Anna: A KVS For Any Scale

Chenggang Wu [#1], Jose M. Faleiro [#2], Yihan Lin [*3], Joseph M. Hellerstein [#4]

[#] *UC Berkeley*
USA
[1] cgwu@berkeley.edu
[2] jmfaleiro@berkeley.edu
[4] hellerstein@berkeley.edu
[*] *Columbia University*
USA
[3] yihan.lin@columbia.edu

***Abstract*—Modern cloud providers offer dense hardware with multiple cores and large memories, hosted in global platforms. This raises the challenge of implementing high-performance software systems that can effectively scale from a single core to multicore to the globe. Conventional wisdom says that software designed for one scale point needs to be rewritten when scaling up by $10-100\times$ [1]. In contrast, we explore how a system can be architected to scale across many orders of magnitude by design.**

We explore this challenge in the context of a new key-value store system called Anna: a partitioned, multi-mastered system that achieves high performance and elasticity via wait-free execution and coordination-free consistency. Our design rests on a simple architecture of coordination-free actors that perform state update via merge of lattice-based composite data structures. We demonstrate that a wide variety of consistency models can be elegantly implemented in this architecture with unprecedented consistency, smooth fine-grained elasticity, and performance that far exceeds the state of the art.

I. Introduction

High performance key-value storage (KVS) systems are the backbone of many large-scale applications ranging from retail shopping carts to machine learning parameter servers. Many KVS systems are designed for large-scale and even globally-distributed usage (e.g., [2]–[4]); others are designed for high-performance single-machine settings (e.g., [5], [6]). In recent years, these distinct hardware targets have begun to converge in the cloud. For example, Amazon now offers EC2 instances with up to 64 physical cores, while continuing to provide the ability to scale across machines, racks and the globe.

Given the convergence of dense hardware and the globe-spanning cloud, we set out to design a KVS that can run well at any scale: providing excellent performance on a single multi-core machine, while scaling up elastically to geo-distributed cloud deployment. In addition to wide-ranging architectural flexibility, we wanted to provide a wide range of consistency semantics as well, to support a variety of application needs.

In order to achieve these goals, we found that four design requirements emerged naturally. The first two are traditional aspects of global-scale data systems. To ensure data scaling, we assumed from the outset that we need to ***partition*** (shard) the key space, not only across nodes at cloud scale but also across cores for high performance. Second, to enable workload scaling, we need to employ ***multi-master replication*** to concurrently serve puts and gets against a single key from multiple threads.

The next two design requirements followed from our ambitions for performance and generality. To achieve maximum hardware utilization and performance within a multi-core machine, our third requirement was to guarantee ***wait-free execution***, meaning that each thread is always doing useful work (serving requests), and never waiting for other threads for reasons of consistency or semantics. To that end, coordination techniques such as locking, consensus protocols or even "lock-free" retries [7] need to be avoided. Finally, to support a wide range of application semantics without compromising our other goals, we require a unified implementation for a wide range of ***coordination-free consistency models*** [8].

Given these design constraints, we developed a system called Anna[1], which meets our performance goals at multiple scales and consistency levels. The architecture of Anna is based on a simple design pattern of coordination-free actors (Section V), each having private memory and a thread of execution mapped to a single core. Actors communicate explicitly via messaging, be it across nodes (via a network) or within a multi-core machine (via message queues [10]). To ensure wait-free execution, Anna actors never coordinate; they only communicate with each other to lazily exchange updates, or repartition state. Finally, as discussed in Section VI, Anna provides replica consistency in a new way: by using lattice composition to implement recent research in coordination-free consistency. This design pattern is uniform across threads, machines, and data-centers, which leads to a system that is simple and easy to reconfigure dynamically.

This paper describes the design and implementation of Anna, providing a set of architectural and experimental lessons for designing across scales:

- **Coordination-free Actors:** We confirm that the coordination-free actor model provides excellent performance from individual multi-core machines up to widely distributed settings, besting state-of-the-art lock-free shared memory implementations while scaling smoothly and making repartitioning for elasticity extremely responsive.

[1]The tiny Anna's hummingbird, a native of California, is the fastest animal on earth relative to its size [9].

- **Lattice-Powered, Coordination-Free Consistency:** We show that the full range of coordination-free consistency models taxonomized by Bailis, et al. [8] can be elegantly implemented in the framework of distributed lattices [11], [12], using only very simple structures in a compositional fashion. The resulting consistency code is small and modular: each of our consistency levels differs by at most 60 lines of C++ code from our baseline.
- **Cross-Scale Validation:** We perform comparison against popular KVSs designed for different scale points: Redis [6] for single-node settings, and Apache Cassandra [3] for geo-replicated settings. We see that Anna's performance is competitive at both scales while offering a wider range of consistency levels.

II. Related Work

Anna is differentiated from the many KVS designs in the literature in its assumptions and hence in its design. Anna was inspired by a variety of work in distributed and parallel programming, distributed and parallel databases, and distributed consistency.

A. Programming Models

The Coordination-free actor model can be viewed as an extension to distributed event-loop programming, notably Hewitt's Actor model [13], more recently popularized in Erlang and Akka. Anna follows the Actor spirit of independent local agents communicating asynchronously, but differs from Actors in its use of monotonic programming in the style of Bloom [14] and CRDTs [11], providing a formal foundation for reasoning about distributed consistency. Anna's actors also bear some resemblance to SEDA [15], but SEDA focuses on preemptable thread pools and message queues, whereas Anna's actors target a thread-per-core model with lattices to ensure consistency and performance.

Recent systems, such as ReactDB [16] and Orleans [17] also explore Actor-oriented programming models for distributed data. In both those cases, the Actor model is extended to provide a higher level abstraction as part of a novel programming paradigm for users. By contrast, Anna does not attempt to change user APIs or programming models; it exposes a simple key/value API to external applications. Meanwhile, those systems do not explore the use of lattice-oriented actors.

B. Key-value Stores

Figure I shows a taxonomy of existing KVS systems based on the scale at which they are designed to operate, the memory model, and the per-key as well as multi-key consistency levels supported. The remainder of this section discusses the state of the art in KVS systems in the context of the four design requirements (Section I) for building any-scale KVS.

1) Single-server storage systems: Most single-server KVS systems today are designed to efficiently exploit multi-core parallelism. These multi-core-optimized KVS systems typically guarantee that reads and writes against a single key are linearizable.

Shared memory is the architecture of choice for most single-server KVS systems. Masstree [18] and Bw-tree [19] employ a shared-memory design. Furthermore, the single-server mechanisms within distributed KVS systems, such as memcached [5] and MongoDB [4], also employ a shared-memory architecture per node. Shared-memory architectures use synchronization mechanisms such as latches or atomic instructions to protect the integrity of shared data-structures, which can significantly inhibit multi-core scalability under contention [7].

PALM [20] and MICA [21][2] each employ a partitioned architecture, assigning non-overlapping shards of key-value pairs to each system thread. KVS operations can therefore be performed without any synchronization because they are race-free by default. However, threads in partitioned systems (with ***single-master*** request handling) are prone to under-utilization if a subset of shards receive a disproportionate fraction of requests due to workload skew. To address workload skew, both PALM and MICA make selective use of shared-memory design principles. For instance, MICA processes only writes in a partitioned fashion, but allows any thread to process reads against a particular key.

Redis [6] uses a single-threaded model. Redis permits operations on multiple keys in a single request and guarantees serializability. While single-threaded execution avoids shared-memory synchronization overheads, it cannot take any advantage of multi-core parallelism.

The systems above are carefully designed to execute efficiently on a single server. Except for Redis, they all use shared-memory accesses; some directly employ the shared-memory architecture, while others employ a partitioned ("shared-nothing") architecture but selectively exploit shared memory to ameliorate skew. The designs of these systems are therefore specific to a single server, and cannot be generalized to a distributed system. Moreover, the shared-memory model is at odds with ***wait-free execution*** (Section IV), and therefore does not meet our performance requirement for any-scale KVS.

Moreover, as noted in Figure I, prior single-node KVS systems invariably provide only a single form of consistency; typically either linearizability or serializability. Furthermore, with the exception of Redis, which is single-threaded, none of the single-node KVS systems provide any consistency guarantees for multi-key operations for groups of keys. Hence, these systems choose a different design point than we explore: they offer strong consistency at the expense of performance and scalability.

2) Distributed KVS: As shown in Figure I, the majority of distributed KVS systems are not designed to run on a single multi-core machine, and it is unclear how they exploit multi-core parallelism (if at all). The exceptions are H-Store [22] and ScyllaDB [23]. Within a single machine, these systems partition the key-value index across threads, which communicate via explicit message-passing. However, as discussed earlier, partitioned systems with single-master request handling cannot scale well under skewed workload.

[2]Note that MICA is a key-value *cache*, and can hence evict key-value pairs from an index in order to bound its memory footprint for improved cache-locality.

System	Scale	Memory Model	Per-Key Consistency	Multi-key Consistency
Masstree	M	SM	Linearizable	None
Bw-tree	M	SM	Linearizable	None
PALM	M	SM	Linearizable	None
MICA	M	SM	Linearizable	None
Redis	S	N/A	Linearizable	Serializable
COPS, Bolt-on	D	MP	Causal	Causal
Bayou	D	MP	Eventual, Monotonic Reads/Writes, Read Your Writes	Eventual
Dynamo	D	MP	Linearizable, Eventual	None
Cassandra	D	MP	Linearizable, Eventual	None
PNUTS	D	MP	Linearizable Writes, Monotonic Reads	None
CouchDB	D	MP	Eventual	None
Voldemort	D	MP	Linearizable, Eventual	None
HBase	D	MP	Linearizable	None
Riak	D	MP	Eventual	None
DocumentDB	D	MP	Eventual, Session, Bounded Staleness, Linearizability	None
Memcached	M & D	SM & MP	Linearizable	None
MongoDB	M & D	SM & MP	Linearizable	None
H-Store	M & D	MP	Linearizable	Serializable
ScyllaDB	M & D	MP	Linearizable, Eventual	None
Anna	**M & D**	**MP**	**Eventual, Causal, Item Cut, Writes Follow Reads Monotonic Reads/Writes, Read Your Writes, PRAM**	**Read Committed, Read Uncommitted**

TABLE I: Taxonomy of existing KVS systems. The scale column indicates whether a system is designed to run on a Single core (S), a multi-core machine (M), in a distributed setting (D), or a combination (M & D). The memory model column shows whether a system uses shared-memory model (SM), explicit message passing (MP), or both (SM & MP).

In terms of consistency, most distributed KVSs support a single, relaxed consistency level. COPS [24] and Bolt-on [25] guarantee causal consistency. MongoDB [4], HBase [26], and memcached [5] guarantee linearizable reads and writes against individual KVS objects. PNUTS [27] guarantees that writes are linearizable, and reads observe a monotonically increasing set of updates to key-value pairs.

Bayou [28] provides eventually consistent multi-key operations, and supports application-specific conflict detection and resolution mechanisms. Cassandra [3] and Dynamo [2] use quorum-based replication to provide different consistency levels. Applications can fix read and write quorum sizes to obtain either linearizable or eventually consistent single-key operations. In addition, both Cassandra and Dynamo use vector clocks to detect conflicting updates to a key, and permit application-specific conflict resolution policies. As noted in Figure I, Azure DocumentDB [29] supports multiple single-key consistency levels.

We note that the majority of distributed KVS systems do not provide any multi-key guarantees for arbitrary *groups* of keys. Some systems, such as HBase, provide limited support for transactions on single shard, but do not provide arbitrary multi-key guarantees. COPS and Bolt-on provide causally consistent replication. Bayou supports arbitrary multi-key operations but requires that each server maintains a full copy of the entire KVS. H-Store supports serializability by performing two-phase commit. However, achieving this level of consistency requires coordination and waiting amongst threads and machines, leading to limited scalability.

State machine replication [30] (SMR) is the de facto standard for maintaining strong consistency in replicated systems. SMR maintains consistency by enforcing that replicas deterministically process requests according to a total order (via a consensus protocol such as Paxos [31] or Raft [32]). Totally ordered request processing requires waiting for global consensus at each step, and thus fundamentally limits the throughput of each replica-set. Anna, in contrast, uses lattice composition to maintain the consistency of replicated state. Lattices are resilient to message re-ordering and duplication, allowing Anna to employ asynchronous multi-master replication without need for any waiting.

III. Lattices

A central component of the design of Anna is its use of lattice composition for storing and asynchronously merging state. Lattices prove important to Anna for two reasons.

First, lattices are insensitive to the order in which they merge updates. This means that they can guarantee consistency across replicas even if the actors managing those replicas receive updates in different orders. Section V describes Anna's use of lattices for multi-core and wide-area scalability in detail.

Second, we will see in Section VI that simple lattice building blocks can be composed to achieve a range of coordination-free consistency levels. The coordination-freedom of these levels was established in prior work [8], and while they cannot include the strongest forms of consistency such as *linearizability* or *serializability*, they include relatively strong levels including *causality* and *read-committed* transactions. Our contribution is architectural: Anna shows that these many consistency levels can all be *expressed* and implemented using a unified lattice-based foundation. Section VI describes these consistency levels and their implementation in detail.

To clarify terminology, we pause to review the lattice formalisms used in settings like convergent and commutative replicated data-types (CRDTs) [11], and the Bloom^L distributed programming language [12].

A *bounded join semilattice* consists of a domain S (the set of possible states), a binary operator $\sqcup$, and a "bottom" value $\bot$. The operator $\sqcup$ is called the "least upper bound" and satisfies the following properties:

Commutativity: $\sqcup(a, b) = \sqcup(b, a)\ \forall a, b \in S$
Associativity: $\sqcup(\sqcup(a, b), c) = \sqcup(a, \sqcup(b, c))\ \forall a, b, c \in S$
Idempotence: $\sqcup(a, a) = a\ \forall a \in S$

Together, we refer to these three properties via the acronym *ACI*. The $\sqcup$ operator induces a partial order between elements of S. For any two elements a, b in S, if $\sqcup(a, b) = b$, then we say that b's order is higher than a, i.e. $a \prec b$. The bottom value $\bot$ is defined such that $\forall a \in S$, $\sqcup(a, \bot) = a$; hence it is the smallest element in S. For brevity, in this paper we use "lattice" to refer to "bounded join semilattice" and "merge function" to refer to "least upper bound".

IV. Distributed state model

This section describes Anna's representation and management of state across actors. Each actor maintains state using lattices, but we observe that this is not sufficient to achieve high performance. As we discuss, the potential advantages of lattices can be lost in the high cost of synchronization in shared-memory key-value store architectures. Accordingly, Anna eschews shared-memory state model for one based on asynchronous message-passing.

A. Limitations of shared-memory

The vast majority of multi-core key-value stores are implemented as shared-memory systems, in which the entirety of the system's state is shared across the threads of a server: each thread is allowed to read or write any part of the state. Conflicting accesess to this state, at the level of reads and writes to memory words, need to be synchronized for correctness. Synchronization prevents concurrent writes from corrupting state, and ensures that reads do not observe the partial effects of in-progress writes. This synchronization typically occurs in the form of locks or lock-free algorithms, and is widely acknowledged as one of the biggest limiters of multi-core scalability. Both locks and lock-free algorithms can severely limit scalability under contention due to the overhead of cache-coherence protocols, which is proportional to the number of physical cores contending on a word in memory [7], [33]. For instance, even a single word in memory incremented via an atomic `fetch-and-add` can be a scalability bottleneck in multi-version database systems that assign transactions monotonically increasing timestamps [34].

Lattices do not change the above discussion; any shared-memory lattice implementation is subject to the same synchronization overheads. On receiving update client requests, actors must update a lattice via its merge function. Although these updates commute at the abstraction of the merge function, threads must synchronize their access to a lattice's in-memory state to avoid corrupting this in-memory state due to concurrent writes. Thus, while lattices' *ACI* properties potentially allow a system to scale regardless of workload, a shared-memory architecture fundamentally limits this potential due to the its reliance on multi-core synchronization mechanisms.

B. Message-passing

In contrast to using shared memory, a message-passing architecture consists of a collection of actors, each running on a separate CPU core. Each actor maintains private state that is inaccessible to other actors, and runs a tight loop in which it continuously processes client requests and inter-core messages from an input queue. Because an actor can update only its own local state, concurrent modification of shared memory locations is eliminated, which in turn eliminates the need for synchronization.

A message-passing system has two alternatives for managing each key; single-master and multi-master replication.

In single-master replication, each key is assigned to a single actor. This prevents concurrent modifications of the key's value, which in turn guarantees that it will always remain consistent. However, this limits the rate at which the key can be updated to the maximum update rate of a single actor.

In multi-master replication, a key is replicated on multiple actors, each of which can read and update its own local copy. To update a key's value, actors can either engage in coordination to control the global order of updates, or can leave updates uncoordinated. Coordination occurs on the critical path of every request, and achieves the effect of totally-ordered broadcast. Although multiple actors can process updates, totally ordered broadcast ensures that every actor processes the same set of updates in the same order, which is semantically equivalent to single-master replication. In a coordination-free approach, on the other hand, each actor can process a request locally without introducing any inter-actor communication on the critical path. Updates are periodically communicated to other actors when a timer is triggered or when the actor experiences a reduction in request load.

Unlike synchronous multi-master and single-master replication, a coordination-free multi-master scheme could lead to inconsistencies between replicas, because replicas may observe and process messages in different orders. This is where lattices come into play. Lattices avoid inconsistency and guarantee replica convergence via their *ACI* properties, which make them resilient to message reordering and duplication. Anna combines asynchronous multi-master replication with lattice-based state management to remain scalable across both low and high conflict workloads while still guaranteeing consistency.

V. Anna Architecture

Figure 1 illustrates Anna's architecture on a single server. Each Anna server consists of a collection of independent threads, each of which runs the coordination-free actor model. Each thread is pinned to a unique CPU core, and the number of threads never exceeds the number of available CPU cores. This 1:1 correspondence between threads and cores avoids the overhead of preemption due to oversubscription of CPU cores. Anna's actors share no key-value state; they employ consistent hashing to partition the key-space, and multi-master replication with a tunable replication factor to replicate data partitions across actors. Anna actors engage in epoch-based key exchange to propagate key updates at a given actor to other masters in the key's replication group. Each actor's private state is maintained in a lattice-based data-structure (Section VI), which guarantees that an actor's state remains consistent despite message delays, re-ordering, and duplication.

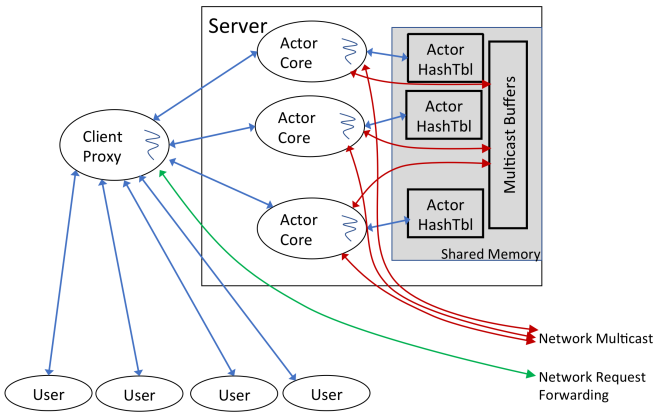


Fig. 1: Anna's architecture on a single server. Remote users are served by client proxies that balance load across servers and cores. Anna actors run thread-per-core with private hashtable state in shared RAM. Changesets are exchanged across threads by multicasting in memory; exchange across servers is done over the network with protobufs.

A. Anna actor event loop

We now discuss Anna's actor event loop and asynchronous multicast in more detail.

Each Anna actor repeatedly checks for incoming requests for puts and gets from client proxies, serves those requests, and appends results to a local *changeset*, which tracks the key-value pair updated within a period of time (the multicast epoch).

At the end of the multicast epoch, each Anna actor multicasts key updates in its changeset to relevant masters responsible for those keys, and clears the changeset. It also checks for incoming multicast messages from other actors, and merges the key-value updates from those messages into its local state. Note that the periodic multicast does not occur on the critical path of request handling.

Anna exploits the associativity of lattices to minimize communication via a *merge-at-sender* scheme. Consider a "hot" key k that receives a sequence of updates $\{u_1, u_2, ..., u_n\}$ in epoch t. Exchanging all these updates could be expensive in network and computation overhead. However, note that exchanging $\{u_1, u_2, ..., u_n\}$ is equivalent to exchanging just the single merged outcome of these updates, namely $\sqcup(... \sqcup (u_1, u_2), ...u_n)$. Formally, denote s as the state of key k on another replica, we have

$$\sqcup(... \sqcup (\sqcup(s, u_1), u_2), ...u_n) = \sqcup(s, \sqcup(... \sqcup (u_1, u_2), ...u_n))$$

by associativity. Hence batches of associative updates can be merged at a sending replica without affecting results; merging at the sender can dramatically reduce communication overhead for frequently-updated hot keys, and reduces the amount of computation performed on a receiving replica, which only processes the merged result of updates to a key, as opposed to every individual update.

VI. Flexible Consistency

As discussed in Section I, high performance KVSs can benefit a wide range of applications, each of which may vary in its consistency requirements. For example, Amazon's Dynamo shopping cart [2] focuses on supporting causally consistent single-key updates. On the other hand, applications that require multiple writes to succeed atomically need transactional support like *read committed* isolation [8].

Recent research has found that a wide array of consistency levels, including *causal consistency* and *read committed*, can be implemented in a coordination-free fashion [8]. A common requirement for coordination-free consistency levels is convergence: replicas of the same items should converge when they process the same set of messages, regardless of the order in which these messages arrive. This can be achieved by handling client requests and gossip in a way that is ACI (Associative, Commutative, Idempotent).

These properties are attractive, but they are far from trivial to achieve in general-purpose programs. Writing a large system to be ACI – and guaranteeing its correctness – is a difficult challenge.

In this section, we describe how Anna leverages ACI *composition* across small components to achieve a rich set of consistency guarantees—a modular software design pattern derived from the Bloom language [12]. Using ACI composition, we were able for the first time to easily build the full range of coordination-free consistency models [8] from the literature in a single KVS.

A. ACI Building Blocks

Proposals for ACI systems go back decades, to long-running transaction proposals like Sagas [35], and have recurred in the literature frequently. An ongoing question of the ACI literature was how programmers could achieve and enforce ACI properties in practice. For the Bloom language, Conway et al. proposed the composition of simple lattice-based (ACI) building blocks like counters, maps and pairs, and showed that complex distributed systems could be constructed with ACI properties checkable by induction [12]. Anna adopts Bloom's lattice composition approach. This bottom-up composition has two major advantages: First, in order to verify that a system is ACI, it is sufficient to verify that each of its simple building blocks is a valid lattice (has ACI properties), and the composition logic is ACI—this is more reliable than directly verifying ACI for a complex data structure. Second, lattice composition results in modular system design, which allows us to easily figure out which component needs to be modified when maintaining or updating the system.

B. Anna Lattice Composition

Anna is built using C++ and makes use of C++'s template structures to offer a flexible hierarchy of lattice types. As shown in Listing 1, the main data member in an `Anna` instance is represented as a C++ template of type `MapLattice`, which is a hash map parameterized by an immutable key type `K`, and a value type `L` that descends from `Lattice`. Any descendant of `Lattice` must implement a `merge` method that is ACI.

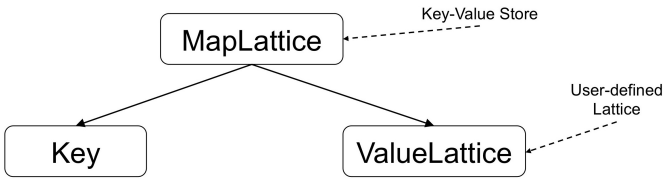


Fig. 2: A general template for ahieving coordination-free consistency

```
template <typename K, typename L>
class Anna {
 protected:
  MapLattice<K, L> kvs;
 public:
  V get(const K& k)
  {
    return kvs.reveal(k);
  }

  void put(const K& k, const L& l)
  {
    return kvs.merge(k, l);
  }
};
```

Listing 1: Anna C++ Template

Users' GET requests are handled via the `MapLattice.reveal` method, which returns the current values associated with the requested keys. PUT requests are handled via the `MapLattice.merge` method, which merges the new key-value pairs into the `MapLattice`. If an input's key does not exist in the hash map, Anna simply stores the new key-value pair into the hash map. Otherwise, the values associated with the key are merged using the merge function of lattice type `L`.

This design allows for a wide range of ACI, coordination-free objects to be stored in Anna. The design of those object classes determine the consistency model that is provided. Figure 2 sketches a general template for achieving this coordination-free consistency. In the style of existing systems such as Cassandra and Bayou, programmers can embed application-specific conflict resolution logic into the merge function of an Anna ValueLattice. Anna gives the programmer the freedom to program their ValueLattices in this ad hoc style, and in these cases guarantees only replica convergence. We define this level of ad hoc consistency as *simple eventual consistency*.

C. Consistency via Lattices: Examples

One of Anna's goals is to relieve developers of the burden of ensuring that their application-specific merge functions have clean ACI semantics. To achieve this, we can compose ad hoc user-defined merge logic within simple but more principled

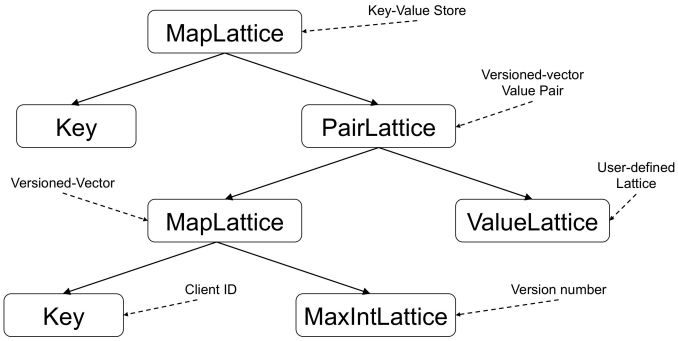


Fig. 3: Lattice composition for achieving *causal consistency*

lattices that maintain update metadata with ACI properties guaranteed by construction. In this section we demonstrate that a variety of well-known consistency levels can be achieved in this fashion. We begin by reviewing two popular consistency levels and demonstrating how Anna's modular design helps achieve their guarantees with minimal programming overhead.

1) Causal Consistency: Causal consistency keeps track of the causal relationship between different versions of the same object. Under *causal consistency*, if a user Alice updates a record, and the update is observed by a user Bob, then Bob's later update to the same record will overwrite Alice's update (instead of invoking the record's merge operator) since the two updates are causally related. However, if Bob updates the record without observing Alice's update, then there is no causal relationship between their updates, and the conflict will be resolved by invoking the record's merge operator.

Figure 3 shows Anna's lattice composition that supports causal consistency. Note that a vector clock can be implemented as a MapLattice whose keys are client proxy ids and values are version numbers associated with each proxy id. A version number can be implemented as a MaxIntLattice whose element is an integer and merge function takes the maximum between the input and its current element. Therefore, the integer associated with MaxIntLattice is always increasing, which can be used to represent the monotonically increasing version number. When the proxy performs a read-modify-write operation, it first retrieves the current vector clock, increments the version number corresponding to the proxy id, and writes the updated object together with the new vector clock to the server. The merge function of PairLattice works in lexicographic order on the pair; where the first element of the pair corresponds to a vector clock, and the second corresponds to the actual value lattice associated with a key. Given two PairLattices $P(a,b)$ and $Q(a,b)$, if $P.a \succ Q.a$, then $P(a,b)$ causally follows $Q(a,b)$, and the result is simply $P(a,b)$; the opposite is true if $Q.a \succ P.a$. However if $P.a$ and $Q.a$ are incomparable, then the two pairs correspond to concurrent writes, and the result is merged as $(P(a) \sqcup Q(a), P(b) \sqcup Q(b))$. The implementation of the merge function of PairLattice for achieving *causal consistency* in given in Listing 2.

As a simple example, consider a scenario where we have two clients (x, y) performing read-modify-write operations to Anna, whose ValueLattice has *set* as the element and *set union*

```
template <typename T>
class CausalPairLattice {
 protected:
  VersionValuePair<T> element;
 public:
  void merge(const VersionValuePair<T> &p)
  {
    // store the previous vector clock
    // before merging
    MapLattice<int, MaxLattice<int>> prev
      = this->element.vector_clock;
    // merge the current and
    // the input vector clocks
    this->element.vector_clock
                 .merge(p.vector_clock);
    if (this->element.vector_clock == prev)
    {
      // do nothing, as the new
      // vector clock is dominated
    }
    else if (this->element.vector_clock
                 == p.vector_clock)
    {
      // overwrite the current value with
      // the new one, as its vector clock
      // is dominated
      this->element.value.assign(p.value);
    }
    else
    {
      // merge the two values, as
      // the vector clocks are not
      // comparable
      this->element.value.merge(p.value);
    }
  }
};
```

Listing 2: Implementation of the merge function of PairLattice for achieving *causal consistency*

as the merge function. Initially, the value corresponding to key k is an empty set, with vector clock (x: 0, y: 0). Consider the following two cases. In the first case, x reads key k, retrieves the vector clock (x: 0, y: 0), and writes value $\{a\}$ with updated vector clock (x: 1, y: 0). After receiving the update, Anna determines that vector clock (x: 1, y: 0) dominates (x: 0, y: 0), and therefore overwrites the empty set with $\{a\}$. Then, y reads k, retrieves the vector clock (x: 1, y: 0), and writes value $\{b\}$ with updated vector clock (x: 1, y: 1). Anna determines that vector clock (x: 1, y: 1) dominates (x: 1, y: 0), and therefore overwrites $\{a\}$ with $\{b\}$. In the second case, x and y simultaneously read key k, retrieve the vector clock (x: 0, y: 0), and write back $\{a\}$ and $\{b\}$ with updated vector clock (x: 1, y: 0) and (x: 0, y: 1). Suppose x's update arrives first. As in the previous case, Anna updates the value of k to $\{a\}$ and sets its vector clock to (x: 1, y: 0). However, when y's update

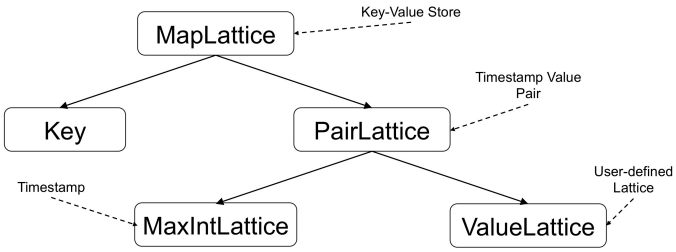


Fig. 4: Lattice composition for achieving *read committed*

arrives, Anna determines that (x: 1, y: 0) and (x: 0, y: 1) are incomparable, and therefore invokes the merge function (*set union*) to resolve conflicts. The resulting value is then set to $\{a, b\}$, with vector clock (x: 1, y: 1).

2) Read Committed: *Read committed* is a widely used isolation level in transactional databases [36]. Anna employs the coordination-free definition of *read committed* introduced in [8]. Here, consistency is discussed at the granularity of transactions, consisting of a sequence of reads and writes to the KVS. *Read committed* prevents both dirty writes and dirty reads, and ensures atomicity of writes. In order to prevent dirty writes in a weakly consistent system, it is sufficient to ensure that writes to each key exhibit a total ordering with respect to transactions. Although different replicas may receive writes in different orders, the final state of the KVS should be equivalent to the result of a serial execution of transaction writes. This can be achieved by appending a timestamp to each transaction (and to each write within the transaction) and applying a "larger timestamp wins" conflict resolution policy at each replica. Note that this monotonically increasing timestamp can be easily implemented using a MaxIntLattice.

To prevent dirty reads, we buffer all writes of a transaction at the client proxy until commit time, ensuring that uncommitted writes never appear in the KVS. To guarantee atomicity of writes, the client sends all writes in one batch to a single Anna actor, ensuring that either all writes reach the Anna server or none. The actor then distributes writes to other actors following the consistent hash ring. Figure 4 shows the lattice composition that supports *read committed* isolation level. The difference between the lattice composition for causal consistency and read committed is that we replace the MapLattice that represented growing vector clocks with a MaxIntLattice that represents transaction timestamps. The merge function of the new PairLattice compares the timestamp (MaxIntLattice) and modifies the ValueLattice to be the ValueLattice corresponding to the larger timestamp. If the timestamps are equal, then it implies that these writes are issued within the same transaction, and in this case the ValueLattice's merge logic is invoked[3]. The implementation of the merge function of PairLattice for achieving *read committed* isolation level is given in Listing 3.

Consider an example where we have two transactions T_1

[3]To support SQL's multiple sequential commands per transaction, we can replace these flat timestamps with a nested PairLattice of (transaction timestamp, command number), both being MaxIntLattices.

```
template <typename T>
class ReadCommittedPairLattice {
 protected:
  TimestampValuePair<T> element;
 public:
  void merge(const TimestampValuePair<T>& p)
  {
    if (p.timestamp >
        this->element.timestamp)
    {
      this->element.timestamp
                   .merge(p.timestamp);
      // overwrite the current value
      // with the new one, as its
      // timestamp is smaller
      this->element.value = p.value;
    }
    else if (p.timestamp ==
             this->element.timestamp)
    {
      // merge the two values, as
      // their timestamps are equal
      this->element.value
                   .merge(p.value);
    }
  }
};
```

Listing 3: Implementation of the merge function of PairLattice for achieving *read committed*

and T_2, with timestamp 1 and 2 respectively. T_1 performs the following sequence of operations: $\{w_1[k_1], w_1[k_2], r_1[k_3]\}$, and T_2 performs $\{w_2[k_1], w_2[k_2], r_2[k_4]\}$. Under *read committed*, T_1 and T_2 perform reads to Anna and buffer all writes locally. Both transactions issue the buffered write requests only after receiving the responses of the read requests and determining that the transactions are safe to commit.

Buffering writes on the client proxy prevents dirty reads. For example, if T_1 failed after $w_1[k1]$, this uncommitted write is not visible to other transactions since it is buffered at the proxy.

Anna avoids dirty writes by using transaction timestamps to consistently order writes. Consider a case where one replica of k_1 receives the writes in the order $\{w_1[k_1], w_2[k_1]\}$, and another replica in the order $\{w_2[k_1], w_1[k_1]\}$. However, the value of both replica converge to $w_2[k1]$, as T_2's write has a larger timestamp, and therefore dominates T_1's write, $w_1[k1]$. Multi-key writes are also eventually consistently ordered via the above timestamp precedence mechanism.

D. More Kinds of Consistency

Anna's modular design allows us to easily identify which component needs to be changed as we switch from simple eventual consistency to other consistency levels. This prevents the CACE (Changing Anything Changes Everything) phenomenon commonly observed in systems with monolithic

Type of Consistency	Lattice	Server	Client Proxy
Causal Consistency	20	12	22
Read Uncommitted	17	7	4
Read Committed	17	10	9
Item Cut Isolation	17	7	10
Monotonic Reads	17	7	4
Monotonic Writes	17	7	4
Writes Follow Reads	17	7	18
Read Your Writes	17	7	4
PRAM	17	7	4

Fig. 5: Lines of code modified per component across consistency levels.

design. To further demonstrate the flexibility of lattice composition, we modified Anna to support several other consistency levels including *read uncommitted*, *item-cut isolation*, and *read your writes* [8]. It turns out that the lattice composition for these consistency levels are the same as that of *read committed*.

Since *read uncommitted* does not require preventing dirty reads, we can easily achieve this level by disabling client-side buffering. Consider the same example in *read committed*, if T1 issues $w_1[k1]$ and then fails, it is possible for other transactions to observe the value v1 even if it is an uncommitted result that need to be rolled back.

Item cut isolation requires that if a transaction reads the same record more than once, it has to read the same value. To provide this guarantee, we buffer the record read at the client side, and when the transaction attempts to read the same record, it invokes the client-side cache instead of querying the server. Again, no modification to the lattice composition is required to achieve this requirement.

Read your writes is a session-based isolation level. Within a session, if a client reads a key after updating it, the read must either reflect the updated value or a value that overwrote the previously written value. Anna achieves this guarantee by attaching a unique timestamp to each client session and applying the same "larger timestamp wins" conflict resolution policy as before. The client also caches all the writes performed within the session. After it retrieves the value of a previously updated key, it merges the value with the cached value before returning the result. This way, Anna ensures that the value being read is at least as recent as the client's own update in terms of the timestamp.

Figure 5 shows the additional number of lines of code (loc) in C++ required on top of *simple eventual consistency* for each coordination-free consistency level. It is easy to conclude that extending Anna beyond *simple eventual consistency* incurs very little programming overhead.

VII. Implementation

The Anna actor and client proxy are implemented entirely in C++. The codebase—including the lattice library, all the consistency levels, the server code, and client proxy code—amounts to about 2000 lines of C++ on top of commonly-used libraries including ZeroMQ and Google Protocol Buffers. In the ensuing discussion, we refer the reader back to Figure 1.

A. Actor

To store the private KVS replica at each actor, we use the unordered map from the C++ standard library. Inter-actor multicast is achieved via the pub-sub communication mechanism of ZeroMQ, a high-performance asynchronous messaging library. To perform well across scales, we leverage ZeroMQ in different ways depending on whether we are communicating within or across machines. When two actors communicate within a single machine, the sender first moves the message into a shared memory buffer, and then sends the address of the buffer to the receiver using ZeroMQ's *inproc* transport, which is optimized for intra-process communication. The receiver, after getting the address, reads the shared buffer, updates its local KVS replica, and garbage collects the shared buffer. When two actors communicate across different machines, the sender first serializes the message into a byte-string using Google Protocol Buffers. It then sends the byte-string using ZeroMQ's *tcp* transport, which is designed for inter-node communication. After receiving the byte-string, the receiver first de-serializes the message, and then updates its KVS replica accordingly.

Anna uses consistent hashing to partition and replicate key-value pairs across actors. Following the design of Dynamo [2], each actor has a unique id, and Anna applies a CRC32 hash on the id to assign the actor to a position on the hash ring. It applies the same hash function to a key in order to determine the actors responsible for storing the key. Each key-value pair is replicated *N-1* times on the clockwise successor actors, where *N* is the user-provided replication factor.

Anna actors support three operations: GET, PUT, and DELETE. GET retrieves the value of a key from a (single) replica. Coordination-free consistency, as discussed in Section VI, does not require a quorum, so GET need not merge values from more than one replica. The GET response may be stale; the staleness is bounded by the multicast period, which is an adjustable parameter to balance performance and staleness. PUT persists the merge of a new value of a key with a (single) replica using the lattice merge logic. DELETE is implemented as a special PUT request with an empty value field. Actors free the heap memory of a key/value pair only when the DELETE's timestamp dominates the key's current timestamp. To completely free the memory for a key, each actor maintains a vector clock that keeps track of the latest-heard timestamps of all actors, which is kept up-to-date during multicast. Actors free the memory for a key only when the minimum timestamp within the vector-clock becomes greater than the DELETE's timestamp. After that time, because Anna uses ordered point-to-point network channels, we can be sure no old updates to the key will arrive. This technique extends naturally to consistency levels that require per-key vector-clocks (such as causal consistency) instead of timestamp. The difference is that before an actor frees a key, it asynchronously queries other replicas for the key's vector-clock to make sure they are no less than the DELETE's vector-clock.

B. Client Proxy

Client proxies interact with actors to serve user requests. In addition to GET, PUT, and DELETE, proxies expose two special operations to the users for consistency levels that involve transactions: BEGIN_TRANSACTION and END_TRANSACTION. All operations that fall in between a pair of special operations belong to a single transaction. Transaction ID is uniquely generated by concatenating a unique actor sequence number with a local timestamp.

Specific data structures are required at the proxy to support certain advanced consistency levels; these data structures are only accessible in a single client-proxy thread. For read committed, we need to create a message buffer that stores all PUT requests from a single transaction. For item-cut isolation, we need to cache key-value pairs that have already been queried within the same transaction. Currently, both the message buffer and the cache are implemented with the unordered map from the C++ standard library.

Client-actor communication is implemented with Linux sockets and Protocol Buffers. The client proxy uses the same consistent hashing function to determine the set of actors that maintain replicas of a given key. For load balancing, requests are routed to a randomly chosen replica. In case of failure and network delay, the client proxy times out and retries the request on other actors.

C. Actor Joining and Departure

In order to achieve steady performance under load burst, actors can be dynamically added or removed from the Anna cluster without stalling the system. Anna handles actor joining and departure in a similar fashion as Dynamo [2] and the work in [37]. Note that a new actor can be spawned from within an existing node, or from a new node. When a new actor joins the cluster, it first broadcasts its id to all existing actors. Each existing actor, after receiving the id, updates its local copy of the consistent hash ring and determines the set of key-value pairs that should be managed by the new actor. It then sends these key-value pairs to the new actor and deletes them from its local KVS replica. If the pre-existing actor receives queries involving keys that it is no longer responsible for, it redirects these requests to the new actor. After the new actor receives key-value pairs from all existing actors, it multicasts its id to all client proxies. Upon receiving the id, client proxies update the consistent hash ring so that relevant requests can be routed to the new actor.

When an actor is chosen to leave the cluster, it first determines the set of key-value pairs every other actor should be responsible for due to its departure. It then sends them to other actors along with its intention to leave the cluster. Other actors ingest the key-value pairs and remove the leaving actor from the consistent hash ring. The leaving actor then broadcasts to all client proxies to let them update the consistent hash ring and retry relevant requests to proper actors.

VIII. Evaluation

In this section, we experimentally justify Anna's design decisions on a wide variety of deployments. First we evaluate

Anna's ability to exploit parallelism on a multi-core server and quantify the merit of our Coordination-free actor model. Second, we demonstrate Anna's ability to scale incrementally under load burst. We then compare Anna against state-of-the-art KVS systems on both a single multi-core machine and a large distributed deployment. Finally, we show that the consistency levels from Section VI all provide high performance.

A. Coordination-free Actor Model

1) Exploiting Multicore Parallelism: Recall that under the Coordination-free actor model, each actor thread maintains a private copy of any shared state, and asynchronously multicasts the state to other replicas. This section demonstrates that the Coordination-free actor model can achieve orders of magnitude better performance than a conventional shared-memory architecture on a multi-core server.

To establish comparison against the shared-memory implementation, we built a minimalist multi-threaded key-value store using the concurrent hash map from the Intel Thread Building Blocks (TBB) library [38]. TBB is an open source library consisting of latch-free, concurrent data structures, and is among the most efficient libraries for writing scalable shared-memory software. We also benchmark Anna against Masstree, another shared-memory key-value store that exploits multi-core parallelism [18]. Finally, we implemented a multi-threaded key-value store using the C++ unordered map *without* any thread synchronization such as latching or atomic instructions. Note that this key-value store is not even thread-safe: torn writes could occur when multiple threads concurrently update the same key. It reflects the ideal performance one can get for any shared-memory KVS implementation like Masstree, TBB, etc.

Our experiments run on Amazon m4.16xlarge instances. Each instance is equipped with 32 CPU cores. Our experiments utilize a single table with 1M key-value pairs. Keys and values are 8 bytes and 1KB in length, respectively. Each request operates on a single key-value pair. Requests are update-only to focus on potential slowdowns from conflicts, and we use zipfian distributions with varying coefficients to generate workloads with different levels of conflict.

In our first experiment, we compare the throughput of Anna against the TBB hash map, Masstree, and the unsynchronized KVS (labeled as "Ideal") on a single multi-core machine. We measure the throughput of each system while varying the number of threads available. We pin each thread to a unique CPU core, and increase thread count up to the hardware limit of 32 CPU cores. In addition to measuring throughput, we use Linux's `perf` profiler to obtain a component-wise breakdown of CPU time. To measure the server's full capacity, requests are pre-generated based on the workload distribution at each thread. Since Anna is flexible about data placement policy, we experiment with different replication factors, from pure partitioning (like Redis Cluster or MICA) to full replication (a la Bayou) to partial replication (like ScyllaDB). As a baseline, Anna employs simple eventual consistency, and threads are set to multicast every 100 milliseconds. We use the same consistency level and multicast rate in all subsequent experiments unless otherwise stated. For each thread count, we repeat the experiment $10\times$ and plot the average throughput.

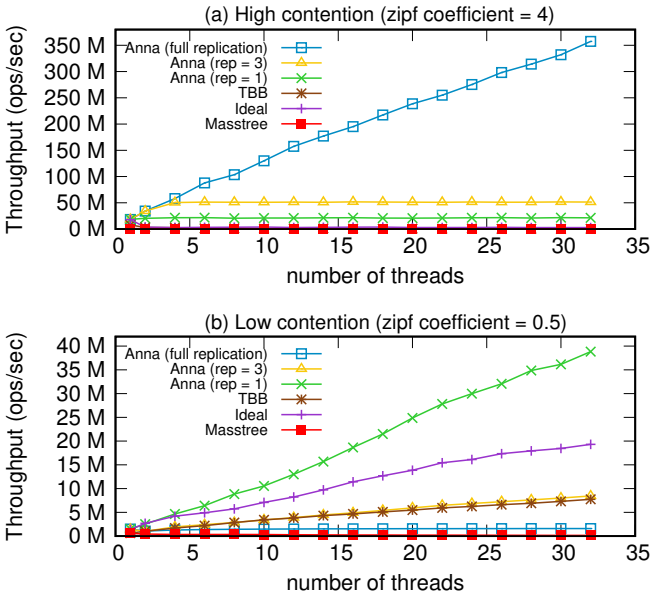


Fig. 6: Anna's single-node throughput across thread counts.

KVS	RH	AI	LM	M	O	CM
Anna (full)	90%	0%	4%	4%	2%	1.1
Anna (rep=3)	91%	0%	5%	2%	2%	1
Anna (rep=1)	94%	0%	5%	0%	1%	1
Ideal	97%	0%	0%	0%	3%	17
TBB	4%	95%	0%	0%	1%	19
Masstree	7%	92%	0%	0%	1%	16

(a) High Contention

KVS	RH	AI	LM	M	O	MF
Anna (full)	3%	0%	3%	92%	2%	32
Anna (rep=3)	25%	0%	4%	69%	2%	3
Anna (rep=1)	93%	0%	5%	0%	2%	1
Ideal	97%	0%	0%	0%	3%	32
TBB	70%	26%	0%	0%	4%	32
Masstree	20%	78%	0%	0%	2%	32

(b) Low Contention

Fig. 7: Performance breakdown for different KVSs under both contention levels when using 32 threads. CPU time is split into 5 categories: Request handling (RH), Atomic instruction (AI), Lattice merge (LM), Multicast (M), and others (O). The number of L1 cache misses (CM) for the high-contention workload and the memory footprint (MF) for the low-contention workload relative to Anna (rep=1) are shown on the right-most column.

Figure 6a and 7a show the result of the high-contention experiment, with zipf coefficient set to 4. We observe that both the TBB hashmap and Masstree fail to exploit parallelism on this workload because most requests perform an update against the same key, and concurrent updates to this key have to be serialized. Furthermore, both the TBB hashmap and Masstree must employ synchronization to prevent a single key-value pair from concurrent modification by multiple threads.

Synchronization overhead is proportional to the number of contending threads, which causes those systems' performance to plateau as we increase the number of threads in the system. Synchronization cost manifests as cache coherence overhead on multi-core hardware [39]. Figure 7a shows that TBB and Masstree spend 92% - 95% of the CPU time on atomic instructions under high contention, and only 4% - 7% of the CPU time is devoted to request handling. As a result, the TBB hash map and Masstree perform $50\times$ slower than Anna (rep = 1) and $700\times$ slower than Anna (full replication).

The unsynchronized store performs $6\times$ faster than the TBB hashmap and Masstree but still much slower than Anna. Although it does not use any synchronization to prevent threads from concurrently modifying the same key-value pairs, it suffers from cache coherence overhead resulting from threads modifying the same memory addresses (the contended key-value pairs). This is corroborated in Figure 7a, which shows that although both Anna and the unsynchronized store spend the majority of the CPU time processing requests, the unsynchronized store incurs $17\times$ more cache misses than Anna.

In contrast, threads in Anna perform updates against their local state in parallel without synchronizing, and periodically exchange state via multicast. Although the performance is roughly bounded by the replication factor under high contention, it is already far better than the shared-memory implementation across the majority of replication factors. Figure 7a indicates that Anna indeed achieves wait-free execution: the vast majority of CPU time (90%) is spent processing requests without many cache misses, while overheads of lattice merge and multicast are small. In short, Anna's Coordination-free actor model addresses the heart of the scalability limitations of multi-core KVS systems.

Figure 6b and 7b show the result of the low-contention experiment, with zipf coefficient 0.5. Unlike the high contention workload, all data are likely to be accessed with this contention level. Anna (rep=1) achieves excellent scalability due to its small memory footprint (data is partitioned across threads). However, despite the linear-scaling of Anna (rep=3), its absolute throughput is $4\times$ slower than Anna (rep=1). There are two reasons that have led to this performance degradation. First, increasing the replication factor increases the thread's memory footprint. Furthermore, under low contention, the number of distinct keys being updated within the gossip period increases significantly. Therefore, we can no longer exploit merge-at-sender to reduce the gossip overhead. Figure 7b shows that 69% of the CPU time is devoted to processing gossip for Anna (rep=3). Following this analysis, Anna (full replication) does not scale because any update performed at one thread will eventually be gossiped to every other thread, and therefore the performance is equivalent to serially executing requests with one thread. Although TBB and Masstree do not incur gossip overhead, they suffer from larger memory footprint and high cost of (conservative) synchronization operations as shown by our profiler measurements in Figure 7b. The lesson learned from this experiment is that for systems that support multi-master replication, having a high replication factor under low contention workloads can hurt performance. Instead, we want to dynamically monitor the data's contention level and selectively replicate the highly contented keys across threads. We come back to this subject in Section IX.

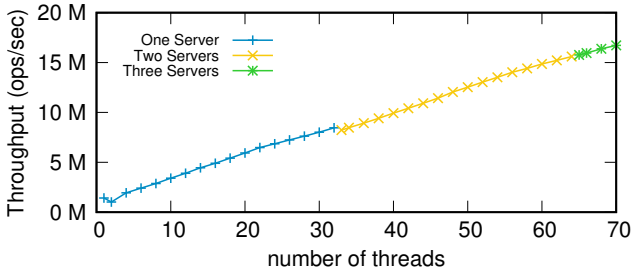


Fig. 8: Anna's throughput while incrementally adding threads to multiple servers.

2) Scaling Across Scales: This section demonstrates Anna's ability to scale smoothly from a single-node deployment to a multi-node deployment. Anna's replication factor is set to 3, and we use the low contention workload from the multi-core scalability evaluation in Section VIII-A1. We measure throughput while varying the number of available threads. The first 32 threads reside on a single node. The next 32 threads reside on a second node, while any remaining threads (at thread count greater than 64) reside on a third node.

Figure 8 shows that Anna exhibits smooth linear scaling with increasing thread count, on both a single node (32 or fewer threads) and multiple nodes (33 or more threads). We observe a small drop in performance as we add a 33^{rd} thread because this is the first thread that resides on the second node, and therefore triggers distributed multicast across the network. We do not observe a similar drop in performance as we add threads on the third node (at 65 threads) because the overhead of distributed multicast already affects configurations with thread counts between 33 and 64. Figure 8 illustrates that Anna is able to achieve near-linear scalability across different scales with the Coordination-free actor model.

B. Elastic Scalability

This section explores how well Anna's architecture achieves elastic scaling under load bursts. Our goal in this study is not to compare thread allocation *policies* per se, but rather to evaluate whether the Coordination-free actor model enables fine-grained elasticity. Hence we focus on Anna's reaction time, assuming an omniscient policy.

The experiment runs a read-modify-write, low contention YCSB workload [40], and uses 25 byte key, 1KB value records in a single table of 1M records. We perform our experiment on Amazon EC2 m4.x16 large instances. Anna is configured with replication factor 3. Note that the performance characteristics of experiments performed in this subsection and the next (VIII-C) differ from previous experiments. In earlier experiments, the goal was to evaluate Anna's maximum processing capacity when handling concurrent update requests; as a result, requests were update-only and pre-generated on actor threads to avoid request overhead due to network. Here, the goal is to evaluate how Anna performs in a more real-world setting, so requests are chosen to have a mix of reads

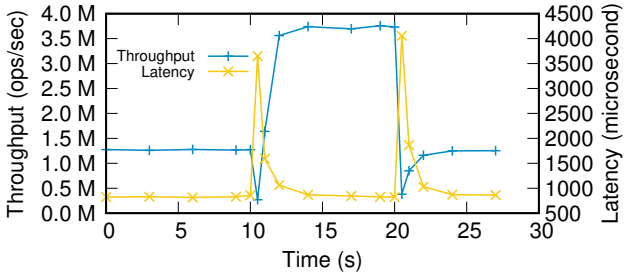


Fig. 9: Anna's ability to elastically scale under load burst while maintaining performance.

and writes, and are being sent from client proxies on other nodes. Therefore in this section we expect to observe network overhead; the effects are further discussed in Section VIII-C.

At the beginning of our experiment, we use one EC2 instance with 32 threads as the server and a sufficient number of client proxy instances to saturate the server. At the 10-second mark, we triple the load from the client proxies to create a burst. At the same time, 64 more threads from two server nodes are added to the Anna cluster. At the 20-second mark, we reduce the load back to the original and remove 64 threads from the cluster. Throughout the YCSB benchmark, we monitor Anna's throughput and the request latency.

As shown in Figure 9, Anna's throughput increases by 2× at the 10-second mark when we add in 64 additional threads, and drops to the original throughput at the 20-second mark when we remove the same number of threads. Throughout the experiment, the request latency stays roughly the same. The brief latency spikes at the 10-second mark and the 20-second mark are due to adding and removing nodes to the cluster.

C. Comparison with Popular Systems

This section compares Anna against widely-deployed, state-of-the-art key value stores. We perform two experiments; the first compares Anna against Redis [6] on a single node, and the second compares Anna against Cassandra [3] on a large distributed deployment. Both experiments run YCSB, and use the same configuration as in Section VIII-B with different contention levels.

1) Single node multi-core experiment: This section compares Anna with Redis on a single multi-core server. While Anna can exploit multi-core parallelism, Redis is a single-threaded KVS system, and cannot exploit any parallelism whatsoever. We therefore additionally compare Anna against Redis Cluster, which knits together multiple independent Redis instances, each of which contain a shard of the KVS.

In this experiment, we use a single EC2 instance as a server and enough client proxy instances to saturate the server. The Redis Cluster baseline runs an independent Redis instance on each available server thread.

Figure 10a compares each system under high contention while varying thread count. As in our earlier high contention experiments, clients pick keys with a zipfian coefficient of 4. Under high contention, each Redis Cluster's instances are subject to skewed utilization, which limits overall throughput.

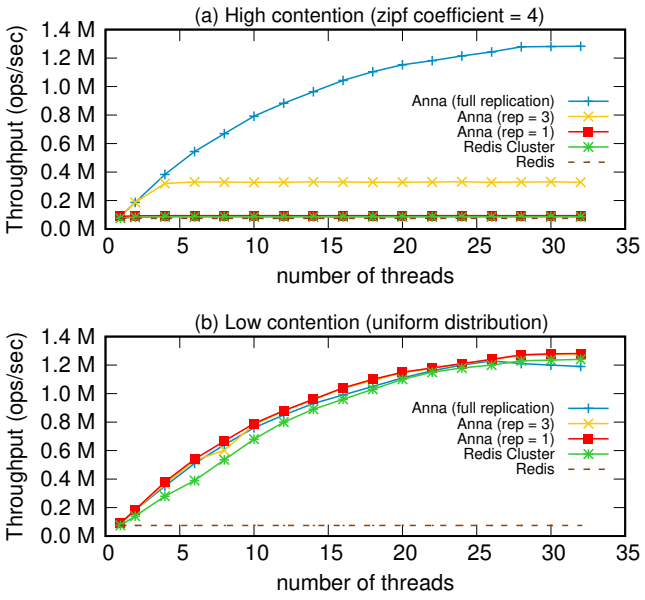


Fig. 10: Throughput comparison between Anna and Redis on a single node.

In contrast, Anna can spread load for hot keys across replicas. When the replication factor is greater than 1, Anna's throughput increases until the number of threads is slightly larger than the replication factor and then plateaus. If the hot keys are fully replicated, we observe that the throughput continues to grow as we increase the number of threads.

Figure 10b shows the result of the low contention experiment. As expected, Redis' throughput remains constant with increasing thread count. In contrast, both Anna and Redis Cluster can exploit multi-core parallelism, and their throughputs scale with increasing thread count. Interestingly, Anna (rep=3) and Anna (full replication) scale quite nicely, and the performance penalty due to gossip is far less significant compared to the result in Section VIII-A. The reason is that when the network is involved, the majority of overhead goes to network packet handling and message serialization and deserialization. Within a single node, gossip is performed using the shared memory buffer, and does not incur network overhead. Therefore, the overhead becomes far less significant. Experiments in this section show that Anna can significantly outperform Redis Cluster by replicating hot keys under high contention, and can match the performance of Redis Cluster under low contention.

Note that unlike the experiments in Section VIII-A, we do not observe linear scalability for Anna and the y axis has reduced by orders of magnitude. This is in keeping with earlier studies [41], [42], which demonstrate that this is due to message overheads: at a request length of 1KB we cannot expect to generate much more than 10Gbps of bandwidth due to message overheads. We attempted to improve the performance by varying the request size and batching the requests. Although these techniques did improve the absolute throughput, the scalability trend remained the same, and we continued to be bottlenecked by the network.

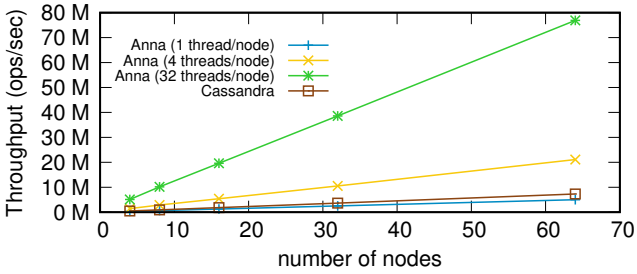


Fig. 11: Anna vs Cassandra, distributed throughput.

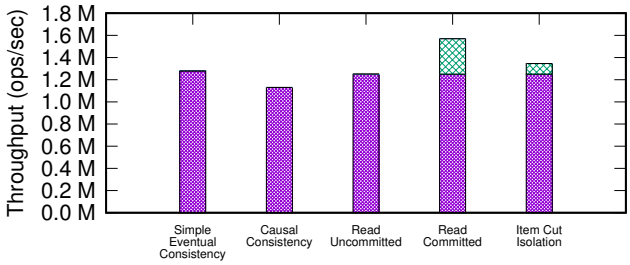


Fig. 12: Performance Across Consistency Levels

2) Distributed experiment: In a distributed setting, we compare Anna against Cassandra, one of the most popular distributed KVS systems [3]. To ensure that Cassandra achieves the best possible performance, we configure it to use its weakest consistency level (ONE), which only requires that an update is reflected on a single node before returning success. Updates are asynchronously propagated in the background.

We deployed Cassandra and Anna across four EC2 geographical regions (Oregon, North Virginia, Ireland, and Tokyo) and measured their scalability by adjusting the number of nodes per region. The replication factor of both Cassandra and Anna are set to 3. As in the multi-core experiment, each server node is a m4.x16large instance and we use multiple client instances to saturate the server. Clients pick keys to update from a uniform distribution.

Figure 11 shows that both Anna and Cassandra scale near-linearly as we increase the number of nodes. However, Anna has better absolute performance due to its low-overhead single-threaded execution. Indeed, when we varied the number of threads available to Anna, we found that Anna could significantly outperform Cassandra with just four threads per node (even though Cassandra used multi-threading). When permitted to use all 32 available cores, Anna outperformed Cassandra by $\sim 10\times$. This experiment demonstrates the importance of Anna's fast single-node mechanisms; even when a system can scale to large clusters, fast single-node mechanisms can make significantly more efficient use of available resources.

D. Performance Across Consistency Levels

Having implemented various consistency levels, we study the performance implications of the additional codepath for more advanced consistency levels. Anna is configured to use all 32 available cores, and the replication factor is set to 3. We use the low contention requests from Section VIII-C1. For transaction-based consistency levels, we group every six operations into one transaction at the YCSB client side. Figure 12 shows the throughput evaluation across different consistency levels. In general, we observe that the overhead incurred by these advanced consistency levels is not significant. As explained in Section VI-C, client-side buffering and caching requirement sometimes lead to *higher* throughput for Anna, which we show using green bars in Figure 12.

For *causal consistency*, we observe a slight degradation in throughput as Anna has to maintain the vector clock associated with each key-value pair, requiring more sophisticated lattice merge logic. In addition, the size of the vector clock for a given key is proportional to the number of client proxies that access the key. Therefore, periodic garbage collection is required to reduce the size of the vector clock. Similar throughput degradation is observed for *read uncommitted* due to the management of timestamps. For *read committed*, throughput increases because the client is required to buffer all write requests and send them as a single batch at the end of a transaction, which amortizes the number of round-trips between the server and the client. For *item cut isolation*, we also observe an increase in throughput because repeated reads to the same record are handled by a client-side cache (which again saves a round-trip between the server and the client). The throughput improvement gained from client-side buffering and caching is highlighted in green. Note that although other consistency levels does not require client-side buffering or caching, it is possible to use these techniques to improve throughput.

IX. Conclusion and Future Work

Conventional wisdom says that software designed for one scale point needs to be rewritten when scaling up by $10-100\times$ [1]. In this work, we took a different approach, exploring how a system could be architected to scale across many orders of magnitude by design. That goal led us to some challenging design constraints. Interestingly, those constraints led us in the direction of simplicity rather than complexity: they caused us to choose general mechanisms (background key exchange, lattice compositions) that work well across scale points. Perhaps the primary lesson of this work is that our scalability goals led us by necessity to good software engineering discipline.

The lattice composition model at the heart of Anna was critical to both performance and expressivity. The asynchronous merging afforded by lattices enabled wait-free performance; the lattice properties provided a conceptual framework for ensuring consistency; the *composition* of simple lattices enabled a breadth of consistency levels. Scale-independence might seem to be in conflict with richly expressive consistency. The lattice composition model resolved that design conflict.

We have further ambitions for Anna in future work, building on its fine-grained elasticity and flexible replication. In particular, we want to focus on per-key elasticity, increasing replication factors by key to deal with transient "hot keys". The ideas have a long history (e.g. [43]–[45]); we suspect there may be more to do in our setting. We also want to look at

designs that take advantage of deeper storage tiers for further cost savings.

REFERENCES

[1] J. Dean, "Challenges in building large-scale information retrieval systems," in *WSDM*, 2009.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP*, 2007.

[3] "Apache Cassandra," https://cassandra.apache.org/.

[4] "MongoDB," https://www.mongodb.com.

[5] "Memcached," https://www.memcached.org.

[6] "Redis," http://redis.io/.

[7] J. M. Faleiro and D. J. Abadi, "Latch-free synchronization in database systems: Silver bullet or fool's gold?" in *CIDR*, 2017.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, pp. 181–192, Nov. 2013.

[9] C. J. Clark, "Courtship dives of anna's hummingbird offer insights into flight performance limits," *Proceedings of the Royal Society of London B: Biological Sciences*, p. rspb20090508, 2009.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *SOSP*, 2009.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011, pp. 386–400.

[12] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier, "Logic and lattices for distributed programming," in *SoCC*, 2012.

[13] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, vol. 3. Stanford Research Institute, 1973, p. 235.

[14] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a calm and collected approach." in *CIDR*, 2011.

[15] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *SOSP*, 2001.

[16] V. Shah and M. V. Salles, "Reactors: A case for predictable, virtualized oltp actor database systems," *arXiv preprint arXiv:1701.05397*, 2017.

[17] P. A. Bernstein, M. Dashti, T. Kiefer, and D. Maier, "Indexing in an actor-oriented database." in *CIDR*, 2017.

[18] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012.

[19] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in *ICDE*, 2013.

[20] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors," *PVLDB*, vol. 4, no. 11, 2011.

[21] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: a holistic approach to fast in-memory key-value storage," in *NSDI*, 2014.

[22] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[23] "Seastar / Scylladb, or how we implemented a 10-times faster Cassandra," https://goo.gl/E7cxGW.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *SOSP*, 2011.

[25] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *SIGMOD*, 2013.

[26] "HBase," https://hbase.apache.org.

[27] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, 2008.

[28] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *SOSP*, 1995.

[29] "Azure DocumentDB," https://azure.microsoft.com/en-us/services/documentdb/.

[30] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[31] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, 1998.

[32] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014.

[33] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *OLS*, 2012.

[34] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *PVLDB*, vol. 8, no. 11, 2015.

[35] H. Garcia-Molina and K. Salem, *Sagas*. ACM, 1987, vol. 16, no. 3.

[36] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *PVLDB*, vol. 8, no. 3, 2015.

[37] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 14, 2014, pp. 583–598.

[38] "Intel Thread Building Blocks," https://www.threadingbuildingblocks.org/.

[39] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core oltp under high contention," in *SIGMOD*, 2016.

[40] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[41] A. Kolesnikov and M. Kulas, "Load modeling and generation for ip-based networks: a unified approach and tool support," in *International GI/ITG Conference, MMB & DFT*. Springer, 2010, pp. 91–106.

[42] "10gb ethernet tests," http://zeromq.org/results:10gbe-tests/.

[43] G. Copeland, W. Alexander, E. Boughter, and T. Keller, "Data placement in Bubba," in *ACM SIGMOD Record*, vol. 17, no. 3. ACM, 1988, pp. 99–108.

[44] H.-I. Hsiao and D. J. DeWitt, *Chained declustering: A new availability strategy for multiprocessor database machines*. University of Wisconsin-Madison, Computer Sciences Department, 1989.

[45] H. T. Vo, C. Chen, and B. C. Ooi, "Towards elastic transactional cloud storage with range query support," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 506–514, 2010.

[46] D. G. Feitelson, "Workload modeling for performance evaluation," in *Performance Evaluation of Complex Systems: Techniques and Tools*, M. C. Calzarossa and S. Tucci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 114–141.

[47] G. Zipf, *Human behavior and the principle of least effort: an introduction to human ecology*. Addison-Wesley Press, 1949. [Online]. Available: https://books.google.com/books?id=1tx9AAAAIAAJ

[48] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Eliminating Boundaries in Cloud Storage with Anna," *ArXiv e-prints*, Aug. 2018.

APPENDIX

A. Workload Skew and Replication Strategy

To motivate our future work on selective replication, we review concrete numbers for the kind of skew that arises from typical workloads. Since many workloads are well-modeled by Zipfian distributions [46], [47], we first look at the key access statistics under various Zipfian workloads to see how many keys are hot, cold, or somewhere in between, and then discuss the implication on the optimal replication strategy.

We generate 1M requests drawn from Zipfian distributions with varying coefficients and record how many keys are accessed above certain thresholds. As in the experiments in Section VIII-A, the database contains 1M key-value pairs.

Figure 13a shows that under low contention (zipf coefficient 0.5), the majority of the keys are accessed very infrequently; only 25 keys are accessed more than 100 times. Even with moderate contention (zipf coefficient 0.8), only 24 keys are accessed more than 1K times and 1 key is accessed more than 10K times out of 1M requests. Therefore, in these cases, there is no need to replicate keys across actors within a machine to boost performance.

zipf / access≥	0.5	0.6	0.7	0.8
1	557938	516168	461883	391841
10	3468	6071	8035	9309
100	25	100	256	463
1K	0	2	9	24
10K	0	0	0	1
100K	0	0	0	0

(a) Zipfian 0.5 - 0.8

zipf / access≥	1.0	1.5	2.0	3.0	4.0
1	217752	13286	1440	129	34
10	7739	1207	255	47	18
100	698	246	77	20	10
1K	70	53	24	9	5
10K	6	11	7	4	3
100K	0	2	2	2	1

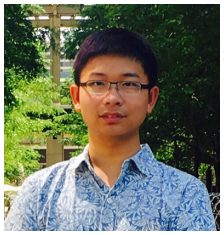
(b) Zipfian 1.0 - 4.0

Fig. 13: Statistics on the number of keys whose access frequency exceeds certain thresholds after processing 1M requests drawn from Zipfian distributions with varying coefficients.

However, as we increase the contention level, access to keys become more concentrated. In Figure 13b, we observe that with zipf coefficient 1.5, all accesses are on 1% (13286/1M) of the keys, and 2 keys get accessed more than 100K times. Under coefficient 4, only 34 keys are ever accessed among the 1M requests, with one key being extremely hot. Therefore, replication in these workloads does help increase performance, as access to highly contended keys can be spread across multiple replicas. Hence, the decision of whether or not to replicate keys depends on the contention level of the workload.

Moreover, the granularity of replication under high contention is crucial. Anna currently employs a single replication factor across all keys, which could be inefficient. For example, in the extreme case where the workload exhibits very high contention (zipf coefficient 4.0), Anna will replicate the entire database across all actors to attempt to maximize performance. However, as shown in Figure 13b, the number of keys being accessed is only on the order of 10s and only 5 them are hot. Therefore, aggressively replicating the entire database leads to 200K× extra storage overhead, and at the same time increases unnecessary multicast overhead for cold keys.

The observation above shows that static deployment and course-grained replication can lead to significant inefficiency. The solution is to monitor the workload contention and *dynamically* adjust the replication strategy. Moreover, *per-key* replication is required to better trade-off between performance and storage resources. This is a topic of ongoing work [48].



Chenggang Wu is a Ph.D. student at UC Berkeley working with Professor Joseph M. Hellerstein. His research interest lies in data-centric systems and distributed systems. He obtained his B.S. degree in computer science from Brown University in 2015.

Jose M. Faleiro is a post-doc in computer science at UC Berkeley. He was previously a Ph.D. student in the computer science department at Yale University. He is broadly interested in data management systems, multi-core systems, and distributed systems.

Yihan Lin is a second-year master student at Columbia University, consolidating her EECS background through her undergraduate education at Xian Jiaotong University and UC Berkeley. Her interests include a wide variety of topics at the intersection of computer science and engineering, distributed system, cloud computing, and data-driven analysis.

Joseph M. Hellerstein is the Jim Gray Professor of Computer Science at the University of California, Berkeley. He is an ACM Fellow, an Alfred P. Sloan Research Fellow and the recipient of three ACM-SIGMOD "Test of Time" awards. Hellerstein is the founding Editor-in-Chief of Foundations and Trends in Database Systems and has served on steering committees for ACM SIGMOD and ACM SOCC. He is a technical advisor at DellEMC, SurveyMonkey and various startup companies. In 2012, Hellerstein co-founded Trifacta, Inc., where he currently serves as Chief Strategy Officer.