The VLDB Journal
https://doi.org/10.1007/s00778-020-00632-7

SPECIAL ISSUE PAPER

Autoscaling tiered cloud storage in Anna

Chenggang Wu[1] · **Vikram Sreekanti**[1] · **Joseph M. Hellerstein**[1]

Received: 1 February 2020 / Revised: 17 August 2020 / Accepted: 26 August 2020

Abstract
In this paper, we describe how we extended a distributed key-value store called Anna into an autoscaling, multi-tier service for the cloud. In its extended form, Anna is designed to overcome the narrow cost–performance limitations typical of current cloud storage systems. We describe three key aspects of Anna's new design: multi-master selective replication of hot keys, a vertical tiering of storage layers with different cost–performance trade-offs, and horizontal elasticity of each tier to add and remove nodes in response to load dynamics. Anna's policy engine uses these mechanisms to balance service-level objectives around cost, latency, and fault tolerance. Experimental results explore the behavior of Anna's mechanisms and policy, exhibiting orders of magnitude efficiency improvements over both commodity cloud KVS services and research systems.

Keywords Autoscaling · Key-value store · Cloud storage system · Data replication · Cost efficiency

1 Introduction

As public infrastructure cloud providers have matured in the last decade, the number of storage services they offer has soared. Popular cloud providers like Amazon Web Services (AWS) [9], Microsoft Azure [10], and Google Cloud Platform (GCP) [24] each have at least seven storage options. These services span the spectrum of cost–performance trade-offs: AWS ElastiCache, for example, is an expensive, memory-speed service, while AWS Glacier is extremely high latency and low cost. In between, there are a variety of services such as the Elastic Block Store (EBS), the Elastic File System (EFS), and the Simple Storage Service (S3). Azure and GCP both offer a similar range of storage solutions.

Each one of these services is tuned to a unique point in that design space, making it well suited to certain performance goals. Application developers, however, typically deal with a non-uniform *distribution* of performance requirements. For example, many applications generate a skewed access distribution, in which some data is "hot" while other data is "cold." This is why traditional storage is assembled hierarchically: hot data is kept in fast, expensive cache while cold data is kept in slow, cheap storage. These access distributions have become more complex in modern settings, because they can change dramatically over time. Realistic workloads spike by orders of magnitude, and hot sets shift and resize. These large-scale variations in workload motivate an autoscaling service design, but most cloud storage services today are unable to respond to these dynamics.

The narrow performance goals of cloud storage services result in poor cost–performance trade-offs for applications. To improve performance, developers often take matters into their own hands by addressing storage limitations in custom application logic. This introduces significant complexity and increases the likelihood of application-level errors. Developers are inhibited by two key types of barriers when building applications with non-uniform workload distributions:

Cost-Performance Barriers. Each of the systems discussed above—ElastiCache, EBS, S3, etc.—offers a different, *fixed* trade-off of cost, capacity, latency, and bandwidth. These trade-offs echo traditional memory hierarchies built from RAM, flash, and magnetic disk arrays. To balance performance and cost, data should ideally move adaptively across storage tiers, matching workload skew and shifting hotspots. However, current cloud services are largely unaware of each other, so software developers and DevOps engineers must cobble together ad hoc memory hierarchies. Applications must explicitly move and track data and requests across storage systems in their business logic. This task is further complicated by the heterogeneity of storage services in terms of deployment, APIs, and consistency guarantees. For example, single-replica systems like ElastiCache are linearizable,

✉ Chenggang Wu
cgwu@berkeley.edu

[1] University of California, Berkeley, 465 Soda Hall, Berkeley 94720, CA, USA

Published online: 09 September 2020

while replicated systems like DynamoDB are eventually consistent.

Static Deployment Barriers. Cloud providers offer very few truly autoscaling storage services; most such systems have hard constraints on the number and type of nodes deployed. In AWS for example, high-performance tiers like ElastiCache are surprisingly inelastic, requiring system administrators to allocate and deallocate instances manually. Two of the lower storage tiers—S3 and DynamoDB—are autoscaling, but are insufficient for many needs. S3 autoscales to match data volume but ignores workload; it is designed for "cold" storage, offering good bandwidth but high latency. DynamoDB offers workload-based autoscaling but is prohibitively expensive to scale to a memory-speed service. This motivates the use of ElastiCache over DynamoDB, which again requires an administrator to monitor load and usage statistics, and manually adjust resource allocation.

In an earlier paper, we presented the initial architecture of a key-value storage system called Anna [60]. That paper described a design with excellent performance across orders of magnitude in scale. Here, we extend Anna to remove its cost–performance and static deployment barriers, enabling it to dynamically adjust configuration and match resources to workloads. While our previous work's evaluation focused on raw performance, this paper also emphasizes *efficiency*: the ratio of performance to cost. For various cost points, Anna outperforms in-memory systems (e.g., AWS ElastiCache, Masstree [40]) by up to an order of magnitude. Anna also outperforms DynamoDB, an elastic database, by more than two orders of magnitude in efficiency.

In Sect. 2, we briefly describe the contributions of our prior work on Anna [60] and preview our approach to making the system adapt across barriers. In Sect. 3, we describe the mechanisms that Anna uses to respond to mixed and changing workloads. Sect. 4 introduces the architecture of Anna including the implementation of these mechanisms, and Sect. 5 describes Anna's policy engine. Sect. 6 introduces Anna's API. In Sect. 7, we present an evaluation of Anna's mechanisms and policies, and we describe how they fare in comparison to the state of the art. Sect. 8 discusses related work, and we conclude with future work in Sect 9.

In the remainder of this paper, we use AWS as the public cloud provider underlying Anna. The design principles and lessons learned here are naturally transferable to other cloud providers with similar offerings.

2 Background

The first paper on Anna [60] presented a distributed key-value store based on a fully shared-nothing, thread-per-core architecture with background gossip across cores and nodes. Anna threads have no shared data structures beyond message queues, enabling each core to spend most of its time doing useful work. Experiments on high-contention workloads showed Anna spending over 90% of its compute cycles serving user requests, while competing, state-of-the-art systems were achieving less than 10%. The vast majority of the other systems' time was spent trying to execute atomic processor instructions on shared data structures. As a result, Anna outperformed the competition by orders of magnitude at many scale points. Relative to other distributed KVSs, Anna's design also enabled an unprecedented richness of coordination-free consistency levels. The basic insight was that the full variety of coordination-free consistency and transactional isolation levels taxonomized by Bailis et al. [11] can be achieved by the monotone composition of simple lattice structures, as suggested by Conway et al. [16]. The original paper maps out a wide range of key-value and NoSQL systems against Bailis' taxonomy of consistency levels.

The first version of Anna focused on performing well at both single-node and distributed scales. This showed that eventual consistency combined with a coordination-free shared nothing architecture makes data management easy in the face of deployment changes and also hinted at the potential to remove deployment barriers. However, the initial architecture lacked the mechanisms to monitor and respond to usage and workloads. Another notable weakness of the initial work was its need to aggressively replicate the entire database across the main memory of many machines to achieve high performance. This gave the system an unattractive cost–performance trade-off and made its resource allocation very rigid. As a result, although a benchmark beater, Anna's first version suffered from the problems highlighted above: it was expensive and inflexible for large datasets with non-uniform access distributions.

2.1 Removing barriers with anna

In this paper, we extend the initial version of Anna to span the cost–performance design space more flexibly, enabling it to adapt dynamically to workload variation in a cloud-native setting. The architecture presented here removes the cost–performance and static deployment barriers discussed in Sect. 1. To that end, we add three key mechanisms: (1) horizontal elasticity to adaptively scale deployments; (2) vertical data movement in a storage hierarchy to reduce cost by demoting cold data to cheap storage; and (3) multi-master selective replication of hot keys across nodes and cores to efficiently scale request handling for non-uniform access patterns. The architecture we present here is simplified by deploying the same storage kernel across many tiers, by entirely avoiding coordination, and by keeping most components stateless through reuse of the storage engine. The

additions to Anna described in this work enable system operators to specify high-level goals such as fault tolerance and cost–performance objectives, without needing to manually configure the number of nodes and the replication factors of keys. A new policy engine automatically responds to workload shifts using the mechanisms mentioned above to meet these service-level objectives (SLOs).

Table 1 The mechanisms used by Anna to deal with various aspects of workload distributions

Workload dynamics	Relevant mechanisms
Volume	Elasticity
Skew	Replication, Tiering
Hotspot	Replication, Tiering

3 Distributions and mechanisms

In this section, we first classify and describe common workload distributions across data and time. We then discuss the mechanisms that Anna uses to respond to the workload properties and changes. We believe that an ideal cloud storage service should gracefully adapt to three aspects of workload distributions and their dynamics in time:

A. Volume. As overall workload grows, the aggregate throughput of the system must grow. During growth periods, the system should automatically increase resource allocation and thereby cost. When workload decreases, resource usage and cost should decrease correspondingly as well.

B. Skewness. Even at a fixed volume, skewness of access distributions can affect performance dramatically. A highly skewed workload will make many requests to a small subset of keys. A uniform workload of similar volume will make a few requests to each key. Different skews warrant different responses, to ensure that the resources devoted to serving each key are proportional to its popularity.

C. Shifting Hotspots. Workloads that are static in both skew and volume can still exhibit changes in distribution over time: hot data may become cold and vice versa. The system must be able to not only handle skew, but also changes in the specific keys associated with the skew (hotspots) and respond accordingly by prioritizing data in the new hot set and demoting data in the old one.

We address these three workload variations with three mechanisms in Anna, which we describe next.

Horizontal Elasticity. In order to adapt to variation in workload volume, each storage tier in Anna must scale elastically and independently, in terms of both storage and request handling. Anna needs the storage capacity of many nodes to store large amounts of data, and it needs the compute and networking capacity of many nodes to serve large numbers of requests. This is accomplished by partitioning (sharing) data across all the nodes in a given tier. When workload volume increases, Anna can respond by automatically adding nodes and repartitioning a subset of data. When the volume decreases, Anna can remove nodes and repartition data among the remainders.

Multi-Master Selective Replication. When workloads are highly skewed, simply adding shards to the system will not alleviate pressure. The small hot set will be concentrated on a few nodes that will be receiving a large majority of the requests, while the remaining nodes lie idle. The only solution is to replicate the hot set onto many machines. However, we do not want to repeat the mistakes of our first iteration of Anna's design, replicating cold keys as well—this simply wastes space and increases overhead. Instead, replication must be selective, with hot keys replicated more than cold keys. Thus, Anna must accurately track which data is hot and which is cold, and the replication factors and current replica locations for each key. Note that this aids both in handling skew in general and also changes in hotspots with fixed skew.

Vertical Tiering. As in a traditional memory hierarchy, hot data should reside in a fast, memory-speed tier for efficient access; significant cost savings are available by demoting data that is *not* frequently accessed to cold storage. Again, Anna must correctly classify hot and cold data in order to promote or demote appropriately to handle skew and hotspots. While the previous two mechanisms are aimed at improving performance, this one primarily attempts to minimize cost without compromising performance.

3.1 Summary

Table 1 shows which mechanisms respond to which properties of workload distributions. There is a direct mapping between an increase (or decrease) in volume—with other factors held constant—and a requirement to automatically add (or remove) nodes. Changes in workload skew require a response to the new hot set size via promotion or demotion, as well as appropriate selective replication. Similarly, a change in hotspot location requires correct promotion and demotion across tiers, in addition to shifts in per-key replication factors. We describe how Anna implements each one of these mechanisms in Sects. 4 and 5 . In Sect. 7, we evaluate how well Anna responds to these dynamics.

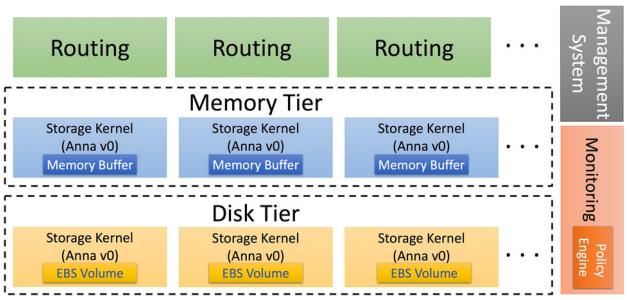

Fig. 1 The Anna architecture

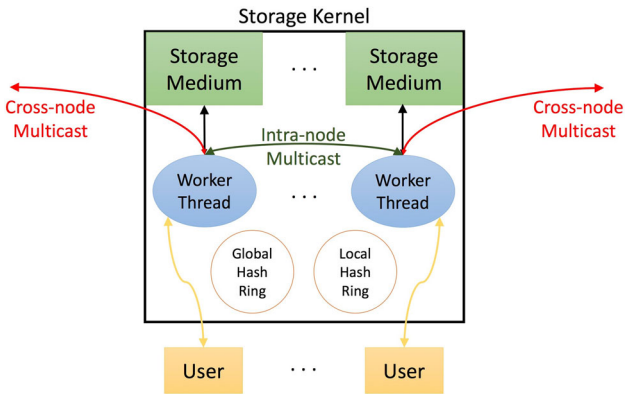

Fig. 2 The architecture of storage kernel

4 Anna architecture

In this section, we introduce Anna's architecture and illustrate how the mechanisms discussed in Sect. 3 are implemented. We present an overview of the core subsystems and then discuss each component in turn. As mentioned in Sect. 1, Anna is built on AWS components. In our initial implementation and evaluation, we validate this architecture over two storage tiers: one providing RAM cost–performance and another providing flash disk cost–performance. Anna's memory tier stores data in RAM attached to AWS EC2 nodes. The flash tier leverages the Elastic Block Store (EBS), a fault-tolerant block storage service that masquerades as a mounted disk volume on an EC2 node. There is nothing intrinsic in our choice of layers. We could easily add a third layer (e.g., S3) and a fourth (e.g., Glacier), but demoting data to cold storage in these tiers operates on much longer timescales that are beyond the scope of this work.

4.1 Overview

Figure 1 presents an overview of Anna, with each rectangle representing a node. In the original Anna paper [60], we described an extremely performant, coordination-free key-value store with a rich variety of consistency levels. In that work, we demonstrated how a KVS could scale from multicore to distributed settings while gracefully tolerating the natural messaging delays that arise in distributed systems. To enable the mechanisms described in Sect. 3, we first extended the storage kernel (labeled as Anna v0 in Fig. 1) to support multiple storage media and then designed three new subsystems: a monitoring system/policy engine, a routing service, and a cluster management system. Each subsystem is bootstrapped on top of the key-value storage component in Anna, storing and modifying its metadata in the system.

The monitoring system and policy engine are the internal services responsible for responding to workload dynamics and meeting SLOs. Importantly, these services are stateless and thus are not concerned with fault tolerance and scaling; they rely on the storage service for these features.

The routing service is a stateless client-facing API that provides a stable abstraction above the internal dynamics of the system. The resource allocation of each tier may be in flux—and whole tiers may be added or removed at workload extremes—but clients are isolated from these changes. The routing service consistently returns a correct endpoint that will answer client requests. Finally, the cluster management system is another stateless service that modifies resource allocation based on decisions reached by the policy engine.

4.2 Storage system

Figure 2 shows the architecture of Anna's storage kernel. The kernel contains many worker threads, and each thread interacts with a thread-local storage medium (a memory buffer or disk volume), processes client requests, and sends & receives multicasts to & from other Anna workers.

The Anna storage kernel is deployed across many storage tiers. The only difference between tiers is the procedure for translating data for persistence (serialization/deserialization, a.k.a. "serde"). Memory-tier nodes read from and write to local memory buffers, while disk-tier nodes serialize data into files that are stored on EBS volumes. Anna's uniformity across storage tiers makes adding additional tiers very simple: we set the serde mode and adjust the number of worker threads based on the underlying hardware. For instance, the total number of threads for memory nodes matches the number of CPU cores to fully utilize computing resources and to avoid costly preemption of threads. However, in other storage tiers where the performance bottleneck lies in serializing the key-value pairs to and from persistent storage, the optimal strategy for resource allocation is different. Our EBS tier allocates 4× as many threads per node (4) as we have physical CPU core (1).

Anna uses consistent hashing [28] to partition and replicate keys. For performance and fault tolerance (discussed further in Sects. 5 and 7), each key may be replicated onto many nodes in each tier and multiple threads in each node.

Following the model of early distributed hash tables, we use virtual nodes [44] in our consistent hashing algorithm. Each physical node (or thread) handles traffic for many virtual nodes (or threads) on the hash ring to ensure an even distribution. Virtual nodes also enable us to add heterogeneous nodes in the future by allocating more virtual nodes to more powerful physical machines.

In the following section, we present a brief overview of the storage kernel's design, which enables it to achieve high-performance coordination-free execution and replica consistency. This overview is a brief summary of the initial design of Anna presented in [60].

4.2.1 Storage Kernel

Recent work has demonstrated that shared-memory coordination mechanisms like locking and atomic "lock-free" instructions slow down low-level memory access performance on a single node by orders of magnitude [22]. Across nodes, consensus algorithms such as Paxos [34] are well-known to cause dramatic latency and availability problems [1,12,13]. Anna's coordination-free execution model avoids these issues entirely in pursuit of excellent performance and scalability. Each worker thread on every node has a private memory buffer to store the data it manages. Data is multi-mastered: each thread processes both reads and writes locally regardless of replication. Each thread periodically runs a background task to multicast recent updates to other workers that maintain replicas of these keys ("gossip", a.k.a. "anti-entropy" [20]). This shared-nothing, asynchronous messaging scheme eliminates thread synchronization and resolves conflicting updates to replicas asynchronously. The resulting code exploits multi-core parallelism within a single machine and smoothly scales out across distributed nodes. Our earlier work [60] shows dramatic benefits from this design, including record performance based on extremely high (90%) CPU utilization in useful work with low processor cache miss rates.

While Anna eliminates contention, consistency becomes tricky: the same set of updates may arrive at different replicas in different orders. Naïvely applying these updates can cause replicas to diverge and lead to inconsistent state. Another contribution of [60] is achieving a wide range of consistency models by encapsulating state into monotone compositions of simple CRDT-style [48] lattices, inspired by the Bloom language [16].

A lattice consists of a domain S (the set of possible states), a "bottom" value $\bot$ that represents the initial state, and a "merge" operator $\sqcup$ that updates the state of the lattice. $\sqcup$ 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: $\sqcup(a, a) = a\ \forall a \in S$

We refer to these three properties as *ACI*. *ACI* is crucial for achieving eventual replica convergence; for different key replicas, although a set of "gossips" may be applied in different orders, *ACI* ensures that the state of replicas eventually converges. Hence, lattices tolerate message reordering and duplication.

Moreover, when a node crashes and restarts, the state of all key replicas on that node temporarily diverges (lags behind) from other replicas. However, by periodically exchanging the state amongst replicas, Anna allows the replicas that lag behind to "catch up" with others, and the *ACI* properties again ensure that these replicas converge to the same state.

By default, Anna stores data in last-writer-wins lattices, which resolve divergent updates by picking the update with the most recent timestamp. However, Anna's lattices can be composed to offer the full range of coordination-free consistency guarantees including causal consistency, item cut isolation, and read-committed transactions [11].

4.3 Metadata management

Anna requires maintaining certain metadata to efficiently support mechanisms discussed in Sect. 3 and help the policy engine adapt to changing workloads. In this section, we introduce the types of metadata managed by Anna and how they are stored and used by various system components.

4.3.1 Types of metadata

Anna manages three distinct kinds of metadata. First, every storage tier has two *hash rings*. A global hash ring, G, determines which nodes in a tier are responsible for storing each key. A local hash ring, L, determines the set of worker threads *within* a single node that are responsible for a key.

Second, each individual key K has a *replication vector* of the form $[< R_1, ...R_n >, < T_1, ...T_n >]$. R_i represents the number of nodes in tier i storing key K, and T_i represents the number of threads *per node* in tier i storing key K. In our current implementation, i is either M (memory tier) or E (EBS tier). During request handling and multicast, both hash rings and key K's replication vector are used to determine the threads responsible for the key. For every tier, i, that maintains a replica of K, we first hash K against G_i, tier i's global hash ring to determine which nodes are responsible for K. We then look at L_i, tier i's local hash ring to determine which threads are responsible for the key.

Lastly, Anna tracks monitoring statistics, such as the access frequency of each key and the storage consumption of each node. This information is analyzed by the policy engine to trigger actions in response to variations in workload. Currently, we store 16 bytes of metadata per key and about 10 KB of metadata per worker thread.

4.3.2 Metadata storage

Clearly, the availability and consistency of metadata are as important as that of regular data—otherwise, Anna would be unable to determine a key's location (under changing node membership and keys' replication vectors) or get an accurate estimate of workload characteristics and resource usage. In many systems [32,49,52,57], metadata is enmeshed in the implementation of "master nodes" or stateful services like ZooKeeper [26]. Anna simply stores metadata in the storage system. Our metadata automatically derives all the benefits of our storage system, including performance guarantees, fault tolerance, and consistency. Anna employs last-writer-wins consistency to resolve conflicts among metadata replicas. Due to the eventual consistency model, worker threads may have stale views of hash rings and replication vectors. This can cause threads to disagree on the location of a key and can potentially cause multiple rounds of request redirection. However, since the metadata will eventually converge, threads will agree on the key's location, and requests will reach the correct destination. Note that multicast is performed every few seconds, while cluster state changes on the order of minutes, so cluster state metadata is guaranteed to converge before it undergoes further changes.

4.3.3 Enabling mechanisms

Interestingly, manipulating two of these types of metadata (hash rings and replication vectors) is the key to enabling the mechanisms described earlier in Sect. 3. In this section, we discuss *only* the implementation of each mechanism. When and why each action is executed is a matter of policy and will differ based on system configuration and workload characteristics—we save this discussion for Sect. 5.

Elasticity. Anna manages cluster churn similarly to previous storage systems [14,19] that use consistent hashing and distributed hash tables. When a new node joins a storage tier, it queries the storage system to retrieve the hash ring, updates the ring to include itself, and broadcasts its presence to all nodes in the system—storage, monitoring, and routing. Each existing node updates its copy of the hash ring, determines if it stores any keys that the new node is now responsible for, and gossips those keys to the new node. Similarly, when a node departs, it removes itself from the hash ring and broadcasts its departure to all nodes. It determines which nodes are now responsible for its data and gossips its keys to those nodes. Once all data has been broadcast, the node goes offline and its resources are deallocated.

Key migration overheads can be significant (see Sect. 7.4). To address this challenge, Anna interleaves key migration with client request handling to prevent system downtime. This is possible due to Anna's support for coordination-free consistency: The client may retrieve stale data during the key migration phase, but it can maintain a client-side cache and merge future retrieved results with the cached value. Anna's lattice-based conflict resolution guarantees that the state of the cached data is monotonically growing.

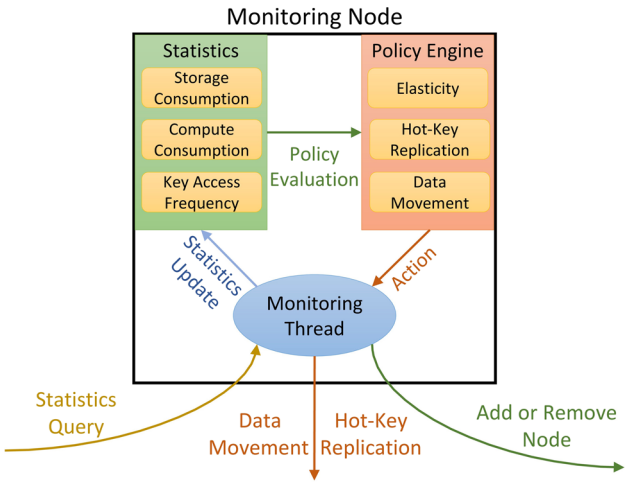

Fig. 3 Monitoring node architecture

Selective Replication & Cross-Tier Data Movement. Both these mechanisms are implemented via updates to replication vectors. Each key in our two-tier implementation has a default replication vector of the form $[< 1, k >, < 1, 1 >]$, meaning that it has one memory-tier replica and k EBS-tier replicas. Here, k is the number of replica faults per key the administrator is willing to tolerate (discussed further in Sect. 4.7 and 5). By default, keys are not replicated across threads within a single node. Anna induces cross-tier data movement by simply manipulating metadata. It increments the replication factor of one tier and decrements that of the other tier; as a result, gossip migrates data across tiers. Similarly, selective replication is achieved by adjusting the replication factor in each tier, under the fault tolerance constraint. After updating the replication vector, Anna updates metadata across replicas via asynchronous multicast.

4.4 Monitoring system and policy engine

In this section, we discuss the design of the monitoring system and the policy engine. As shown in Fig. 3, each monitoring node has a monitoring thread, a statistics buffer, and a policy engine. The monitoring thread is stateless and periodically retrieves the stored statistics from the storage engine and triggers the policy engine. Note that per-key statistics such as key access frequency are reported by each worker thread that maintains the key replica. The monitoring thread aggregates the per-key statistics across all replicas before sending them to the policy engine. The policy engine analyzes these statistics and issues actions to meet its SLOs. Anna currently supports three types of actions: *elasticity change*, *hot-key*

replication, and *cross-tier data movement*. The implementation of these actions is covered above in Sect. 4.3.3. We discuss when each of these actions is triggered and describe the end-to-end policy algorithm in Sect. 5.

4.5 Routing service

The routing service isolates clients from the underlying storage system: A client asks where to find a key and is returned the set of all valid addresses for that key. Anna's routing service only maintains soft state. Each routing node caches the storage tiers' hash rings and key replication vector metadata to respond to the clients' key address requests. If a key has any memory-tier replicas, the routing service only returns memory-tier addresses to maximize performance. The client caches these addresses locally to reduce request latency and load on the routing service.

When a client's cached address set becomes invalid because of a change in cluster configuration, a storage server receiving an invalid request will give the client the correct set of addresses. These will again be cached until they are invalidated, and the routing service will also refresh its cached cluster state.

4.6 Cluster management

Anna uses Kubernetes [30] as a cluster management tool. Kubernetes is responsible for allocating and deallocating nodes, ensuring that nodes are alive, and rebooting failed nodes. An Anna deployment has four kinds of nodes: storage nodes, routing nodes, monitoring nodes, and a single, stateless "cluster management" node described below.

A "pod" is the atomic unit of a Kubernetes application and is a collection of one or more Docker [21] containers. Each node in Anna is instantiated in a separate Kubernetes pod, and each pod contains only one instance of a Anna node. Storage system and routing service pods are pinned on separate EC2 instances for resource isolation purposes. The monitoring system is less resource intensive and can tolerate preemption, so it is not isolated. Finally, Anna maintains a singleton cluster management pod, whose role is to issue requests to add or remove nodes to the Kubernetes cluster. A simple, stateless Python server in this pod receives REST requests from the policy engine and uses bash scripts to add or remove nodes.

4.7 Fault tolerance

Anna guarantees k-fault tolerance by ensuring $k+1$ replicas are live at all times. The choice of k determines a trade-off between resilience and cost. The $k+1$ replicas of each key can be spread across tiers arbitrarily, according to hotness.

When a storage node fails, other nodes detect the failure via a timeout and remove the node from the hash ring. When such a timeout happens, Anna automatically repartitions data using the updated hash ring. The cluster management pod then issues a request to spawn a new node, which enters the join protocol discussed in Sect. 4.3.3.

Anna does not rely on the persistence of EBS volumes for fault tolerance in the disk tier. Similar to nodes in the memory tier, these nodes lose their state when they crash—this is desirable because it allows all tiers to be symmetric, regardless of the durability of the underlying storage medium.

Both routing nodes and monitoring nodes only store soft state and do not require any recovery mechanisms. If a routing node fails, it queries other routing nodes for up-to-date cluster information, and if a monitoring node fails, it retrieves system statistics from the storage service.

When the cluster management pod fails, Kubernetes automatically revives it. No recovery is necessary as it does not manage any state. The state of the cluster will not change while the pod is down since it is the actor responsible for modifying resource allocation. As a result, the policy engine will redetect any issue requiring an elasticity change before the crash and reissue the request upon revival.

In summary, Anna consists of a stateful storage kernel that is partitioned and selectively replicated for performance and fault tolerance with multi-master updates. Every other component is either stateless and optionally caches soft state that is easily recreated. As a result, the only single point of failure in Anna is the Kubernetes master. Kubernetes offers high-availability features to mitigate this problem [31]. We also note that Kubernetes is not integral to the design of Anna; we rely on it primarily to bootstrap the system and reduce the engineering burden of mundane tasks such as receiving heartbeats, allocating VMs, and deploying containers.

5 Policy engine

Anna supports three kinds of SLOs: an average request latency (L_{obj}) in milliseconds, a cost budget (B) in dollars/hour, and a fault tolerance (k) in number of replicas. The fault tolerance indicates the allowed number of replica failures, k. The latency objective, L_{obj}, is the average expected request latency. The budget, B, is the maximum cost per hour that will be spent on Anna.

As discussed in Sect. 4.7, Anna ensures there will never be fewer than $k+1$ replicas of each key to achieve the fault tolerance goal. The latency objective and cost budget goals, however, are conflicting. The cheapest configuration of Anna is to have $k+1$ EBS nodes and 1 memory node (for metadata). Clearly, this configuration will not be very performant. If we increase performance by adding memory nodes to the system, we might exceed our budget. Conversely, if we

strictly enforce the budget, we might not be able to achieve the latency objective.

Anna administrators only specify *one* of the two goals. If a latency SLO is specified, Anna minimizes cost while meeting the latency goal. If the budget is specified, Anna uses no more than $\$B$ per hour while maximizing performance.

In Sects. 5.1, 5.2, and 5.3 , we describe heuristics to trigger each policy action—data movement, hot-key replication, and elasticity. In Sect. 5.4, we present Anna's complete policy algorithm, which combines these heuristics to achieve the SLO. Throughout this section, we represent each key's replication vector as $[< R_M, R_E >, < T_M, T_E >]$ (a general form is defined in Sect 4.3.1) since our initial prototype only uses two tiers—M for memory and E for EBS.

5.1 Cross-tier data movement

Anna's policy engine uses its monitoring statistics to calculate how frequently each key was accessed in the past T seconds, where T is an internal parameter. If a key's access frequency exceeds a configurable threshold, P, and all replicas currently reside in the EBS tier, Anna promotes a single replica to the memory tier. If the key's access frequency falls below a separate internal threshold, D, and the key has one or more memory replicas, all replicas are demoted to the EBS tier. The EBS replication factor is set to $k + 1$, and the local replication factors are restored to 1. Note that in Anna, all metadata is stored in the memory tier, is never demoted, and has a constant replication factor. If the aggregate storage capacity of a tier is full, Anna adds nodes (Sect. 5.3) to increase capacity before performing data movement. If the budget does not allow for more nodes, Anna employs a least recently used caching policy to demote keys.

5.2 Hot-Key replication

When the access frequency of a key stored in the memory tier increases, hot-key replication increases the number of memory-tier replicas of that key. In our initial implementation, we configure only the memory tier to replicate hot keys. Because the EBS tier is not intended to be as performant, a hot key in that tier will first be promoted to the memory tier before being replicated. This policy will likely vary for a different storage hierarchy.

The policy engine classifies a key as "hot" if its access frequency exceeds an internal threshold, H, which is s standard deviations above the mean access frequency. Because Anna is a shared-nothing system, we can replicate hot keys both across cores in a single node and across nodes. Replicating across nodes seems preferable, because network ports are a typical bottleneck in distributed system, so replicating across nodes multiplies the aggregate network bandwidth to the key. However, replicating across cores within a node can also be beneficial, as we will see in Sect. 7.2. Therefore, hot keys are *first* replicated across more nodes before being replicated across threads within a node.

The policy engine computes the target replication factor, R_{M_ideal}, using the ratio between the observed latency for the key and the latency objective. Cross-node replication is only possible if the current number of memory replicas, R_M, is less than the number of memory-tier nodes in the cluster, N_M. If so, we increment the key's memory replication factor to $min(R_{M\_ideal}, N_M)$. Otherwise, we increment the key's *local* replication factor on memory-tier machines up to the maximum number of worker threads (N_{T_memory}) using the same ratio. Finally, if the access frequency of a previously hot key drops below a threshold, L, its replication vector is restored to the default: R_M, T_M, and T_E are all set to 1 and R_E is set to k.

5.3 Elasticity

Node Addition. Anna adds nodes when there is insufficient storage or compute capacity. When a tier has insufficient storage capacity, the policy engine computes the number of nodes required based on data size, subject to cost constraints, and instructs the cluster management service to allocate new nodes to that tier.

Node addition due to insufficient compute capacity only happens in the memory tier because the EBS tier is not designed for performance. Compute pressure on the EBS tier is alleviated by promoting data to the memory tier since a memory node can support 15× the requests at 4× the cost. The policy engine uses the ratio between the observed latency and the latency objective to compute the number of memory nodes to add. This ratio is bounded by a system parameter, c, to avoid overly aggressive allocation.

Node Removal. Anna requires a minimum of one memory node (for system metadata) and $k + 1$ EBS nodes (to meet the k-fault SLO when all data is demoted). The policy engine respects these lower bounds. We first check if any key's replication factor will exceed the total number of storage nodes in any tier after node removal. Those keys' replication factors are decremented to match the number of nodes at each tier before the nodes are removed. Anna currently only scales down the memory tier based on compute consumption and not based on storage consumption. This is because selective replication can significantly increase compute consumption without increasing storage consumption. Nonetheless, this may lead to wasteful spending under adversarial workloads; we elaborate in the next section.

Grace Periods. When resource allocation is modified, data is redistributed across each tier, briefly increasing request latency (see Sect. 7.4). Due to this increase, as well as data location changes, key access frequency decreases. To prevent overcorrection during key redistribution, we apply a grace

period to allow the system to stabilize. Key demotion, hot-key replication, and elasticity actions are all delayed till after the grace period.

5.4 End-to-end Policy

In this section, we discuss how Anna's policy engine combines the above heuristics to meet its SLOs. If the average storage consumption of *all* nodes in a particular tier has violated configurable upper or lower thresholds (S_{upper} and S_{lower}), nodes are added or removed, respectively. We then invoke the data movement heuristic from Sect. 5.1 to promote and demote data across tiers. Next, the policy engine checks the average latency reported by clients. If the latency exceeds a fraction, f_{upper} (defaulting to 0.75), of the latency SLO and the memory tier's compute consumption exceeds a threshold, C_{upper}, nodes are added to the memory tier. However, if not all nodes are occupied, hot keys are replicated in the memory tier, as per Sect. 5.2. Finally, if the observed latency is a fraction, f_{lower} (defaulting to 0.5), below the objective and the compute occupancy is below C_{lower}, we invoke the node removal heuristic to check if nodes can be removed to save cost.

The compute threshold, C_{upper}, is set to 0.20. Consistent with our previous work [60], each storage node saturates its network bandwidth well before its compute capacity. Compute occupancy is a proxy for the saturation of the underlying network connection. This threshold varies significantly based on the hardware configuration; we found that 20% was optimal for our experimental setup (see Sect. 7).

5.4.1 Discussion

Storage Node Saturation. There are two possible causes for saturation. If all nodes are busy processing client requests, Anna must add more nodes to alleviate the load. Performing hot-key replication is not productive: Since all nodes are busy, replicating hot keys to a busy node will, in fact, decrease performance due to additional gossip overhead. The other cause is a skewed access distribution in which most client requests are sent to a small set of nodes serving the hot keys, while most nodes are free. The optimal solution is to replicate the hot keys onto unsaturated nodes. If we add nodes to the cluster, the hot keys' replication factors will not change, and clients will continue to query the few nodes storing those keys. Meanwhile, the newly added nodes will idle. As discussed in Sect. 5.4, Anna's policy engine is able to differentiate the two causes for node saturation and take the appropriate action.

Policy Limitations. There are cases in which our policy engine fails to meet the latency objective and/or wastes money. Due to current cloud infrastructure limitations, for example, it takes about 5 min to allocate a new node. An adversary could easily abuse this limitation. A short workload spike to trigger elasticity, followed by an immediate decrease would lead Anna to allocate unnecessary nodes. These nodes will be underutilized, but will only be removed if the observed latency drops below $f_{lower} * L_{obj}$. Unfortunately, removing this constraint would make Anna susceptible to reducing resource allocation during network outages, which is also undesirable. We discuss potential solutions to these issues in future work.

Knobs. There are a small number of configuration variables mentioned in this section, which are summarized in Table 2. We distinguish variables that are part of the external SLO Spec from the internal parameters of our current policy. In our evaluation, our parameters were tuned by hand to match the characteristics of the AWS services we use. There has been interesting work recently on auto-tuning database system configuration knobs [55]; our setting has many fewer knobs than those systems. As an alternative to auto-tuning our current knobs, we are exploring the idea of replacing the current threshold-based policy entirely with a dynamic reinforcement learning policy that maps directly and dynamically from performance metrics to decisions about system configuration changes. These changes to the policy engine are easy to implement, but tuning the policy is beyond the scope of this paper: It involves extensive empirical work on multiple deployment configurations.

6 Anna API

As shown in Table 3, Anna exposes seven APIs to the application: `Get`, `Put`, `Delete`, `GetAll`, `PutAll`, `GetDelta`, and `Subscribe`. The first three APIs are straightforward and discussed in detail in [60]. Here, we introduce four new APIs that we added on top of [60].

GetAll. Unlike the `Get` API, which queries a single replica of a key, `GetAll` queries all replicas of a key, merges them on the client side, and returns the merged result to the user. This allows the user to observe the most up-to-date state of a key in Anna. As an extension, the Anna client can query a subset of the replicas, achieving a trade-off between performance and data freshness.

When `GetAll` is used in place of `Get` on an 8-byte key with a replication factor of 3, we observe no performance change for the median latency (0.58 ms) and a 6% performance degradation (from 0.68 to 0.72 ms) for the 99-th percentile latency. This is because the client needs to wait for responses from every node that stores a replica of the key before responding to the user, which impacts the tail latency. The node configuration of this micro-benchmark is the same as other experiments in Sect. 7.

PutAll. Anna's regular `Put` API updates a single replica of a key and relies on asynchronous gossip for the update to

Table 2 A summary of all variables mentioned in Sect. 5

Variable name	Meaning	Default value	Type
L_{obj}	Latency Objective	2.5ms	SLO Spec
B	Cost Budget	N/A (user-specified)	SLO Spec
k	Fault Tolerance	2	SLO Spec
T	Monitoring report period	15 s	Policy Knob
H	Key hotness threshold	3 standard deviations above the mean key access frequency	Policy Knob
L	Key coldness threshold	The mean key access frequency	Policy Knob
P	Key promotion threshold	2 accesses in 60 s	Policy Knob
$[S_{lower}, S_{upper}]$	Storage consumption thresholds	Memory: [0.3, 0.6] EBS: [0.5, 0.75]	Policy Knob
$[f_{lower}, f_{upper}]$	Latency thresholds	[0.5, 0.75]	Policy Knob
$[C_{lower}, C_{upper}]$	Compute occupancy thresholds	[0.05, 0.20]	Policy Knob
c	Upper bound for latency ratio	1.5	Policy Knob

Table 3 Summary of Anna's APIs

API	Description
`Get(key)->value`	Retrieves the value of `key` from a single replica
`Put(key, value)`	Performs an update to a single replica of `key`
`Delete(key)`	Deletes `key`
`GetAll(key)->value`	Retrieves the value of `key` from all replicas and returns the merged result
`PutAll(key, value)`	Performs an update to all replicas of `key`
`GetDelta(key, id)`	Retrieves the value of `key` only when it has changed from the previously queried version, identified by `id`
`Subscribe(key, address)`	Subscribes to `key` and receives the updated value at `address`

propagate to other replicas. However, if the node accepting the client update crashes before gossiping, the update will be lost. To avoid potential data loss, `PutAll` sends the updates to all replicas of a key and returns only when all replicas accept the update. As an extension, the Anna client can send the update to a subset of the replicas, achieving a trade-off between performance and fault tolerance.

When `PutAll` is used in place of `Put` on an 8-byte key with a replication factor of 3, we observe similar performance changes as in the comparison between `GetAll` and `Get`; the median latency increases from 0.58 to 0.59 ms and the 99-th percentile latency increases from 0.66 to 0.69 ms.

GetDelta. A client interacting with Anna may query the same key multiple times. `GetDelta` returns the value of the key only when its payload has changed since the previous query. This API takes two arguments: a key and a version identifier of the key from the previous query. `GetDelta` is currently supported under last-writer-wins consistency and causal consistency. When enabled, the client sends the version metadata (timestamp and vector clock, respectively) as part of the request for Anna to check if the payload has changed. In case the payload has not changed, Anna responds with just an acknowledgement. Since the payload is not shipped as part of the response, this optimization significantly reduces network overhead for workloads that involves large payload.

For keys with small payload, however, `GetDelta` may be less efficient than `Get`, as the overhead of shipping the version metadata becomes pronounced. Accordingly, the Anna client makes `GetDelta` request only when the payload is larger than the version identifier. Otherwise, it falls back to the regular `Get` protocol.

To support the `GetDelta` API, the Anna client maintains the following metadata for each previously queried key: the size of the payload and the key's version identifier. The size information is stored as an 8-byte integer. For last-writer-wins consistency, each key's timestamp is 8 bytes. For causal consistency, each key's vector clock typically ranges from 10 bytes to 1KB. The Anna client stores these identifiers in a buffer and uses LRU policy for eviction. When the previously queried key's version identifier is unavailable, Anna falls back to the regular `Get` protocol.

Subscribe. A client can subscribe to a key and get notified when the value of the key changes. `Subscribe` takes two arguments: a key to subscribe and a notification IP address. When the value of the key changes, Anna pushes the new value to the notification address. This API is suitable for applications that operate in a passive, event-driven mode. To support this API, for each key under subscription, Anna maintains a list of IPs that listen for the key updates.

7 Evaluation

In this section, we present an evaluation of Anna. We first explore the optimal instance type for Anna's memory-tier storage node that balances the CPU, memory, and network bandwidth (Sect. 7.1). This instance type is used throughout the experiments. We then explore the advantage of different replica placement strategies in Sect. 7.2. Next, we show the benefit of selective replication in Sect. 7.3. We demonstrate Anna's ability to detect and adapt to variation in workload volume, skew, and hotspots in Sects. 7.4 and 7.5 . Section 7.6 covers Anna's ability to respond to unexpected failures. Finally, Sect. 7.7 evaluates Anna's ability to trade off performance and cost according to its SLO.

Anna uses `r4.2xlarge` instances for memory-tier nodes and `r4.large` instances for EBS-tier nodes. Each node has 4 worker threads; at peak capacity they can handle a workload that saturates the network link of the node. `r4.2xlarge` memory nodes have 61GB of memory, which is equally divided among all worker threads. Each thread in a EBS node has access to its own 64GB EBS volume. In our experiments, Anna uses two `m4.large` instances for the routing nodes and one `m4.large` instance for the monitoring node. We include these nodes in all cost calculation below. Unless otherwise specified, all experiments are run on a database with 1 million key-value pairs. Keys and values are 8 bytes and 256KB long, respectively. We set the k-fault tolerance goal to $k = 2$; there are 3 total replicas of each key. This leads to a total dataset size of about 750GB: $1M\ keys \times 3\ replicas \times 256KB\ values$.

Our workload is a YCSB-style read–modify–write of a single key chosen from a Zipfian distribution. We adjust the Zipfian coefficient to create different contention levels—a higher coefficient means a more skewed workload. We use the regular `Get` and `Put` APIs in our experiments. The clients were run on `r4.16xlarge` machines, with 8 threads each. Client machines ran in the same AWS region (us-east-1) as the server machines. Unless stated otherwise, experiments used 40 client machines for a total of 320 concurrent, single-threaded clients.

7.1 Node configuration

Our first experiment explores the most cost-effective node configuration offered by the cloud vendor. Specifically, we explore which node types deliver the best performance relative to their cost on AWS for Anna's memory tier. We mainly focus on the `r4` instance family, as nodes from this instance family are well suited for high-performance in-memory databases.

For each instance type, we first adjust the number of nodes to match a fixed cost budget of $6.384 per hour. This corresponds to 3 `r4.8xlarge` nodes, 6 `r4.4xlarge` nodes, 12 `r4.2xlarge` nodes, 24 `r4.xlarge` nodes, and 48 `r4.large` nodes. We then record system's latency and throughput as we increase the workload volume (number of concurrent client threads) until the system is saturated. In this experiment, we set the Zipfian coefficient to 0.5. We see in Fig. 4 that performance characteristics vary significantly across node types. Interestingly, although larger instance types such as `m4.16xlarge`, `r4.8xlarge`, and `r4.4xlarge` have abundant CPU cores and memory, their peak throughput is relatively poor (less than 30K operations per second). According to Anna's monitoring system, their CPU utilization at peak throughput are all below 15% and performance is bottlenecked by the network bandwidth (10Gbps–25Gbps), consistent with the observations made in [60]. On the other hand, small instances such as `r4.large` and `r4.xlarge` deliver better peak throughput (60K operations per second). However, these instances are bottlenecked by the CPU, as we observe that their CPU utilization are above 95% at peak throughput.

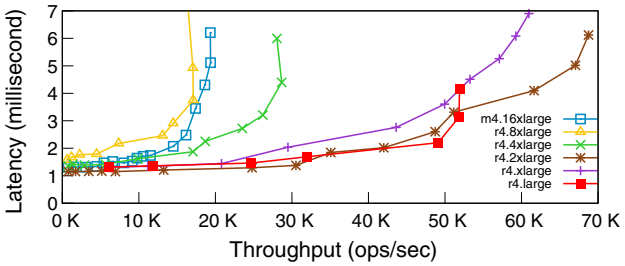

Fig. 4 Performance comparison across different node types. The cost across all node types is set to $6.384 per hour. This corresponds to 3 `r4.8xlarge` nodes, 6 `r4.4xlarge` nodes, 12 `r4.2xlarge` nodes, 24 `r4.xlarge` nodes, and 48 `r4.large` nodes. The Zipfian coefficient of the workload is set to 0.5

In comparison, `r4.2xlarge` instance achieves the best peak throughput (70K operations per second) with a CPU utilization of around 80%. Therefore, it has the best balance of CPU, memory, and network resources for our workload and we pick this instance for Anna's memory tier for the rest of the experiments.

7.2 Replica placement

In this section, we compare the benefits of intra-node vs. cross-node replication; for brevity, no charts are shown for this topic. On 12 memory-tier nodes, we run a highly skewed workload with the Zipfian coefficient set to 2. With a *single* replica per key, we observe a maximum throughput of just above 2000 operations per second (ops). In the case of cross-node replication, four *nodes* each have one thread responsible for each replicated key; in the intra-node case, we have only one node with four *threads* responsible for each key. Cross-node replication improves performance by a factor of four to

8000 ops, while intra-node replication only improves performance by a factor of two to 4000 ops. This is because the four threads on a single node all compete for the same network bandwidth, while the single threads on four separate nodes have access to four times the aggregate bandwidth. Hence, as discussed in Sect. 5.2, we prioritize cross-node replication over intra-node replication whenever possible but *also* take advantage of intra-node replication.

7.3 Selective replication

A key weakness of our initial work [60] (referred to as Anna v0) is that all keys are assigned a uniform replication factor. A poor choice of replication factor can lead to significant performance degradation. Increasing the replication factor boosts performance for skewed workloads, as requests to hot keys can be processed in parallel on different replicas. However, a uniform replication factor means that cold keys are *also* replicated, which increases gossip overhead (slowing down the system) and storage utilization (making the system more expensive). By contrast, Anna selectively replicates hot keys to achieve high performance, without paying a storage cost for replicating cold keys.

This experiment explores the benefits of selective replication by comparing Anna's memory tier against Anna v0, AWS ElastiCache (using managed Memcached), and a leading research system, Masstree [40], at various cost points. We hand-tune Anna v0's single replication factor to the optimal value for each Zipfian setting and each cost point. This experiment uses a database of 100,000 keys across all cost points; we use a smaller database since the data must fit on one node, corresponding to the minimum cost point. We configure keys in Anna to have a default replication factor of 1 since neither ElastiCache nor Masstree supports replication of any kind. To measure the performance for a fixed price, we also disabled Anna's elasticity mechanism.

Figure 5a shows that Anna consistently outperforms both Masstree and ElastiCache under low contention. As discussed in our previous work, this is because Anna's thread-per-core coordination-free execution model efficiently exploits multi-core parallelism, while other systems suffer from thread synchronization overhead through the use of locks or atomic instructions. Neither Anna nor Anna v0 replicates data in this experiment, so they deliver identical performance.

Under high contention (Fig. 5b), Anna's throughput increases linearly with cost, while both ElastiCache and Masstree plateau. Anna selectively replicates hot keys across nodes and threads to spread the load, enabling this linear scaling; the other two systems do not have this capability. Anna v0 replicates the *entire* database across all nodes. While Anna v0's performance scales, the absolute throughput is worse than Anna's because naively replicating the entire database

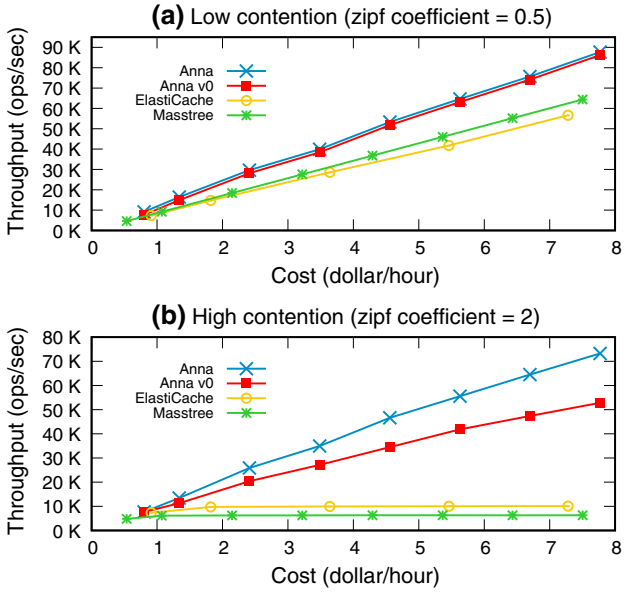

Fig. 5 Cost-effectiveness comparison between Anna, Anna v0, ElastiCache, and Masstree

increases multicast overhead for cold keys. Furthermore, Anna v0's storage consumption is significantly higher: At $7.80/hour (14 memory nodes), Anna v0's constant replication generates 13× the original data size, while Anna incurs <1% extra storage overhead.

7.4 Dynamic workload skew and volume

We now combine selective replication and elasticity to react to changes in workload skew and volume. In this experiment, we start with 12 memory-tier nodes and a latency objective of 3.3 ms—about 33% above our unsaturated latency. All servers serve a light load at time 0. At minute 3, we start a high-contention workload with a Zipfian coefficient of 2. We see in Fig. 6a that after a brief spike in latency, Anna replicates the highly contended keys and meets the latency SLO (the dashed red line). At minute 13, we reduce the Zipfian coefficient to 0.5, switching to a low contention workload. Simultaneously, we increase the load volume by a factor of 4. Detecting these changes, the policy engine reduces the replication factors of the previously hot keys. It finds that all nodes are occupied with client requests and triggers addition of four new nodes to the cluster. We see a corresponding increase in the system cost in Fig. 6b.

It takes 5 min for the new nodes to join the cluster. Throughput increases to the saturation point of all nodes (the first plateau in Fig. 6b), and the latency spikes to the SLO maximum from minutes 13 to 18. At minute 18, the new nodes come online and trigger a round of data repartitioning, seen by the brief latency spike and throughput dip. Anna then further increases throughput and meets the latency SLO.

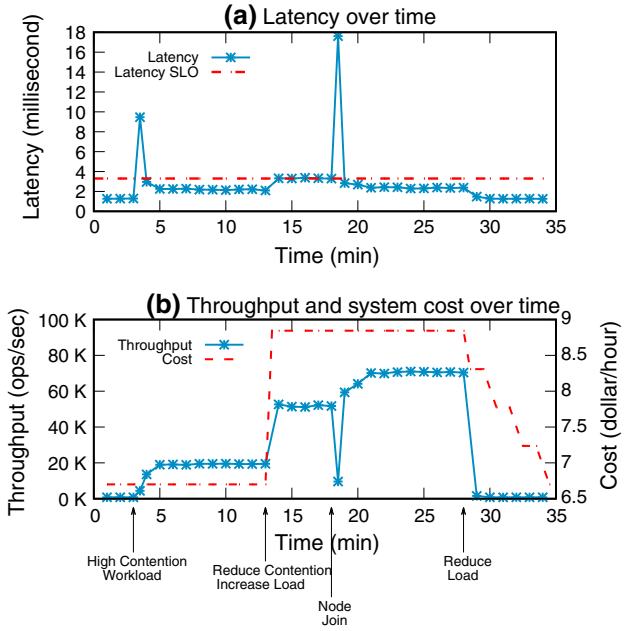

Fig. 6 Anna's response to changing workload

At the 28-min point, we reduce the load, and Anna removes nodes to save cost.

Throughout the 32-min experiment, the latency SLO is satisfied 97% of the time. We first violate the SLO during hot-key replication by 4× for 15 s. Moreover, the latency spikes to 7× the SLO during redistribution for about 30 s. Data redistribution causes multicast overhead on the storage servers and address cache invalidation on the clients. The latency effects are actually not terrible. As a point of comparison, TCP link latencies in data centers are documented tolerating link delays of up to 40× [5].

From minutes 13–18, we meet our SLO of 3.3 ms exactly. With a larger load spike or lower initial resource allocation, Anna could have easily violated its SLO during that period, putting SLO satisfaction at 83%—a much less impressive figure. Under any reactive policy, large workload variations can cause significant SLO violations. As a result, cloud providers commonly develop client-specific service-level agreements (SLAs) that reflect access patterns and latency expectations. In practice, these SLAs allow for significantly more leeway than a service's internal SLO [46].

7.5 Varying hotspot

Next, we introduce multiple tiers and run a controlled experiment to demonstrate the effectiveness of cross-tier promotion and demotion. The goal is to evaluate Anna's ability to detect and react to changes in workload hotspots. We do not consider a latency objective and disable autoscaling; we narrow our focus to how quickly Anna identifies hot data.

We fix total data size while varying the number of keys and the length of the values. This stress-tests selective replication, for which the amount of metadata (i.e., a per-key replication vector) increases linearly with the number of keys. Increasing the number of keys helps us evaluate how robust Anna's performance is under higher metadata overheads.

We allocate 3 memory nodes (insufficient to store all data) and 15 EBS-tier nodes. At time 0, most data is in the EBS tier. The blue curve in Fig. 7 shows a moderately skewed workload, and the green curve shows a highly skewed workload. At minute 0, we begin a workload centered around one hotspot. At minute 5, we switch to a different, largely non-overlapping hotspot, and at minute 10, we switch to a third, unique hotspot. The y-axis measures what percent of queries are served by the memory tier—the "cache hit" rate.

With 1 million keys and 256KB values (Fig. 7a), we see that Anna is able to react almost immediately and achieve a perfect hit rate under a highly skewed workload (the green curve). The hot set is very small—on the order of a few thousand keys—and all hot keys are promoted in about 10 s. The moderately skewed workload shows more variation. We see the same dip in performance after the hotspot changes; however, we do not see the same stabilization. Because the working set is much larger, it takes longer for hot keys to be promoted, and there is a probabilistic "fringe" of keys that are in cold storage at time of access, leading to hit rate variance. Nonetheless, Anna is still able to achieve an average of 81% hit rate less than a minute after the change.

Increasing the number of keys (Fig. 7b, c) increases the time to stabilization. Achieving a hit rate of 99.5% under the highly skewed workload (the green curves) takes around 15 and 25 s for 10 million and 100 million keys, respectively. Under the moderately skewed workload (the blue curves), the hit rate in both settings takes around 90 s to stabilize. We observe a slightly reduced average hit rate (79% and 77%, respectively) due to a larger probabilistic fringe of cold keys. Overall, despite orders of magnitude more keys, Anna still adapts and achieves a high memory-tier hit rate. In Sect. 9, we discuss opportunities to improve time to stabilization further via policy tuning.

7.6 Recovery

We evaluate Anna's ability to recover from node failure and compare against Redis on AWS ElastiCache. We choose Redis because it is the only KVS in our experiments with recovery features. Both systems were run on 42 memory-tier nodes and maintain three replicas per key. The results are shown in Fig. 8. Note that we report *normalized* throughput here to compare against each system's baseline.

Both systems are run at steady state before a random node is terminated non-gracefully at minute 4, marked with a red line in Fig. 8. Anna (the blue curve) experiences a 1.5-min dip

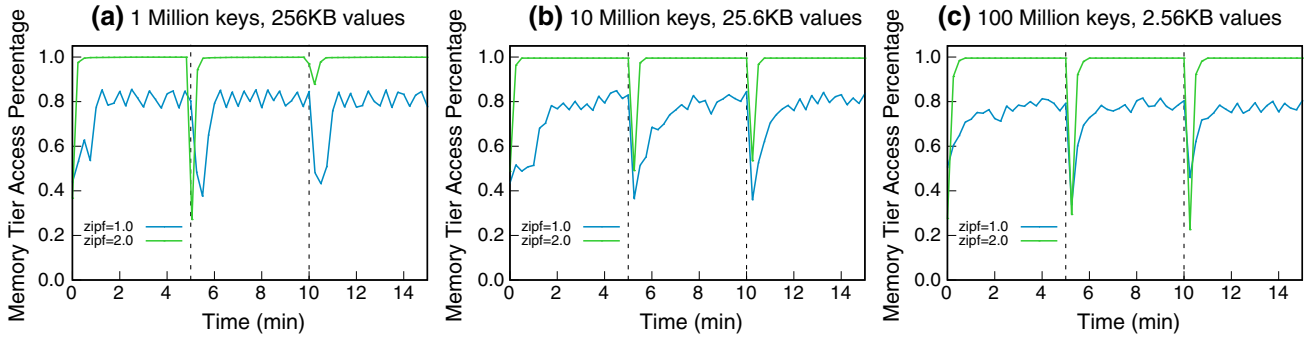

Fig. 7 Adapting to changing hotspots in workload

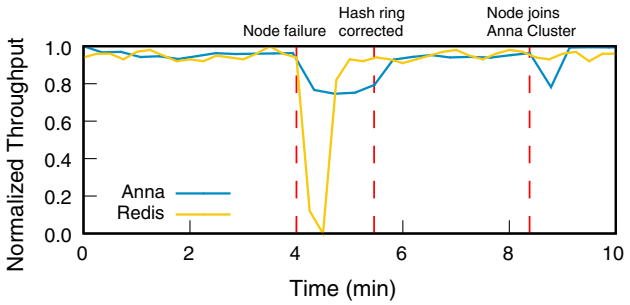

Fig. 8 Impact of node failure and recovery for Anna and Redis (on AWS ElastiCache)

in performance while requests to the now-terminated node timeout. The performance change is not immediately apparent as the node continues serving requests for a few seconds before all processes are terminated. Nonetheless, Anna maintains above 80% of peak throughput because replication is multi-mastered; the remaining replicas still serve requests. After a minute, the system detects a node has departed and updates its hash ring. There is slightly diminished performance (about 90% of peak) from minutes 5.5–8.5, while the system operates normally but with 1 fewer node. At minute 8.5, we see another dip in throughput as a new node joins and the system repartitions data[1]. By minute 9, the system returns to peak performance.

User requests are set to time out after 500 ms. We observe a steady state latency of about 18 ms. After node failure, roughly $\frac{1}{42}$ of requests query the failed node and wait until timeout to retry elsewhere. This increases latency for those requests, but reduces load on live nodes; as a result, other requests observe latencies drop to about 10 ms. Hence, with one failed node, we expect to see an average latency of $510 \times \frac{1}{42} + 10 \times \frac{41}{42} = 21.90$ ms. This implies throughput at roughly 82% of peak and matches the performance in Fig. 8. A larger cluster would further mitigate the performance dip.

[1] Note that repartitioning overhead is not as high as in Sect. 7.4 because here we are using more machines and only add one new node, as opposed to four in that experiment.

Redis maintains 14 shards, each with one primary and two read replicas. We terminate one of the primary replicas. The yellow curve in Fig. 8 shows that throughput immediately drops to 0 as Redis stops serving requests and elects a new leader for the replica group with the failed node. A new node is allocated and data is repartitioned by minute 6, after which Redis returns to peak performance. As a single-master system that provides linearizability, it is not designed to run in an environment where faults are likely.

In summary, Anna is able to efficiently respond to node failure while maintaining over 80% peak throughput, whereas Redis pauses the system during leader election. Anna's high availability makes it a much better fit for cloud deployments. On the other hand, Redis's performance normalizes after a minute as its node spin-up time is much lower than ours (about 4 min)—we return to this point in Sect. 9.

7.7 Cost-performance trade-offs

Finally, we assess how well Anna is able to meet its SLOs. We study the Pareto efficiency of our policy: How well does it find a frontier of cost–performance trade-offs? We sweep the SLO parameter on one of the two axes of cost and latency and observe the outcome on the other. Anna uses both storage tiers and enable all policy actions. We evaluate three contention levels—Zipfian coefficients of 0.5 (about uniform), 0.8, and 1.0 (moderately skewed). For a database of 1M keys with a three replicas per key, Anna needs four EBS nodes to store all data and one memory node for metadata; this is a minimum deployment cost of $2.06 per hour.

At each point, we wait for Anna to achieve steady state, meaning that nodes are not being added or removed and latency is stable. In Fig. 9a, we plot Anna's steady state latency for a fixed cost SLO. We measure average request latency over 30 s. At $2.10/h (4 EBS nodes and 1 memory node), only a small fraction of hot data is stored in the memory tier due to limited storage capacity. The observed latency ranges from 50 to 250 ms across contention levels. Requests under the high-contention workload are more likely to hit

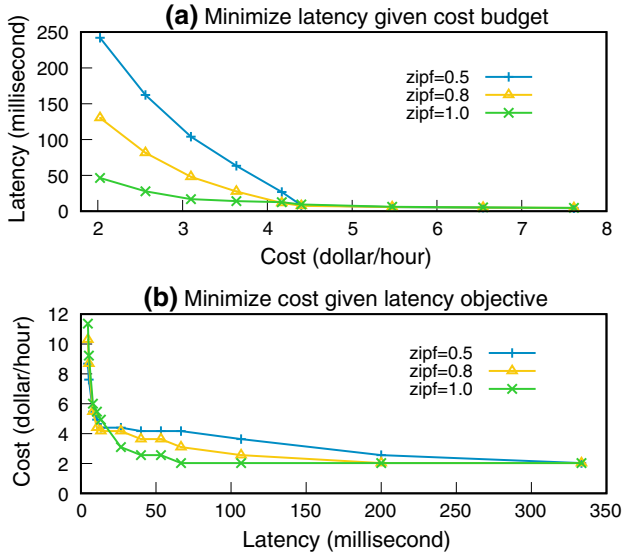

Fig. 9 Varying contention, we measure **a** Anna latency per cost budget; **b** Anna cost per latency objective

the small set of hot data in the memory tier. As we increase the budget, latency improves for all contention levels: more memory nodes are added and a larger fraction of the data is memory-resident. At $4.40/h, Anna can promote at least one replica of all keys to the memory tier. From here on, latency is under 10 ms across all contention levels. Performance differences between the contention levels are negligible thanks to hot-key replication.

We also compare the throughput between Anna and DynamoDB at each cost budget. Similar to Anna, DynamoDB is a serverless KVS. System deployment details, such as node instance type, node count, and resource utilization, are hidden from the users. DynamoDB allows users to define high-level objectives such as read/write capacity units which translate to maximum throughput, and cost budget. The system autoscales based on the workload to meet these goals. Note that in this experiment, DynamoDB is configured to provide the same eventual consistency guarantees and fault tolerance metric ($k = 2$) as Anna. As shown in Table 4, Anna outperforms DynamoDB by 36× under a low-cost regime and by as much as 355× at higher costs. Our observed DynamoDB performance is actually somewhat better than AWS's advertised performance [6], which gives us confidence that this result is a reasonable assessment of DynamoDB's efficiency.

Lastly, we set Anna to minimize cost for a stated latency objective (Fig. 9b). Once more, when the system reaches steady state, we measure its resource cost. To achieve sub-5ms latency—the left side of Fig. 9b—Anna requires $9-11 per hour depending on the contention level. This latency requires at least one replica of all keys to be in the memory tier. Between 5 and 200 ms, higher-contention workloads are cheaper, as hot data can be concentrated on a few memory nodes. For the same latency range, lower contention workloads require more memory and are thus more expensive. Above 200 ms, most data resides on the EBS tier, and Anna meets the latency objective at about $2 an hour.

Table 4 Throughput comparison between Anna and DynamoDB at different cost budgets

Cost	Anna	DynamoDB
$2.50/hour	1271 ops/s	35 ops/s
$3.60/hour	3352 ops/s	55 ops/s
$4.40/hour	23017 ops/s	71 ops/s
$5.50/hour	33548 ops/s	90 ops/s
$6.50/hour	38790 ops/s	108 ops/s
$7.60/hour	43354 ops/s	122 ops/s

8 Related work

As a KVS, Anna builds on prior work, from both the databases and distributed systems literature. Nonetheless, it is differentiated in how it leverages and combines these ideas to achieve new levels of efficiency and automation.

Autoscaling Cloud Storage. A small number of cloud-based file systems have considered workload responsiveness and autoscaling. Sierra [54] and Rabbit [7] are single-master systems that handle the problem of read and write offloading: when a node is inactive or overloaded, requests to blocks at that node need to be offloaded to alternative nodes. This is particularly important for writes to blocks mastered at the inactive node. SpringFS [61] optimizes this work by finding a minimum number of machines needed for offloading. By contrast, Anna supports multi-master updates and selective key replication. When nodes go down or get slow in Anna, writes are simply retried at any existing replica, and new replicas are spawned as needed by the policy.

ElastMan [4] is a "bolt-on" elasticity manager for cloud KVSs that responds to changing workload volume. Anna, on the other hand, manages the dynamics of skew and hotspots in addition to volume. ElastMan's proactive policy is an interesting feature that anticipates workload changes like diurnal patterns; we return to this in Sect. 9.

Consistent hashing and distributed hash Tables [28,45,50] are widely used in many storage systems [14,19] to facilitate dynamic node arrival and departure. Anna allows request handling and key migration to be interleaved, eliminating downtime during node membership change while ensuring consistency, thanks to its lattice-based conflict resolution.

Key-Value Stores. There has been a wide range of work on key-value stores for both multicore and distributed systems—more than we have room to survey. Our earlier work [60]

offers a recent snapshot overview of that domain. In this paper, our focus is not on the KVS kernel, but on mechanisms to adapt to workload distributions and trade-offs in performance and cost.

Selective Key Replication. Selective replication of data for performance has a long history, dating back to the Bubba database system [17]. More recently, the ecStore [56], Scarlett [8], E2FS [15], and SWORD [43] systems perform single-master selective replication, which creates *read-only* replicas of hot data to speed up read performance. Content delivery network (CDN) providers such as Google Cloud CDN [24], Swarmify [53], and Akamai [3] use similar techniques to replicate content close to the edge to speed up delivery. In comparison, Anna's multi-master selective replication improves *both* read and write performance, achieving general workload scaling. Conflicting writes to different replicas are resolved asynchronously using our lattices' merge logic [60].

Selective replication requires maintaining metadata to track hot keys. ecStore uses histograms to reduce hot-key metadata, while Anna currently maintains access frequencies for the full key set. We are exploring two traditional optimizations to reduce overhead: heavy hitter sketches rather than histograms [35] and the use of distributed aggregation for computing sketches in parallel with minimal bandwidth [39].

Another effort to address workload skew is Blowfish [29], which combines the idea of replication and compression to trade-off storage and performance under time-varying workloads. Adding compression to Anna to achieve fine-grained performance cost trade-off is an interesting future direction.

Tiered Storage. Beyond textbook caching, there are many interesting multi-tier storage systems in the literature. A classic example in the file systems domain is the HP AutoRaid system [58]. Databases also considered tertiary storage during the era of WORM devices and storage robots [37,51]. Broadcast Disks envisioned using multiple broadcast frequencies to construct arbitrary hierarchies of virtual storage [2]. More recently, there has been interest in filesystem caching for analytics workloads. OctopusFS [27] is a tiered file system in this vein. Tachyon [36] is another recent system that serves as a memory cache for analytics working sets, backing a file system interface. Our considerations are rather different than prior work: The size of each tier in Anna can change due to elasticity, and the volume of data to be stored overall can change due to dynamic replication.

9 Conclusion and future work

Anna provides a simple, unified API to efficient key-value storage in the cloud. Unlike popular storage systems today, it supports a non-trivial *distribution* of access patterns by eliminating common static deployment and cost–performance barriers. Developers declare their desired trade-offs, instead of managing a custom mix of heterogenous services.

Behind this API, Anna uses three core mechanisms to meet SLOs efficiently: *horizontal elasticity* to right size the service by adding and removing nodes, *vertical data movement across tiers* to reduce cost by demoting cold data, and *multi-master selective replication* to scale request handling at a fine granularity. The primary contribution of Anna is its integration of these features into an efficient, autoscaling system representing a new design point for cloud storage. These features are enabled by a policy engine which monitors workloads and responds by taking the appropriate actions.

Our evaluation shows that Anna is extremely efficient. In many cases, Anna is orders of magnitude more cost-effective than popular cloud storage services and prior research systems. Anna is also unique in its ability to automatically adapt to variable workloads.

Although Anna's design addresses the main weaknesses of modern cloud storage that we set out to study, it also raises a number of interesting avenues for research.

Proactive Policy Design. Our current policy design is entirely reactive, taking action based on current state. To improve this, we are interested in *proactive* policies that anticipate upcoming workload changes and act in advance [4,38,42,47]. By combining advanced predictive techniques with Anna's swift adaptivity, we believe Anna will further excel in meeting SLOs & SLAs.

Defining SLOs & SLAs. Currently, the system administrator defines a single latency objective corresponding to an overall average. For any system configuration, there are adversarial workloads that can defeat this SLO. For example, in Sect. 7.4, a larger load spike could have forced Anna above its stated SLO for a long period. SLOs, SLAs and policies can be designed for both expected- and worst-case scenarios, using pricing and incentives.

A fundamental issue is that users with large working sets require more resources at the memory tier to hit a given SLO. This is clear in Fig. 9: If each workload corresponds to a user, the user with lower Zipfian parameter costs more to service at a given SLO. SLAs should be designed to account for costs varying across users.

Reducing Autoscaling Overhead. The 5-min delay for node addition noted in Sect. 7.4 is a significant problem. It limits the effectiveness of any autoscaling policy, since feedback from allocating a new node is delayed for an eternity in compute time. A standard solution for cloud providers today is to maintain a standby pool of "warm" nodes that are partially prepared for use. To make this cost-effective, these nodes have to run alternative containers that monetize the idle resources. An alternative solution is to make "cold" container startup much faster than it is today. This is a well-studied problem for desktop operating systems [41] and VM research [18,33,59]. The open-source Firecracker

project [23] is a new micro-VM framework for the cloud that helps solve this problem by reducing VM startup time to 100s of milliseconds.

Evaluating Other Tiers. Currently, Anna is implemented over two tiers, but cloud providers like AWS offer a much wider array of price–performance regimes. There is an opportunity to add services at both ends of the price–performance spectrum that can leverage Anna's autoscaling scaling and coordination-free execution. As mentioned earlier, our storage kernel requires very little modification to support new storage layers. Our policy engine also naturally supports more than two tiers. However, our current thresholds in Sect. 5 are the result of significant empirical measurement and tuning. These parameters will need to be adjusted to the underlying storage hardware. This effort could be replaced by auto-tuning approaches that learn models of configurations, workloads, and parameter settings. There has been recent work on similar auto-tuning problems [25,55].

Appendix

We include pseudocode for the algorithms described in Sect. 5 here. Note that some algorithms included here rely on a latency objective, which may or may not be specified. When no latency objective is specified, Anna aspires to its unsaturated request latency (2.5 ms) to provide the best possible performance but caps spending at the specified budget.

Algorithm 1 DataMovement

Input: Key, $[< R_M, R_E >< T_M, T_E >]$
1: **if** access(Key, T)$> P$ & $R_M = 0$ **then**
2: adjust(Key, $R_M + 1, R_E - 1, T_M, T_E$)
3: **else if** access(Key, T)$< D$ & $R_M > 0$ **then**
4: adjust(Key, $0, k + 1, 1, 1$)

Algorithm 2 HotKeyReplication

Input: Key, $[< R_M, R_E >< T_M, T_E >]$
1: **if** access(Key, T)$> H$ & $R_M < N_M$ **then**
2: SET $R_{M\_ideal} = R_M * L_{obs}/L_{obj}$
3: SET $R'_M = min(R_{M\_ideal}, N_M)$
4: adjust(Key, R'_M, R_E, T_M, T_E)
5: **else if** access(Key, T)$> H$ & $R_M = N_M$ **then**
6: SET $T_{M\_ideal} = T_M * L_{obs}/L_{obj}$
7: SET $T'_M = min(T_{M\_ideal}, N_{T\_memory})$
8: adjust(Key, R_M, R_E, T'_M, T_E)
9: **else if** access(Key, T)$< L$ & $(R_M > 1 \parallel T_M > 1)$ **then**
10: adjust(Key, $1, k, 1, 1$)

Algorithm 3 NodeAddition

Input: $tier, mode$
1: **if** $mode = storage$ **then**
2: SET N_{target} = required_storage($tier$)
3: **if** $Cost_{target} > Budget$ **then**
4: SET N_{target} = adjust()
5: add_node(tier, $N_{target} - N_{tier\_current}$)
6: **else if** $mode = compute$ & $tier = M$ **then**
7: SET $N_{target} = N_{M\_current}$*min($L_{obs}/L_{obj}, c$)
8: **if** $Cost_{target} > Budget$ **then**
9: SET N_{target} = adjust()
10: add_node($M, N_{target} - N_{M\_current}$)

Algorithm 4 NodeRemoval

Input: $tier, mode$
1: **if** $mode = storage$ & $tier = E$ **then**
2: SET N_{target} = max(required_storage(E), $k + 1$)
3: reduce_replication()
4: remove_node($E, N_{E\_current} - N_{target}$)
5: **else if** $mode = compute$ & $tier = M$ **then**
6: **if** $N_{M\_current} > 1$ **then**
7: reduce_replication()
8: remove_node(M, 1)

Algorithm 5 AnnaPolicy

Input: $tiers = \{M, E\}, keys$
1: **for** $tier$ in $tiers$ **do**
2: **if** storage($tier$)$> S_{upper}$ **then**
3: NodeAddition($tier, storage$)
4: **else if** storage($tier$)$< S_{lower}$ **then**
5: NodeRemoval($tier, storage$)
6: **for** $key \in keys$ **do**
7: DataMovement(key)
8: **if** $L_{obs} > f_{upper} * L_{obj}$ & compute(M)$> C_{upper}$ **then**
9: NodeAddition($M, compute$)
10: **else if** $L_{obs} > f_{upper} * L_{obj}$ & compute(M)$<= C_{upper}$ **then**
11: **for** $key \in keys_{memory}$ **do**
12: HotKeyReplication(key)
13: **else if** $L_{obs} < f_{lower} * L_{obj}$ & compute(M)$< C_{lower}$ **then**
14: NodeRemoval($M, compute$)

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer **45**(2), 37–42 (2012)
2. Acharya, S., Alonso, R. Franklin, M., Zdonik, S.: Broadcast disks: data management for asymmetric communication environments. In: Mobile Computing, pp. 331–361. Springer (1995)
3. Akamai. https://www.akamai.com
4. Al-Shishtawy, A., Vlassov, V.: Elastman: Elasticity manager for elastic key-value stores in the cloud. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference, CAC '13, pp. 7:1–7:10. ACM, New York (2013)
5. Alizadeh, M., Greenberg, A., Maltz, D.A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., Sridharan, M.: Data center tcp (dctcp). In: Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10, pp. 63–74. ACM, New York (2010)
6. Amazon Web Services. Amazon dynamodb developer guide (api version 2012-08-10), Aug. 2012. https://docs.aws.amazon.

com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html. Accessed May 3, (2018)
7. Amur, H., Cipar, J., Gupta, V., Ganger, G.R., Kozuch, M.A., Schwan, K.: Robust and flexible power-proportional storage. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, pp. 217–228. ACM, New York (2010)
8. Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., Harris, E.: Scarlett: Coping with skewed content popularity in mapreduce clusters. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, pp. 287–300. ACM, New York (2011)
9. Amazon web services. https://aws.amazon.com
10. Microsoft azure cloud computing platform. http://azure.microsoft.com
11. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. PVLDB **7**(3), 181–192 (2013)
12. Birman, K., Chockler, G., van Renesse, R.: Toward a cloud computing research agenda. ACM SIGACt News **40**(2), 68–80 (2009)
13. Brewer, E.: A certain freedom: Thoughts on the cap theorem. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '10, pp. 335–335. ACM, New York (2010)
14. Apache cassandra. http://cassandra.apache.org
15. Chen, L., Qiu, M., Song, J., Xiong, Z., Hassan, H.: E2fs: an elastic storage system for cloud computing. J. Supercomput. **74**(3), 1045–1060 (2018)
16. Conway, N., Marczak, W.R., Alvaro, P., Hellerstein, J.M., Maier, D.: Logic and lattices for distributed programming. In: Proceedings of the Third ACM Symposium on Cloud Computing, SoCC '12, pp. 1:1–1:14. ACM, New York (2012)
17. Copeland, G., Alexander, W., Boughter, E., Keller, T.: Data placement in Bubba. In: ACM SIGMOD Record, volume 17, pp. 99–108. ACM (1988)
18. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High availability via asynchronous virtual machine replication. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, pp. 161–174. San Francisco (2008)
19. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pp. 205–220. ACM, New York (2007)
20. Demers, A., Greene, D., Houser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. ACM SIGOPS Oper. Syst. Rev. **22**(1), 8–32 (1988)
21. Kubernetes–build, ship, and run any app, anywhere. https://www.docker.com
22. Faleiro, J.M., Abadi, D.J.: Latch-free synchronization in database systems: Silver bullet or fool's gold? In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, CIDR '17 (2017)
23. Firecracker. https://firecracker-microvm.github.io
24. Google cloud platform. https://cloud.google.com
25. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: a self-tuning system for big data analytics. CIDR **11**, 261–272 (2011)
26. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX annual technical conference, volume 8. Boston, USA (2010)
27. Kakoulli, E., Herodotou, H.: Octopusfs: A distributed file system with tiered storage management. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, pp. 65–78. ACM, New York (2017)
28. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: onsistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC '97, pp. 654–663. ACM, New York (1997)
29. Khandelwal, A., Agarwal, R., Stoica, I.: Blowfish: Dynamic storage-performance tradeoff in data stores. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pp. 485–500. USENIX Association, Santa Clara (2016)
30. Kubernetes: Production-grade container orchestration. http://kubernetes.io
31. Kubernetes. Set up high-availability kubernetes masters. https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/. Accessed May 3, (2018)
32. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pp. 239–250. ACM, New York (2015)
33. Lagar-Cavilla, H.A., Whitney, J.A., Scannell, A.M., Patchin, P., Rumble, S.M., De Lara, E. Brudno, M., Satyanarayanan, M.: Snowflock: rapid virtual machine cloning for cloud computing. In: Proceedings of the 4th ACM European conference on Computer systems, pp. 1–12. ACM (2009)
34. Lamport. L.: The part-time parliament. ACM Transactions on Computer Systems (TOCS), **16**(2), (1998)
35. Larsen, K.G., Nelson, J., Nguyen, H.L., Thorup, M.: Heavy hitters via cluster-preserving clustering. CoRR, arXiv:1604.01357, (2016)
36. Li, H., Ghodsi, A., Zaharia, M., Shenker, S., Stoica, I.: Tachyon: Reliable, memory speed storage for cluster computing frameworks. In: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, pp. 6:1–6:15. ACM, New York (2014)
37. Lomet, D., Salzberg, B.: Access methods for multiversion data. SIGMOD Rec. **18**(2), 315–324 (1989)
38. Ma, L., Van Aken, D., Hefny, A., Mezerhane, G., Pavlo, A., Gordon, G.J.: Query-based workload forecasting for self-driving database management systems. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pp. 631–645 (2018)
39. Manjhi, A., Nath, S., Gibbons, P.B.: Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, pp. 287–298. ACM, New York (2005)
40. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 183–196. ACM (2012)
41. Microsoft Corp. Delivering a great startup and shutdown experience, May 2017. https://docs.microsoft.com/en-us/windows-hardware/test/weg/delivering-a-great-startup-and-shutdown-experience. Accessed May 3 (2018)
42. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D.V., Wang, Z., Wu, Y., Xian, R., Zhang, T.: Self-driving database management systems. In: CIDR 2017, Conference on Innovative Data Systems Research (2017)
43. Quamar, A., Kumar, K.A., Deshpande, A.: Sword: Scalable workload-aware data placement for transactional workloads. In: Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13, pp. 430–441. Association for Computing Machinery, New York (2013)
44. Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in structured p2p systems. In: International Workshop on Peer-to-Peer Systems, pp. 68–79. Springer (2003)

45. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network, vol. 31. ACM, New York (2001)
46. Ross, A., Hilton, A., Rensin, D.: Slos, slis, slas, oh my - cre life lessons, january 2017. https://cloudplatform.googleblog.com/2017/01/availability-part-deux--CRE-life-lessons.html. Accessed May 3, (2018)
47. Roy, N., Dubey, A., Gokhale, A.: Efficient autoscaling in the cloud using predictive models for workload forecasting. In: Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, CLOUD '11, pp. 500–507. IEEE Computer Society, Washington (2011)
48. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. editors, Stabilization, Safety, and Security of Distributed Systems, pp. 386–400. Springer, Berlin (2011)
49. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), MSST '10, pp. 1–10. IEEE Computer Society, Washington (2010)
50. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01, pp. 149–160. ACM, New York (2001)
51. Stonebraker, M.: The design of the postgres storage system. In: Proceedings of the 13th International Conference on Very Large Data Bases, VLDB '87, pp. 289–300. Morgan Kaufmann Publishers Inc., San Francisco (1987)
52. Storm. https://github.com/apache/storm
53. Swarmify. https://swarmify.com
54. Thereska, E., Donnelly, A., Narayanan, D.: Sierra: Practical power-proportionality for data center storage. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys '11, pp. 169–182. ACM, New York (2011)
55. Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1009–1024. ACM (2017)
56. Vo, H.T., Chen, C., Ooi, B.C.: Towards elastic transactional cloud storage with range query support. PVLDB **3**(1–2), 506–514 (2010)
57. Waas, F.M.: Beyond conventional data warehousing - massively parallel data processing with greenplum database–(invited talk). In: Dayal, U., Castellanos, M., Sellis, T. editors, Business Intelligence for the Real-Time Enterprise–Second International Workshop, BIRTE 2008, Auckland, New Zealand, August 24, 2008, Revised Selected Papers, pp. 89–96 (2008)
58. Wilkes, J., Golding, R., Staelin, C., Sullivan, T.: The hp autoraid hierarchical storage system. ACM Trans. Comput. Syst. **14**(1), 108–136 (1996)
59. Wood, T., Shenoy, P.J., Venkataramani, A., Yousif, M.S., et al.: Black-box and gray-box strategies for virtual machine migration. NSDI **7**, 17–17 (2007)
60. Wu, C., Faleiro, J.M., Lin, Y., Hellerstein, J.M.: Anna: A kvs for any scale. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE) (2018)
61. Xu, L., Cipar, J., Krevat, E., Tumanov, A., Gupta, N., Kozuch, M.A., Ganger, G.R.: Springfs: Bridging agility and performance in elastic distributed storage. In: Proceedings of the 12th USENIX FAST, pp. 243–255. USENIX (2014)