Techniques and Efficiencies from Building a Real-Time DBMS

V. Srinivasan
Aerospike
Mountain View, U.S.A.
srini@aerospike.com

Andrew Gooding
Aerospike
Mountain View, U.S.A.
andy@aerospike.com

Sunil Sayyaparaju
Aerospike
Bengaluru, India
sunil@aerospike.com

Thomas Lopatic
Aerospike
Berlin, Germany
thomas@aerospike.com

Kevin Porter
Aerospike
Mountain View, U.S.A.
kporter@aerospike.com

Ashish Shinde
Aerospike
Bengaluru, India
ashish@aerospike.com

B. Narendran
Aerospike
Berkeley Heights, U.S.A.
bnarendran@aerospike.com

ABSTRACT

This paper describes a variety of techniques from over a decade of developing Aerospike (formerly Citrusleaf), a real-time DBMS that is being used in some of the world's largest mission-critical systems that require the highest levels of performance and availability. Such mission-critical systems have many requirements including the ability to make decisions within a strict real-time SLA (milliseconds) with no downtime, predictable performance so that the first and billionth customer gets the same experience, ability to scale up 10X (or even 100X) with no downtime, support strong consistency for applications that need it, synchronous and asynchronous replication with global transactional capabilities, and the ability to deploy in any public and private cloud environments.

We describe how using efficient algorithms to optimize every area of the DBMS helps the system achieve these stringent requirements. Specifically, we describe, effective ways to shard, place and locate data across a set of nodes, efficient identification of cluster membership and cluster changes, efficiencies generated by using a 'smart' client, how to effectively use replications with two copies replication instead of three-copy, how to reduce the cost of the real-time data footprint by combining the use of memory with flash storage, self-managing clusters for ease of operation including elastic scaling, networking and CPU optimizations including NUMA pinning with multi-threading. The techniques and efficiencies described here have enabled hundreds of deployments to grow by many orders of magnitude with near complete uptime.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi: 10.14778/3611540.3611556

PVLDB Reference Format:
V.Srinivasan, Andrew Gooding, Sunil Sayyaparaju, Thomas Lopatic, Kevin Porter, Ashish Shinde, and B. Narendran. Techniques and Efficiencies from Building a Real-Time DBMS. PVLDB, 16(12): 3676-3688, 2023.
doi: 10.14778/3611540.3611556

1 INTRODUCTION

Real-time services requiring extremely high performance and availability have used a variety of solutions over time, including mainframes, clustered relational databases, in-memory databases, and most recently, NoSQL [2] and New SQL databases like CockroachDB [17] and YugabyteDB [18]. Real-time applications create enormous strain on these systems, as follows:

- Overwhelming consumer demand from mobile devices that produces enormous real-time load on their systems, with the DBMS quickly becoming the bottleneck.
- Requirement for a richer set of application features combined with a great real-time consumer experience.
- Virtually 100% online user interactions from multiple remote endpoints making real-time security, risk computation and fraud detection mandatory.
- Round-the-clock availability making any breach of a service-level agreement (SLA) via increased latencies, downtime, and maintenance windows, unacceptable.

Here are some real-world use cases that illustrate the point.

1.1 Use Cases

Let us consider high traffic use cases in three areas, an operational database at the edge of the datacenter, a real-time system of record (SOR) and a global transaction system for high throughput applications. All of these areas require high performance, high availability and various levels of data consistency.

1.1.1 Operational system at the edge

Typically, in these use cases, models generated in an off-line process are applied to transactional meta data in real-time for applications like fraud detection, recommendation engines, real-time bidding for advertising, etc. As seen in this example of fraud detection for payment systems at PayPal [3], the machine learning and AI algorithms require gathering, maintaining, and accessing in real-time a vast amount of historical data related to entities in the payment network like users, devices, network switches, Wi-Fi routers, etc. When a new transaction arrives, it is necessary to use the transaction meta-data and analyze it combined with recent history of the activities attributed to the actors in this transaction. Therefore, a database that can retrieve and save more data for analysis can improve its effectiveness by minimizing false positives and false negatives while still meeting the stringent real-time SLA of ~100-200ms for generating a fraud score. Consistency is important as the algorithms are only effective if recent behavioral data is input to the AI model for scoring. So, providing consistent access to recently written transaction histories is imperative for preventing fraud before it happens rather than detecting it after the fact.

1.1.2 Real-time system of record

One of the best examples of this was highlighted by Airtel [11] in India, one of the largest mobile operators in the world. Airtel re-invented their customer engagement model with "Customer 360," a methodology that understands each customer intelligently and contextually and powers a completely personalized customer experience. Using a high-performance database as the real-time data backbone behind this "digital brain," Airtel could mine trillions of records to create deep learning capabilities at a more than 25,000 transactions per second run-rate with sub-millisecond latency. Access to the latest data is critical here since having outdated data would result in poor customer experience for hundreds of millions of users.

1.1.3 Global distributed transaction system

Let's look at interbank money transfers. The basics of money transfer - simply moving money and tracking those transactions at massive scale - haven't changed. But the benefits of optimizing - digitally transforming - a core, existing practice are massive at scale. An overnight wire transfer (the old standard) is costly, time-consuming, and very inefficient with various points of human and machine intervention. Today this must happen in seconds without any manual intervention. A good example of this is the Banca d'Italia launch of TIPS (Target Instant Payment Settlement) [5], which "guarantees settlement within seconds and is unique in doing this directly in central bank money." These systems have a dual requirement of being global (active-active across multiple data centers) with the ability to continue without losing a single write in the case of an entire data center failure. Consistency requirements are most stringent here and not a single record can be lost in any case whatsoever. At the same time extremely high availability and excellent real-time performance (very high throughput at low latency) are absolutely necessary to have a successful system.

The challenges of building database systems to handle the above are manifold. First, extremely high performance and complete system availability are required. Tunable consistency is important: the first case above can benefit from a high level of consistency, the second will suffer if consistency falls below a relatively high threshold and the third needs strong consistency all the time. Note that the performance and ability to scale to hundreds of millions to billions of users is critical for the success of all three applications. The user experiences provided to the first user and the billionth user need to be identical.

As these systems encounter the problem of explosive growth, efficiencies of how the DBMS uses the resources available to it are critical for its ability to support such mission-critical deployments at higher and higher scale. The rest of this paper will describe the techniques we have developed for running an efficient DBMS for real-time applications. We will divide the discussion into the following areas: data partitioning (Section 2), cluster self-management (Section 3), minimizing replicas and maximizing availability (Section 4), geo-distributed transactions (Section 5), storage optimizations (Section 6), and CPU and network performance (Section 7). We conclude by summarizing our results in Section 8.

2 DATA PARTITIONING

A key aspect to parallel execution in Aerospike [7], formerly Citrusleaf [42], is the ability to divide and conquer the problem using a data partitioning scheme that distributes data across nodes as shown in Figure 1. A record's primary key is hashed into a 160-bit digest using the RIPEMD algorithm, which is extremely robust against collisions [8]. The digest space is partitioned into 4096 non-overlapping 'partitions.' A partition is the primary unit of data segmentation. Records are assigned to partitions based on the primary key's hash digest. Even if the distribution of keys in the key space is skewed, the distribution of keys in the digest space and therefore in the partition space is uniform.

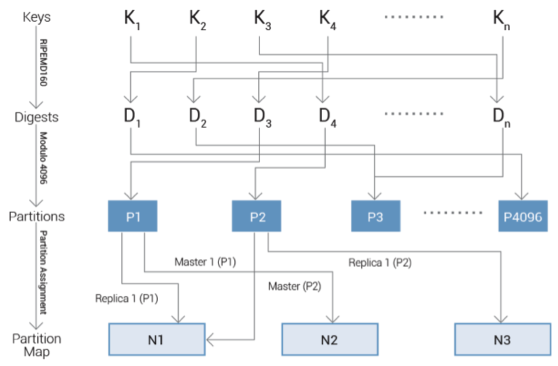


Figure 1: Data Partitioning

The next step is to assign partitions to nodes at random. The partition assignment algorithm has the following objectives:

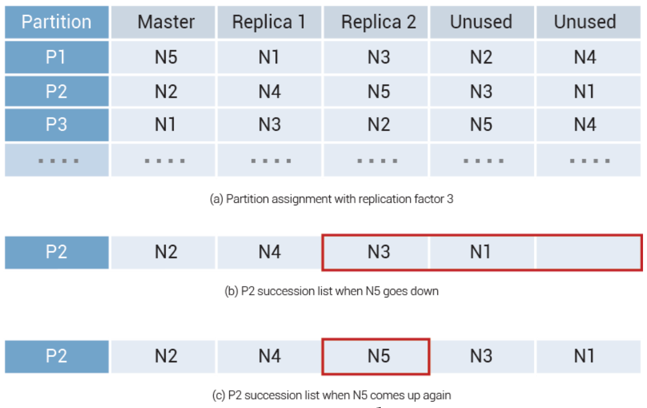
1. **Deterministic**, so that each node in the distributed system can independently compute the same partition map,
2. **Uniform distribution** of master partitions and replica partitions across all nodes in the cluster, and

3. **Minimize data migrations** of partitions during cluster changes (e.g., node arrivals and departures).

The algorithm is described as pseudo code in Table 1 and is deterministic, achieving objective 1. The heart of the assignment is the NODE_HASH_COMPUTE function, which maps a node id and a partition id to a hash value. Note that a specific node's position in the partition replication list is its sort order based on the node hash. We have found that running a Jenkins one-at-a-time hash [40] on the FNV-1a [41] hashes of the node and partition ids gives a good distribution and achieves objective 2 to an extent.

Table 1: Partition Assignment Algorithm

```
function REPLICATION_LIST ASSIGN(partitionid)
  node_hash = empty map
  for nodeid in node_list
    node_hash[nodeid] = NODE_HASH_COMPUTE(nodeid, partitionid)
  replication_list = sort_ascending(node_hash using hash)
  return replication_list
function NODE_HASH_COMPUTE(nodeid, partitionid)
  nodeid_hash = fnv_1a_hash(nodeid)
  partition_hash = fnv_1a_hash(partitionid)
  return jenkins_one_at_a_time_hash(<nodeid_hash, partition_hash>)
```



Partition	Master	Replica 1	Replica 2	Unused	Unused
P1	N5	N1	N3	N2	N4
P2	N2	N4	N5	N3	N1
P3	N1	N3	N2	N5	N4
....	....	....	....	....	....

(a) Partition assignment with replication factor 3

P2	N2	N4	N3	N1	

(b) P2 succession list when N5 goes down

P2	N2	N4	N5	N3	N1

(c) P2 succession list when N5 comes up again

Figure 2: Partition to Node Assignment

Figure 2 shows the partition assignment for a 5-node cluster with a replication factor of 3. Only the first three columns (equal to the replication factor) in the partition map are used; the last two columns are unused.

Consider the case where a node goes down. It is easy to see from the partition replication list that this node would simply be removed from the replication list, causing a left shift for all subsequent nodes as shown in Figure 2(b). If this node did not host a copy of the partition, this partition would not require data migration. If this node hosted a copy of the data, a new node would take its place. This would, therefore, require copying the records in this partition to the new node. Once the original node returns and becomes part of the cluster again, it would simply regain its position in the partition replication list, as shown in Figure 2(c). Adding a brand-new node to the cluster would have the effect of inserting this node at some position in the various partition replication lists, and, therefore, result in the right shift of the subsequent nodes for each partition. Assignments to the left of the new node are unaffected. This discussion shows how the algorithm minimizes data migrations during cluster reconfiguration and achieves objective 3.

2.1 Uniform Partition Balance

While assigning keys to partitions automatically generates a uniform distribution of keys into partitions, assigning partitions to nodes may be skewed if we are not careful. We noticed that as cluster sizes increased to a hundred nodes or higher, there was significant skew in the partition assignment to nodes. For example, a 100-node cluster should have approximately of 40-41 partitions assigned to each node, but we routinely noticed skew in these assignments up to 20-30% depending on cluster size (e.g., a 100-node cluster could have some nodes with 45 partitions and other with just 35). The most occupied node now hits system limits earlier and the entire cluster suffers from inefficient usage of its available capacity.

To fix this, we came up with a modified algorithm that creates uniform balancing of partitions across nodes while minimizing the data migrations, as much as possible. The original algorithm was designed to incur the fewest amount of data migrations on node arrivals and departures.

We define a threshold for the number of claims that need to be assigned to a node for each replica-set before the uniform-balance adjustments begin. For a system that does not use rack awareness, we set the claim threshold to be:

(n_partitions – 128) / n_active_nodes

Empirically we found that since rack-aware configuration applies additional restrictions to the possible balance adjustments, it needed more buffer to ensure near-uniform balance. Therefore, for systems that use rack awareness the claim threshold is set to:

(n_partitions – 1024) /n_active_nodes

For a hundred node cluster the threshold is 39 for a standard cluster or 30 for a rack-aware cluster, respectively. Therefore, balance adjustments will start at shortly before or after partition 3968 (standard) or 3072 (rack-aware).

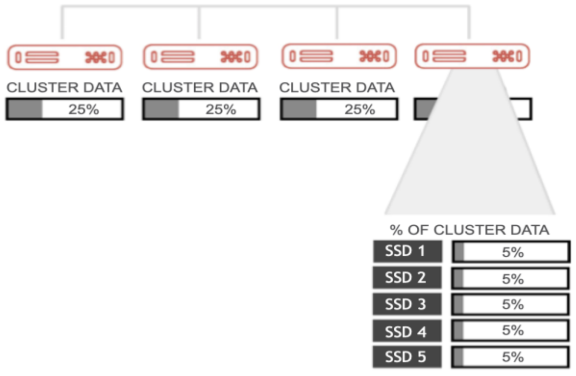


Figure 3: 20-way parallelism with 4 nodes and 5 disks each

The uniform balance adjustments take place when the node initially selected to claim a replica has reached the claim threshold. We choose the node that has the most unfilled claims or, in the case of a tie, also has fewer overall claims allocated to it. Interestingly, we initially assumed that we should choose the node with the most overall claims on a tie but, counter-intuitively, simulations empirically demonstrated that it was better to choose the node with fewer overall claims during a tie. Our rationale is that if the node with fewer claims were to overflow, it would appear to be a node that received a remainder.

2.2 Parallelism without Hot Spots

The uniform data-partitioning significantly enables parallel processing to use system resources in a balanced and efficient manner. Data is uniformly distributed across homogeneous cluster nodes, and, within a node, data can be further distributed randomly into storage devices. This combination provides the opportunity for highly concurrent workload execution resulting in high levels of parallelism, as illustrated in Figure 3. Extrapolating, a 100-node cluster with 16 storage devices per node can drive a 1600-way parallel execution of a high throughput workload of individual record writes in the millions of transactions per second. With the scale up available in a hybrid memory/flash configuration, up to 100TB can be stored per node resulting in database storage of 10 petabytes (in a 100-node cluster) that can be processed at a very high rate of throughput with sub-millisecond read/write latency.

Remarkably, this architecture also works well in reverse when data needs to be fetched from the database. Such a workload can consist of single record operations, batch operations (multi-get) or even a database query with or without a matching secondary index. The data partitioning and distributed layout across nodes and storage devices helps again as the 4K partitions lend themselves to be scanned in parallel.

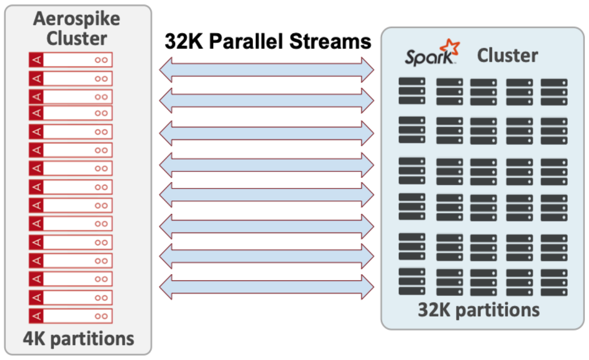


Figure 4: Parallel processing, Aerospike & Apache Spark

Furthermore, scans can be initiated to start from predetermined positions within a partition (in order of hash digests). This means that data can be scanned in parallel in more streams than there are data partitions. Therefore, the database queries can be aligned with the parallel processing available in machine learning systems like Apache Spark^{TM} [21]. For example, it is possible to deploy 32K parallel streams scanning data from the database into a machine learning system in parallel at the rate of hundreds of terabytes per hour (Figure 4). Similar parallel integrations are possible between Aerospike and Presto/Trino for SQL queries.

3 CLUSTER SELF-MANAGEMENT

The basic Aerospike partitioning scheme described in Section 2 is unique to Aerospike and is the basis for the self-management of the cluster. There are three components to clustering (Figure 5):

1. the **heartbeat subsystem** that stores and exchanges information (status of neighboring nodes in adjacency lists) between nodes.
2. the **clustering subsystem** that maintains the membership information (node succession list) corresponding to the current active cluster.
3. the **exchange subsystem** that communicates the partition state and triggers the rebalancing algorithm.

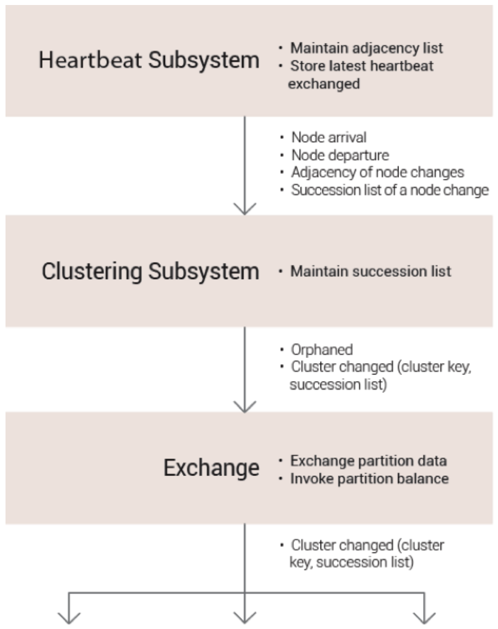

Heartbeat Subsystem
• Maintain adjacency list
• Store latest heartbeat exchanged
• Node arrival
• Node departure
• Adjacency of node changes
• Succession list of a node change
Clustering Subsystem
• Maintain succession list
• Orphaned
• Cluster changed (cluster key, succession list)
Exchange
• Exchange partition data
• Invoke partition balance
• Cluster changed (cluster key, succession list)

Figure 5: Clustering Subsystems

When a node is added to the cluster to provide increased capacity or throughput, the arriving node must contact at least one existing cluster member. The system is configured to use multicast IP or a well-known automatically updated DNS address or a list of individual IP addresses to contact the other nodes in the cluster. When the new node locates the cluster, it begins the process of joining the cluster, first at the heartbeat level, and then through the process of being accepted by the current principal node in the cluster. If accepted, data partitions are allocated to the new node and will be "migrated" from existing nodes to the new hardware. The new node becomes the data master for some partitions and a replica for other partitions, according to the data distribution system outlined in Section 2. As data is migrated, the current data layout is broadcast to connected clients, which automatically route future requests to the correct node.

When a node is removed from the cluster due to hardware failure or shut down for an upgrade, the inverse happens. The heartbeat system will detect absence of the node. The cluster either elects a new principal, or the existing principal will reorganize the cluster. A departing node's partitions will be reallocated to existing nodes by default, maintaining replication factor reliability. For each of the departing node's partitions, one of its current replicas is promoted to be the new master if necessary, and a new replica is selected from among the remaining nodes. Data migration is performed to fill the new replica. Like node addition, clients automatically receive a new partition map as replicas are promoted, and as data migration continues, clients will become aware of new data location.

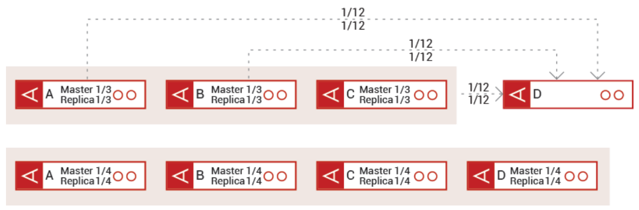


Figure 6: Adding a Node

3.1 Increasing Availability During Node Arrivals and Departures

There are two key optimizations to increase availability during cluster changes. The first is quantum-based event detection for cluster changes that allows multiple complex network changes to be merged into one event and thus handled efficiently with fewer state transitions. The second ensures that the total amount of partition related data that needs to be transferred during cluster change events is bounded by the fraction of the cluster nodes that impact the total number of partitions (4096) times the replication factor (typically 2 or 3), as determined by the partition rebalancing algorithms described in Section 2. This scheme reduces, by an order of magnitude, the information being exchanged in a large cluster of 50-100 nodes. An example node addition is illustrated in Figure 6.

Additionally, it is important to set longer clock quanta settings for detecting cluster changes for public cloud environments for enabling the clustering system to complete all the message transfers and reach a quiescent state within a few seconds after a cluster change. This allows the system to quickly ramp up to the full extent of transactional throughput after only a few seconds of slow down during cluster node arrival and departure events.

3.2 Data Migration Optimizations

The process of moving records from one node to another node is termed a *migration*. After every cluster view change, the objective of data migration is to have the latest version of each record available at the current master and replica nodes for each of the data partitions.

Clusters will self-heal even at demanding times without operator intervention. Capacity planning and system monitoring capabilities provide you the ability to handle virtually any unforeseen failure with negligible loss of service. You can configure and provision your hardware capacity and set up replication/synchronization policies so that the database recovers from failures without affecting users.

The data rebalancing mechanism ensures that the transaction volume is distributed evenly across all nodes and is robust in the event of node failure happening during rebalancing itself. The system is designed to be continuously available, so data rebalancing doesn't impact cluster behavior. There is only a short period when the cluster internal redirection mechanisms are used while clients discover the new cluster configuration by assembling a copy of the partition map by polling the server nodes. Thus, this mechanism optimizes for continuous transactional availability in a scalable shared-nothing environment.

To optimize data migrations, Aerospike defines a notion of partition ordering using partition version numbers that change every time cluster node composition changes. These version numbers help determine whether a partition retrieved from disk needs to be migrated or not. The process of data migration would be a lot more efficient and easier if a total order could be established over partition versions. However, enforcing total ordering of partition version numbers is problematic. When version numbers diverge on cluster splits caused by network partitions, this would require the partial order to be extended to a total order (order extension principle). Yet, this would still not guarantee the retention of the latest versions of each record since the system will end up either choosing the entire version of the partition, or completely rejecting it. Moreover, the amount of information needed to create a partial order on version numbers would only grow with scale. Thus, Aerospike maintains this partition lineage up to a certain number of partition changes.

When two versions come together, nodes negotiate the difference in actual records and send over the data corresponding only to the differences between the two partition versions. In certain cases, migration can be avoided completely based on the knowledge that the content of a partition is a subset of the same partition on another node. In other cases, like rolling upgrades, the delta of changes is small and are shipped over and reconciled instead of shipping the entire partition content.

3.3 Operation during Data Migration

If a read operation lands on a master node while migrations are in progress, Aerospike guarantees that the copy of the most recently written record value available within the cluster will be returned. For partial writes to a record, Aerospike guarantees that the partial write will happen on the copy that eventually wins. To ensure these semantics, operations enter a *duplicate resolution* phase during migrations. While this feature provides added data correctness, it adds latency to transactions during this period. More specifically, the read and write latency for the first transaction on a record may be affected (if the record's partition has not completed migration).

Therefore, it is good to complete migrations as quickly as possible, but a migration should not be prioritized over normal read/write operations and other cluster management operations. Therefore, Aerospike contains several configuration options and performance throttles that can be applied in real-time to either speed, or delay, data migrations. These flow control systems reduce the impact of data migrations on normal application read/write workloads or can be used to improve cluster reorganization cycles in a lightly loaded cluster.

Uniform distribution of data, indexes, and transaction workload across cluster nodes make capacity planning and scale-up and scale-down decisions precise and simple for Aerospike clusters. Aerospike needs redistribution of data only on changes to cluster membership. This contrasts with alternate key range based partitioning schemes, which require redistribution of data whenever a range becomes "larger" than the capacity of a node.

Note also, that the Aerospike smart client shares the partition maps with the server by polling cluster nodes. This enables Aerospike clients to adapt quickly to changing conditions within the cluster and is another facet of Aerospike's ability to scale to larger data sizes and workloads continuously and efficiently.

4 MINIMIZING REPLICAS & MAXIMIZING AVAILABILITY

In AP-mode configuration where availability is prioritized over consistency, Aerospike allows reads and writes to continue during split-brain situations as well as in situations where the number of cluster nodes unavailable is at or more than the replication factor. This could cause inconsistencies and lost writes as shown in writes happening to item A in a split-brain scenario shown in Figure 7.

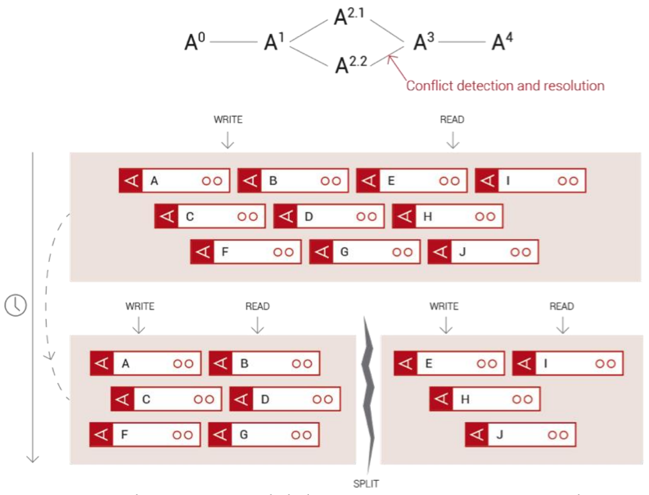


Figure 7: Choosing Availability over Consistency Results in Lost Writes

It is uncommon to violate consistency in a properly running system except during the following two scenarios:

- When the cluster splits into two or more sub-clusters that continue to take reads and writes, and
- When the cluster simultaneously loses a set of nodes that is equal to or greater than the replication factor, causing some partitions to completely disappear from the remaining cluster. Note that the failures need to be within an interval shorter than what is required for partitions to migrate to other nodes.

The required improvement for data correctness, then, is a scheme that disallows multiple masters for the same partition active at the same time. This limitation automatically limits availability in the cluster. For example, a simple scheme on a split-brain would be to avoid writes during such an event and allow only reads. This however is unnecessarily restrictive.

Most systems for providing such strong consistency require a minimum of three copies to ensure proper consistency [1]. So, if a cluster splits as shown in Figure 7, one of the two sub parts can allow writes if it has a majority (two out of three) copies of the data item. Aerospike optimizes this further by regularly storing only two copies but using an adaptive scheme that adds more write copies on the fly in situations where they are necessary, thus optimizing the performance in the normal case while incurring a small amount of overhead in edge cases that rarely occur. Note that a two-copy system still needs a minimum of three nodes to preserve availability.

Our scheme allows us to effectively have the theoretically correct result of a three-copy distributed system [1], while paying for only two copies of the data. This in turn reduces network traffic resulting from handling additional copies as well as CPU and storage costs. In clusters handling petabytes of data, this could make a huge difference in terms of hardware and operational costs.

As we shall demonstrate in the rest of this Section, achieving higher availability using such an optimized replication scheme requires a sophisticated algorithm for maintaining strong consistency.

4.1 Roster

With strong consistency configured, Aerospike defines a roster for strong consistency within a cluster. This roster is the set of nodes which are intended to be present at steady state.

When all the roster nodes are present, and all the partitions are in their correct computed location, the cluster is in its steady state and provides optimal performance. As we described in the partition algorithm earlier, the master and replica partitions are assigned to nodes in a cluster using a random assignment of partitions to nodes. In the case of strong consistency, these partitions are referred to as roster-master and roster-replica. To simplify the discussion, we will restrict ourselves to a system with replication factor set to 2. Every partition in the system will have one master and one replica. First, some terminology:

roster-replica – For a specific partition, the roster-replica refers to the nodes that would house the replicas of this partition if all nodes in the roster were part of the single cluster, i.e., the cluster was whole.

roster-master – For a specific partition, the roster-master refers to the node that would house the master of this partition if all nodes in the roster were part of the single cluster, i.e., the cluster was whole.

The following rules are now applied to the visibility of partitions:

1. If a sub cluster (e.g., split-brain) has ***both*** the roster-master and all the roster-replicas for a partition, then the partition is active for both reads and writes in that sub cluster
2. If a sub cluster contains a majority of roster nodes and has ***either*** the roster-master or a roster-replica for the partition within its component nodes, the partition is active for both reads and writes in that sub cluster. If the roster-master is not present, a roster-replica will be promoted, and other nodes will become "effective replicas"
3. If a sub cluster has ***exactly half*** of the nodes in the roster and it has the ***roster-master*** within its component nodes, the partition is active for both reads and writes. There are some further refinements of these rules later in Section 4.2.

The above rules also imply the following:

100% availability on rolling upgrade: If a sub cluster has fewer than replication factor number of nodes missing, then it is termed a ***super-majority*** sub-cluster and all partitions are active for reads/writes within the cluster

100% availability on two-way split-brain: If the system splits into exactly two sub clusters, then all partitions are active for reads and writes in one or the other sub cluster (we will later show how to use this in a creative way for a rack-aware based HA architecture)

Consider as an example, partition *p* in a 5-node cluster where node 4 is the roster-replica for p and node 5 is the roster master for p. You can see below in Figure 8, Figure 9 and Figure 10, examples of when a partition is available or not available in various network partitioning situations. Note that Figure 11 represents the state of the cluster that further split from the state depicted in Figure 10. So, the state of partition p in Nodes 3, 4 and 5 reflect this transition.

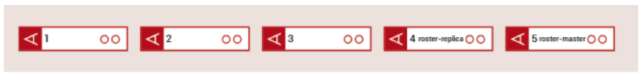


Figure 8: The cluster is whole, p is active

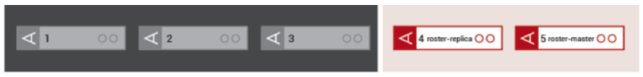


Figure 9: A minority sub-cluster with both roster-master and roster-replica, cluster is split, p is active

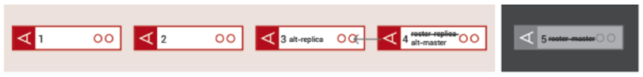


Figure 10: The roster-replica is in majority sub-cluster, promoted to master, alt replica created in node 3, cluster is split, p is active

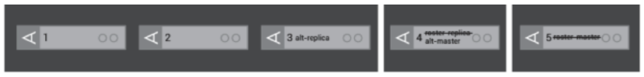


Figure 11: The roster-master and roster-replica are in minority clusters, cluster is split, p is inactive

4.2 Full partitions versus subsets

As you can see above, in steady state, partitions are considered ***full*** if they have all the relevant data. In some cases, for example in Figure 10 above where an alternate replica of the partition p was created in Node 3, the partition on node 3 is only a ***subset*** until all of the data in the partition copy on Node 4 is synchronized with Node 3. Note that Node 4 has a full copy of partition p since it split off from a fully available cluster. Certain rules must be followed for maintaining consistency. We will illustrate these using the following scenario.

In a cluster with five nodes A, B, C, D, E, let us consider partition q that has Node A as roster-master and Node B as roster-replica. Let us consider a rolling upgrade (Figure 12) where one node is taken down at a time.

- Initially Nodes A and B start out as full partitions for q.
- When Node A is taken down, Node B which is roster-replica promotes to alternate master for q and Node C becomes alternate replica for q. Node C's copy of partition q is now a subset.
- Soon enough, Node A rejoins the cluster (as subset) after the successful software upgrade and the node B now goes down for its turn to be upgraded. At this point, there has not been enough time for the roster-master A to complete synchronization of all its data with B (that was Full).
- So, we are left with Node A as roster-master that is a subset for partition q and node C that is another subset for q. At this point, because this is a super-majority cluster, we are guaranteed that among all the nodes in the cluster, all updates to the partition are available.

Therefore, we can state the following:

1. Every update must be written to at least two nodes (replication factor 2) and at most one node has been down at any one time. So, all changes must still be in one of these nodes.
2. However, what this means is that for all reads to records that go to A (roster-master) every request has to resolve itself on a record-by-record basis with the partition subset stored in node C. This will temporarily create extra overhead for reads.
3. Write overhead is never increased as Aerospike writes to both copies all the time.

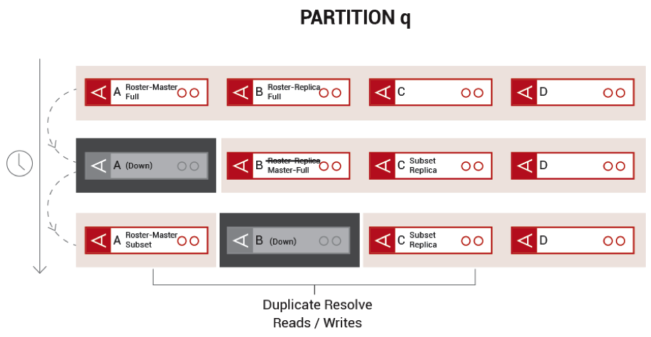


Figure 12: Subset and full partitions during rolling upgrade

Based on the above, the earlier rules are qualified further as follows:

1. If a sub cluster (e.g., split-brain) has ***both*** the roster-master and all roster-replicas for a partition, and a full partition exists within the sub cluster, then the partition is active for both reads and writes in that sub cluster.
2. If a sub cluster has a majority of nodes and has ***either*** the roster-master or any roster-replica for the partition within the component nodes and it has a full partition, then the partition is available.
3. If a sub cluster has ***exactly half*** of the nodes in the full roster and it has the ***roster-master*** within its component nodes, and it has a full copy of the partition, then the partition is active for both reads and writes.
4. If the sub cluster has a super majority (i.e., fewer nodes than replication factor are missing from the sub-cluster), then a combination of subset partitions are sufficient to make the partition active.

Note there are some special kinds of nodes that are excluded while counting the majority and super-majority conditions.

- A previously departed node rejoins the cluster with missing data (e.g., one or more empty drives).
- A node that was not cleanly shutdown is enabled.

Such nodes will have a special flag called "evade flag" set until they are properly inducted into the cluster with all the data.

While we discussed the above using replication factor 2, the algorithm extends to higher replication factors. All writes are written to every replica, so the write overhead will increase as replication factors increase beyond 2.

From the discussion here, our scheme provides equivalent level of availability with 2 copies as a traditional quorum-based system using 3 copies. The reduction continues to grow as the replication factor increases - where other systems store N replicas, Aerospike only need to store (N / 2) + 1 to achieve a similar availability level during common network failures.

4.3 Transactional consistency

4.3.1 Never Lose Writes

The write logic is shown in Figure 13. All writes are committed to every replica before the system returns success to the client. In case a write to one of the replicas fails, the master will ensure that the write is completed to the appropriate number of replicas within the cluster (or sub cluster in case the system has been compromised.)

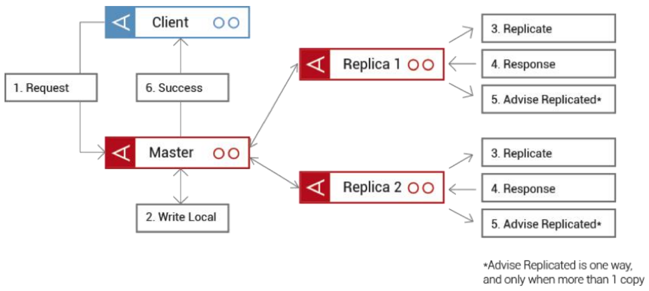


Figure 13: Write Logic

4.3.2 Strong Consistency for Reads

In the strong consistency configuration, reads are always sent to the master partition. Note that the main invariant that the client software depends on is that the server maintains the single master paradigm. However, Aerospike being a distributed system, it is possible to have a single point in time when multiple nodes think they are master for a partition. Consider for example the case where node A of a cluster is separated from the other three nodes B, C, and D. B automatically takes over for partition q and C becomes a new replica (being the next in the row for partition q in the partition mapping table), however, there may be a period where A is active but about to detect its separation and halt processing.

It is important to differentiate the versions of the partitions where the writes are being done, to properly detect an incomplete, in-process transaction from a fully committed transaction. Note that the only successful writes are those written to all the replicas. Other writes need to be detected as indeterminate or in-doubt, and the system must then resolve these subsequently by replacing them with successful writes. Also, it is only possible for exactly one sub cluster to take over as master for a partition based on rules mentioned earlier. Even in this case, it is not possible to separate out the writes that happen in a master overhang period by using just machine level timestamps – which are naturally skewed from each other. So, Aerospike has implemented a Hybrid Logical Clock [14], which includes a hybrid of three clocks. Notably, it is critical to add the concept of regime for each partition. This regime, a Lamport clock [15], is incremented every time a master handoff from node to node for a partition happens. Only the old master uses the earlier regime and all writes to the new master will use the next regime. Therefore, writes applied at a master node that has not yet processed the cluster change but unable to replicate the write to its replica(s) can result in one of two outcomes when the cluster comes back together:

1. The value written can either be discarded because the record was written in the sub-cluster excluding this node during the partitioning event.
2. The write can be rolled forward as the eventual record value in case no further write has happened in the sub cluster before the full cluster forms again.

Aerospike uses the following fields for isolating record updates:

- 40 bits of record last update time (LUT)
- 6 bits of partition regime
- 10 bits of record generation

The 6 bits of regime provides about 27 seconds of buffer based on 1.5 seconds for the heartbeat timeout window and accounts for around 32 cluster changes happening in the period. The combination of regime and LUT and record generation provides an accurate path to determine which of the records in the system hold the right value for reading and writing.

4.3.3 Linearizable operations

Based on the above, to linearize reads at the server, every read to the master partition needs to verify that the partition regimes are in sync for the partition in which the key is located. If the regimes agree, then the read is guaranteed to be current. If the regimes do not agree this means that a cluster change may be in process, and it is important to retry the read from the client. Thus, for every write, all copies of the partition being written need to also have the same regime.

4.3.4 Session/Sequential Consistency

If the occasional stale read between database clients is acceptable, radically higher performance may be achieved. In session consistency, the read from the master is all that is needed on the server-side, but the client needs to store a regime counter as part of its partition table based on the latest regime value it has encountered for a partition on its read. This ensures that the client rejects any reads from servers of an older regime than the one it has already read. This could happen due to an especially large master overhang caused by slow system behavior or suspension/slowdown of virtual machines in cloud environments, etc. This mode still maintains strong consistency, but by reading only from the master, an extra network round trip between master and replica servers is avoided.

The strong consistency scheme above guarantees the strongest possible consistency for single-record transactions while allowing 100% consistency and availability during rolling upgrades where fewer than replication factor number of nodes are taken down (Figure 12). This is particularly useful since most systems need to undergo routine maintenance for security fixes and the like. To be able to do that without any compromise to availability and consistency is extremely valuable.

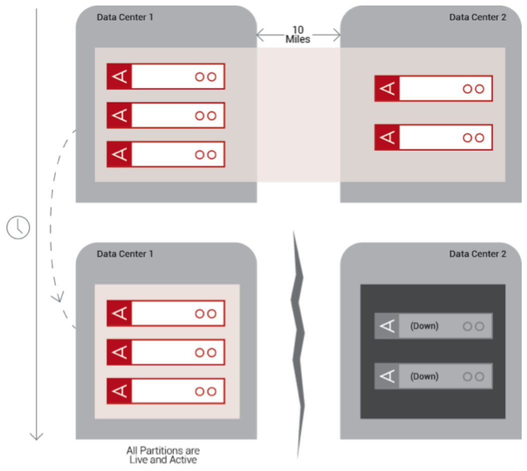


Figure 14: Rack awareness for high availability

4.4 Rack awareness for high availability

Aerospike supports a rack-aware scheme in which the various copies of a specific partition are always allocated to different racks. This can be used to setup systems to survive site failures without operator intervention (See Figure 14). In many production environments (especially in financial services where low latency and high availability are both paramount), a common installation is to have two data centers within 10 miles of each other. This enables a transaction to be committed across both data centers within a few milliseconds while still providing high fault tolerance due to the different physical location of the two data centers. Using the Aerospike strong consistency scheme, it is possible to setup systems where the outage or disconnection of an entire site will result in zero loss of data and the system can continue from there. One of these configurations is illustrated in Figure 15.

5 GEO-DISTRIBUTED TRANSACTIONS

Geo-distribution can be done using synchronous or asynchronous replication. We will describe both cases below briefly.

5.1 Synchronous active-active replication

Aerospike supports multi-site clustering where a single cluster spans multiple geographies, as shown in Figure 15. This allows users to deploy shared databases across distant sites and cloud regions with no risk of data loss. Automated failovers, high resiliency, and strong performance are the foundation of Aerospike's implementation. Two features underpin Aerospike multi-site clustering: rack awareness and strong consistency both of which were described in detail in Section 4.

Rack awareness allows replicas of data partitions to be stored on different hardware failure groups (different racks). Through replication factor settings, administrators can configure each rack to store a full copy of all data, maximizing data availability and local read performance. As we saw already in Sections 2 and 3, Aerospike evenly distributes data among all nodes within each rack.

Only one node maintains a master copy of a given data partition at any time. Other nodes (located on other racks) store replicas, which Aerospike automatically synchronizes with the master. As noted in Section 4, the roster combined with the partition map tracks the locations of masters and replicas; it also understands the racks and nodes of a healthy cluster.

In the configuration illustrated in Figure 15, each data center has one rack with three nodes, and each node has a copy of the roster. Given a replication factor of three, this example shows the roster-master copy of a data partition on Node 9 (Rack 2); replicas exist on Node 1 (Rack 1) and Node 4 (Rack 3).

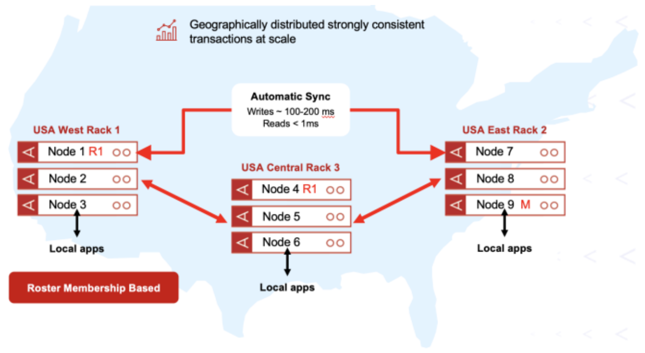


Figure 15: A multi-site cluster that spans large distances

Aerospike clients can be configured to route an application's request to read a data record to the appropriate rack/node in its local data center. In this deployment configuration (Figure 15), a full copy of the database exists in each site (rack). Therefore, by intelligently processing read requests, Aerospike can deliver sub-millisecond read latencies in this cluster during normal operation.

Writes are processed differently. For consistency across the cluster, Aerospike routes each write to the rack/node with the current master of the data. The master node ensures that the write is applied to its copy and all replicas before committing the operation. Routing writes and synchronizing replicas introduces overhead, so writes aren't as fast as reads. In the cross-continent configuration, we have observed latencies averaging between 100 to 200 milliseconds.

An Aerospike multi-site cluster follows the same rules for enforcing strong data consistency as a single-site cluster (described in Section 4), automatically taking corrective actions for most common scenarios. For example, if the roster-master becomes unavailable due to a node or network failure, Aerospike designates a new master from the available replicas and creates new replicas as needed to satisfy the replication factor. In a multi-site cluster, the new master will typically be on another rack.

Consider a scenario in which one site becomes unavailable, perhaps due to a network or power failure. Let's say that the USA East site (Rack 2) is unreachable by the rest of the cluster. Aerospike will automatically form a new sub-cluster consisting of USA West (Rack 1) and USA Central (Rack 3) to continue to service reads and writes without any operator intervention. In this degraded system, by applying the consistency rules described in Section 4 all data will be made available with complete consistency in Racks 1 and 3, while no transactions (reads or writes) will be allowed in Rack 2. Note that when Rack 2 rejoins the cluster, the partitioning schemes, clustering algorithms and strong consistency rules will ensure that the rejoining will be done smoothly with no operator intervention and eventually the cluster will return to steady state after accurately and safely merging in the changes that happened during the split-brain situation.

Note that there are split-brain situations where the system will just become either wholly or partially unavailable to preserve consistency (e.g., the case where all three sites in Figure 15 lose contact with each other simultaneously).

5.2 Asynchronous active-active replication

Cross-data replication (XDR) transparently and asynchronously replicates data between Aerospike clusters. Firms often use XDR to replicate data from Aerospike-based edge systems to a centralized Aerospike system. XDR also enables firms to support continuous operations during a crisis (such as a natural disaster) that takes down an entire cluster.

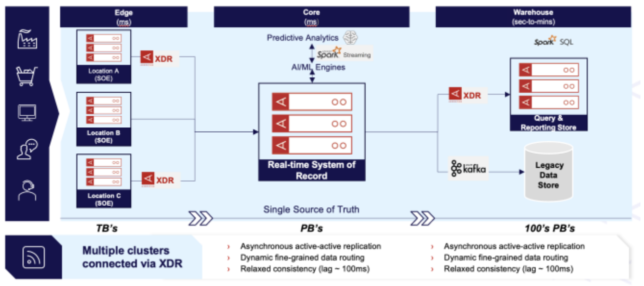

Edge (ms)
Core (ms)
Warehouse (sec-to-mins)
Location A (SOE)
Location B (SOE)
Location C (SOE)
XDR
Predictive Analytics
Spark Streaming
AI/ML Engines
Real-time System of Record
Spark SQL
Query & Reporting Store
kafka
Legacy Data Store
Single Source of Truth
TB's
PB's
100's PB's
Multiple clusters connected via XDR
Asynchronous active-active replication
Dynamic fine-grained data routing
Relaxed consistency (lag ~ 100ms)
Asynchronous active-active replication
Dynamic fine-grained data routing
Relaxed consistency (lag ~ 100ms)

Figure 16: Asynchronous replication using XDR

It is notable that the asynchronous replication scheme in Aerospike no longer uses a log-based strategy [16] as there were severe problems with keeping track of all the data items to be shipped to various destinations. For example, slow-to-reach destinations were causing delays in shipping to other destinations and shipping proceeded at the rate of the slowest destination. Given the high rate of write throughput handled, keeping separate copy of change data logs for every destination is not a solution that scales well.

The shipping algorithm in Aerospike is now based on a combination of last update time (LUT) and last ship time (LST). Aerospike keeps track of a record's digest and Last Update Time (LUT) based on write transactions in its index (typically memory resident). Additionally, Aerospike tracks a record's partition's LST. Any record in a partition whose LUT is greater than the partition's LST is a candidate for shipping. The LST is persisted by partition. If the LUT of a record is more recent than the LST of the record's partition, the record will be shipped to remote clusters through the corresponding links that are active. Once shipping has completed the partition's LST is updated.

Aerospike supports the ability to ship sub-parts of changed records – a bin (or column) – and implements a convergence feature which can resolve write conflicts in active-active topologies. This feature makes sure that the data is eventually the same in all the connected sites at the end of replication even if there are simultaneous updates to the same record in multiple sites. To achieve this, extra information about each bin's LUT is stored and used appropriately.

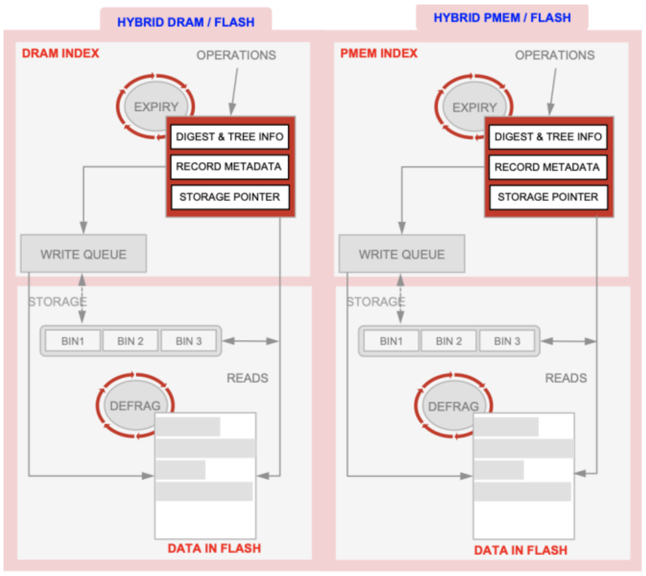


Figure 17: Hybrid Memory Architecture (DRAM and PMEM)

A typical setup of a globally distributed deployment with asynchronous replication is illustrated in Figure 16. As you can see there can be many globally distributed sites communicating to each other in complex topologies, supporting both active-active and active-passive deployments. The absence of a manifested digest log and the ability to ship sub-parts means that the asynchronous replication in Aerospike can be used in complex deployment topologies without concern about varying shipping speeds across multiple destinations with different network characteristics.

6 STORAGE OPTIMIZATIONS

6.1 Hybrid Memory Architecture with DRAM

A key component of the storage model in Aerospike is based on a Hybrid Memory Architecture (HMA) where data resides in flash storage (SSD), and indexes reside entirely in DRAM (or PMEM). In this hybrid DRAM/Flash configuration (Figure 17), no disk I/O is required to traverse the primary index, followed by a single lookup of the data record from flash storage. Such a design can keep read latency low at high throughput because the characteristic of I/O in NAND Flash has little penalty for random access reads.

Any database access must traverse the index tree, acquire metadata such as versions or sequence numbers as illustrated in Figure 17. A data element in a local cache can be returned without I/O access, or a single I/O will be executed to bring the entire element into local memory. Writes need to be propagated to replicas within the database cluster. Replica writes can be synchronous or asynchronous, depending on the level of durability required.

Note that the fundamental parallelism in HMA for random-access reads eliminates the need for caching that is used in traditional buffer-pool based systems (see Figure 18). The read latency is essentially dependent on the access time to SSD and we can consistently achieve a few hundred microseconds on high quality NVMe drives. For addressing wear leveling issues in SSDs Aerospike implements a log structured mechanism using large block writes with defragmentation to reclaim storage (Figure 19).

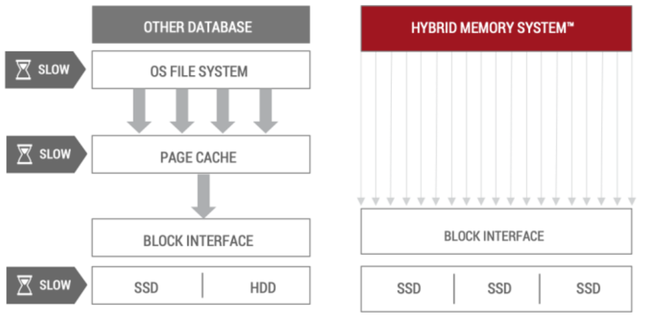


Figure 18: HMA versus traditional buffer-pool based DB access

The techniques used here are able to support extremely high application write rates while maintaining DRAM-class read response times with data correctness. However, this comes with a cost, namely rebuilding DRAM indexes is expensive (takes as much time as scanning all the data in storage) and committing every transaction to storage device is also unacceptably expensive. By combining large block writes, parallelized access to multiple SSDs, a native storage file system with direct device access that bypasses the operating system's file system, this architecture can deliver high throughput reads and writes with low latency as seen in Section 8.

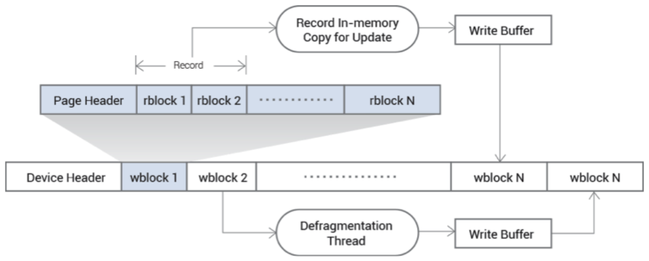


Figure 19: Aerospike write architecture to SSD

Avoiding index rebuild on large nodes

To avoid rebuilding the primary index on every process restart, the index can be stored in a shared memory space disjoint from the service process's memory space. In case maintenance only requires a restart of the database service, the index need not be reloaded. The database daemon simply attaches to the current copy of the index in shared memory and is ready to handle transactions. This form of service starts re-using an existing index is termed '*fast start'*; it eliminates scanning the device to rebuild the index. However, a cold start of the node will still need to rebuild the index from scratch but a tool can be used to dump the index on disk before process shutdown (after quiescing) so it can be populated on restart.

6.2 Hybrid Memory Architecture with PMem

One technology that helps mitigate both the slow restart and commit to device problems is Intel® Optane™ DC persistent memory [19], based on Intel® 3D XPoint™, a new class of storage technology architected specifically for data-intensive applications. Note that this technology is being phased out, but new alternatives are expected to emerge over the next few years.

The tiered architecture, using this technology's AppDirect mode, allows multiple deployment environments to use this as the primary key index layer, where its high performance and parallelism work best. By using a persistence layer for indexes, full restarts of Aerospike are possible without primary index rebuilds. In tests performed by Intel (shown in Figure 20), we notice that performance is nearly identical between PMem indexes (using the native PMem implementation) and DRAM indexes. Both achieved a million transactions per second per server.

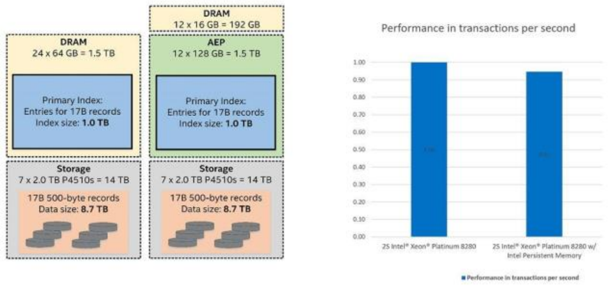


Figure 20: Throughput comparison DRAM versus PMem

7 CPU AND NETWORK PERFORMANCE

CPU and network performance can help to handle high ingestion rates using tech like Intel® Ethernet 800 Series with Application Device Queues (ADQ) [12].

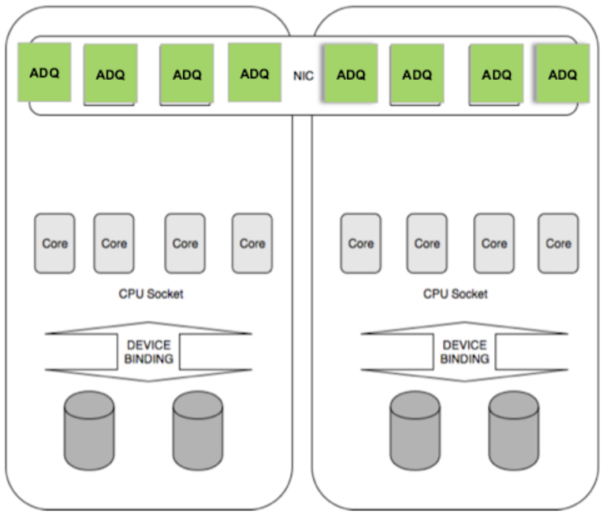


Figure 21: Device queue & CPU core alignment

NICs are meant to separate related network packets into separate device queues to make processing more efficient by keeping high-priority traffic from getting stuck behind slow requests. ADQ sets the bar higher by letting applications define tailored rules (Traffic Classes) for sorting packets into device queues. The 800 Series NIC directly sorts packets using these rules, completely offloading the host processor.

The workload involves many clients sending database requests to the database server. These are dispatched in parallel to many service threads. All requests are treated equally, so the ADQ strategy is to spread network traffic equally to all the CPU cores in the system. One device queue has been defined for each CPU core in the system and each queue is configured to generate service interrupts on a particular CPU. The Traffic Classes also route all requests from a given client to the same device queue, as shown in Figure 21. Aligning device queues and CPU cores reduces context-switching overhead. Better still, it keeps data in local processor caches. Together these factors contribute to lower latency, more predictable response times and higher throughput. Busy polling was also employed to increase performance to reduce interrupts by servicing all packets that have accumulated. A polling interval of 50 milliseconds was found to give optimal results because every millisecond matters.

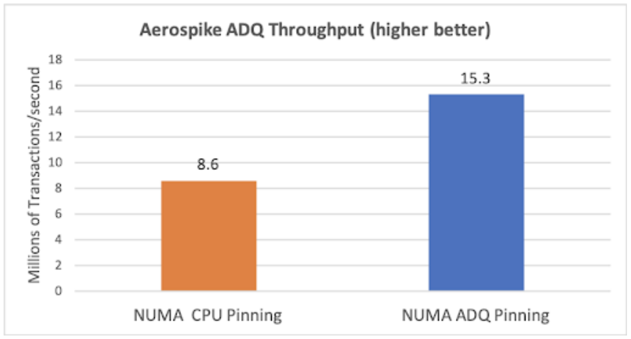


Figure 22: ADQ Throughput

ADQ performance was measured in a test environment comprising one dual-socket, dual-NIC server and six clients, all connected to a 100G switch. Performance data were gathered by running 18 instances of the Aerospike C language benchmark client (3 per client node) against the server. The configuration used is as follows:

Intel^R Xeon^R Xeon Platinum 8280 Server (2.7GHz, 28 cores)
768 GB total DRAM
2 Intel^R Ethernet 800 Series 100G NIC cards
2 Aerospike 4.7 servers (NUMA configuration with CPU/ADQ pinning)
6 Intel^R Xeon^R E5-2699 v4 clients (2.2 GHz, 22 cores)
125 GB total DRAM
Single Intel^R Ethernet 700 Series 40G NIC card
3 instances of Aerospike C benchmark (async mode) per client

Relative performance was measured by comparing ADQ NUMA pinning with the baseline case of CPU pinning. When enabling ADQ the performance was recorded at 15.3M transactions/sec (>75% improvement), as shown in Figure 22. Regarding latency, 99% of the requests were below 320 µsecs (>45% improvement in response time predictability.

8 CLOUD PERFORMANCE RESULTS

The techniques presented here have been extensively validated in scores of deployments at scale across multiple industries over the past decade. To illustrate the benefits, we will briefly share the results of a petabyte cloud benchmark that was run on a 20-node AWS [38] cluster with Aerospike. Each EC2 i3en.24xlarge node featured 768 GB of DRAM and 8 x 7500 NVMe SSDs. For clients, the benchmark employed 40 AWS EC2 c5n.9xlarge nodes with 96 GB DRAM each and EBS-only storage; operations were executed using Aerospike's C client.

The Aerospike server was configured with compression that yielded a 4 times reduction in size for the user profile database, causing it to store 500 TB of compressed user data (250 TB of unique user data and 250 TB of replicated data) with ½ trillion unique keys with replication factor 2.

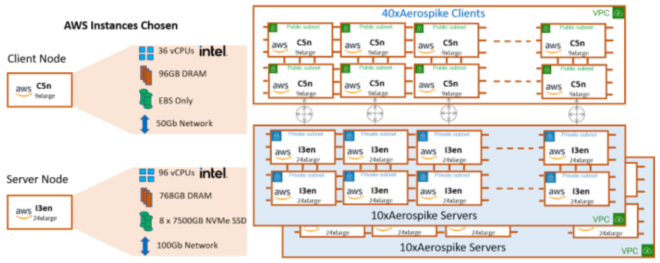


Figure 23: Petabyte cloud benchmark configuration

Table 2: Benchmark results

Test #	Data	Workload	TPS	Latency < 1ms
1	User Profile (1PB unique, uncompressed)	Read only	5,009,980 reads	100%
2		80/20 read/write	3,017,340 reads 754,160 writes	100% 99%

Per the results of Test 1 in Table 2, Aerospike processed more than 5 million read-only TPS with sub-millisecond latencies for user profile applications. Test 2 featured an 80/20 mix of read/write mix operations run against the user profile database of ½ trillion unique records. Under those conditions, Aerospike delivered more than 3.7 million TPS for user profile applications, nearly all with sub-millisecond latencies.

9 CONCLUSION

As validated by the above results, the techniques we have developed generate enormous efficiencies of scale (e.g., delivering millions of TPS on a 20-node 1PB, trillion object, cluster on public cloud). These efficiencies go a long way in keeping the cost per transaction low enough that handling of tens of billions of transactions per day has become routine in Aerospike use cases like real-time bidding, fraud detection for financial transactions, real-time instant payments, e-commerce, gaming, and others.

In terms of the future, we are working to enhance our system with multi-record transaction capabilities and improve integration with the parallel processing capabilities of Apache Spark as well as federated SQL queries of Presto/Trino. By doing so, we aim to use the techniques and efficiencies described here for both high throughput transactions and analysis of extremely large datasets within real-time SLAs while maintaining low total cost of ownership, thus ensuring that such capabilities are available to every small, medium, and large enterprise in the world.

ACKNOWLEDGMENTS

We thank Ginger Gilsdorf, Tibor Szaboky, and Cindy Saracco for their contributions to the performance results described in this paper.

REFERENCES

[1] Leslie Lamport. Lower Bounds for Asynchronous Consensus. Microsoft Research, Microsoft Corporation, MSR-TR-2004-72 (2006).

[2] Michael Stonebraker. SQL databases v. NoSQL databases. *Communications of the ACM,* 53, 4 (April 2010), 10-11. DOI: https://doi.org/10.1145/1721654.1721659

[3] Mikhail Kourjanski. ML Data Pipelines for Real-Time Fraud Prevention @ PayPal. QCon 2018. https://www.infoq.com/presentations/paypal-ml-fraud-prevention-2018/

[4] Michael Stonebraker. New opportunities for New SQL. *Communications of the ACM,* 55, 11 (November 2012), 10-11. DOI: https://doi.org/10.1145/2366316.2366319

[5] TIPS. European Central Bank Website. https://www.ecb.europa.eu/paym/target/tips/html/index.en.html

[6] Intel® Optane Technology, *Intel Website.* https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[7] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, Thomas Lopatic. Aerospike: Architecture of a Real-Time Operational DBMS. *Proceedings of the VLDB Endowment,* Vol. 9, No. 13 (2016).

[8] Florian Mendel, Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. On the collision resistance of RIPEMD-160. *Proceedings of the 9th international conference on Information Security*, (2006).

[9] Kyle Kingsbury, Aerospike 3.99.0.3 Jepsen report, March 7, 2018, http://jepsen.io/analyses/aerospike-3-99-0-3

[10] Eric Brewer, CAP Theorem, *Wikipedia* https://en.wikipedia.org/wiki/CAP_theorem

[11] Harmeen Mehta, Customer 360: Powering Airtel's "Digital Brain" for Personalized Customer Engagement, *Aerospike Summit*, 2019. https://www.aerospike.com/resources/videos/summit19/ty-airtel/

[12] Intel® Ethernet 800 Series with Application Device Queues (ADQ), *Intel Website.* https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/application-device-queues-technology-brief.html

[13] Intel® Optane SSD DC P4800X Series NVMe, *Intel Website.* https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html

[14] Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa, Bharadwaj Avva, and Marcelo Leone, Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases. University of Buffalo, Tech report, 2014-04. https://cse.buffalo.edu/tech-reports/2014-04.pdf

[15] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM,* 21, 7 (July 1978), 558-565.

[16] Donald. J. Haderle & Cindy. M. Saracco, (2013). The History and Growth of IBM's DB2. IEEE Annals of the History of Computing, Annals of the History of Computing, IEEE, IEEE Annals Hist. Comput, 35(2), 54ñ66. https://doi-org.proxy1.ncu.edu/10.1109/MAHC.2012.55

[17] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al., Cockroachdb: The resilient geo-distributed sql database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 1493–1509.

[18] YugabyteDB. URL https://www.yugabyte.com

[19] Intel® Optane Technology, *Intel Website.* https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[20] Caleb Henry, (2020). SpaceX Launches 58 Starlink Satellites, Three Planet SkySats on Falcon 9. URL https://spacenews.com/spacex-launches-58-starlink-satellites-three-planet-skysats-on-falcon-9/

[21] Apache Spark Website. https://spark.apache.org

[22] Qualcomm 5G Website. https://www.qualcomm.com/research/5g

[23] Steven Vaughan-Nichols, (2020). SpaceX Starlink Internet Prepares for Beta Users. URL https://www.zdnet.com/article/spacex-starlink-internet-prepares-for-beta-users/

[24] Clark Fredricksen, Jaganath Achari and Supratibh Srivastava. Running Ad Tech Workloads with Aerospike at Petabyte Scale. URL https://aws.amazon.com/blogs/industries/running-ad-tech-workloads-on-aws-with-aerospike-at-petabyte-scale/

[25] Theresa Melvin. HPE: AI at Hyperscale – How to go faster with a smaller footprint. Aerospike Summit 2019. URL https://aerospike.com/resources/videos/summit19/ty-hpe/

[26] Jason Yanowitz. Signal: Rebuilding on a Strong Foundation: from Cassandra to Aerospike, One Year On. Aerospike Summit 2019. URL https://aerospike.com/resources/videos/summit19/ty-signal/

[27] Matthias Baumhof, Nick Blievers. Replacing Cassandra: A Digital Transformation for the World's Largest Digital Identity Network. Aerospike Summit 2018. URL https://aerospike.com/resources/videos/replacing-cassandra-a-digital-transformation-for-the-worlds-largest-digital-identity-network/

[28] Mikhail Kourjanski. The Science Behind Delightful User Experience. Aerospike User Summit 2018. https://aero-media.aerospike.com/2018/05/1000.E-Science-Behind-Delightful-User-Experience-PayPal.pdf

[29] Apache Cassandra Website. URL https://cassandra.apache.org/_/index.html

[30] Non-uniform memory access. Wikipedia. URL https://en.wikipedia.org/wiki/Non-uniform_memory_access

[31] Dennis Padia. SAP HANA Persistent Memory using Intel's DCPM or IBM's vPMem. URL https://blogs.sap.com/2020/01/24/sap-hana-of-persistent-memory-using-intels-dcpm-or-ibms-vPMem/

[32] Terracotta distributed cache. URL https://www.terracotta.org/

[33] Couchbase developer site. URL https://developer.couchbase.com/

[34] Planning and testing DataStax enterprise deployments. URL https://docs.datastax.com/en/dse-planning/doc/planning/capacityPlanning.html

[35] Oracle RAC. URL https://www.oracle.com/database/real-application-clusters/

[36] PrestoDB. URL https://prestodb.io/

[37] Trino (formerly PrestoSQL). URL https://trino.io/

[38] Amazon Web Services. URL https://aws.amazon.com/

[39] Yahoo Cloud Serving Benchmark. URL https://github.com/brianfrankcooper/YCSB

[40] Jenkins's one-at-a-time hash, https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time

[41] FNV-1a hash, https://en.wikipedia.org/wiki/Fowler–Noll–Vo_hash_function#FNV-1a_hash

[42] Srinivasan, V. & Bulkowski, B., Citrusleaf: A Real-Time NoSQL DB which Preserves ACID., PVLDB 4, (2012).