Hyder – A Transactional Record Manager for Shared Flash

Philip A. Bernstein
Microsoft Corporation
philbe@microsoft.com

Colin W. Reid
Microsoft Corporation
colinre@microsoft.com

Sudipto Das†
University of California, Santa Barbara
sudipto@cs.ucsb.edu

ABSTRACT

Hyder supports reads and writes on indexed records within classical multi-step transactions. It is designed to run on a cluster of servers that have shared access to a large pool of network-addressable raw flash chips. The flash chips store the indexed records as a multiversion log-structured database. Log-structuring leverages the high random I/O rate of flash and automatically wear-levels it. Hyder uses a data-sharing architecture that scales out without partitioning the database or application. Each transaction executes on a snapshot, logs its updates in one record, and broadcasts the log record to all servers. Each server rolls forward the log against its locally-cached partial-copy of the last committed state, using optimistic concurrency control to determine whether each transaction commits. This paper explains the architecture of the overall system and its three main components: the log, the index, and the roll-forward algorithm. Simulations and prototype measurements are presented that show Hyder can scale out to support high transaction rates.

1. INTRODUCTION

The hardware platform for database systems is undergoing major changes with the advent of solid-state storage devices, high-speed data center networks, large main memories, and multi-core processors. These changes enable new database software architectures. This paper explores one such architecture: a log-structured multiversion database, stored in flash memory, and shared by many multi-core servers over a data center network. The architecture, shown in Figure 1, is embodied in a prototype system called Hyder.

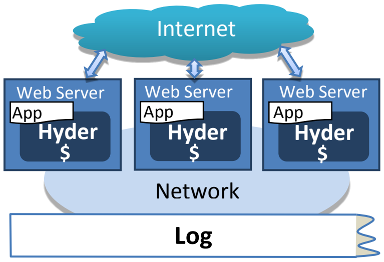


Figure 1 The Hyder architecture

Hyder is a transactional indexed-record manager. It supports read and write operations on indexed records within classical multi-step transactions. This is the basic functionality of the storage engine of a SQL database system. Hyder is designed to run on a cluster of servers that have shared access to a large pool of network-addressable storage, commonly known as a data-sharing architecture. Ideally, the storage is comprised of raw flash chips, although solid-state disks and possibly hard disks could work as well. Its main feature is that it scales out without partitioning the database or application. It is therefore well-suited to a data center environment, where scaling out is important and where specialized flash hardware and networking can be cost-effective.

1.1 Today's Alternative to Hyder

To understand the value of Hyder's no-partition scale-out feature, consider today's alternative: a data center architecture for database-based services, shown in Figure 2. The database is partitioned across multiple servers. The parts of the application that make frequent access to the database are encapsulated in stored procedures. The rest of the application runs in servers, either co-located with the web server or in a separate layer of servers (as shown). The application servers usually have a cache (denoted "$" in the figure) to minimize accesses to the database. Often, the application servers are partitioned, so their caches can be partitioned, thus enabling more of the database to be cached. Typically the application is responsible to choose which data to cache, to refresh this cache periodically, and to maintain cache coherence across the servers.

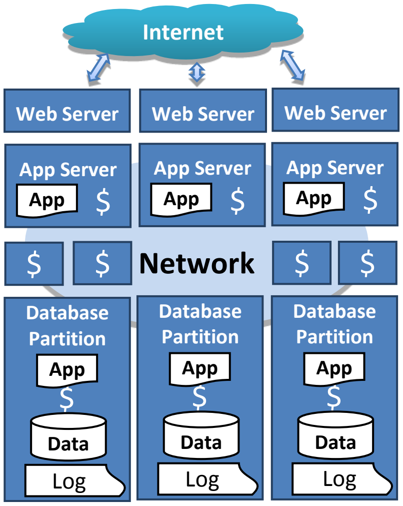


Figure 2 Scale-out architecture with partitioning

Some data that needs to be cached cannot be partitioned, such as a many-to-many relationship that is traversed in both directions. The friend-status relation for social networking is a well-known example. Such data is stored in separate cache servers.

Designing such systems is hard and requires special skills. The designer needs to choose a partition strategy that balances the load across servers, and minimizes or avoids distributed transactions (primarily due to the expense of two-phase commit). Migration mechanisms are needed to enable the system to grow by splitting and relocating overloaded partitions. And the application programmer needs to split application logic between the layers of servers, which implies distributed debugging.

1.2 Benefits of Hyder Architecture

Hyder simplifies application design by avoiding partitioning, distributed programming, layers of caching, and load balancing. Since it uses a data-sharing architecture, all servers can read from and write to the entire database (see Figure 1). This enables the database software to run in the application process, which simplifies application development by avoiding any distributed programming. It also avoids the expense of remote procedure calls

† Work performed while employed at Microsoft Research.

5th Biennial Conference on Innovative Data Systems Research (*CIDR '11*)
January 9-12, 2011, Asilomar, California, USA

between the application and database. Since each server caches data it recently accessed or updated, the content of each server's cache reflects the accesses of transactions that ran recently on that server. So there is no need to partition the cache, either across servers of a given type or across layers of processes, because there are no layers. And since any transaction can execute on any server, load balancing is simply a matter of directing each transaction request to a lightly-loaded server.

In Hyder, each update transaction executes on one machine and writes to one shared log. Hence, it does not require two-phase commit. This saves message delays. It also avoids two-phase commit's blocking behavior, which requires operator intervention or heuristic decisions, both of which are undesirable, especially in a cloud-computing system.

Hyder simplifies cache coherence by using a multi-versioned database. This means there is no update-in-place. Therefore, although caches on different servers can have different versions of data, the caches are inherently coherent in that all copies of a given version are identical. Therefore, a query can be decomposed into subqueries that run against the same database version on different servers, e.g., to improve the response time of a query that accesses a lot of data. Moreover, each server continually and eagerly refreshes its cache, so it is rarely more than a few tenths of a second behind the last committed database state in the log.

Hyder scales out well without partitioning. Hyder's scaling limit of update transactions depends mostly on total update-transaction workload across all servers and on network and storage performance. Since there is no server-to-server communication, it does not depend on the number of servers.

Hyder's scale-out limits can be increased through careful application partitioning. However, most applications will never require it. For example, the effort to design a partitioned application is a wasted expense for a new application that never becomes popular, or whose popularity can be served by Hyder's scale-out limits. An application that experiences a few days of fame may need to scale out on short notice to avoid a success-disaster. Moreover, if an application does become popular quickly, Hyder can scale out without application redesign, thereby buying time for the application vendor to redesign for greater scale-out.

Some application scenarios that might benefit from Hyder's ability to scale out without partitioning are as follows:

- A cloud-based service for database applications may run tenants on a large cluster of servers. Most of these applications are likely to be small and can easily run on a single server, but some will need to scale out quickly.
- A packaged system that has a small number of servers can be purchased to run transaction processing applications. As usage increases, the system can grow incrementally by adding servers, without any software or database reconfiguration.
- An application that processes updates from social networks that form dynamically is not easily partitioned. Examples include multi-player games, real-time advertising of short-term sales events, and on-line news of a real-world disaster.

1.3 Hyder Software Layers

Data-sharing architectures are not new. They are supported by IBM DB2 Data Sharing [18], Oracle RAC [9], and Oracle (formerly DEC) Rdb [23]. Hyder differs from these systems in two ways. First, these systems usually require a soft partitioning of the applications, so that ownership of database pages does not have to move too frequently between servers. By contrast, Hyder requires no partitioning of applications to run well, though it can benefit from such partitioning. Second, Hyder uses a radically different architecture, with no lock manager. We therefore expect it to exhibit different performance tradeoffs than locking-based solutions; such a comparison is postponed as future work.

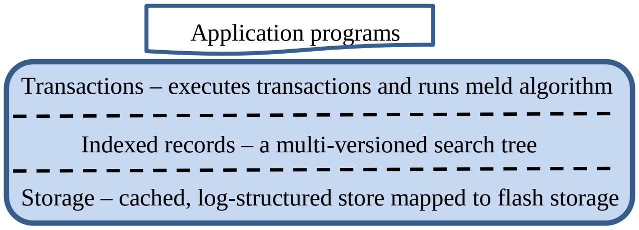

Application programs
Transactions – executes transactions and runs meld algorithm
Indexed records – a multi-versioned search tree
Storage – cached, log-structured store mapped to flash storage

Figure 3 Hyder software layers

Hyder has the following three layers, shown in Figure 3:

- The storage layer offers a highly-available, load-balanced, self-managing, cached, log-structured store mapped to shared flash storage. The log *is* the database.
- The indexed record layer supports multi-versioned binary search trees mapped to log-structured storage.
- The transaction layer executes transactions. It runs a log roll-forward algorithm that continually refreshes the database cache. It uses optimistic concurrency control [20] to ensure transaction isolation.

This layering covers the functionality in the box labeled Hyder in Figure 1. There needs to be an application programming interface on top, such as SQL, but that is outside the scope of Hyder. It should be straightforward for Hyder to support an ISAM API or an API implementation that is layered on a narrow storage interface, such as that of MySQL [24]. Unfortunately, not all SQL database systems are so well modularized.

1.4 The Life of a Transaction

The life of an update transaction *T* is illustrated in Figure 4. *T* executes on one server, called *T*'s **executer**. When *T* starts, it is given the latest local copy of the database root, which defines a static snapshot of the entire database (step (1) in Figure 4). *T*'s updates are stored in a transaction-local cache. When *T* finishes executing, the after-images of its updates are gathered into a record called its **intention** (step (2)), which is broadcast to all servers (step (3)) and appended to the log (step (4)). For serializable isolation, *T*'s readset is included in the intention too.

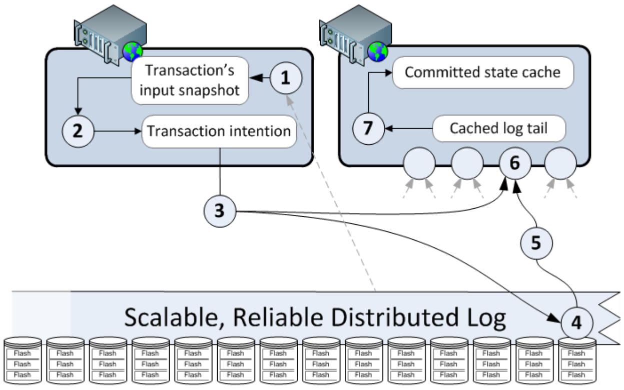


Figure 4 Steps in the life of a transaction

The log protocol ensures intentions are totally ordered, none are lost, and the offset of each intention is made known to all servers (step (5)). Since each server receives every intention and its offset, it can assemble a local copy of the tail of the log (step (6)). A server can detect if it failed to receive an intention from a hole in the sequence. In that case it can read the missing intention from the log.

Each server rolls forward the log on its cached partial-copy of the last committed state (step (7)). Unlike conventional database systems, appending *T'*s intention record *I* to the log does not commit *T*. Instead, when a server rolls forward *I*, it runs a procedure called **meld** that determines whether *T* actually committed. Conceptually speaking, *I* contains a reference *R* to *T's* **snapshot**, which is the last committed transaction in the log that contributed to the database state that *T* read (Figure 5). The intentions between *R* and *I* are called *T's* **conflict zone**. The meld procedure determines if any committed transaction in *T's* conflict zone includes an operation that conflicts with *T* with respect to *T's* isolation level. If there are no conflicts, then *T* is committed and the meld procedure merges *I* into the server's cached partial-copy of the database (the output of step (7)). Otherwise, *T* aborted. Since all servers (including *T's* executer) read the same log, they all make the same commit/abort decision regarding *T*.

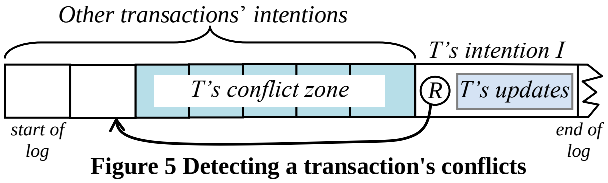


Figure 5 Detecting a transaction's conflicts

A transaction *T* executes in only one server (i.e., steps 1-3). However, all servers roll forward *T* using the meld procedure, including *T'*s executor. Thus, even *T'*s executor does not know whether *T* committed or aborted until after it melds *T'*s intention. At that point, it can notify *T's* outcome to *T's* caller.

Each server runs independently with no cross-talk to other servers. In particular, there is no lock manager and hence no locking bottleneck to constrain scale-out. Moreover, there is no two-phase commit. The only point of arbitration between servers is the atomic append of an intention to the log.

Of course, the architecture does have points of contention that can limit performance of update transactions: appending intentions to the log, broadcasting intentions to all servers, melding the log at each server, and aborting transactions due to conflicts. We quantify their effect in Section 5. Our measurements and simulations show that with today's technology, the system's throughput can reach 80K transactions per second (TPS) using a micro-benchmark of transactions with ten operations. Scale-out is practically unlimited for read-only transactions, since they do not consume any critical system resource; in particular, they are not melded. They can read a recent snapshot—the same snapshot at all servers, since all servers can access all versions of the whole database.

1.5 Hardware Trends

Hyder's architecture is enabled by four main hardware trends: high performance data center networks, large main memory, many-core processors, and solid state storage.

Networks – Commodity data center networks are 1 Gb/sec, with 10 Gb/sec available now and 40 and 100 Gb/sec already standardized. This enables many servers to share storage and broadcast the log with high performance. Network broadcasts are a bottleneck, but network bandwidth and latency will improve over time.

Main memory – At today's DRAM prices and with 64-bit processors, commodity servers can maintain huge in-memory caches. This reduces the rate at which servers need to access storage for cache misses.

Many-core – Given the declining cost of computation, Hyder can afford to squander computation by rolling forward the log on all servers to maintain consistent views across servers without server-to-server communication and by dedicating several cores per server to the meld activity.

Storage – Raw flash memory offers ~10^4 more I/O operations per second per gigabyte (GB) than hard disks. It costs ~100μs to read a 4KB page, ~200μs to write one, and there is no performance benefit to sequential access. This makes it feasible to spread the database across a log with less concern for physical contiguity than with hard disks. Solid-state disks (SSDs) are slower, but are improving. Sequential writes on some SSDs are already faster.

Although flash densities can double only two or three more times, other nonvolatile technologies are coming, notably phase-change memory (PCM). Instead of flash, Hyder might work with hard disks if servers have a large enough database cache to avoid too many cache misses, since they generate random reads, though some aspects of the storage layer would have to change.

Flash has two weaknesses that influence the Hyder design. First, pages cannot be destructively updated. A block of 64 pages must be erased before programming (writing) each page once. Erases are slow, ~2ms, and blocks other operations to a large portion of the chip. Second, flash has limited erase durability. MLC (cheap flash) can be erased ~10K times. For SLC (3x the price of MLC), it is ~100K times. Thus, flash needs wear-leveling, to ensure all pages are erased at about the same rate.

These two weaknesses are mitigated by Hyder's use of log-structured (i.e., append-only) storage, which allows the head of the log to be garbage-collected and cleaned while the tail is being written, and which spreads writes evenly across storage

By contrast, SSDs are not append-only; they support update-in-place. To support update-in-place while mitigating the weaknesses of flash, an SSD typically implements a log-structured file system [28]. This allows it to turn random page-writes into page-appends and automatically wear-levels the flash. This requires address mapping logic, garbage collection, and storage headroom. This functionality adds cost and can degrade performance, all in support of an update-in-place operation that Hyder does not need. Although Hyder can work well on SSDs, we believe it will perform better when implemented on raw flash. This is a classic end-to-end argument, where the application uses an append operation and the flash hardware works best as append-only, so there is no value in inserting an update-in-place operation in between.

1.6 Contributions

A short abstract about Hyder appeared in [4]. This paper expands that overview with descriptions of the following contributions:

- A fault-tolerant append-only log that offers an arbitration point between independent transaction servers.
- A log-structured multi-version binary-search-tree index
- An efficient meld algorithm that detects conflicts and merges committed updates into the last-committed state.

- A simulation analysis of the Hyder architecture under a variety of workloads and system configurations.

We describe Hyder's three layers bottom-up in Sections 2-4. Section 5 covers performance, Section 6 discusses related work, and Section 7 is the conclusion.

2. THE LOG

The log is comprised of multiple flash storage units, which we call **segments**. A segment could be a solid-state disk, a flash chip, or some other non-volatile solid-state storage component.

Each log record is a multi-page **stripe**, where each page of a stripe is written to a different segment. The pages of a stripe are written using a RAID-like erasure code, to enable recovery from a corrupted page, failed segment, or lost message.

A **failure zone** is a set of segments that can fail together due to a single-point failure, such as all of the segments in a rack (because they share a power supply and network connection). The best fault-tolerance is obtained when segments storing a stripe are all in different failure zones. A set of segments that store stripes is called a **stripe set**.

To enable a stripe set to be allocated in a dense sequence of segments, segments are addressed round-robin across failure zones. That is, they are assigned **segment IDs** such that if there are *n* failure zones, no two segments in any sequence of *n* segments are in the same failure zone. For example, if *n* = 8, then the first failure zone has segment IDs 0, 8, 16, ..., the second failure zone has segment IDs 1, 9, 17, …, etc. This arrangement ensures that the failure of a failure zone loses at most one page in any stripe.

2.1 Implementing AppendStripe

The log operations are AppendStripe and GetStripe. AppendStripe takes a stripe and stripe id as parameters and returns a **stripe reference**, which tells where the stripe was stored on each segment of the stripe set. AppendStripe is atomic. It is also idempotent, so a caller that fails to receive a reply from an AppendStripe can simply reissue the operation. GetStripe returns the stripe identified by a given stripe reference.

Log operations could be implemented by a server process or storage device that stores all pages of each stripe at the same offset of each segment. This requires the storage device to support an operation to write pages to a specific location, as is offered by SSDs. Given the widespread availability of SSDs, we believe such an implementation is worthwhile, as suggested in [2]. However, as discussed in Section 1.5, we believe that raw flash can offer better performance by mapping log-appends directly into storage-appends. We describe a log design for raw flash in this section.

To implement log operations on stripes, we propose to use a custom controller to access pages on flash storage using the operations AppendPage, GetPage, and EraseSegment. AppendPage appends a given page to a given segment and returns its address. GetPage returns a copy of the page stored at a given address. EraseSegment erases the entire content of the segment. A controller design that implements atomic and idempotent AppendPage and GetPage operations is sketched in [26].

The AppendPage operation is the only synchronization point between servers. If two servers append a page concurrently to the same segment, the segment's controller will serialize the operations and write the pages to successive storage locations. The usual technique of checksums on pages can be used to ensure that AppendPage is atomic.

If a stripe is comprised of *n* pages, then an AppendStripe operation is implemented by invoking *n* AppendPage operations on *n* segments in *n* different failure zones. Since failure zones are physically far apart, each segment must be updated via a different segment controller. Servers execute AppendStripe operations concurrently, which may execute in different orders in different controllers. Thus, in general, the pages of a stripe are not at the same offset of different segments of the stripe set. That is why AppendStripe returns a stripe reference and not a single offset.

If a log record is too large to fit in one stripe, then it needs to be split into multiple stripes. One of the log record's stripes contains the root of the log record. The other stripes can be appended to the log. Large, cold objects contained in the log record can be stored on hard disks, which are cheaper per-GB. All log-record data outside the root stripe must be written before the root stripe, so their addresses can be included in the root stripe. As we will see in Section 3, the log record is structured as a tree. Thus, the root stripe contains an upper portion of the tree that includes the root and points to subtrees that are stored in other stripes.

Each log record is a stripe, and log records must be totally ordered. This order is not self-evident from the order of a stripe's pages, since the order of its pages is different on different segments. To resolve this ambiguity, we define the relative order of stripes by the order of their pages in the first segment of the stripe set. This segment is called the **edge** of the stripe set.

Since a failure may prevent all pages of a stripe to be appended to its stripe set, some bookkeeping is required for servers to agree whether or not all of the pages of a stripe were written. This is done by maintaining a persistent log of stripe references, called the **end-write log**. After the AppendStripe operation receives acknowledgments for all of its corresponding AppendPage operations, it appends to the end-write log an **end-write record** that contains the stripe reference for the appended stripe. For fault tolerance, the end-write record must be appended to multiple segments. The choice of the number of segments should be based on an analysis of flash device failure rates and end-write log recovery time. We expect three will be enough. For multi-stripe log records, a server should wait to receive end-write acknowledgments for all of the non-root stripes before it appends the root stripe, to ensure that the root stripe has no dangling references.

Each log record is broadcast to all servers. First, the stripe is broadcast. After the stripe's end-write has been written, it too is broadcast so that servers know the stripe was successfully written.

The number of physical messages that are broadcast depends on several factors: the size of a log record, the size of a network packet, and the amount of batching of records into packets. In particular, end-writes are small, so many can be stored in a packet, at the cost of some latency to wait until the packet fills.

2.2 Failure Handling

There are three types of failures to consider: message loss, server failure, and segment failure. A short preview of our solution is in [2]. We discuss some basic cases here. For simplicity of exposition, we assume the log record fits in one stripe.

In many cases, lost messages can be retrieved by accessing the log. For example, if a server receives an end-write for a stripe but not all of the pages that the end-write references, then it can retrieve the missing pages from the log. If it receives a partial log record that does not include an edge page, then it can simply ignore the record (unless and until the edge page shows up). If it detects a hole in the sequence of end-writes it has received, it should read the missing page(s) from an end-write segment.

However, what if it receives an edge page E, but does not receive an end-write for a stripe that includes E (within a timeout period)? Edge page E has reserved a slot in the log-record sequence, so later log records cannot be processed until it is known whether E's stripe will show up. Therefore, if it does not receive the rest of E's log record with a timeout period, it declares a log failure and initiates a recovery protocol.

Our recovery protocol is a variation on Vertical Paxos [22]. It seals the edge segment from further appends (e.g., set the segment's class to "sealed," as described below) and runs a consensus protocol to reach agreement on which stripes near the end of the log are complete and should be included. It then creates a new configuration of segments for the log and resumes normal operation. A similar process is used to cope with a permanent segment failure. There are many details to consider, which will be the subject of a future paper.

2.3 Sliding Window Striping

Allowing multiple servers to write stripes independently introduces some storage management problems. First, since flash storage is not cheap per-GB, variable-length stripes should be supported. This implies that initial segments of a stripe set will fill up faster than later ones. When the initial segment fills up, a new configuration must be allocated and all servers must agree to it.

Second, all servers have to agree on which segments are being used for each type of data: end-write records, totally-ordered root log stripes, and unordered non-root log stripes.

To coordinate server agreement, each controller maintains a persistent integer state-variable for each segment it manages, called its **class**. The integer value describes the type of data that can currently be appended to the segment. For concreteness, let us use zero for empty, one for a segment that stores pages of unordered (i.e., non-root) log stripes, two for ordered (i.e., root) log stripes, three for end-writes, and four for "sealed" (i.e., where no more appends are possible).

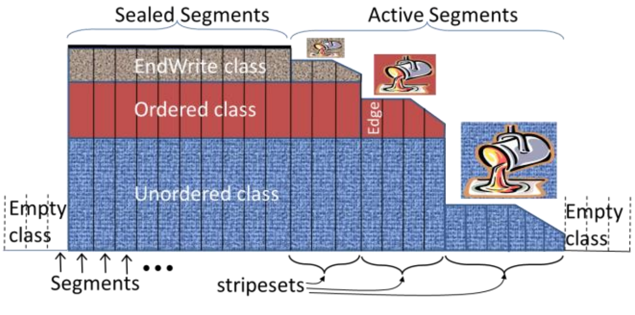


Figure 6 Sliding-window striping

We assign stripe sets to successive segment ranges in decreasing order of class. For example, there might be 3 segments for end-write stripes, then 4 for root stripes, and then 6 for non-root stripes (see Figure 6). Segments before the first end-write segment are sealed and those after the last non-root segment are empty.

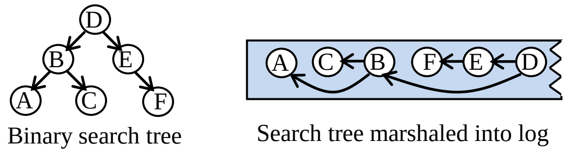
A class variable is added as a parameter to the AppendStripe operation. AppendStripe(C, S) for class C and segment S behaves as follows. If C < class(S), then do not perform the append and return an exception to the caller. If C = class(S) (i.e., the class of S), then perform the append. If C > class(S), then set class(S) to C and perform the append. This mechanism is used to ensure that all servers agree on which type of data to append to each segment.

Suppose the first segment of the end-write class fills up. Then the end-write stripe set would **advance**, thereby "capturing" the first segment (i.e., edge) of the ordered class (see Figure 6). The ordered class is now short by one segment, so it advances, capturing the first segment of the unordered class, and so on. In this sense, the stripe sets behave like sliding windows over the segments.

Higher layers of the system are responsible for garbage collection by copying reachable data in the first sealed segment to the end of storage and ensuring there is an ample supply of empty segments.

3. INDEX STRUCTURE

The index layer of Hyder stores the database as a search tree, where each node is a <key, payload> pair. For concreteness, we use a binary search tree in this paper. The tree is marshaled into the log (see Figure 7). Its basic operations are get, insert, delete and update of nodes and ranges of nodes, identified by their keys. The index layer also maintains a node cache of recently accessed parts of the tree.



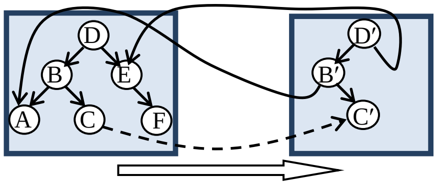
Binary search tree

Search tree marshaled into log

Figure 7 Marshaling a search tree into the log

If the tree stores a relational database, the key would be composite, beginning with the ID of a database, followed (for example) by sub-schemas, tables, and key values. Nodes in upper levels of the tree would span the set of databases. Their descendants would span sub-schemas, then tables, and then rows with key values. Each table can have secondary (i.e., non-clustered) indices, each of which is a table that maps a secondary key to the primary keys of rows that contain that secondary key, as is done in Microsoft SQL Server. Prefix and suffix truncation should be used to conserve space.

A node of the tree cannot be updated in place. To modify a node *n*, a new copy *n′* of *n* is created. Since *n*'s parent *p* needs to be updated with a new pointer to *n′*, a new copy *p′* of *p* is needed. And so on, up the tree. An example is shown in Figure 8. Notice that D′ is the root of a complete tree, which shares the unmodified parts of the tree with its previous version, rooted at D.



To update C's value, create a new version C′ and replace C's ancestors up to the root.

Figure 8 Copy-on-write

Insert and delete operations may trigger tree rebalancing, which updates more nodes than those on the path to the node being inserted or deleted. It may therefore be beneficial to rebalance periodically, rather than on every insert and delete.

An updated tree is logged in a transaction’s intention. For good performance, it is important to minimize its size. For this reason, binary trees are a good choice. A binary tree over a billion keys has depth 30. A similar number of keys can be indexed by a four-layer B-tree with 200-key pages. But an update of that B-tree creates a new version of four pages comprising the root-to-leaf path, which consumes much more space than a 30-node path.

On the other hand, binary trees are known to have poor processor cache performance. In this respect, B-trees perform better and therefore may be preferable for a read-intensive workload, where it is important to optimize processor performance. For update-heavy workloads, we are experimenting with B+-trees that have low fanout and use prefix and suffix key-compression. Initial results suggest it might be able to generate intentions whose size is competitive with binary trees. B-trees also may be preferable in an implementation over hard disks, to reduce the random seeks, especially for sequential access in key-order. A comparative evaluation of these tradeoffs would be a worthwhile investigation. So would an evaluation of the many known optimizations for improving the processor performance of binary trees [19], since those optimizations may be less effective on trees that are fragmented in intentions spread across a log.

To simplify garbage collection of the log, each node pointer includes the segment ID of its earliest reachable segment. This is easy to maintain, as each new node simply picks the minimum of its two children. A segment that is older than any segment pointed to by a node is garbage. A simple tree traversal can identify all nodes in the oldest segment. They can be copied to the end of the log by a copier transaction, thereby freeing up the segment for reuse. Since the copier does not update the data it is copying, a write-conflict with an active transaction does not cause it to abort.

4. MELD

The transaction layer has two components. The executor runs new transactions, as described in the beginning of Section 1.4. The second component is the meld procedure, which is described here.

The meld procedure detects whether an intention experienced a conflict, and if not, merges its updates into the local copy of **the last committed state** (**LCS**). Rather than scanning the intention’s conflict zone to determine whether the transaction experienced a conflict, meld is made more efficient by maintaining metadata in LCS that is sufficient to detect conflicts accurately. Thus, meld takes an intention and its server’s LCS as input, and it returns a new version of LCS. That is, meld is a function—non-destructive with no side effects.

Meld is optimized further by not having to consider every item in the readset and writeset. It stops looking for conflicts as soon as it encounters a subtree of an intention whose corresponding LCS subtree did not change while the transaction executed. The speed-up from this optimization is significant for transactions that operate on non-hot data. In particular, it implies that for multi-stripe log records, meld often only needs to read the root stripe to process the log record’s intention.

Meld must be deterministic. That is, it must produce exactly the same sequence of states on all servers. Otherwise, an intention generated by a transaction on one server might not be properly interpreted by meld running on another server. Hence, meld runs sequentially, processing intentions in log order. Meld can be parallelized to some extent, at least by parsing log records into objects on one thread before interpreting the log records (i.e., the roll forward activity) on another thread.

The update-transaction throughput across all servers is limited by the speed of meld. Therefore, meld must be fast. This is especially important because single-threaded processor performance is not expected to improve much for some time. Our current implementation can meld up to 400K TPS for small transactions.

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

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

For example, suppose transaction T_1 executes on an empty database, inserting nodes B, C, D, and E. (See Figure 9.) Then transactions T_2 and T_3 execute on T_1’s output. T_2 inserts node A, and T_3 inserts F. T_2 and T_3 do not conflict, so after melding them LCS should include both of their updates.

Each node n in an intention I has a unique **version number** (**VN**), which is calculated based on the number of updated descendants in I and the LCS version against which I is melded. It also has a **source content VN** (**SCV**) that refers to the previous version of the node (i.e., the version in *I’s* snapshot), and a flag **DependsOn** that is TRUE if I depends on n not having changed during $T(I)$’s execution. We denote node n in I_j by n_j. We denote its VN, SCV, and DependsOn flag by VN(n_j), SCV(n_j), and DependsOn(n_j). Similarly, VN of node n in the LCS is denoted VN(n_{LCS}).

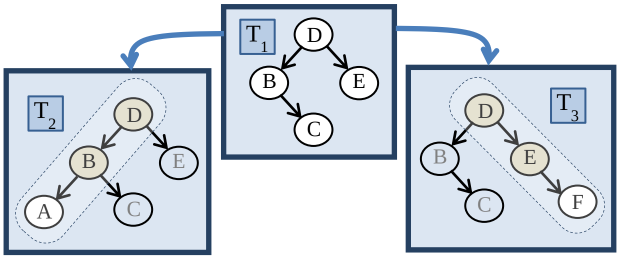
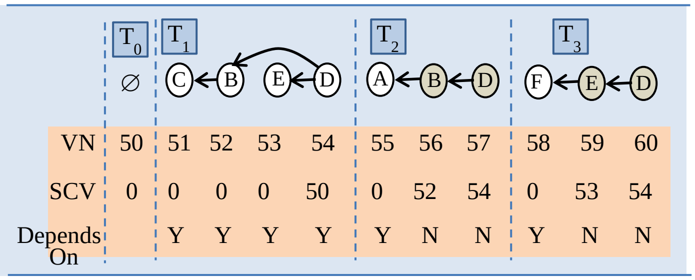


Figure 9 Example transactions to be melded

The need for DependsOn arises in part because some nodes in I were updated only because a descendant was updated. For example, in Figure 8, B′ was updated only because C′ was updated. In this case, DependsOn(C′) = TRUE while DependsOn(B′) = FALSE.

VN, SCV, and DependsOn enable meld to detect conflicts. If SCV(n_I) $\neq$ VN(n_{LCS}), then a committed transaction modified n while $T(I)$ was executing. In this case, if DependsOn(n_I) = TRUE, then $T(I)$ experienced a conflict and should be aborted.

This is an oversimplification, because it might be that SCV(n_I) $\neq$ VN(n_{LCS}) only because one of n’s descendants was updated, not because n’s content changed. If DependsOn(n_I) = TRUE because I depends on the content of n, then such an update does not constitute a conflict, since n’s content in LCS has not changed. On the other hand, if DependsOn(n_I) = TRUE because $T(I)$ read the entire key range rooted at n and $T(I)$ uses serializable isolation, then SCV(n_I) $\neq$ VN(n_{LCS}) does imply a conflict since the content of some record in the subtree rooted at *n* changed. By enriching the definition of SCV(*n*) and adding other metadata to *n*, we can distinguish these cases, but for pedagogical simplicity we will not make this distinction here. Details are in [5].



	T_0	T_1				T_2			T_3		
	∅	C	B	E	D	A	B	D	F	E	D
VN	50	51	52	53	54	55	56	57	58	59	60
SCV	0	0	0	0	50	0	52	54	0	53	54
Depends On		Y	Y	Y	Y	Y	N	N	Y	N	N

Figure 10 Log of the transactions of Figure 6

Suppose transactions T_1-T_3 (from Figure 9) are sequenced in the log as shown in Figure 10**.** Meld processes the log as follows:

1. Meld deduces that T_1 is serial, because SCV(D_1) = 50 = VN(D_{LCS}), which means T_0 immediately precedes T_1. So it melds T_1 by returning a new LCS whose root is D_1, where VN(D_1) = 54. Notice that this step required examining D_1 but none of its descendants.
2. Similarly, meld deduces that T_2 is serial and returns a new LCS whose root is D_2, where VN(D_2) = 57.
3. For T_3, meld sees that SCV(D_3) ≠ VN(D_{LCS}) (i.e., 54 ≠ 57). Since DependsOn(D_3) = FALSE, these unequal VN's do not indicate a conflict. However, a descendant of D_3 might depend on a node in LCS that was updated in T_3's conflict zone. So meld has to drill deeper into I_3 by visiting E_3.
4. Meld sees that SCV(E_3) = VN(E_{LCS}) = 53. Thus, the subtree rooted at E did not change while T_3 was executing. So there is no conflict and meld can declare T_3 as committed.

In (1), (2), and (4) above, meld is able to truncate the traversal of *I*. This happens whenever meld encounters a subtree of the intention that did not change while its transaction was executing. This is a very important optimization, which significantly reduces the time to meld the intention compared to a naïve algorithm that tests every node in the intention for a conflict. For example, consider the performance benefit if nodes A and F were leaves of very long paths and meld were truncated high in the tree, like it was here.

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

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

An ephemeral intention's nodes, called **ephemeral nodes**, must have unique VN's, since they are needed by meld to process the next intention that follows it. To do this, meld dynamically assigns VNs to nodes when it processes each intention (as it does for nodes physically in the intention). Like all of meld's activities, the generation of ephemeral intentions, ephemeral nodes, and their VN's is done deterministically, so that all servers produce identical states against which new transactions can execute.

There are many other details of meld that are needed for more precise detection of conflicts (i.e., with no false positives), coverage of all isolation levels including proper handling of phantoms, optimized per-node and per-intention metadata to minimize the size of intention records, flushing of ephemeral nodes, checkpointing, server initialization and fast recovery, and garbage collection. These topics are covered in [5].

5. PERFORMANCE

We cannot report yet on the performance of an end-to-end prototype, because our implementation of the transaction and indexed record layers is not yet integrated with a flash-based implementation of the log. However, through a combination of measurements of implemented components, known hardware speeds, mathematical analysis, and fairly extensive simulations, we can predict how a complete implementation would perform.

5.1 Bottleneck Analysis

As mentioned in Section 1.4, Hyder has four potential bottlenecks: appending intentions to the log, broadcasting intentions to all servers, melding the log at each server, and aborting transactions due to conflicts. We discuss each one in turn.

Logging – Although the log could be stored on solid-state disks, raw flash chips with a custom controller are preferred. Custom hardware is quite feasible, given today's rapid innovation in flash board products and use of custom hardware in data centers and database appliances. One benefit of custom hardware is speed. For example, putting nonvolatile write buffers in front of 20 flash chips that are served round-robin would reduce the 200 μs write latency of raw flash to 10 μs. Since log-appends are a bottleneck, this speedup over SSDs can be very important. If each log stripe contains one intention, then 10 μs write latency implies a limit of 100K TPS. Batching multiple intentions per log stripe can improve throughput further.

Networking – Switched networks process point-to-point messages in parallel. Therefore, with large server buffer caches, reads will not significantly reduce throughput for Hyder. By contrast, broadcasts block the network—a major bottleneck. The switching-time of current network switches is under 1 μs for 10 Gb Ethernet. Interconnect, protocol, and distance delays add to that. On the other hand, technology improvements continue to reduce latency. Switching time on 40 Gbps Ethernet is already under 400 ns.

Meld – Our current meld implementation can meld up to 400K TPS for transactions with two operations, dropping to 130K TPS for transactions with eight operations. Although single-threaded processor performance is not expected to improve much, meld can be parallelized further, which should yield a several-fold speedup. We report on extensive meld performance experiments in [5].

Optimistic Concurrency Control – Like any concurrency control algorithm, the abort rate of Hyder's optimistic concurrency control depends on the fraction of concurrently-executing transactions that conflict [20]. In turn, this depends on the probability that two randomly-selected transactions conflict, and the average number of transactions that execute concurrently at any given time. For a given arrival rate of new transactions, the faster they execute, the fewer that execute concurrently, and hence the smaller their conflict zones and the lower their abort rate.

The worst case consists of concurrent transactions that read and write the same data. In that case, a serializable execution must be serial, so the maximum throughput is the inverse of execution time. For example, with 200 µs transaction latency as observed in our prototype, the maximum throughput on a single write-hot data item is 5K TPS. Assuming Poisson arrivals and exponential service times, a naïve analysis predicts throughput of ~1.6K TPS per write-hot data item. Clearly, it is beneficial to detect such high-conflict transactions and run them on one server, which serializes and batches the transactions to obtain higher throughput.

5.2 Simulation Analysis

We present a detailed analysis of Hyder's performance at high loads using different workloads, access patterns, and isolation levels. The first part of our analysis focuses on system performance when there are no resource bottlenecks. Later, we evaluate and analyze Hyder's behavior in the presence of high data contention and resource contention.

We developed a simulation for Hyder using a discrete event simulator. The simulation model has three main modules: compute nodes, a network, and a log, representing the components in the system as shown in Figure 1. The compute nodes execute transactions, maintain a cache of recently accessed database objects using a least-recently-used replacement policy, and run the meld procedure. Each node has limited processing capacity; we use a four-core processor.

We model the database as a set of integer keys. The combined size of the key and payload is set to 100 bytes.

Each transaction is comprised of a fixed number of reads and writes controlled by the read/write ratio. Keys accessed by a transaction depend on the access distribution: We use **hotspot** (x% operations accessing y% data) and **uniform access** distributions.

A transaction consumes 2 µs of latency for each read that is serviced from cache and 10 µs of latency for each write. A cache miss results in reading the missing intention from the log. Each update transaction uses two broadcasts to all servers: one to broadcast the intention and one for the log to broadcast the intention's offset.

We model the network as a switch that simulates network delays. The network allows concurrent unicast messages to different destinations while a broadcast ties up the entire network. We use a switching delay of 400 ns for a 1500 byte frame resulting in a broadcast throughput of 40 Gbps, assuming 80% utilization. A much higher unicast throughput is supported. If we used a 10 Gbps network instead, it would be the bottleneck. If 80% utilized, it would reduce throughput by 1/3, compared to 40Gbp at 30%.

The log simulates appends to the flash by a 10 µs delay. Each intention is appended within a single append latency. That is, we assume an intention fits in one log stripe.

The meld process performs explicit conflict detection by checking a new transaction for conflicts against the set of committed transactions in its conflict zone. We use serializable and snapshot isolation. Meld processing is simulated by a latency of 10 µs as measured in our prototype implementation of meld.

Under the configurations used, the peak capacity of the system is 100K TPS. We therefore use a peak offered load of 80K TPS to ensure the resources are appropriately exercised without overloading them. We set the database size to 1 million elements, the transaction size to 10 operations (8 reads and 2 writes), and the cache size to 300K elements, with serializable as the default isolation level.

We compute the intention record sizes as a function of the depth of the database tree and the number of operations in the transactions that must be included in the intention; serializable isolation requires both the readsets and writesets, while snapshot isolation requires only the writeset. If an intention does not fit into a network frame, we use multi-frame messages and assume that the network guarantees their in-order delivery.

We use the following convention for series names in the graphs: **Hot-x-y** refers to hotspot access patterns with x% operations accessing y% data items and **Uniform** represents uniform access patterns; **SI** and **SR** represent snapshot and serializable isolation.

5.2.1 Abundant Resources

Effect of Skew

To analyze the effect of skew in access patterns, we use hotspot and uniform access distributions. For hotspot distributions, we vary x from 80% to 95% and we vary y from 5% to 20%. In all experiments, the throughput increases linearly with the offered load and is almost equal to the offered load. Figure 11 plots throughput as a function of the offered load and access distributions using serializable isolation. The linear increase in throughput with offered load is evident from the linear fit curve on the throughput values. This linear increase is observed for all access distributions and isolation levels.

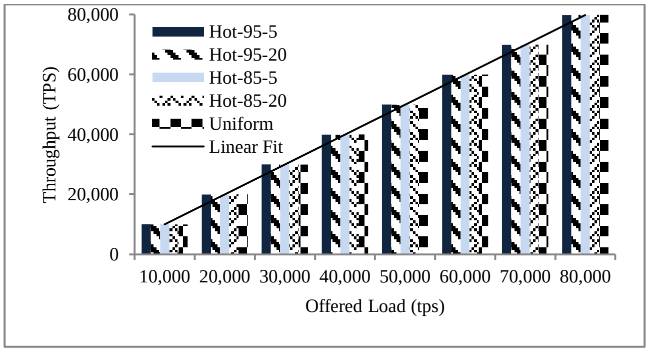


Figure 11 Transaction throughput (in TPS) as a function of the offered load and access distributions.

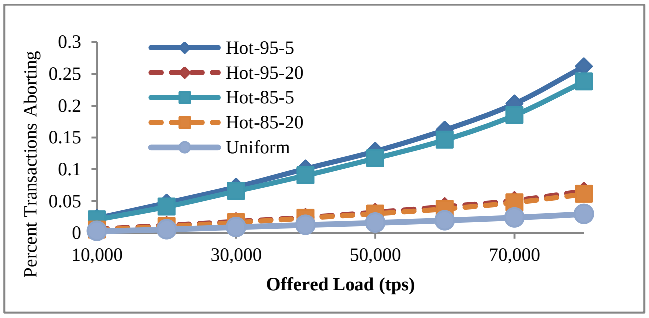


Figure 12 Percent transactions aborting as a function of offered load and access distributions.

Figure 12 plots the abort rate. It shows that although the abort rate increases with an increase in load, it is still negligibly small (in the range of 0.25%) at the peak offered load of 80K TPS. Though not shown, snapshot isolation has even lower abort rates, as expected. Furthermore, as long as the skew is not very high, the abort rate does not increase significantly with an increase in offered load. This shows that given abundant network and flash capacity, the Hyder architecture can process a high volume of update transactions, and the throughput increases linearly with the offered load and is almost equal to the offered load.

We observed an average cache miss penalty of ~20 μs and an average transaction latency of ~50-60 μs for hotspot distributions and ~180 μs for uniform distributions. Due to the small cache miss rates for hotspot access distributions, the network throughput is dominated by broadcasts of intentions; peak network utilization for serializable isolation was ~10-12 Gbps, of which more than 90% was due to broadcasts. For uniform access distributions, unicast traffic is much higher as a result of cache misses; peak network utilization for serializable isolation was on the order of 65 Gbps, of which only 15% were broadcasts. Similarly, for hotspot access, less than 1% of the requests to the log are reads, while for uniform, this load rises to 80%.

The intention record size measured for serializable isolation was ~15.7 KB and for snapshot isolation was ~3.6 KB. The smaller intention size with snapshot isolation results in lower network load; for SI, network utilization peaks at about 3 Gbps and 15.5 Gbps for skewed and uniform distributions respectively with a share of broadcast and unicast traffic similar to that with serializable isolation.

Even at an offered load of 80K TPS, the mean conflict zone length was 7-10 intentions. This small conflict zone length is because transaction latencies were in the range of hundreds of microseconds. The conflict zone length also depends on the whether requests arrive in bursts or are spread out.

Effect of Cache Size

We now analyze the impact of cache hit rates on Hyder's performance. We vary the cache size from 50K to 300K items. At 50K items, the entire hot set does not fit in the cache (for the access patterns used in the experiments) resulting in a low cache hit rate, while at 300K elements, the hot set fits in the cache, resulting in a higher cache rate. A larger cache results in higher cache hit rates which reduce transaction latency, which in turn results in smaller conflict zones and lower abort rates.

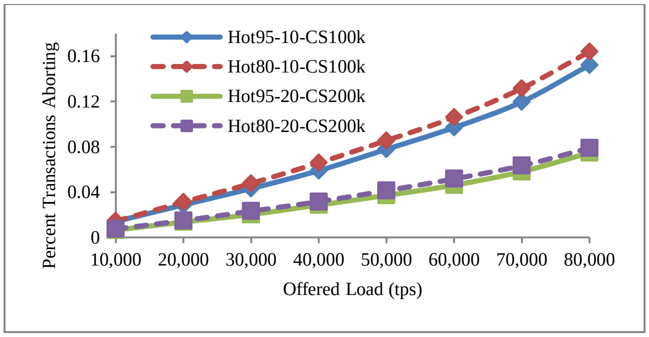


Figure 13 Percent transactions aborting as a function of offered load for different access distributions and cache sizes.

Figure 13 shows an interesting interplay between cache size and data skew and its impact on the abort rate. In each experiment, the cache size is set to the size of the hot set. For instance, a cache of 100K items is used for an access distribution where the hot set size is 10% of the database. The naming convention for the series is: "Access Distribution"-"Cache size"; thus, Hot95-10-CS100k refers to Hotspot distribution with 95% of operations accessing 10% of the data items, and with a cache size of 100K elements.

When the size of the hot set is kept constant and the percentage of operations accessing the cold set is increased, an increase in the abort rate is observed. Referring to Figure 13, the abort rate for the 80-10 access distribution is higher than that of the 95-10 distribution. This increased abort rate is counter-intuitive since a decrease in skew of the data items accessed should reduce the probability of a conflict. A closer analysis reveals that the 80-10 distribution has more accesses to the cold set. This increases the number of cache misses, which increases transaction latency, and in turn increases the conflict zone length and hence the conflict probability. This increase in transaction latency and conflict zone length is evident from Figure 14 which plots transaction latency as the primary vertical axis (on the left) and conflict zone length as the secondary vertical axis. This behavior is observed until the skew reduces to a point where the effect of higher conflict rate swamps the effect of shorter conflict zone. However, this counterintuitive behavior is not observed when the cache size is big enough to easily fit the hot set, as in Figure 12; the reason is that the larger cache is able to accommodate a small portion of the cold set and the entire hot set, which reduces the impact of skew on the cache hit rate and hence on the conflict zone length.

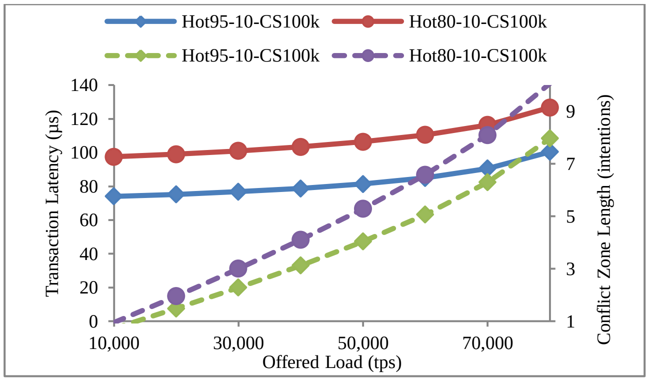


Figure 14 Transaction latency as a function of offered load for different access distributions and cache sizes.

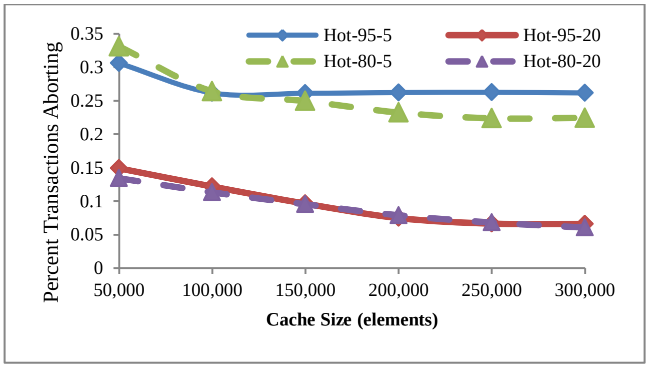


Figure 15 Percent transactions aborting as a function of cache size for different access distributions.

Figure 15 plots the abort rate as a function of cache size for different access distributions using serializable isolation; the offered load is kept constant at 80K TPS. As expected, as cache size increases the abort rate decreases, since more requests are served from the cache which results in shorter transaction latencies and conflict zones. When the cache size is 50K elements, it equals the hot set size for the 95-5 and 80-5 access distributions. We therefore observe behavior similar to Figure 13 where lower skew results in a higher abort rate.

For the access distributions 95-5 and 80-5, as the cache size grows, the entire hot set fits into the cache. Now the reduction in transaction latency resulting from higher skew is insignificant and the impact of higher skew on conflict probability dominates. As a result, when the cache is larger than the hot set, a higher abort rate is observed for a more skewed distribution; in Figure 15, the 95-5 distribution has a higher abort rate than the 80-5 distribution.

When the cache is too small to fit the hot set, a higher skew in access distribution does not result in a higher cache hit rate, since elements in the hot set also incur cache misses. As a result, the transaction latency, and hence the conflict zone length, for more skewed distributions are approximately equal to those of less skewed distributions. Therefore, the effect of skew dominates, resulting in a higher abort rate for skewed distributions. This behavior is evident from the experiments with 95-20 and 80-20 distributions where the hot set has 200K data items.

When the cache size is under 200K elements, the more skewed distribution (95-20) has a higher abort rate. When the cache size equals 200K elements, as expected, the 80-20 distribution experiences a marginally higher abort rate. Beyond 200K elements, the abort rate of 80-20 is again lower compared to that of 95-20. The cache size also impacts network utilization since cache misses result in unicast network messages to read the specified intention record from the log.

Effect of other parameters

For a given access distribution, increasing the number of reads reduces the abort rate. Similarly, increasing the transaction size increases the abort rate. Abort rates as high as 3.5% are observed for a skewed access distribution for transactions with 25 operations where the hot set comprises only 5% of the database. However, the abort rate decreases considerably for distributions with less skew, accompanied by a smaller gradient of increase.

We also experimented with a workload of variable-size transactions. As expected, increasing the average transaction size increases the abort rate. However, it is interesting that with variable-sized transactions whose average size is S, the abort rate is lower than with fixed-size transactions of size S. A closer analysis reveals that variable-size transactions broadcast fewer messages on average than fixed-size transactions, which reduces transaction latency, conflict zones, and hence conflict probability.

5.2.2 Data Contention

In the previous experiments, data contention was low and had an insignificant effect on throughput, even at the peak offered load of 80K TPS and with 95% of the operations accessing 5% of the database. Now, we focus on the performance of workloads with high data contention. We use smaller databases (10K to 100K), higher skew (99% of operations accessing 1% of the data items), more writes per transaction (1:1 read/write ratio), and larger transaction size. Figure 16 plots the throughput as a function of offered load. We use serializable isolation for all the experiments.

The first three series correspond to database sizes of 10K, 50K, and 100K and use transactions with 8 reads and 2 writes. The database size is kept constant of 100K for the remaining two experiments. The fourth series corresponds to transactions with 5 reads and 5 writes (1:1 read/write ratio), while the fifth series uses transactions with 16 reads and 4 writes.

We observe a significant impact of abort rate on throughput; the throughput curve plateaus out only for a database size of 10k or transaction size of 20 where abort rates are as high as 50-60%. As expected, the impact is much less for snapshot isolation, where no significant effect on throughput was observed (not shown).

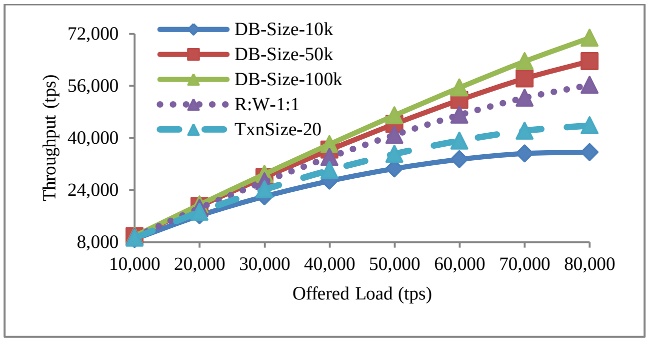


Figure 16 Transaction throughput (in tps) as a function of offered load for Serializable Isolation. Triangular markers correspond to Database size of 100k elements with different read/write ratios and transaction sizes.

5.2.3 Resource Contention

Pure data contention resulting from conflicting accesses does not significantly affect performance. However, heavy resource contention causes thrashing. For example, if the transaction execution rate reaches the maximum rate of log appends, throughput drops sharply (see Figure 17). Overloading the network or meld algorithm results in similar behavior. During an overload, transactions execute longer (latency of ~100 ms vs. ~200 µs under normal load), which increases the abort rate. Aborted transactions consume resources, which increases resource contention, resulting in a negative feedback loop. The effect is stronger for transactions with skewed access, because of their higher abort rate for a given conflict zone length.

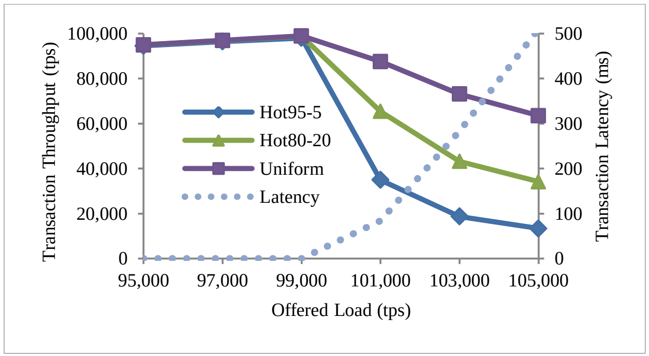


Figure 17 Thrashing resulting from resource contention.

Of course, Hyder should back off the workload when it detects thrashing. Moreover, it should do a trial meld before appending a transaction to the log. If it detects a conflict, it will not waste a broadcast slot and log append, thereby defeating the negative feedback loop, making the throughput drop-off much less steep.

5.3 Future Performance Work

Our simulation analysis shows that for practical workloads, such as highly-skewed hotspot access patterns and a database of 1M items, data contention induced by resource contention occurs long before "pure" data contention due to conflicts. Therefore, it is essential to use load-control to prevent resource contention. Moreover, to deal with very high conflict rates (as in Section 5.2.2), techniques for advanced transaction scheduling cognizant of conflict patterns will be useful. For example, a soft partitioning of transactions that access write-hot data could enable them to be synchronized in the server before appending their intentions to the log. In turn, this will complicate load balancing, since the partitioning will constrain where such transactions should run.

The critical resources in Hyder—the network, log, and meld—have hard scaling limits imposed by the underlying hardware, which limits Hyder's scalability and peak capacity. Parallelism can be explored to increase the peak capacity of these critical resources. One such extension is to partition the database to enable parallel logging and melding. These extensions are worthwhile areas for future work.

6. RELATED WORK

Hyder is a data-sharing system. Other well-known data-sharing systems are Oracle RAC [9], Oracle Rdb [23] and IBM DB2 [18]. These systems use a shared lock manager to ensure that at most one server at a time has access to a page. If a server releases its write lock on a page, it must make the updated page available to other servers either by writing the page to disk or servicing reads from other servers. Thus, the granularity of data sharing between servers is a page, rather than an indexed record in Hyder. A failure that affects the lock table requires server coordination to recover, which is not needed in Hyder since servers are independent. On the other hand, the failure of a log segment in Hyder requires coordinated recovery, which is not required in systems that use server-local logs. These locking-based systems are known to require soft partitioning of the transaction load to avoid having pages ping-pong between servers. While Hyder can benefit from such partitioning, it performs well without it. The performance of these systems was studied in [25]. A comparison of their behavior to Hyder is suggested as future work.

There are many shared-nothing database systems designed for scale-out. The benefits of this architecture were demonstrated in the 1980s in many systems, such as Bubba, Gamma, Tandem, and Teradata. DeWitt and Gray [16] provide an excellent summary of that work. More recently, shared-nothing key-value stores have been developed for cloud computing with similar goals, such as Google's Bigtable [8], Amazon's Dynamo [15], Facebook's Cassandra [21], and Yahoo!'s PNUTS [10]. However, unlike Hyder, these systems do not support ACID transactions and they require the database to be partitioned. Microsoft's SQL Azure [3][6] and ElasTraS [14] are cloud-oriented database systems that do support ACID transactions but restrict each one to access one partition. The partitioned access requirement is relaxed somewhat in G-Store [13], since data access groups can be reformed dynamically.

A transactional, partitioned, key-value store that does not restrict a transaction's access pattern is described in [1]. Like Hyder, it uses optimistic concurrency control. Unlike Hyder, it is a shared-nothing system and hence requires two-phase commit. During its execution, a transaction tracks the versions of data it reads and writes. At commit time it sends this information to each server it accessed; the server either validates its readset and applies its writes as part of phase one, or detects a readset item changed and tells the transaction to abort. Among the key-value stores mentioned in the previous paragraph, its functionality is most similar to Hyder's, but its mechanisms are quite different. It would be interesting to compare their performance.

Another transactional, partitioned key-value store that uses optimistic concurrency control to synchronize transactions with two-phase commit for atomicity is ecStore [31].

Hyder strongly resembles a primary-copy replicated database, in the sense that it generates a single log and sends it to many servers, each of which applies updates to its local copy (i.e., a replica). However, Hyder is different from primary-copy replication in two ways. First, Hyder does not use a primary copy! Second, log records include the updates of both committed and aborted transactions. In most primary-copy replication systems, only committed transactions are sent to replicas. We do not know of any replication systems with the latter two properties. Gupta et al. [17] propose a replication mechanism similar to Hyder, but different in that it uses a single server as the serialization point and supports weaker forms of consistency that are sufficient for their application domain. A survey of recent work on primary-copy replication can be found in [7].

Multi-master replication, sometimes called optimistic replication, is similar to Hyder in that conflicting transactions can run to completion and their conflicts are detected later. Unlike Hyder, these mechanisms do not support standard isolation levels. Surveys of this work appear in [29] and [30].

The startup company RethinkDB has built a log-structured storage engine for MySQL, targeted for use on solid-state storage [27]. The database is append-only and "lock-free." However, they have not yet published details about their index structure or concurrency control algorithm. They support primary-copy replication, but are apparently not a data-sharing system.

Dan et al. [11] present an analytical model and simulation study of the effect of data and resource contention on transaction throughput for optimistic concurrency control (OCC) [20]. We observed similar behavior in our simulations: data contention alone causes system throughput to plateau out and resource contention causes thrashing. Transaction latency during high resource contention in Hyder also has behavior similar to that reported in [11].

The interplay of skew in data access distribution, cache hit ratio, and abort probability for lock-based concurrency control was studied by Dan et al. [12]. Our experimental study extends these results by demonstrating a similar interplay when using OCC. We further analyze the interplay of cache size with working set size for a hotspot distribution. Yu et al. [32] model the impact of improved cache hit rate for transactions restarted after an abort and study its effect on system throughput. If an aborted transaction restarts at the same compute node in Hyder, we expect to observe a similar behavior; however, we did not observe a significant impact since most of our simulations observed very low abort rates.

7. CONCLUSION

We have described the architecture and major algorithms of Hyder, a transactional record manager that scales out without partitioning. It uses a novel architecture: a log-structured multiversion database, stored in flash memory, and shared by many multi-core servers over a data center network. We demonstrated the feasibility of the architecture by describing two new mechanisms, a shared striped log and a meld algorithm for performing optimistic concurrency control and applying committed updates to each server's cached partial copy of the database. We presented performance measurements and simulations that demonstrate linear scalability up to the limits of the underlying hardware.

Many variations of the Hyder architecture and algorithms would be worth exploring. There may also be opportunities to use Hyder's logging and meld algorithms with some modification in other contexts, such as file systems and middleware. We suggested a number of directions for future work throughout the paper. No doubt there are many other directions as well.

8. ACKNOWLEDGMENTS

This work has benefited from challenging discussions with many engineers at Microsoft over the past three years, too numerous to list here. We thank José Blakeley, Dave Campbell, Quentin Clark, Bill Gates, and Mike Zwilling for their early support of the project. We also thank our collaborators on the meld implementation, Ming Wu and Xinhao Yuan, and on the log implementation, Mahesh Balakrishnan, Dahlia Malkhi, and Vijayan Prabhakaran. We also thank Gustavo Alonso, Goetz Graefe, Dahlia Malkhi, Sergey Melnik, and Ming Wu for their reviews of early drafts of this paper and the anonymous referees for many excellent suggestions.

9. REFERENCES

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

[2] Balakrishnan, M., P. Bernstein, D. Malkhi, V. Prabhakaran, and C. Reid: Brief Announcement: Flash-Log, A High Throughput Log. DISC 2010, LNCS 6343: 401-403

[3] Bernstein, P.A., I. Cseri, N, Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, T. Talius: Adapting Microsoft SQL Server for Cloud Computing. ICDE 2011, to appear.

[4] Bernstein, P.A. and C.W. Reid: Scaling Out Without Partitioning. HPTS 2009, 3 pp.

[5] Bernstein, P.A., C.W. Reid, M. Wu, X. Yuan: Optimistic Concurrency Control by Melding Trees. Submitted for publication.

[6] Campbell, D.G., G. Kakivaya, N. Ellis: Extreme scale with full SQL language support in Microsoft SQL Azure. SIGMOD 2010: 1021-1024.

[7] Cecchet, E., G. Candea, A. Ailamaki: Middleware-based database replication: the gaps between theory and practice. SIGMOD 2008: 739-752

[8] Chang, F., J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber: Bigtable: A Distributed Storage System for Structured Data. ACM TOCS 26(2): (2008).

[9] Chandrasekaran, S. and R. Bamford: Shared Cache - The Future of Parallel Databases. ICDE 2003: 840-850.

[10] Cooper, B.F., R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, R. Yerneni.: PNUTS: Yahoo!'s hosted data serving platform. PVLDB 1(2): 1277-1288 (2008).

[11] Dan, A., D.F. Towsley, W.H. Kohler: Modeling the Effects of Data and Resource Contention on the Performance of Optimistic Concurrency Control. ICDE 1988: 418-425.

[12] Dan, A., D.M. Dias, P.S. Yu: The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment. VLDB 1990: 419-431.

[13] Das, S., D. Agrawal, A. El Abbadi: G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud. SOCC 2010: 163-174.

[14] Das, S., S. Agarwal, D. Agrawal, A. El Abbadi: ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. UCSB CS Tech Report 2010-04.

[15] DeCandia, G, D. Hastorun, M. Jampani, G. Kakulapati , A. Lakshman , A. Pilchin , S. Sivasubramanian , P. Vosshall , W. Vogels: Dynamo: Amazon's Highly Available Key-value Stor. Proc. 21st SOSP: 205-220 (2007).

[16] DeWitt, D.J and J. Gray: Parallel Database Systems: The Future of High Performance Database Systems. *CACM* 35(6): 85-98 (1992)

[17] Gupta, N., A.J. Demers, J. Gehrke, P. Unterbrunner, W.M. White: Scalability for Virtual Worlds. ICDE 2009:1311-1314

[18] Josten, J.W., C. Mohan, I. Narang, and J.Z. Teng. “DB2's Use of the Coupling Facility for Data Sharing.” IBM Systems Journal 36(2): 327-351 (1997).

[19] Kim, C., J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, P. Dubey: FAST: fast architecture sensitive tree search on modern CPUs and GPUs. SIGMOD 2010: 339-350

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

[21] Lakshman, A. and P. Malik: Cassandra: a decentralized structured storage system. Operating Systems Review 44(2): 35-40 (2010)

[22] Lamport, L., D. Malkhi, L. Zhou: Vertical paxos and primary-backup replication. PODC 2009: 312-313.

[23] Lomet, D.B., R. Anderson, T.K. Rengarajan, and P. Spiro: How the Rdb/VMS Data Sharing System Became Fast. http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-92-4.html, Technical Report CRL 92/4 (May 1992).

[24] MySQL 5.5 Reference Manual, Chapter 13, http://dev.mysql.com/doc/refman/5.5/en/index.html

[25] Rahm, E., Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems. ACM TODS 18(2): 333-377 (1993)

[26] Reid, C.W. and P.A. Bernstein: Implementing an Append-Only Interface for Semiconductor Storage. IEEE Data Eng. Bulletin 33 (4) (Dec. 2010).

[27] RethinkDB, http://www.rethinkdb.com/.

[28] Rosenblum, M. and J.K. Ousterhout: The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. 10(1): 26-52 (1992).

[29] Saito, Y. and M. Shapiro. Optimistic replication. ACM Computing Surveys 37(1): 42-81 (2005).

[30] Terry, D.B.: Replicated Data Management for Mobile Computing. Morgan & Claypool, 2008.

[31] Vo, H. T., C. Chen, B. C. Ooi: Towards Elastic Transactional Cloud Storage with Range Query Support. PVLDB 3(1) : 506-517 (Sept. 2010).

[32] Yu, P.S. and D.M. Dias: Impact of large memory on the performance of optimistic concurrency control schemes. Int’l Conf on Databases, Parallel Architectures and their applications. PARBASE-90: 86-90.