Warp: Lightweight Multi-Key Transactions for Key-Value Stores

Robert Escriva[†], Bernard Wong[‡], Emin Gün Sirer[†]
[†] *Computer Science Department, Cornell University*
[‡] *Cheriton School of Computer Science, University of Waterloo*

Abstract

Traditional NoSQL systems scale by sharding data across multiple servers and by performing each operation on a small number of servers. Because transactions necessarily require coordination across multiple servers, NoSQL systems often explicitly avoid making transactional guarantees in order to avoid such coordination. Past work in this space has relied either on heavyweight protocols, such as two-phase commit or Paxos, or clock synchronization to perform this coordination.

This paper presents a novel protocol for providing ACID transactions on top of a sharded data store. Called linear transactions, this protocol allows transactions to execute in natural arrival order unless doing so would violate serializability. We have fully implemented linear transactions in a commercially available data store. Experiments show that Warp achieves 3.2× higher throughput than Sinfonia's mini-transactions on the standard TPC-C benchmark with no aborts. Further, the system achieves 96% the throughput of HyperDex even though HyperDex makes no transactional guarantees.

1 Introduction

NoSQL systems have become the de facto back-end for modern Big Data applications because they allow unprecedented performance at large scale. The defining characteristic of these systems is their distributed architecture, where the system shards data across multiple servers to improve scalability. To further improve scalability, these systems typically avoid cross-server communication which makes it difficult to implement ACID transactions.

Distributed transactions demand coordination among multiple servers. In traditional RDBMSs, transaction managers coordinate clients with servers, and ensure that all participants in multi-phase commit protocols run in lock-step. Such transaction managers constitute bottlenecks, and modern NoSQL systems have eschewed them for more distributed implementations. Scatter [19] and Google's Megastore [5] shard the data across different Paxos groups based on their key, thereby gaining scalability, but incur higher coordination costs for actions that span multiple groups. An alternative approach, pursued in Calvin [43], is to serialize all operations using a consensus protocol and use deterministic execution to improve performance. Google's Spanner [12] relies on the TrueTime API to assign timestamps to transactions without cross-server synchronization. Compared to traditional NoSQL systems with simple and scalable designs, these systems introduce *spurious coordination* between transactions. Spurious coordination is when a transaction processing protocol unnecessarily delays or reorders transactions' execution in order to enforce an order between transactions that could be applied in natural arrival order. Coarse-grained consensus groups and centralized sequencers both exhibit varying degrees of spurious coordination.

This paper introduces Warp, a NoSQL system that provides support for efficient, one-copy serializable ACID transactions with no spurious coordination. Warp uses a novel server-side commit protocol called *linear transactions* which executes transactions in natural arrival order directly on the servers which hold relevant data, unless doing so would violate serializability. The linear transactions protocol uses a novel dependency tracking technique to enable servers to locally decide when a transaction may commit. Servers are free to commit transactions in their natural arrival order, except in instances where an arriving transaction contains dependencies on other, outstanding transactions.

Three techniques, working in concert, enable linear transactions to simultaneously achieve scalability and performance, without additional transaction managers or clock synchrony assumptions. First, linear-transactions improve scalability by arranging servers into per-transaction dynamically-determined chains, where each chain contains, solely, those servers which store data affected by the transaction. Similar to how traditional NoSQL systems store and retrieve a single object using just $O(1)$ servers, Warp processes a transaction over k objects using the minimal set of $O(k)$ servers

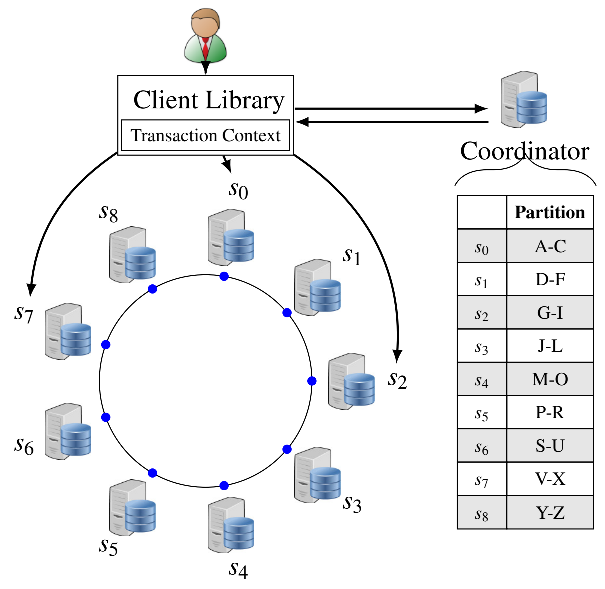
Second, linear transactions eliminate spurious coordination by ordering only those transactions which have data items in common and their transitively-ordered transactions. Approaches to transaction management which compute a total order on *all* transactions necessarily require costly global coordination. Such a total order leads to spurious coordination, and thus, inefficiency, which some systems reduce by partitioning the consensus groups into smaller units [5, 19]. Linear transactions avoid these overheads by avoiding a total order; the relative order between any two transactions is only explicitly decided when the two transactions' have data items in common.

Finally, linear transactions improve performance by allowing multiple writes to the same object proceed in parallel. Locking-based isolation approaches would necessarily serialize transactions as writers wait to obtain contended locks. Traditional optimistic two-phase commit protocols will abort transactions that try to concurrently prepare, leading to high abort rates when writing popular objects. Linear transactions allow the writes to proceed in parallel without serializing the execution or introducing spurious aborts.

Intuitively, a system which guarantees serializability requires some form of consensus or additional synchronicity assumptions to provide such guarantees, and Warp is no exception. Our approach relies on a replicated state machine called the coordinator to establish the membership of the servers in the cluster, as well as the mapping of key ranges to servers. A crucial distinction from past work that perform consensus agreement on the data path, however, is that linear transactions involve its consensus component only in response to failures.

Overall, this paper makes three contributions. First, we outline a novel protocol for providing efficient, one-copy serializable transactions on a distributed, sharded data store. The protocol eliminates spurious coordination, tolerates a user-specified threshold of faults, guarantees atomicity and provides isolation. Second, we describe our implementation of the commercially available Warp key-value store, including the design of the client. The system has been fully implemented, supports C/C++, Python, Java, Ruby, and Node.JS bindings. Third, we show through macro- and microbenchmarks that Warp achieves throughput that Warp achieves 3.2× higher throughput than mini-transactions on the standard TPC-C benchmark with no aborts, and has low overhead; the system achieves 96% the throughput of HyperDex, upon which Warp builds, even though HyperDex provides no transactional guarantees.

The rest of this paper is organized as follows. Section 2 describes linear transactions protocol for NoSQL systems. Section 3 describes our full implementation of Warp. Section 4 evaluates the performance of Warp. Section 5 surveys existing systems and provides context and Section 6 concludes.



	Partition
s_0	A-C
s_1	D-F
s_2	G-I
s_3	J-L
s_4	M-O
s_5	P-R
s_6	S-U
s_7	V-X
s_8	Y-Z

Figure 1: Warp's architecture consists of storage servers, the coordinator, and the client library. The coordinator maintains the partitioning of the key space across servers, and supplies this mapping to the client library. The client library uses this mapping to directly contact storage servers.

2 Design

Warp builds upon the HyperDex [17] key-value store, and is comprised of three components: storage servers, clients, and the coordinator. Each storage server maintains a subset of keys in the system; collectively, the storage servers hold all data stored by the system. Clients issue requests to the storage servers, both to store new objects, and to retrieve previously stored objects. The coordinator maintains meta-state for the system, specifically the partitioning of the key space across storage servers. Figure 1 illustrates Warp's overall architecture.

The Warp client library provides isolation by allowing its users to optimistically perform read and write operations that only affect local state, and verifying that isolation is not violated at commit time. To perform a read, the library retrieves the requested data from the storage servers and caches the object within the transaction context. Subsequent reads within the transaction will be satisfied by this cache, if possible. Write operations executed within the transaction are not visible on the servers immediately. To perform a write, the library saves the write to the context without contacting any storage server. Multiple writes to the same key will overwrite the locally maintained object. Transactions are unaware of any modifications written within a transaction until the client commits the transaction. At commit time, the library submits the set of all objects read and all objects written to the storage servers as a linear transaction.

2.0.1 Chain Construction

Clients use the transaction context to construct chains of servers that process transactions operations in a predictable order. The client library sorts the keys read or written by a transaction in lexicographical order, and maps the sorted list onto a set of servers. The lexicographical sort ensures that transactions with multiple keys in common pass through their shared set of servers in the same order.

Figure 2 shows how transactions that read and write the same keys have overlapping chains. Transaction T_1 reads key k_H and writes key k_A, while transaction T_2 reads keys k_P and k_T. Transaction T_3 writes keys k_A, k_H, k_P, and k_T. The object-to-server mapping dictates that T_1's chain pass through servers s_0 and s_2 because these servers hold k_A and k_H respectively. Similarly, T_2 forms a chain through s_5 and s_6. T_3 writes the same keys touched by T_1 and T_2 and has a chain that passes through all four servers. The library submits the linear transaction to the first server in the chain, which then forwards the transaction to subsequent servers in the chain.

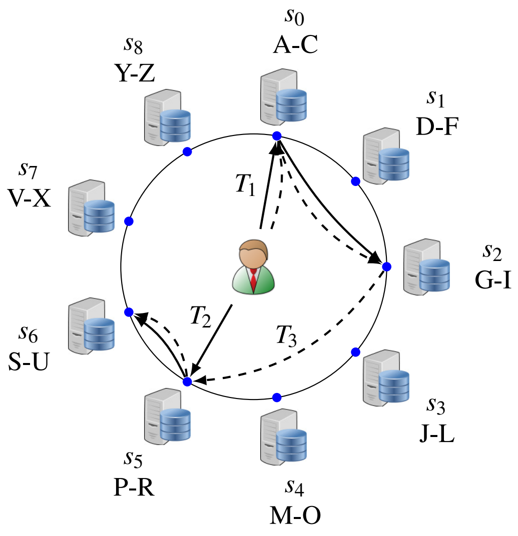


Figure 2: Clients deterministically construct dynamic chains based upon the keys read and written by transactions. In this example, a client submits T_1, T_2, and T_3. Transactions T_1 and T_2 operate on disjoint keys, $\{k_A, k_H\}$ and $\{k_P, k_T\}$ respectively. T_3 touches all four keys and forms a chain that includes the chains of T_1 and T_2

.

2.1 Commit Protocol

The linear transactions commit protocol ensures that all transactions either commit in a serializable fashion, or abort with no effect. It consists of one forward and one backward pass through each linear transaction's chain, where the storage servers directly propagate the linear transaction. The forward pass validates the values optimistically read by the client, and ensures that they remain unchanged by other transactions. Both the forward and backward passes propagate dependency information to enforce a serializable order across all transactions. The backward pass propagates the result of the transaction – whether it committed or aborted – and commits the data for committed transactions to disk. An efficient background garbage collection process limits the number of embedded dependencies by removing dependencies for transactions that have completed both passes.

2.1.1 Validation

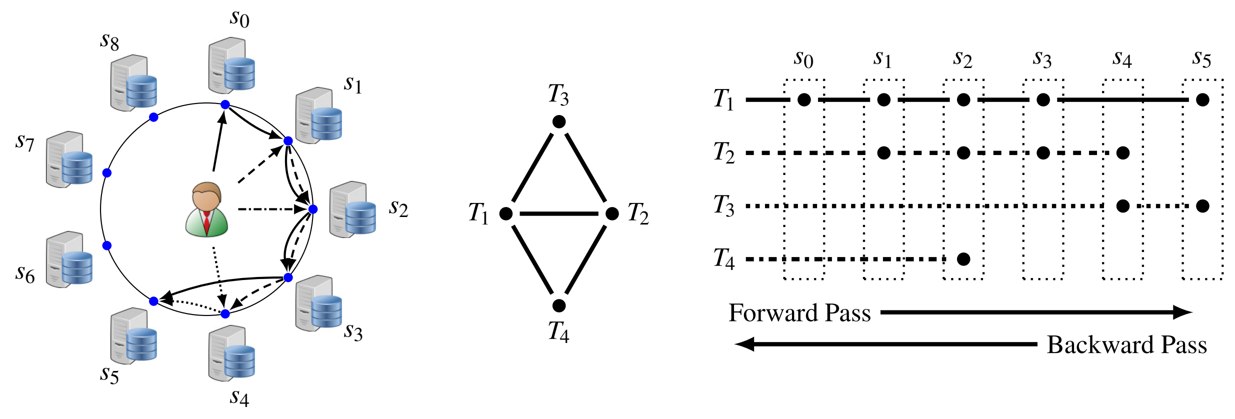
The forward pass validates transactions by ensuring that newly arriving transactions do not invalidate previously validated transactions. Servers check each transaction to ensure that it does not read values written by, or write values read by, previously validated transactions. Servers also check each value against the latest stored in its local key-value store to ensure that the value was not changed by a committed transaction. Transactions which fail either of these checks fail the validation step.

Servers abort a transaction that fails validation by sending an abort message backwards through the chain members that previously validated the transaction. These members remove the transaction from their local state, enabling other transactions to validate in its place.

Servers validate each transaction exactly once, on its forward pass through the chain; any transaction that completes its forward pass may commit at all servers. Consequently, the last server in a chain may commit the transaction immediately, and begin sending a commit message backwards through the chain.

2.1.2 Dependency Management

Providing a serializable order across all transactions requires that the transaction commit order does not create any dependency cycles. The protocol does this by maintaining a dependency graph across transactions, where the vertices are transactions and each directed edge specifies a *conflicting pair* of transactions. A conflicting pair is a pair of transactions where one transaction writes at least one key read or written by the other. In the dependency graph, the direction of an edge indicates the order in which the transactions in the conflicting pair must commit. For example, a graph with conflicting pair (T_X, T_Y) and dependency $T_X \rightsquigarrow T_Y$ tells the protocol to commit T_X before T_Y at every server.



(a) Concrete Deployment (b) Dependency Graph (c) Propagation Graph

Figure 3: Transactions are ordered by servers where they overlap.

Intuitively, the density of the dependency graph directly impacts the ways in which servers may reorder transactions. For instance, if the dependency graph has a path between every pair of vertices, then every transaction commits in exactly one order with respect to every other transaction. Committing transactions in an order that contravenes the graph breaks serializability. On the other hand, a dependency graph that consists solely of vertices with no edges permits maximum flexibility and allows transactions to commit in any order. Enforcing any commit order with such a graph would introduce spurious coordination, and reduce opportunities for optimization.

The linear transactions protocol incrementally builds dependency graphs by embedding dependency information into transactions' commit messages. Embedded within the forward and commit messages for each transaction is the complete list of dependencies that must commit before the transaction. This list corresponds to every vertex in the dependency graph that contains a path to the given transaction. For example, a transaction T_X that includes T_Y and T_Z as part of its dependency list implies that the DAG has paths $T_Y, \ldots, T_X$, and $T_Z, \ldots, T_X$. To improve efficiency, the protocol only maintains a list of dependencies, and does not need to track the structure of the graph beyond this list.

At a high level, the protocol follows two rules to ensure that servers capture all dependencies between transactions, and commit transactions in the order specified by their dependencies. First, whenever one transaction commits after another at a server, the server extends the dependency list of the first transaction to include the second transaction and the transactions in the second transaction's dependency list. Second, a transaction may only commit after all its dependencies commit; it will be delayed at the server until this rule is met.

Figure 3 shows three different ways of visualizing conflicting pairs. In Figure 3a, we see four transactions' chains starting at the client and passing through different servers. Because these transactions overlap at several servers, they form multiple conflicting pairs, shown as the undirected graph in Figure 3b. The protocol will determine the order between these transactions to determine the directionality of the edges. Figure 3c shows the same graph, separated on a per-transaction basis. Each edge in Figure 3b corresponds to one or more servers in Figure 3c where the transactions overlap. Each horizontal line in Figure 3c represents a transaction's chain and indicates which servers will execute the transaction.

2.1.3 Forward Pass

The forward pass establishes state to detect conflicting pairs, and embeds dependencies on completed transactions. Servers efficiently identify all conflicting pairs by maintaining local state on a per-key basis. For each key, a server maintains a list of every currently-executing transaction that includes the key and the set of dependencies specified by previously committed transactions that included the key. Each transaction is added to this local state on the forward pass so that other transactions may identify it as a conflicting pair on their backward pass.

To capture dependencies on completed transactions, servers augment each transaction's dependency list with the dependencies stored in the per-key state. This ensures that when one transaction completes its backward pass before another completes its forward pass, the former will be a dependency of the latter. The natural ordering of the system determined the order, and the dependencies merely capture the relationship for other servers.

2.1.4 Backward Pass

The backward pass propagates the commit message to the servers in the chain in reverse order, and, during the reverse traversal, determines if the transaction conflicts with any other transactions. For a conflicting pair of transactions, the protocol selects the last server in common between the transactions' chains, called the *decider*, to decide the commit order. Because this server is the first server encountered on both transactions' backward passes, it can decide an order and pass the requisite dependencies to every subsequent server in both chains.

Figure 3 depicts the deciders for multiple transactions. Take, for example, transactions T_1 and T_2 whose chains both pass through s_1, s_2, and s_3. The decider for this pair of transactions is server s_3 because it is the last server in common to the chains. It decides an order between T_1 and T_2, say T_1 commits before T_2, and embeds this dependency into the second transaction, i.e. T_2. The commit message for T_2 passed to server s_2 contains the information $T_1 \rightsquigarrow T_2$ as well as any dependencies transitively obtained from T_1.

Deciders add dependency information to transactions' commit messages to tell the other servers the proper order in which to commit conflicting transactions. If a transaction's commit message contains a dependency, the decider processes the two transactions in the specified order; otherwise, the decider is free to use the natural arrival order. Regardless of the commit order, the decider always embeds a dependency upon the first-to-commit into the second-to-commit transaction ensures that, no matter the arrival order of commit messages, a server will delay the dependent transaction until its dependency commits. Servers are free to commit transactions without unsatisfied dependencies in natural arrival order. For example, in Figure 3, server s_3 decides the order between T_1 and T_2 using the information embedded by s_4 and s_5. If s_4 were to embed $T_1 \rightsquigarrow T_3$ and $T_3 \rightsquigarrow T_2$ into the commit message for T_2, then s_3 would know to order T_1 first. If T_2 arrives with no such dependency, then s_3 knows it can safely commit T_2, because no matter what, T_1 may be ordered after T_2. T_1 either contains a pre-existing dependency upon T_2, or it doesn't, and s_3 will explicitly add the dependency.

Non-deciders for a conflicting pair of transactions commit the transactions in the same order as the decider. If a transaction arrives with a dependency upon another in a conflicting pair, the non-decider delays the transaction until the dependency commits. This ensures that the conflicting pair commits in the same order at every server. Servers s_1, s_2 in Figure 3 are non-deciders for the conflicting pair (T_1, T_2) and use the information from s_3 to decide commit order.

Deciders order transactions that belong to multiple conflicting pairs in a way that preserves the acyclic invariant. Take, for instance, server s_2 in Figure 3. This server is the decider for (T_2, T_4) and a non-decider for (T_1, T_2). Servers take care to ensure that decisions made as a decider remain acyclic regardless of the order decided for other conflicting pairs. The simplest way for servers to ensure this is to wait for dependency information for conflicting pairs for which they are non-deciders before deciding any other conflicting pairs. In our example, s_2 would wait for an order for (T_1, T_2) before deciding (T_2, T_4).

2.1.5 Garbage Collection

The linear transactions protocol employs garbage collection to prevent the system from indefinitely maintaining dependencies on fully committed transactions. When a transaction completes its backward pass, its dependencies are fixed, and its effects are completely reflected in the data store. Although there is no harm in allowing other transactions to include the completed transaction as a dependency, the dependency is unnecessary because the system will naturally order every subsequent transaction after the completed transaction. The garbage collection mechanism identifies completed transactions and removes them from the servers' state.

Servers efficiently communicate the set of transactions to garbage collect by assigning transactions sequential identifiers. The first server to process a transaction assigns it the next-lowest value from the server's local counter; the combination of the server's ID and the assigned value uniquely identify the transaction. Servers periodically broadcast an upper bound such that every transaction assigned a lower value by the server may be garbage collected. Thus, servers may garbage collect multiple transactions by transmitting small messages of constant size.

Upon receipt of a garbage collection broadcast, a server updates its local state to remove dependencies which have been garbage collected. It removes from each individual transaction's state any dependency upon garbage-collected transactions. A server removes a transaction from its hash table only after its dependencies are garbage collected. This ensures that transitively propagated transactions continue to propagate until they too are garbage collected.

2.2 Correctness

The linear transactions protocol maintains serializability by ensuring that the dependency graph between committed transactions is acyclic. This section provides a proof sketch for how the protocol maintains this acyclic dependency invariant at all times.

Dependency propagation retains the complete set of transitive dependencies for every transaction by ensuring that dependencies are always embedded. There are only six permutations in which the messages for an arbitrary conflicting pair (T_X, T_Y) may pass through any single server. Figure 4 enumerates all six permutations. The

1:	$F(T_X)$	$B(T_X)$	$F(T_Y)$	$B(T_Y)$
2:	$F(T_X)$	$F(T_Y)$	$B(T_X)$	$B(T_Y)$
3:	$F(T_X)$	$F(T_Y)$	$B(T_Y)$	$B(T_X)$
4:	$F(T_Y)$	$B(T_Y)$	$F(T_X)$	$B(T_X)$
5:	$F(T_Y)$	$F(T_X)$	$B(T_Y)$	$B(T_X)$
6:	$F(T_Y)$	$F(T_X)$	$B(T_X)$	$B(T_Y)$

$\xrightarrow{\hspace{8em}}$

Time

Figure 4: The six possible patterns in which the messages for conflicting pair (T_X, T_Y) may pass through a single server. Patterns 4-6 are isomorphic to patterns 1-3 respectively. $F(T_X)$ indicates the forward pass of T_X, while $B(T_X)$ indicates the backward pass of T_X. Shaded messages indicate places where servers embed dependencies.

astute reader will note that only the first three patterns are unique; the remaining patterns are isomorphic to these three. Pattern 1 captures the dependency $T_X \rightsquigarrow T_Y$ on the forward pass because the protocol embeds it directly into the message. In patterns 2 and 3, the transactions overlap in execution at the server, and the decider orders one transaction before the other; the decider embeds the requisite transitive dependencies into the second-to-commit transaction. In all six cases, the server embeds one transaction and its dependencies into the other.

The linear transactions protocol guarantees serializability because cycles between transactions cannot exist. Assume for the sake of contradiction that a cycle did exist. This would mean that a decider directly contradicted a transitively-defined, pre-existing dependency[1]. Because a decider never decides the converse of a known dependency, it follows that the decider in our hypothetical scenario did not know of the dependency. But this cannot be, because the transitive closure of a transaction's dependencies are preserved at each point where two transactions become ordered. For example, it's possible for the transitive dependency $T_1 \rightsquigarrow T_3 \rightsquigarrow T_2$ to exist in Figure 3. If server s_3 were somehow unaware of this order and specified the dependency $T_2 \rightsquigarrow T_1$, the dependency graph would contain a cycle. Server s_3 will never add such an edge, however, because the dependency $T_1 \rightsquigarrow T_2$ stems from a transitive relationship via T_3, and servers s_4 and s_5 will necessarily transmit the dependencies when ordering (T_2, T_3) and (T_1, T_3). Server s_3 would then have the information necessary to preserve the acyclic invariant of the dependency graph.

Garbage collection does not affect the correctness of the protocol because only transactions that complete their backward pass may be garbage collected. Subsequent transactions may be ordered only after the garbage collected transaction as shown by Patterns 1 and 4 in Figure 4. Because every server will naturally order subsequent transactions after the garbage-collected state, it is unnecessary to explicitly maintain a dependency.

2.3 Fault Tolerance and Durability

In a large-scale deployment, failures are inevitable. Linear transactions accommodate a natural way to overcome such failures. Specifically, linear transactions permit a subchain of $f+1$ replicas to be inlined into the longer chain in place of a single data server. This allows the system to remain available despite up to f failures within a subchain. Chain replication maintains a well-ordered series of updates within each subchain. Operations that traverse the linear transaction chain in the forward direction pass forward through all inlined chains. Likewise, operations that traverse the chain in reverse traverse inlined chains in reverse.

Note that the notion of fault-tolerance provided by linear transactions is different from the notion of durability within traditional databases. While durability ensures that data may be re-read from disk after a failure, the system remains unavailable during the failure and recovery period; in contrast, linear transactions' fault tolerance mechanism ensures that the system remains available so long as the number of failures remains below the configured threshold.

2.4 Atomicity, Consistency, Isolation

The protocol guarantees atomicity, consistency, and isolation for all transactions. These properties naturally follow from the one-copy serializability upheld by the protocol. Each transaction completes in its entirety at a well-defined point in the partial order, where its effects are either completely visible to subsequent transactions, or it aborts without effect. Every server ensures that the stored objects are well-formed and match their data types. Overall, linear transactions guarantee that operations within a transaction execute with mutual exclusion from each other, as if there were a single giant lock protecting the database.

2.5 Spurious Coordination

Linear transactions eliminate spurious coordination because servers commit transactions in natural arrival order unless doing so would violate serializability. Servers only delay processing a transaction when the transaction contains a dependency on a non-committed transaction, and the delay introduced is necessary to preserve serializability. Transactions which do not contain such dependencies commit in natural arrival order without delay. Because spurious coordination is unnecessary synchronization between transactions, and the protocol only delays/synchronizes those transactions that it can *prove*

[1]Note that a decider cannot contradict a non-transitively-defined dependency, because the decider itself is the only server that may instill a direct, non-transitive dependency.

require delay, the protocol exhibits no spurious coordination.

3 Implementation

We have fully implemented the system described in this paper. The code base consists of 89,979 lines of code, approximately 7,500 lines of which are exclusively devoted to processing transactions. The Warp distribution provides bindings for C, C++, Python, Ruby, Java and Node.JS and supports a rich API that goes well beyond the simple `get`/`put` interface of typical key-value stores. A system of virtual servers maps a small number of servers to a larger number of partitions, permitting the system to reassign partitions to servers without repartitioning the data. The implementation uses a replicated state machine as the coordinator to ensure that there are no single points of failure.

3.1 Rich API

Warp supports an expansive API that enables applications to build complex applications. The expanded API includes support for rich data structures, multiple independent schemas, and nested transactions.

3.1.1 Data Structures

Warp's API includes support for complex data structures that go beyond strings, including integer, float, list, set, and map types. Warp supports atomic operations, such as conditional updates that change an object only if it matches application-specified predicates. These predicates include equality and range comparisons, regular expressions, and checks for elements within containers. In addition to the conditional atomic operations, Warp supports atomic mathematical operations on numeric value types. For container types, clients may atomically add and remove objects from the containers, and perform nested operations on the contained values. Overall, the Warp API consists of sixty-two different operations that clients may use within a transaction.

3.1.2 Independent Schemas

Warp enables applications to create multiple, independent, schemas each of which specify multiple typed attributes that comprise the object. These schemas resemble tables from traditional database systems. Storage servers validate that objects match the schema under which they are stored. A linear transaction may read and write objects in multiple schemas without restriction. Clients construct the chain for transactions that touch multiple schemas by lexicographically ordering servers first by schema, then by key.

3.1.3 Nested Transactions

Warp supports arbitrarily nested transactions that enable clients to build complex applications. Each nested transaction maintains its own locally-managed transaction context with a pointer to the parent transaction's context. Reads recursively query the parent context until either a cached value is read, or the root context issues the query to a storage server. Writes are stored in the transaction context to which they are issued. At commit time, the client merges a nested transaction into its parent context, by merging the read and write sets. Nested transactions abort if the values read in the child are modified in the parent or vice-versa. The client sends a linear transaction to the storage servers only when the root transaction commits.

3.2 Virtual Servers

Warp uses a system of virtual servers to map multiple partitions of the mapping to a single server. Clients construct their linear transaction chains by constructing a chain through the virtual servers, and then mapping these virtual servers to their respective servers. A server that maps to multiple virtual servers in a chain will appear at multiple places in the chain, where it acts as each of its virtual servers independently. Within each physical server, state is partitioned by virtual server, so that each virtual server functions as if it were independent.

3.3 Coordinator

A replicated state machine called the coordinator partitions the key space across all data servers, ensures balanced key distribution, and facilitates membership changes as servers leave and join the cluster. Since the coordinator is not on the data path, it's implementation is not critical to the performance of linear transactions.

The coordinator partitions data across servers and ensures balanced key distribution by using Copyset Replication [10] to group servers into replica sets. Each independent schema is partitioned across the generated copysets to create an object-to-server mapping. The coordinator over-partitions the key space to enable it to remap partitions from over-burdened replica sets to under-loaded replica sets if necessary.

As servers join and leave a cluster, the coordinator regenerates copysets to respond to new members. Servers dynamically compute the previous and next servers in each linear transaction's chain using the mapping; when the mapping changes, servers retransmit transactions whose chain changed. Because transactions are only garbage collected after they complete, and servers only retransmit incomplete transactions, servers are always able to retransmit the requisite transactions.

The coordinator is implemented on top of the Replicant replicated state machine system. Replicant uses chain replication [44] to sequence the input to the state machine and a quorum-based protocol to reconfigure chains on failure. The details of Replicant are beyond the scope of this paper; the function of the coordinator could easily be taken on by configuration services such as Chubby [7], ZooKeeper [22], or OpenReplica [3].

4 Evaluation

We evaluate Warp using both macro and micro benchmarks against other storage systems using the TPC-C benchmark. The primary focus of our evaluation is on examining the performance of Warp transactions relative to other transaction processing techniques. To that end, implemented Sinfonia's mini-transacitons [2] on top of HyperDex (here-after referred to as 2PCDex). Because Warp is based upon HyperDex, we ran all benchmarks against a non-transactional HyperDex to quantify Warp's overhead. We ensure a true apples-to-apples comparison by building all three systems use the same code base, with minimal changes to the commit protocol; the client-facing interfaces are identical, and the benchmark code is identical.

We performed our experiments on our dedicated lab-size cluster consisting of thirteen servers, each of which is equipped with two Intel Xeon 2.5 GHz E5420 processors, 16 GB of RAM, 500 GB SATA 3.0 Gbit/s hard disks, and Gigabit Ethernet. The servers are running 64-bit Debian 7 with the Linux 3.2 kernel. We deployed HyperDex, 2PCDex, and Warp on each server.

Each storage system was configured with appropriate settings for a real deployment of this size. This includes setting the replication factor to be the minimum value necessary to tolerate one failure of any process or machine. This means that both the coordinators and the storage servers can each tolerate one failure. HyperDex is configured to provide linearizability, while 2PCDex and Warp provide full one-copy serializability.

4.1 TPC-C

The TPC-C benchmark simulates an e-commerce application by specifying a mixed transaction workload. The workload specified by TPC-C is inherently difficult to process because it includes both read-heavy and update-heavy transaction profiles and the update-heavy transactions often include updates to a small number of keys in the system. For instance, the *new-order* transaction generates the order's identifier using a sequentially-increasing counter associated with one of one-hundred districts. The *payment* transaction increments the year-to-date totals for one of one-hundred districts and one of ten warehouses. Consequently, the workload necessarily requires that transactions operate on a small amount of shared data.

We implemented four of the five TPC-C transaction profiles and retained as much functionality of the benchmark as is feasible to implement on a key-value store. All operations operate on primary keys, to conform to

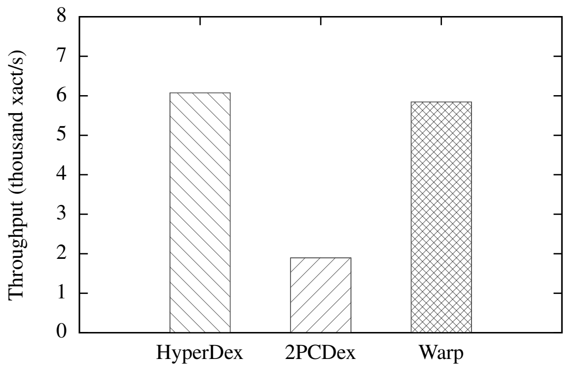


Figure 5: Total transactional throughput of the three systems. Warp outperforms 2PCDex by a factor of 3.2, and achieves 96% the throughput of HyperDex, which Warp uses as its underlying key-value store.

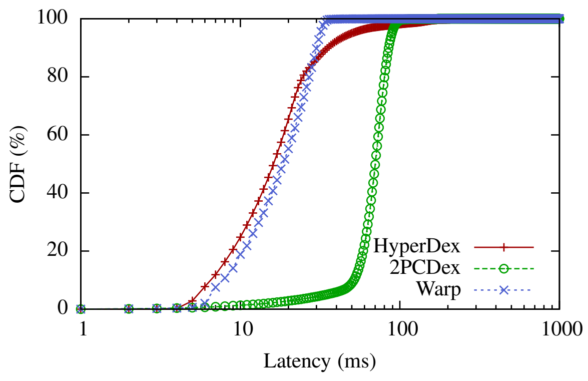
Profile	R	W	RMW	% Mix
New Order	12	3	11 (1)	45
Payment	0	1	3 (2)	45
Order Status	12	0	0	5
Stock Level	201 (1)	0	0	5

Table 1: A summary of the workloads of TPC-C. For each transaction profile, the chart shows the average number of read-only (R), write-only (W), and read-modify-write (RMW) operations. The last column shows the distribution of profiles in the randomly generated set of transactions. The majority of the workload consists of new-order and payment transactions which are both write-heavy transactions that each update one to two hot keys.

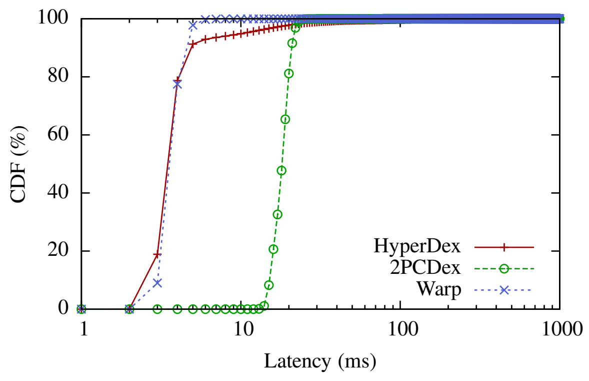
the key-value model; where the TPC-C benchmark specified a mix of secondary-attribute search and primary-key retrieval, we always retrieved by primary key. We did not implement the delivery transaction because it is specified as a background process and would most likely be handled by a queue mechanism in a real deployment. Where possible, we use Warp's atomic addition primitives rather than reading the old value and writing a new value. Table 1 provides an overview of each transaction type, and describes the average number of objects each transaction will read (R), write (W), and read-modify-write (RMW). For each category, we specify in parenthesis the number of hot objects – that is district or warehouse objects – the transaction includes.

Figure 5 shows the overall transactional throughput for HyperDex, 2PCDex, and Warp. We can see that Warp achieves a throughput that is 3.2 times higher than 2PCDex, and 96% the throughput of HyperDex.

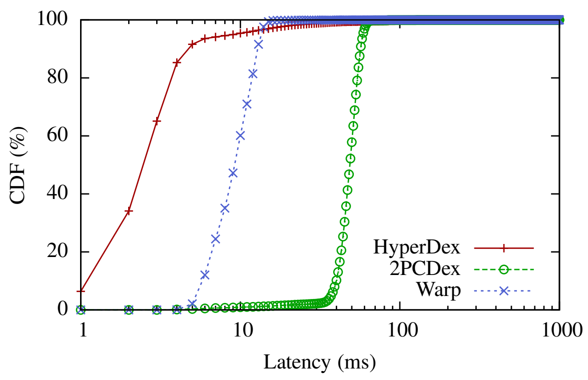
Warp's throughput is nearly identical to HyperDex's throughput because the most new-order and payment transactions that comprise the bulk of all transactions



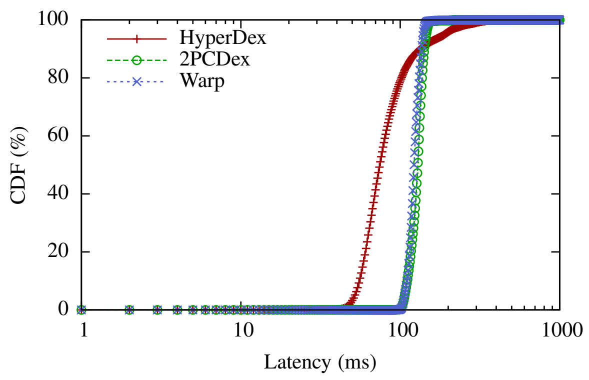
(a) New Order Transactions



(b) Payment Transactions



(c) Order Status Transactions



(d) Stock Level Transactions

Figure 6: CDFs of latency for each transaction type.

have similar latency for HyperDex and Warp. Figures 6a and 6b show latency CDFs for the new-order and payment transaction profiles.

Write operations' latency is mostly independent from whether or not the writes are performed within a transaction because the number of round-trips required to commit the transaction is the same as the number of round trips required to perform the writes individually. For example, a client that performs three writes that traverse a chain of length two will incur a cost of six round-trips. A linear transaction across all three writes will be six nodes long and incur the same three round-trip cost.

For read operations, Warp incurs a predictable communication cost, shown in Figures 6c and 6d. The cost of a transactional read is the cost of the optimistic read to bring the value to the client, plus the cost of a pass through the transaction chain. Consequently, a transactional read is measurably more expensive as shown in Figure 6c. Figure 6d shows the stock-level transaction which executes within a transaction context, but does not commit the transaction because the TPC-C benchmark does not require that the transaction be isolated. These two figures highlight the cost of transactional reads; the former shows the overall cost, while the latter underscores the fact that the cost is only paid at commit time.

2PCDex's mini-transactions exhibit significantly higher latency because they are more likely to abort. Servers in our TPC-C benchmark retry aborted transactions until they succeed before proceeding to subsequent transactions. 2PCDex aborts transactions that operate on the same keys, while Warp does not abort any transactions in the benchmark. Figure 7 shows the abort rates for 2PCDex and Warp. Only 5% of 2PCDex transactions complete immediately; the rest abort and need to retry. Warp completes 100% of transactions on the first try.

Although it may seem possible to relax the mini-transactions protocol to permit transactions to write to the same key simultaneously, doing so would break serializability. A modified 2PCDex could not prevent the potential cycle illustrated in Figure 3, as concurrently prepared transactions could commit in different orders on different servers. Even the atomic operations provided by HyperDex cannot enable such a relaxed commit protocol.

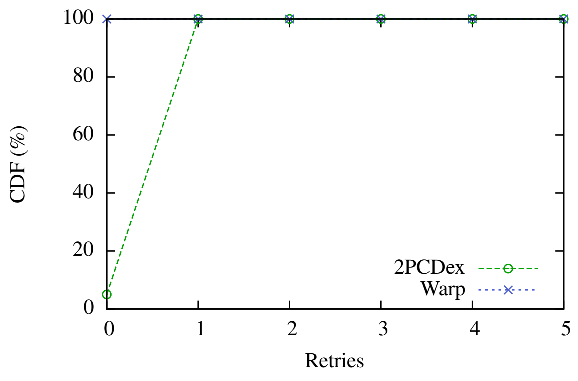


Figure 7: A CDF showing how frequently transactions abort and retry in the TPC-C benchmark. Warp does not retry any transactions, while 95% of 2PCDex transactions abort and retry.

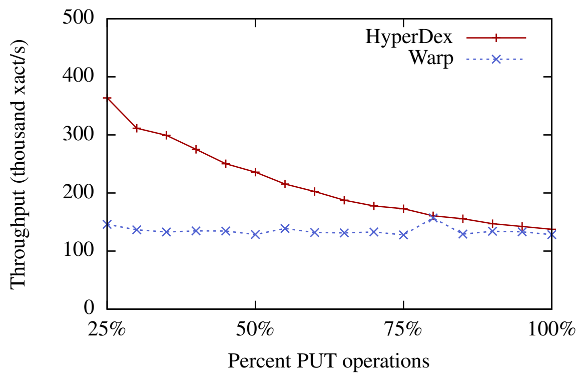


Figure 8: The ratio of read/write operations does not materially affect the throughput for transactions. The dominating cost of a transaction is a round trip though the transaction chain. In this experiment, the transactions are fixed in size, and thus their chains' lengths are fixed. Consequently, we see that throughput is fixed as well.

4.2 Micro-Benchmarks

In order to gain insight into the behavior of Warp's linear transactions, we examine the results from several micro-benchmarks that expose the behavior of Warp. In all of these micro-benchmarks, objects have 12 B keys and 64 B values, and are constructed uniformly at random. Ten million objects are preloaded onto the cluster before performing each benchmark.

4.2.1 Read/Write Ratio

In order to quantify the effects of the read/write ratio on a transactions' throughput, we constructed a micro-benchmark that varies the read-write ratio for operations of constant size. This micro-benchmark constructs transactions that involve exactly eight objects, and randomly read from or write to random objects. Each operation is randomly chosen to be a read or a write so that the total percentage of write operations matches the independent variable. Figure 8 shows the average throughput achievable for given read/write ratios.

4.2.2 Transaction Size

Naturally, the use of chains introduces a trade off: as transactions grow to contain more keys, the length of the resulting chains naturally increases as well. Figure 9 quantifies this trade-off by constructing write transactions with different numbers of keys. We employ the same micro-benchmark from the previous section, and use a 100% write workload.

To test the performance impact of transaction size, we modified our previous microbenchmark to vary the number of keys in a transaction rather than the read/write ratio. In this experiment, the microbenchmark issues transactions with a configurable number of `put` operations on random keys. Figure 9 shows the results of this experiment. We can see that, as expected, the number of operations per second is relatively independent of the transaction size. This demonstrates that longer transaction chains do not introduce additional overhead, and that, for this workload, the transaction rate is a linear function of the transaction size.

4.2.3 Scalability

The performance of linear transactions should scale linearly with the number of servers in the cluster, as the number of servers that participate in a linear transaction is dependent only on the transaction size. Adding more servers to the cluster should therefore yield a proportional increase in performance by spreading the work across more servers. Figure 10 shows the aggregate throughput of a two-key transaction from our micro-benchmark with different cluster sizes. Not surprisingly, Warp throughput scales linearly with cluster size.

5 Related Work

Transaction management has been an active research topic since the early days of distributed database systems. Existing approaches can be broadly classified into the following categories based upon the mechanisms employed and resulting guarantees.

Centralized: Early RDBMS systems relied on physically centralized transaction managers [8]. While centralization greatly simplifies the implementation of a transaction manager, it poses a performance and scalability bottleneck and is a single point of failure. Warp, like many other systems, is based on a distributed architecture.

Distributed: The traditional approach to distributing transaction management is to provide a set of specialized transaction managers that serve as intermediaries

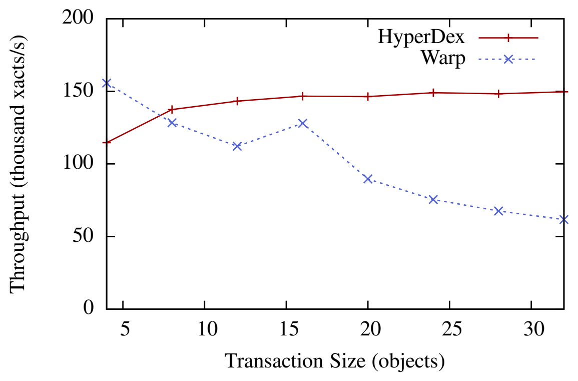


Figure 9: The total throughput of Warp is dependent on the throughput of the underlying key-value store, and largely independent of the transaction size. This graph shows the throughput of a 100% write workload as the number of keys in a transaction increases.

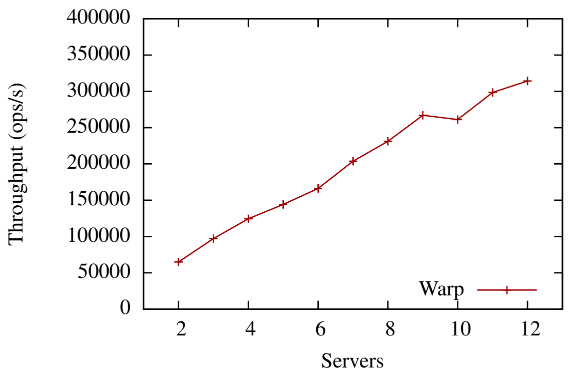


Figure 10: Warp is a scalable system. This graph shows the aggregate throughput of the system as servers are added. The number of clients and the workload remain fixed for all deployment sizes. Each point represents the average across three runs. With each additional server, the overall throughput increases proportionally, exhibiting linear scaling.

between clients and back-end data servers. These transaction managers perform lock or timestamp management [6], and employ a protocol, such as two phase-commit, for coordination. Warp does not employ any specialized transaction managers; storage servers directly execute transactions over their data.

A recent proposal [30] suggests physically separating the transaction processing component from the storage component so that transaction processing remains agnostic to the structure of the storage. Deuteronomy [27] separates the transactional component from the data storage component so that transaction management remains isolated from scaling decisions made at the storage layer. ElasTraS [14] uses two layers of transaction management to separately process read-only and read-write transactions, where each layer and the underlying storage are independently scalable.

Separating transaction management from data storage does not fundamentally make the transaction management more scalable because it does not alter the spurious coordination naturally present in the transaction manager. Warp's eliminates spurious coordination and centralized bottlenecks by employing a completely distributed protocol.

Consensus-based: Recent work has examined how to use a general consensus protocol, such as Paxos [25] or Zab [22], to serialize transactions in a fault-tolerant manner. Although consensus seems unrelated to transaction management, the classic two-phase commit algorithm is actually a special $f = 0$ case of Paxos that cannot tolerate coordinator failure [20].

Straightforward application of consensus protocols, however, would maximize spurious coordination by applying a total order across all transactions. Consequently, consensus-based systems typically use some combination of data partitioning [5, 19, 37, 39], Generalized Paxos [23] or transaction batching [38, 43] to increase opportunities for parallel execution.

Warp, too, uses consensus, but only to maintain system meta-state. The linear transactions protocol, inspired by chain-replication [44] and value-dependent chaining [17], relies upon consensus for system membership and coordination, but never for actual transaction processing. Because consensus is not on the data path for any linear transaction, the protocol is able to completely eliminate any consensus-induced spurious coordination.

Synchronized clocks: Some notable systems in this space take advantage of synchronized clocks to order transactions. Adya et. al. [1] support serializable transactions and use loosely synchronized clocks as a performance optimization. Spanner [12] uses tightly synchronized clocks, with bounded error, to achieve high-throughput and external consistency for transactions across multiple data centers.. Granola [13] orders independent transactions with no locking overhead or abort mechanism, and orders these transactions using time synchronization as an optimization.

Warp makes no assumptions about clock synchrony, and consequently is impervious to the negative side-effects of incorrectly assuming too-tight of a synchronization. Most systems in this category remain correct should synchronicity assumptions be violated, but suffer varying degrees of performance degradation. A notable exception is Spanner, which preserves serializability only when its assumptions are upheld. Consequently, such systems require more tuning and operations overhead than Warp.

Client-managed transactions: Some systems build on existing storage by implementing transactions directly in the client library. Such systems mediate concurrent transactions by embedding additional attributes into the stored objects to enable concurrency control. CrSO [18] uses HBase versions and a centralized status oracle to check for read-write or write-write conflicts at commit time. Percolator [32] maintains Google's search index by storing both data and locks in BigTable.

The downside to client-managed approaches is that they require mechanisms to cope with client failure. CrSO requires a background process to cleanup stale versions of objects written in failed transactions. Percolator uses a background mechanism to break locks held by failed processes. Warp incurs no such cost because failed clients leave behind no state to cleanup.

Geo-Replication: For geo-replicated storage, many systems avoid synchronous WAN latencies by making guarantees weaker than serializability. COPS-GT [28] and Eiger [29] provide read and write transactions, respectively, that commit locally propagate to remote data centers in a causally-consistent fashion. Walter [40] implements parallel snapshot isolation using counting sets to resolve conflicting versions, similar to commutative data types [26]. Lynx [45] uses chains for geo-replication, but requires a priori knowledge of transactions and static analysis to prevent non-serializable executions.

Warp provides a strictly stronger guarantee of general purpose serializable transactions, but lacks optimizations for geo-replication. We believe that the end-to-end principle should be applied to cross-data center applications, because the guarantees required by applications are more readily met with application-specific mechanism than data-store specific features.

Workload Partitioning: Some systems improve performance by constraining applications' transactions to operate within single partitions of the data store. G-Store [15] provides serializable transactions on top of HBase, by grouping keys' primary replicas on a single server so that transactions require no cross-server communication. H-Store [42] targets OLTP applications and efficiently supports such constrained tree applications by guaranteeing that transactions are executed by a single server. Warp imposes no constraint on transactions' partitioning, enabling maximal flexibility in data placement.

Mini-Transactions: Sinfonia [2] introduces the mini-transaction primitive which allows an application to specify sets of checks, reads, and writes and commit the result using a modified two-phase commit. Although the content that a client submits in linear transaction resembles that of a mini-transaction, Warp and Sinfonia differ in their commit behavior. Sinfonia will abort concurrent transactions that write to the same keys, even in write-only transactions. Warp allows multiple transactions that write the same key to simultaneously execute and commit in a serializable order. This difference cannot be overcome by simply loosening the commit requirements within Sinfonia, because the resulting mechanism would have no way to enforce an acyclic dependency graph.

NoSQL Stores: NoSQL systems are defined by their distributed architecture that offers performance and scalability, often obtained by avoiding strong consistency or transactional guarantees. Note that the trade-off is often an engineering decision to mask latency, and is not fundamental. Amazon's Dynamo Dynamo [16] and its derivatives [24, 33, 35] guarantee only eventual consistency in order to increase write availability by writing data to sloppy quorums. Yahoo!'s PNUTS [11] guarantees a slightly stronger timeline consistency, but makes no guarantee of cross-object atomicity or isolation. Google's BigTable [9] provides linearizable access to individual rows, but does not make cross-object guarantees. BigTable's consistency is the same as HyperDex [17], the system Warp builds upon. Warp's guarantee is strictly stronger as it extends serializability across multiple objects.

More generally, these NoSQL systems have roots in Distributed Data Structures [21] and distributed hash tables [34, 36, 41, 46], which provide efficient access to individual objects, usually in the form of a key-value store. Other notable work on key-value stores includes FAWN-KV [4], a linearizable key-value store built to reduce power consumption in storage systems, and RAMCloud [31], which builds a key-value store for low-latency networks. The goals of these systems are orthogonal to those in Warp, and the techniques could be combined to make a transactional key-value store with low power consumption (maximizing transactions per watt), or low latency (minimizing transaction completion time).

6 Conclusion

This paper described Warp, a key-value store that provides one-copy-serializable ACID transactions. The main insight behind Warp is a protocol called linear transactions which enables the system to completely distribute the task of ordering transactions. Consequently, transactions on separate servers will not require expensive coordination and the number of servers that process a transaction is independent of the number of servers in the system. The system achieves high performance on a variety of standard benchmarks, performing nearly as well as the non-transactional key-value store that Warp builds upon.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely

Synchronized Clocks. In Proceedings of the *SIGMOD International Conference on Management of Data,* pages 23-34, San Jose, California, May 1995.

[2] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. Sinfonia: A New Paradigm For Building Scalable Distributed Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 159-174, Stevenson, Washington, October 2007.

[3] Deniz Altınbüken and Emin Gün Sirer. Commodifying Replicated State Machines With OpenReplica. Cornell University, Technical Report, 2012.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array Of Wimpy Nodes. In Proceedings of the *Symposium on Operating Systems Principles,* pages 1-14, Big Sky, Montana, October 2009.

[5] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage For Interactive Services. In Proceedings of the *Conference on Innovative Data Systems Research,* pages 223-234, Asilomar, California, January 2011.

[6] Philip A. Bernstein and Nathan Goodman. Concurrency Control In Distributed Database Systems. In *ACM Computing Surveys,* 13(2):185-221, 1981.

[7] Michael Burrows. The Chubby Lock Service For Loosely-Coupled Distributed Systems. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 335-350, Seattle, Washington, November 2006.

[8] Donald D. Chamberlin, A. M. Gilbert, and Robert A. Yost. A History Of System R And SQL/Data System. In Proceedings of the *International Conference on Very Large Data Bases,* pages 456-464, 1981.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System For Structured Data. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 205-218, Seattle, Washington, November 2006.

[10] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and and Mendel Rosenblum. Copysets: Reducing The Frequency Of Data Loss In Cloud Storage. In Proceedings of the *USENIX Annual Technical Conference,* pages 37–48, San Jose, California, June 2013.

[11] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. In *Proceedings of the VLDB Endowment,* 1(2):1277-1288, 2008.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. In *ACM Transactions on Computer Systems,* 31(3):8, 2013.

[13] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In Proceedings of the *USENIX Annual Technical Conference,* 2012.

[14] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic, Scalable, And Self-Managing Transactional Database For The Cloud. In *ACM Transactions on Database Systems,* 38(1):5, 2013.

[15] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-Store: A Scalable Data Store For Transactional Multi Key Access In The Cloud. In Proceedings of the *Symposium on Cloud Computing,* pages 163-174, Indianapolis, Indiana, June 2010.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In Proceedings of the *Symposium on Operating Systems Principles,* pages 205-220, Stevenson, Washington, October 2007.

[17] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proceedings of the *SIGCOMM Conference,* pages 25-36, Helsinki, Finland, August 2012.

[18] Daniel Gómez Ferro, Flavio Junqueira, Benjamin Reed, and Maysam Yabandeh. Lock-Free Transactional Support For Distributed Data Stores. Poster Session. Symposium on Operating Systems Principles, Cascais, Portugal, 2011.

[19] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas E. Anderson. Scalable Consistency In Scatter. In Proceedings of the *Symposium on Operating Systems Principles,* pages 15-28, Cascais, Portugal, October 2011.

[20] Jim Gray and Leslie Lamport. Consensus On Transaction Commit. In *ACM Transactions on Database Systems,* 31(1):133-160, 2006.

[21] Steven D. Gribble. A Design Framework And A Scalable Storage Platform To Simplify Internet Service Construction. PhD thesis, U.C. Berkeley, 2000.

[22] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-Free Coordination For Internet-Scale Systems. In Proceedings of the *USENIX Annual Technical Conference,* 2010.

[23] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-Data Center Consistency. In *The Computing Research Repository,* abs/1203.6049, 2012.

[24] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. In Proceedings of the *International Workshop on Large Scale Distributed Systems and Middleware,* Big Sky, Montana, October 2009.

[25] Leslie Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems,* 16(2):133-169, 1998.

[26] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. CRDTs: Consistency Without Concurrency Control. In *The Computing Research Repository,* abs/0907.0929, 2009.

[27] Justin J. Levandoski, David B. Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction Support For Cloud Data. In Proceedings of the *Conference on Innovative Data Systems Research,* pages 123-133, Asilomar, California, January 2011.

[28] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle For Eventual: Scalable Causal Consistency For Wide-Area Storage With COPS. In Proceedings of the *Symposium on Operating Systems Principles,* pages 401-416, Cascais, Portugal, October 2011.

[29] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics For Low-Latency Geo-Replicated Storage. In Proceedings of the *Symposium on Networked System Design and Implementation,* Lombard, Illinois, April 2013.

[30] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling Transaction Services In The Cloud. In Proceedings of the *Conference on Innovative Data Systems Research,* Asilomar, California, January 2009.

[31] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, and Mendel Rosenblum. Fast Crash Recovery In RAMCloud. In Proceedings of the *Symposium on Operating Systems Principles,* pages 29-41, Cascais, Portugal, October 2011.

[32] Daniel Peng and Frank Dabek. Large-Scale Incremental Processing Using Distributed Transactions And Notifications. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 251-264, Vancouver, Canada, October 2010.

[33] Project Voldemort. `http://project-voldemort.com/`.

[34] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the *SIGCOMM Conference,* pages 161-172, San Diego, California, August 2001.

[35] Riak. `http://basho.com/`.

[36] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, And Routing For Large-Scale Peer-To-Peer Systems. In Proceedings of the *IFIP/ACM International Conference on Distributed Systems Platforms,* pages 329-350, 2001.

[37] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. Scalaris: Reliable Transactional P2p Key/value Store. In Proceedings of the *SIGPLAN Workshop on ERLANG,* pages 41-48, Victoria, Canada, 2008.

[38] Daniele Sciascia and Fernando Pedone. Geo-Replicated Storage With Scalable Deferred Update Replication. In Proceedings of the *International Conference on Dependable Systems and Networks,* pages 1-12, Budapest, Hungary, June 2013.

[39] Daniele Sciascia, Fernando Pedone, and Flavio Junqueira. Scalable Deferred Update Replication. In Proceedings of the *International Conference on Dependable Systems and Networks,* pages 1-12, Boston, Massachusetts, June 2012.

[40] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage For Geo-Replicated Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 385-400, Cascais, Portugal, October 2011.

[41] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service For Internet Applications. In Proceedings of the *SIGCOMM Conference,* pages 149-160, San Diego, California, August 2001.

[42] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End Of An Architectural Era (It's Time For A Complete Rewrite). In Proceedings of the *International Conference on Very Large Data Bases,* pages 1150-1160, 2007.

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

[44] Robbert van Renesse and Fred B. Schneider. Chain Replication For Supporting High Throughput And Availability. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 91-104, San Francisco, California, December 2004.

[45] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability With Low Latency In Geo-Distributed Storage Systems. In Proceedings of the *Symposium on Operating Systems Principles,* Pennsylvania, November 2013.

[46] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: A Fault-Tolerant Wide-Area Application Infrastructure. In *SIGCOMM Computer Communications Review,* 32(1):81, 2002.