Aerospike: Architecture of a Real-Time Operational DBMS

V. Srinivasan
Sunil Sayyaparaju
Ashish Shinde

Brian Bulkowski
Andrew Gooding
Thomas Lopatic

Wei-Ling Chu
Rajkumar Iyer

Aerospike, Inc.
vldb2016@aerospike.com

ABSTRACT

In this paper, we describe the solutions developed to address key technical challenges encountered while building a distributed database system that can smoothly handle demanding real-time workloads and provide a high level of fault tolerance. Specifically, we describe schemes for the efficient clustering and data partitioning for the automatic scale out of processing across multiple nodes and for optimizing the usage of CPUs, DRAM, SSDs and networks to efficiently scale up performance on one node.

The techniques described here were used to develop Aerospike (formerly Citrusleaf), a high performance distributed database system built to handle the needs of today's interactive online services. Most real-time decision systems that use Aerospike require very high scale and need to make decisions within a strict SLA by reading from, and writing to, a database containing billions of data items at a rate of millions of operations per second with sub-millisecond latency. For over five years, Aerospike has been continuously used in over a hundred successful production deployments, as many enterprises have discovered that it can substantially enhance their user experience.

1. INTRODUCTION

Real-time Internet applications typically require very high scale; they also need to make decisions within a strict SLA. This typically requires these applications to read from, and write to, a database containing billions of data items at a rate of millions of operations per second with sub-millisecond latency. Therefore, such applications require extremely high throughput, low latency and high uptime. Furthermore, such real-time decision systems have a tendency to increase their data usage over time to improve the quality of their decisions, i.e., the more data can be accessed in a fixed amount of time, the better the decision becomes.

The original need for such systems originated in Internet advertising technology that uses real-time bidding [27]. The Internet advertising ecosystem has evolved with many different players interacting with each other in real time to provide the correct advertisement to a user, based on that user's behavior. You can see the basic architecture of the ecosystem illustrated in Figure 1.

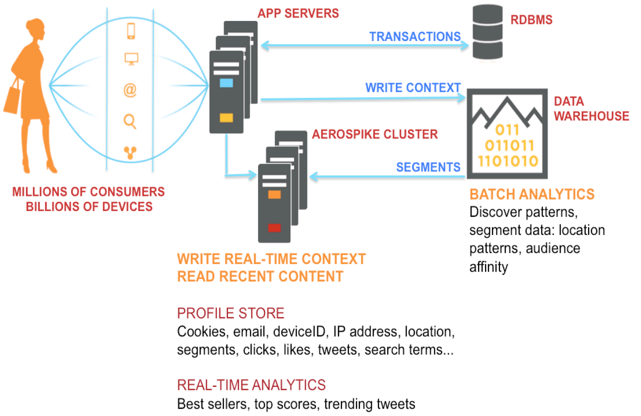


Figure 1: RTB technology stack

In order to participate in the real-time bidding [22] process, every participant in this ecosystem needs to have a high-performance read-write database with the following characteristics:

- Sub-millisecond database access times to support an SLA of 50ms for real-time bidding and 100ms for rendering the ad itself
- Extremely high throughput of 50/50 read-write load, e.g., 3 to 5 million operations/second for North America alone
- Database with billions of objects each with sizes between 1KB and 100KB, for a total DB size of 10-100TB
- Fault-tolerant service that can handle these mission-critical interactions for revenue generation with close to 100% uptime
- Global data replication across distributed data centers for providing business continuity during catastrophic failures

As has been the case in the Internet industry for a while now, recently, traditional enterprises have also experienced a huge increase in their need for real-time decision systems. Here are a few examples:

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

In Financial Services, the recent explosion in mobile access to applications has increased the load on customer facing applications by an order of magnitude. This requires enterprises to shield traditional DBMSs by offloading the new workload to a high-performance read-write database while still maintaining the original data (and a few core applications) in the traditional DBMS.

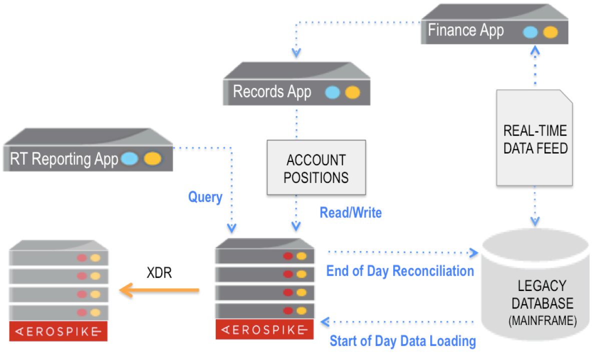


Figure 2: Financial services technology stack

As shown in Figure 2, a database like Aerospike is used as the system of record during the trading period while compliance related applications still run on the master DBMS.

In the electronic payments world, fraud detection requires a sophisticated rules-based decision engine that decides whether or not to approve a transaction based on the customer's past purchases (i.e., transaction history), and the kind of purchase they are making right now, from which device, and to which payee. All required data sets reside in a high performance DBMS that can support real-time read-write access from fraud detection algorithms, as shown in Figure 3.

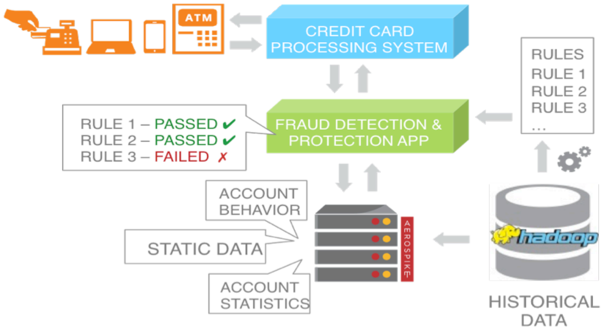


Figure 3: Fraud detection technology stack

Previously, Telecommunication Providers (Telcos) had been using home grown real-time billing systems for tracking voice calls. However, tracking the mobile data traffic that is dominating their network today creates additional load that is several orders of magnitude higher than the previous load for tracking voice traffic. The typical use case here involves using a very fast in-memory database at the edge of the network to monitor traffic patterns and generate billing events. Changes to a user's data plans are immediately reflected in the switches that direct traffic to and from devices, as shown in Figure 4.

Based on the above, it is clear that real-time applications originally pioneered in the Internet industry are crossing over into traditional Enterprises in a big way.

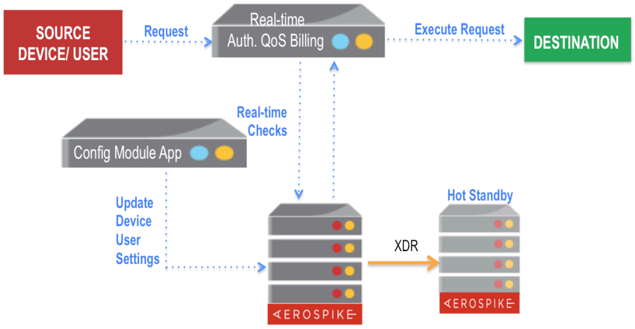


Figure 4: Telco technology stack

In these systems, a small number of business transactions (e.g., 500 to 1000 per second) typically result in a hundred-fold (or even thousand-fold) increase in database operations, as shown in Figure 5. The decision engine typically uses sophisticated algorithms to combine the real-time state (of user, device, etc.) with applicable insights to decide on a suitable action within a short amount of time (typically 50-250 milliseconds).

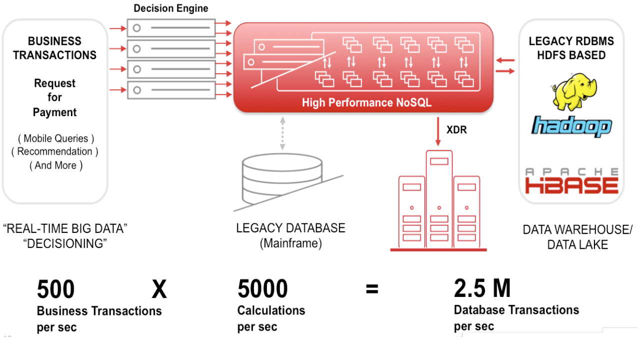

Decision Engine
BUSINESS TRANSACTIONS
Request for Payment
(Mobile Queries)
(Recommendation)
(And More)
High Performance NoSQL
XDR
LEGACY RDBMS HDFS BASED
hadoop
APACHE HBASE
"REAL-TIME BIG DATA" "DECISIONING"
LEGACY DATABASE (Mainframe)
DATA WAREHOUSE/ DATA LAKE
500 Business Transactions per sec
X
5000 Calculations per sec
=
2.5 M Database Transactions per sec

Figure 5: Real-time decision engine

An important commonality observed in these use cases is that each is mission-critical and requires much higher performance at scale than that provided by previous generations of operational database systems. We will describe in the subsequent sections the key technical aspects of next-generation operational DBMSs needed to deliver such high scale, real-time processing to all enterprises.

The rest of the paper is organized as follows. Section 2 describes the distributed system architecture and addresses issues related to scale-out under the sub-topics of cluster management, data distribution and client/server interaction. Section 3 gives a brief overview of geographical replication. Section 4 talks about system level optimization and tuning techniques to achieve the highest levels of scale-up possible. Section 5 talks about storage architecture and how it leverages SSD technology. Section 6 presents some cloud benchmark results. Finally, Section 7 presents our conclusion.

2. AEROSPIKE ARCHITECTURE

The Aerospike database platform (Figure 6) is modeled on the classic shared-nothing database architecture [25]. The database cluster consists of a set of commodity server nodes, each of which has CPUs, DRAMs, rotational disks (HDDs) and optional flash storage units (SSDs). These nodes are connected to each other using a standard TCP/IP network.

Client applications issue primary index based read/write/batch operations and secondary-index based queries against the cluster via client libraries that provide a native language interface idiomatic to each language. Client libraries are available for popular programming languages, viz. Java, C/C++, Python, PHP, Ruby, Go, JavaScript and C#.

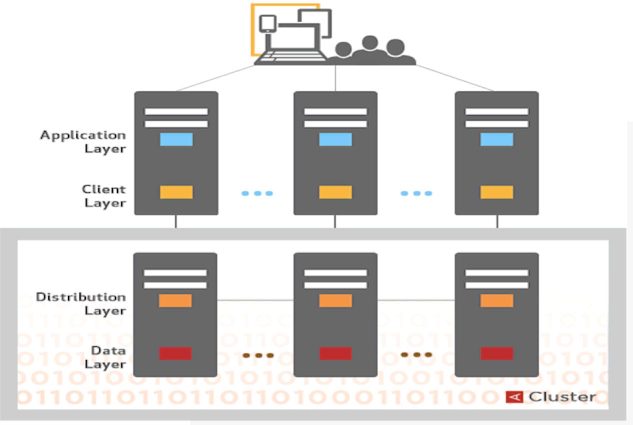


Figure 6: Aerospike architecture

2.1 Cluster Management

The cluster management subsystem handles node membership and ensures that all the nodes in the system come to a consensus on the current membership of the cluster. Events such as network faults and node arrival or departure trigger cluster membership changes. Such events can be both planned and unplanned. Examples of such events include randomly occurring network disruptions, scheduled capacity increments, and hardware/software upgrades.

The specific objectives of the cluster management subsystem are:

- Arrive at a single consistent view of current cluster members across all nodes in the cluster.
- Automatically detect new node arrival/departure and seamless cluster reconfiguration.
- Detect network faults and be resilient to such network flakiness.
- Minimize time to detect and adapt to cluster membership changes.

2.1.1 Cluster View

Each Aerospike node is automatically assigned a unique node identifier, which is a function of its MAC address and of the listening port. Cluster view is defined by the tuple: <*cluster_key, succession_list*> where,

- *cluster_key* is a randomly generated 8-byte value that identifies an instance of the cluster view.
- *succession_list* is the set of unique node identifiers that are part of the cluster.

The cluster key uniquely identifies the current cluster membership state, and changes every time the cluster view changes. It enables Aerospike nodes to differentiate between two cluster views with an identical set of member nodes.

Every change to the cluster view has a significant effect on operation latency and, in general, on the performance of the entire system. This means there is a need to quickly detect node arrival/departure events, and subsequently, for an efficient consensus mechanism to handle any changes to the cluster view.

2.1.2 Cluster Discovery

Node arrival or departure is detected via heartbeat messages exchanged periodically between nodes. Every node in the cluster maintains an adjacency list, which is the list of other nodes that have recently sent heartbeat messages to this node. Nodes departing the cluster are detected by the absence of heartbeat messages for a configurable timeout interval; after this, they are removed from the adjacency list.

The main objectives of the detection mechanism are:

- To avoid declaring nodes as departed because of sporadic and momentary network glitches.
- To prevent an erratic node from frequently joining and departing from the cluster. A node could behave erratically due to system level resource bottlenecks in the use of CPU, network, disk, etc.

The following sections describe how the aforementioned objectives are achieved:

2.1.2.1 Surrogate heartbeats

In the flaky or choked network, it is possible to arbitrarily lose certain packets. Therefore, in addition to regular heartbeat messages, nodes use other messages that are regularly exchanged between nodes as an alternative secondary heartbeat mechanism. For instance, replica writes are used as a surrogate for heartbeat messages. This ensures that, as long as either the primary or secondary heartbeat communication between nodes is intact, network flakiness on the primary heartbeat channel alone will not affect the cluster view.

2.1.2.2 Node Health Score

Every node in the cluster evaluates the health score of each of its neighboring nodes by computing the average message loss, which is an estimate of how many incoming messages from that node are lost. This is computed periodically as a weighted moving average of the expected number of messages received per node versus the actual number of messages received per node, as follows.

Let **t** be the heartbeat messages transmit interval, **w** be the length of the sliding window over which average is computed, **r** be the number of heartbeat messages received in this window, $\mathbf{l_w}$ be the fraction of messages lost in this window, **α** be a smoothing factor and $\mathbf{l_{a(prev)}}$ be the average message loss computed until now. $\mathbf{l_{a(new)}}$, the updated average loss, is then computed as follows:

$$l_w = \textit{messages lost in window} \;/\; \textit{messages expected in window}$$
$$= (w * t - r) / (w * t)$$
$$l_{a(new)} = (\alpha * l_{a(prev)}) + (1 - \alpha) * l_w$$

A node whose average message loss exceeds twice the standard deviation across all nodes is an outlier and deemed unhealthy. An erratically behaving node typically has a high average message loss and also deviates significantly from the average node behavior. If an unhealthy node is a member of the cluster, it is removed from the cluster. If it is not a member, it is not considered for membership until its average message loss falls within tolerable limits. In practice, **α** is set to 0.95, giving more weightage to average value over recent ones. The window length is 1000ms.

2.1.3 Cluster View Change

Changes to the adjacency list, as described in Section 2.1.2, trigger a run of the Paxos consensus algorithm [20] that arrives at the new cluster view. A node that sees its node identifier as the highest in its adjacency list acts as a Paxos proposer and assumes the role of the Principal. The Paxos Principal then proposes a new cluster view. If the proposal is accepted, nodes begin redistribution of the data to maintain uniform data distribution across the new set of cluster nodes. A successful Paxos round takes 3 network round trips to converge, assuming there are no opposing proposals.

The Aerospike implementation works to minimize the number of transitions the cluster would undergo as an effect of a single fault event. For example, a faulty network switch could make a subset of the cluster members unreachable. Once the network is restored, there would be a need to add these nodes back to the cluster.

If each lost or arriving node triggers the creation of a new cluster view, the number of cluster transitions would equal the number of nodes lost or added. To minimize such transitions, which are fairly expensive in terms of time and resources, nodes make cluster change decisions only at the start of fixed cluster change intervals (the time of the interval itself is configurable). The idea here is to avoid reacting too quickly to node arrival and departure events, as detected by the heartbeat subsystem, and instead, process a batch of adjacent node events with a single cluster view change. This avoids a lot of potential overhead caused by duplicate cluster view changes and data distributions. A cluster change interval equal to twice the timeout value of a node ensures that all nodes failing due to a single network fault are definitely detected in a single interval. It will also handle multiple fault events that occur within a single interval.

Aerospike's cluster management scheme allows for multiple node additions or removals at a time. This provides an advantage over schemes that require node additions to occur one node at a time. With Aerospike, the cluster can immediately be scaled out to handle spikes in load, without downtime.

2.2 Data Distribution

Aerospike distributes data across nodes as shown in Figure 7. A record's primary key is hashed into a 160-byte digest using the RipeMD160 algorithm, which is extremely robust against collisions [12]. The digest space is partitioned into 4096 non-overlapping 'partitions'. It is the smallest unit of data ownership in Aerospike. Records are assigned partitions based on the primary key 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. This data-partitioning scheme is unique to Aerospike and it significantly contributes to avoiding the creation of hotspots during data access, which helps achieve high levels of scale and fault tolerance.

Aerospike colocates indexes and data to avoid any cross-node traffic when running read operations or queries. Writes may require communication between multiple nodes based on the replication factor. Colocation of index and data, when combined with a robust data distribution hash function, results in uniformity of data distribution across nodes. This, in turn, ensures that:

1. Application workload is uniformly distributed across the cluster,
2. Performance of database operations is predictable,
3. Scaling the cluster up and down is easy, and
4. Live cluster reconfiguration and subsequent data rebalancing is simple, non-disruptive and efficient.

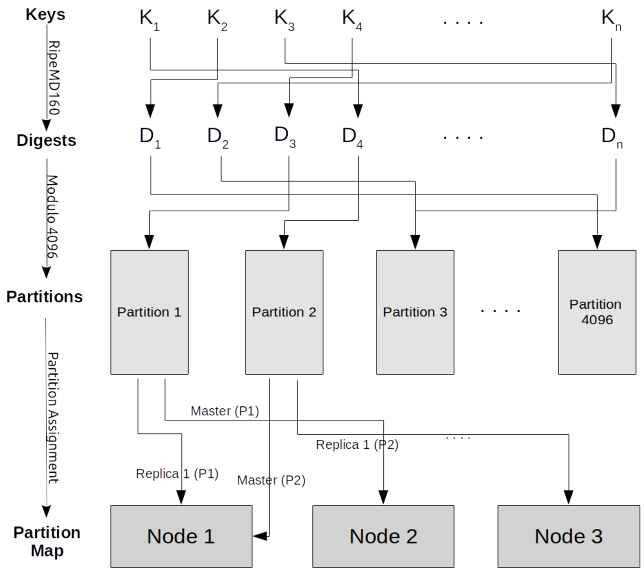


Figure 7: Data distribution

A partition assignment algorithm generates a replication list for every partition. The replication list is a permutation of the cluster succession list. The first node in the partition's replication list is the master for that partition, the second node is the first replica, the third node is the second replica, and so on. The result of partition assignment is called a partition map. Also note that, in a well-formed cluster, there is only one master for a partition at any given time. By default, all the read/write traffic is directed toward master nodes. Reads can also be uniformly spread across all the replicas via a runtime configuration setting. Aerospike supports any number of copies, from a single copy to as many copies as there are nodes in the cluster.

The partition assignment algorithm has the following objectives:

1. Be deterministic so that each node in the distributed system can independently compute the same partition map,
2. Achieve uniform distribution of master partitions and replica partitions across all nodes in the cluster, and
3. Minimize movement of partitions during cluster view changes.

The algorithm is described as pseudo code in

Table 1 and is deterministic, achieving objective 1. The heart of the assignment is the NODE_HAS_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 [19] hash on the FNV-1a [13] hashes of the node and partition ids gives a fairly good distribution and achieves objective 2 as well.

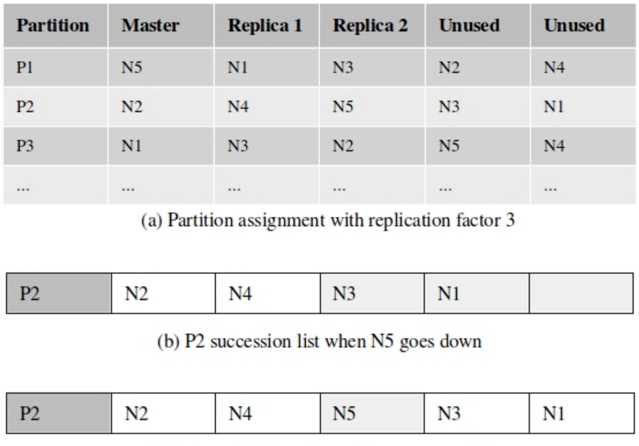
Figure 8(a) shows the partition assignment for a 5-node cluster with a replication factor of 3. Only the first three columns (equal to the replication factor) in the partition map are used; the last two columns are unused.

Table 1: Partition assignment algorithm

```
function REPLICATION_LIST_ASSIGN(partitionid)
  node_hash = empty map
  for  nodeid in succession_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>)
```

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 8(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 8(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.



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 8: Master/replica assignment

The discussion above gives an idea of the way in which the algorithm minimizes the movement of partitions (a.k.a. migrations) during cluster reconfiguration. Thus the assignment scheme achieves objective 3.

When a node is removed and rejoins the cluster, it would have missed out on all transactions applied while it was away and would need to catch up. Alternatively, when a brand new node joins a running cluster with lots of existing data, and happens to own a replica or master copy of a partition, the new node needs to obtain the latest copy of all the records in that partition and to also be able to handle new read and write operations. The mechanisms by which these issues are handled are described below in section 2.2.1.

2.2.1 Data Migrations

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. Once consensus is reached on a new cluster view, all the nodes in a cluster run the distributed partition assignment algorithm and assign the master and one or more replica nodes to each of the partitions.

The master node of each partition assigns a unique partition version to that partition. This version number is copied over to the replicas. After a cluster view change, the partition versions for every partition with data are exchanged between the nodes. Each node thus knows the version numbers for every copy of the partition.

2.2.1.1 Delta-Migrations

Aerospike uses a few strategies to optimize migrations by reducing the effort and time they take, as follows.

Define a notion of partition ordering using versions that helps 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 easy if a total order could be established over partition versions. For example, if the value of a partition's version on node 1 is less than the value of the same partition's version on node 2, the partition version on node 1 could be discarded as obsolete. 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 time. Aerospike maintains this partition lineage up to certain degree.

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 partition version order. In other cases, like rolling upgrades, the delta of changes may be small and could be shipped over and reconciled instead of shipping the entire partition content.

2.2.1.2 Operations During Migrations

If a read operation lands on a master node while migrations are in progress, Aerospike guarantees that the copy of the record that eventually wins 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. During duplicate resolution, the node containing the master copy of the partition for a specific record reads the record across all its partition versions and resolves to one copy of the record (the latest). This is the winning copy and it is henceforth used for the read or write transaction.

2.2.1.3 Master Partition Without Data

An empty node newly added to a running cluster will be master for a proportional fraction of the partitions and have no data for those partitions. A copy of the partition without any data is marked to be in a DESYNC state. All read and write requests on a partition in DESYNC state will necessarily involve duplicate resolution since it has no records. One of Aerospike's optimizations involves electing the partition version with the highest number of records as the acting master for this partition. All reads are directed to the acting master. If the client applications are satisfied with reading older versions of records, duplicate resolution on reads can be turned off. Thus, read requests for records present on the acting master will not require duplicate resolution and have nominal latencies. This acting master assignment only lasts until migration is complete for this partition.

2.2.1.4 Migration Ordering

Clearly, duplicate resolution adds to the latency when migrations are ongoing in the cluster. Therefore, it is important to complete migrations as quickly as possible. However, a migration cannot be prioritized over normal read/write operations and other cluster management operations. Given this constraint, Aerospike applies a couple of heuristics to reduce the impact of data migrations on normal application read/write workloads.

2.2.1.4.1 Smallest Partition First

Migration is coordinated in such a manner as to let nodes with the fewest records in their partition versions start migration first. This strategy quickly reduces the number of different copies of a specific partition, and does this faster than any other strategy. This implies that duplicate resolution would need to talk to a fewer number of nodes over time as smaller sized versions finish migration first and latency improves as migrations complete.

2.2.1.4.2 Hottest Partition First

At times, client accesses are skewed to a very small number of keys from the key space. Therefore the latency on these accesses could be improved quickly by migrating these hot partitions before other partitions, thus reducing the time spent in duplicate resolution.

2.2.2 Scheduled Maintenance

Node restarts for maintenance, though not very frequent, are unavoidable. The cluster runs at reduced capacity when a node is down; it is therefore important to reduce node downtime. The contributors to downtime are:

1. Maintenance time, and
2. Time to load the Aerospike primary index.

Aerospike's primary index is in-memory and not stored on a persistent device. On a node restart, if the data is stored on disk, the index is rebuilt by scanning records on the persistent device. The time taken to complete index loading is then a function of the number of records on that node, and of the device speed.

To avoid rebuilding the primary index on every process restart, Aerospike's primary index is stored in a shared memory space disjoint from the service process's memory space. In case maintenance only requires a restart of the Aerospike service, the index need not be reloaded. The service attaches to the current copy of the index and is ready to handle transactions. This form of service start re-using an existing index is termed '*fast start'*; it eliminates scanning the device to rebuild the index.

2.2.3 Summary

Uniform distribution of data, associated metadata like indexes, and transaction workload make capacity planning and scaling up and 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 on its node.

2.3 Client-Server

Databases don't exist in isolation. They must therefore be architected as part of the full stack so that the end-to-end system scales. The client layer needs to absorb the complexity of managing the cluster. There are various challenges to overcome here and a few of them are addressed below.

2.3.1 Discovery

The client needs to know about all the nodes of the cluster and their roles. In Aerospike, each node maintains a list of its neighboring nodes. This list is used for the discovery of the cluster nodes. The client starts with one or more seed nodes and discovers the entire set of cluster nodes. Once all nodes are discovered, the client needs to know the role of each node. As described in section 2.2, each node owns a master or replica for a subset of partitions out of the total set of partitions. This mapping from partition to node (partition map) is exchanged and cached with the clients. Sharing of the partition map with the client is critical in making client-server interactions extremely efficient. This is why, in Aerospike, there is single-hop access to data from the client. In steady state, the scale-out ability of the Aerospike cluster is *purely* a function of the number of clients or server nodes. This guarantees the linear scalability of the system as long as other parts of the system – like network interconnect – can absorb the load.

2.3.2 Information Sharing

Each client process stores the partition map in its memory. To keep the information up to date, the client process periodically consults the server nodes to check if there are any updates. It does this by checking the version that it has stored locally against the latest version of the server. If there is an update, it requests for the full partition map.

Frameworks like `php-cgi`, `node.js` cluster can run multiple instances of the client process on each machine to get more parallelism. As all the instances of the client are on the same machine, they should be able to share this information between themselves. Aerospike uses a combination of shared memory and robust mutex code from the `pthread` library to solve the problem. `Pthread` mutexes support the following properties that can be used across processes:

```
PTHREAD_MUTEX_ROBUST_NP
PTHREAD_PROCESS_SHARED
```

A lock is created in a shared memory region with these properties set. All the processes compete periodically (once every second) to take the lock. Yet, only one process will get the lock. The process that gets the lock fetches the partition map from the server nodes and shares it with other processes via shared memory. If the process holding the lock dies, and when a different process tries to get the lock, it gets the lock with the return code `EOWNERDEAD`. It should call `pthread_mutex_consistent_np()` to make the lock consistent for further use. After this, it is business as usual.

2.3.3 Cluster Node Handling

For each of the cluster node, at the time of initialization, the client creates an in-memory structure on behalf of that node and stores its partition map. It also maintains a connection pool for that node. All of this is torn down when the node is declared down. The setup and tear-down is a costly operation. Also, in case of failure, the client needs to have a fallback plan to handle the failure by retrying the database operation on the same node or on a different node in the cluster. If the underlying network is flaky and this repeatedly happens, this can end up degrading the performance of the overall system. This leads to the need of having a balanced approach to identifying cluster node health. The following strategies are used by Aerospike to achieve this balance.

2.3.3.1 Health Score

The client's use of transaction response status code alone as a measure of the state of the DBMS cluster is a sub-optimal scheme. The contacted server node may temporarily fail to accept the transaction request. Or it could be that there is a transient network issue, while the server node itself is up and healthy. To discount such scenarios, clients track the number of failures encountered by the client on database operations at a specific cluster node. The client drops a cluster node only when the failure count (a.k.a "happiness factor") crosses a particular threshold. Any successful operation to that node will reset the failure count to 0.

2.3.3.2 Cluster Consultation

Flaky networks are often tough to handle. One-way network failures (A sees B, but B does not see A) are even tougher. There can be situations where the cluster nodes can see each other but the client is unable to see some cluster nodes directly (say, X). In these cases, the client consults all the nodes of the cluster visible to itself and sees if any of these nodes has X in their neighbor list. If a client-visible node in the cluster reports that X is in its neighbor list, the client does nothing. If no client-visible cluster nodes report that X is in their neighbor list, the client will wait for a threshold time and then permanently remove the node by tearing down the data structures referencing the removed node. Over several years of deployments, we found that this scheme greatly improved the stability of the overall system.

3. CROSS DATACENTER REPLICATION

This section describes how to stitch together multiple DBMS clusters in different geographically distributed data centers to build a globally replicated system. Cross Datacenter Replication (XDR) supports different replication topologies, including active-active, active-passive, chain, star, and multi-hop configurations.

3.1.1 Load Sharing

In a normal deployment state (i.e., when there are no failures), each node logs the operations that happen on that node for both the master and replica partitions. But each node only ships to remote clusters the data for master partitions on that node. The changes logged on behalf of replica partitions are used only when there are node failures. If a node fails, all the other nodes detect this failure and takeover the pending work on behalf of the failed node. This scheme scales horizontally as one can just add more nodes to handle increasing replication load.

3.1.2 Data Shipping

When a write happens, the system first logs the change, reads the whole record and ships it. There are a few optimizations to save the amount of data read locally and shipped across.

The data is read in batches from the log file. We first see if the same record is updated multiple times in the same batch. The record is read exactly once on behalf of all the changes in that batch. Once the record is read, we compare its generation with the generation recorded in the log file. If the generation on the log file is less than the generation of the record, we skip shipping the record. There is an upper bound on the number of times we skip the record, as the record may never be shipped if the record is getting updated continuously. These optimizations provide a huge benefit when there are hot keys in the system whose records are updated frequently.

3.1.3 Remote Cluster Management

The XDR component on each node acts as a client to the remote cluster. It performs all the roles just like a regular client, i.e., it keeps track of remote cluster state changes, connects to all the nodes of the remote cluster, maintains connection pools, etc. Indeed, this is a very robust distributed shipping system as there is no single point of failure. All nodes in the source cluster ship data proportionate to their partition ownership and all nodes in the destination cluster receive data in proportion to their partition ownership. This shipping algorithm allows both source and destination clusters to have different cluster sizes.

Our model ensures that clusters continue to ship new changes as long as there is at least one surviving node in the source or destination clusters. It also adjusts very easily to new node additions in source or destination clusters and is able to equally utilize all the resources in both clusters.

3.1.4 Pipelining

For cross data-center shipping, Aerospike uses an asynchronous pipelined scheme. As mentioned in section 3.1.3, each node in the source cluster communicates with all the nodes in the destination cluster. Each shipping node keeps a pool of 64 open connections that are used in a round robin manner to ship records. The record is shipped asynchronously, i.e., multiple records are shipped on the open connection; afterwards, the source waits for the responses. So, at any given point in time, there can be multiple records on the connection waiting to be written at the destination. This pipelined model is the main way we are able to deliver high throughput on high-latency connections over WAN. When the remote node writes the shipped record, it sends an acknowledgement back to the shipping node with the return code. We set an upper limit on the number of records that can be inflight for the sake of throttling network utilization.

4. OPTIMIZATIONS FOR SCALE UP

For a system to operate at extremely high throughput with low latency, we have found that it is necessary not just to scale out across nodes, but also to scale up on one node. This section talks about system-level details, which help Aerospike scale up to millions of transactions per second at sub-millisecond latencies per node. The techniques covered here apply to any data storage system in general. The ability to scale up on nodes effectively means the following:

1. Scaling up to higher throughput levels on fewer nodes.
2. Better failure characteristics, since probability of a node failure typically increases as the number of nodes in a cluster increase.
3. Easier operational footprint. Managing a 10-node cluster versus a 200-node cluster is a huge win for operators.
4. Lower total cost of ownership. This is especially true once you factor in SSD-based scaling described in section 5.

The basic philosophy here is to enable the system to take full advantage of the hardware by leveraging it in the best way possible.

4.1.1 Multi-Core System

Contemporary commodity processors have a multi-core, multi-socket architecture [15] with up to 64 cores. Caches in these systems have Non-Uniform Memory Access (NUMA) [24], which allow system memory bandwidth to scale with an increasing amount of physical processors. System memory in this kind of setup has asymmetric latency and throughput behavior based on access to data in the local cache in the same socket (vs. remotely from another socket). Applications sensitive to latency would require memory traffic to stay local and need to use a threading model that has locality of access per-socket, in order to be able to scale with the number of physical processors.

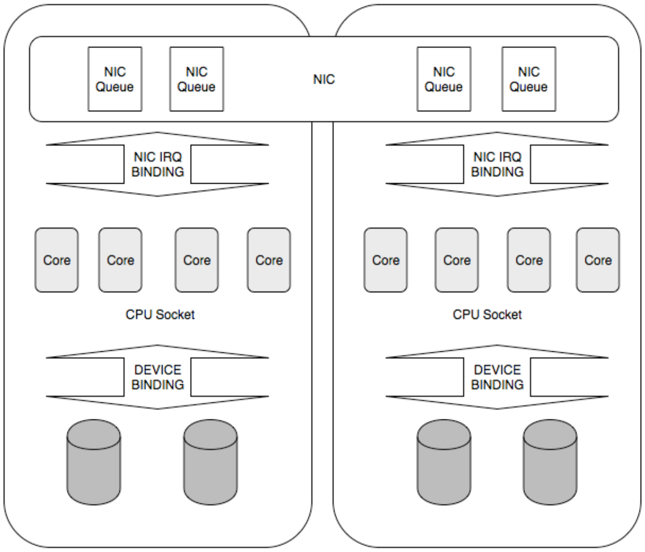


Figure 9: Multi-core architecture

Aerospike, as shown in Figure 9, groups multiple threads per CPU socket instead of per core, thus aligning with a NUMA node. These transaction threads are also associated with specific I/O devices. The interrupt processing for the client side network communication and disk side I/O is also bound to the core where these threads are running. This helps reduce the amount of shared data accessed across multiple NUMA regions, and reduces latency cost.

4.1.2 Context Switch

Another major factor working against the performance of a low latency system is thread context switch [10]. To avoid costs associated with context switches, operations in Aerospike are run in the network listener thread itself. To fully exploit parallelism, the system creates as many network listeners as there are cores. The client request is received, processed and responded back to, without yielding the CPU. In this model, it is necessary that the implementation be non-blocking, short, and predictable, so that the response at the network happens in real time.

4.1.3 Memory Fragmentation

Aerospike handles all its memory allocation natively rather than depend on the programming language or on a runtime system. To this effect, Aerospike implements various special-purpose slab allocators to handle different object types within the server process. Aerospike's in-memory computing solution effectively leverages system resources by keeping the index packed into RAM. With ever increasing data size, hardware (RAM sizes > 100s GB), and high transaction rates, memory fragmentation is a major challenge.

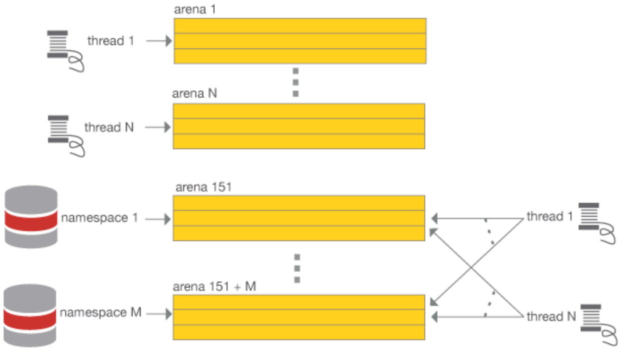


Figure 10: Memory arena assignment

To deal with such fragmentation, Aerospike chose to integrate with the `jemalloc` memory allocator library [18]. Beyond simply relying on the allocator to be internally efficient, we have used a few key extensions of `jemalloc` over the standard C library memory allocation interface in order to direct the library to store classes of data objects according to their characteristics. Specifically, as shown in Figure 10, by grouping data objects by namespace into the same arena, the long-term object creation, access, modification, and deletion pattern is optimized, and fragmentation minimized.

4.1.4 Data Structure Design

For data structures like indexes and global structures, which need concurrent access, there are three candidate models of design:

- Multi-threaded data structure with complex nested locking model for synchronization, e.g., step lock in a B+tree
- Lockless data structures

- Partitioned, single-threaded data structures

Aerospike adopts the third approach, in which, all critical data structures are partitioned, each with a separate lock. This reduces contention across partitions. Access to nested data structures like index trees does not involve acquiring multiple locks at each level; instead, each tree element has both a reference count and its own lock. This allows for safe and concurrent read, write, and delete access to the tree, without holding multiple locks.

These structures are carefully designed to make sure that frequently and commonly accessed data has locality and falls within a single cache line in order to reduce cache misses and data stalls. For example, the index entry in Aerospike is exactly 64 bytes, the same size as a cache line.

In production systems like Aerospike, it is not just the functional aspects, but also system monitoring and troubleshooting features that need to be built in and optimized. This information is maintained in a thread-local data structure and can be pulled and aggregated together at query time.

4.1.5 Scheduling and Prioritization

In addition to basic KVS operations, Aerospike supports batch queries, scans, and secondary index queries. Scans are generally slow background jobs that walk through the entire data set. Batch and secondary index queries return a matched subset of the data and, therefore, have different levels of selectivity based on the particular use case. Balancing throughput and fairness with such a varied workload is a challenge. This is achieved by following three major principles.

1. Partition jobs based on their type: Each job type is allocated its own thread pool and is prioritized across pools. Jobs of a specific type are further prioritized within their own pool.
2. Effort-based unit of work: The basic unit of work is the effort needed to process a single record including lookup, I/O and validation. Each job is composed of multiple units of work, which defines its effort.
3. Controlled load generation: The thread pool has a load generator, which controls rate of generation of work. It is the threads in the pool that perform the work.

Aerospike uses cooperative scheduling whereby worker threads yield CPU for other workers to finish their job after X units of work. These workers have CPU core and partition affinity to avoid data contention when parallel workers are accessing certain data.

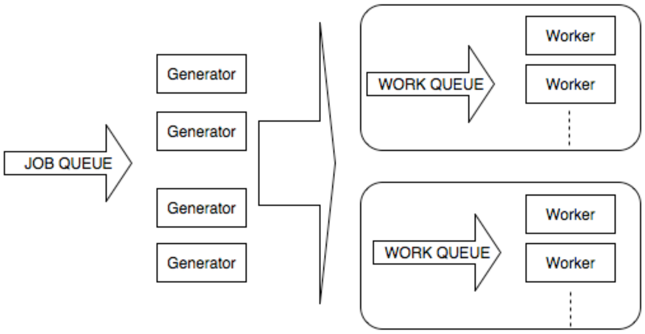


Figure 11: Job management

Concurrent workloads of a certain basic job type in Aerospike are generally run on a first-come, first-served basis to allow for low latency for each request. The system also needs the ability to make progress in workloads like scans and queries, which are long-running, and sometimes guided by user settings and/or by the application's ability to consume the result set. For such cases, the system dynamically adapts and shifts to round-robin scheduling of tasks, in which many tasks that are run in parallel are paused and re-scheduled dynamically, based on the progress they can make.

5. STORAGE

It is not just the throughput and latency characteristic, but also the ability to store and process large swaths of data that defines the ability of a DBMS to scale up. Aerospike has been designed from the ground up to leverage SSD technology. This allows Aerospike to manage dozens of terabytes of data on a single machine. In this section, we describe the storage subsystem.

5.1.1 Storage Management

Aerospike implements a hybrid model wherein the index is purely in memory (not persisted), and data is only on a persistent storage (SSD) and is read directly from the disk. Disk I/O is not required to access the index, which makes performance predictable. Such a design is possible because the read latency characteristic of I/O in SSDs is the same, regardless of whether it is random or sequential. For such a model, optimizations described in section 2.2.2 are used to avoid the cost of a device scan to rebuild indexes.

This ability to do random read I/O comes at the cost of a limited number of write cycles on SSDs. In order to avoid creating uneven wear on a single part of the SSD, Aerospike does not perform in-place updates. Instead, it employs a copy-on-write mechanism [23] using large block writes. This wears the SSD down evenly, which in turn, improves device durability. Aerospike bypasses the Operating System's file system and instead uses attached flash devices directly as a block device using a custom data layout.

When a record is updated, the old copy of the record is read from the device and the updated copy is written into a write buffer. This buffer is flushed to the storage when completely full.

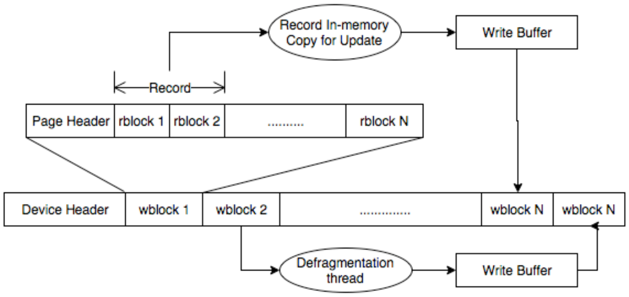


Figure 12: Storage layout

The unit of read, RBLOCKS, is 128 bytes in size. This increases the addressable space and can accommodate a single storage device of up to 2TB in size. Writes in units of WBLOCK (configurable, usually 1MB) optimize disk life.

Aerospike operates on multiple storage units of this type by striping the data across multiple devices based on a robust hash function; this allows parallel access to the data while avoiding any hot spots.

5.1.2 Defragmentation

Aerospike uses a log-structured file system with a copy-on-write mechanism [23]. Hence, it needs to reclaim space by continuously running a background defragmentation process. Each device stores a MAP of block and information relating to the fill-factor of each block. The fill-factor of the block is the block fraction utilized by valid records. At boot time, this information is loaded and kept up-to-date on every write. When the fill-factor of a block falls below a certain threshold, the block becomes a candidate for defragmentation and is then queued up for the defragmentation process.

While defragmenting a block, the valid records are read and moved to the new write buffer which, when full, is flushed to the disk. To avoid intermixing new writes and old writes, Aerospike maintains two different write buffer queues, one for normal client writes, and another for records that move while defragmenting.

In the running system, the blocks continually get fed into this queue to be defragmented, which adds to the write rate onto the disk. Setting a very high fill-factor threshold (normally 50%) increases device burnout rate, while a low setting decreases space utilization. Based on available disk buffer space that can be immediately consumed by writes (fully empty WBLOCKs), the defragmentation rate is adjusted to make sure that there is efficient space utilization.

5.1.3 Performance and Tuning

5.1.3.1 Post Write Queue

Instead of maintaining a LRU page cache, Aerospike maintains a so-called post write queue. This is a Least Recently Written (LRW) cache of data. There are a lot of application patterns where data that is written is immediately read back with temporal locality. A replication service, which ships recently changed records (as explained in Section 3), has this behavioral characteristic. Also, this cache requires no extra memory space over and above the write block caching that is used to perform writes to the disk. The post write queue improves the cache-hit rate and reduces I/O load on the storage device.

5.1.3.2 Shadow Device

In the cloud environment, there are devices available with different I/O and latency characteristics. For example, in Amazon EC2 instances, persistence can be achieved to different degrees by using an ephemeral disk (which survives process restarts, but not instance restarts) and EBS (which survives both process and instance restarts) [8]. Ephemeral devices are fast and attached to the instance directly. In contrast, EBS devices are slow and attached to the instance over the network. To scale up in systems like these, Aerospike employs a shadow device technique where writes are concurrently applied locally to the ephemeral storage, and remotely to EBS. Reads, however, are always done from ephemeral stores that can support much higher random access rate than EBS.

5.1.4 Summary

Note that SSDs can store an order of magnitude more data per node than DRAM. The IOPS supported by devices keep increasing; for instance, NVMe drives can now perform 100K IOPS per drive [16]. For the past several years, there have been several Aerospike clusters with 20-30 nodes that have used this setup and run millions of operations/second 24x7 with sub-millisecond latency [17].

6. BENCHMARK RESULTS

The Aerospike DBMS with the above architecture has been in deployment continuously since 2010. In the following sections, we present some of the benchmarks we ran to measure the scalability of the Aerospike database, and the benchmarking results. Note that, unless otherwise stated, all the measurements are expressed in single record read/write transactions per second (TPS).

6.1 Scale Up

Here, we present a set of experiments that demonstrate Aerospike's ability to scale up. We measure performance in terms of single-record read/write transaction throughput. The numbers were achieved by applying the techniques discussed in section 4. Two experiments were performed: one on a non-virtualized machine, and another on a virtualized cloud environment.

6.1.1 Non-Virtualized

This experiment [3] was performed on a 4 node cluster each with *8 core dual Socket Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz with 32GB DRAM and 1 NIC with 16 queues*. Tests were run using YCSB [9], which is a popular NoSQL benchmark. Aerospike ran with a hybrid configuration under which reads are performed from memory and writes are done to both memory and to the disk for persistence.

This experiment was done with the following workload

- Record containing 10 columns/bins of 10-byte string.
- 50 million records.
- YCSB workload A (Balanced 50/50) and workload B (Read Heavy 95/5) with Zipfian key distribution.

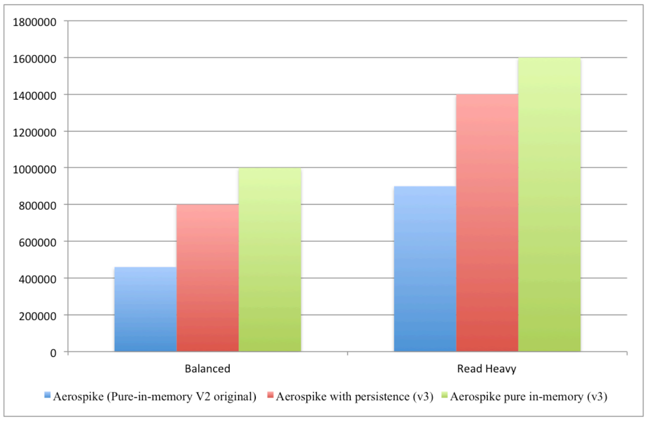


Figure 13: YCSB benchmark

As shown in Figure 13, the performance of Aerospike in terms of throughput nearly doubled after applying the techniques discussed in section 4.

6.1.2 Virtualized Environment

With the intention of mimicking real world scenarios, the experiments [1] were done on AWS EC2 instances [7]. In order to achieve a good spread, runs were performed on a single node cluster with instances ranging from low-end m3.xlarge to high-end r3.8xlarge. Workload was generated using the Aerospike Java Benchmark Tools [5]. In this setup, data was not persisted on disk.

The benchmark was done with the following workload:

- Records containing 10 columns/bins of 10-byte string each
- 10 million records
- 100% read load with normal key distribution

Figure 14 shows that we were able to achieve up to 1 million TPS on a single 8xlarge instance, and that Aerospike's performance scales up linearly as we move to larger and more powerful instance types.

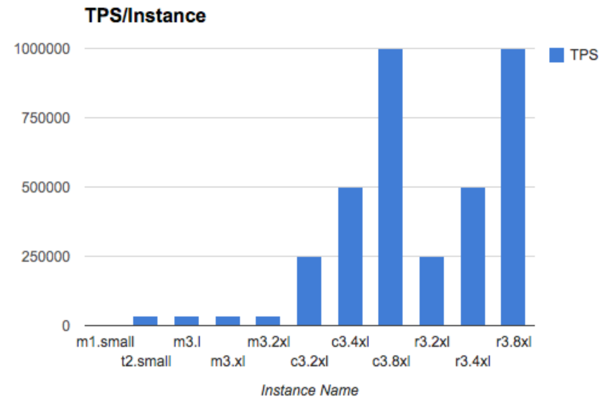


Figure 14: Performance on AWS EC2 instance types

6.2 Scale Out

This experiment was intended to evaluate Aerospike's scale out capability. Typical cloud deployments start with a low initial size and grow as the needs grow. Keep in mind that this set of experiments was performed using low-end instances. We performed experiments both in Amazon AWS EC2 and in Google Compute Engine [14] environments.

6.2.1 Amazon AWS

This experiment [26] was done on EC2 instances r3.xlarge instance types with the setup described in Section 6.1.2.

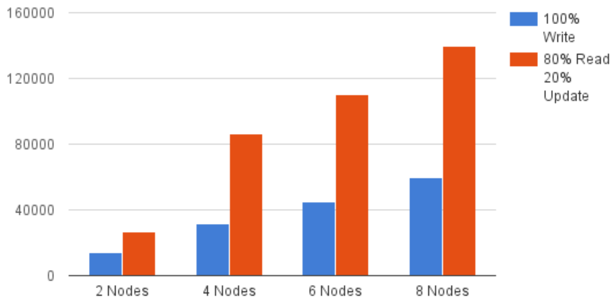


Figure 15: AWS – linear scalability

As shown in the Figure 15, the throughput scales linearly in both workloads (read-only and read-write) with the increase in the number of nodes from 2 to 8.

6.2.2 Google Compute Engine

This experiment [4] was done on n1-standard-8 instances running Debian 7 backports OS image. The run was done with data-in-memory with persistence on 500GB non-SSD disks. The experiment was performed with the following setup:

- Records containing 3 columns/bins of 50 byte string each
- 100 million records
- a 100% read and 100% write workload

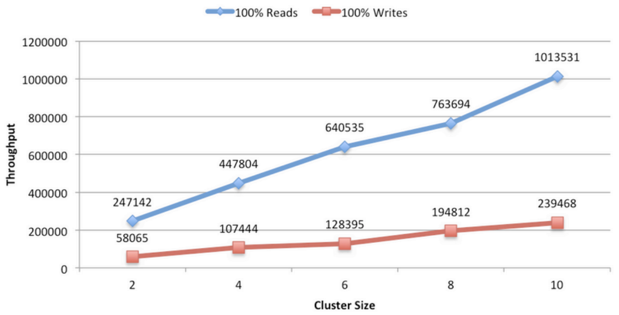


Figure 16: GCE – linear scalability

The results in Figure 16 show the linear scalability of Aerospike with the increase in the number of nodes in the cluster – from 2 to 10 nodes.

6.3 Storage

Aerospike is optimized to handle both in-memory and on-disk configurations equally well, as demonstrated by this experiment [2]. The setup is similar to the one described in section 6.2.2 with 10 nodes. The only change is that data is on the local SSD instead of on RAM, and both reads and writes hit the disk.

As shown in Figure 17, Aersopike's performance on SSDs is close to RAM with different workloads. Latencies of SSDs are higher than those of RAM and get amplified in the 100% reads case as seen in the graph. All the other cases also show a similar behavior.

We have partnered with Intel to check Aerospike's performance on upcoming SSDs. In experiment [6], we traded out DRAM for NVM (non-volatile memory) and ran it on Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz with 128GB RAM with P3700 PCIe Devices. Aerospike was able to achieve 1 million TPS with sub-millisecond latencies.

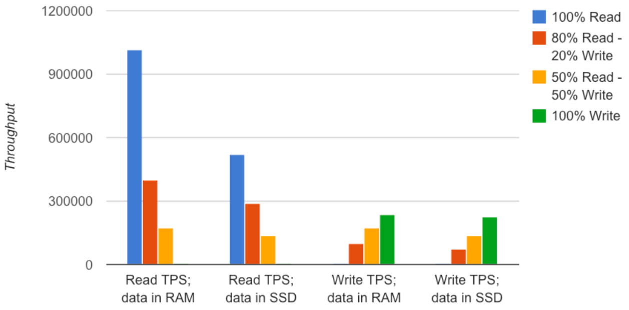


Figure 17: GCE – RAM vs. SSD

6.4 Summary

The results demonstrate the scale-out and scale-up characteristics of Aerospike, both in non-virtualized and in virtualized environments with data in memory and on disk. It highlights the high-throughput, low latency, and linear scalability of Aerospike.

The performance numbers in virtualized cloud environments is typically lower than in non-virtualized environments. In contrast to the common belief of slowness due to the overheads of virtualization, we found that the limiting factor is generally the artificial throttling done by cloud environments. And most of the time, it is the network that gets throttled. As things improve in networking, this will unleash the potential of Aerospike in cloud environments too.

7. CONCLUSION

It is now clear that real-time decision systems are spreading fast from the Internet to the Enterprise; this trend is accelerating and quickly becoming mainstream. The techniques described in this paper have helped us harness the high performance of contemporary commodity hardware to build a very high-throughput and low-latency read-write DBMS. We have found that this sort of DBMS is much sought after for building a variety of real-time decision systems in different industry application categories such as Financial Services, Telecommunication, Travel, E-Commerce, etc. An important lesson learned here is that scaling up on individual nodes of a distributed database is as important as scaling out across multiple nodes. In fact, DBMS clusters that use powerful nodes with SSDs allow applications to scale to Internet levels on much smaller cluster sizes. This, in turn, helps Enterprises affordably deploy world-class real-time decision systems as sophisticated as the ones that until recently were only available at large Internet companies.

8. ACKNOWLEDGEMENTS

We acknowledge the contribution of all the members of the Aerospike Engineering and Operations Team, who helped develop the Aerospike DBMS. Special thanks to Young Paik, Psi Mankoski, Ken Sedgwick, Tibor Szaboky, Meher Tendjoukian, Maud Calegari and Sri Varun Poluri for helping with this effort.

9. REFERENCES

[1] Aerospike 1 Million TPS, http://highscalability.com/blog/2014/8/18/1-aerospike-server-x-1-amazon-ec2-instance-1-million-tps-for.html

[2] Aerospike Demonstrates RAM-like Performance with Local SSDs, http://googlecloudplatform.blogspot.in/2015/01/Aerospike-demonstrates-RAM-like-performance-with-Local-SSDs.html

[3] Aerospike Doubles Performance, Shows 1 M TPS in YCSB Tests, http://www.aerospike.com/blog/aerospike-doubles-in-memory-nosql-database-performance

[4] Aerospike Hits One Million Writes Per Second with just 50 Nodes on Google Compute Engine, http://googlecloudplatform.blogspot.in/2014/12/aerospike-hits-one-million-writes-Per-Second-with-just-50-Nodes-on-Google-Compute-Engine.html

[5] Aerospike Java Benchmark Tools, http://www.aerospike.com/docs/client/java/benchmarks.html

[6] Aerospike on Intel SSD, https://communities.intel.com/community/itpeernetwork/blog/2015/02/17/reaching-one-million-database-transactions-per-second-aerospike-intel-ssd Amazon EC2, https://aws.amazon.com/ec2/

[7] Amazon EC2 Instances Types, http://aws.amazon.com/ec2/instance-types/

[8] Amazon Elastic Block Store, https://aws.amazon.com/ebs/

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. SoCC. (2010).

[10] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. ExpCS, (2007).

[11] Dewitt, D., Gerber, B., Graefe, G., Heytens, M., Kumar, K.,Muralikrishna. GAMMA – A High Performance Dataflow Database Machine. PVLDB, (1986).

[12] 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).

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

[14] GCE Google Compute Engine, https://cloud.google.com/compute/docs/

[15] Intel multi-core architecture optimization, http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

[16] Intel® Solid-State Drive Data Center Family for PCIe*, http://www.intel.com/content/www/us/en/solid-state-drives/intel-ssd-dc-family-for-pcie.html

[17] Intel Aerospike-Appnexus case study, http://www.intel.in/content/dam/www/public/us/en/documents/case-studies/ssd-aerospike-appnexus-study.pdf

[18] Jemalloc, http://www.canonware.com/jemalloc/

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

[20] Lamport, L. Paxos Made Simple, Fast, and Byzantine. OPODIS, 7-9, (2002).

[21] Mathias, Adrian Richard David. The order extension principle. Axiomatic set theory. Vol. 13. No. Part II., Proceedings of Symposia in Pure Mathematics, (1974).

[22] Real-time bidding, https://en.wikipedia.org/wiki/Real-time_bidding

[23] Rosenblum, Mendel, and John K. Ousterhout. The design and implementation of a log-structured file system. TOCS, (1992).

[24] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for NUMA-aware contention management on multicore systems. USENIXATC, (2011).

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

[26] The Cloud Does Equal High Performance, http://highscalability.com/blog/2014/8/20/part-2-the-cloud-does-equal-high-performance.html

[27] Yuan, Y., Wang, F., Li, J. and Qin, R., 2014, October. A survey on real time bidding advertising. In Service Operations and Logistics, and Informatics (SOLI), IEEE, (20