Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

Sudipto Das
Microsoft Research
sudiptod@microsoft.com

Bailu Ding†
Cornell University
blding@cs.cornell.edu

Markus Pilman†
ETH Zurich
mpilman@inf.ethz.ch

ABSTRACT

Scaling-out a database system typically requires partitioning the database across multiple servers. If applications do not partition perfectly, then transactions accessing multiple partitions end up being distributed, which has well-known scalability challenges. To address them, we describe a high-performance transaction mechanism that uses optimistic concurrency control on a multi-versioned tree-structured database stored in a shared log. The system scales out by adding servers, without partitioning the database.

Our solution is modeled on the Hyder architecture, published by Bernstein, Reid, and Das at CIDR 2011. We present the design and evaluation of the first full implementation of that architecture. The core of the system is a log roll-forward algorithm, called meld, that does optimistic concurrency control. Meld is inherently sequential and is therefore the main bottleneck. Our main algorithmic contributions are optimizations to meld that significantly increase transaction throughput. They use a pipelined design that parallelizes meld onto multiple threads. The slowest pipeline stage is much faster than the original meld algorithm, yielding a 3x improvement of system throughput over the original meld algorithm.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*Concurrency, Distributed databases, Transaction processing*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Scale-out transaction processing; optimistic concurrency control.

1. INTRODUCTION

Most approaches to scaling out a distributed transaction processing system beyond a few servers require some degree of database partitioning. However, not all databases are partitionable such that most transactions refer to only one partition. For example, if there is a frequently-accessed many-to-many relationship that is equally likely to be traversed in both directions, then no partitioning of the relationship instances will ensure that most transactions are single-partition. A good example is a friend-status relation of users in a social network application. If the relation is partitioned by user, then each user U's status must appear in the partition of all of U's friends. Therefore, when U's status changes, it must be updated in many partitions (unless U has at most one friend).

† Work performed while employed at Microsoft Research.

SIGMOD'15, May 31 - June 04, 2015, Melbourne, VIC, Australia
Copyright 2015 ACM 978-1-4503-2758-9/15/05…$15.00
http://dx.doi.org/10.1145/2723372.2737788

Recently a technique that scales out without partitioning the database was introduced in a system called Hyder. It uses a network-attached log as the database, which is accessible by all servers [7]. The only point of arbitration between servers is when they append records to the totally-ordered log that they share. That is why the system can scale out without partitioning.

In this technique, the database is organized as a tree-structured index. Each server executes each transaction against a locally-cached partial-copy of its latest database snapshot (Figure 1, step (1)). The snapshot is defined by a log position S and hence is immutable. It is the state produced by all transactions up to S that committed.

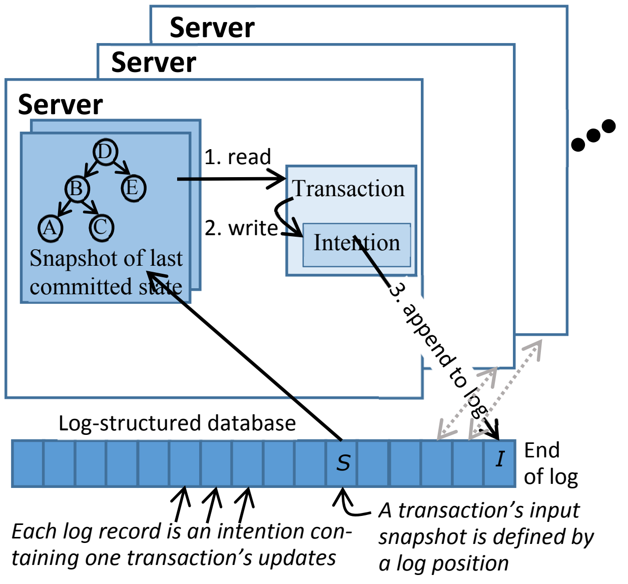

Server
Server
Server
D
B
E
A
C
Snapshot of last committed state
1. read
Transaction
2. write
Intention
3. append to log
Log-structured database
S
I
End of log
Each log record is an intention containing one transaction's updates
A transaction's input snapshot is defined by a log position

Figure 1: System architecture

Each transaction T executes optimistically with no synchronization. While T executes, it accumulates its updates in an **intention record** (Figure 1, step (2)). Conceptually, the intention contains new versions of the data that T updated and a logical pointer to a log position that defines the database snapshot that T read. Physically, an intention is quite different than this and is described later.

When T finishes executing, the system appends T's intention to the log (Figure 1, step (3)). In general, other transactions' intentions are appended to the log while T was executing. Therefore, there are usually many intentions in the log that follow the position of T's snapshot S and precede T's intention I, as shown in Figure 1.

Every server runs an algorithm called **meld** that rolls forward the log against the latest version of its locally-cached copy of the database. Meld analyzes each intention for conflicts using an optimistic concurrency control algorithm. If the intention did not experience a conflict (with respect to a given isolation level), then the transaction commits and meld merges the updated values in the intention into the server's locally-cached database. If the intention did experience a conflict, then it is discarded and has no effect.

The complete persistent database is in the log. The database at each server is only a partial cached copy. All servers have data that was updated by recently executed transactions, since that data was needed by meld. But different servers also have different data, if they read data that is not needed for conflict detection at weak isolation levels and hence is not logged. That data comes from executing read-only, read-committed, or snapshot-isolated transactions.

If a transaction accesses data not in its server's cache, then the server must do a random access to the log to get the data. Therefore, for good performance, the log should be stored on solid state disks.

To avoid synchronization between servers, meld is designed to be deterministic. All servers execute the same meld algorithm. Since the log is shared by all servers, they execute meld on the same sequence of intentions. Thus, for every intention they all make the same commit or abort decision, and they apply the committed intentions to their cached database copies in the same order. In effect, the system operates as a replicated database where each server has a cached subset of the database.

A natural implementation of meld processes intentions sequentially in log order, as in [8]. Since meld is sequential and the sequential processing speed of today's processors is not increasing very fast, meld is obviously a potential bottleneck. Other potential bottlenecks are the rate at which the log can be written and read, the network that connects servers to the log, and the abort rate due to conflicts between transactions. However, in our system, which uses a state-of-the-art network and servers, meld has been the bottleneck that limits transaction throughput.

Since queries execute against snapshots, they are not logged or melded. Hence, they scale out linearly until the log's read bandwidth or the network bandwidth is saturated.

One goal of this paper is to speed up meld. One approach is to speed up the sequential meld algorithm that analyzes intentions one-by-one to detect conflicts. However, the meld algorithm described in [8] is already heavily optimized, and we have been unable to improve it. If speed-ups are possible, we predict they will be small.

The second approach is to parallelize meld, which is the main subject of this paper. We do it by using other threads to preprocess intentions in ways that reduce the amount of work required by meld. The speed of that **final meld** step is what ultimately determines transaction throughput.

There are three preprocessing steps. The first is **deserialization**, which transforms each intention from its log format into an object structure. The original meld algorithm in [8] used several deserialization threads to reduce meld's execution time by up to 45%.

In this paper, we introduce two new preprocessing steps that execute after deserialization and before final meld. Each step executes on a deserialized intention I that final meld has not yet processed. The speed of meld is largely determined by the size of each intention and the fraction of each intention that meld has to process. Group meld reduces the former and premeld reduces the latter.

The first optimization, called **premeld**, does a trial meld of I that looks for conflicts with committed transactions in the part of the log that the final meld processing step has already processed. It also "refreshes" I by replacing stale data in I by committed updates from that earlier part of the log. This reduces the fraction of I that the final meld step has to process.

The second optimization, called **group meld**, combines adjacent intentions. As we will see, intentions usually have overlapping information, which collapses into one copy in the grouped intention. This speeds up final meld, which has to process that overlapping information only once instead of twice.

Implementing premeld and group meld presents two technical challenges. First, since every server runs premeld and group meld, they both need to be deterministic. Otherwise, different servers will end up in different database states, leading to database corruption. Second, since the meld algorithm is quite complicated, it is important to factor out meld's core behavior that can be used in premeld, group meld, and final meld, to ensure that all three algorithms have consistent semantics. We were able to accomplish this with a relatively small number of modifications of the original meld algorithm. The resulting algorithm can be abstracted as a generic operator, which could be used in other contexts.

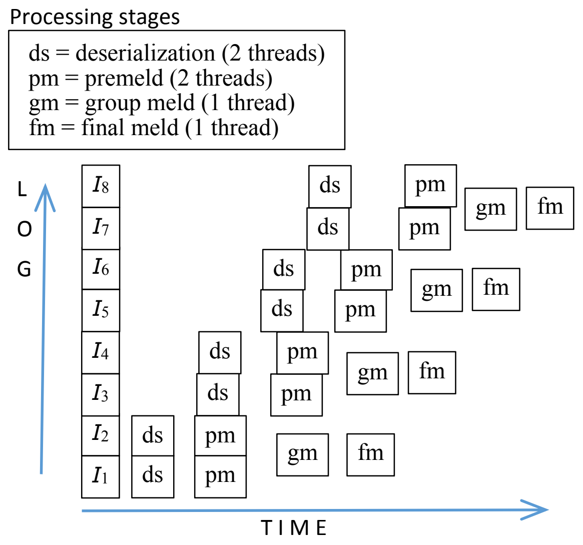


Figure 2: Pipeline parallelism of meld using four stages

In essence, our use of three preprocessing steps is a pipeline parallelism strategy (see Figure 2). We split the meld task for each intention into a pipeline of preprocessing stages (labelled ds, pm and gm), followed by final meld processing (labelled fm). The earlier stages execute in parallel with final meld processing, on intentions that the final meld step has not yet reached. For example, in Figure 2, in the first time unit, we see two parallel threads executing the deserialization (ds) stage on log entries I_1 and I_2. In the second time unit, two parallel threads execute the premeld (pm) stage on I_1 and I_2, and two parallel threads execute the ds stage on log entries I_3 and I_4. Next, we see two parallel threads execute ds on log entries

I_5 and I_6, two threads execute pm on I_3 and I_4, and one thread executes group meld (gm) on the combination of I_1 and I_2.

To show the benefit of these optimizations, we developed **Hyder II**, a distributed implementation of this architecture on a shared network-attached log. The performance measurements reported in this paper are the first ones for a complete end-to-end implementation of meld on a cluster of machines connected to a shared persistent log. By contrast, the implementation described in [8] was entirely in main memory on one server, and the one in [7] was a simulation. Moreover, our implementation uses a log based on solid-state disks (SSDs) [4], rather than the custom hardware proposed in [7].

In our experiments, five premeld threads give a 3x throughput improvement, and one group meld thread gives a 1.6x improvement. We found little benefit in running both premeld and group meld, compared to running premeld alone. Therefore, although we exercised the complete pipeline shown in Figure 2, for the workloads we tried, if enough processor cores are available to run several premeld threads, then premeld should be used. Otherwise, group meld can give a significant speedup with only one core.

In summary the main contributions of this paper are the following:

1. Two new optimizations for optimistic concurrency control validation of transactions on a multi-versioned tree-structured index and incorporating them into the meld algorithm.
2. A modified meld algorithm that can be reusable as an operator for premeld, group meld, and meld itself.
3. A comprehensive performance evaluation of meld in an end-to-end distributed implementation. We show the new optimizations give a 3x speedup over the best published meld technique. When executing a mix of 5% read-write and 95% read-only transactions with serializable isolation, Hyder II scales almost linearly, reaching peak throughput of 670K transactions per second (**tps**) for transactions with ten operations.

The paper is organized as follows. As background, Section 2 describes the basic meld algorithm. Sections 3 and 4 present our new optimizations. Section 5 describes our implementation, and Section 6 reports on experiments that evaluate its performance. Section 7 summarizes related work, and Section 8 is the conclusion. Three appendices expand on certain aspects of the meld algorithm.

2. MELD

The meld algorithm operates on a database organized as a tree-structured index, such as a binary search tree or B-tree. Since it operates on main memory structures and is serialized to a sequential log (rather than written out in fixed-size pages), a binary tree consumes less storage per record than a B-tree [7]. So we use binary trees in our examples and in our implementation. Each node has a key and value, i.e., a payload.

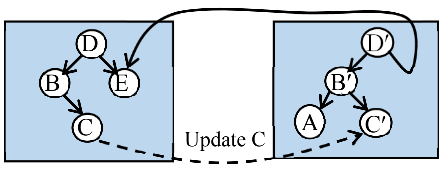


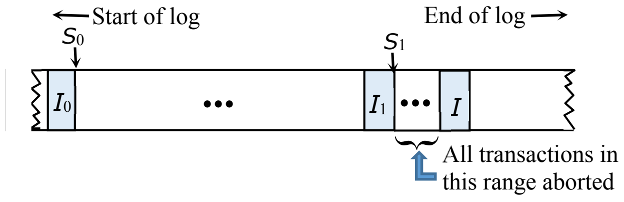
Figure 3: Inserting node A and updating node C

Since a transaction operates on an immutable database snapshot, updates must use **copy-on-write**. That is, an update must create a new version rather than modify data in place. For example, consider Figure 3. The left side shows a database snapshot. A transaction inserts a new node A and updates node C, resulting in an intention shown on the right. The update of C must create a new version C′, which requires updating B's pointer to C, which requires creating a new version B′, and so on up the tree.

Conceptually, the intention I for a transaction T contains T's updates. Physically, I defines the entire database state that T produced. That is, if T executed stand-alone on its input snapshot, then the resulting database state would be I. Given the use of copy-on-write, for every node n that T modified, I contains every node on a path from the root to n, including n itself. For example, in Figure 3 the transaction's updates are "insert A" and "update C". But its intention, in the right box, defines the state that the transaction produced, which includes B′ and D′.

If a transaction's isolation level is serializable, then its intention also contains the nodes in its readset. Such nodes are annotated to identify them as having been read but not written.

The meld algorithm operates on an intention I and the **last committed state (**or **LCS)**, which is the database state produced by the last transaction that preceded I in the log and committed (e.g., in Figure 4, state S_1 produced by intention I_1). Its goal is to determine whether the transaction that generated I, T(I), experienced a conflict. If so, it discards I. If not, then T(I) commits so meld merges I's updated nodes into the last committed state. In effect, meld produces the intention that T(I) would have produced if it had executed against snapshot I_1 instead of I_0 (see Figure 4).



S_0 – state that intention I read (T(I)'s snapshot)
S_1 – last committed state to meld with I

Figure 4: Log positions relevant to meld

Every database state S includes metadata that tells how S relates to the previous database state, Prev(S), from which it was generated. For an intention I, the metadata tells how I relates to T(I)'s snapshot. For a state S produced by meld, it tells how S relates to meld's input state, e.g., S_1 in Figure 4.

The metadata is attached to each node n in a state S. Examples of that metadata are the following:

- ➢ A reference to the node n_{prev} in Prev(S) that has the same key as n (if n_{prev} exists).
- ➢ A flag indicating that n's payload differs from n_{prev}'s payload, e.g., because S is an intention and T(S) updated n.
- ➢ The identity of the first version of n that produced n's payload.
- ➢ If S is an intention, a flag indicating that T(S) read n.

With the exception of the flag that indicates T(S) read n, the metadata is redundant, in that it could be computed from the log. However, it would be quite expensive to do so, leading to an inefficient meld implementation. Part of the beauty of the meld algorithm is that it can compute this metadata incrementally and efficiently.

Meld uses this metadata to help it detect conflicts and merge its input intention and database state. Most details of how it does this are not important for understanding the optimizations in this paper. Still, to give the reader a sense of how conflicts are detected, we sketch a basic version of the meld algorithm in Appendix A.

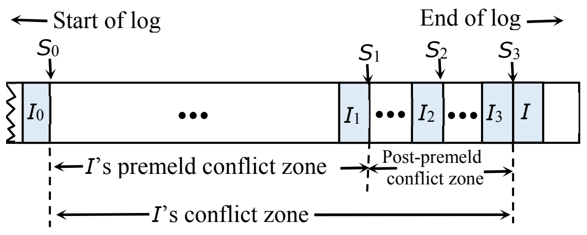
There is one detail of the meld algorithm that is relevant to the optimizations in this paper, namely, the generation of **ephemeral nodes**. To understand ephemeral nodes and how they arise, suppose meld executes on state S and intention I, and determines that node n was updated by T(I) and was unchanged by any other transaction concurrent with T(I). This implies that the subtree under n in I was unchanged, since if it were then n would have changed too (as in Figure 3). Therefore, meld can simply replace n in S (denoted n_S) by n in I, which also replaces n_S's subtree by n's subtree. (This merging of subtrees is why the algorithm is called "meld.") If n_S is not the root of S (as is usually the case), then n_S's parent p must be replaced in S by a new node p', due to copy-on-write. Notice that p' is produced by meld, not by a transaction. Therefore p' is not written to an intention that is stored in the log—it only exists in main memory. In this sense, it is ephemeral. As in Figure 3, ephemeral nodes are generated for all of the ancestors of p' too.

An ephemeral node e might be read or overwritten by a later transaction T. T's intention will have a node with the same key as e and that node will refer to e. Since meld is deterministic and every server will meld T's intention, all servers must generate e with the same identity so that T's intention is interpreted the same way on every server. As we will see, this complicates premeld.

3. PREMELD

3.1 Premeld Overview

For a given intention I, the intentions in the log between I and the snapshot that T(I) read are called I's **conflict zone** (see Figure 5). The process of melding I with its input LCS involves combining I's updates with parts of the database that were updated by committed transactions in I's conflict zone. One way to parallelize meld is to use a parallel thread to do a preliminary meld of I with a state that is earlier than its input LCS, such as S_1 in Figure 5. This preliminary meld is called **premeld**. The LCS S_1 that it merges with I is called I's **premeld input** and the log region between I's snapshot and its premeld input is called I's **premeld conflict zone**.



S_0 - state that intention I read (I's snapshot)
S_1 – state to minimeld with I (I's premeld input)
S_2 – state when I arrived at the meld log
S_3 – state to meld with I (I's input LCS)

Figure 5: Log Positions Relevant to Premeld

Premeld serves two purposes. First, it checks for conflicts in a prefix of I's conflict zone, that is, its premeld conflict zone. If it discovers a conflict, then final meld can skip processing I, since it knows that I will abort.

Second, if premeld finds no conflicts, then it produces an updated intention I_m that appears to have executed against I's premeld input. Later, final meld will meld I_m (instead of I) into I's input LCS. This requires less work than melding I, because there is less conflict checking to do. So the effect is to speed up final meld.

Surprisingly, the premeld algorithm is almost exactly the same as meld, but with different inputs. We will explain that later. First, let us look at why the optimization is effective and how it works.

3.2 Premeld Details

Meld's throughput is a function of the number of nodes it has to traverse. *This depends on the number of nodes in each intention I with descendants that were changed by committed transactions in I's conflict zone*.

For example, consider the states in Figure 6, which correspond to Figure 5. Intention I executed on snapshot S_0 and updated leaf E, producing E″. Since T(I) did not read or write node C, C is outside of I. While T(I) was executing, another transaction T_1 updated node C, producing C′. Since T_1's intention was appended to the log before I and it committed, its updated node C′ appears in I's input LCS. When melding I with its input LCS, an ephemeral node D^e is generated, which connects to C′ and E″. This generates a new ephemeral parent for D^e, and so on up the tree. If this were a tree with a million nodes, the root-to-leaf path would have 20 nodes, so 19 ephemeral nodes would be generated.

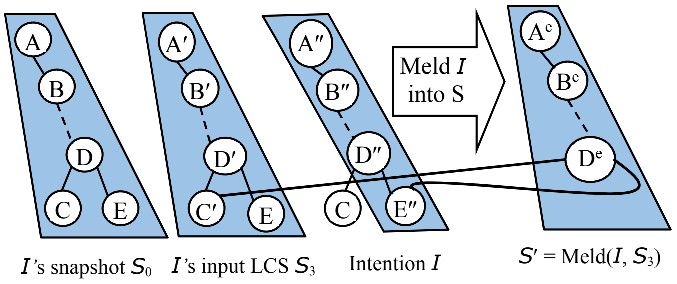


Figure 6: Detailed example of premeld

Suppose that node C was the left child of node A instead of the left child of node D. In this case, melding I with its input LCS would generate only one ephemeral node, namely A^e, connecting to C′ and B″. That is, it would graft I's subtree rooted at B″ into the LCS. Comparing these two cases, we see that in the first case 19 nodes in I had a descendant that changed in I's conflict zone, while in the second case, only one node (i.e., node A) in I has such a descendent.

Now, suppose that the state in Figure 6 that is labelled "I's input LCS S_3" is instead I's premeld input, that is, state S_1 in Figure 5. Since premeld performs the same computation as meld, its output is still S'. However, as we describe in the next section, S' is interpreted as an intention, not as an LCS, and substitutes for I as input to meld. So final meld will meld S' into the LCS, which is S_3.

To produce S', premeld paid the cost of generating the ephemeral nodes on the path from C′ to the root. When melding S' into S_3, that work will be repeated only for nodes on that path that have a descendant that is updated in the post-premeld conflict zone. That conflict zone is usually much smaller than I's original conflict zone, and hence is unlikely to have many updated descendants of nodes on that root-to-C′ path. Therefore, few if any nodes on that path will be traversed during final meld. Thus, as a result of I's premeld, which executed on a separate hardware thread in parallel with the final meld thread, the final meld of S', which substitutes for I, will be faster than if it had been had it executed on I's entire conflict zone. Since final meld is performed on a separate thread, the speedup of final meld increases overall transaction throughput.

Premeld is beneficial mostly because I's post-premeld conflict zone is much smaller than its premeld conflict zone. This ensures that most of the work of meld is done by premeld, rather than final meld. The post-premeld conflict zone size is governed by the distance in the log between *I* and its premeld input. By design, this is kept quite small. In our experiments, it is just a few hundred transactions. By contrast, when throughput is ~50K tps, the conflict zone is typically 10K-30K transactions, and the premeld conflict zone is just a few hundred less. Thus, we see a ratio of 100:1 between the sizes of premeld and post-premeld conflict zones.

We considered running premeld on an intention *I* immediately after T(*I*) terminated and before *I* was appended to the log. This would merge in updates applied by meld at T(*I*)'s server during T(*I*)'s execution. However, since a transaction executes in tens of microseconds, the benefit would be much smaller than running premeld after the log append, which takes milliseconds. Running premeld after the log append makes it useless to also run it before.

3.3 Meld Implements Premeld Semantics

We now explain why meld, with a small modification, implements the semantics required for premeld. It is based on two observations: (i) each transaction is represented as a pair of states, and (ii) the output of meld is a pair of states and hence is a transaction.

First, consider the representation of a transaction. In the database field, we normally think of a transaction either as a program or as a sequence of operations on shared data that is generated by an execution of a program. From a transaction processing standpoint, it is a little unusual to think of a transaction execution as a pair of states <S_{in}, S_{out}>. However, from a programming language standpoint, it is a natural representation. In our case, it is how a transaction's semantics is physically represented.

The execution of a transaction *T* generates an intention *I*, which represents the database state produced by *T* executing against its input snapshot. *I* includes the identity of the snapshot against which it executed. That identity is represented as metadata attached to each node in *I*. Taken together, that per-node metadata defines an input database state. Therefore, we can abstract its semantics as simply <S_{in}, S_{out}>, where S_{in} is its snapshot and S_{out} is its intention.

Second, to see why meld's output is a transaction, remember that its goal is to merge the updates of an intention *I* with *I*'s input state S_{in}, producing an output state S_{out}. This is equivalent to saying that S_{out} is the result of running T(*I*) on S_{in}. In this sense, we can regard its output as a transaction <S_{in}, S_{out}>. In fact, its output is physically represented that way, since the metadata attached to each node of S_{out} describes how it relates to its previous version.

There is one omission in this explanation: readsets. Since meld's output is interpreted as a transaction that undergoes an optimistic concurrency control check, it is not enough to represent each transaction by a pair of states, <S_{in}, S_{out}>. It must also include its readset, to enable conflict detection for serializable isolation. An intention generated by executing a transaction includes this information. However, an intention generated by meld in [8] does not. This is correct in [8] because meld's result is interpreted only as a database state, not as a transaction. Thus, that output need not retain readset information for further optimistic concurrency control checks.

However, when we use meld to implement premeld, its output will be interpreted as a transaction that will be melded at least one more time by final meld. Therefore, it needs to include readset information. Since meld's output transaction represents the execution of the same program as the one that produced meld's input intention, the output transaction's readset should remain unchanged. Therefore, meld should copy the transaction's readset to its output. This requires changing only one line of the algorithm in [8]: When meld determines that a read-only subtree of an intention *I* is the same as the corresponding subtree *L* of the LCS, it should return *I*'s subtree as the output, rather *L*'s subtree, because *I*'s subtree has the readset metadata while *L*'s subtree does not. That is, line 7 of meld in Appendix C of [8] should read "*rtree* = *itree*; **return false**;" instead of "*rtree* = *lcstree*; **return false**;"

3.4 Determinism of Premeld

Meld runs independently on all servers that share the log, producing a sequence of database snapshots. We want the system to behave as if it has one shared database. Hence, all servers should generate an identical sequence of snapshots. Thus, meld must be deterministic.

It is not enough that the snapshots on different servers contain the same set of [key, payload] pairs. They must be physically identical. To see why, recall that the intention generated by each transaction may include ephemeral nodes. When the intention is melded, some of those ephemeral nodes may become part of the database. A later transaction *T* may refer to such an ephemeral node, *e*, in its intention. For example, *T* may have read *e* or updated a node whose root-to-leaf path refers to *e*. *T*'s intention *I* will be broadcast to all servers. To meld that intention, other servers may need to follow *I*'s reference to *e*. Therefore, the reference must point to the same node on all servers. This requires meld to produce physically identical trees with the same node identities on all servers.

To understand whether trees on all servers will use the same node identities, consider the two ways that a node identity can be generated. First, the node can be inserted by a transaction *T* and stored in *T*'s intention. Since the intention is broadcast to all servers, the node has the same identity at all servers. Second, it can be an ephemeral node created by meld. Ephemeral node identities are allocated sequentially in the order they are generated. Therefore, ephemeral nodes must be generated in the same order on all servers to ensure they have the same identity on all servers. This property holds if meld is deterministic, such as the algorithm described in [8]. In this case meld will produce physically identical trees on all servers.

However, when premeld is used, ephemeral nodes might no longer be generated in the same order on all servers, for two reasons. First, for a given intention, premeld might execute against different database states on different servers. For example, suppose the log contains a sequence of intentions I_1, I_2, I_3, I_4. One server might run premeld for I_4 on the state produced by I_1, while another server runs premeld for I_4 on the state produced by I_2. These premeld executions might generate different sequences of ephemeral nodes. Thus, an ephemeral node with a given identity, *e*, might contain a different [key, value] pair at different servers. From then on, their servers' database states will diverge, which causes the system to malfunction. For example, an intention *I* that was generated at server S_x and that points to *e* will later arrive at another server S_y, where *I* will point to *e*, but the content of *e* is different at the two servers. A detailed example is in Appendix C.

To solve this problem, we ensure each server runs the same number of premeld threads, and each premeld thread executes on the same sequence of [intention, database state] pairs. We do this by means of index calculations. If there are *t* premeld threads, then each intention is assigned to the thread identified by its intention id modulo *t*. Each intention is premelded against the state created by the $(t \times d)^{th}$ intention that precedes it, where *d* is a fixed, global parameter, called the **premeld distance**. If that state precedes *I*'s snapshot, then premeld does nothing. If that state is not yet available because final meld is slow, then the premeld thread waits until final meld creates the state. Note that the system must retain each state until the intention that premelds against it has executed.

The second reason that different ephemeral node identities might differ on different servers is that premeld executes in parallel with final meld. If these threads generate ephemeral nodes in different orders on different servers, then a node may have different identities on different servers. To avoid this, the final meld thread and each premeld thread generate thread-local sequence numbers. We use a two-part identity for each ephemeral node, consisting of the id of the thread that generated it plus the sequence number of the ephemeral node in that thread. Combined with the previous technique of premelding each intention to the same state at all servers, this ensures premeld generates the same ephemeral node identities at all servers. Premeld is shown in Algorithm 1 below.

Algorithm 1: PREMELD.

Input: Intention I_v at position v in the log; a sequence of database states corresponding to log positions: S_1 after I_1, S_2 after I_2, etc.; an integer premeld distance d; an integer number of threads t; and the most recent database state, S_r

Output: Abort, I_v, or the result of running meld on I_v and $S_{v-(t \times d)-1}$

```
1. $m \leftarrow v - (t \times d) - 1$ // the database version to meld against
2. $snap \leftarrow$ ReadVersion($I_v$) // the snapshot on which T($I_v$) executed
2. if $snap \geq m$ then
3.     return $I_v$ // $S_m$ is older than $S_{snap}$. So there's no need to meld.
4. else if $m \geq r$ then // Final meld hasn't produced $S_m$ yet
5.     wait for $S_m$ to become available
6. $S_{out} \leftarrow$ Meld($I_v$, $S_m$)
7. if $S_{out}$ says T($I_v$) experienced a conflict then return Abort
8. else return $S_{out}$
```

To maximize the benefit of meld, we want to minimize the size of the post-premeld conflict zone. We therefore want a small value of d. But it should be large enough to give premeld enough time to finish before final meld is ready for premeld's output. Otherwise, final meld will stall, reducing throughput. In our experiments, five premeld threads with d=10 resulted in the best throughput.

4. GROUP MELD

A second way to speed up meld is to combine a pair of adjacent intentions in the log into a single intention, called a **group intention**. Later, final meld evaluates the group intention as one intention. This is beneficial because the group intention is generally smaller than the concatenation of its two input intentions, because the input intentions contain overlapping sets of nodes. So it takes less time to meld it. Two overlapping nodes collapse into a single node in the group intention. Thus, the more nodes the two input intentions have in common, the smaller the group intention.

For example, Figure 7 shows a database tree comprising six nodes. Intention I_1 updated A′, so it includes B′ and D′, and I_2 updated C″, so it includes B″ and D″. I_1 and I_2 have overlapping nodes D and B, each of which collapses to a single node when I_1 and I_2 are melded into a group intention I_3 (which includes new ancestors D‴ and B‴, due to copy-on-write). Together, I_1 and I_2 have six nodes, but the group intention that combines them has only four nodes.

Like premeld, group meld uses the standard meld algorithm, as modified in Section 3.3 to include readsets. However, group meld requires special logic in one case: Suppose nodes n_1 and n_2 in I_1 and I_2 (respectively) have the same key (e.g., A′ and A″), and I_1 precedes I_2. Suppose n_1 is in T(I_1)'s readset and n_2 is in T(I_2)'s readset or writeset. Then the readset metadata for the melded node n_{12} must refer to the maximum of n_1's and n_2's conflict zones. That is, if T(I_1)'s snapshot precedes T(I_2)'s snapshot, then n_{12}'s readset must refer to T(I_1)'s snapshot, so that meld checks for conflicting updates in between the two snapshots. Thus, meld must replace n_2 by an ephemeral node, so it can update n_2's readset metadata. This arises in the meld algorithm in several spots, which requires special-case logic for group meld. Note that if T(I_2)'s snapshot precedes T(I_1)'s, then no action is needed because n_2's existing metadata dominates.

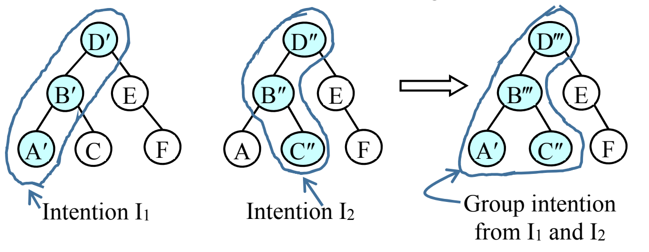


Figure 7: Combining two intentions into a group intention

The **group meld** optimization melds every adjacent pair of intentions, thereby halving the number of intentions in final meld. Like premeld, it has to be deterministic. A way to ensure this is to have each group intention be an odd-numbered intention in the log followed by the next one, which is even. Thus, intentions 1 and 2 form a group intention, then intentions 3 and 4, and so forth.

A disadvantage of group meld is that the group intention commits if and only if both intentions commit. Thus, it ties the commit/abort decision of each intention with that of another. This effect, called **fate sharing**, could increase the abort rate by up to a factor of two.

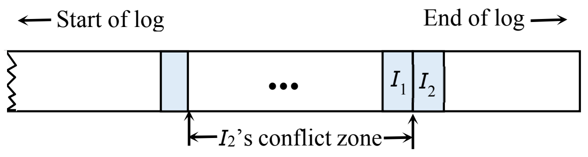


Figure 8: I₁ and I₂ meld into a group intention

However, there is one case where fate sharing does not occur. When melding two intentions into a group intention, if the first intention I_1 includes an update that conflicts with the second intention I_2, then group meld aborts I_2, because I_1 is in I_2's conflict zone (see Figure 8). Therefore, I_1 becomes the group intention and can still commit.

When premeld and group meld are combined in a pipeline, they can execute in either order. In our experiments, we ran premeld first, since it is more likely to find aborted transactions than group meld. The sooner in the pipeline that aborted transactions are identified, the better, since it reduces the amount of downstream work.

5. IMPLEMENTATION DETAILS

Our Hyder II implementation runs Hyder's meld algorithm [8] on a shared log manager. The core design comprises a cluster of **transaction servers** that execute transactions and the meld algorithm, and **log servers** that store the database. Our implementation allows for a single machine to run both a transaction server and log server. However, in our experiments, they run on different machines. The database is an immutable balanced binary search tree (a red-black tree [17]). As argued in [7], a binary tree is preferable to a B-tree, because it leads to smaller intentions.

5.1 Log Service

The log ensures persistence and enforces a total order on transactions. We used CORFU [4] as the log service, due to its good performance and feasibility compared to the custom hardware proposed in [7]. CORFU runs on a set of log servers on the same local area network as the transaction servers running transactions and meld. However, the transaction servers see it as a single shared log service, not as a set of log servers. Each log entry is a fixed-sized page, called an **intention block**. An **append** operation adds a given block to the end of the log and returns the position in the log where the block was stored. Given a log position, a **read** operation returns the block stored at that position. To ensure good append and read bandwidth, the log is striped across a set of storage units that are directly attached to the log servers. As we show in the next section, the log manager is much faster than meld and hence is never a bottleneck.

There are nontrivial error cases when read or append operations fail or fail to reply at all [4]. As these problems are not unique to Hyder II and this paper is about optimizing throughput and not fault tolerance, we regard them as out of scope.

Hyder II stores intentions in intention blocks. Depending on the size of the intentions, many intentions might fit in one block or an intention might span many blocks. In the latter case, the blocks of an intention need not be contiguous; the position of the intention in the log is identified by the position of its last intention block.

5.2 Transaction Servers

Hyder II is implemented in C#. Our implementation comprises the multi-versioned index structure storing the database tree, the transaction executer, and the meld pipeline that rolls forward the log. Bernstein et al. [8] presented an optimized and detailed design of the meld algorithm for the in-memory setting. Our meld implementation uses that algorithm, with the modification described at the end of Section 3.3. However, we re-implemented it in C# to efficiently use the new implementation of the database backed by our network-attached log.

A transaction executor dispatches transactions and processes their termination. Each transaction *T* executes on the current last-committed state (**LCS**) reported by the final meld thread at the time *T* starts. Due to copy-on-write, *T*'s first write generates a new intention and all of *T*'s subsequent writes add to it. After the transaction finishes executing, the executor serializes the intention into one or more blocks, appends it to the log, and then broadcasts it to the other transaction servers. *T*'s outcome will become known to *T*'s executor only after meld on the same server processes the intention. Until then, *T* is blocked. However, the executer is non-blocking and moves on to execute the next transaction while awaiting the commit decision from meld. The executer stops processing transactions if the number of transactions awaiting their outcome exceeds a configurable threshold which is determined empirically. Ideally, the threshold for the number of concurrent transactions allowed should be dynamically determined by admission control logic, which is future work. Such admission control is essential to avoid thrashing when the offered load exceeds capacity.

An intention tree is serialized into an intention block via a post-order tree traversal. Post-order ensures that each node points to children that are either in the log or already serialized. Once the block fills, serialization continues with a new block. The intention block containing the root of the database tree is appended last.

Each transaction server has a log reader that transforms the sequence of intention blocks in the log into an in-memory representation of the intentions that feeds the meld pipeline. For each intention block, it spins off a deserialization task, which generates a tree of objects. A configurable number of threads run the deserialization tasks. They transform each node pointer in an intention into an object pointer if the object is in memory. Otherwise, the node pointer is left as a log position; if dereferenced later, the log position is used to fetch the corresponding intention block from the log.

The pipeline of premeld, group meld, and final meld processes the deserialized intentions in log order. Configuration parameters are used to turn-off premeld and/or group meld, and to control the number of threads for premeld and group meld.

Meld is inherently sequential and ideally CPU-bound. Any stalls during meld significantly reduce its throughput. Since meld does not need synchronization, the only sources of stalls are late-arriving intentions and cache misses. If meld is ready to process the next intention *I* but has not yet received *I*, it stalls until *I* is read from the log. Similarly, if meld accesses a block that is not cached, it must read the block from the log. The latency to read a log block is on the order of milliseconds while the latency to meld an intention is typically tens to a few hundred microseconds. Therefore, it is important to avoid intention block misses in the final meld thread. Pipelined parallelism helps mask log access latencies, since premeld and group meld fetch blocks well before final meld needs them. To ensure that final meld does not stall on a cache miss, additional threads can be used to pre-fetch blocks not accessed in the earlier phases of the meld pipeline.

For optimal meld performance, it is also critical to avoid context switches. Therefore, meld spins if the next intention is not available to process. Each premeld, group meld, and final meld thread is also affinitized to a core to prevent inter-core thread migration.

5.3 Optimizations

Implementing Hyder II in managed code had advantages as well as challenges. It improved productivity and reduced the time to complete the first implementation. However, we had to iterate over the implementation to make its performance competitive with the optimized native in-memory implementation reported in Bernstein et al. [8]. A performance comparison with [8] is in Section 6.4.

Our first challenge was memory management and garbage collection (GC) overhead. Much progress has been made to minimize the overhead of GC. Even so, we found GC to be detrimental to performance. Therefore, each thread in Hyder II, such as the transaction executer, deserializer, and meld, has its own local memory pool to reuse objects whenever possible. Pre-allocated thread-local memory pools significantly reduced GC overhead and stalls. Careful profiling revealed that GC was also triggered by some language constructs, such as lambda expressions and library functions that do not allow reusing objects, and asynchronous API calls that automatically generate a callback on call completion. Eliminating these poor memory users was a tedious and iterative process, but we were rewarded with significant performance improvement. As described in Section 2, meld generates ephemeral nodes. Since the pipelined design has more instances of meld, it can generate more ephemeral nodes. Therefore, ephemeral nodes must be carefully managed to avoid unnecessary demand for memory.

The second problem was serialization and deserialization of the intention blocks and manipulation of byte arrays, which is CPU-intensive. This needed careful optimization.

The third problem was network errors. After an intention is appended to the log, it is broadcast to all servers. Our initial implementation simulated this broadcast over UDP. However, as the network utilization and contention increased, significant packet loss resulted in many out-of-order intention deliveries, which degraded performance. Even though simulating the broadcast using TCP was more expensive than using UDP, the switch to TCP resulted in another significant performance boost.

6. EXPERIMENTS

We present an experimental study of our Hyder II implementation. We start by analyzing the performance of the log service to ensure it is not a bottleneck. We then evaluate Hyder II for various configurations and demonstrate the benefits of the premeld and group meld optimizations for a variety of workload mixes and isolation levels. We also compare Hyder II's throughput with that reported for Hyder [8] and Tango [5], whose designs are similar to Hyder II's. Hyder's performance studies have been limited to simulations [7] or a single-node implementation with an in-memory log designed to test the limits of the meld operator [8]. This paper presents the first detailed experimental evaluation of Hyder's architecture.

6.1 Experimental Setup and Workload

We ran our experiments on a cluster of twenty servers. Each server has a two-socket Intel Xeon E5-2650L 1.8 GHz processor (16 physical cores, 32 logical processors, two NUMA nodes), 192 GB RAM, a commodity Intel SATA SSD, and a dual-port Intel 10 Gbps network adapter. Servers in the rack are connected through a 64 port 10 Gbps top-of-rack switch to provide a high speed private network interconnect within the cluster.

The CORFU log service consists of a token server, a sequencer for log entries, and six disk servers, which store the log on direct-attached SSDs. The installation has a number of Hyder II transaction servers, which use CORFU's client API to append and read intention blocks. A Hyder II server generates two types of network traffic: unicast traffic with the log servers for appends and reads, and broadcast traffic among the transaction servers after an intention is appended to the log. To avoid interference, each type of traffic uses a different port of the dual-port network adapter.

We use a workload generator adapted from the Yahoo! Cloud Serving Benchmark (YCSB), adding support for multi-operation transactions [10]. The workload generator allows us to vary the number of operations in a transaction, the distribution of reads and writes within a transaction, the fraction of range vs. point lookups, the database size, and the distribution from which the keys are selected.

Unless otherwise specified, we use a database with 10M items, each with a 4 byte key and 1K payload. In most experiments, we vary the number of transaction servers generating and executing the workload. Each server has 20 update threads. Each thread allows up to 80 in-flight transactions, that is, transactions that have been appended to the log but not yet received a commit decision. This limit simulates coarse-grained admission control to avoid overload and thrashing due to high resource contention. Each transaction has 8 reads and 2 writes. The keys read and written are selected uniformly from the keys in the database. For experiments that vary one of these default parameters, the number of servers is fixed at 6.

The uniform distribution of reads and writes is adversarial for any transaction or data partitioning scheme, since a transaction's accesses do not naturally cluster. We select this workload to demonstrate Hyder II's ability to scale-out without partitioning.

As discussed earlier, Hyder II has four critical resources: log appends, network bandwidth, meld, and data contention. An update transaction stresses all of them. So most of our analyses involve a write-only workload to help us identify resource bottlenecks.

All of our measurements are the average result of three runs. We do not report the standard deviation, since it is low in all cases except under heavy contention, a well-known phenomenon. We observed a standard deviation in the range of 2-5% of the averages.

6.2 Summary of Results

The key takeaways from our experimental results are:

- The optimizations proposed to parallelize meld result in significant throughput improvements. Group meld improves throughput by 1.6x with only one thread. Premeld improves throughput by 3x with five threads across a variety of workloads and isolation levels.
- When premeld and group meld are combined in a pipeline, they do not improve throughput beyond that of premeld.
- For transactions with 8 reads and 2 writes, snapshot isolation (SI) improves throughput by 3x compared to serializable. With SI, premeld results in an additional 2x improvement, while group meld's improvements are not significant.
- Read-only workloads scale almost linearly. With a workload of 5% read-write and 95% read-only transactions, we achieve about 670K transactions per second with ten servers and serializable isolation executing a non-partitionable workload.

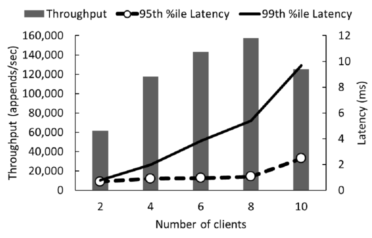
6.3 Performance of the Log Service

This section analyzes the throughput and latency of our shared log implementation. Our goal is to ensure that log appends are not a bottleneck that limit Hyder II's throughput.

The log service uses six disk servers. Clients append and read blocks of data. In Hyder II, log "clients" are transaction servers. The log service has a pre-configured block size; we use 8K blocks for Hyder II and for these experiments.

Figure 9 plots the throughput and latency of appends as we vary the number of clients appending to the log and the number of concurrent threads per server. Figure 9(a) and Figure 9(b) report results for 20 and 30 threads per server, respectively. As we increase the number of concurrent appends, the 95th and 99th percentile latencies increase, plotted on the right vertical axis. However, even the 99th percentile latencies are under 10 milliseconds. The peak throughput is more than 140K appends/sec. In our later experiments, Hyder II's transaction load generates at most 110K append/sec using a log configuration identical to these experiments. Hence, the log is not a bottleneck.

(a) 20 threads per client



(b) 30 threads per client

Figure 9: Throughput and latency of append operations to a shared log.

6.4 Performance of Hyder II

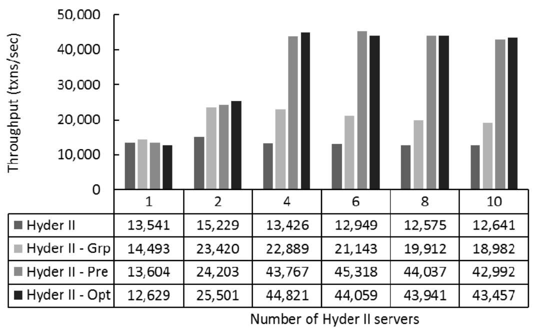
We first analyze Hyder II's performance using a workload executing all write transactions with serializable isolation to compare the benefits of premeld and group meld in isolation and in combination. Subsequently, we experiment with workloads with varying fractions of read-only transactions, with different isolation levels, and varying other workload parameters such as data access distribution, and transaction size. In all experiments, "throughput" means the number of *committed* transactions per second.

6.4.1 Workload with all Write Transactions

A write transaction stresses all the critical resources. Hence, a workload comprising all write transactions helps identify the architecture's performance limits. In this experiment, we use the default workload parameters described in Section 6.1. Figure 10 reports Hyder II's throughput in transactions per second (tps) as we vary the number of servers. The first bar corresponds to Hyder II with only final meld. The second bar (**Hyder II-Grp**) corresponds to Hyder II with group meld applied before final meld. The third (**Hyder II-Pre**) corresponds to Hyder II with premeld applied before final meld. The fourth (**Hyder II-Opt**) corresponds to applying both group meld and premeld. For experiments with premeld, there are 5 premeld threads with a premeld distance of 10. For experiments with group meld, there is one group meld thread grouping two intentions into one.

In Figure 10, Hyder II's peak throughput with no optimizations is 15K tps, which is achieved with 2 transaction servers. By contrast, peak throughput of Hyder II-Grp is 23.5K tps, Hyder II-Pre is 45.3K tps, and Hyder II-Opt is 44.8K tps. Thus, group meld improves throughput by 1.6x and premeld improves it by 3x. However, combining the optimizations does not improve throughput beyond that of premeld. Therefore, when enough cores are available to run many premeld threads, premeld should be used alone. Otherwise, group meld should be used to give a significant improvement with just one core.

The improvement with premeld is more significant as we increase the concurrency. For instance, with 10 servers, each with 20 threads and 80 in-flight transactions, there are up to 16K concurrent transactions, i.e., transactions that started executing but have not yet been melded. At such high degrees of concurrency, Hyder II-Pre has 3.5x the throughput of Hyder II.

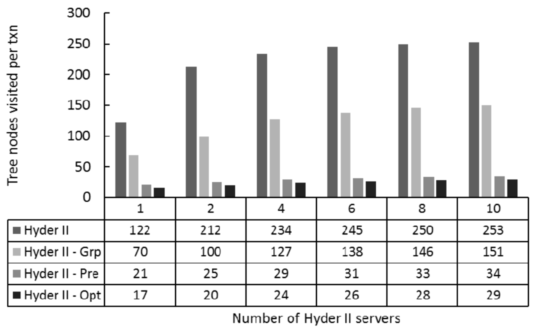


	1	2	4	6	8	10
■ Hyder II	13,541	15,229	13,426	12,949	12,575	12,641
■ Hyder II - Grp	14,493	23,420	22,889	21,143	19,912	18,982
■ Hyder II - Pre	13,604	24,203	43,767	45,318	44,037	42,992
■ Hyder II - Opt	12,629	25,501	44,821	44,059	43,941	43,457

Figure 10: Throughput (committed transactions/sec) for Hyder II and the impact of premeld and group meld optimizations.

As concurrency increases beyond six transaction servers, there is some reduction in throughput of Hyder II-Pre. This is due to more contention and queuing delays for resources, such as log appends (see Figure 9) and network broadcast, which increases transaction processing latency and in turn throttles the executers when they reach their in-flight-transaction limit. That is, the small reduction in throughput is primarily due to the "back-pressure" on the executers due to higher resource contention.

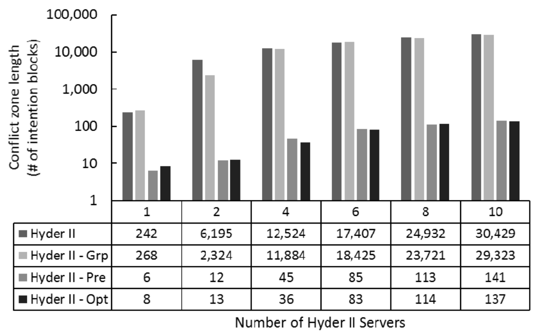
Premeld and group meld help reduce the work done by final meld by reducing the number of tree nodes that final meld visits. Figure 11 reports the number of nodes visited per transaction by final meld for each optimization technique. As is evident, group meld reduces the number of nodes by 2x and premeld reduces them by 8-10x.



	1	2	4	6	8	10
■ Hyder II	122	212	234	245	250	253
■ Hyder II - Grp	70	100	127	138	146	151
■ Hyder II - Pre	21	25	29	31	33	34
■ Hyder II - Opt	17	20	24	26	28	29

Figure 11: Group meld and premeld reduce the number of nodes visited in the final meld thread.

Premeld merges the intention with a recent LCS, eliminating a large fraction of the conflict zone that final meld has to process. Most of the readset and writeset validations that require traversing deep into the database tree are processed by premeld. Final meld mostly terminates high up in the tree, merging the updates into the LCS and creating ephemeral nodes as needed. As a result, even though the number of nodes is reduced by 8-10x, the throughput improvement is only in the range of 3-3.5x.



	1	2	4	6	8	10
■ Hyder II	242	6,195	12,524	17,407	24,932	30,429
■ Hyder II - Grp	268	2,324	11,884	18,425	23,721	29,323
■ Hyder II - Pre	6	12	45	85	113	141
■ Hyder II - Opt	8	13	36	83	114	137

Figure 12: Impact of group meld and premeld on the effective conflict zone observed by the final meld thread.

Figure 12 reports the number of intention blocks in the conflict zone observed by final meld. In our configuration, one intention spans two intention blocks on average. Thus, the number of intentions in the conflict zone is about half of the intention-block numbers reported. As the figure shows, premeld shrinks the conflict zone by 40x-500x by melding the intention with an updated LCS very late in the intention's conflict zone. However, the conflict zone size is unchanged by group meld. Its benefit stems from overlapping nodes and paths in the two intentions—final meld needs to process only one of these overlapped nodes, which reduces the number of nodes processed by final meld. Combining premeld and group meld does not significantly change the number of nodes visited or the conflict zone length, which explains why the two optimizations together do not result in any improvement beyond that of premeld.

While both premeld and group meld reduce the work done by the final meld thread, they perform redundant work in parallel threads. Figure 13 reports on the total number of tree nodes visited per transaction by each optimization. The work done in the critical path of final meld (the pattern-filled bar) decreases with every optimization, while the work done in parallel threads (solid bars) is often higher in aggregate than the sequential final meld thread without any optimizations.

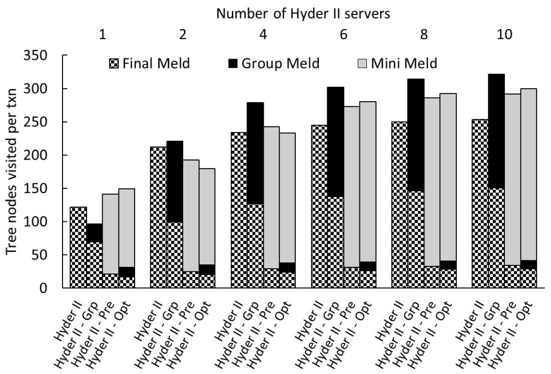


Figure 13: Number of tree nodes visited in different stages of the meld pipeline. The hatched bar corresponds to final meld. The solid bars correspond to group meld and premeld.

6.4.2 *Comparison with Hyder and Tango*

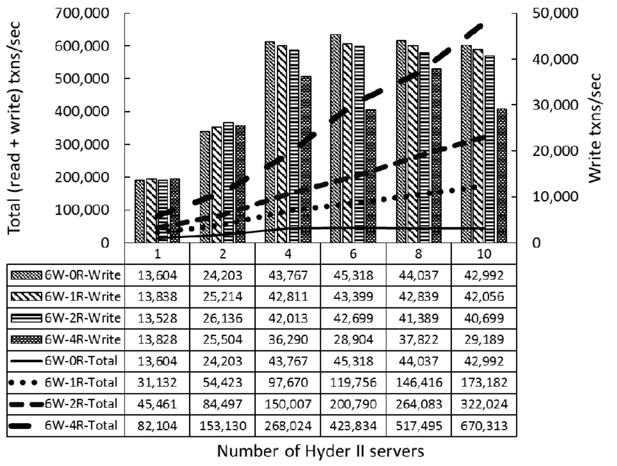
Tango is the system whose architecture is the closest to Hyder II. In this section, we compare them, along with the in-memory setup of Hyder [8]. Tango and Hyder reported throughput for a database of 100K items. To compare them with Hyder II, we repeated the experiment of Section 6.4.1 for a database of 100K items running a workload similar to that used in Hyder and Tango. In this experiment Hyder-II's peak throughput was ~20K tps (compared to 15K tps in Figure 10). Tango reported throughput of 15-25K tps on similar hardware [5]. Thus, Hyder II's performance is comparable to Tango's, in spite of its tree index, which is more expensive to maintain than Tango's hash index. With the premeld optimization, Hyder II's performance is significantly better than Tango's.

The throughput of Hyder's in-memory implementation, using a workload with ~10 operations per transaction was 50-60K tps [8]. The lower throughput of Hyder II's meld operator can be attributed to multiple factors. Experiments for in-memory Hyder involved only one server, and the workload generator limited the conflict zone length to 256. Hyder II needs to operate at much higher degrees of concurrency with tens of thousands of concurrent transactions (as in Figure 12) to mask the latency of serializing and deserializing intentions, appending to the log, and broadcasting intentions. Premeld reduces the conflict zone length for final meld to the same range as the in-memory implementation, which results in a throughput of 50-60K tps for a 100K item database. Other factors contributing to Hyder-II's lower throughput are that Hyder-II's experiments use a slower processor and a distributed shared log, and Hyder II is implemented in C# while Hyder is written in C++.

6.4.3 *Workload with Read-Write Transaction Mix*

Read-only transactions in Hyder II run on a database snapshot (the LCS when the transaction starts) and commit locally, without stressing any of Hyder II's critical resources. Therefore, read-only transaction throughput should scale linearly as we increase the number of servers and the number of read-only transactions. To demonstrate this, we run an experiment that executes an increasing number of read-only transactions with a fixed load of write transactions using the setup of Section 6.4.1.

Both read-only and write transactions execute ten operations; write transactions have two updates and eight reads, as in Section 6.4.1. We use separate executers for read-only transactions. This ensures adding more read-only transactions does not reduce the number of write transactions admitted to the system. We gradually increase the number of read-only transaction executors while keeping the write workload fixed with six dedicated executors per server.



	1	2	4	6	8	10
▨ 6W-0R-Write	13,604	24,203	43,767	45,318	44,037	42,992
▧ 6W-1R-Write	13,838	25,214	42,811	43,399	42,839	42,056
☰ 6W-2R-Write	13,528	26,136	42,013	42,699	41,389	40,699
▦ 6W-4R-Write	13,828	25,504	36,290	28,904	37,822	29,189
— 6W-0R-Total	13,604	24,203	43,767	45,318	44,037	42,992
• • • 6W-1R-Total	31,132	54,423	97,670	119,756	146,416	173,182
▬ ▬ 6W-2R-Total	45,461	84,497	150,007	200,790	264,083	322,024
▬ 6W-4R-Total	82,104	153,130	268,024	423,834	517,495	670,313

Figure 14: Linear scaling of transaction throughput with a mix of read-only and read-write transactions.

Figure 14 plots the throughput of write transactions and of read and write combined ("total"). As we increase the number of servers, the offered load increases. We vary the number of read-only executers from 0 to 4, denoted by 0R, 1R, 2R, and 4R in the figure. With 6W-0R, there are no read-only transactions, re-creating the setup in Section 6.4.1. For brevity, we only report numbers using premeld.

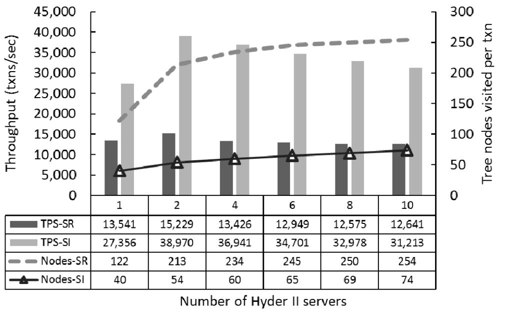
First consider the lines, which plot total throughput (i.e., of read and write transactions) on the left vertical axis (labeled total txns/sec). They correspond to the last four rows of the table. As we increase the number of read-only executors (from 0R to 4R) and servers (from 1 to 10), total throughput scales almost linearly. With 10 transaction servers, total throughput for 6W-4R peaks at ~670K. This linear scalability demonstrates that Hyder II's architecture scales out without partitioning.

The bars in the graph plot the throughput of write transactions on the right vertical axis (labeled write txns/sec). They correspond to the first four rows of the table. As in Section 6.4.1, write throughput peaks at 45K tps with 6 servers (6W-0R). For a given number of servers, as the number of read-only executors increases from 0 to 1 to 2, there is a small decrease in write throughput. This is due to higher CPU contention since more cores are processing transactions (10 in 6W-4R vs. 6 in 6W-0R), thereby reducing the number of cores available to broadcast and deserialize intentions.

With four read-only executors, no cores are dedicated to broadcast and deserialization. Both activities slow down and meld throughput decreases, causing a corresponding drop in write throughput (the right bars for 6, 8 and 10 servers). This shows the importance of reserving enough cores for basic system functions, such as meld, broadcast, and deserialization.

6.4.4 *Snapshot Isolation*

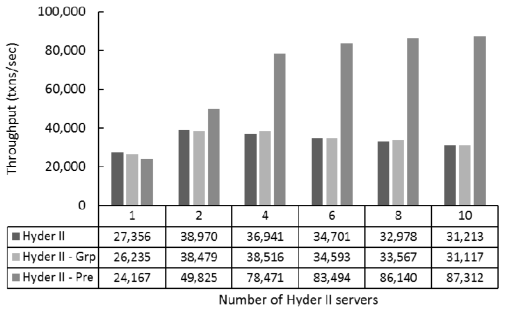
In this experiment, we compare Hyder II's performance when transactions execute with serializable isolation (**SR**) versus snapshot isolation (**SI**). With SI, meld does not need to validate the readset, so the readset is not included in the intention. This considerably reduces the intention sizes and hence the load on many critical resources, such as the log, network, and meld. In this experiment, each transaction performs 8 reads and 2 writes. Therefore, eliminating the readset results in almost a 4x reduction in intention sizes. This results in about a 2.5x improvement in throughput, as shown in Figure 15, which plots the result of running Hyder II with no optimizations. The reduction (3x-4x) in the work done by meld is also evident from the number of tree nodes visited by meld, shown by the lines plotted on the right vertical axis.



	1	2	4	6	8	10
■ TPS-SR	13,541	15,229	13,426	12,949	12,575	12,641
■ TPS-SI	27,356	38,970	36,941	34,701	32,978	31,213
Nodes-SR	122	213	234	245	250	254
Nodes-SI	40	54	60	65	69	74

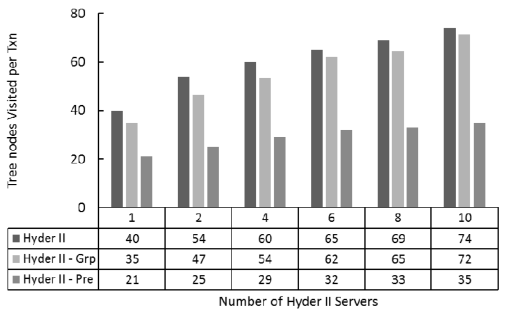
Figure 15: Serializable (SR) vs. snapshot isolation (SI).

The reason why a 4x reduction in the number of nodes yields only a 2.5x increase in throughput is that SI eliminates only readset nodes and reads are cheaper to meld than writes. Reads only require conflict testing while writes require creating ephemeral nodes.



	1	2	4	6	8	10
■ Hyder II	27,356	38,970	36,941	34,701	32,978	31,213
■ Hyder II - Grp	26,235	38,479	38,516	34,593	33,567	31,117
■ Hyder II - Pre	24,167	49,825	78,471	83,494	86,140	87,312

Figure 16: Impact of the premeld and group meld optimizations with snapshot isolation.



	1	2	4	6	8	10
■ Hyder II	40	54	60	65	69	74
■ Hyder II - Grp	35	47	54	62	65	72
■ Hyder II - Pre	21	25	29	32	33	35

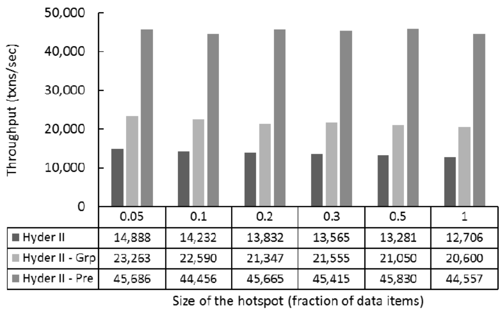
Figure 17: Only premeld results in a reduction in number of tree nodes visited by final meld with SI.

In Figure 16 we report the benefits of group meld and premeld for transactions executing with SI. Hyder II with premeld continues to demonstrate 2x-3x higher throughput than without the optimization. This benefit arises from premeld's ability to reduce the number of nodes visited by final meld (see Figure 17). Group meld's 10% reduction does not significantly improve performance, because in SI each intention only has two writes. Therefore, there are fewer overlapping nodes in two consecutive intentions.

6.4.5 *Varying Workload Parameters*

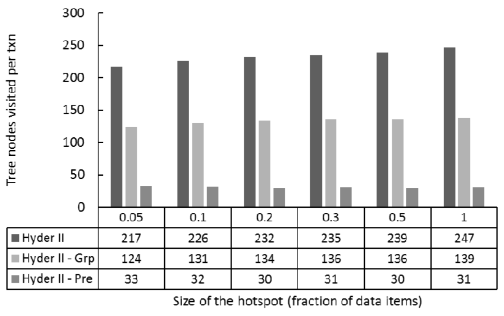
Our workload generator is able to vary different parameters, such as the distribution used to select data items accessed by a transaction, number of operations per transaction, database size, and number of read and write operations per transaction. We evaluated Hyder II's performance for a variety of these workloads and observed similar benefits with premeld providing 3x-3.5x improvement in throughput. We present the results varying the access distribution here; more results are included in Appendix B.

The goal of this experiment is to evaluate the impact of data access distribution on the premeld and group meld optimizations. The workload creates a hotspot where fraction x of data items is accessed by fraction $(1.0 - x)$ of operations. We vary x from 0.05 to 1.0; $x = 1$ results in the uniform access distribution used earlier.



	0.05	0.1	0.2	0.3	0.5	1
■ Hyder II	14,888	14,232	13,832	13,565	13,281	12,706
■ Hyder II - Grp	23,263	22,590	21,347	21,555	21,050	20,600
■ Hyder II - Pre	45,686	44,456	45,665	45,415	45,830	44,557

Figure 18: Premeld is effective with skewed data accesses.



	0.05	0.1	0.2	0.3	0.5	1
■ Hyder II	217	226	232	235	239	247
■ Hyder II - Grp	124	131	134	136	136	139
■ Hyder II - Pre	33	32	30	31	30	31

Figure 19: Number of tree nodes visited as a function of skew.

Figure 18 plots the throughput and Figure 19 plots the number of tree nodes visited by the final meld. As the skew increases, the probability of a conflict increases, which would increase the abort rates. Interestingly, without any optimizations, Hyder II's throughput increases with increase in skew. With skew of 0.05, the abort rate is slightly higher, about 0.14% compared to 0.02% with uniform. However, with increasing skew, the work done by final meld decreases, since transactions access similar data, thus allowing meld to terminate higher in the tree. Since final meld is the bottleneck, speeding it up results in higher throughput. Notice that skew has negligible impact on the work done by final meld when premeld is turned on, and hence negligible effect on throughput. However, the benefit of premeld still prevails with Hyder II-Pre's throughput being 3.5x that of Hyder II.

6.4.6 Analyzing Premeld

Recall that with *t* premeld threads, an intention is premelded against the state created by the $(t \times d)^{th}$ intention that precedes it. Therefore, for a fixed *t*, the smaller the value of the premeld distance *d*, the smaller the conflict zone length for the final meld thread, which in turn increases throughput. In this experiment, we evaluate the impact on *d* by setting *t=5*. Figure 20 reports transaction throughput as a function of premeld distance with five premeld threads, which empirically validates the expected behavior. Since *d*=10 results in best throughput, we used this setting in all our experiments.

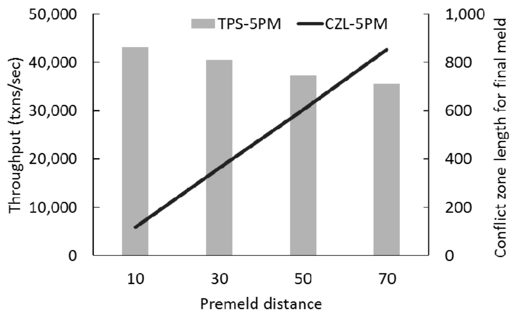


Figure 20: Analyzing the behavior of premeld by varying the premeld distance.

7. RELATED WORK

Hyder II is based on optimistic concurrency control and log-structured storage. We discuss each in turn, followed by systems that have some similarity with Hyder II.

Optimistic concurrency control (OCC) was introduced in [19], and its performance studied in [1][2][16][21][26][30]. It was initially unpopular due to lower throughput than locking, but has attracted interest recently as its non-blocking behavior is highly desirable for parallel hardware. Tashkent uses a centralized OCC validator over distributed data [11]. A distributed B-tree implementation with OCC is in [3], but it uses distributed transactions, single-version data, and simple version ids for conflict detection. OCC for an in-memory database is described in [20]. Hybrid schemes have also been proposed to combine the benefits of locking and OCC [21][26][29]. None of these systems are similar to Hyder II.

Log-structured storage has been widely studied in file systems [3] [12][24][27] and database systems [5][7][8][15][22][25][28][32] [33]. It was initially proposed for write-once media [12], and then used to improve disk I/O performance [27]. It has regained popularity due to the recent adoption of flash-based storage [1][5][7][22], which inherently does copy-on-write.

The main performance benefit of log-structured storage is from batching updates and from using sequential rather than random I/O [27]. Some implementations batch updates into larger sequential writes [15][22][33]. Others merge update batches into a hierarchical snapshot of the data, such as bLSM [28]. However, none of these systems supports a server cluster with shared-storage and ACID transactions, like Hyder II. Tokutek [32] uses a log-structured multi-versioned index, but with locking, not OCC.

Amoeba [24] is a log-structured distributed file system that uses OCC. Like Hyder II, it uses copy-on-write for file updates, and uses OCC to check update conflicts. Unlike Hyder II, it serializes conflict checks across servers and does not consider replication.

The system closest to Hyder II is Tango [5], which implements a distributed object store on the CORFU log manager [3]. CORFU provides the interface of a distributed SSD and takes care of distributed wear-leveling, data distribution, fault tolerance, and scalability. Its concurrency control protocol is inspired by Hyder [7]. But it uses a hashed access method and hence does not require optimized validation over trees or merging of trees, i.e., meld. Since it uses hashing, it suffers the usual weakness of failing to handle range predicates, especially over continuous domains.

OXenstored [14] is another system that uses a similar log-based strategy for OCC conflict detection. It works over multi-version tries [13] and does very coarse-grain conflict detection, and hence has much lower throughput than Hyder II.

Eve [17] uses a similar copy-on-write tree as Hyder II, but a very different technique for ensuring all replicas reach the same state. It batches requests that are not conflicting and executes requests within a batch in parallel. Operations that require creating a new tree node are postponed until the end of the batch to achieve determinism. It uses state machine replication to check if enough replicas reached the same state. If not, it reruns the batch sequentially.

Calvin [31] is a transaction system that scales out to multiple servers. As in Hyder II, determinism plays a prominent role in Calvin; both approaches rely on determinism to avoid synchronization across servers. Calvin replicates requests to run transactions and executes them deterministically in a pre-determined order. By contrast, Hyder II replicates the result of executing transactions, totally orders them in a shared log, and then melds them deterministically.

8. CONCLUSION

We presented an optimized version of the optimistic concurrency control algorithm in [8], for a log-structured, multi-versioned, tree-structured database. The algorithm, called meld, performs deterministic roll-forward of the log, analyzing each successive transaction in the log for conflicts. Our optimized version of meld uses pipeline parallelism to preprocess the log and thereby speed up the algorithm by 3x or more. We showed that the system scales out over multiple servers without partitioning the database.

A follow-on paper by two of the authors [6] leverages an approximate partitioning of the workload to parallelize meld into multiple threads, where each thread executes over a different partition of the transactions. Transactions that span partitions execute on a separate multi-partition thread, which is synchronized with each of the single-partition threads with which the transactions might conflict. To minimize this synchronization, some static analysis of transactions would likely be beneficial, such as that proposed in [9][23].

Acknowledgements

The authors are heavily indebted to Mahesh Balakrishnan for providing us with the CORFU log service and extending it for our experimental environment. We also thank Colin Reid and Soner Terek for input on premeld, the .Net runtime team for help with tooling and techniques to optimize performance, Ming Wu for help with the original Hyder implementation, and Andy Pavlo for many excellent suggestions that helped us improve the paper.

9. REFERENCES

[1] Adya, A., R. Gruber, B. Liskov, U.Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. SIGMOD 1995, pp. 23-34.

[2] Agrawal, D., A. Bernstein, P. Gupta, S. Sengupta. Distributed optimistic concurrency control with reduced rollback. Distributed Computing 2, 1 (1987), pp. 45-59.

[3] Aguilera, M.K., W.M. Golab, and M.A. Shah: A practical scalable distributed B-tree. PVLDB 1(1): 598-609, 2008.

[4] Balakrishnan, M., D. Malkhi, J.D. Davis, V. Prabhakaran, M. Wei, T. Wobber: CORFU: A distributed shared log. ACM Trans. Comput. Syst. 31(4): 10 (2013).

[5] Balakrishnan, M., D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J.D. Davis, S. Rao, T. Zou, A. Zuck: Tango: distributed data structures over a shared log. SOSP 2013, pp. 325-340

[6] Bernstein, P.A. and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. IEEE Data Eng. Bull 38, 1 (March 2015).

[7] Bernstein, P.A., C.W. Reid, S. Das: Hyder - A transactional record manager for shared flash. CIDR 2011, pp. 9-20

[8] Bernstein, P.A., C.W. Reid, M. Wu, X. Yuan: Optimistic concurrency control by melding trees. PVLDB 4(11): 944-955 (2011).

[9] Bernstein, P.A., D.W. Shipman, J.B. Rothnie Jr.: Concurrency Control in a System for Distributed Databases (SDD-1). ACM TODS 5(1): 18-51 (1980).

[10] Cooper, B.F, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears: Benchmarking cloud serving systems with YCSB. SoCC 2010, pp. 143-154.

[11] Elnikety, S., S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. EuroSys 2006, pp. 117-130.

[12] Finlayson, R., and D. Cheriton. Log Files: An extended file service exploiting write-once storage. SOSP 1987, pp. 139-148.

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

[14] Gazagnairem, T. and V. Hanquez: OXenstored—An efficient hierarchical and transactional database using functional programming with reference cell comparisons. ICFP 2009, pp. 203-214.

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

[16] Gruber, R. E. Optimistic concurrency control for nested distributed transactions, 1989.

[17] Guibas, L.J., and R. Sedgewick: A Dichromatic Framework for Balanced Trees. FOCS 1978, pp. 8-21.

[18] Kapritsos, M., Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. OSDI 2012, pp. 237-250.

[19] Kung, H. T., and J.T. Robinson. On optimistic methods for concurrency control. ACM TODS 6, 2 (June 1981), 213-226.

[20] Larson, P.-A., S. Blanas, C. Diaconu, C. Freedman, J.M. Patel, M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. PVLDB 5, 4 (2011): 298-309.

[21] Lausen, G. Concurrency control in database systems: A step towards the integration of optimistic methods and locking. ACM 1982, pp. 64-68.

[22] Lee, S.-W., and B. Moon. Design of flash-based DBMS: An in-page logging approach. SIGMOD 2007, pp. 55-66.

[23] Mu, S., Y. Cui, Y. Zhang, W. Lloyd, J. Li. Extracting more concurrency from distributed transactions. OSDI 2014, pp. 479-494.

[24] Mullender, S. J., and A.S. Tanenbaum. A distributed file service based on optimistic concurrency control. SOSP 1985, pp. 51-62.

[25] O'Neil, P.E., E. Cheng, D. Gawlick, E.J. O'Neil. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (June 1996): 351-385.

[26] Phatak, S. and B.R. Badrinath, Bounded locking for optimistic concurrency control. Rutgers Univ., Dept. of Computer Science, Tech. Report #DCS-TR-380.

[27] Rosenblum, M., and J.K. Ousterhout. The design and implementation of a log-structured file system. SOSP 1991, pp. 1-15.

[28] Sears, R., and R. Ramakrishnan. blsm: A general purpose log structured merge tree. SIGMOD 2012, pp. 217-228.

[29] Sheth, A. P., and M.T. Liu. Integrating locking and optimistic concurrency control in distributed database systems. ICDCS 1986, pp. 89-99.

[30] Thomasian, A., and E. Rahm. A new distributed optimistic concurrency control method and a comparison of its performance with two-phase locking. ICSCS 1990, pp. 294-301.

[31] Thomson, A., T. Diamond, S-C Weng, K. Ren, P. Shao, D.J. Abadi: Calvin: fast distributed transactions for partitioned database systems. SIGMOD 2012, pp. 1-12

[32] Tokutek. http://www.tokutek.com/.

[33] Wu, C.-H., L.-P Chang, T.-W. Kuo. An efficient R-tree implementation over flash-memory storage systems. GIS 2003, pp. 1724.

APPENDIX

A. SKETCH OF MELD

We present a simplified version of meld, to give the reader a deeper understanding of the algorithm. However, none of it is required background for the rest of this paper.

Each node is identified by a version number (abbr. VN), which is calculated from its log address. Thus, each new version of a node has a new VN. For example, in Figure 3, D′, B′, and C′, have different VN's than D, B, and C, respectively.

Each node also has some metadata that is used by meld to determine whether the intention that wrote the node experienced a conflict. The metadata includes an **Altered flag**, **Depends-On flag**, **source structure version** (**SSV**), and **source content version** (**SCV**). This is simplified metadata; a complete description is in [8].

For a node n in an intention I, Altered(n) is TRUE if T(I) updated n's payload. The flag DependsOn(n) is TRUE if I depends on n's payload not having changed during T(I)'s execution, that is, T(I) read n and ran with repeatable read or serializable isolation level. Additional metadata is needed for phantom avoidance; see [8].

For a node n in an intention I, SSV(n) is the VN of the node in I's snapshot that has the same key value as n. For example, in Figure 3, SSV(C′) = VN(C), SSV(B′) = VN(B), and SSV(D′) = VN(D). If n is a new node inserted by I, then SSV(n) = null, such as SSV(A) in Figure 3.

For a node n in an intention I, SCV(n) is the VN of the node that first generated the payload of SSV(n). If SCV(n) = SSV(n) in I, then the transaction that produced the node whose VN is SSV(n) also updated n's payload. If SCV(n) < SSV(n), then a node in SCV(n)'s subtree was updated after the transaction that produced SCV(n)'s payload. This caused a copy-on-write of SCV(n)'s node, which led directly or indirectly (though subsequent updates to that subtree) to a node whose VN is SSV(n).

Meld works by a recursive preorder traversal of I, starting at I's root and comparing each node n in I with the corresponding node n_L in LCS. There are two cases. In case one, n_L has the same key as n and meld works as follows:

- If n is NULL or outside of I, then return n
- If SSV(n) = VN(n_L), then n and its subtree were not changed after T(I) read them. Therefore, meld can simply replace n_L by n, which also replaces n_L's subtree by n's subtree. Otherwise, SSV(n) ≠ VN(n_L), so something in n's subtree changed and we have to determine if that change implies a conflict.
- If Altered(n) = TRUE and either

  (Altered(n_L) = TRUE and SCV(n) ≠ VN(n_L)) or
  (Altered(n_L) = FALSE and SCV(n) ≠ SCV(n_L)),

  then T(I)'s update of n conflicts with a write in its conflict zone, so I aborts.
- Otherwise, if DependsOn(n) = TRUE and either

  (Altered(n_L) = TRUE and SCV(n) ≠ VN(n_L)) or
  (Altered(n_L) = FALSE and SCV(n) ≠ SCV(n_L)),

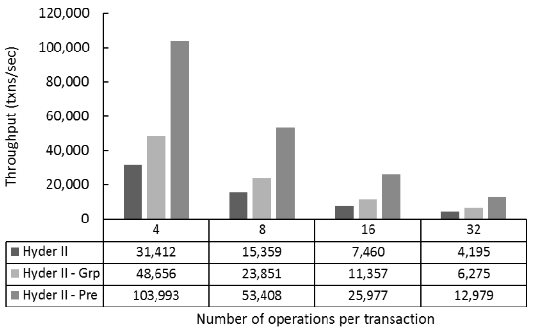
  then T(I)'s read of n conflicts with a write in its conflict zone, so I aborts.
- Otherwise, copy n into an ephemeral node n_e.
  - Set the left child of n_e to be the result of melding n's and n_L's left children
  - Set the right child of n_e to be the result of melding n's and n_L's right children.

In case two, there is no n_L that has the same key as n. This complicates the recursion and is described in detail in [8].

B. ADDITIONAL EXPERIMENTS

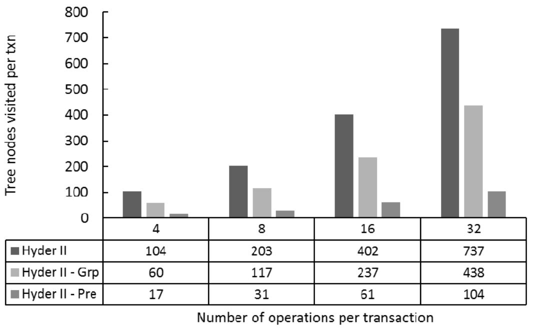
We now present more experiments analyzing Hyder II's performance by varying various workload parameters such as the number of operations per transaction (or transaction size) and the fraction of update operations in a ten-operation write transaction. Other workload parameters are set to the default values described in Section 6.1. The number of transaction servers is set to 6.

As the number of operations per transaction increases, the number of tree nodes included in an intention increases. This increases the load on each of Hyder II's critical resources. In this workload, we fix the fraction of update operations in a transaction to 0.2, with at least one update operation in the transaction. Therefore, as the transaction size increases, the number of update operations in the transaction also increases. This results in more work for the meld pipeline, since meld has to create ephemeral nodes for the updated nodes for parts of the tree that were updated concurrently. Thus, as the transaction size increases, we expect the transaction throughput to decrease proportionately, as is observed in Figure 21. The increase in work done by final meld is evident in Figure 22. However, premeld continues to be as effective, with a 3x performance improvement and ~7x reduction in the number of nodes visited. The other contributor to the cost of final meld, the number of ephemeral nodes created, shows a similar pattern. With four operations per transaction, the meld pipeline in Hyder II-Pre creates an average of 23 ephemeral nodes per transaction. With transaction size of 32, it grows to 171 ephemeral nodes per transaction.



	4	8	16	32
■ Hyder II	31,412	15,359	7,460	4,195
■ Hyder II - Grp	48,656	23,851	11,357	6,275
■ Hyder II - Pre	103,993	53,408	25,977	12,979

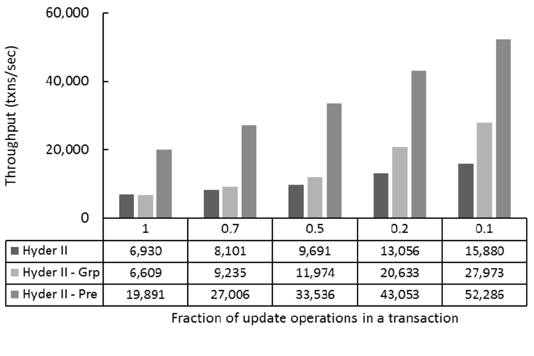
Figure 21: Varying the number of operations in a transaction. Every transaction has at least one write operation.



	4	8	16	32
■ Hyder II	104	203	402	737
■ Hyder II - Grp	60	117	237	438
■ Hyder II - Pre	17	31	61	104

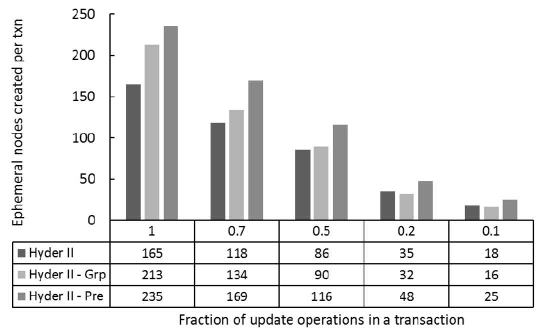
Figure 22: Premeld significantly reduces the number of tree nodes visited by final meld even as we vary the number of operations in a transaction.

A similar increase in the number of ephemeral nodes created by the meld pipeline is observed as we increase the number of update operations in a transaction. In this experiment, we fix the number of operations per transaction at 10 and vary the fraction of update operations from 0.1 to 1.0. As the fraction of update operations decreases, the throughput increases (see Figure 23). Figure 24 plots the number of ephemeral nodes created by the meld pipeline. It shows that a higher update fraction results in the creation of more ephemeral nodes, because updates lead to the creation of ephemeral ancestor nodes. As expected, premeld and group meld result in slightly more ephemeral nodes being created.



	1	0.7	0.5	0.2	0.1
■ Hyder II	6,930	8,101	9,691	13,056	15,880
■ Hyder II - Grp	6,609	9,235	11,974	20,633	27,973
■ Hyder II - Pre	19,891	27,006	33,536	43,053	52,286

Figure 23: Impact of varying the fraction of update operations per transaction on the throughput.



	1	0.7	0.5	0.2	0.1
■ Hyder II	165	118	86	35	18
■ Hyder II - Grp	213	134	90	32	16
■ Hyder II - Pre	235	169	116	48	25

Figure 24: As the number of update operations per transaction increases, so does the number of ephemeral nodes created.

C. DETERMINISM OF PREMELD WITH EPHEMERAL NODES

We present an example of an intention that is premelded with different database states at two different servers, leading to an ephemeral node identity that has different content at the two servers. The example is shown in Figure 25.

The initial database state, S_0, is shown in row 1. It is a tree whose root is A with children B and D. The number "0" in parentheses after each node indicates that it was written by transaction T_0 (not shown), which initialized the database.

The log is shown in row 2. It contains intentions written by a sequence of transactions $T_1 \ldots T_4$, all of which executed against state S_0. A dashed line indicates a pointer to a node (shown in gray) outside of the transaction's intention. The number in parentheses after each node identifies the transaction that wrote it. The transactions performed the following actions:

- T_1 updated the content of B, which generated a new version of node A due to copy-on-write.
- T_2 updated the content of D, which generated a new version of node A due to copy-on-write.
- T_3 inserted node C, which generated new versions of nodes B and A due to copy-on-write.
- T_4 inserted node E, which generated new versions of nodes D and A due to copy-on-write.

The sequence of states in the third row shows the result of applying meld to each transaction, in turn. None of the transactions conflict with each other, so they all commit. Since all transactions ran against S_0, for each transaction after T_1 meld produces ephemeral nodes, denoted by $e_1 \ldots e_5$.

In row 4, premeld at server X processes T_4 against state S_1, producing T_{4x}. Since it runs on a separate thread, the ephemeral node that it generates has a compound identifier [1,1], comprised of its premeld thread id followed by a sequence number. The final meld thread melds T_1, T_2, and T_3 as before. However, instead of melding T_4, it melds the result of premeld, T_{4x}, producing S_{4x}. Notice that meld recognizes that $B(e_2)$ is the immediate successor of state B(1) and therefore retains it in the meld result. However, since it merges the subtree $B(e_2)$ in S_3 with the updated D subtree that melds D(2) in S_3 and D(4) in T_{4x}, it produces a new ephemeral root, $A(e_4)$.

In row 5, at a different server Y, premeld processes T_4 against state S_2, unlike X which processed premeld against S_1. It produces T_{4y}. Like at server X, server Y's final meld thread melds T_1, T_2, and T_3 and then melds the result of premeld, T_{4y}, producing S_{4y}.

Notice that servers X and Y both generate a node identified by $e_{1,1}$. However, they are entirely different nodes. Suppose a transaction executes at server Y and produces an intention that references node $D(e_{1,1})$. When that intention is melded at server X, the reference will be incorrectly interpreted as pointing to the root $A(e_{1,1})$, which is the wrong node and is from an earlier state. From this point on, the system at node X will go haywire.

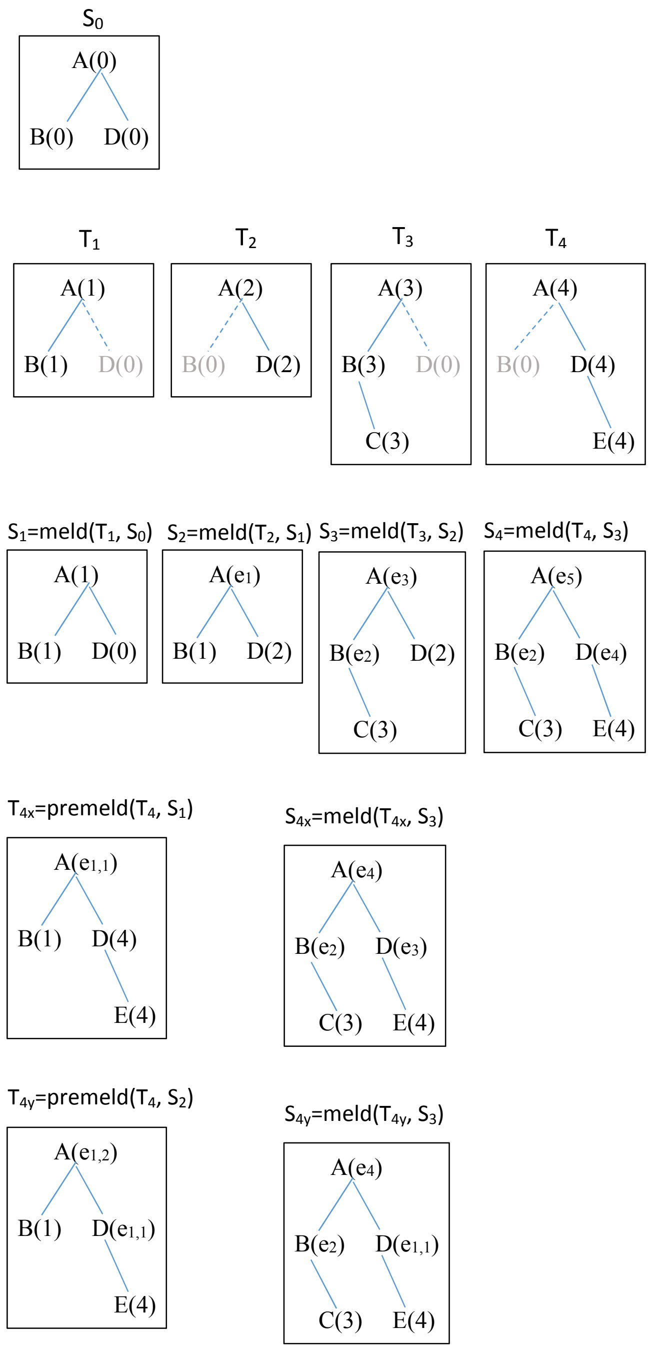

S_0
A(0) B(0) D(0)
T_1
A(1) B(1) D(0)
T_2
A(2) B(0) D(2)
T_3
A(3) B(3) D(0) C(3)
T_4
A(4) B(0) D(4) E(4)
S_1=meld(T_1, S_0)
A(1) B(1) D(0)
S_2=meld(T_2, S_1)
A(e_1) B(1) D(2)
S_3=meld(T_3, S_2)
A(e_3) B(e_2) D(2) C(3)
S_4=meld(T_4, S_3)
A(e_5) B(e_2) D(e_4) C(3) E(4)
T_{4x}=premeld(T_4, S_1)
A($e_{1,1}$) B(1) D(4) E(4)
S_{4x}=meld(T_{4x}, S_3)
A(e_4) B(e_2) D(e_3) C(3) E(4)
T_{4y}=premeld(T_4, S_2)
A($e_{1,2}$) B(1) D($e_{1,1}$) E(4)
S_{4y}=meld(T_{4y}, S_3)
A(e_4) B(e_2) D($e_{1,1}$) C(3) E(4)

Figure 25: Running premeld against different database states