San Francisco, California, USA, 2016. ACM.

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment*, 7(4), 2013.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association.

[21] Deborah DuBourdieux. Implementation of Distributed Transactions. In *Berkeley Workshop*, pages 81–94, 1982.

[22] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[23] Jose M. Faleiro and Daniel J. Abadi. Rethinking Serializable Multiversion Concurrency Control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[24] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation

Serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[25] Evan P. C. Jones, Daniel J Abadi, and Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 603–614. ACM, 2010.

[26] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Mchoppinoadden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[27] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[28] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[29] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1659–1674, San Francisco, California, USA, 2016. ACM.

[30] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, October 2011. ACM.

[31] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), Lombard, IL*, volume 13, pages 313–328, 2013.

[32] Sharad Mehrotra, Henry F Korth, and Avi Silberschatz. Concurrency Control in Hierarchical Multidatabase Systems. *The VLDB Journal - The International Journal on Very Large Data Bases*, 6(2):152–172, 1997.

[33] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 479–494, Broomfield, CO, October 2014. USENIX Association.

[34] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase Reconciliation for Contended In-memory Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 511–524, Broomfield, CO, 2014. USENIX Association.

[35] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, Melbourne, Victoria, Australia, 2015. ACM.

[36] Dan R. K. Ports and Kevin Grittner. Serializable Snapshot Isolation in PostgreSQL. *Proceedings of the VLDB Endowment*, 5(12):1850–1861, August 2012.

[37] Calton Pu. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of the 4th International Conference on Data Engineering*, pages 548–555. IEEE, 1988.

[38] Yoav Raz. The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Mangers Using Atomic Commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 292–312. Morgan Kaufmann Publishers Inc., 1992.

[39] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[40] Werner Schaad, Hans-J Schek, and Gerhard Weikum. Implementation and Performance of Multi-Level Transaction Management in a Multidatabase Environment. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering, 1995: Distributed Object Management*, pages 108–115. IEEE, 1995.

[41] Ralf Schenkel and Gerhard Weikum. Integrating Snapshot Isolation into Transactional Federations. In *Cooperative Information Systems*, pages 90–101. Springer, 2000.

[42] Lui Sha, John P Lehoczky, and Douglas E Jensen. Modular Concurrency Control and Failure Recovery. *IEEE Transactions on Computers*, 37(2):146–159, 1988.

[43] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[44] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[45] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, Cascais, Portugal, 2011. ACM.

[46] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[47] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, Scottsdale, Arizona, USA, 2012. ACM.

[48] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, Farminton, Pennsylvania, 2013. ACM.

[49] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, Monterey, California, 2015. ACM.

[50] William E Weihl. Distributed Version Management for Read-Only Actions. *IEEE transactions on Software Engineering*, (1):55–64, 1987.

[51] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(2):249–282, 1989.

[52] Gerhard Weikum. A Theoretical Foundation of Multi-level Concurrency Control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 31–43, Cambridge, Massachusetts, USA, 1986. ACM.

[53] Gerhard Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[54] Gerhard Weikum and Hans-J. Schek. Database Transaction Models for Advanced Applications. pages 515–553. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[55] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. SALT: Combining ACID and BASE in a Distributed Database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, October 2014. USENIX Association.

[56] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 279–294, Monterey, California, 2015. ACM.

[57] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, San Francisco, California, USA, 2016. ACM.

[58] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 263–278, Monterey, California, 2015. ACM.

[59] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geodistributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.