AEROSPIKE

IN-MEMORY NOSQL DATABASE

Architecture Overview

WHITEPAPER

About Aerospike

Aerospike delivers the first flash-optimized in-memory NoSQL database for revenue-critical, real-time context-driven applications that personalize the customer experience across Web, mobile, search, social, video, and email channels across the Internet. The database of choice in real-time bidding and cross-channel marketing, Aerospike is the user context store for global-scale platforms, such as AppNexus, Bluekai, eXelate, The Trade Desk and [x +1], processing terabytes of data and billions of transactions per day, with predictable low latency and continuous availability. With unmatched price/performance, Aerospike also scales up and out to accelerate real-time applications in ecommerce, retail, financial services and telecom. The free Aerospike Community Edition manages up to 200GB of unique data and can be upgraded to the Aerospike Enterprise Edition with support for larger deployments and cross data center replication. Aerospike is headquartered in Silicon Valley; investors include Alsop Louie, Draper Associates and NEA. For more information, visit www.aerospike.com

Table of Contents:

Overview:

Aerospike is a next-gen NoSQL database built from scratch in C to scale out on commodity hardware. Aerospike is commonly used as a cookie store or user profile store to personalize consumer experiences across the Internet.

Aerospike is:

- An operational NoSQL database with simple real-time analytics capabilities built in.
- A fast key value store with support for complex objects, user defined functions and distributed queries.
- The first in-memory NoSQL database optimized for flash (indexes in DRAM and data in DRAM or natively stored on flash devices) with dramatic price/performance benefits.
- A distributed database with a shared-nothing, clustered architecture and single row ACID properties. Aerospike ensures high availability, replicating data synchronously within a cluster and asynchronously across data centers.
- The system of engagement for personalized and revenue critical user interactions, Aerospike was architected to scale with zero touch and zero downtime.

From our experiences developing mission-critical applications with web-scale databases and our interactions with our customers, we developed a general philosophy of operational efficiency that further guided product development. It is these three principles - NoSQL flexibility, traditional database reliability, and operational efficiency – that drive the Aerospike architecture.

The advantages of Aerospike Database include:

- **100% data and service availability**
  - Synchronous replication ensures immediate Consistency (ACID)
  - Applications need not know when nodes fail; Aerospike Smart Client™ auto re-routes requests
  - If a node or rack fails, Aerospike Smart Cluster™ auto fails-over and re-balances
  - Aerospike Smart Cluster™ synchronously replicates data across identical nodes using the Aerospike Smart Partitions™ algorithm that automatically balances data and ensures there are no hotspots
  - Aerospike Cross Data Center Replication™ asynchronously replicates data across clusters in different data centers and provides BCP (Business Continuity Processing) during catastrophic failures
  - Rolling upgrades ensure no downtime and require no maintenance windows

- **Highly optimized system performance**
  - Fast processing in C, parallelized across nodes, cores, threads, SSDs
  - Aerospike's Real-Time Engine ensures that background tasks (index creation, data migration, etc.) do not interfere with low latency request processing
  - Aerospike Smart Client™ sends requests directly to node where data is resident

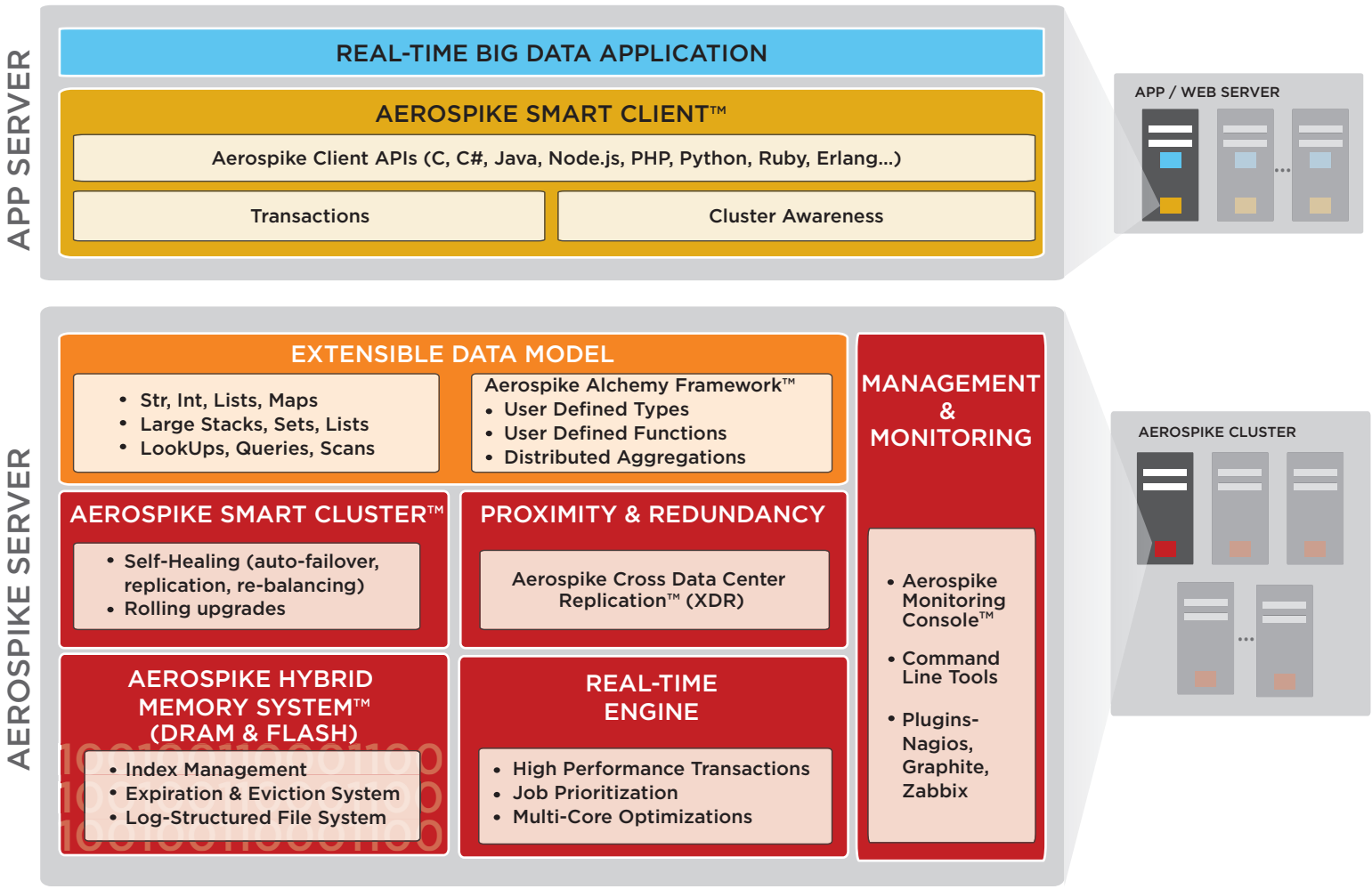
- **Fewer servers mean fewer failures and lowest TCO with Aerospike Hybrid-Memory System™**
  - 10x fewer servers than comparable DRAM based solutions
  - Manages vastly more data per node with SSDs than other systems

System Architecture

The Aerospike architecture can be broken into four main layers –

- Aerospike Smart Client™
- Extensible Data Model layer
- Smart Clusters™ layer
- Hybrid (DRAM & Flash) Memory System™

We'll very briefly describe the layers in this section, and in the later sections dive deeper into each layer's functionality. We will also describe the architecture of the Aerospike Cross Data Center Replication (XDR) system.



System Architecture Diagram

The components of the Aerospike System Architecture

Aerospike Smart Client™

The Aerospike Client libraries make up the Smart Client. To make application development easier and Aerospike provides a 'smart client' – in addition to implementing the APIs exposed to the transaction, the Client also tracks cluster configuration and manages the transaction requests, making any change in cluster membership completely transparent to the Application.

Extensible Data Model

The Aerospike system is fundamentally a key-value store where the keys can be associated with a set of named values (similar to a 'row' standard RDMBS terminology.) Data is collected into policy containers called 'namespaces', and within a namespace the data is subdivided into 'sets' (similar to 'tables') and 'records' (similar to 'rows'). Unlike RDBMS systems, the Aerospike system is entirely schema-less which means that sets and bins do not need to be defined up front, but can be added during run-time if additional flexibility is needed.

Aerospike Smart Cluster™

In the Smart Cluster layer, server cluster nodes communicate to ensure data consistency and replication across the cluster. The system uses a shared-nothing architecture, which enables it to be linearly scalable. The Smart Cluster is also at the heart of most of the system's ACID guarantees.

In addition, the Smart Cluster ensures that the Aerospike cluster remains fully operational when individual server nodes are removed from or added to the cluster. The distribution layer is self-managing. It automatically detects and responds to changes, eliminating the need for manual operations.

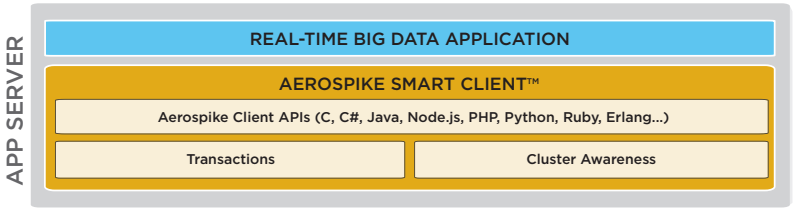
Aerospike Hybrid (DRAM & Flash) Memory System™

On each server node, the Hybrid Memory System handles management of stored data in-memory or on disk. It maintains indices corresponding to the data in the node. The Hybrid Memory System is optimized for operational efficiency – indices are stored in a very tight format to reduce memory requirements, and the system can be configured to use low level access to the physical storage media to further improve performance.

Cross Data Center Replication (XDR)

Aerospike clusters can be located in different data centers and data can be replicated across clusters for geographic redundancy or disaster recovery. This capability can also used for reducing latency by locating servers in close proximity to users. Clusters in different data centers can be of different sizes and each namespace can be configured to replicate asynchronously to one or multiple data centers at the same time. The system is very flexible and can be setup to implement any combination of star (master/slave or active/passive) or ring (master/master or active/active) topology.

Aerospike Smart Client™



Aerospike Smart Client

Aerospike provides a Smart Client layer between the application and the server. This layer handles many of the administrative tasks needed to manage communication with the node – it knows the optimal server for each transaction, handles retries, and manages any cluster reconfiguration issues in a way that is transparent to the application. This is done to improve the ease and efficiency of application development – developers can focus on key tasks of the application rather than database administration. The Client also implements its own TCP/IP connection pool for further transactional efficiency.

The Client Layer itself consists only of a linkable library (the 'Client') that talks directly to the cluster. This again is a matter of operational efficiency – there are no additional cluster management servers or proxies that need to be set up and maintained.

Note that Aerospike Clients have been optimized for speed and stability. Developers are welcome to create new clients, or to modify any of the existing ones for their own purposes. Aerospike provides full source code to the Clients, as well as documentation on the wire protocol used between the Client and servers. Clients are available in many languages, including C, C#, Java, Node.js, Ruby, PHP, and Python.

To now dive into more technical detail, consider that the Client Layer has the following responsibilities:

1. Providing an API interface for the Application
2. Tracking cluster configuration
3. Managing transactions between the Application and the Cluster

1. An API interface for the Application

Aerospike provides a simple and straightforward interface for reading and writing data. The underlying architecture is based around a key-value store where the 'value' may actually be a set of named values, similar to columns in a traditional RDBMS. Developers can read or write one value or multiple values with a single API call. In addition, Aerospike implements optimistic locking to allow consistent and reliable read-modify-write cycles without incurring the overhead of a lock.

Additional operations available include batch processing, auto-increment, and reading or writing the entire contents of the database. This final operation – reading and writing the entire database – is used for backup and restore, and is implemented in a way that allows the system to remain functional while the backup or restore is happening.

The APIs also provide several optional parameters that allow application developers to modify the operation of transaction requests. These parameters include the request timeout (critical in real-time operations where transactions are only valid if they can be completed within a specified time) and the policy governing automatic retry of failed requests.

For more information on the data model that underlies the APIs, see the Hybrid Memory System section.

2. Tracking Cluster Configuration

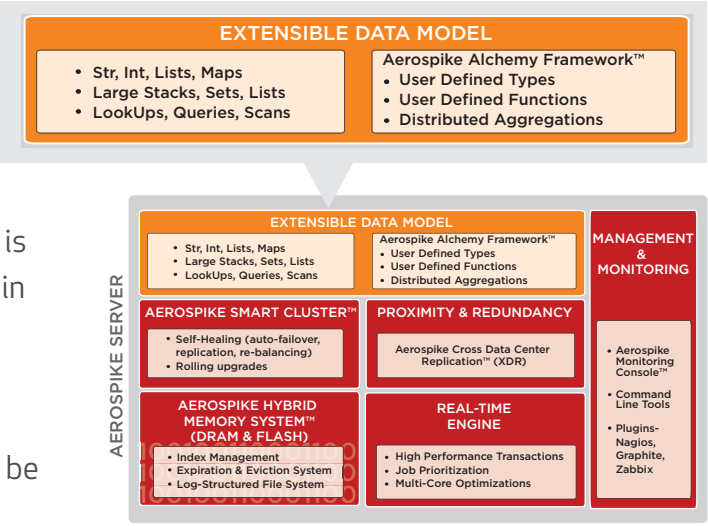
To ensure that requests are routed to the optimal cluster node, the Client tracks the current configuration of the server cluster. To do this, the Client communicates periodically with the cluster, maintaining an internal list of server nodes. Any changes to the cluster size or configuration are tracked automatically by the Client, and such changes are entirely transparent to the Application. In practice, this means that transactions will not fail during the transition, and the Application does not need to be restarted if nodes are brought on- or off- line.

3. Managing Transactions between the Application and the Cluster

When a transaction request comes in from the application, the Client formats that request into an optimized wire protocol for transmission to the servers. The Client uses its knowledge of the cluster configuration to determine which server is most likely to contain the requested data and parcels out the request appropriately.

As part of transaction management, the Client maintains a connection pool that tracks the active TCP connections associated with outstanding requests. It uses its knowledge of outstanding requests to detect transactional failures that have not risen to the level of a server failure within the cluster. Depending on the desired policy, the client will either automatically retry failures or immediately notify the Application of the transaction failure. If transactions on a particular node fail too often, the Client will attempt to route requests that would normally be handled by that node to a different node that also has a copy of the requested data. This strategy provides an additional level of reliability and resilience when dealing with transient issues.

Extensible Data Model



Extensible Data Model

The Aerospike system is fundamentally a key-value store where the keys can be associated with a set of named values (similar to a 'row' standard RDMBS terminology.) Data is collected into policy containers called 'namespaces', and within a namespace the data is subdivided into 'sets' (similar to 'tables') and 'records' (similar to 'rows'). Unlike RDBMS systems, the Aerospike system is entirely schema-less which means that sets and bins do not need to be defined up front, but can be added during run-time if additional flexibility is needed.

Within a namespace the data is subdivided into 'sets' (similar to 'tables') and 'records' (similar to 'rows'). Each record has an indexed 'key' that is unique in the set, and one or more named 'bins' (similar to columns) that hold values associated with the record. Values in the bins are strongly typed, and can include strings, integers, and binary data, as well as language-specific binary blobs that are automatically serialized and de-serialized by the system. Note that although the values in the bins are typed, the bins themselves are not – the same bin value in one record may have a different type than the bin value in different record. The other two special bin types are list and map. In addition to simple data types, Aerospike Extensible Data Model also supports:

1. Data Types

a. Complex data types

A **list** consists of a sequence of values and the type of each of these values can be one of the allowed bin types (including another list).

A **map** is simply a mapping of keys to values. The keys themselves can be integers or strings and each value can be one of the allowed bin types (including a map).

In this model, the application can store recursive data structures such as a list of maps, a map of maps, etc., to an arbitrary level of nesting. However, the total object size is still limited by the maximum block size configured in the installation. Since the Aerospike system is entirely schema-less, maps and lists do not need to be defined *a priori*. Instead, they can be added at run-time by any application that requires additional flexibility. These data types can be manipulated on the Aerospike clients as well as by the user-defined functions described later in this paper.

b. Large Data Types (LDT)

Large Data Types (LDTs) provide a way for applications to store data whose size exceeds the write block size (normal records in Aerospike cannot be larger than the write block size, which is typically between 128 KB and 2 MB). Large data types allow a group of objects to be stored in a single bin of an Aerospike record.

A single record can have multiple bins with such types. Implementation of such bin types is primarily a layer over the existing database functionality that is described earlier in this paper. Essentially, Large Data Types in Aerospike are implemented using the extensibility provided by the Aerospike platform and can be thought of as a User Defined Type. The LDT feature consists of four new collection types: stack, set, list and map.

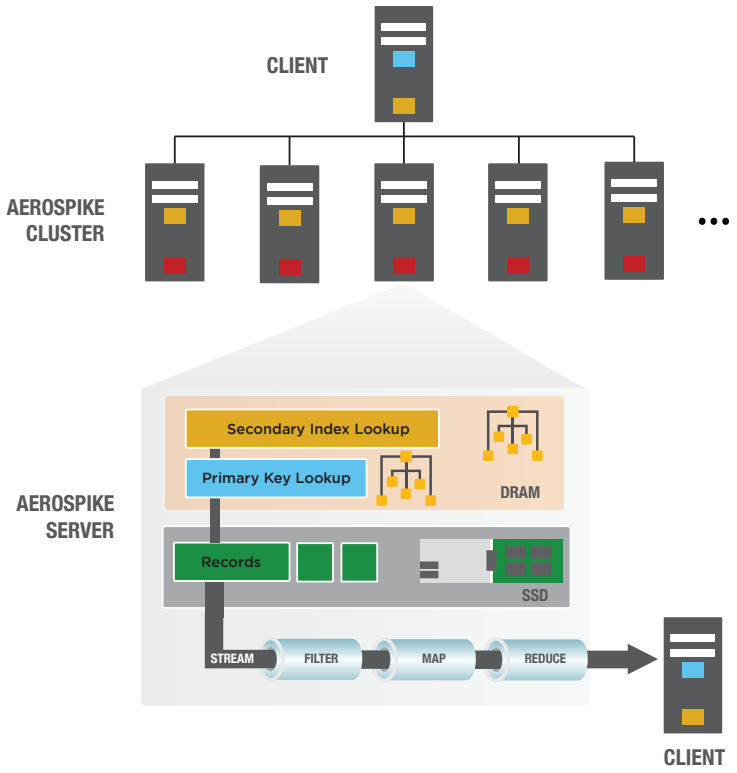


Figure A: Large Data Types

- **Large Stack** – these are well suited for time series data, providing Last In, First Out (LIFO) access to millions of time-ordered elements. Large Stacks can grow very large and the system automatically removes the oldest elements thus allowing perpetual data capture without running out of the storage capacity.
- **Large List** – these are typically used to store an ordered list of millions of items ordered by keys that may be unique or non-unique. The list is implemented as a B+Tree that can be quickly scanned for a key range.
- **Large Set or Map** – Large Set contains a unique collection of elements. Large Map supports a dictionary of objects, accessed by a unique name. A Large Set or Large Map is implemented as a hash map that can be quickly scanned. A Large Map and Set can store millions of elements and a record can have multiple instances of these bin types.

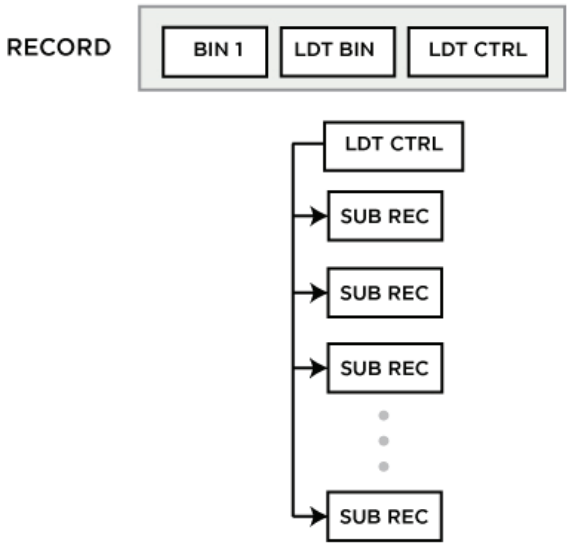
Aerospike LDT Layout

The LDT mechanism in Aerospike is implemented using Record UDFs that use an Aerospike storage system feature, the sub-record. A sub-record has all of the characteristics of an Aerospike record, except that it is permanently linked with its parent-record (what we refer to as a "top record"). The sub-record thus will always be co-located with its top record for the purpose of underlying cluster-oriented storage behavior; i.e., the sub-records will always stay in the same partition as their top records.

The figure on the right shows an LDT bin that stores a complex structure of sub-records bundled together as one large related object. Each of the LDT bins can be scanned and component objects can be queried using predicates.

All LDTs (stack, list, map, set) follow this general design. The LDT bin in the top record stores control information and references to sub-records. User's are not allowed access or visibility into the internals of the LDT mechanism – they are allowed access to the LDT only via a Record UDF on the top record.

A single record can have multiple bins with such LDTs. The only restriction is the top-record implementation of Large Set, which stores Large Set data directly in the bins of the top record.



Large Data Types

The LDT mechanism makes efficient use of storage. Each LDT starts out with actual data values stored within the top record itself and expands to have sub-records as needed. From the system perspective LDT is like any other bin type in Aerospike. Hence replication, migration, and duplicate resolution for the record with LDT are engineered to work seamlessly with the current system and to not overload the system.

The LDT design provides Aerospike, and its customers, several benefits. First, since an LDT can use a virtually unlimited number of sub-records, the size of an LDT is limited only by the size of storage. Second, a data object update is mostly limited to a single sub-record, so updates to very large collections will be efficient. The performance cost of updating a sub-record is related only to the size of the sub-record, and not the size of the entire data in an LDT bin. Third, the Record UDFs provide full access to the LDT bins on the server. Therefore, transformation and filtering of data can be performed on the server thus severely reducing network traffic between clients and server nodes.

2. Queries using Secondary Indexes

Aerospike's secondary indexes are built using the same designs that were applied in the implementation of its high performance primary indexes. Aerospike is tuned for queries that use low selectivity/high cardinality secondary indexes. Therefore, Aerospike secondary indexes are built on every node and the query to retrieve results is also currently sent to every cluster node.

The basic architecture is illustrated in Figure B on the next page and the steps are:

1. "Scatter" requests to all nodes

2. Indexes in DRAM for fast mapping of secondary to primary keys

3. Indexes co-located on each node with data on SSDs to guarantee ACID, manage migrations

4. Read records in parallel from all SSDs using lock free concurrency control

5. Aggregate results on each node

6. "Gather" results from all nodes on client

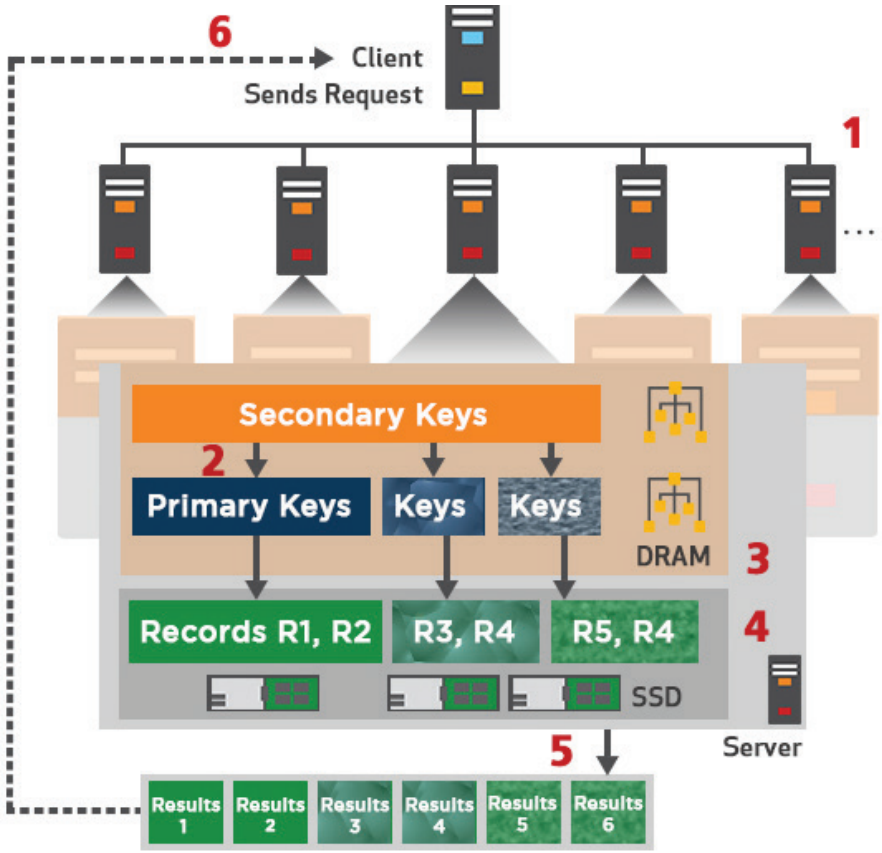


Figure B: Distributed queries with secondary indexes

a. Dynamic Index Creation

Aerospike supports dynamic creation of secondary indexes. To build a secondary index, users need to specify a namespace, set, bin and type of the index (e.g., integer, string, etc.). Once the index creation command is provided to one node of the cluster, the rest of the nodes are notified of this request using system metadata and index creation commences on all nodes in parallel. Index entries will only be created for records that match all of the index specifications.

Once a secondary index creation command is received and synchronized across all nodes, each node of the cluster creates the secondary index in **write-active** mode and starts a background job, which scans all of the data and inserts entries into the secondary index. The scan job that populates the secondary index will interact with read/write transactions in exactly the same way a normal scan would, except there is no network component to the index creation scan unlike for the normal scan. During the index creation period, all new writes that affect the indexing attribute will update the index. As soon as the index creation scan is completed and all of the index entries are created, the index is immediately ready for use by queries and is marked **read-active**.

The index creation scan only reads records that are already committed by transactions (no dirty reads). This means that the scan can execute at full speed, provided there are no updates of records blocking the reads. It is therefore important to set the index build priority at the right level to ensure that the index creation scan does not adversely affect the latencies of ongoing read and write transactions. The job prioritization settings in the Aerospike real-time engine can be used to effectively control the resource utilization of the index creation scan. The default

settings should suffice most of the time since those are based on years of deployment experience with balancing long running tasks such as data rebalancing and backups against low latency read/write transactions.

b. Querying an Index

A secondary index lookup can evaluate a very long list of primary key records. For this reason, we have elected to do secondary index lookups in small batches. There is also some batching on the client responses so that, if a memory threshold is reached, the response is immediately flushed to the network. This behaves much like the return values of an Aerospike batch request. The general idea is to keep the memory usage of an individual secondary lookup to a constant size regardless of the query's selectivity. The query process ensures the result is in sync with the actual data as of the time the query is executed and the record is scanned. Nowhere during the query process is uncommitted data read as part of the query results. However, the query itself could return data that has since been deleted.

Getting accurate query results is complicated during data migrations. When a node is added or removed from the cluster, the Data Migration Module is invoked to transition the data to and from nodes as appropriate for the new configuration. During such migration activity, it is possible that a partition may be available in different versions on many nodes. In order for a query to find the location of a partition that has the required data, Aerospike query processing makes use of additional partition state that is shared among nodes in the cluster and makes a judicious selection of a Query Node for each partition where the query may be executed. The Query Node for a given partition is chosen based on various factors (e.g., the number of records in the partition, the number of copies of the partitions that exist in the cluster, etc.). The goal here is to get the most accurate result to the query and the system is designed to do that.

3. Data Processing using Record User Defined Functions (UDFs)

User-defined functions (UDFs) are created and defined by the application programmer. UDFs extend the database schema in an application-specific manner by offloading data processing onto the database itself. This has several benefits:

- Enforcing data validation rules independent of the actual applications running outside the database
- Minimizing network traffic between client and server by filtering and transforming the database record at the cluster node and returning only the necessary subset of data to the client
- Allowing applications to extend the database schema without any change to the database software

Aerospike supports two types of user-defined functions, one that works on a record and another that works on a stream of records. A Record UDF is used when applications need to perform an operation on a single record. A Re-

cord UDF can have one or more parameters and it can be used to create or update a record based on these parameters. A Record UDF always expects the first argument to be a single record from the database.

In Aerospike, a Record UDF can be applied to an existing record or a non-existent record. This works because the key for a given record dictates where the record is stored in the cluster. The predictable hashing of the key allows us to call the UDF on the node that is ultimately responsible for storing the record. With a Record UDF, you can choose to create a record if it doesn't exist.

Common uses of a Record UDF include:

- Rewriting records in the database
- Perform calculations using data within a record
- Generate a new record based on arguments to a UDF

The most important thing to note about UDF execution is that all UDFs need to be run in a separate environment within the database. For UDFs written in the Lua programming language, the UDF needs to be called within a special environment called the Lua-state. For efficiency reasons, on server startup, a pool of Lua-state objects is created for each UDF module. When a Record UDF request arrives at the server node for a record with a given key, the following steps occur:

- The requested record is locked
- An instance of the Lua-state is removed from the pool, initialized and assigned to this request
- The desired Lua function is invoked within the chosen Lua-state instance
- Once the function completes execution, the server releases the record lock, sends the return value(s) to the client and releases the Lua-state back into the pool for later use on a subsequent request.

Note that if no Lua-state objects are available in the pool when a new Record UDF request is received, the server creates a new Lua-state object on the fly. For best performance, the server must be configured to have sufficient resources to minimize the creation of new objects in this manner.

4. Distributed Aggregations using Stream UDF

The Aerospike aggregation framework performs distributed stream-oriented processing of large data sets across multiple nodes of the cluster. The Aerospike aggregation framework uses key Aerospike features as follows:

- Secondary Indexes are used for feeding the list of input records into the aggregation process.
- Special stream-oriented User-Defined Functions are used for implementing the various stages.

A Stream UDF is used for distributed aggregations in which a stream of records returned from a query may be filtered, transformed and aggregated on each node in the cluster and again re-aggregated on the client to generate the desired result.

A Stream UDF expects the first argument to be a single stream, which the function will process with additional operations. Stream UDFs are primarily used to:

- Transform results from a query, and
- Aggregate results from a query

Aggregation can consist of one or more stages that are implemented by stream UDFs:

- Filtering – removes elements that don't satisfy a specified condition
- Mapping – maps elements to a new value
- Grouping – groups elements according to some rule (e.g., group by membership in a set or value of an attribute)
- Aggregating – combines elements to create a new value (e.g., count, average, etc.)
- Sorting – arranges elements in an ordered sequence

Other relational algebra operations like projection, selection, and renaming are also supported.

Figure A illustrates the basics of distributed aggregation. The source of a stream is always a secondary index based query. As described earlier in the Section on Queries using secondary indexes, the secondary index query is executed on each cluster node and the initial portion of the evaluation is exactly identical to the evaluation of the secondary index based query described in the Queries Section (Figure B).

The difference between regular queries and aggregations is that the output records on each node are used to generate a stream that is provided as input to the first Stream UDF in the aggregation defined by the application. Note that all of the UDFs that form the stream are properly chained together into an aggregation pipeline whose

components consist of Stream UDFs, as shown in Figure A. The query processing implemented can be quite complex, including MapReduce finalize types of queries or sophisticated quick returning analytics queries on data that is changing in real-time.

The idea here is to enable users to define a sequence of operations (filter, map, reduce, etc.) to be performed on a sequence of values. These operations can then be applied to a stream of records produced from an operation, such as a query using a secondary index. A portion of the operations to be applied to the stream or records will run on each node in the cluster while the other portion will run on the client application. The Stream UDF pipeline is evaluated in a lazy fashion, which keeps the resource requirements bounded so that it is possible to do an efficient space/time tradeoff.

Note that aggregation must be performed on all nodes to reduce the data being transmitted to the client. The process must also include a final aggregation phase that is executed on the client side to generate the complete, final results. Since an Aerospike client is directly connected to each cluster node, this processing is quite simple to perform. The final results are sent to the client from each node and the client composes the final result after performing any residual processing. In this case, the final phase(s) of the Stream UDF must be evaluated on the client.

In order to ensure that aggregation performance is balanced, we use various techniques:

- Global queues are used to manage the records fed through various processing stages and thread pools are used to effectively utilize CPU parallelism.

- The query state is shared across the entire thread pool so that the system is able to properly manage the Stream UDF pipeline.

- Note that except for the initial data fetch portion, every stage in the aggregation is a CPU bound operation. So it is important that the processing of various stages finish quickly and optimally. To facilitate this, we have used techniques such as batching of records, caching of UDF states, etc. to optimize the system overhead of processing large numbers of records in real-time.

- Furthermore, for operations on namespaces with data stored in memory (no storage fetch), the stream processing is implemented in a single thread context – even in such cases, the system can still parallelize the operations across partitions of the data since Aerospike natively divides data into a fixed number of partitions.

5. System Metadata

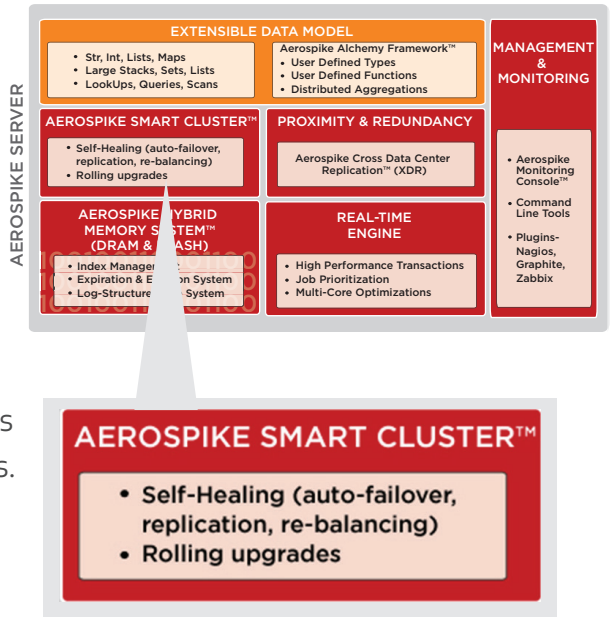
System metadata is a set of information required to configure a given UDFs and secondary indexes. metadata is dynamic in nature, and needs to be the same on all servers in a cluster. For example, the metadata for the UDF module specifies all user-defined functions currently available for invocation on all nodes of the cluster. The metadata for the Secondary Indexes defines the properties of all secondary indexes currently available in the cluster. The system metadata serves as a cluster-wide catalog that creates and maintains a consistent state of definitions across the entire cluster. Additionally, system metadata provides an orderly means of transitioning from one consensus state to another after any changes to module definitions are caused by cluster reconfiguration or client requests.

The System metadata module performs its actions as a distributed state machine, similar to, and triggered via, the existing Cluster Administration Module. Each node performs the same operations to add and remove metadata, with the Paxos sub-system performing the task of first constructing and then synchronizing the metadata across all nodes in the cluster.

Aerospike Smart Cluster™

The Smart Cluster layer is responsible for both maintaining the scalability of the Aerospike clusters, and for providing many of the ACID reliability guarantees. The implementation of Smart Cluster is 'shared nothing' – there are no separate 'managers', eliminating bottlenecks and single points of failure such as those created by master/slave relationships.

All communication between both cluster nodes and Application machines happens via TCP/IP. We have found that in modern Linux environments, TCP/IP requests can be coded in a way that allows many thousands of simultaneous connections at very high bandwidths. We have not found the use of TCP/IP to impact system performance when using GigE connections between components.



Aerospike Smart Cluster

There are three major modules within the Smart Cluster:

1. the **Cluster Administration Module**
2. the **Data Migration Module**
3. the **Transaction Management Module**

1. Cluster Administration Module

The Cluster Administration Module is a critical piece of both the scaling and reliability infrastructure, since it determines which nodes are currently in the cluster.

Each node periodically sends out a heartbeat to all the other nodes, informing them that it is alive and functional. If any node detects a new node, or fails to receive heartbeats from an existing node, that node's Cluster Administration Module will trigger a Paxos consensus voting process between all the cluster nodes. This process determines which nodes are considered part of the cluster, and ensures that all nodes in the cluster maintain a consistent view of the system. The Cluster Administration Module can be set up to run over multicast IP (preferred) or unicast (requiring slightly more configuration).

To increase reliability in the face of heavy loading - when heartbeats could be delayed - the system also counts any transactional requests between nodes as secondary heartbeats.
Once membership in the cluster has been agreed upon, the individual nodes use a distributed hash algorithm to divide the primary index space into data 'slices' and then to assign read and write masters and replicas to each of the slices. Because the division is purely algorithmic, the system scales without a master and eliminates the need for additional configuration that is required in a sharded environment. After cluster reconfiguration, data migration between the slices is handled by the Data Migration Module, featured next.

2. Data Migration Module

When a node is added or removed from the cluster, the Data Migration Module is invoked to transition the data to and from nodes as appropriate for the new configuration. The Data Migration Module is responsible for balancing the distribution of data across the cluster nodes, and for ensuring that each piece of data is duplicated across nodes as specified by the system's configured Replication Factor. The data migration process, by which data is moved to the appropriate node, is completely transparent to the both the Client and the Application.

The Data Migration Module, like much of the Aerospike code, has been carefully constructed to ensure that loading in one part of the system does not cause overall system instability. Transactions and heartbeats are prioritized above data migration, and the system is capable of fulfilling transactional requests even when the data has not been migrated to the 'correct' node. This prioritization ensures that the system is functional even while servers are coming on- or off- line, and that changes to the configuration of the cluster do not bring the system down.

3. Transaction Processing Module

The Transaction Processing Module provides many of the consistency and isolation guarantees of the Aerospike system. This module processes the transaction requests from the Client, including resolving conflicts between different versions of the data that may exist when the system is recovering from being partitioned.

In the most common case, the Client has correctly identified the correct node responsible for processing the transaction – a read replica in the case of a read request, or in the case of a write request, the write master for that data's slice (Requests that involve simultaneous reads and writes go to the write master.) In this situation, the Transaction Processing Module looks up the data, applies the appropriate operation and returns the result to the Client. If the request modifies data, the Transaction Processing Module also ensures immediate consistency within the system – before committing the data internally and returning the result to the client, it first propagates the changes to all other replicas.

In some cases, the node that receives the transaction request will not contain the data needed to complete the transaction. This is a rare case, since the Client maintains enough knowledge of cluster state to know which nodes should have what data. However, if the cluster configuration changes, the Client's information may briefly be out of date. In this situation, the Transaction Processing Module forwards the request to the Transaction Processing Module of the node that is responsible for the data. This transfer of control, like data migration itself, is completely transparent to the Client. Finally, the Transaction Processing Module is responsible for resolving conflicts that can occur when a cluster is recovering from being partitioned. In this case at some point in the past, the cluster has split (partitioned) into two (or more) separate pieces. While the cluster was partitioned, the Application wrote two different values to what should be the identical copies of the data. When the cluster comes back together, there is a conflict over what value that data should have.

The Transaction Processing Module is responsible for detecting this condition, and can resolve it in one of two ways. If the system has been configured for automatic conflict resolution, the Transaction Processing Module will consider the data with the latest timestamp to be canonical. On a read operation, that data will be returned to the Client. If the system has not been configured for automatic conflict resolution, the Transaction Processing Module will hand both copies of the data to the Client (where they are then forwarded to the Application). It becomes the Application's responsibility to decide which copy is correct.

Aerospike Hybrid (DRAM & Flash) Memory System™

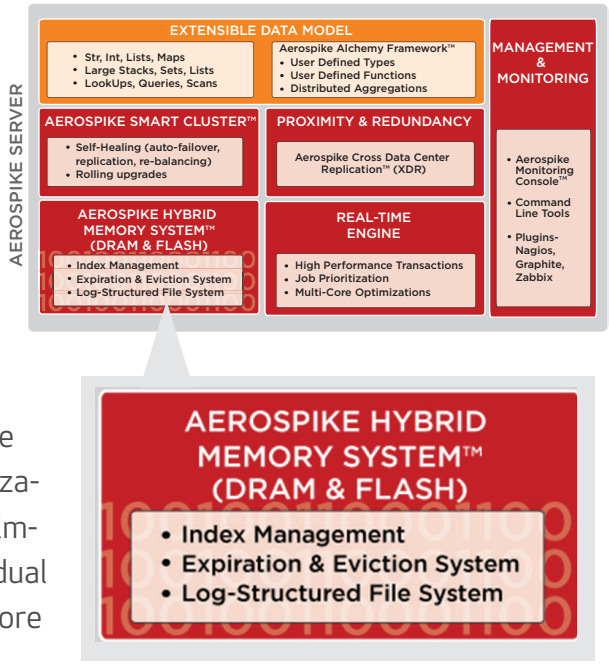
The Hybrid Memory System holds the indexes and data stored in each node, and handles interactions with the physical storage. It also contains modules that automatically remove old data from the database, and defragment the physical storage to optimize disk usage.

1. Data Storage

Aerospike can store data in DRAM, traditional rotational media, and SSDs, and each namespace can be configured separately. This configuration flexibility allows an application developer to put a small namespace that is frequently accessed in DRAM, but put a larger namespace in less expensive storage such an SSD. Significant work

has been done to optimize data storage on SSDs, including by-passing the file system to take advantage of low-level SSD read and write patterns.

Aerospike's data storage methodology is optimized for fast transactions. Indices (via the primary Key) are stored in DRAM for instant availability, and data writes to disk are performed in large blocks to minimize latencies that occur on both traditional rotational disk and SSD media. The system also can be configured to store data in direct format – using the drive as a low-level block device without format or file system – to provide an additional speed optimization for real-time critical systems. (Because storing indices in DRAM impacts the amount of DRAM needed in each node, the size of an individual index has been painstakingly minimized. At present, Aerospike can store indices for 100 million records in 7 gigabytes of DRAM.)



Aerospike Hybrid Memory System

The contents of each namespace are spread evenly across every node in the cluster, and duplicated as specified by the namespace's configured Replication Factor. For optimal efficiency, the location of any piece of data in the system can be determined algorithmically, without the need for a stored lookup table.

2. Defragmentor and Evictor

Two additional processes – the Defragmenter and the Evictor – work together to ensure that there is space both in DRAM and disk to write new data. The Defragmenter tracks the number of active records on each block on disk, and reclaims blocks that fall below a minimum level of use.

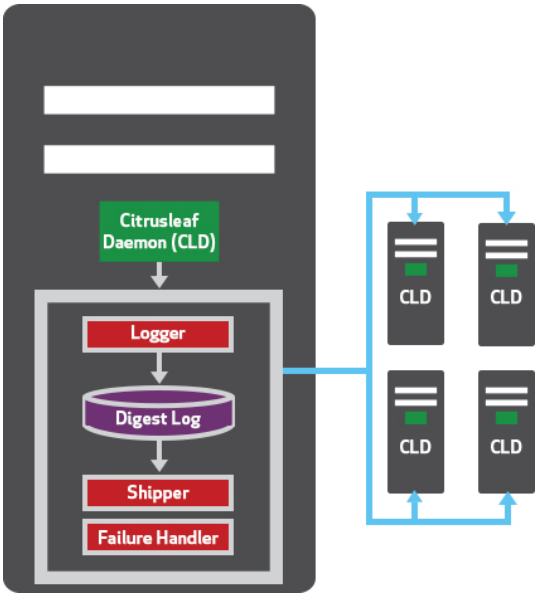
The Evictor is responsible for removing references to expired records, and for reclaiming memory if the system gets beyond a set high water mark. When configuring a namespace, the administrator specifies the maximum amount of DRAM used for that namespace, as well as the default lifetime for data in the namespace. Under normal operation, the Evictor looks for data that has expired, freeing the index in memory and releasing the record on disk. The Evictor also tracks the memory used by the namespace, and releases older, although not necessarily expired, records if the memory exceeds the configured high water mark. By allowing the Evictor to remove old data when the system hits its memory limitations, Aerospike can effectively be used as an LRU cache.

Note that the age of a record is measured from the last time it was modified, and that the Application can override the default lifetime any time it writes data to the record. The Application may also tell the system that a particular record should never be automatically evicted.

Cross Data Center Replication (XDR)

The XDR system contains the following components:

1. Digest Logger Module



XDR Architecture

- The digest logger listens to all of the changes happening in the cluster node and stores this information in a log file called the digest log.
- The digest log stores minimal information which is enough to ship the actual records at a later stage – typically, only the key's hash value along with some other metadata is stored in the log and this keeps the file size small.
- The digest log file is a ring buffer. i.e., the most recent data could potentially overwrite the oldest data (if the buffer is full). It is important to size the digest log file appropriately to ensure that it stores enough information to handle link failure across data centers. Because the digest log file is stored in an efficient way, this is not a major problem. A few GBs of digest log file can hold days' worth of log.

2. Data Shipper Module

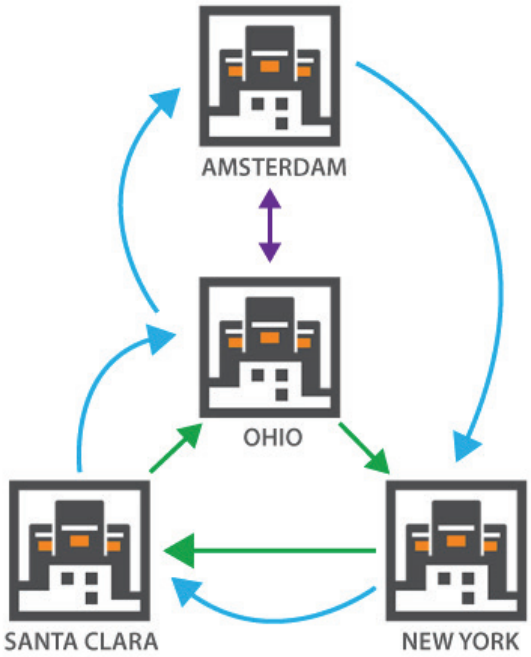
- The data shipper is responsible for reading the digest log and shipping the data.
- This module acts like a client application to both the local cluster and the remote cluster albeit with some special privileges and settings. The data is read from the local cluster as normal reads and also shipped to the remote cluster as normal writes on the remote cluster. This approach has a huge benefit that there is no dependency in the configuration of the local and remote clusters. For e.g. the local cluster can be m-node cluster while the remote is n-node cluster. The local and remote clusters can have different settings for replication factor, number of disks per-namespace, disk/memory size configuration etc.
- The speed at which XDR ships the data to the remote site is predominantly dependent on two factors, the write speed on each node and the network bandwidth that is available. Typically in the production environments, there are cycles of peak loads on low times. The bandwidth that is available should be at-least

enough to catch up the pending-to-be-shipped data during the low periods.

- It is important to adjust the XDR batch settings appropriately to ensure that XDR will ship data at a higher rate to one or more remote clusters than the rate at which writes are occurring in a local cluster. To handle situations in a production system, we have hooks in the system which can be dynamically changed without having to restart the XDR service.

3. Failure Handler Module

- The Failure handler is responsible for monitoring and performing the required activity to make sure the system continues shipping data in case of local node failure.



- Node failures in the local data center are handled by XDR keeping a copy of the digest log (shadow log) at the replica node. Normally, XDR does not ship using the shadow logs. Once a node has failed, the XDR logs in the nodes that are replica nodes (for partitions that were master in the failed node) will take over the responsibility for shipping the data. This way XDR ensures that there is no data shipping glitches during a single or multiple node failure.

- In this case of node failures in remote datacenter, XDR piggybacks on the standard Aerospike intra-cluster replication scheme. The normal failure handling between client and server takes care of this (XDR access to a remote datacenter is purely a client access).

XDR can be enabled/disabled at namespace level. XDR can be setup in either active-passive mode or active-active mode. XDR supports complex configurations with multiple data centers. The changes occurring on one data center can be shipped to multiple destinations. In addition, the writes due to XDR replication can be forwarded to a different cluster. Combining the above two functionalities, a lot more complex topologies of replication can be formed (e.g., a ring configuration, a star configuration, and a mixture of both).

Summary

Aerospike builds on speed, scale, reliability and powerful extensibility. We achieved these goals, as follows:

Speed – Aerospike secondary indexes are stored in DRAM and modeled on the fast design of the primary indexes. As described above, multi-threading and parallelism is used to effectively distribute query load across CPU cores and other optimizations are used to minimize overhead. Caching Lua-state objects enables efficient execution of UDFs.

Scale – The Aerospike secondary index system quickly reduces the data fed into any stream processing by an order of magnitude. It is well established that indexing used wisely is much better for query processing than brute force parallelism. Aerospike uses indexing, I/O parallelism, and CPU parallelism along with its innovative real-time prioritization techniques to achieve extremely high vertical and horizontal scale. Aerospike can execute low selectivity queries in parallel on cluster nodes and thus enables an application to simply increase the amount of data queried by increasing the number of nodes without substantially increasing query response times.

Reliability – The Aerospike replication and shared-nothing architecture provide the seamless reliability that is a hallmark of Aerospike deployments. We have implemented the new extensibility features while still making them conform to the reliability requirements of the platform.

Extensibility – All features of the system described here extend the functionality in such a way as to enable a rich set of applications to use Aerospike. Complex Data Types, queries using secondary indexes, Record UDF, and distributed aggregations using Stream UDF are all extensions that will enable new applications that can perform arbitrarily complex processing and obtain fast and accurate query results on data that is being updated in real-time. The most interesting extension of them all is Large Data Types that was itself a user data type built using the other extensibility features of the system, thus proving the versatility of Aerospike architecture.

2525 E. Charleston Rd #201 | Mountain View | CA | 94043
+1 408.462.AERO (2376) www.aerospike.com | sales@aerospike.com

AEROSPIKE