Optimistic Concurrency Control by Melding Trees

Philip A. Bernstein Colin W. Reid Ming Wu
Microsoft Corporation
{philbe, colinre, miw}@microsoft.com

Xinhao Yuan†
Tsinghua University, Beijing, China
xinhaoyuan@gmail.com

ABSTRACT

This paper describes a new optimistic concurrency control algorithm for tree-structured data called meld. Each transaction executes on a snapshot of a multiversion database and logs a record with its intended updates. Meld processes log records in log order on a cached partial-copy of the last committed state to determine whether each transaction commits. If so, it merges the transaction's updates into that state. Meld is used in the Hyder transaction system and enables Hyder to scale out without partitioning. Since meld is on the critical path of transaction execution, it must be very fast. The paper describes the meld algorithm in detail and reports on an evaluation of an implementation. It can perform over 400K update transactions per second for transactions with two operations, and 130K for transactions with eight operations.

1. INTRODUCTION

This paper describes a new optimistic concurrency control technique called **meld**. It is intended for use in a system with many servers that execute transactions on shared, tree-structured, multiversion data. The system could be a transactional indexed-record manager, which is used as an independent key-value store or the storage layer of a database system, such as Hyder [4]. It could be a log-structured file system that supports transactions [17]. Or it could be an in-memory transactional store on a shared-memory multiprocessor, where transactions execute on different cores.

Meld is targeted for use in a system where servers share one database log that is the only representation of the database. Every server maintains a locally-cached (partial) copy of the database in main memory, and it can execute transactions. When a transaction finishes, it appends a log record, called an **intention**, which contains the final values of the data items it modified. For serializable isolation the intention also identifies the transaction's readset. Unlike conventional database systems, appending an intention to the log does not commit the transaction. Meld makes that decision.

Every server executes the meld operator. Meld processes log records one-at-a-time in log-order. For each record, it determines whether the corresponding transaction commits. If so, it merges the transaction's updates into the server's cached database state.

Meld is deterministic. Thus, every server that runs meld makes the same commit and abort decision for every transaction. Yet the only point of arbitration between the servers is the operation that appends a transaction record to the shared log. Therefore, the system can grow simply by adding another server, giving it access to the log, giving it a recent snapshot of the database, and having it start running meld with the log record following the last one that contributed to the snapshot. The system can thereby scale out to many servers without partitioning the database or transactions.

The database is a multi-versioned tree-structured key-value store. Each transaction *T* executes against a database version. *T*'s intention *I* contains a reference *R* to *T*'s **snapshot**, which is defined by the last committed transaction in the log that contributed to the database state that *T* read (Figure 1). The intentions between *R* and *I* are called *T*'s **conflict zone**.

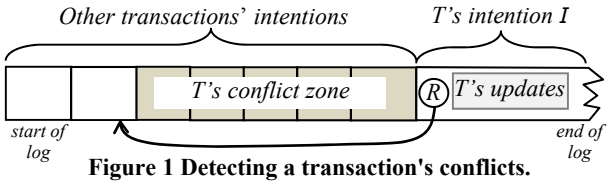

Figure 1 Detecting a transaction's conflicts.

Meld is an optimistic concurrency control algorithm. It determines if any committed transaction in *T*'s conflict zone conflicts with *T*. If not, meld decides to commit *T* and merges *T*'s updates into the locally-cached partial-copy of the database. If there is a conflict, meld decides to abort *T* and discards its updates. Meld then acknowledges *T*'s commit or abort, if *T* executed at the same server as meld. The definition of conflict can include read-write, write-write, or phantom conflicts, depending on the isolation level that *T* requires. This supports all of the isolation in the SQL standard.

At an architectural level, there are several main potential bottlenecks in a system that uses meld: the number of aborts caused by conflicts, the rate at which servers can append to and retrieve from the log, and the speed of the meld operator. The number of aborts largely depends on application behavior. The rates that the log can be written and accessed are moving targets that will improve over time, as the underlying technology speeds up. By contrast, the speed of meld will not benefit from technology improvements for the foreseeable future. Meld is inherently a sequential algorithm. Although parts of it can be parallelized, it ultimately must process log records in log sequence. It therefore depends on the performance of a single processor core, which is not expected to speed up very much in the coming years.

It is important that meld be very fast, for two reasons. First, the update-transaction rate of the entire multi-server system is limited by the rate that meld can process intention records. Second, the longer it takes to meld a transaction's intention, the greater the number of intentions in each transaction's conflict zone and hence the greater the chance that each transaction aborts.

Meld attains its high performance by minimizing the number of nodes it has to examine to check for a conflict. Rather than scanning the intention's conflict zone to determine whether the transaction experienced a conflict, meld maintains enough metadata in the last committed state to detect conflicts accurately. By leveraging that metadata and the database's tree structure, it can often

† Work performed while employed at Microsoft Corporation

Proceedings of the VLDB Endowment, Vol. 4, No. 11

determine the absence of conflicts without comparing all readsets and writesets of transactions. This innovation speeds up transaction processing even if meld runs on only one server. Thus, meld is potentially useful for any system that uses optimistic concurrency control, not just one for shared storage.

The main contributions of our paper are a highly optimized algorithm for meld and detailed measurements of its implementation.

Meld was initially developed for the Hyder system, which is designed to run on a cluster of servers that have shared access to a large pool of network-addressable storage, ideally raw flash chips. Hyder relies on meld to scale out without partitioning. The basic meld algorithm is sketched briefly in [4]. This paper extends that sketch by presenting the entire algorithm, including many optimizations. These optimizations are essential to attain high transaction rates comparable to those of locking systems—mechanisms that have been optimized over the last 40 years.

Hyder's performance is analyzed in [4], showing how far it can scale out and under what conditions. That analysis takes the speed of meld as an input parameter. This paper shows what values can be expected for that parameter. We present performance measurements of our meld implementation different loads.

The paper is organized as follows. The basics are covered in Section 2 on the search-tree index structure and Section 3 on a skeletal version of the meld algorithm itself. These sections are taken from [4]. The rest of the paper is new material. Section 4 explains the detailed data structures needed to guide conflict detection by the meld algorithm. Section 5 tells how to meld concurrent transactions. Section 6 covers delete operations. Section 7 presents the control structure that drives the mechanisms in Sections 3-6. Section 8 presents an evaluation of our implementation. Section 9 covers related work, and Section 10 is the conclusion. The appendix discusses checkpointing, recovery, transaction execution, and garbage collection, and presents meld pseudo code and additional performance measurements.

2. INDEX STRUCTURE

Meld operates on a database that is structured as a search tree. Each node n is a [key, payload] pair (denoted **key(*n*)** and **payload(*n*)**). The system maintains a node cache of recently accessed parts of the tree. Its basic operations are get, insert, delete and update of nodes and ranges of nodes, identified by their keys.

Most relational databases are stored as trees. The top of the tree is the database catalog, which identifies the databases. Each node that represents a database has immediate descendants that represent subschemas, whose descendants represent tables. Each table has descendants that represent rows, which are [key, payload] pairs, where the payload contains the non-key attributes. Secondary indices can also be implemented as tables, where each row's key is the secondary index key and its payload is a primary key. For example, this approach is used in Microsoft SQL Server.

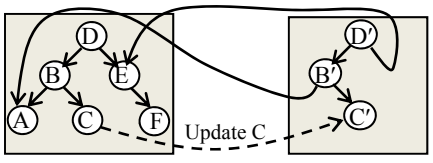

Figure 2 Copy-on-write in a binary search tree.

For the purpose of this paper, the database is stored as a binary search tree (see Figure 2). Since the database is multiversioned, to modify a node n, a new copy n' of n is created. Since n's parent p needs to be updated with a new pointer to n', a new copy p' of p is needed. And so on, up the tree (again, see Figure 2). Notice that D′ is the root of a complete tree, which shares the unmodified parts of the tree with its previous version, rooted at D.

An updated tree is marshaled into a transaction's intention and appended to the log (Figure 3). The details of how a transaction generates its intention are described in Appendix E.

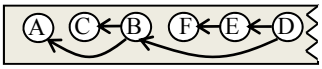

Figure 3 The tree of Figure 2 marshaled into a log.

For good performance, it is important to minimize intention size. This makes binary trees a good choice. A binary tree over a billion keys has depth 30. By contrast, a B-tree over a billion keys with 200-key pages has depth 4. With raw flash, SSDs or shingled disks, B-tree pages cannot be updated in-place. Thus, an update of a B-tree leaf creates a new version of four pages on the root-to-leaf path, which consumes space for 800 keys, much more than a 30-node path. Still, B-trees with low fanout and prefix and suffix key-compression can be competitive. They may be preferable to reduce random seeks, especially for sequential access in key-order. The meld algorithm presented here can be adapted for B-trees and for search trees where payloads reside only in leaves.

3. BASIC MELD

This section presents a simplified description of meld that introduces the main concepts. Later sections expand on it with features that improve the accuracy and speed of conflict detection.

Meld has two inputs: an intention record I and **the last committed state** (**LCS)** of the database, which resulted from processing the last committed transaction that preceded I. Intention I contains a binary search tree with all of the nodes that I's transaction (denoted **T(*I*)**) inserted or modified and their ancestors. This includes deletions, since deleting a node requires modifying its parents. It also includes some metadata about each node, such as the previous version and a flag indicating whether T(I) depends on the previous version being unchanged. When a strong isolation level is used, the intention also contains nodes that T(I) read. Meld returns either an indication that T(I) experienced a conflict or, if not, the new last committed state, which includes *I's* updates. In what follows, we do not make a sharp distinction between I and T(I) when the distinction is clear from context.

A **serial intention** is an intention whose conflict zone is empty. That is, its snapshot is the state in the intention that immediately precedes it in the log. In that case, meld is trivial, since the intention's root defines its output's LCS. This case arises only when the update load is very light—when the transaction inter-arrival time is more than the end-to-end processing time of a transaction.

A non-serial intention is called **concurrent.** Melding a concurrent I is more complex, because LCS includes updates that were not in *I's* snapshot. Meld must check that these updates do not conflict with I. If they do not, then it must merge *I's* updates into LCS. That is, it cannot simply replace LCS by I, as in the serial case.

For example, suppose transaction T_1 executes on an empty database, inserting nodes B, C, D, and E. (See Figure 4.) Then transactions T_2 and T_3 execute on T_1's output. T_2 inserts node A, and T_3 inserts F. T_2 and T_3 do not conflict, so after melding them LCS should include both of their updates.

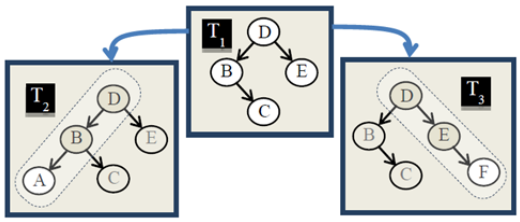

Figure 4 Example transactions to be melded.

Each node n in an intention I has a unique **version number** (**VN**), a **source content version** (**SCV**), and a flag **DependsOn**. For now, assume that the source content version refers to the version of the node in I's snapshot. The **DependsOn** flag is TRUE if I depends on n not having changed during T(I)'s execution, that is, because T(I) read n and ran with repeatable read or serializable isolation level. We denote node n in I_j by n_j, and its VN, SCV, and DependsOn flag by VN(n_j), SCV(n_j), and DependsOn(n_j) respectively. Similarly, VN of node n in LCS is denoted VN(n_{LCS}).

The need for DependsOn arises in part because some nodes in I were updated only because a descendant was updated. For example, in Figure 2, B′ was updated only because C′ was updated. In this case, DependsOn(C′) = TRUE and DependsOn(B′) = FALSE.

VN, SCV, and DependsOn enable meld to detect conflicts. If SCV(n_I) ≠ VN(n_{LCS}), then a committed transaction modified n while T(I) was executing. In this case, if DependsOn(n_I) = TRUE, then T(I) experienced a read-write conflict and should abort.

This is an oversimplification, because n may have been updated (and hence SCV(n_I) ≠ VN(n_{LCS})) only because one of n's descendants was updated, not because its content changed. Such an update may not be a conflict. We enrich the definition of SCV in Section 4.2 to avoid such false positives.

Suppose transactions T_1-T_3 from Figure 4 are sequenced in the log as shown in Figure 5**.** Meld processes the log as follows:

1. Meld deduces that T_1 is serial, because SCV(D_1) = 50 = VN(D_{LCS}), which means T_0 immediately precedes T_1. So it melds T_1 by returning a new LCS whose root is D_1, where VN(D_1) = 54.
2. Similarly, meld deduces that T_2 is serial and returns a new LCS whose root is D_2, where VN(D_2) = 57.
3. For T_3, meld sees that SCV(D_3) ≠ VN(D_{LCS}) (i.e., 54 ≠ 57). Since DependsOn(D_3) = FALSE, these unequal VN's do not indicate a conflict. However, a descendant of D_3 might depend on a node in LCS that was updated in T_3's conflict zone. So meld has to drill deeper into I_3 by visiting E_3.
4. Meld sees that SCV(E_3) = VN(E_{LCS}) = 53. Thus, the subtree rooted at E did not change while T_3 was executing. So there is no conflict and meld can declare T_3 as committed.

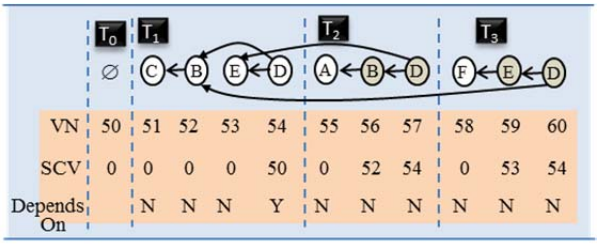

Figure 5 Log of the transactions of Figure 4.

In (4) above, meld truncates the traversal of I_3 before touching F. This traversal truncation occurs whenever meld encounters an intention I's subtree that did not change while T(I) was executing. This optimization significantly reduces the time to meld I compared to a naïve algorithm that tests every node in I for a conflict.

Now that meld knows that T_3 committed, it has to merge T_3's updates into LCS. Unlike the serial cases of T_1 and T_2, it cannot simply return D_3 as the root of the new LCS, because SCV(D_3) ≠ VN(D_{LCS}). This means that LCS includes updates that are not in I_3, namely, T_2's insertion of node A. Therefore, meld must create a new copy of D that points to B_2 on the left and E_3 on the right.

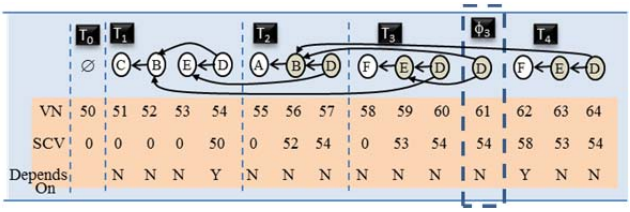

Figure 6 Adding ephemeral intention ϕ_3 and T_4 to Figure 5.

Since every node must reside in some intention, meld creates a new intention that contains the new D (see ϕ_3 in Figure 6). This is called an **ephemeral intention**, because it exists only in memory. It is uniquely associated with the transaction that caused it to be created, in this case T_3, and logically commits immediately after T_3. In this case, it has just one node D, but in general it can have many nodes. We discuss ephemeral intentions in Section 5.

Notice that a server needs to cache only paths in the tree that meld accessed for recent transactions, not the whole database tree.

4. DATA STRUCTURES

4.1 Version Numbers and Pointers

The meld algorithm requires that versions of nodes with the same key have distinct version numbers. To meet this requirement, it is tempting to store a version number with each node in the log and increment it in each new version of the node. However, this does not work, since concurrent transactions can independently generate successor nodes to a given version. Moreover, it consumes log space, which reduces the rate at which the log is read and written, and hence reduces transaction throughput. For these reasons, version numbers of new nodes are not explicitly stored in the log. Instead, they are calculated lazily during meld as a function of the location and number of updated nodes in each intention.

Meld assigns a **commit sequence number** (**CSN**) to each committed intention I**.** CSN(I) equals the CSN of the intention that generated LCS into which I is melded, plus the number of nodes in I. For example, in Figure 5, CSN(I_1) = CSN(I_0) + 4 = 54, CSN(I_2) = CSN(I_1) + 3 = 57, and CSN(I_3) = CSN(I_2) + 3 = 60.

For each node n in intention I, VN(n) equals CSN(I) minus the zero-origin index of n in I, where nodes are indexed starting with the last node in I and going backward**.** In Figure 5, since CSN(I_3) = 60, we have VN(D) = 60 – 0 = 60, VN(E) = 60 – 1 = 59, etc.

Each node has two child pointers. A pointer to a node n in intention I is stored in the log as a [CSN(I), i] pair, where i is the index of n in I. Since an intention I' often has pointers to multiple nodes in a given intention I, it is worth having I' store just one copy of CSN(I). A pointer to a node in I refers to this copy by its offset in I' with fewer bits than would be needed to refer to CSN(I).

4.2 Metadata to Detect Basic Conflicts

Each node n in intention I includes VN(n), DependsOn(n), SCV(n), an **Altered** flag, a **previous version number** (**PVN**) and a **new content version** (**NCV**). PVN is used here only to help explanations; it is not used by the algorithm. We explained VN(n) and DependsOn(n) earlier. We explain the other metadata here.

We define **PVN**(n) to be the version number of the node in I's snapshot with the same key as n. If n is newly inserted, then it has no predecessor. We define **SCV(n)** to be the version number of the node that first generated the payload in node(PVN(n)), where node(PVN(n)) is the node whose VN is PVN(n). For all nodes n in Figure 5, SCV(n) = PVN(n), but in general this need not be the case. Suppose SCV(n) = PVN(n) in I, and consider a node n' in the next intention I' where key(n') = key(n) and where n' has the same payload as n. That is, n' is in I' only because one of its descendants was modified. In this case, SCV(n') = SCV(n). For example, in Figure 6, suppose we add a transaction T_4 that updates node F. Like I_3, T_4's intention I_4 stores the path D→E→F. The value of SCV(F) in I_4 (i.e., SCV(F_4)) is 58, because F was updated in I_3. However, SCV(E_4) ≠ 59. Instead, SCV(E_4) = SCV(E_3) = 53, because E_4 has the same payload as E_3, which came from E_1, where VN(E_1) = 53.

The flag Altered(n) is true if n's payload is modified in the intention. The field NCV(n) is calculated from SCV(n) and Altered(n). If Altered(n) is true then NCV(n) = VN(n). Otherwise NCV(n) = SCV(n). That is, NCV(n) defines SCV for the successor of n. The definitions of SCV and NCV enable more precise conflict checking than the one in Section 3. It ensures that SCV(n_I) ≠ NCV(n_{LCS}) if and only if a committed transaction modified n's payload while T(I) was executing.

4.3 Metadata to Detect Phantoms

To support phantom detection, each node n has a flag **DependsOnTree** and optionally a **DependencyRange**. DependsOnTree(n) is TRUE if T(I) depends on **the subtree rooted at node(PVN(n))** (**subtree(PVN(n))**) not having changed during T(I)'s execution, because T(I) read nodes in subtree(PVN(n)) and ran with repeatable read or serializable isolation. That is, T(I) should abort if subtree(PVN(n)) ≠ subtree(n_{LCS}). If DependencyRange is absent, then T(I) depends on the entire key range of subtree(n). If DependencyRange is present, then it identifies the key range in subtree(n) that T(I) depends on.

A node d with DependsOnTree(d) = TRUE might be far from I's root. With a little extra bookkeeping, meld can avoid traversing the entire path to d by detecting if d has an ancestor n in I whose subtree was unchanged in I's conflict zone. To do this, each node n in I has a flag **SubtreeIsOnlyReadDependent**, which is TRUE iff none of its descendants are updated (i.e., have Altered = TRUE). That is, n's subtree is present in I only due to nodes (like d) that are read-dependent on I's snapshot. In addition, n has a **source structure version** (**SSV(n)**), which is the oldest version of n that has exactly subtree(PVN(n)). It also has a **new structure version** (**NSV(n)**) which is calculated: If SubtreeIsOnlyReadDependent(n) = TRUE then NSV(n) = SSV(n). Otherwise NSV(n) = VN(n). It is the oldest version of n whose subtree is exactly subtree(VN(n)).

Meld uses the above metadata when processing a node n in I and comparing it to node m in LCS with the same key as n. If SSV(n) = NSV(m), then meld stops traversing the subtree with root n in I.

Notice that SubtreeIsOnlyReadDependent, SSV, and NSV are exactly analogous to Altered, SCV and NCV. The difference is that the former track any change in a node's subtree, while the latter track changes in a node's content.

If a secondary index is stored as a table as described in Section 2, then DependsOnTree can be used to detect changes in key ranges just like for a primary index. Thus, the phantom detection technique of this section applies to secondary indices too.

In our current implementation, with some compression techniques (e.g., the one mentioned in Section 4.1 and variable-length integer encoding), the average metadata size per node that needs to be stored in the log can be less than 30 bytes.

5. EPHEMERAL INTENTIONS

Every committed concurrent (i.e., non-serial) intention I needs to be merged with LCS, which generates an ephemeral intention, such as ϕ_3 in Figure 6. It is created deterministically by all servers that execute meld and is sequenced immediately after the concurrent intention that triggered it. It therefore has a unique well-defined CSN. Thus, even though it exists only in main memory and not in the log, its nodes (called **ephemeral nodes**) can be referenced unambiguously by later transactions.

A node n in I need not generate an ephemeral node if SubtreeIsOnlyReadDependent(n) = TRUE. In that case, since no nodes in subtree(n) were updated, the ephemeral node generated for n's parent can point to n_{LCS}. However, meld must continue to traverse down the tree until it determines whether or not there is a conflict.

Ephemeral intentions are problematic when memory is limited. An ephemeral intention is generated every time a concurrent intention is melded. We do not allow an ephemeral intention I_ϕ to be discarded if it is reachable in LCS, since a future transaction may access it and it exists only in main memory. Although I_ϕ can be re-created by re-melding the concurrent intention I_C that created it, it may be rather expensive to regenerate I_ϕ since I_C might no longer be in main memory at regeneration time.

Fortunately, the normal operation of the system tends to trim ephemeral nodes. For example, in Figure 6, T_4 causes D in ϕ_3 to be replaced. Since T_4 is serial, it replaces D in ϕ_3 by D in I_4. Once meld commits I_4, the ephemeral node D in ϕ_3 is no longer reachable in future committed states. If T_4 were concurrent because another transaction committed in between T_3 and T_4, then meld would create a new ephemeral intention with a copy of D, but the total amount of ephemeral state would not increase. Updates that contend on the same regions of the tree, such as the root in this case, tend to create ephemeral nodes in those regions. But intentions with updates on those regions tend to trim ephemeral state. So trimming tends to clean up ephemeral state over time as the ephemeral nodes are copied into subsequent intentions.

Although the meld process tends to rapidly make most ephemeral nodes unreachable from LCS, some ephemeral nodes may still remain reachable. These ephemeral nodes can be tracked, for example, by storing an ephemeral flag indicating whether each subtree contains any ephemeral nodes. A more elaborate method can track the number ephemeral nodes or the maximum number of pointers that must be traversed to reach the ephemeral nodes.

The ephemeral flag enables efficient enumeration of all reachable ephemeral nodes. A flushing mechanism can periodically execute a transaction T that does nothing but make a copy of a set of eph-

emeral nodes that together have no reachable ephemeral nodes. It appends T's intention, called a **flush intention**, to the log.

A flush intention has no dependencies, so it will not experience a conflict. Hence, the regular meld process commits this intention. This makes the original ephemeral nodes unreachable in the new committed state, effectively replacing the original ephemeral nodes with the flush intention's persisted nodes. A new ephemeral intention is normally created as a result of melding the flush intention, but its size is based only on the intentions that intervened during the flush. Since the number of intervening intentions for each concurrent flush intention tends to be fairly stable for a given workload, flushing stabilizes the amount of reachable ephemeral state.

6. DELETIONS

The deletion of a node n in intention I needs to be recorded in the tree for two reasons. First, depending on the semantics of the delete operation, an update or delete of n in I's conflict zone might require meld to abort I. Second, I's deletion affects meld's conflict detection of any concurrent transaction whose conflict zone contains I and that read n, updated n, or scanned a range of nodes that includes n. We considered three possible approaches.

One approach to deleting a node n is to include n in I's tree as a **tombstone**, which is marked as deleted. Although the use of tombstones enables accurate conflict detection of deletions, it consumes space and requires mechanisms to track tombstones and eventually remove them from the tree. Further details are in Appendix B on Garbage Collection.

A second approach is to actually delete node n in I's tree (which still requires including n's parent p in I). This requires adding metadata that ensures that meld will detect conflicts on n for two cases: (i) in I's conflict zone and (ii) in a later transaction I' whose conflict zone includes I. For case (i), since n is absent from I, meld will not detect a conflict with a transaction in I's conflict zone that updated n. To fix this, we can add a DependsOnTree flag to nodes (not just to pointers as in Section 4.3), and set it to TRUE for p (i.e., for any node whose child is deleted). Given this flag, meld will abort I if anything in p's subtree changed in I's conflict zone, such as an update to n. Unfortunately, it will abort I if there are other changes in p's subtree that do not conflict with I. This avoids the complexity of tracking and deleting tombstones, but it increases the chance of a false conflict.

For case (ii), suppose I' read or wrote n. Then DependsOn(n) = TRUE in I'. If no committed transaction re-inserted a node with n's key in between I and I', then when processing I', meld will not find n in LCS and hence will abort I'. If a committed transaction did re-insert a node with n's key in between I and I', that node's NCV will be greater than SCV(n) in I', which meld will detect as a conflict. Now suppose I' inserted a node with n's key. Then as for a write of n by I', meld will detect a conflict if a committed transaction re-inserted a node with n's key in between I and I'.

For case (ii), now suppose I' deleted n. Given that I already deleted n, this a no-op. Meld will not detect this as a conflict, since n is absent from I' and I. This is treated like case (i), that is, both intentions should set DependsOnTree(p) = TRUE and Altered(p) = TRUE, which ensures meld will detect this as a conflict.

A third approach, which is used in our current implementation, combines the other two. It deletes nodes in I's tree and adds the list of deleted nodes to I. The latter are essentially tombstones, but they are not in I's tree. Case (i) is therefore handled by checking that the SCV of each node in the delete list is unchanged in LCS. Case (ii) is handled as in the second approach.

7. THE MELD ALGORITHM

Meld is a recursive algorithm that traverses the intention record I, comparing its nodes to LCS. Each recursive call takes a node n in I and a node n_L in LCS as input. It returns an indication of a conflict, and if there is none, the root n_L' of a subtree to replace n_L in LCS that includes I's results in the subtree's key range. Either $n_L' = n$ or n_L, or n_L' is a newly-created ephemeral node.

For each node n in the intention, meld looks for the node n_L in LCS with the same key as n. It starts at the root of I and of LCS. If their keys are equal, we call it the **symmetric case**, because I and LCS have the same structure. In this case the recursion is simple. Since n's left subtree covers the same key range as n_L's left subtree, meld can be called recursively on n's left child and n_L's left child. Similarly for the right children.

Insertions, deletions, or rotations by committed transactions in I's conflict zone can cause the structure of subtrees of I and LCS to diverge. This is the **asymmetric case**, and it complicates the recursion. We explain the algorithm for each case in turn.

7.1 Symmetric Case

Consider a call to meld on node n in I and n_L in LCS where key(n) = key(n_L). If SubtreeIsOnlyReadDependent(n) = TRUE, then meld recursively performs a special meld process on n's subtrees. It is basically the same as normal meld except that it does not create ephemeral nodes, but only detects conflict. When the special meld returns back to the current level, the meld simply returns n_L.

If SSV(n) = NSV(n_L), then n_L's subtree in LCS is unchanged since I executed. That is, no committed transaction in I's conflict zone modified a node in n_L's subtree. Therefore (the recursive call to) meld simply returns n. This is the serial case, for a subtree of I rooted at n. Since it uses NSV(n_L) instead of VN(n_L), it leverages the read-write optimization described at the end of Section 4.3.

If SSV(n) ≠ NSV(n_L), then n_L's subtree in LCS changed after I's snapshot, which is the case of a concurrent meld. So meld checks for a conflict between n and n_L, and if there is none, it creates an ephemeral node and continues to recursively meld n's subtrees. There are several conflict checks, which we covered earlier:

- If Altered(n) = TRUE and SCV(n) ≠ NCV(n_L), then there is a write-write conflict, so abort.
- If DependsOn(n) = TRUE and Altered(n_L) = TRUE, then there is a read-write conflict, so abort (under serializable isolation).
- If DependsOnTree(n) = TRUE, then there is a read-write conflict. Since SSV(n) ≠ NSV(n_L), something in subtree(n_L) changed.

If none of the above conflicts are detected, then meld recursively melds n's subtrees as follows:

1. Create a new ephemeral node n_ϕ for n.
2. Concurrently do the following for the left and right child of n
   a. If the left (resp. right) child of n is in another intention or is NULL, then the to-be-melded intention did not update that subtree. Therefore, assign the left (resp. right) child of n_L to the left (resp. right) child of n_ϕ.
   b. Otherwise, the to-be-melded intention updated that subtree. Therefore, recursively call meld on the left (resp. right) children of n and n_L. If it returns a conflict,

then return a conflict and exit. Otherwise, it returns a node n'. Assign n' to the left (resp. right) child of n_ϕ.

3. SSV(n_ϕ) = NSV(n_L); Altered(n_ϕ) = FALSE;
4. SubtreeIsOnlyReadDependent(n_ϕ) = FALSE;
5. If Altered(n), then {
   Payload(n_ϕ) = Payload(n); SCV(n_ϕ) = NCV(n); }
   else {Payload(n_ϕ) = Payload(n_L); SCV(n_ϕ) = NCV(n_L);
   If (NSV(left child of n_ϕ) = NSV(left child of n_L) and
   NSV(right child of n_ϕ) = NSV(right child of n_L))
   SubtreeIsOnlyReadDependent(n_ϕ) = TRUE; }
6. Return n_ϕ

7.2 Asymmetric Case

Consider a call to meld on node n in I and n_L in LCS where key(n) ≠ key(n_L). Since the trees are not aligned, we can no longer rely on the range of keys in the two subtrees being the same. Thus, we cannot simply meld the **left subtree of *n*** (**leftSubtree(*n*)**) with leftSubtree(n_L), since in general the subtrees cover different key ranges. (And the same for the right, of course.) For example, if $n < n_L$, leftSubtree(n_L) may have keys greater than key(n). Some of those keys may be in rightSubtree(n), and they too must be melded with leftSubtree(n_L) (see Figure 7).

To handle this, we add a third parameter to meld that identifies the key range being melded. In the above example, we meld n with leftSubtree(n_L), but only for the range less than key(n_L) (which does not include all keys in rightSubtree(n)). We then find a node d in rightSubtree(n) that has all nodes in subtree(n) with keys greater than key(n_L). We meld d with rightSubtree(n_L), but only for the range greater than key(n_L) (since leftSubtree(d) may have nodes with keys less than key(n_L)).

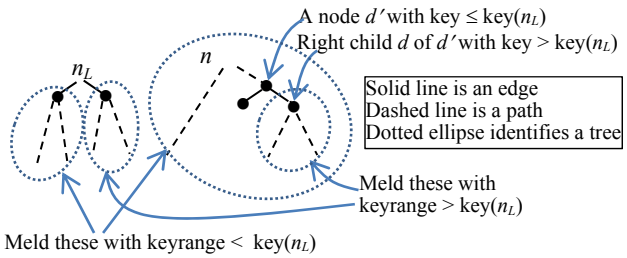

Figure 7 Splitting ranges during recursive meld.

As the recursion moves down the tree, it needs to keep track of the key ranges. Each recursive call splits the key range it was given as input and chooses subtrees to meld for each partition. Given that meld is working on a key range of [low, high] based on its input parameter, it makes two recursive calls on subtrees. One is on leftSubtree(n_L) and n with a key range of [low, key(n_L)). The other is on rightSubtree(n_L) and d with a key range of (key(n_L), high], where d is found by starting at n and traversing right children until encountering the first node d with a key greater than key(n_L).

Notice that the recursive calls are made on half-open intervals that do not include key(n_L). Before the recursive calls, meld looks for a node n' in subtree(n) with key(n') = key(n_L). If it finds such an n' it checks for conflicts between n and n_L as for the symmetric case. If there is no conflict it creates an ephemeral node for n'.

Each recursive call, with nodes n and n_L and *keyrange* as input, returns a node n_L' that is a copy of n_L with the following changes:

1. For every node m in subtree(n_L), if there is a node n' in subtree(n) with key(n') = key(m) and Altered(n') = TRUE, then replace m by n' in subtree(n_L').
2. For every node n' in subtree(n), if there is no node m in subtree(n_L) with key(m) = key(n') and key(n') ∈ *keyrange*, then add n' to subtree(n_L').
3. If m is in subtree(n_L), then m is in subtree(n_L'), no matter whether there is a node n' in subtree(n) with key(m) = key(n'), unless I deleted it.

In (1), I updated n'. In (2), I inserted n'. In (3), I deleted n'.

It is easy to rebalance the index tree in a recursive manner in the meld algorithm which is recursive by itself. To do so, each node just needs to maintain the depth of the subtree rooted at it.

See Appendix C for more detailed pseudo-code of the algorithm.

8. EXPERIMENTS

We implemented a prototype of meld with a transactional key-value store in C++ on the Windows platform. It contains a log simulator and a client that consists of a transaction executor, intention dispatcher, and meld operator. The dispatcher receives each appended intention record and passes it to the meld operator.

In a distributed configuration, the dispatcher receives each intention from the network. In this case, it needs to unmarshal the intention into an in-memory tree structure before meld can operate on it. This task, called **pre-deserialization**, uses significant processor time. Although meld can perform this task, it is better if the dispatcher does it and executes it in parallel with meld. In fact, many intentions can be deserialized in parallel by employing concurrent intention dispatchers executing on different cores. We found this shortened meld's execution time by up to 45%.

We evaluate the performance of meld with synthetic benchmarks. These benchmarks have a simple, regular structure that enables us to explain the behavior we are witnessing (unlike benchmarks with mixed transaction load, such as TPC-C/E). Each benchmark is a sequence of transactions consisting of multiple key-value operations: read, update, insert of key-value pairs identified by keys. All of the transactions access an initial database table containing 128K key-value pairs where keys and values are both 8-byte strings. Every transaction requires serializable isolation; hence its intention record contains the data items it reads. The only effect of weaker isolation levels on meld throughput is to reduce the number of nodes traversed by dropping readsets from intentions; we cover this as a read-% of zero in Section 8.2. We use a single client to process the transaction sequences in our experiments in order to precisely control the behavior of the benchmarks.

During the generation of transaction sequences, we control the number of operations per transaction (or transaction size), the ratio among different operation types, and the number of concurrent transactions in the conflict zone of each transaction (which we call the **concurrency degree**). We believe these factors reflect the behavioral characteristics of many types of applications.

All experiments are performed on a 4-core, 8-thread Intel Xeon x5550 2.67GHz, with 4×256KB L2 cache and 8MB shared L3 cache, 12GB main memory, running Windows Server 2008 R2. The meld thread is always assigned to a dedicated CPU thread.

A study of the effect of database caching and speed of reading and writing the log is in [4] and is beyond the scope of this paper.

Therefore, the entire database table and all the appended intention records are main-memory resident to avoid any I/O during meld.

We use meld latency and throughput as the main performance metrics. The latency of each meld is its execution time and the meld throughput is simply the reciprocal of the average meld latency. The measured meld performance does not include the cost of pre-deserialization, since it is off the critical path.

8.1 Symmetric Case

To measure the symmetric case of meld, we use benchmarks consisting of only read and update operations. Figure 8(a) shows the average meld latency relative to the number of operations per transaction. Figure 8(b) shows the average throughput in melds (i.e., transactions) per second. The ratio between read and update operations is 1:1, and keys in the table are accessed randomly with a uniform distribution. The different curves in the figures are for experiments with different concurrency degrees; the series **con-d*i*** represents an experiment with concurrency degree *i*. Since we mainly focus on the performance of the meld algorithm in this paper, the experiments do not distinguish between aborted and committed transactions. Abort rates are given in Appendix F.

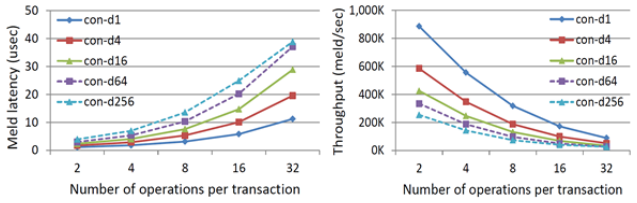

Figure 8 Meld Latency and Throughput vs. Transaction Size.

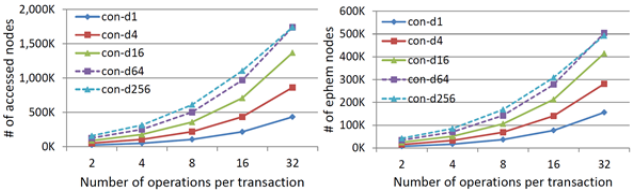

Figure 9 Number of Node Accesses and Created Ephemeral Nodes vs. Transaction Size.

As shown in Figure 8(b), meld can do more than 400K update transactions (i.e., melds) per second (TPS) when transaction size is 2 and concurrency degree is ≤ 16. It performs more than 100K TPS when transaction size is ≤ 8 and concurrency degree is ≤ 16.

The performance of meld decreases with an increasing number of operations per transaction. The reason is that the more operations in each transaction, the more keys each transaction accesses, and hence the more probable that concurrent transactions access keys with common ancestors in the index tree. This forces meld to traverse deeper in the tree to detect conflicts, and hence results in more nodes in the intention subtree and LCS being accessed and more ephemeral nodes being created during meld, which, we believe, are the major contributors to meld latency. Figure 9 confirms this. The curves are very consistent with those of average meld latency in Figure 8(a). For a similar reason, meld performs worse when the concurrency degree increases.

The number of operations per transaction is purely an application behavior characteristic. By contrast, the concurrency degree can be affected by many factors, such as the rate of transaction arrivals, the speed of local transaction execution, and the speed of meld. Moreover, concurrency degree and meld performance can interact with each other. Increasing the concurrency degree makes meld slower, which increases the conflict zone size of executing transactions and hence can increase the concurrency degree even more. This happens when the system is overloaded, which suggests that the system should avoid starting new transactions when the concurrency degree exceeds some threshold.

8.2 Optimization on Read-Only Subtree

We next evaluate meld performance with different read-to-update ratios. This ratio affects meld performance due to the SubtreeIsOnlyReadDependent optimizations on read-only subtrees (cf. Section 4.3 and 5), clearly an important case in many applications.

Figure 10 shows meld throughput versus the fraction of operations that are reads. The series **con-d*i*-#op*n*** represents an experiment with concurrency degree *i* and transaction size *n*. Meld performs better when reads predominate, demonstrating the effectiveness of the optimization. Without it, meld performance should be unaffected by different read-update ratios and be very close to its performance in the case when all transactions are update-only.

In the figure, when the transaction size is 2, more than 60% of operations are reads, and concurrency degree is 16 or less, meld achieves over 400K TPS. Even at concurrency degree 64, meld can still do over 400K TPS when 80% of operations are reads.

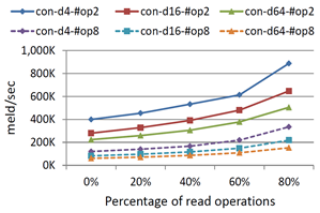

Figure 10 Throughput vs. Read Percentage.

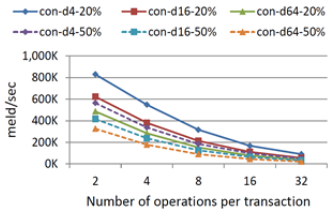

Figure 11 Throughput vs. Transaction Size for Asymmetric Case.

8.3 Asymmetric Case

The asymmetric case incurs more overhead than the symmetric case since it has to use the key of an LCS node at the current tree depth to split the corresponding intention subtree and may introduce rebalancing. To measure asymmetric case, we use benchmarks consisting of only read and insert operations. We omit the evaluation results for the asymmetric case when deletes are present, since they are similar to the case with only reads and inserts.

In the experiments, the intervals between every two adjacent keys in the initial database table are the same. The read operations access the key-value pairs in the initial table with a uniform random distribution. The insert operations insert key-value pairs with unique keys that are randomly distributed among the intervals between keys in the initial table, also uniformly distributed. Figure 11 shows meld throughput corresponding to different transaction sizes. The series **con-d*i*-*x*%** represents an experiment where concurrency degree is *i* and *x* percent of the operations are inserts.

As in the symmetric-case experiments, meld performs worse when transaction size increases. However, the asymmetric case introduced through inserts does not significantly degrade meld performance relative to the corresponding symmetric case. For example, compare a read-insert ratio of 1:1 in the asymmetric case to a read-update ratio of 1:1 in the symmetric case. Only when the

concurrency degree is 64 and transaction size is 32 is the asymmetric case slower than the symmetric case, and only by 22%. When the concurrency degree is < 64 and transaction size is < 32, meld throughput for the asymmetric case is < 10% lower.

We observed that inserts do not increase the number of accessed nodes very much compared to the symmetric case (< 20%). This is because there are not enough operations of committed transactions in the conflict zone of each transaction to significantly change the structure of the index tree, and in most cases, meld can return early by grafting LCS or intention subtrees when no asymmetry is encountered.

In summary, meld performance is excellent for small transactions and very good for transactions with up to 8 operations. Since concurrency degree can degrade meld performance, systems using meld should keep it as low as possible. This can be achieved by load control and optimizing local transaction execution. Meld is friendly to reads and not very sensitive to inserts and deletes.

9. RELATED WORK

Due to their optimization of write-intensive workloads on traditional disk and flash memory, log-structured storage techniques have been widely investigated [8][14][15][20]. Their optimizations are mainly for batching updates to achieve large sequential writes. Some of these mechanisms merge the batched updates into the latest snapshot of the storage; however, they do not perform concurrency control, and hence are different than meld.

Optimistic concurrency control (OCC) was originally proposed in [11]. Its benefits and tradeoffs have been extensively explored in [1][2][10][13][16][18][19]. Some of these mechanisms combine pessimistic and optimistic methods [13][16][18]. Their techniques are orthogonal to ours. Compared to other pure OCC schemes, a major uniqueness of meld is its ability to determine the absence of conflicts without processing the entire read and write set.

Tashkent [5] proposed to avoid separating the durability and commit ordering of update transactions. Their system uses centralized conflict testing with OCC. But they give no details on fine-grained conflict checking, which is not their focus. An OCC-based distributed B-tree system is described in [3]. It uses single-version data and simple version numbers to detect conflicts.

The rich literature of locking protocols over tree-structured data can be found in books on transaction processing, such as [9]. Locking protocols for XML trees are compared in [11].

As far as we know, our meld algorithm is an entirely new way to perform OCC. The only similar work we know of is OXenstored in [7]. That algorithm works on multiversion tries [6] and generates intentions much like ours. It requires that the smallest sub-trie S of the intention I that contains all of the transaction's updates was not modified in I's conflict zone. This is a much coarser-grained conflict test than ours, since another transaction's update to a node of S not referenced by I still causes I to abort. They do not discuss any of the issues in Sections 3-7, such as optimizing an intention's representation, supporting different isolation levels, or optimizing detection of read-write or delete conflicts.

10. CONCLUSION

This paper introduced the meld algorithm, a new technique for optimistic concurrency control over trees. Meld operates on a log of transaction intentions that it treats as a multiversion database. It determines which transactions committed by detecting the presence of conflicting updates by committed transactions in the range of log records between the snapshot the transaction read and its intention. Meld merges the updates of each committed transaction into a cached partial-copy of the last committed database state.

We described the meld algorithm in detail, including many of its optimizations. We also provided the results of extensive experiments to explore the behavior of our implementation of meld. For transactions with two to ten operations, our meld implementation processes 100K – 400K transactions per second.

There are many research opportunities for further work on meld. An obvious extension is to modify the algorithm for other search tree structures, notably B-+trees and binary trees where payloads are only at the leaves. It would be worthwhile to study its performance on different storage devices, where cache misses affect meld speed, and to compare it to other optimistic and locking algorithms on multi-version trees. In some settings, it may be better to represent an intention by its operations, instead of its after-images, which might lead to a rather different algorithm than the one presented here.

11. REFERENCES

[1] Adya, A., R. Gruber, B. Liskov, and U. Maheshwari: Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *SIGMOD 1995*, pp. 23-34, 1995.

[2] Agrawal, D., A. J. Bernstein, P. Gupta, and S. Sengupta: Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback. *Distributed Computing*, 2(l):45-59, 1987.

[3] Aguilera, M.K., W.M. Golab, and M.A. Shah: A Practical Scalable Distributed B-tree. *PVLDB* 1(1): 598-609, 2008.

[4] Bernstein, P.A., C.W. Reid, and S. Das: Hyder—A Transactional Record Manager for Shared Flash. *CIDR 2011*, pp. 9-20, 2011.

[5] Elnikety, S., S. Dropsho, and F. Pedone. Tashkent: Uniting Durability with Transaction Ordering for High-Performance Scalable Database Replication. *Proc. of EuroSys 2006*, pp. 117-130, 2006.

[6] Fredkin, E.: Trie memory. *CACM*, 3(9):490–499, 1960.

[7] Gazagnairem, T. and V. Hanquez: OXenstored—An Efficient Hierarchical and Transactional Database using Functional Programming with Reference Cell Comparisons. *Int'l. Conf. on Functional Prog. (ICFP 2009)*, pp. 203-214, 2009.

[8] Graefe, G: Write-optimized B-trees. *VLDB 2004*, pp. 672-683, 2004.

[9] Gray, J.N, and A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan-Kauffman. 1992.

[10] Gruber, R.E.: *Optimistic Concurrency Control for Nested Distributed Transactions*. Tech. Report MIT/LCS/TR-453, MIT, June 1989.

[11] Haustein, H.P., T. Härder, K. Luttenberger: Contest of XML Lock Protocols. *VLDB 2006*, pp. 1069-1080, 2006.

[12] Kung, H. T. and J.T. Robinson: On Optimistic Methods for Concurrency Control. *ACM TODS* 6(2): 213-226, 1981.

[13] Lausen, G. Concurrency Control in Database Systems: A Step towards the Integration of Optimistic Methods and Locking. *Proc. ACM Annual Conf. 1982*, pp. 64-68, 1982.

[14] Lee, S-W. and B. Moon. Design of Flash-based DBMS: an In-page Logging Approach. *SIGMOD 2007*, pp. 55–66, 1982.

[15] O'Neil, P., E. Cheng, D. Gawlick, E. O'Neil: The Log-Structured merge-tree (LSM-tree). *Acta Inf.* 33(4): 351-385, 1996.

[16] Phatak, S.H. and B. R. Badrinath: *Bounded locking for optimistic concurrency control*. Rutgers University TR DCS-TR-380, 1999.

[17] Seltzer, M.I. Transaction Support in a Log-Structured File System. *ICDE 1993*, pp. 503-510, 1993.

[18] Sheth A. P. and M. T. Liu: Integrating Locking and Optimistic Concurrency Control in Distributed Database Systems. *ICDCS 1986*, pp. 89–99, 1986.

[19] Thomasian, A. and E. Rahm: A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking. *ICDCS 1990*, pp. 294-301, 1990.

[20] Wu, C-H. L-P. Chang, T-W. Kuo: An Efficient R-tree Implementation over Flash-memory Storage Systems. *ACM GIS 2003*, pp. 17–24, 2003.

APPENDIX

A. CHECKPOINT AND RECOVERY

To recover from a failure, a server needs the root of some committed intention in the log, from which it can start running meld. The log does not distinguish committed from aborted intentions. Therefore, to recover from the log alone, a server would have to replay meld from the beginning of the log. To avoid this, a server needs to periodically checkpoint LCS.

It is tempting simply to write a checkpoint record that points to the last committed intention I known to the server. However, this does not help because I might have pointers to ephemeral nodes, which were created by melding earlier intentions. Moreover, in the common case that I is a concurrent intention, the result of melding I is an ephemeral intention, which of course does not appear in the log.

To avoid these problems, a server S periodically executes a checkpoint transaction T_{ck} that makes a copy of all ephemeral nodes reachable from LCS, as explained in Section 5, and flushes them into the log. T_{ck}'s intention, called a **checkpoint intention**, includes a **restart CSN**. If the root of LCS is persistent, then the restart CSN is that root. Otherwise, LCS is ephemeral and the restart CSN is the CSN of the concurrent transaction that triggered the ephemeral intention that defines LCS.

To recover from a failure, a server first obtains a checkpoint intention, ideally the last one in the log. It can get this from another server or from a well-known location in persistent storage. Or it can find the end of the log as in any log-based recovery algorithm (e.g., [9], pp. 513-514) and scan backwards until it finds a checkpoint intention. Then the server reconstructs the in-memory ephemeral nodes from the checkpoint intention and starts running meld from the CSN of the intention that follows the checkpoint intention's restart CSN.

The system can be initialized by writing a checkpoint intention that includes a root and a restart CSN of zero.

From a correctness standpoint, the normal meld algorithm can ignore the checkpoint record. However, from an efficiency standpoint, it is beneficial for meld to leverage the checkpoint's persistent copies of ephemeral nodes by replacing ephemeral nodes in LCS by the corresponding nodes in the checkpoint, similarly to how it processes a flush intention.

B. GARBAGE COLLECTION

To recycle log space, the system needs to garbage collect old versions of nodes in the log that are still reachable. It does this via two operations, **sweep** and **trim**.

Sweep is a flush transaction that copies nodes that are reachable in LCS and are in the oldest region (i.e., **head**) of the log into a new **sweep intention**. To help sweep find these nodes, we add an **OldestDescendant** field to each node n, which references the oldest node in subtree(n), i.e., the one closest to the log head. When a new version of a node is created, OldestDescendant is set to the minimum OldestDescendant of its two children.

Given a threshold address θ near the head of the log, a sweep transaction T_{sw} starts from LCS's root and traverses the database tree, copying all nodes that precede θ. T_{sw} will not experience a conflict since it has no dependencies or alterations (the details are quite subtle). After meld commits T_{sw}'s intention, no log addresses preceding θ are reachable from LCS.

Trim is an atomic operation provided by the log to truncate the log before a given location so that the physical space can be reclaimed. A server invokes trim at a location that is unreachable from the latest checkpoint snapshot. This guarantees that the untrimmed log space contains at least one checkpoint, so the system is recoverable. To avoid expensive server synchronization, a server can perform trim independently. It should avoid trimming space that is reachable by a committed state that might be the snapshot of an active transaction at any server. Still, a slow server might allow a transaction to access a node located in the trimmed space. Therefore, the log needs to return an exception in response to an attempt to access trimmed space. This can be done if logical log addresses of trimmed space are never reused, e.g., by having logical log addresses increase monotonically and never wrap.

An orthogonal garbage collection issue arises if tombstones are used for deletions, using the first approach suggested in Section 6. A tombstone needs to be retained in LCS until the intentions of all transactions that might conflict with it have already been melded. This can be done by having an upper limit on the size of a conflict zone, where size is measured in number of transactions, nodes, or bytes. Meld aborts every intention that exceeds this limit—a deterministic server-local test. The limit can be set high enough that a transaction with a larger conflict zone is likely to experience a conflict in any case, so the limit does not cause many transactions to abort unnecessarily.

C. PSEUDO CODE OF MELD

In this section, we present detailed pseudo code of the meld algorithm. The algorithm uses data structures for an index tree node and a pointer to it, which are shown in Figure 12.

```
struct TreePointer {
    // Points to the in-memory tree node structure.
    // Is null if no in-memory node exists in the cache.
    // Not stored in the log.
    TreeNode *node;
    // The location of the tree node in the log.
    // Stored in the log.
    SequenceNumber csn;
    uint32 offset;
};

struct TreeNode {
    TreePointer left, right;
    // For garbage collection
    TreePointer OldestDescendant;
    uint8 depth; // For balance information
    NodeVersion VN; // Partially stored in the log
    // The following fields are stored in the log
    NodeVersion SSV;
    NodeVersion SCV;
    bool Altered, DependsOn;
    bool DependsOnTree;
    bool SubtreeIsOnlyReadDependent;
    KeyRange DependencyRange;
    // The following fields are calculated
```

Figure 12 Data structure of node and pointer of index tree.

The following is a brief explanation of four functions and an annotation that are used in the pseudo code:

Meld(): the main function to do meld. It returns **true** if a conflict is encountered.

MeldNoEphem(): Sub-function to meld without creating ephemeral nodes. Its logic is basically the same as **Meld**().

LeftRangeSplit(): Given a key K and key range, return a new key range with all keys less than K.

RightRangeSplit(): Given a key K and key range, return a new key range with all keys larger than K.

[**out**]: An annotation on a parameter that means that the value of the parameter can be changed by the callee function.

For conciseness, the pseudo code of meld that follows does not include details about the handling of deletions.

Algorithm Meld

Input:
lcstree : a pointer to an LCS subtree.
itree : a pointer to an intention subtree or to an LCS subtree directly referenced by a node in the intention subtree.
kr : the current key range. The key of a node in the intention subtree should be melded into the resulting snapshot if and only if it is in this range.

Output:
rtree : a pointer to the resulting subtree.

Return:
true if a conflict is detected, otherwise **false**.

```
1:  if (itree.node == null)
        // itree is empty
2:      rtree = lcstree; return false;
3:  if (itree.node.SubtreeIsOnlyReadDependent)
        // itree is read-only.
        // Meld without creating an ephemeral node
4:      if (MeldNoEphem(lcstree, itree, kr))
5:          return true;
6:      else
7:          rtree = lcstree; return false;
8:  if (lcstree.node == null)
        // Pick up nodes within the key range from itree. Detect
        // possible conflicts, e.g., whether the key of a node
        // in itree but not in lcstree is caused by a deletion of
        // lcstree's transaction and itree's transaction does not
        // insert it. The insertion of itree's transaction for a node
        // can be easily identified by checking whether the
        // node's SCV is valid, i.e., whether the payload of the
        // node has a source.
        …
9:      return false;
10: if (itree points to a node outside of its intention)
        // itree points to a node whose subtree is the
        // same age as or older than the lcs subtree.
11:     rtree = lcstree; return false;
12: if (lcstree.node.NSV == itree.node.SSV)
        // itree's updates are based on lcstree, which was
       // not modified in itree's transaction's conflict zone
13:     rtree = itree; return false;
       // the intention generating lcstree is concurrent
        // with itree's intention
14: if (itree.node.DependsOnTree)
15:     return true;
16: rnode = CreateEphemNode();
      // create ephemeral node
17: bool IsSymmetric = false;
18: if (lcstree.node.NCV == itree.node.SCV)
        // There is no concurrent update on this node of itree
19:     IsSymmetric = true;
20: else if (lcstree.node.key == itree.node.key)
        // There is a concurrent update on this node of itree
21:     IsSymmetric = true;
22:     if (itree.node.DependsOn)
23:         return true;

24: if (IsSymmetric)
25:     Meld the left subtrees of lcstree and itree using
        the left subrange of kr split by lcstree.node.key.
        Then Meld the right subtrees symmetrically. Store
        the result in rnode.left and rnode.right respectively.
        Then rebalance the tree rooted at rnode.
        // prepare fields of rnode
        …
26:     rtree.node = rnode; return false;
    // asymmetric case
27: pnode = the node in itree with lcstree.node.key if it exists,
            else null;
28: if (pnode != null)
29:     if (lcstree.node.NCV != pnode.SCV
        && pnode.DependsOn)
30:         return true;

31: if (lcstree.node.key > itree.node.key)
32:     leftkr = LeftRangeSplit(kr, lcstree.node.key);
33:     if (Meld(lcstree.node.left, itree, leftkr,
                [out] rnode.left))
34:         return true;
35:     ttree = itree; tnode = itree.node;
36:    while (tnode && tnode.key <= lcstree.node.key)
37:         ttree = tnode.right; tnode = ttree.node;
38:     rightkr = RightRangeSplit(kr, lcstree.node.key);
39:     if (Meld(lcstree.node.right, ttree, rightkr,
                [out] rnode.right))
40:         return true;
41:     Rebalance the tree rooted at rnode.
        // prepare rnode fields
        …
42:     rtree.node = rnode; return false;
43: else
        // symmetric to lines 32-42 above
        …
```

D. CORRECTNESS ARGUMENT

We sketch a proof of correctness of the meld algorithm. We use the term **grafting conditions** for conditions that stop further recursive calls to meld. If meld returns an LCS subtree, we call them **grafting-lcs** conditions, e.g., the conditions at lines 1, 3, and 10 in the pseudo-code (Appendix C). Otherwise, if meld returns an intention subtree, we call them **grafting-intention** conditions, e.g., the conditions at line 8 and 12.

A correct meld algorithm should preserve three properties:

1. It detects all conflicts.
2. If no conflict is detected, it returns a new LCS tree that
   a. includes all updated nodes of the intention tree plus all nodes of the LCS tree that are not replaced by updated nodes in the intention tree, and
   b. does not duplicate any node which should be unique, i.e., does not contain two nodes with the same key.

In the proofs of these properties, we continue to use notation *lcstree, itree,* and *rtree* to represent the LCS subtree, the corresponding intention subtree, and the new resulting LCS subtree of each recursive call of meld, respectively. Before we prove the three properties, we first argue that proving them only needs to consider the grafting conditions.

Meld recursively traverses the LCS and intention trees starting from their roots, from which two versions of the entire database tree are reachable. It enumerates all the nodes in the two trees except the ones that are truncated by the grafting conditions.

For every node in the intention tree that is enumerated, meld checks the node's dependency information to detect the conflict that may be caused by it (lines 14, 22, and 30). Therefore, for the proof of property 1, we only need to consider the grafting conditions to show that they do not stop the traversal of a subtree that might manifest a conflict.

Now consider property 2a. During the traversal, for any enumerated pair of nodes in the LCS tree and intention tree that have the same key, meld replaces them with a newly created ephemeral node (lines 16, 26, and 42). Hence, if it does a full traversal of both trees without encountering a grafting condition, the property holds. Therefore, in the proof of property 2a, we only need to consider the effect of grafting conditions.

For property 2b, by the same argument, when meld encounters LCS and intention nodes that have the same key, it replaces them by only one ephemeral node. Therefore, in this case meld does not duplicate these nodes in the new LCS tree. All other cases of generating *rtree* involve a grafting condition, which picks up a subtree of *lcstree* or *itree*. Therefore, in the proof of property 2b, we again only need to consider the effect of grafting conditions.

We now sketch proofs of properties 1, 2a, and 2b in the following three theorems, respectively. Thus, for each of the properties, as long as the action associated with each grafting condition does not break the claims of the corresponding theorem, the theorem is true and the property holds.

THEOREM 1. *Meld detects all conflicts.*

Proof sketch: This claim is true if meld checks every node in the intention subtree that might incur a conflict. According to the arguments in the last paragraph, we only need to consider grafting conditions.

For grafting-lcs conditions:

1. *itree* is empty (line 1).
   It is clear that no node in *itree* needs to be checked unless *itree*'s intention deletes a node from *lcstree*. The pseudo code does not handle deletions. But if it did and deletion were handled by tombstones, then tombstones would be enumerated and detected as conflicts like any other update. If tombstones were not used, then this can be handled by delete sets, described in Section 6.
2. *itree* points to a node outside of its intention (line 10).
   Since an intention's transaction never updates nodes outside its intention, it is clear that no node in *itree* needs to be checked.
3. *itree* is a read-only subtree (line 3).
   Meld invokes function MeldNoEphem() to detect possible read-write conflicts incurred by nodes in *itree*.

For grafting-intention conditions:

1. *lcstree* is empty (line 8).
   In this case, meld picks all the nodes of *itree* whose keys are in the current key range and detects conflicts that may be caused by them. Since each invocation of meld guarantees that its parameter *itree* contains all the nodes in the intention tree whose keys are within the corresponding key range, the nodes of *itree* whose keys are not within the current key range are handled by other invocations of meld.
2. *lcstree.node.NSV* == *itree.node.SSV* (line 12).
   This is the serial case, and hence no conflict can happen in *itree*. □

THEOREM 2. *If no conflict is detected, then meld merges the LCS tree and intention tree into a new LCS tree that contains all updated nodes of the intention tree plus all nodes of the LCS tree that are not replaced by updated nodes in the intention tree.*

Proof sketch:

For grafting-lcs conditions, *rtree* returns *lcstree*, so all keys of *lcstree* are returned. Therefore, we only consider whether some node in *itree* may be left out.

1. *itree* is empty (line 1).
   There are no nodes in *itree* so none can be missed.
2. *itree* points to a node outside of its intention (line 10).
   Since *itree* points to a subtree that is the same age or older than *lcstree*, *lcstree* has the latest version of the nodes in the current key range. Therefore, no node in that subtree of *itree* needs to be included in *rtree*.
3. *itree* is a read-only subtree (line 3).
   Since *itree* is a read-only subtree, it does not include any new nodes beyond those in *lcstree*.

For grafting-intention conditions:

1. *lcstree* is empty (line 8).
   In this case, since Meld does not directly return *lcstree* or *itree*, the nodes in both of them need to be considered. No nodes in *lcstree* can be missed since *lcstree* is empty. For *itree*, meld picks all the nodes in *itree* that are in the key range of the current *lcstree*. Therefore, as long as all the nodes belonging to the intention and within the key range are under *itree*, no node can be missed. This is guaranteed by the recursive calls in both the symmetric and asymmetric cases.
2. *lcstree.node.NSV* == *itree.node.SSV* (line 12)
   In this case, *rtree* returns *itree*, and hence we only consider whether nodes in *lcstree* can be missed. Since *itree* is generated based on the structure and content of *lcstree*, all the nodes in *lcstree* should also exist in *itree* unless *itree*'s intention deletes them. Clearly, no node in *lcstree* can be missed. □

THEOREM 3. *If no conflict is detected, meld merges the LCS tree and intention tree into a new LCS tree that does not duplicate*

any node that should be unique, i.e., does not contain two nodes with the same key.

Proof sketch:

During the handling of all the grafting conditions, meld returns the nodes in *lcstree* or *itree*, but not both. Hence, the only case where duplication can be introduced is if some node in *itree* whose key is outside the current key range is included in the returned *rtree*. This can only happen in the second grafting-intention condition (line 12):

lcstree.node.NSV == *itree.node.SSV*

In this case, *rtree* returns *itree* assuming all the nodes in *itree* are within the current key range. This can be guaranteed by the transaction executor, i.e., during the execution of transaction. If it generates a subtree *i* whose key range is beyond the key range of the subtree *j* whose NSV equals to the SSV of *i*, it has to invalidate the SSV of *i*. □

E. Transaction Execution

Our meld algorithm depends on version information in LCS nodes and intention trees. The latter is maintained during transaction execution, which we describe in this section.

When a transaction *T* starts, it initializes its intention tree *I* with a pointer *p* to the root of its snapshot. This is the root of the latest locally-cached partial copy of the database generated by the meld in *T*'s server. Notice that *p* crosses intentions, in that it points from *I* to the root of a (usually ephemeral) intention in the snapshot. Suppose *T* executes under serializable isolation, which requires capturing its readset. In this case, when it reaches a node *n* in the snapshot, it creates a new copy *n′* of *n* in *I*. This copy *n′* includes *n*'s child pointers, if it has any, which are cross-intention pointers. *T* then assigns NCV(*n*) and NSV(*n*) to SCV(*n′*) and SSV(*n′*), respectively. NCV(*n*) and NSV(*n*) are calculated based on *n*'s source versions, alteration flags, and VN(*n*) as described in Sections 4.2 and 4.3. Under weaker isolation levels, *T* copies node *n* only if it is on a path to a node that *T* updated.

VN(*n′*) cannot be calculated and stored during *T*'s execution, since the CSN of *T*'s intention cannot be known until it is appended to the log and melded. Suppose we serialize *T*'s intention tree *I* into an intention record by a post-order traversal. If *n′* has a right child *rc* in *I*, VN(*rc*) = VN(*n′*) − 1. If *n′* has a left child *lc* in *I*, VN(*lc*) = VN(*n′*) − |subtree(*rc*)|, where |subtree(*rc*)| is the size (i.e., number of nodes) in the right subtree of *n′* in *I*,. To enable this calculation, while serializing *I*, we store the size of *I* and of each node that is the right child of its parent.

As a result, when a node in the log is visited by traversing from its parent in the same intention, its VN can be calculated. Note that the execution of the transaction may generate pointers crossing intentions through which a node in the log can also be accessed. We therefore need to associate the calculated VN of the node with those pointers.

T's intention tree should not be destroyed immediately after being serialized and appended into the log, since it can be used by meld in the same server to avoid unnecessary deserialization.

F. Experiments on Other Parameters

Abort rate. The abort rate of transactions depends only on the behavior of the application and optimistic concurrency control (OCC). Given an application, all OCC algorithms (including meld) using the same conflict predicate. Hence, they abort the same transactions and generate the same abort rate. Conversely, the only effect of the abort rate on the performance of OCC algorithms (including meld) is that the algorithm need not check conflicts with already aborted transactions.

Figure 13 shows the abort rate corresponding to the experiment for the symmetric case presented in Section 8.1. When the transaction size is 32 and concurrency degree is 256 (d256), the abort rate is over 40%. Since the d256 case has more than twice the aborts of d64, it is truncating many more traversals of intentions than d64. This is probably why d256 has almost the same latency as d64 in Fig 8a, unlike the lower concurrency degrees whose latency increases significantly as concurrency degree increases.

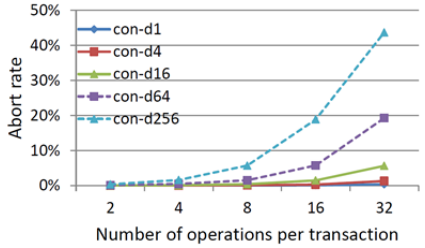

Figure 13 Abort Rate vs. Transaction Size.

Comparison to brute force. To demonstrate the effectiveness of the optimizations used by meld, we conducted experiments that compare meld with a brute-force version that disables all optimizations and examines all nodes in the intention tree to detect a conflict. Figure 14 shows meld performance normalized to its brute-force counterpart. The **rw** series presents the symmetric case with a read-update ratio of 1:1. The **ri** series gives the asymmetric case with a read-insert ratio of 1:1. The results show that meld optimizations yield approximately a 2x speedup.

Our experiments favored the brute-force solution by caching all the data used by meld in main memory to avoid I/O. In a configuration that cannot fit all data in cache, the brute-force version will suffer more since it needs to access more data which may incur more cache misses.

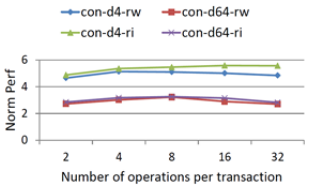

Figure 14 Meld Performance Normalized to Brute-force.

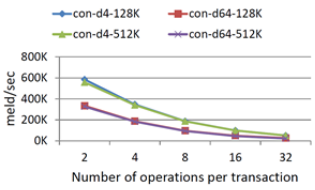

Figure 15 Meld Performance on Different Database Size.

Tree depth. We also evaluated how meld performance would be affected by tree depth. To do this, we enlarged the size of the initial database table to 512K key-value pairs, which adds two levels to the tree depth relative to the original configuration in Section 8. Figure 15 compares meld performance with initial table sizes of 128K and 512K key-value pairs in the symmetric case with a read-update ratio of 1:1. The curves for each concurrency degree are nearly identical, and hence barely visible in the figure. This shows that the tree depth has little effect on meld performance, since in a deeper tree many melds short-circuit at the same depth as in shallower tree. Results for the asymmetric case are similar.