Architecture

Aerospike is a distributed, scalable NoSQL database, it is architected with three key objectives:

To create a flexible, scalable platform that would meet the needs of today's web-scale applications

To provide the robustness and reliability expected from traditional databases.

To provide operational efficiency (minimal manual involvement)

First published in the Proceedings of VLDB (Very Large Databases) in 2011 the Aerospike architecture consists of 3 layers:

The cluster-aware Client Layer includes open source client libraries that implement Aerospike APIs, observes nodes and knows where data reside in the cluster.

The self-managing Clustering and Data Distribution Layer oversees cluster communications and automates fail-over, replication, cross data center synchronization and intelligent re-balancing and data migration.

The flash-optimized Data Storage Layer reliably stores data in RAM and Flash.

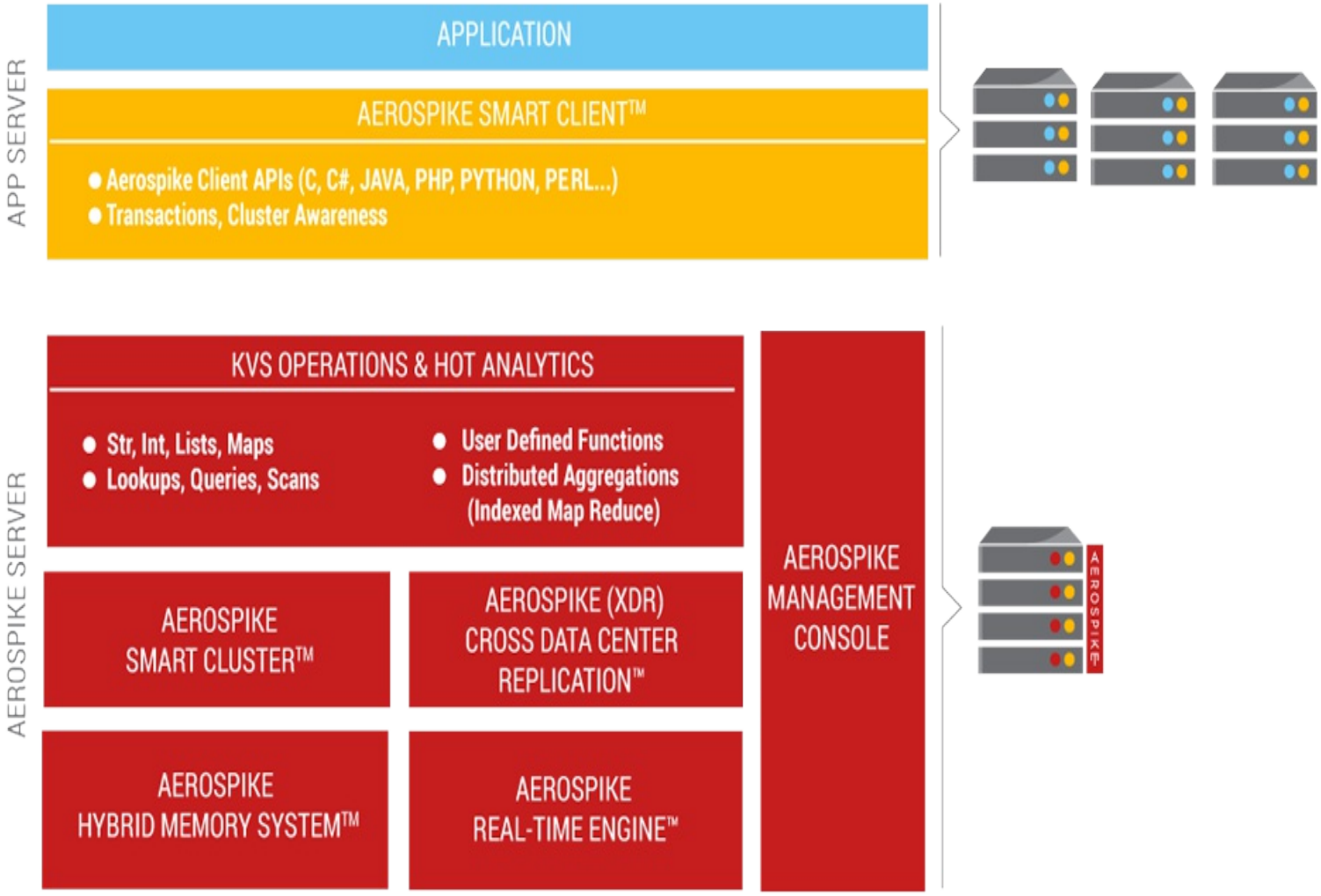
The big picture

The Aerospike architecture consists of 3 layers:

1. The cluster-aware Smart Client™ includes open source client libraries that implement Aerospike APIs and communicates directly to the cluster via the Aerospike wire protocol. The client is also a first class observer of the cluster, tracking nodes and using a robust algorithm to know where data reside in the cluster.

2. The self-managing Clustering and Data Distribution Layer oversees cluster communications and automates failover, replication, cross-data center synchronization, intelligent re-balancing and data migration.

The flash-optimized Data Storage Layer reliably stores data in RAM and Flash.



Client Layer

The Aerospike Smart Client™ is designed for speed. It is implemented as an open source linkable library available in C, C#, Java, PHP, Go, node.js and Python, and developers are free to contribute new clients or modify them as needed. The Client Layer has the following functions:

Implements the Aerospike API, and talks directly to the cluster via the client-server protocol.

Tracks nodes and knows where data is stored, instantly learning of changes to cluster configuration or when nodes go up or down.

Implements its own TCP/IP connection pool for efficiency. Also detects transaction failures that have not risen to the level of node failures in the cluster and re-routes those transactions to nodes with copies of the data.

Transparently sends requests directly to the node with the data and re-tries or re-routes requests as needed, for example during cluster reconfiguration.

This architecture reduces transaction latency, offloads work from the cluster and eliminates work for the developer. It also ensures that applications do not have to be restarted when nodes are brought up or down. Finally, it eliminates the need to setup and manage additional cluster management servers or proxies.

Cluster Layer

The Aerospike “shared nothing” architecture is designed to reliably store terabytes of data with automatic fail-over, replication and cross data-center synchronization. This layer scales linearly and is designed to eliminate manual operations with the systematic automation of all cluster management functions. It includes 3 modules:

The Cluster Management Module - tracks nodes in the cluster. The key algorithm is a Paxos-like consensus voting process which determines which nodes are considered part of the cluster. Aerospike implements active and passive heartbeats to monitor inter-node connectivity.

Data Migration Module - when a node is added or removed and cluster membership is ascertained, each node uses distributed hash algorithm to divide the primary index space into data 'slices' and assign owners. Data Migration Module then intelligently

balances the distribution of data across nodes in the cluster, and ensures that each piece of data is duplicated across nodes and across data centers, as specified by the system's configured replication factor. Division is purely algorithmic, the system scales without a master and eliminates the need for additional configuration that is required in a sharded environment.

The Transaction Processing Module - reads and writes data as requested and provides many of the consistency and isolation guarantees. This module is responsible for

1. **Sync/Async Replication** : For writes with immediate consistency, it propagates changes to all replicas before committing the data and returning the result to the client.
2. **Proxy** : In rare cases during cluster reconfigurations when the Client Layer may be briefly out of date, it transparently

proxies the request to another node.

5. **Duplicate Resolution** : when a cluster is recovering from being partitioned, it resolves any conflicts that may have occurred between different copies of data. Resolution can be configured to be automatic, in which case the data with the latest timestamp is canonical, or user driven, in which case both copies of the data can be returned to the application for resolution.

Once the first cluster is up, additional clusters can be installed in

other data centers and set up with cross data-center replication – this ensures that if one data center goes down, the remote cluster can take over the workload with minimal or no interruption to users.

Scalability

Aerospike scales up and out on commodity servers.

Each server node takes full advantage of the hardware available and is multi-threaded, multi-core, multi-cpu aware. Each server node also scales up to manage up to 16TB of data, with parallel access to up to 20 SSDs

Aerospike scales out linearly with each identical server added to parallel process 100TB+ across the cluster.

Synchronous replication within the cluster for immediate consistency, optionally supporting asynchronous replication.

Number of replicas from 1 to the number nodes in cluster, a typical deployment has 2 or 3 copies of each record

Asynchronous replication across clusters using Cross Datacenter Replication (XDR) where writes are automatically replicated across one or more locations. XDR has fine grained control - individual sets or namespaces can be replicated across clusters in master/master, master/slave with complex ring and star topologies.

Clustering

Aerospike uses distributed processing to ensure data reliability. In theory, many databases could actually get all of the required throughput from a single server with a huge SSD. However this would not provide any redundancy and if the server went down, all database access would stop. So a more typical configuration is several nodes, with each node having several SSD devices. Aerospike typically runs on fewer servers than other databases.

The distributed, shared-nothing architecture means that nodes can self-manage and coordinate to ensure that there are no outages to the user, while at the same time the cluster can be easily expanded as traffic increases. The Smart Client layer hides the complexity of the cluster from the developer, presenting a simple API for rapid development

Heartbeat

The nodes in the cluster keep track of each other through a heartbeat so that they can coordinate among themselves. The nodes are peers – there is no one node that is the master. All of the nodes track the other nodes in the cluster. When a node is added or removed, it is detected by the other nodes in the cluster using a heartbeat mechanism. Aerospike has two ways of defining a cluster

Multicast: In this mode, a valid multicast protocol IP:PORT is used to broadcast the heartbeat message.

Mesh: In this case, the IP address of one Aerospike server is used

as a seed to discover the cluster. Each node maintains a heartbeat connection to all other nodes, discovered via the seed node.

Where available, multicast is preferred.

High Speed Distributed Consensus

Aerospike's 24/7 reliability guarantee starts with its ability to detect cluster failure and quickly recovering from that situation to re-form the cluster. Once a cluster change is discovered, all surviving nodes use a Paxos-like consensus voting process to determine which nodes form the cluster and use the Aerospike Smart Partitions™ algorithm to automatically re-allocate partitions and re-balance. The hashing algorithm is deterministic – that is, it always maps a given record to the same partition. Data records stay in the same partition for their entire life although partitions may move from one server to another.

All of the nodes in the Aerospike system participate in a Paxos distributed consensus algorithm, which is used to ensure agreement on a minimal amount of critical shared state. The most critical part of this shared state is the list of nodes that are participating in the cluster. Consequently, every time a node arrives or departs, the consensus algorithm runs to ensure that agreement is reached.

This process takes a fraction of a second. After consensus is achieved, each individual node agrees on both the participants and their order within the cluster. Using this information the master node for any transaction can be computed along with the replica nodes.

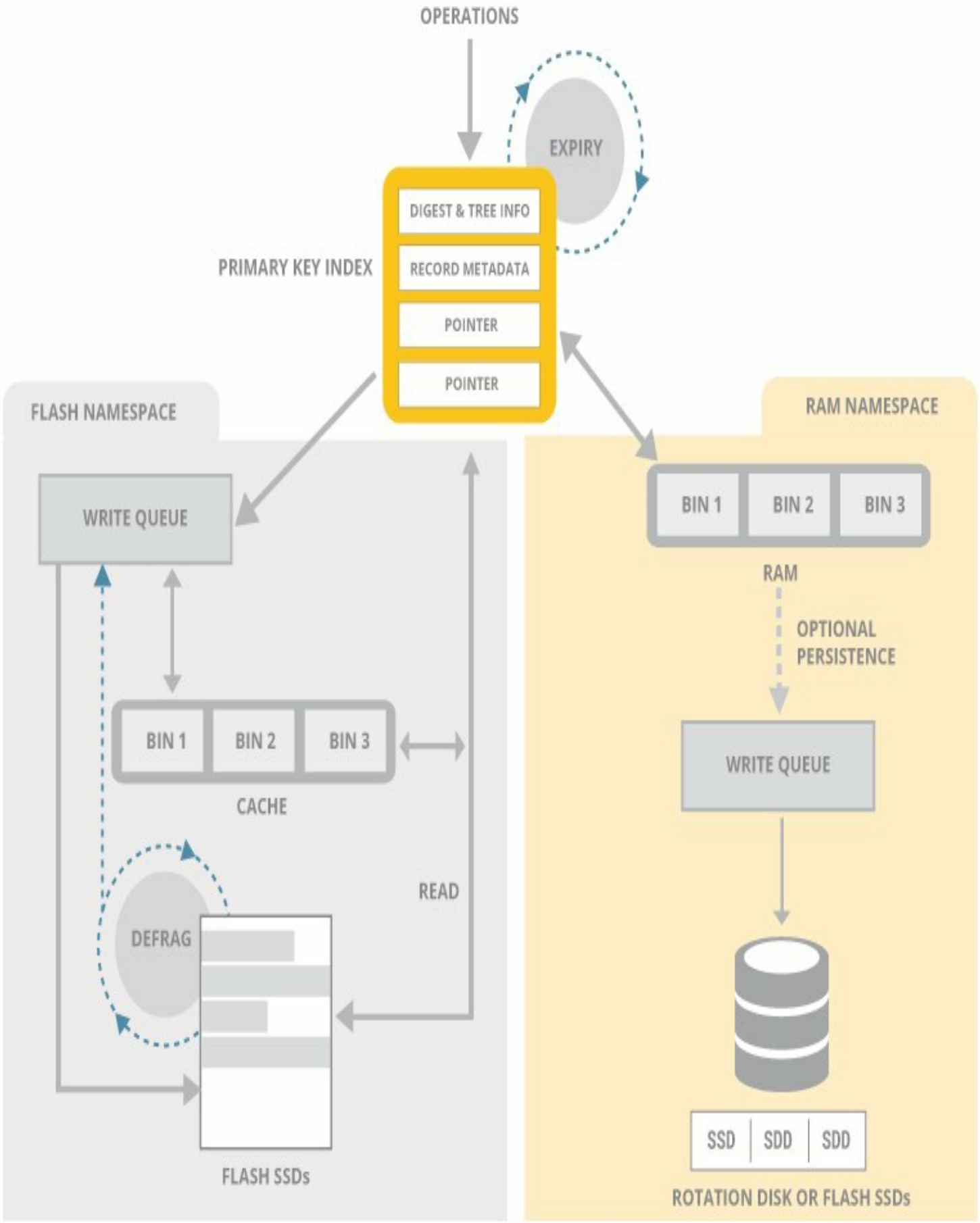
Data Storage Layer

Aerospike is a key-value store with a schema-less data model. Data is organized into policy containers called namespaces, semantically similar to databases in an RDBMS system. Within a namespace, 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.

Sets and Bins do not need to be defined up front, but can be added during run-time for maximum flexibility.

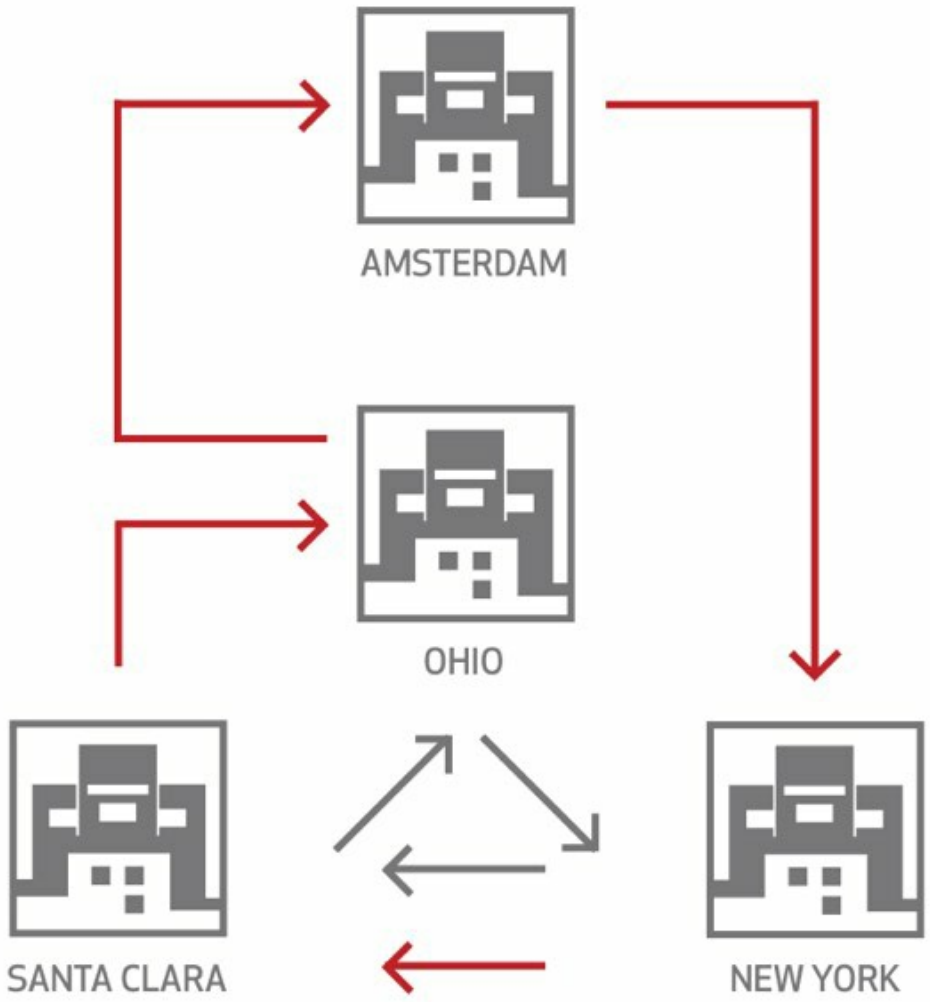
Values in Bins are strongly typed, and can include any of supported data-types. Bins themselves are not typed, so different records could have the same Bin with values of different types.

Indexes (primary keys and secondary keys) are stored in RAM for ultra-fast access and values can be stored either in RAM or more cost-effectively on Flash(SSDs). Each namespace can be configured separately, so small namespaces can take advantage of RAM and larger ones gain the cost benefits of Flash(SSDs).


OPERATIONS
EXPIRY
DIGEST & TREE INFO
RECORD METADATA
POINTER
POINTER
PRIMARY KEY INDEX
FLASH NAMESPACE
RAM NAMESPACE
WRITE QUEUE
BIN 1
BIN 2
BIN 3
RAM
OPTIONAL PERSISTENCE
BIN 1
BIN 2
BIN 3
CACHE
WRITE QUEUE
READ
DEFRAG
SSD
SDD
SDD
FLASH SSDs
ROTATION DISK OR FLASH SSDs

Cross Data Center Replication

XDR - Cross Datacenter Replication - is an Aerospike feature which allows one cluster to synchronize with another cluster over a longer delay link. This replication is asynchronous, but delay times are often under a second. Each write is logged, and that log is used to replicate to remote clusters.

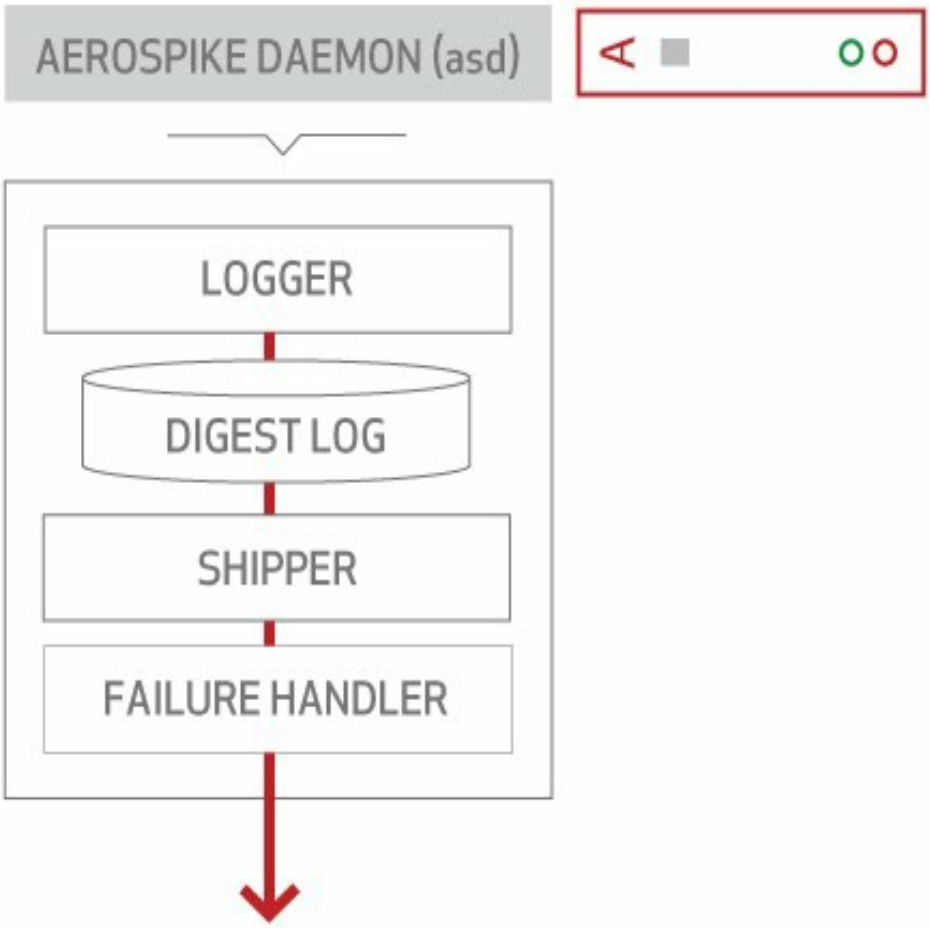


An XDR process runs on every node of the cluster along side the

Aerospike process. The main task of the XDR process is to monitor database updates on the "local" cluster and send copies of all write requests to one or more "remote" destination clusters. XDR determines the remote cluster details from the main Aerospike configuration files. It is possible to ship different namespaces (or even sets) to different clusters. In addition, the XDR process takes care of failure handling. Failures can be either local node failures or link failure to the remote cluster or both.

Each node runs:

- Aerospike daemon process (asd)
- XDR daemon process (xdr), which comprises:
  - Logger module
  - Shipper module
  - Failure Handler module



All the writes/updates that happen on each node in the local cluster need to be forwarded to one or more remote clusters. In order to do this efficiently, XDR uses a log of all digests of the keys that need to be shipped. When XDR writes to the remote cluster, it will get the current value of the key at that time. This is done so that if there are multiple changes to the same record, only the latest value is sent. XDR does not store any values of the record in the digest log.

The details of this are that writes and updates by the main database system are communicated with its XDR process over a named pipe. This communication is received by the Logger module which stores this information in the digest log. To keep

the digest log file small, only minimal information (the key digest) is stored in the digest log – just enough to ship the actual records at a later time. The Digest Log is a ring buffer file. i.e the most recent data will potentially overwrite the oldest data (if the buffer is full). This allows us to keep a check on this file's size. But this also introduces a capacity planning issue. The Digest Log file must configured big enough to avoid the loss of any pending data to be shipped and planning for long term communication outages between data centers.

Data Model

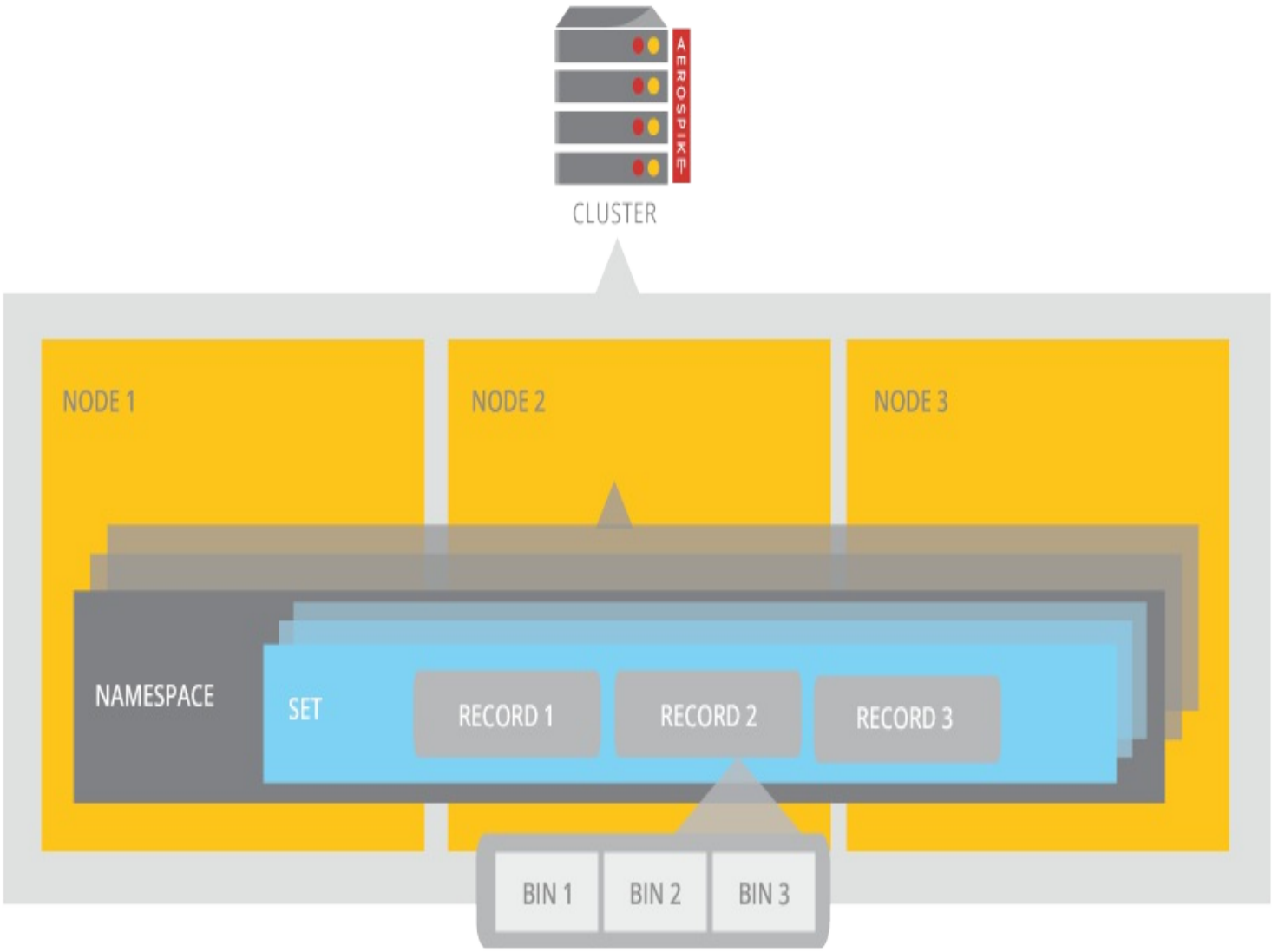
Aerospike has a schema-less data model, which means that the data stored in the database does not conform to a rigid schema.

This provides flexibility in how you store data in Aerospike – changes to the data do not require modifying a schema and data currently in the database does not need to conform to a specific schema.

Aerospike's schema-less model allows you to add new types of Bins on the fly. However you still need to be disciplined about Bin names and how you store your data. Applications must use Bins consistently in order for query and aggregation to work correctly.

Data Hierarchy

This is a high-level view of how data is organized in a cluster.



Namespace

Namespaces are the top level containers for data. A namespace can actually be a part of a database or it can be a group of databases. Collecting data into a namespace relates to how the data is going to be stored and managed. A namespace contains records, indexes and policies. A policy dictates the behavior of the

namespace, including:

How data is stored: RAM or Flash/SSD

How many replicas should exists for a record.

When records should expire.

Consistency

Namespaces can only be defined with a cluster wide restart. Hence, it is difficult to add or remove namespaces.

A database may specify multiple namespaces, each with differing policies, depending on the applications' needs. A namespace is considered a physical container, because it binds the data to a storage device, either in RAM with an optional backing store on disk, or directly to a Flash device (SSD).

Set

Within a namespace, records can belong to a logical container called **Set**. A set provides applications the ability to group records in collections. A Set is similar to a table as it is a collection of records, but it has no schema. Sets use the policy defined by the namespace to which they belong, such as storage, replication and consistency. A **Set** may define additional policies specific to the set, such as a secondary index definition. Set names are like a prefix to the primary key, and cannot be deleted or renamed. Sets can be iterated upon using the Scan operation. The total number of “named” sets that can be created in a namespace is 1023 in addition to 1 unnamed set (or null set).

Note: Moving a record from one Set to another may move it from one server to another.

Record

Aerospike Database is a row store so the focus is on the individual records (called rows in a standard RDBMS). A record is the basic unit of storage in the database and its Bins (columns) are stored contiguously. A mentioned earlier, records may belong to a namespace or a set within a namespace. A record is addressable via a key, which is used to uniquely identify the records in the namespace. A record comprises a Key/Digest, Metadata & Bin names and values.

Key / Digest

In the application, each record will have a **key** associated with it. This **key** is what the application will use to read or write the record. However, when the **key** is sent to the database, the **key** is hashed into a 160-bit **digest**. Within the database, the digest is used to address the record for all operations. The **key** is used primarily in the application, while the digest is primarily used for addressing the record in the database. The **key** may be either an Integer, String, or Bytes value.

Metadata

Each record is stored with metadata comprising:

Generation - reflects how many times the record has been modified. This number is handed back to the application on a read, and may be used to make sure that data that is being written has not been modified since the last read.

Time-to-live (TTL) - specifies how long the record will exist. Aerospike will automatically expire records based on their TTL. A

record's TTL is incremented each time a write-operation is executed on the object.

Aerospike	RDBMS
Namespace	Tablespace or Database
Set	Table
Record	Row
Bin	Column or Field

Bin

Bins are the equivalent of columns or fields in a conventional RDBMS. Within a record, data is stored in one or many Bins. A Bin consists of a name and a value. A Bin does not specify the type, instead the type is defined by the value contained in the Bin. This dynamic typing provides much flexibility in the data model. For example, a record may contain a Bin named "id" with a string value of "bob". The value of the Bin can always change to a different string value, but also to a value of a different type, such as the integer 72.

Records within a namespace or set are schema-less one record maybe composed of very different collection of Bins from another. Bins can be added or removed at any point in the lifetime of a record.

A Bin name is limited to 14 characters. There is a limit of 32K unique Bin names per namespace due to an optimized string-table implementation. If more Bin names are required, consider using a map. A map can store an arbitrary set of key-value pairs. These values can be efficiently accessed via a User Defined Function

(UDF).

DATA TYPES

Types are mapped to the equivalent language type. Methods on the Bin class convert the Aerospike type to the native language type.

Language	String	Integer	BLOB	List	Map
Java	String	Long	byte[]	List	Map
C#	String	Long	byte[]	List	Dictionary
C	as_string	as_integer	as_bytes	as_list	as_map
Ruby	String	FixedNum		Array	Hash
PHP	string	int	string	array	array
Python	string	long	bytearray	list	dict
Node.js	String	number	Uint8Array	array	Map
Go	string	uint64	[]byte	Slice	Map

String

Strings are stored as UTF-8, which is more compact than Unicode. In order to allow cross-language compatibility, client libraries convert from the native character set Unicode, to UTF-8, and back. A string is of any length and is limited only by the record size.

Integer

Integers are stored as 8 byte quantities, which limits integer

values in the current version. At the client, the integer is presented as a language specific “long”. Integers are useful for timestamps, counters and any numeric value.

Bytes (BLOB)

Bytes are byte arrays of a specific size. This allows any binary data of any type to be stored. Note: These are NOT null terminated. Binary objects (blobs) are efficient and limited in size only by the size of the record (default is 128K). Many deployments use their own serializer, compress the object and store the object directly. However, doing so means that data cannot be accessed easily through a UDF.

Language Serialized Objects: If a complex language type, such as a class in Java, is passed into a data call, the Aerospike client will use the language's native serialization system. That data will be stored with a "blob type" specific to the language. This allows a client of the same language to read the data readily.

Double

Aerospike stores a floating point number using the IEEE 754 format.

List

List is a collections of values ordered in the insert order. A list may contain values of any of the supported data types, including other Lists and Maps. The size is limited only by the maximum record size (default is 128K). A list is ideally suited to storing a JSON Array and is stored internally in a language neutral way. A list written in C# can be read in Python. A combination of Lists

and Maps can be used to store JSON documents. It is better to use a List rather than a String to store a JSON array.

The entire list must be stored or retrieved via the client API incurring network communication cost unless a User Defined Function is used to manipulate elements of the list.

Lists are rendered into msgpack for storage. Lists are serialized on the client, and sent to the server using the wire protocol. When used for simple get and put operations, the network format is written directly to storage without serializing or converting.

Map

Map is a collection of key-value pairs, such that each key may only appear once in the collection and is associated with a value. The key and value of a map may be of any of the supported data types, including other Lists and Maps. The size is limited only by the maximum record size (default of 128K).

A Map is ideally suited to storing a JSON Object and is stored internally in a language neutral way. A Map written in Java can be read in C#. A combination of Lists and Maps can be used to store JSON documents. It is better to use a Map rather than a String to store a JSON document.

The entire map must be stored or retrieved via the client API incurring network communication cost unless a User Defined Function is used to manipulate elements of the map.

Maps are rendered into msgpack for storage. Maps are serialized on the client, and sent to the server using the wire protocol. When used for simple get and put operations, the network format is written directly to storage without serializing or converting.

Developer setup

This chapter describes how to setup an developer environment for use with Aerospike and includes:

Install and Configure a single server development cluster on AWS, or your own computer

Manage your development cluster

Install the language specific Aerospike clients (drivers)

Server installation

The Aerospike Server is a linux only program. Specific binaries are available for RedHat derivatives, Ubuntu and Debian, with a generic Linux build as well. These distributions are available at:

http://www.aerospike.com/download/server/latest/

You should install the following server packages:

Aerospike server

Aerospike tools

Aerospike Management Console

Aerospike runs natively on Linux. To install the software you must have root/sudo privileges

To download the server software

```
wget -O aerospike-server.tgz
http://www.aerospike.com/download/server/latest/artifact/el6
```

To install the Aerospike server:

```
tar xvf aerospike-server.tgz
cd aerospike-server-community-<version>
sudo ./asinstall
```

To install AMC:

```
wget -O http://www.aerospike.com/download/amc/latest/
sudo rpm -ivh aerospike-amc-<version>.rpm
```

For OS X and Windows use Vagrant

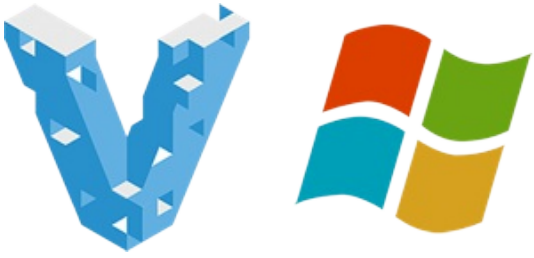
Vagrant

The Aerospike Server is a linux only program, for it to run on Windows, or OS X, it needs a Virtual Machine.

Aerospike provides a Vagrant box and Virtual Box configuration that includes the Aerospike Server, the Aerospike Tools and the Aerospike Management Console.

Vagrant boxes prebuilt with Aerospike are available on the Vagrant Cloud service. Vagrant is an open-source tool for building and managing virtualized development environments. Vagrant manages virtual machines running on Oracle VirtualBox, or VMWare. For more information on this tool, read Why Vagrant?

Windows



Aerospike is supported on Windows in a virtual machine managed by Vagrant. See Using Vagrant for an overview and installation instructions.

Once you have Vagrant installed, you are ready to install Aerospike, follow the instructions at:
http://www.aerospike.com/docs/operations/install/vagrant/win/

Vagrant requires each virtual machine to have its own working

directory. You can create the working directory anywhere.

```
mkdir ~/aerospike-vm && cd ~/aerospike-vm
```

Initialize the Aerospike Virtual Machine

```
vagrant init aerospike/centos-6.5
```

Start Aerospike and the AMC

```
vagrant up
```

Verify that Aerospike and AMC are running

```
$ vagrant ssh -c "sudo service aerospike status"
asd (pid XXXX) is running…
Connection to 127.0.0.1 closed.

$ vagrant ssh -c "sudo service amc status"
Retrieving AMC status.…
AMC is running.
Connection to 127.0.0.1 closed.
```

OS X


Aerospike is supported on OS X in a virtual machine managed by Vagrant. See Using Vagrant for an overview and installation instructions.

Once you have Vagrant installed, you are ready to install Aerospike, follow the instructions at:

http://www.aerospike.com/docs/operations/install/vagrant/mac/

Like on Windows,]=====\Vagrant requires each virtual

machine to have its own working directory. You can create the working directory anywhere.

Create an Aerospike Directory:

```
mkdir ~/aerospike-vm && cd ~/aerospike-vm
```

Initialize the Aerospike Virtual Machine

```
vagrant init aerospike/centos-6.5
```

To start up the virtual machine and asd, run the following from the command prompt:

```
vagrant up
```

Verify Aerospike and AMC are Running

```
vagrant ssh -c "sudo service aerospike status"
# asd (pid XXXX) is running…
# Connection to 127.0.0.1 closed.

vagrant ssh -c "sudo service amc status"
# Retrieving AMC status.…
# AMC is running.
# Connection to 127.0.0.1 closed.
```

Server operation

The Aerospike daemon can be controlled using the init script located at:

```
/etc/init.d/aerospike
```

which supports the following options:

start

coldstart

status

restart

stop

For Aerospike Enterprise Edition, `start` instructs the server to Fast Restart, if you are running Aerospike Community this command behaves the same as coldstart. For more information regarding restart modes, see Restart Modes on the Aerospike web site.

Starting and stoping

Controlling the server requires you to be root or have sudo privileges.

Start server

```
sudo service aerospike start
```

Check on server status

```
sudo service aerospike status
```

Stop server

```
sudo service aerospike stop
```

Restart server

```
sudo service aerospike restart
```

Command line tools

Tool	Description	Ac
asinfo	asinfo is a command-line utility that provides an interface to Aerospike's cluster command and control functions. This includes the ability to change server configuration parameters while the Aerospike is running.	a s
asadm	asadm is an interactive python utility primarily used to get summary info for the current health of a cluster and also for executing dynamic configuration and tuning commands across the cluster	the
aql	aql provides an SQL-like command line interface for database, UDF and index management. Aerospike does not support SQL as a query or management language, but instead provides aql to provide an interface similar to tools that utilize SQL.	the
asmonitor	asmonitor is the place to start when	the

	monitoring performance, tuning configuration settings and diagnosing issues with your cluster. These pages describe how to run asmonitor and the various commands that you can use from the asmonitor console to monitor your Aerospike Database.	
ascli	scli provides a set of commands (get, put, remove, etc) that can be executed against an Aerospike database.	the
asloglatency	asloglatency analyzes Aerospike log files and returns the latency measurements. It returns latency analysis for a given time period in a tabular, easy to read form. The utility analyzes a histogram by parsing latency log lines during successive time slices, and calculating the percentage of operations in each time slice that exceeded various latency thresholds.	a s
asbackup	asbackup is used to backup namespaces or sets from an Aerospike cluster to local storage.	the
asrestore	Restores data backed up via asbackup.	the

Aerospike Management Console

The Aerospike Management Console (AMC) is a Web based tool to monitor/manage an Aerospike cluster. It provides live updates to the current status of a cluster. It comes in 2 editions.

The AMC Community Edition is free for use to anyone. It includes features to let you see at a glance the throughput, storage usage, and configuration of a cluster.

For those on the Enterprise Edition of the database, there is an Enterprise Edition of the AMC that includes additional features for additional alerts and management of a cluster.

Installing AMC

Aerospike Management Console is developed and optimized for Linux. Aerospike provides packaging for some of the popular Linux distributions and a generic tar.gz which can be used on any Linux distribution, it also can be installed on OS X.

RedHat and CentOS

OS X

Starting and stoping

The Aerospike Management Console (AMC) is used to see how the server is doing. To start or stop the AMC, login (ssh) to the server where AMC is installed and use these commands:

Start server

```
sudo service amc start
```

Check on server status

```
sudo service amc status
```

Stop server

```
sudo service amc stop
```

Restart server

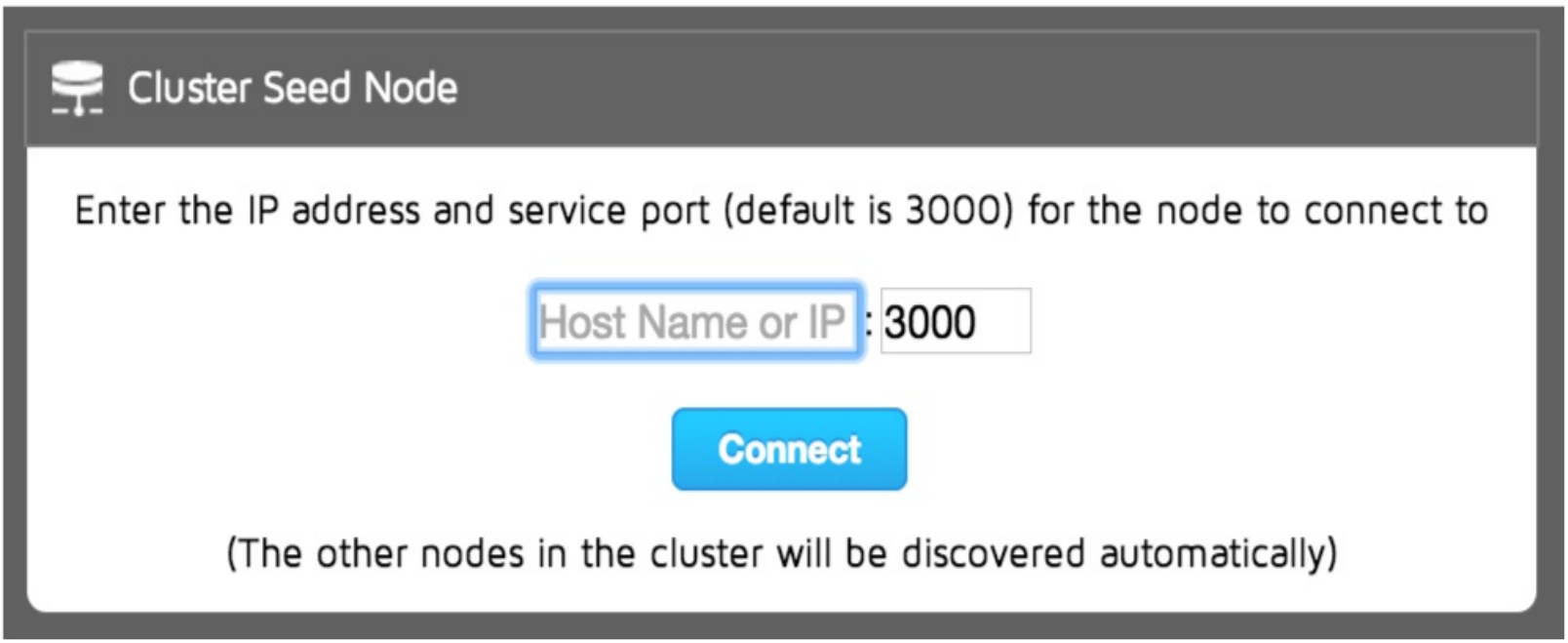
```
sudo service amc restart
```

Connecting to AMC

In a browser, connect to the AMC using the following URL:

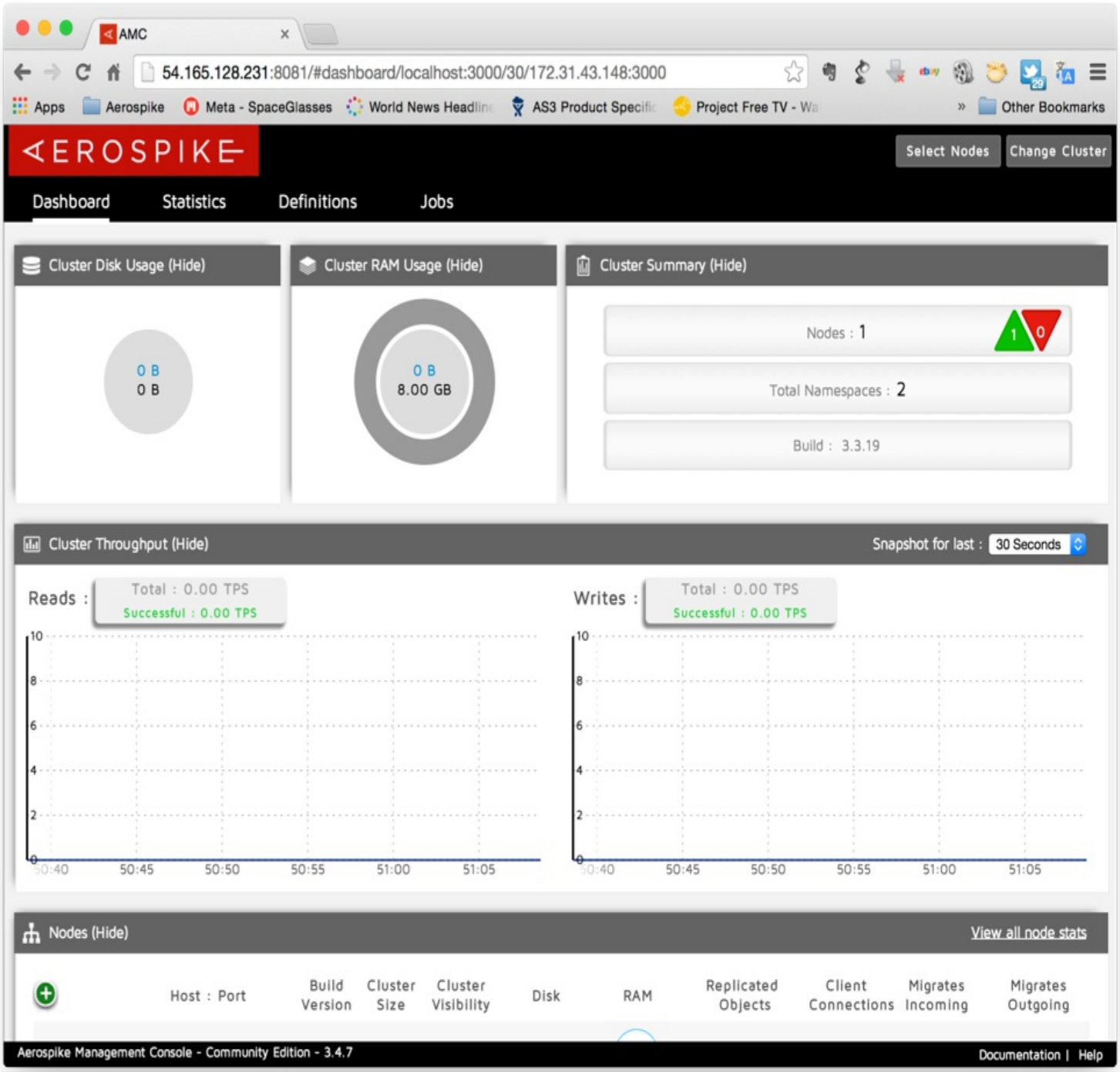
```
http://<ip address or host name where AMC is running>:8081
```

When AMC first connects to a cluster, you will be prompted for a node address in the cluster.



Enter an IP address or DNS name of one of the nodes in the cluster. Once AMC connects to the cluster you will see a screen

similar to this:



Please refer to the AMC user guide for details.

Logging

Server logging

The Aerospike server logs numerous server related events. These are logged by sub-system and severity.

It is always helpful to be able to look at the logs of the server. You must have root/sudo privileges to see the logs in the default location:

```
/var/log/aerospike/aerospike.log
```

It is often useful to keep a window open with a tail of the log:

```
sudo tail -f /var/log/aerospike/aerospike.log
```

Log entries are used to debug User Defined Functions (UDFs). To log UDF entries in a separate log file, do the following:

Edit the Aerospike configuration file:

```
/etc/aerospike/aerospike.conf
```

Locate the “logging” stanza and modify it to look like this:

```
logging {
 file /var/log/aerospike/aerospike.log {
  context any warning
  }
  file /var/log/aerospike/udf.log {
  context any critical
  context udf info
  context aggr info
 }
}
```

Restart Aerospike (sudo service aerospike restart)

Server logging levels

Every message produced by the server has a context (which names the part of the code that generated the message), and a severity level (which describes how acute the message is). The severity levels are as follows:

Severity	Meaning
critical	Critical error messages
warning	Warning messages
info	Informational messages
debug	Debugging information
detail	Verbose debugging information

Use this command to see the log locations on a server

```
asinfo -v logs
```

it responds with

```
0:/var/log/aerospike/aerospike.log;1:/var/log/aerospike/udf.log
```

Client logging

Client logging is enabled, in code and in each language client instance.

Here is an example of enabling client logging in Java to pass the Aerospike log messages to log4j:

```
private static Logger log = Logger.getLogger(FailoverTest.class);
```

```
• • •

Log.setCallback(new Callback() {

 @Override
 public void log(Level level, String message) {
  switch(level){
  case INFO:
   log.info("INFO:"+message);
   break;
  case DEBUG:
   log.debug("DEBUG:"+message);
   break;
  case ERROR:
   log.error("ERROR:"+message);
   break;
  case WARN:
   log.warn("WARN:" +message);
   break;
  }

 }
});
```

Java setup

Aerospike Java Client requires Java SDK 6 or newer. Oracle Java 8 is recommended. If Oracle Java SDK is not already installed, download Oracle Java SDK and follow instructions to install.

There are multiple ways of getting hold of the Aerospike Java Client Library:

- Adding the Aerospike Jar as a dependency.
- Download and install from the Aerospike Download page. The download also includes the benchmark tool and example code.
- Download and build from source GitHub

Add Aerospike Jar as Dependency

The Aerospike Java Client jar is available from the central maven repository. You can use the build or project tool of your choice to define the dependency on Aerospike Java Client.

If Maven is not already installed, Download Maven and follow instruction in the downloaded package to install Maven.

```
<dependencies>
  <dependency>
    <groupId>com.aerospike</groupId>
    <artifactId>aerospike-client</artifactId>
    <version>[3.1.0,)</version>
  </dependency>
</dependencies>
```

Eclipse plugin

Optionally, you can install Aerospike Developer Tools Eclipse plugin (Luna or better). The plugin includes:

Cluster Explorer

AQL editor, executor and code generator

UDF Management

Data browser

To install:

Add an Update site for the Aerospike Developer Tools

https://github.com/aerospike/eclipse-tools/raw/master/aerospike-site

Install Aerospike Developer Tools

C# setup

NuGet

The compiled library is also available via NuGet. To install, run the following command in the Package Manager Console:

```
PM> Install-Package Aerospike.Client
```

Download

Download Aerospike C# Client package and unzip files into a local folder. This package contains full source code for:

Aerospike.sln contains the C# client library and demonstration programs with full functionality. Supported compile targets are AnyCPU, x64 (64-bit) and x86 (32-bit). The projects are:

AerospikeClient - C# client library.

AerospikeDemo - C# demonstration program for examples and benchmarks.

AerospikeAdmin - Aerospike user administration. This application is only valid for enterprise servers that are configured to require user authentication.

The solution supports the following configurations:

Debug

Release

Debug IIS : Same as Debug, but store reusable buffers in

HttpContext.Current.Items.

Release IIS : Same as Release, but store reusable buffers in HttpContext.Current.Items.

Build

Follow these steps to build the solution:

Double click on Aerospike.sln. The solution will be opened in Visual Studio.

Click menu Build -> Configuration Manager.

Click desired solution configuration and platform.

Click Close.

Click Build -> Build Solution.

Go setup

Aerospike Go Client requires Go version v1.2+. (It is possible to build the code in Go versions prior to 1.2, but our testing library depends on v1.2)

To install the latest stable version of Go, visit http://golang.org/dl/

Aerospike Go client implements the wire protocol, and does not depend on the C client. It is goroutine friendly, and works asynchronously.

Supported operating systems:

- Major Linux distributions (Ubuntu, Debian, Redhat)
- Mac OS X
- Windows (untested)

Installation

Install Go 1.2+ and setup your environment as Documented here.

Get the client in your GOPATH :

```
go get github.com/aerospike/aerospike-client-go
```

To update the client library:

```
go get -u github.com/aerospike/aerospike-client-go
```

NOTE: Be sure to set your GOPATH correctly

Node.js setup

The Aerospike Node.js Client uses the Aerospike C Client. Currently, it is available on 64-bit Linux variant as well as on 64-bit Mac OS 10.8 and above.

Aerospike Node.js Client library requires Node.js 0.10.x. Node.js can be installed by visiting http://nodejs.org

To install aerospike as a dependency of your project, run:

```
npm install aerospike
```

If you want to add aerospike dependency to your package.json, then run:

```
npm install aerospike --save-dev
```

PHP

Source download

Download the latest source from the GitHub repository: https://github.com/aerospike/aerospike-client-php/releases/latest

Unzip the files to a local folder. The package contains the source code for building the PHP extension. Alternatively you can use Composer (https://getcomposer.org/):

```
composer require aerospike/aerospike-client-php "*"
```

OSX

You need command line tools to compile the extension:

```
xcode-select --install
```

Use Homebrew to install the prerequisites. Install brew from: http://brew.sh/

The PHP client has dependencies on several packages, you will need to install the before building the Aerospike client, and they are required for your application in production.

Enter the following commands:

```
brew update && brew doctor
brew install automake
brew install openssl
brew install lua
```

RedHat Linux

RedHat Linux and variants such as CentOS use the yum package manager.

```
sudo yum groupinstall "Development Tools"
sudo yum install openssl-devel
sudo yum install lua-devel
# on Fedora 20+ use compat-lua-devel-5.1.5
sudo yum install php-devel php-pear
```

Debian

Debian and Ubuntu Linux use the apt package manager.

```
sudo apt-get install build-essential autoconf
sudo apt-get install libssl-dev
sudo apt-get install liblua5.1-dev
sudo apt-get install php5-dev php-pear
```

Build

A build script gets the latest version of the C client and compiles the source:

```
cd src/aerospike
./build.sh
```

After compilation, the build script will provide further instructions, please follow them.

One instruction is for creating a PHP config (.ini) for the new module. Example:

```
extension=aerospike.so
aerospike.udf.lua_system_path=/usr/local/aerospike/client-php/sys-lua
aerospike.udf.lua_user_path=/usr/local/aerospike/client-php/usr-lua
```

Install

Once the build script compiles the extension you need to install it:

```
sudo make install
```

Confirm that the module loads for your PHP interpreter:

```
php -m | grep aerospike
```

Documentation

The API documentation for the client is in the GitHub repo:

https://github.com/aerospike/aerospike-client-php/blob/master/doc/aerospike.md

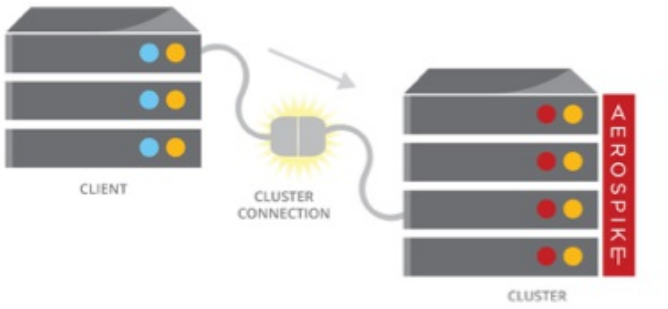
See also the Quick Guide

Key-value operations

In this section, we present a brief overview of the client API features available to developers. Please refer to detailed API documentation at http://www.aerospike.com/docs/. and comprehensive free online developer and deployment training is available at http://www.aerospike.com/training/

Connecting to Cluster

Connecting to the cluster involves connecting to any one node and auto-discovering the cluster nodes.



Initial Nodes

Create a **AerospikeClient** object, or structure, specifying the IP address and port of one or more nodes in the cluster. The client makes initial contact to the specified node and then discovers all the nodes in the cluster. The best practice is to specify several nodes in the cluster when creating the client. The client will iterate through the array of nodes until it successfully connects to a node. Then through that node, the client will discover the other nodes in the cluster. The client creates a maintenance thread which periodically pings nodes for the current state of the cluster.

Example code:connecting

```
// C# connection to a cluster
string asServerIP = "172.16.159.152";
int asServerPort = 3000;
AerospikeClient client = new AerospikeClient(asServerIP,
                                    asServerPort);

// Go connection to a cluster
client, err := NewClient("54.163.76.64", 3000)

// Java connection using multiple nodes
```

```
Host[] hosts = new Host[] {
  new Host("a.host", 3000),
  new Host("another.host", 3000),
  new Host("and.another.host", 3000)
};
AerospikeClient client =
              new AerospikeClient(new ClientPolicy(), hosts);

/* PHP connection using multiple nodes */
$config = array("hosts" => array(
                                    array("addr" => $HOST_ADDR,
                                    "port" => $HOST_PORT)));
$client = new Aerospike($config, false);
if (!$client->isConnected()) {
    echo standard_fail($client);
    . . .
    exit(2);
}
```

Thread Safe

The AerospikeClient instance is thread-safe and can be used concurrently. Each get/set call is a blocking, synchronous network call to Aerospike. Connections are cached with a connection pool for each server node.

Cleanup

When all transactions are finished and the application is ready to have a clean shutdown, call the close method to remove the resources held by the client instance. The close() function will:

- Close socket connections to every node in the cluster
- Free object references used by the Client instance
- Terminate the monitor thread
- Terminate threads in the thread pool and remove the pool.

The client instance is no longer usable once close() has been called.

Example code:close

```
// C# Close
client.Close();

// Java Close
client.close();

// Go Close
defer client.Close()

/* PHP Close */
$client->close();
```

Client Policy

A Client Policy is container object/structure for attributes used in creating a connection to an Aerospike cluster. It modifies the default behavior when a Client instance connects to a cluster.

Attribute	Type	Description
Connection timeout	Integer	Initial host connection timeout in m when opening a connection to the s time.
Fail if not connected	Boolean	Throws an exception if host connec
Max socket idle	Integer	Maximum socket idle in seconds
Max threads	Integer	Maximum number concurrent threa methods in the client instance
Tend interval	Integer	Interval in milliseconds that the clie cluster state.

User	String	User authentication to cluster (Ent
Password	String	Password authentication to cluste stored by the client and sent to serv
Default policies		Default policies for Read, Write, Qu

Example code:client policy

```
// Java connection with Client policy
ClientPolicy clientPolicy = new ClientPolicy();
clientPolicy.maxThreads = 200; //200 threads
clientPolicy.maxSocketIdle = 3; // 3 seconds
AerospikeClient client = new AerospikeClient(clientPolicy,
              "a.host", 3000);
```

Recommendation: The default values for connecting to Aerospike have been established through extensive testing, and are appropriate for most use cases. It is recommended to change these attributes in consultation with Aerospike support.

Key

A Key is an object, or structure, that represents the primary key and it consist of:

Attribute	Description
Namespace	String representing the Namespace name
Set	String representing the Set name, if omitted rec will be in the “null” Set
Key	The value of the primary key
Digest	20 byte hash of the key value, The Digest can be place of the primary key

Digest

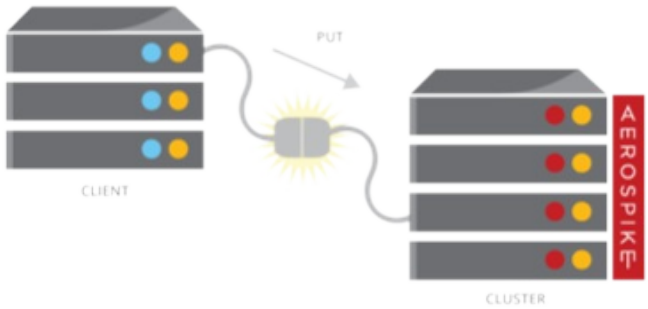
The primary key value is hashed using th RIPEMD160 algorithm to produce a 20 byte (160 bit) digest. RIPEMD160 always produces a 20 byte digest regardless of the size of the primary key. By default, Aerospike does not store the value of the primary key on the server, it stores the digest.

Some Aerospike operations (scan, query, aggregate, etc) return a Key object (or structure). This object does not, by default, contain the original key, it contains the 20 byte digest.

Write operations

Put

The put operation writes a record. You need to identify the record in the database via a key. A key can be a string, integer or blob. The Set and Bins will be created automatically if they do not exist. One or more Bins can be updated in the record by a Bin or an array of Bins to the put() call. To delete a Bin, simply set the Bin value to be NULL.



Default behavior

Create the record if the record doesn't exist.

Update Bin values in the record if the Bins already exist.

Add the Bins if they do not exist.

If the record already has Bins, keep them in the record

Example code: put

```
// Java write
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
wPolicy.expiration = 5; // Time To live of 5 seconds

Key key = new Key("test", "users", username);
```

```
Bin bin1 = new Bin("username", username);
Bin bin2 = new Bin("password", null);

client.put(wPolicy, key, bin1, bin2);
```

```
// Go Write record
wPolicy := NewWritePolicy(0, 0) // generation, expiration
wPolicy.RecordExistsAction = UPDATE

key, _ := NewKey("test", "users", username)
bin1 := NewBin("username", username)
bin2 := NewBin("password", password)

err := client.PutBins(wPolicy, key, bin1, bin2)
panicOnError(err)
```

```
// PHP Write Record
$bins = array('username' => $username);
$bins['password'] = $password;
$key = $this->client->initKey('test', 'users', $username);
$status = $this->client->put($key, $bins);
```

WritePolicy

The policy specifies the transaction's re-transmission policy, timeout interval, record expiration, and what to do if the record already exists. Instantiating the policy initializes client defaults

To change the write behavior through the write policy object. For example: Only write if the record does not already exist or completely replace a record, ONLY if it exists

Write Record with Time-to-live (TTL)

It is possible to specify a time-to-live value for a record on a write. The expiration units are seconds.

```
wPolicy.expiration = 5; // Time To live of 5 seconds
```

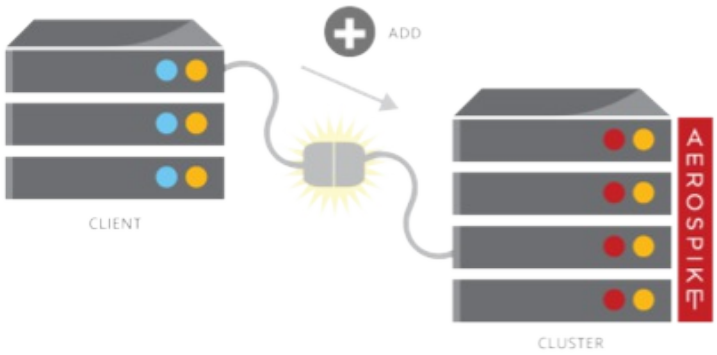
If a 0 is specified, then the default time-to-live specified on the server side will be applied each time a record is updated. If a -1 is specified, then the record will never be expired.

Replica Writes

By default a Write to replicas are immediate and use the replication count configured in the Namespace. Replica writes can optionally be controlled on a per operation basis.

Add

The Add operation increments, or decrements an integer Bin value(only) by the amount specified. An integer Bin is 8 bytes unsigned. To decrement, you supply a negative value.



One or more Bin parameters are passed to the operation and all increments/decrements are atomic in the operation on the specified record.

The write policy specifies the transaction timeout, record expiration and how the transaction is handled when the record already exists.

Example code: add

```
// C# Add
Key userKey = new Key("test", "users", "user1234");
Bin counter = new Bin("tweetcount", 1);
client.Add(null, userKey, counter);
```

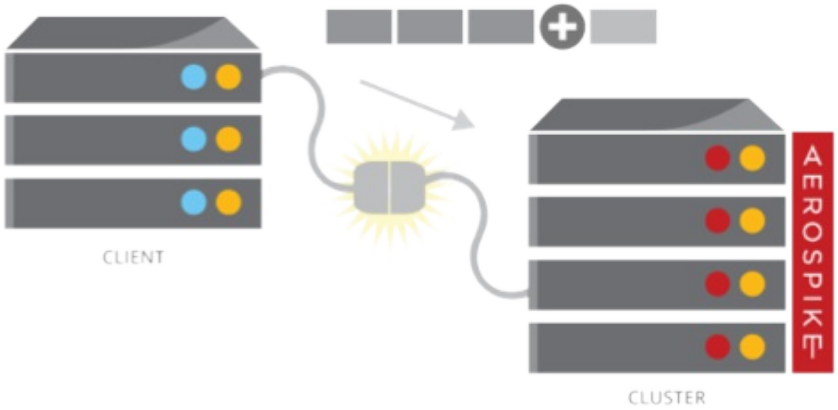
```
// Java add
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
Bin counter = new Bin("tweetcount", 1);
client.add(wPolicy, key, counter);

// PHP add
$this->client->add(null, $key, $bin);
```

Counters

Many applications require a counter, these are simple to implement in Aerospike. In the section Key-value techniques we cover how to increment an integer and read it's new value atomically.

Append and Prepend

The append operation adds a string to the end of a string or byte array **Bin** value, the prepend operation adds the value to the start of the value. One or more **Bin** parameters are passed to the operation and the all appends/prepends are atomic in the operation on the specified record.



If the bin does not exist, it will be created.

Example code:append/prepend

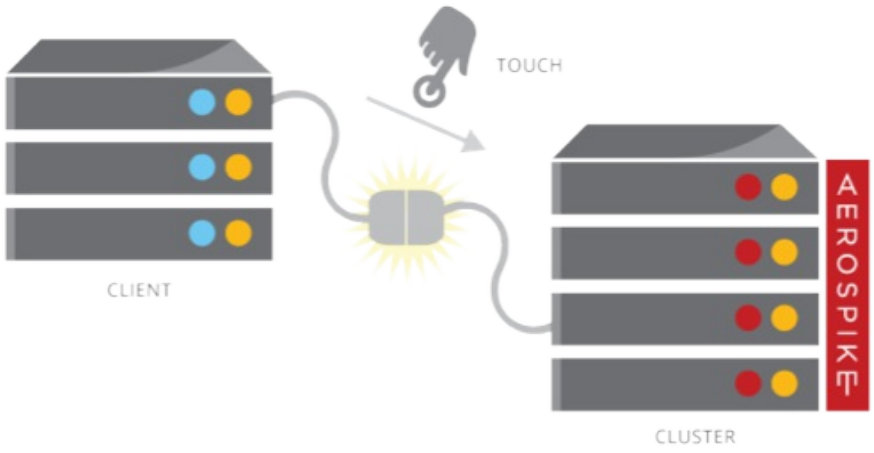
```
// C# Append
Key userKey = new Key("test", "users", "user1234");
```

```
Bin bin2 = new Bin("greet", " world");
client.Append(null, userKey, bin2);

// Java append
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", "user1234");
Bin bin2 = new Bin("greet", " world");
client.append(wPolicy, key, bin2);

// C# Prepend
Key userKey = new Key("test", "users", "user1234");
Bin bin2 = new Bin("greet", "hello ");
client.Prepend(null, userKey, bin2);

// Java prepend
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", "user1234",);
Bin bin2 = new Bin("greet", "hello ");
client.prepend(wPolicy, key, bin2);
```

Note: If you append a String to an Integer Bin you will get an error.

Touch

The touch operation is a light weight write that touches one, or more, records, resetting the time to expiration. This is similar to the unix touch command where you change the modification date on a file by *touching* it.



Providing more than one Key will allow you to *touch* many records

in one operation.

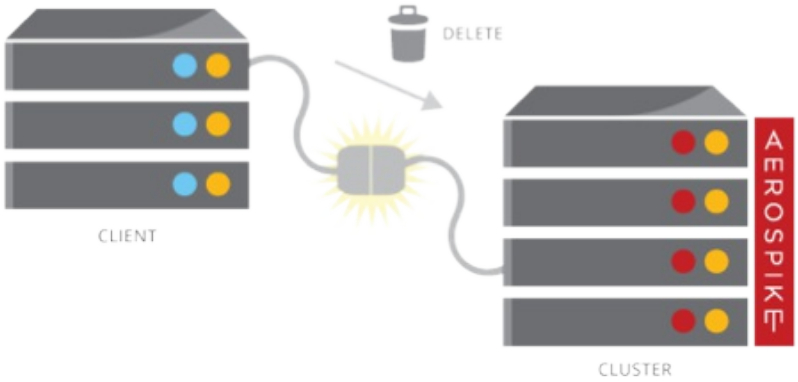
Example code:touch

```
// C# Touch
Key userKey = new Key("test", "users", "user1234");
Key userKey2 = new Key("test", "users", "user1235");
client.Touch(null, userKey, userKey2);

// Java touch
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.touch(wPolicy, key);

// PHP touch
$key = $this->client->initKey('test', 'users', $username);
$status = $this->client->touch($key, 500);
```

Note: Multi–record touch operation is not atomic.

Delete

When a record is deleted, the index entries in the primary index, and any secondary indexes, are removed.

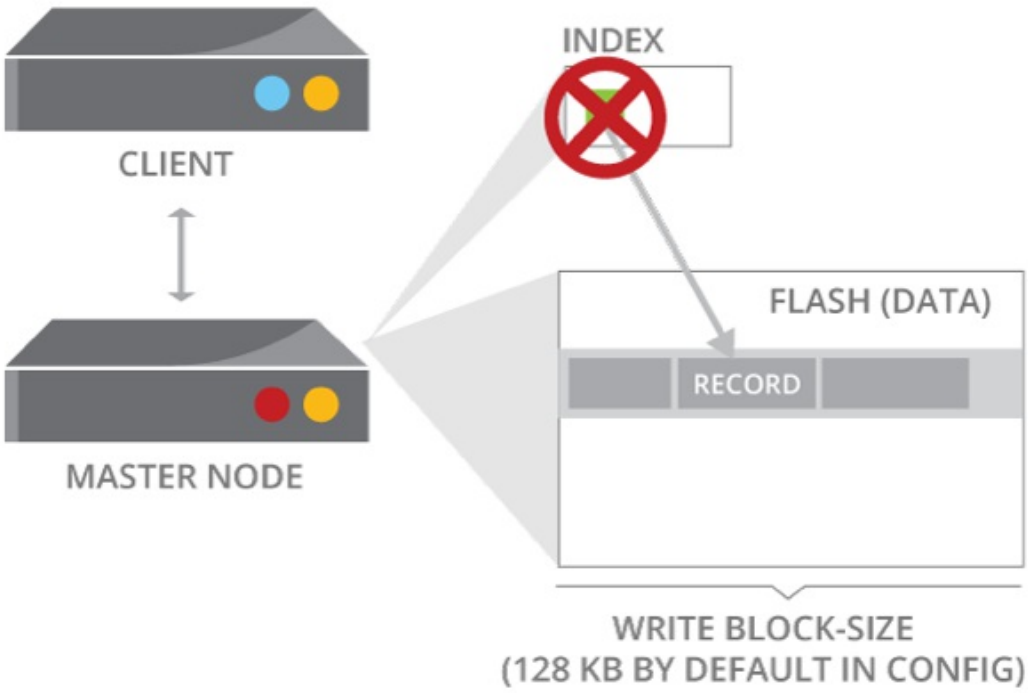


At some future time, the automatic defragment process (thread) will remove the record from the storage layer when it defragments the storage block where the record is located. This has the unusual side effect that a deleted record can be “resurrected” on a server restart if the server is configured to load data when it starts.

Example code:delete

```
// C# Delete record.
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.Delete(wPolicy, key);


// Java Delete record.
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
client.delete(wPolicy, key);

// PHP delete
$key = $this->client->initKey('test', 'users', $username);
$this->client->remove($key);
```

Deleting a record from Flash removes the entries from the indexes only, which is very fast, but does not tombstone deleted records.The background defragmentation will physically delete the record at a future time.



The process:

Client finds Master Node from partition map.

Client makes delete request to Master Node.

Master node removes the entry from index(es)

Master Node coordinates deletion with replica nodes (not shown).

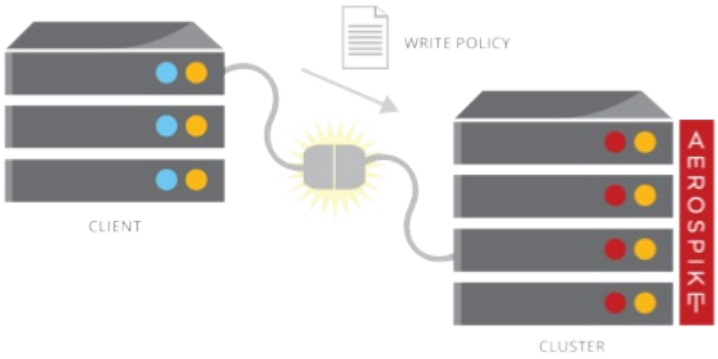
Master Node returns success to client.

Data is eventually deleted from flash by defragmentation.

Note: If a server is restarted AND data is reloaded from storage, there is potential for “deleted” records to be “resurrected”. This only occurs if the defragmentation task has not processed a storage block containing the deleted record.

Write Policy

WritePolicy’s purpose is to modify the default behavior of a write operation and is optional. If omitted, the default WritePolicy, contained in the ClientPolicy, will be used to control write operations.



A WritePolicy is a data structure only and you can use it as many times as you like. Attributes of a WritePolicy are as follows:

Attribute	Description
Timeout	Total transaction timeout in

	milliseconds for both client and server.
Expiration	Seconds record will live before being removed by the server.
Generation	Expected generation. Generation is the number of times a record has been modified on the server
Generation policy	how to handle record writes based on record generation
Record exists action	how to handle writes where the record already exists
Commit level	consistency guarantee when committing a transaction on the server

Example code:WritePolicy

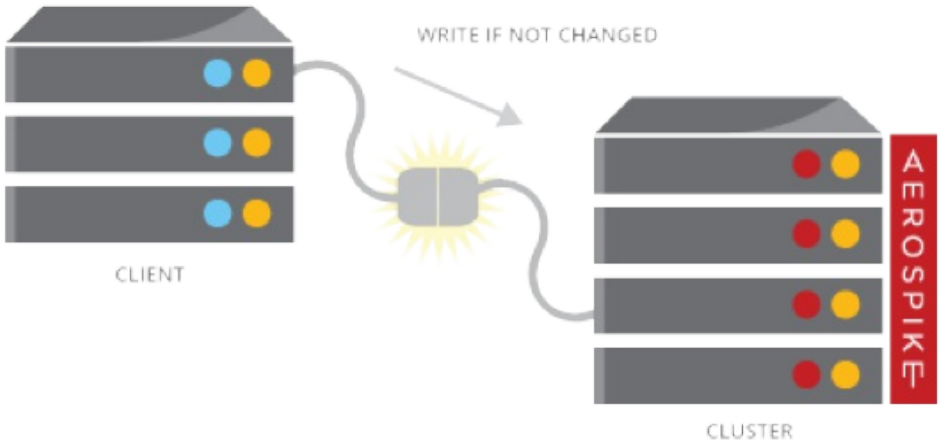
```
// C# write policy
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
wPolicy.timeout = 50; // 50 milliseconds

// PHP write policy
$status = $db->put($key, $new_bin, 0,
array(Aerospike::OPT_POLICY_GEN=>array(Aerospike::POLICY_GEN_EQ,
$current_gen)));

// Java write policy
WritePolicy wPolicy = new WritePolicy();
wPolicy.recordExistsAction = RecordExistsAction.UPDATE;
wPolicy.timeout = 50; // 50 milliseconds
```

The WritePolicy is quite powerful. We will cover in detail how to use these attributes.

Write Policy: Generation Policy

Aerospike uses multi-version concurrency control (optimistic

concurrency). The record generation indicates how many times the record has been written and is incremented on each write.



The generation policy is a field in the write policy. Valid values are:

Value	Description
NONE	Do not use record generation to rest
EXPECT_GEN_EQUAL	Update/delete record if expected gen equal to server generation. Otherwis
EXPECT_GEN_GT	Update/delete record if expected gen greater than the server generation. C fail. (Used by restore utility)

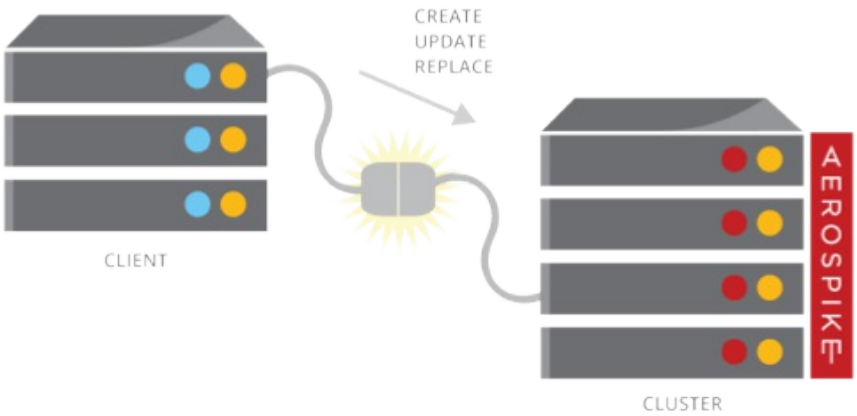
A generation policy is commonly used in a read-modify-write pattern where the generation policy is set to EXPECT_GEN_EQUAL, and the generation attribute on the WritePolicy is set to the read record generation. The write operation will expect the record generation to be equal to the value read previously, this would indicate that the record has not been written by another operation. If the value is different the

write operation will fail.

See Key-value techniques

Write Policy: Record Exists Action

The RecordsExistsAction specifies the action to take when writing to a Record that already exists.



The valid policy values are:

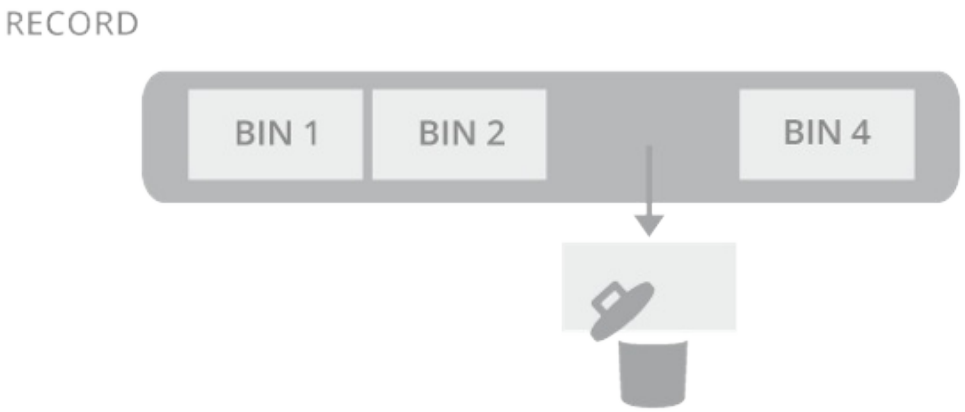
Value	Description
Update	This is the default and in this mode the write o update the record.
Update Only	This option will only update and existing recor not exist. This is similar to the UPDATE verb i
Create Only	This option will create a record if it does not ex exists. This is similar to the INSERT verb in SC
Replace	This option creates or replaces all the Bins in a are ignored and deleted. It has a performance g

	the record does not need to be read from storag
Replace Only	This option is the same as above with the addit fail if Record does not exist.

Note: Replacing a whole record has better storage performance than replacing a subset of Bins in a record. When you know that you are replacing the whole record, please use the REPLACE flag.

Deleting a Bin

To delete one or more Bins in an operation, set the Bin value to null.



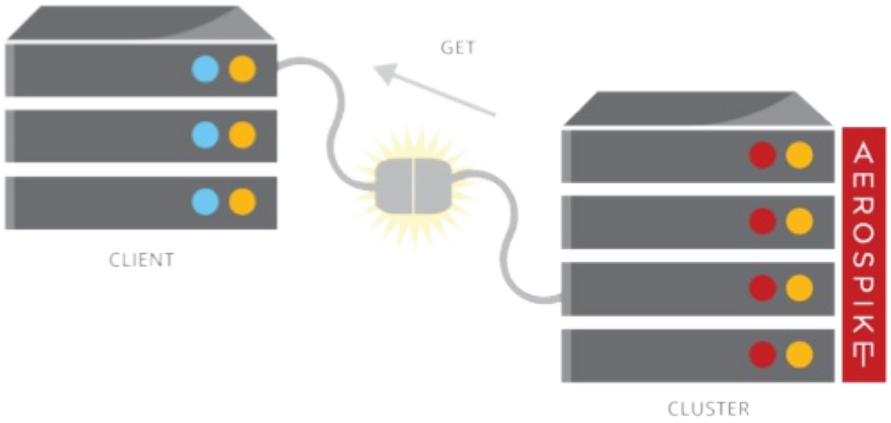
Deleting Bins has a similar storage cost of writing Bins in a Record

Example code:delete a bin

```
// C# delete a bin
WritePolicy wPolicy = new WritePolicy();
Bin bin1 = Bin.asNull("shoe-size"); // Set bin value to null to drop bin.
client.Put(wPolicy, key, bin1);


// Java delete a bin
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
Bin bin1 = Bin.asNull("shoe-size"); // Set bin value to null to drop bin.
client.put(wPolicy, key, bin1);
```

Read operations

Read a Record



The read operation allows you to:

Read all Bins in a Record. Remember – a Bin is like a column.

Read specified Bins in a Record.

Read only the Generation and Expiration of a record.

Each of the methods returns a Record, which contains Expiration, Generation, and Bins of the record.

To read all the Bins of a Record, call the `get()` method without a list of Bins. This will read the entire Record, including Generation and Expiration from the database, and return a Record Object.

To read specific bins of a Record, call the `get()` method and specify the names of the Bins to be returned.

The type returned is a Record and it consists of:

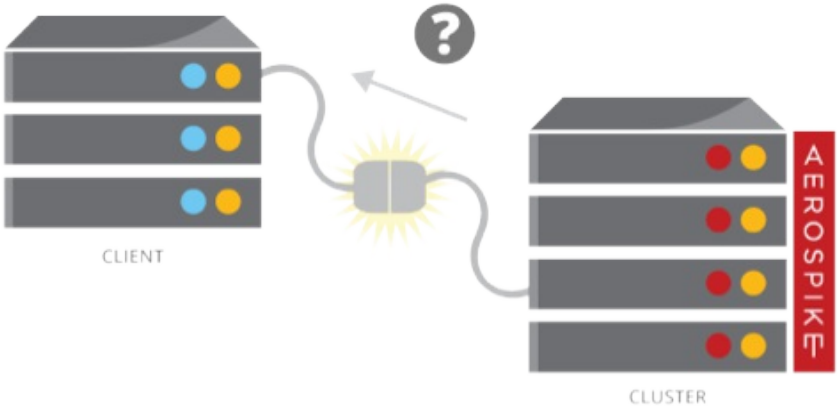
Attribute	Type	Description
Expiration	Integer	the date record will expire, in seconds f 00:00:00 GMT
Generation	Integer	a count of the modifications to the reco
Bins	Map	A map/dictionary of requested name/va

Example code: get

```
// C#
Key userKey = new Key("test", "users", username);
Record record = client.Get(null, userKey);

// PHP
$status = $db->get($key, $record);
if ($status == Aerospike::OK) {
 . . .

// Java read specific bins
Key key = new Key("test", "users", userName);
Record record = this.client.get(null, key,
  "username",
  "password",
  "gender",
  "region");

// Java get header data
Key key = new Key("test", "users", userName);
Record record = this.client.getHeader(null, key);
```

Exists

This operation is a light weight alternative to reading the Record. It checks the existence of a Record and returns a Boolean.



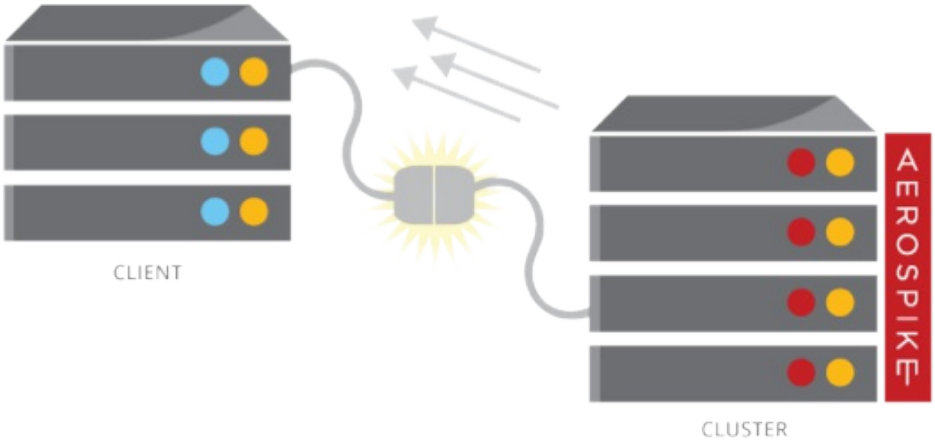
The Exists operation does not hold a lock, it tests existence at the time you call it. Checking to see if a Record exists and then reading it (if true) is not a sensible approach. Why? Because another client instance could have successfully deleted the Record between your exists and read operations.

Code example:exists

```
// C# Exists
Key userKey = new Key("test", "users", "user1234");
bool recordKeyExists = client.Exists(null, userKey);

// Java exists
Key key = new Key("test", "users", username);
boolean itsHere =  client.exists(policy, key);

// PHP exists
$status = $db->exists($key, $metadata);
if ($status == Aerospike::OK) {
 . . .
```

Batch Reads

In addition to reading single Record each time, it is also possible to read multiple Records from the cluster in one operation.



Internal to the client, the operation will group the keys based on which Aerospike Server node is best to handle the request, and uses a thread pool to handle the requests to all nodes concurrently. All network requests are combined efficiently and sockets are reused.

After all the nodes have returned the Record data, the Records will be return to the caller in an array of Records in the same order as the Keys passed in. If a Record is not found on the database, the array entry will be NULL.

This is not an atomic operation, Records are read as they are found.

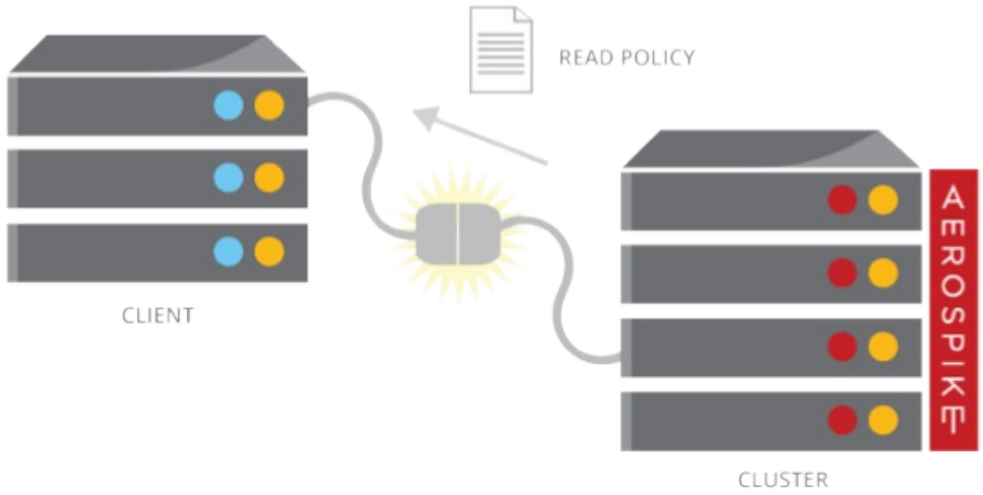
Example code:batch reads

```
// C# batch
Key[] keys = new Key[tweetCount];
for (int i = 0; i < keys.Length; i++)
{
       keys[i] = new Key("test", "tweets",
              (username + ":" + (i + 1)));
}
Record[] records = client.Get(null, keys);


// Java batch
Key[] keys = new Key[27];
for (int i = 0; i < keys.length; i++) {
 keys[i] = new Key("test", "tweets",
       (username + ":" + (i + 1)));
}
Record[] records = client.get(null, keys);
```

```
// PHP batch
$keys = array($key1, $key2, $key3, $key4);
$res = $db->getMany($keys, $records);
```

Read Policies

Policies are container objects/structures for transaction policy attributes used in all database operation calls.



A Read Policy object can be optionally on each Read operation, to modify default values.

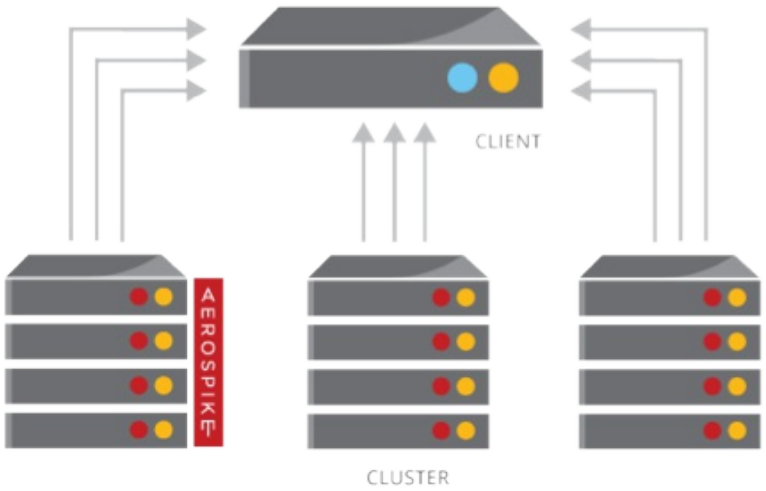
Attribute	Type	Description
Timeout	Integer	Total transaction timeo both client and server.
Max retries	Integer	Maximum number of r the current transaction
Sleep between retries	Integer	Milliseconds to sleep b transaction fails and th exceeded.
Consistency	CONSISTENCY_ONE	How replicas should be

level	CONSISTENCY_ALL	operation to provide th guarantee
Replica	MASTER MASTER_PROLES RANDOM	Send read commands t the key's partition repl

NOTE: Timeout trumps retries. If your timeout is 10ms, number of retries is 3 and sleep between retries is 5ms, you will timeout before the 2nd retry.

Scan

The Scan operation provides the ability to read all Records in a specified Namespace and, optionally, a Set.



By default, the Scan will query nodes for Records in parallel using threads. All bins, or the Bins you specify, will be returned with each Record.

If an exception is thrown, parallel Scan threads to other nodes will also be terminated and the exception will be propagated back through the initiating Scan call.

Scan returns Records to the user defined callback, multiple

threads will call your can callback method or function concurrently. It is drinking from a fire hose, therefore, your scan callback implementation should be thread safe.

Example code:scan

```
// Java Scan
client.scanAll(null, "test", "tweets", new ScanCallback() {

       @Override
 public void scanCallback(Key key, Record record)
                                    throws AerospikeException {
   System.out.println(record.getValue("tweet"));
 }
}, "tweet");

// C# Scan
client.ScanAll(null, "test", "tweets",
                            scanTweetsCallback, "tweet");

public void scanTweetsCallback(Key key, Record record)
{
       Console.WriteLine(record.GetValue("tweet"));
}

// PHP Scan
$status = $db->scan("test", "users", function ($record) {
       echo "\nName: " . $record['bins']['name']
       . "\nEmail:" . $record['bins']['email'];
});
```

A Scan Policy is used to control the optional parameters in a scan operation

Attribute	Description
Concurrent nodes	Issue scan requests in parallel, th true
Fail on cluster change	Terminate scan if cluster is addi a node, the default is false
Include bin data	Indicates if bin data is retrieved. record digests are retrieved. The
Maximum concurrent nodes	The maximum number of concu

	to server nodes, the default is al cluster. This is only valid when nodes is true.

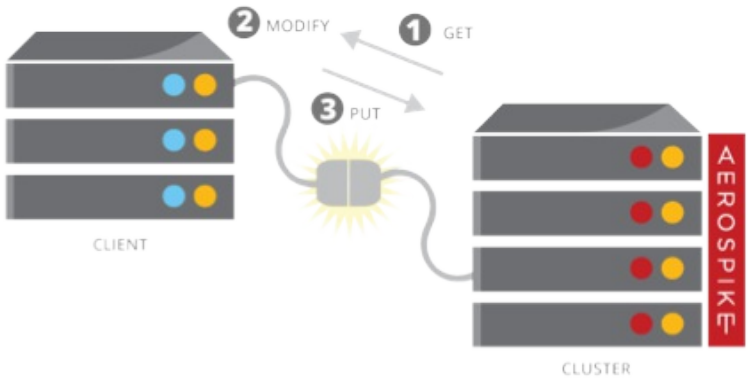
Example code:scan policy

```
// Java Scan policy
ScanPolicy policy = new ScanPolicy();
policy.concurrentNodes = true;
policy.includeBinData = true;
client.scanAll(policy, . . .);
```

Key-value techniques

Read-Modify-Write

A common usage pattern is to do Read-Modify-Write of a record, or Check-and-Set.



This involves:

Reading the Record.

Modifying the Record in the application.

Set the WritePolicy to be EXPECT_GEN_EQUAL.

Set the Generation to the value the record read.

Write the modified data with the generation previous Read.

If the generation number of the record in the cluster the same, the Write will proceed. If the generation number is different, the Record has been written by another request and the Write will fail. The data in your application is now stale and you will need to re-read the Record.

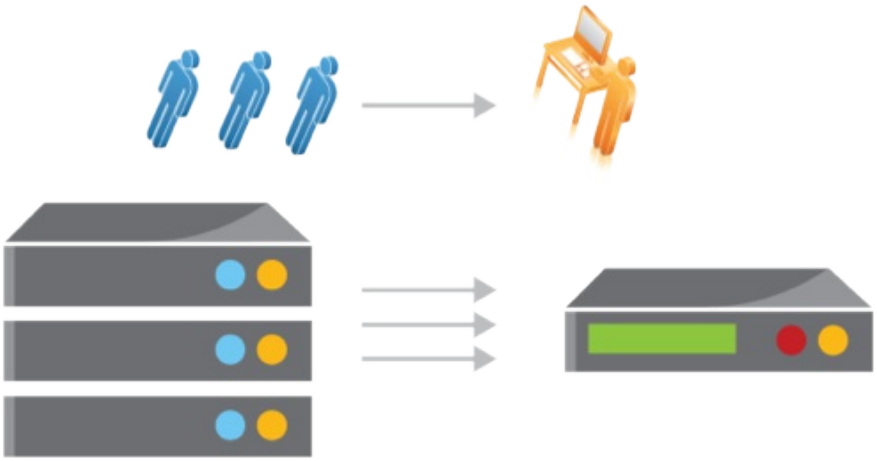
Example code:read-modify-write

```
// Java read-modify-write
```

```
Key key = new Key("test", "users", userName);
Record record = this.client.get(null, key);

WritePolicy wPolicy = new WritePolicy();
wPolicy.generationPolicy = GenerationPolicy.EXPECT_GEN_EQUAL;
wPolicy.generation = record.generation;

Bin bin1 = new Bin("password", password);
client.put(wPolicy, key, bin1);
```

Hot Keys

A hot key is a Record that is frequently accessed by a large number of (almost) concurrent requests. This is often occurs with a counter that counts every action, request, click, etc.



How hot is hot? – Depending on the hardware you can have a single key with over 100K tps, and depending on the operation.

If the number of requests exceeds the queue size for a key, the client well receive the result code 14 - KEY_BUSY indicating that the key is busy.

The size of the queue is controlled by the configuration setting transaction-pending-limit, located in the Service stanza of the Aerospike configuration file. The default value is 20. This means that 20 concurrent requests will be accepted before the client would be returned KEY_BUSY

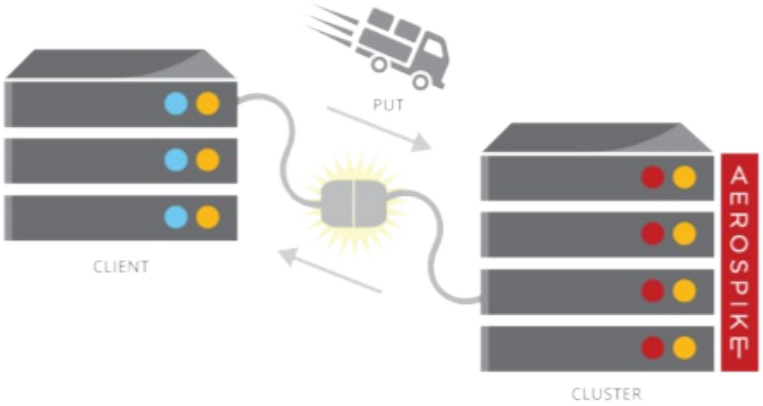
You can mitigate Hot Keys by:

Put high Hot Keys in a RAM namespace.

Separate high frequency hot key counters in to separate records e.g. - Instead of a single counter for every click on a URL, have N counters segmented by a demographic (location, age group, favorite color, etc.) and stored in separate Keys. Aggregate the value in your application by using a Batch read.

Operate

The Client API provides a way to perform multiple operations on a record within a single transaction. This feature also allows the bins of a Record to be modified and read back to the client within a single transaction, i.e. it allows an application to perform an atomic modification that returns the result.



It is like an Update and a Select in one atomic operation. You package a set of operations into the request, the write operations are done first, then the read operations. The record is returned to the client at the end of the operation.

Example code:operate

```
// C# Operate
Record record = client.Operate(policy, userKey,
       Operation.Add(new Bin("tweetcount", 1)),
       Operation.Get("lasttweeted"));
```

```
// PHP Operate
$operations = array(
    array("op" => Aerospike::OPERATOR_WRITE,
              "bin" => "Occupation",
              "val" => "Doctor of Freudian Circuit Analysis"),
    array("op" => Aerospike::OPERATOR_INCR,
              "bin" => "patients_cured", "val" => 4),
    array("op" => Aerospike::OPERATOR_READ,
              "bin" => "Occupation"),
    array("op" => Aerospike::OPERATOR_READ,
              "bin" => "patients_cured"));
$res = $db->operate($key, $operations, $returned);

// Java operate
WritePolicy wPolicy = new WritePolicy();
Key key = new Key("test", "users", username);
Record record = client.operate(wPolicy, key,
       Operation.add(new Bin("tweetcount", 1)),
       Operation.get("tweetcount"));
```

The following are the operations which can be performed using the client API

Operation	Description	Conditions
Write	Write a value to a Bin	
Read	Read the value of a Bin	
Read header	Read only the meta-data (generation and time-to-live) of the record.	
Add	Add an integer to the value of a bin.	Existing valu
Append	Append string to the existing value of a bin.	Existing valu
Prepend	Prepend string to the existing value of a bin.	Existing valu
Touch	Rewrite the same record	Generation a

		updated.

Complex types

Aerospike supports the database native types of **List** and **Map**. Lists and Maps can contain other nested Lists and Maps along with any supported database type, so you can store complex objects in a single **Bin** and read or write the whole object from the client API.

Because the List or Map is stored in a database native format, you can, for example, write it in C# and read it in Python. Also, the database native type can be manipulated by User Defined Functions (UDFs).

A JSON object is either a Map(Dictionary) containing Lists and Maps, or a List containing Lists and Maps. This means that a JSON object can either be stored as a string or in a database native way as either a List or a Map in.

Example code:JSON

```
// Java JSON example
Bin bin7 = new Bin("interests", new JSONObject(...));
```

Errors and Result Codes

Error Handling

If an error occurs as a result of the database operation, an AerospikeException is thrown. An AerospikeException has a **ResultCode** field, which describes the exact cause of the error given by the Aerospike Server; and a human readable message describing the error. There can also be conditions where an error occur on the client side, and a sub-classed AerospikeException is thrown. Examples of such conditions include:

AerospikeException.Connection - Exception thrown when client can't connect to the server.

AerospikeException.Timeout - Exception thrown when database request expires before completing.

Example code:Error handling

```
// Java error handling
try {

       . . .

} catch (AerospikeException e) {
       System.out.println("EXCEPTION - Message: "
              + e.getMessage());
       System.out.println("EXCEPTION – Result code: "
              + e.getResultCode());
}

// PHP error handling
if ($status == Aerospike::OK) {
    echo success();
} else {
    echo standard_fail($db);
}
```

Result codes

Result codes are return values indicating the success of an operation, or the type of failure. This table shows common result codes for key value operations.

Code	Symbolic name	Description
0	OK	Operation was successfu
1	SERVER_ERROR	Unknown error in the se Aerospike support.
2	KEY_NOT_FOUND	On retrieving, touching c that doesn't exist.
3	GENERATION_ERROR	On modifying a record w generation.
4	PARAMETER_ERROR	Bad parameter(s) were p operation call.
5	KEY_EXISTS_ERROR	On create-only (write un record that already exists
6	BIN_EXISTS_ERROR	On create-only (write un bin that already exists.
7	CLUSTER_KEY_MISMATCH	Expected cluster ID was is changing during a Scan
8	SERVER_MEM_ERROR	Server has run out of me
9	TIMEOUT	Client or server has time
10	NO_XDS	XDR product is not avail
11	SERVER_NOT_AVAILABLE	Server is not accepting re

12	BIN_TYPE_ERROR	Operation is not support bin type (single-bin or m
13	RECORD_TOO_BIG	Record size exceeds limit
14	KEY_BUSY	Too many concurrent ope record.
15	SCAN_ABORT	Scan aborted by server.
16	UNSUPPORTED_FEATURE	Unsupported Server Feat UDF).
17	BIN_NOT_FOUND	Specified bin name does
18	DEVICE_OVERLOAD	The server node's storage up with the write load.
19	KEY_MISMATCH	Key type mismatch. – Dig been seen
100	UDF_BAD_REPONSE	A user defined function r code.

User Defined Functions

In many databases, a user-defined function provides a mechanism for extending the functionality of the database server by adding a function that can be evaluated from application code. For example: within SQL statements.

User Defined Functions allow the compute to move close to data. The function is run on the same server where the data resides.

User Defined Functions are common in many databases: MySQL, SQL server, Oracle, DB2, Redis, Postgress, Aerospike and others

User Defined Functions in Aerospike

A User-Defined Function (**UDF**) is a piece of code, written by a developer in the Lua programming language. It runs inside the cluster and is available on every server node.

There are two types of UDFs in Aerospike: Record UDFs and Stream UDFs. A **Record UDF** operates on a single record. A **Stream UDF** (discussed in a later module) operates on a stream of records, and it can comprise multiple stream operators to perform very complex queries.

The complexity of UDFs can range from a single function that is only a few lines long, to a multi-thousand line module that contains many internal functions and multiple external functions.

Development cycle

When contemplating the construction of a new UDF, it is

important to consider the application data model and the interaction between record Bins. The general pattern (or life-cycle) for UDF development is:

Design the application data model

Design the UDFs to perform desired functions on the data model

Create/Test the UDFs

Register UDFs in the Aerospike Servers

Iterate function test/development cycle

System Test

System Deployment

User Defined Function design and development is usually an iterative activity, where the first version is simple and then, potentially, evolves over time to something complex.

Restrictions

UDFs can use the full Lua programming language with a few restrictions.

Globals - Global variables are not allowed. Global functions can only be called by Aerospike Server, but not by other Lua functions. To call a custom Lua function from another Lua function, you will need to declare it as local function.

Coroutines - As a UDF acts on a single record, or a single element from a stream, it is already highly paralllel. Lua cooperative multi-tasking is ineffective in increasing parallelism or concurrency.

Debug module – Not enabled due to the performance impact the module will have on the UDF.

os.exit() – Not enabled because it does not make sense for a Lua script to cause the database process to exit.

Lua

"Lua" means "Moon" in Portuguese.

"Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection" - www.lua.org

This example is a brief overview of the the typical Lua language constructs that you will encounter. See http://luatut.com/crash_course.html

Example code:factor example

```
01:function get_all_factors(number)                    -- function definition
02:  local factors = {}                                -- local variable definition
03:  for possible_factor=1, math.sqrt(number), 1 do -- for loop
04:    local remainder = number%possible_factor     -- assignment (%
```

```
05:    if remainder == 0 then                          -- if equal test
06:      local factor, factor_pair
07:          = possible_factor, number/possible_factor -- multiple assignment
08:      table.insert(factors, factor)                 -- table insert
09:      if factor ~= factor_pair then                 -- if not equal
10:        table.insert(factors, factor_pair)
11:      end
12:    end
13:  end
14:  table.sort(factors)
15:  return factors                                    -- return value
16:end
17:-- The Meaning of the Universe is 42.
18:-- Let's find all of the factors driving the Universe.
19:the_universe = 42                                   -- global variable and assignment
20:factors_of_the_universe = get_all_factors(the_universe) -- function call
21:-- Print out each factor
22:print("Count",  "The Factors of Life, the Universe, and Everything")
23:table.foreach(factors_of_the_universe, print)
```

Language features illustrated:

Line 1 – Global function declaration

Line 2 – local variable, initialized to a Table

Line 3 – For loop

Line 5 – If test

Line 8 – table operation: insert

Line 14 – table operation: sort

Line 15 – function return

Line 17 – single line comment

Line 19 – global variable declaration and initialization

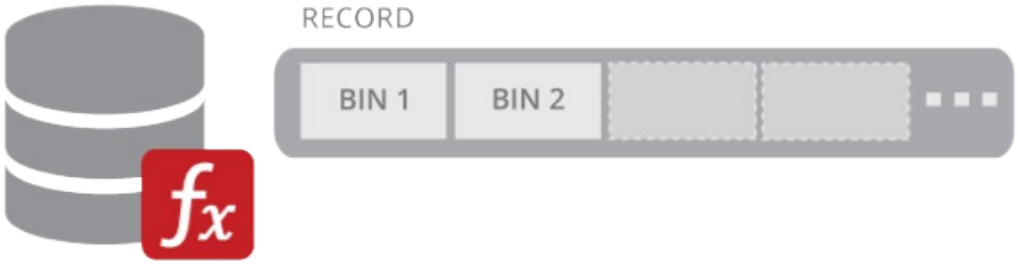
Line 20 – function call

If you are unsure why the universe is equal to 42 see http://hitchhikers.wikia.com/wiki/42

*

Developing Record UDFs

With a Record UDF you can move some of your logic into the server, usually to save Network. Don't try to save client CPU, because then you'll just be spending on expensive server CPU.



An example of a UDF would be retrieving/adding a subpart of a string or byte array. In advertising, an example would be to get the most recent list of search terms. The UDF "get most recent search terms()"; could contain more complicated logic like: return last 5 searches, up to 12 hours old, and clean up old list elements while you're there.

Record UDFs are applied to a single record only, UDFs that access “other records” are not supported.

A UDF can augment both read and write behavior by:

- Create/delete Bins in the record.
- Read any/all Bins of the record.
- Modify any/all Bins of the record.
- Delete/Create the specified record.
- Access parameters passed in from the client.

The first argument will always be a record.

A good practice is to use a variable name other than 'record' to reference the parameter passed into the UDF to represent the Aerospike Record. "record" is a Record module used to call built-in record functions e.g. `record.ttl()` .

A Record UDF can have more than one argument. Each additional argument must be of one of the types supported by the database: integer, string, list and map. If a UDF has N parameters, and only (N-k) parms are passed in, then the last k parms will have a value of `nil`.

A Record UDF can return a single value of any of the types supported by the database: integer, string, bytes, list and map.

Aerospike Modules (Libraries)

Aerospike supplies a **Module** or library for use with Record UDFs.

The Aerospike module provides the following functions to access the record passed in to the UDF (first parameter):

Function	Description
aerospike:create()	Create a new record in the database. If cr then return 0 (zero) otherwise it is an er
aerospike:update()	Update an existing record in the databas successful, then return 0 (zero) otherwis
aerospike:exists()	Checks for the existence of a record in th record exists, then true is returned, othe
aerospike:remove()	Remove an existing record from the data

	success, then return 0 (zero), otherwise

Note: Lua also offers a special syntax for object-oriented calls, the colon operator. An expression like `o:foo(x)` is just another way to write `o.foo(o, x)`, that is, to call `o.foo` adding `o` as a first extra argument.

Logging functions are also provided, these functions send log message to the Aerospike Server's logs. The logging functions all accept a format string as the first parameter with a variable argument list, much like printf in C.

While logging, you will need to ensure the value used as arguments for the format string are valid for the format string parameters. When in doubt, you can use the Lua `tostring()` function to convert a value to a string. For example:
`info("Hello %s", tostring(name))`

There are 4 levels of logging, and the function names reflect this

Function	Description
info(format, ...)	Log a message as a INFO in the Aerospike log.
warn(format, ...)	Log a message as a WARNING in the Aer server log.
debug(format, ...)	Log a message as a DEBUG in the Aerosp server log.
trace(format, ...)	Log a message as a TRACE in the Aerospi log.

Logging isn't free, while the Aerospike logging subsystem is very

efficient, each log message will still consume resource and ultimately result in an I/O. In production, UDFs should log debug messages in exceptional circumstances only.

To list the log files in-use:

```
asinfo -h 127.0.0.1 -p 3000 -v logs
```

To change logging lever dynamically use:

```
asinfo -h 127.0.0.1 -p 3000 -v "log-set:id=0;udf=info"
```

where

`id` is the logging file, `0` is the first file, `1` the second, etc

`udf` is the logging sub system

`info` is the level.

You can configure Aerospike to log messages to a file different from the default Aerospike server log, this example shows how to log messages for the `udf` subsystem to the file `udf.log`, at the level `debug`:

```
file /var/log/aerospike/udf.log {
  context any critical
  context udf debug
 }
```

Writing a module

A Lua Module is a collection of variables and functions contained in a single file, which can be imported (used) by other Lua Modules.

There are various options for how to define a Lua Module, however we found the best method is to use what is called the "local module". A local module is a file that defines all variables and functions as locals, and exports the functions that can be used by other modules by returning a Lua Table.

Define a local table that will collect the exported members, this is a function table. In the following example, we define an local table called "exports", which will be populated with the members to be exported. For example: a module (file) named `mymod.lua`:

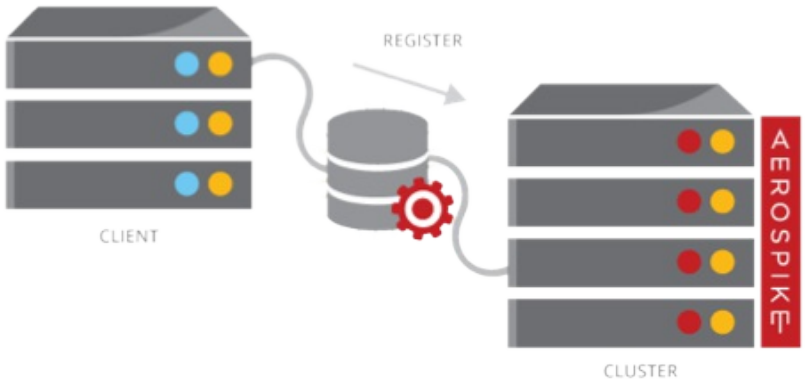
```
local exports = {}
function exports.one()
  return 1
end
function exports.two()
  return 2
end
return exports
```

Another module can then require `mymod.lua` and use the functions of the module.

```
local MM = require('mymod')
function three()
  return MM.one() + MM.two()
end
```

Managing UDFs

A module is a file containing one or more user-defined functions. A module and all it's non-Aerospike dependencies must be registered (uploaded) with the cluster, before the UDF can be invoked. A deployment may have one or many modules.



When a UDF module is registered, it is maintained in the cluster. As nodes are added to the cluster, the UDF modules are synchronized.

Registration/De-registration Options

UDF registration is an administrator operation, and should be controlled using normal change control procedures. A UDF module can be registered into a cluster using one of the provided tools:

aql* – A command-line utility with SQL-like commands

ascli* – Aerospike command line interface

Client API - provides a number of functions that allow you to programmatically manage user-defined functions in a cluster, this is useful of you want to develop your own tool.

*These tools are are included the “tools” package as part of the tools distribution for OSX and Linux.

Lifecycle of a UDF

When a package is registered, it is immediately compiled into byte-code and made available to subsequent invocations from clients. If the registration is replacing an existing package, currently executing UDFs will use the existing package, all new invocations after the update will only use the most recently updated module. When a modules is removed, it's UDFs are no longer be available.

If a client is in the middle of invoking a UDF when it's module is removed or updated, it will be able to complete the operation without interruption. Once the function completes, any subsequent invocation will either use the updated function, or fail if the module is removed.

A module’s non-Aerospike dependencies must be manually copied to each node in the cluster, and place in this directory: `/opt/aerospike/usr/udf/lua`

Managing modules with aql

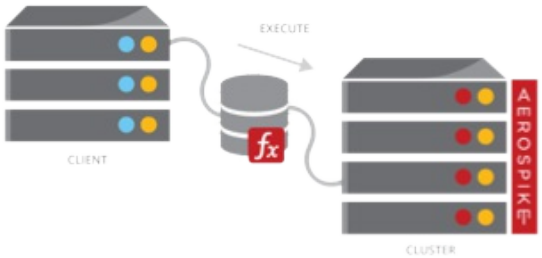
The easiest way to manage UDF modules is with aql with these simple commands:

Operation	aql command

Register a Module	`register module 'src/test/resources/`
Show registered modules	`show modules`
See the details of a Module (Base64 encoded source code)	`desc module example1`
Remove a module	`remove module exampl`

Note: the file extension .lua is part of the module name.

Executing a Record UDF



To execute a UDF, you specify the:

Key object containing the Namespace, Set and value of the primary key

Module name

Function name within the module

Arguments (parameters) that will be passed to the function

```
// Java UDF execution
String result = (String) client.execute(null, userKey, "updateUserPwd",
                    "updatePassword", Value.get(password));

// PHP UDF execution
$args = array($new_password);
$status = $this->client->apply($key, 'updateUserPwd', 'updatePassword',
                               $args);
if ($status !== Aerospike::OK) {
       . . .

// Go UDF execution
updatedPassword, err := client.Execute(nil, userKey, "updateUserPwd",
                                       "updatePassword", NewValue(password))
```

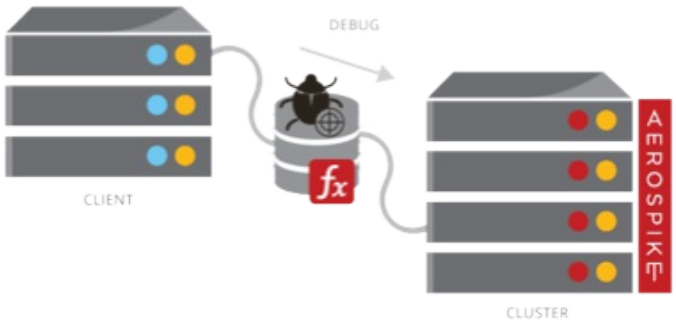
In your code you will use the Client API to execute UDFs, but for developer and unit testing you can optionally use aql to execute a UDF.

```
execute updateUserPwd.updatePassword('cats123') ON test.users WHERE PK = 'user1'
```

Note: PK is a reserved word for the Primary Key

Debugging User Defined Functions

There is no facility to connect a debugger to the Lua contexts running inside the Aerospike daemon. Debugging is done vial log messages.



Tips to make debugging easier

Develop and debug on a single node cluster, using a VM on your laptop, even if you're using OSX or Windows. In a multi-node cluster, the server that will hold your record is randomly allocated, so where a request is executed can't be predicted. Using a single-node cluster ensures that you will be able to tail and grep the correct log file – because there is only one log.

Test the UDF on a single node cluster. A VM (virtual box/vagrant) running on your laptop with the log files shared between the Host operating system and the Guest operating system. Modify the `aerospike.conf` file to log UDF entries in a separate log file:

`/etc/aerospike/aerospike.conf`

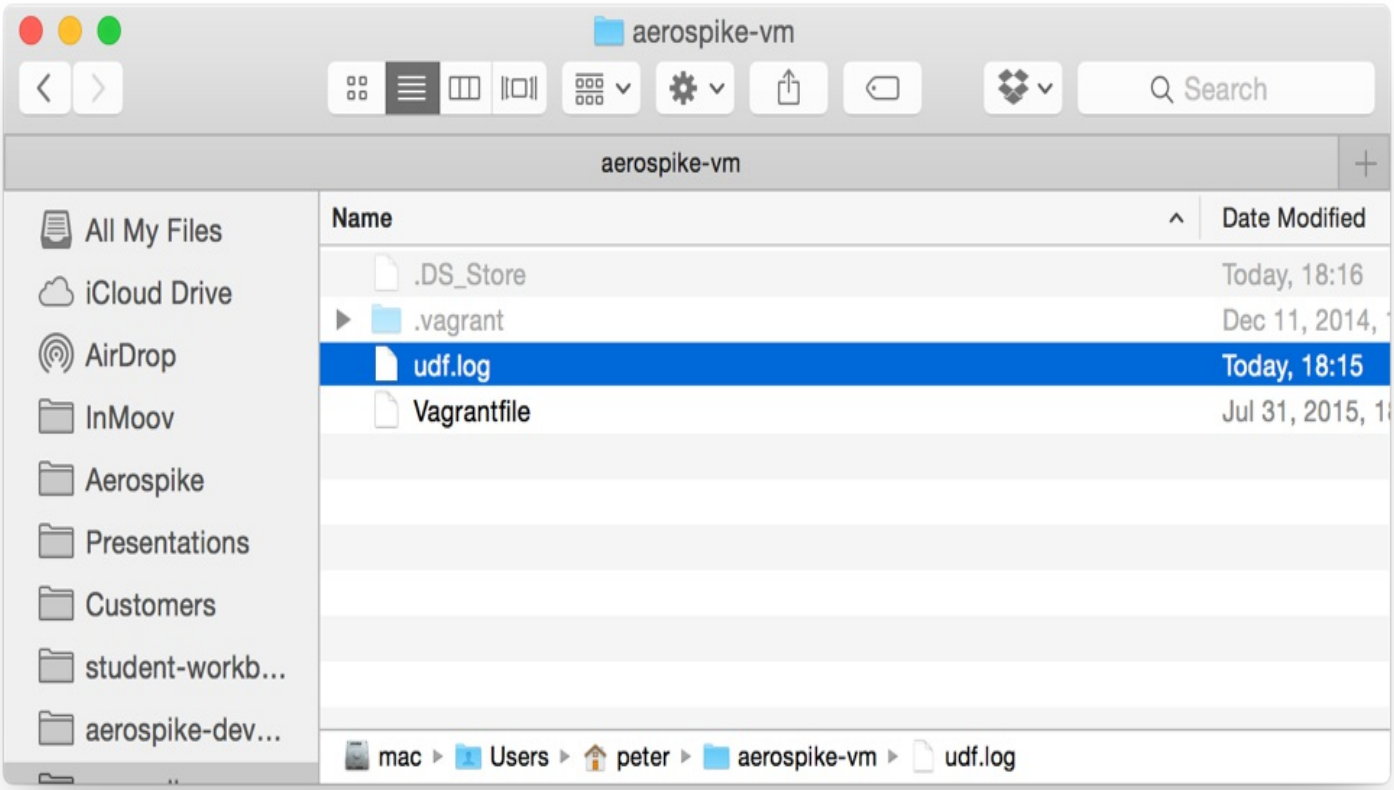
Locate the “logging” stanza and modify it to look like this:

```
logging {
        # Log file must be an absolute path.
        file /var/log/aerospike/aerospike.log {
                context any info
        }
        file /vagrant/udf.log {
                context any critical
                context udf debug
        }
}
```

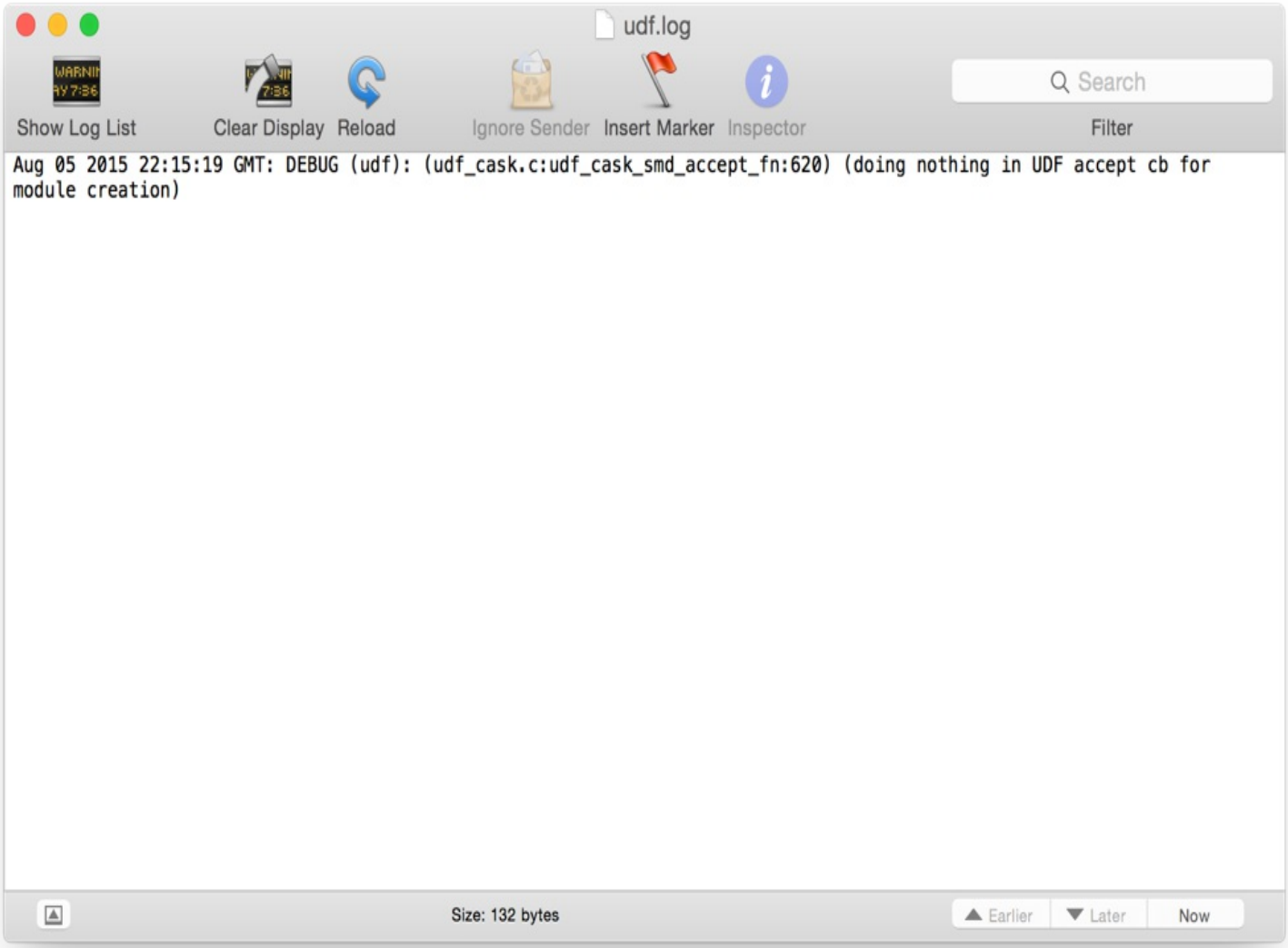
Restart Aerospike

```
sudo service aerospike restart
```

This writes the UDF log to the directory shared between OS X and Centos - this is the directory shared between the Host operating system and Guest operating system is where the vagrant box is defined:



Note the udf.log file. Double click on this and view it in the OS X log file viewer



When you execute the UDF, via aql, or from your application, the UDF log messages will appear in the log viewer.

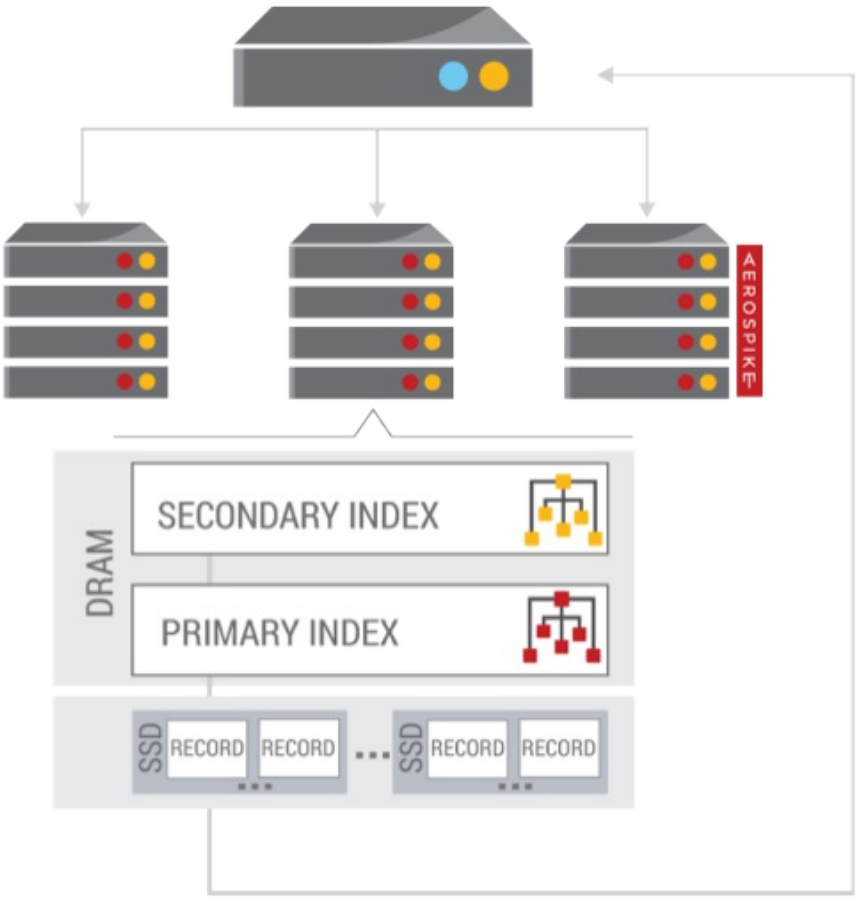
Use aql to register you UDF.

```
register module 'src/test/resources/example1.lua'
```

You will probably register and test you UDF many times before you can use it from your application. aql provides you a way to load data, register modules, execute UDFs and cleanup data, in a repeatable way.

Queries

Aerospike query provides value-based look up through the use of secondary indexes. These are best suited for high cardinality queries. Query result returned as set of records, like a select statement in SQL.



The query is sent to all nodes in parallel in a scatter-gather pattern. The query is scattered. Worker threads in the client gather the results from all the nodes in the cluster.

$$Selectivity = Cardinality / Rows * 100$$

Cardinality is the number of unique values. For example the population of Europe is approx. 744 million but the cardinality based on male and female is 2.

Selectivity of an index is the cardinality divided by the number of rows and expressed as a percentage.

You specify the Namespace, Set, Bins you want to retrieve and a single predicate filter (a where clause) in a Statement and then use that Statement in the Query() method. The results are returned as a handle to a RecordSet.

The RecordSet is a collection that can be iterated over to retrieve each record. The RecordSet contains a blocking queue, the iterator reads from one end of the queue and the query jobs, running in each node in the cluster, fill the other end. When the queue is full, the protocol to the server pauses the query jobs. As the queue empties, the query jobs refill the queue until the query is exhausted.

Managing Secondary Indicies

Before executing a Query, a secondary index needs to be created. An index consumes RAM for every index entry, and background index creation can take a substantial amount of resources. Index creation should be scheduled carefully on an operational system.

Creating a Secondary Index

An Index is created using:

Namespace (database)

Set (table) including the null Set

Bin name AND type (column)

Remember: a named Bin can have a different type in different records – No Schema. Creating indexes takes time and resources.

The easiest way to create and manage indices in an Aerospike cluster is using the aql tool. For example:

```
CREATE INDEX tweetcount_index ON test.users (tweetcount) NUMERIC
```

aql is officially distributed as part of the tools package. It is a Linux and OSX command line tool that implements Aerospike Query Language (AQL) which is a SQL like language. A Java based aql implementation and an Eclipse plugin is available at

Aerospike Labs http://www.aerospike.com/community/labs/. You can also create indexes with the client API.

Listing secondary Indexes

The best way to see the details of an index is by using the aql command:

```
SHOW INDEXES <index name>
```

The namespace is optional and if omitted the indexes for all name spaces are shown. The results:

```
aql> show indexes
+--------+---------------+-----------+----------+-------+--------------------+------------+--------------+
| ns     | bins          | set       | num_bins | state | indexname          | sync_state | type         |
+--------+---------------+-----------+----------+-------+--------------------+------------+--------------+
| "test  | "FL_DATE_BIN  | "flights  |          | "RW   | "flight_date       | "synced    | "INT SIGNED  |
| "test  | "tweetcount   | "users    |          | "RW   | "tweetcount_index  | "synced    | "INT SIGNED  |
| "test  | "username     | "tweets   |          | "RW   | "username_index    | "synced    | "TEXT        |
| "test  | "ts           | "tweets   |          | "RW   | "ts_index          | "synced    | "INT SIGNED  |
+--------+---------------+-----------+----------+-------+--------------------+------------+--------------+
4 rows in set (0.002 secs)
OK
```

Each secondary index has a value called sync_state, which specifies whether the secondary index is in sync with the primary index.

Sync_state	Description
synced	The secondary index is in sync with the primar
need_sync	The secondary index may not be in sync with th
State	**Description**
WO	Secondary index in Write Only Mode. Normal A update secondary index but queries cannot be p
RW	Secondary index in Read Write Mode. Normal a
	A secondary index should be in RW state on all can use it.

A secondary index should be in RW state on all the nodes before query can use it. When an index has sync_state=need_sync and state=RW, then it may need to be repaired. aql does not provide the ability to repair indexes. For the time being, you will need to use Aerospike Info (asinfo) to execute a repair, on each node in the cluster.

```
$ asinfo -v "sindex-repair:ns=test;indexname=ind_name;set=set_name;"
```

Deleting a secondary index.

The index is deleted instantaneously, memory in each node associated with the index is released and the index definition is removed.

```
aql> drop index test.demo flight_date
OK, 1 index removed.
```

Note: Deleting a secondary index is not atomic and are instantaneous. If and index is deleted, and Queries are using it, they will return a smaller record set than expected.

Executing a Query

To execute a query you first need to prepare a Statement by supplying a list of Bins to be returned, and supplying a Filter (predicate) that constrains the query.

Consider this aql example:

```
select username from test.users where tweetcount between 5 and 10
```

To code the equivalent, you should follow these steps:

Prepare a Statement containing the “tweet” Bin, and a range filter on the “tweetcount” Bin who’s value is between 5 and 10 (inclusive)

Execute the statement with a Query operation – the query can have an optional Query Policy to modify the default behavior. The Query operation returns a RecordSet collection.

Process the results in the RecordSet by integrating through the collection.

Preparing a statement

A **Statement** is a data structure that provides parameters to the query. In object oriented languages, a statement is an object instance.

The Statement works with a specific Namespace and Set (including the null set) and cannot be used to query across Sets (joins). A list of Bin names is optional and will determine the Bins

returned, if no Bins names are specified, all the Bins in the record are returned.

Only one filter is valid in a statement and the filter is used to qualify the query. The filter can be:

Equality filter – equivalent to the SQL “where tweetcount = 5”. Equality filters can be used with Integer and String Bin types.

Range filter – equivalent to the SQL “where tweetcount between 5 and 10”. Range filters can be used only with Integer Bin.

After a Statement is prepared, it can be used as many times as you want.

Example code:preparing a statement

```
// C# statement
string[] bins = { "username" };
Statement stmt = new Statement();
stmt.SetNamespace("test");
stmt.SetSetName("users");
stmt.SetBinNames(bins);
stmt.SetFilters(Filter.Range("tweetcount", 5, 10));

// Go statement
stmt := NewStatement("test", "tweets", "tweet")
stmt.Addfilter(NewEqualFilter("username", username))
```

The above code is the equivalent of this aql:

```
select username from test.users where tweetcount between 5 and 10
```

Execute a Query

By default, the query execution will send (scatter) the query to all nodes in the cluster and will be executed in parallel. Each node will execute the query and return records to the client. The records are made available to the application in a RecordSet collection. The query executor uses a separate worker thread for

each node and puts the returned records on a queue internal to the RecordSet. The application thread can concurrently pop records off the queue through the record iterator.

Example code:query

```
// C# Query execution
RecordSet rs = client.Query(null, stmt);

// Java query policy
QueryPolicy qPolicy = new QueryPolicy();
qPolicy.maxConcurrentNodes = 5;
RecordSet recordSet = client.query(qPolicy, stmt);

// Go Query execution
recordset, err := client.Query(nil, stmt)
panicOnError(err)

// PHP Query execution
$where = $db->predicateBetween("age", 0, 99);
$status = $db->query("test", "characters", $where, function ($record) use (&$total,
&$not_centenarian) {
    $total += (int) $record['bins']['age'];
    $not_centenarian++;
}, array("email", "age"));
```

Query Policy

Query policy modifies the default query request.

Max concurrent nodes - Maximum number of concurrent requests to server nodes at any point in time. If there are 16 nodes in the cluster and max concurrent nodes is 8, then queries will be made to 8 nodes in parallel. When a query completes, a new query will be issued until all 16 nodes have been queried. Default (0) is to issue requests to all server nodes in parallel.

Record queue size - Number of records to place in queue before blocking. Records received from multiple server nodes will be placed in a queue. A separate thread consumes these records in parallel. If the queue is full, the producer threads will block until

records are consumed.

Recommendation: Use the defaults by omitting the QueryPolicy

Processing results

A RecordSet is a collection you can iterate over to retrieve the queried records. Each element in the collection will contain a Key and a Record. Elements in the RecordSet are not ordered (no "orderby" constraint), and are not limited. So if your query selects 1 million records, 1 million records will be in your RecordSet. This is an important memory consideration. Internally, the RecordSet is implemented as a queue. Elements are placed in the queue as they are returned from a cluster node and they are removed from the queue when next() is called. The internal queue is thread safe.

Each element in the collection contains a Record and a Key.

The **`next()`** method returns a Boolean and fetches the element from the queue and makes it available to be used.

The **`getKey()`** method returns a Key object, because the Key object is returned from the server. This will contain a Digest and not the value of the primary key.

The **`getRecord()`** method returns a Record object. This object will contain the generation count along with the requested Bins.

A common way to process a RecordSet is to use a "while" loop with the Boolean condition being the return from the `next()` method. When you complete the processing of the RecordSet, remember to close it to flag it as as being consumed.

Example code:processing a **RecordSet**

```
// C# record set
RecordSet rs = client.Query(null, stmt);
while (rs.Next())
{
       Record r = rs.Record;
       Console.WriteLine(r.GetValue("tweet"));
}
rs.Close();

// Java record set
RecordSet rs = client.query(null, stmt);
while (rs.next()) {
 Record r = rs.getRecord();
 console.printf(r.getValue("tweet").toString() + "\n");
}
rs.close();
```

Remember: Close the **RecordSet**

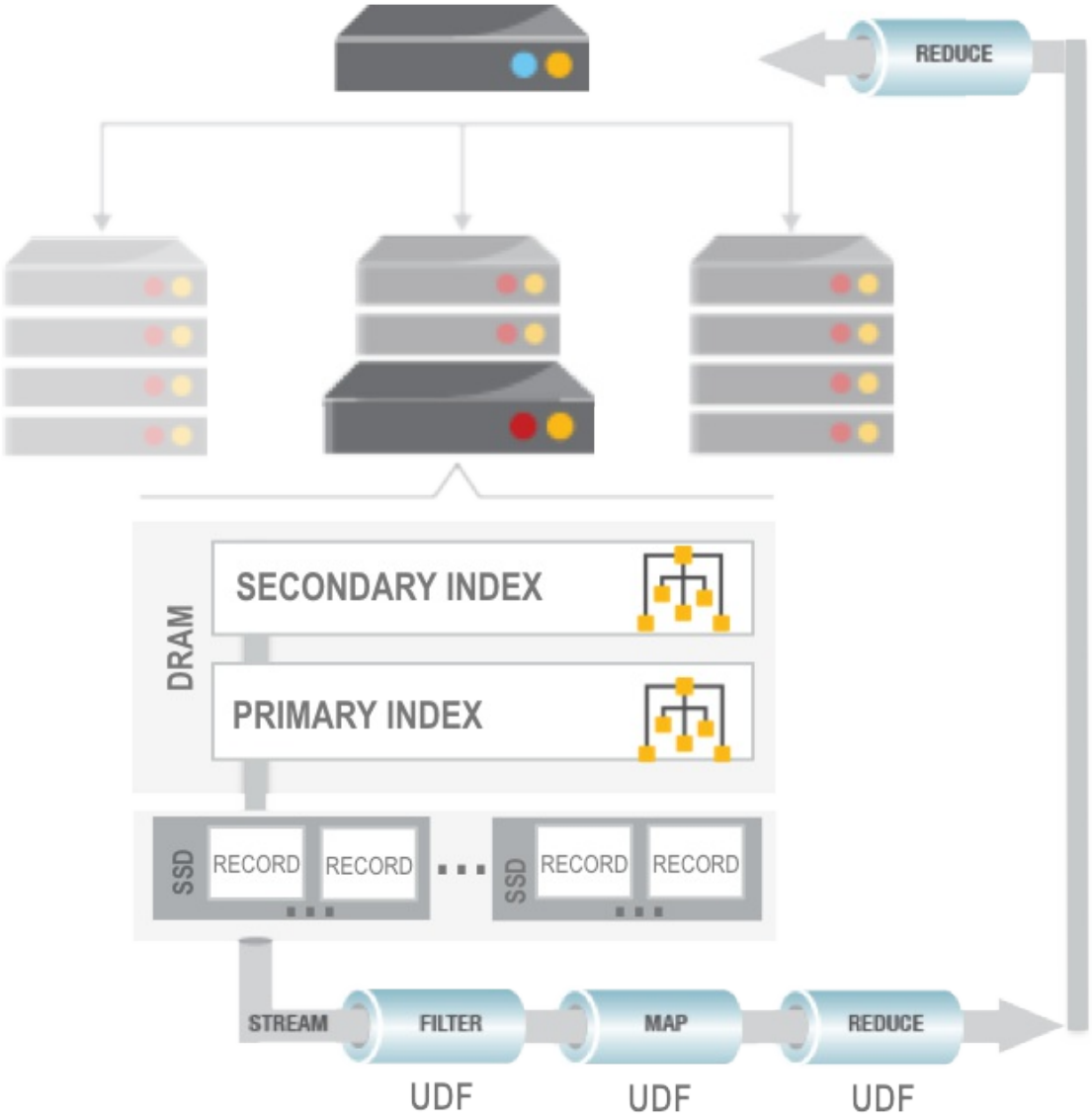
Errors and Result Codes

In addition to the result codes from standard operations, Queries have specific result code relating to secondary indexes and the query operations.

Code	Symbolic name	Description
200	INDEX_FOUND	Secondary index already e
201	INDEX_NOTFOUND	Requested secondary inde
202	INDEX_OOM	Secondary index memory
203	INDEX_NOTREADABLE	Secondary index not avail
204	INDEX_GENERIC	Generic secondary index ***support.***
205	INDEX_NAME_MAXLEN	Index name maximum len
206	INDEX_MAXCOUNT	Maximum number of indi
210	QUERY_ABORTED	Secondary index query ab
211	QUERY_QUEUEFULL	Secondary index queue fu
212	QUERY_TIMEOUT	Secondary index query tir
213	QUERY_GENERIC	Generic query error. ***Call***

Aggregations

Aerospike Aggregations are a programmatic framework that is similar to a MapReduce system, in that an initial query emits a stream of results in a highly parallel fashion. That stream flows through a pipeline which *may* specify additional Filter, steps, Map steps, Aggregation steps, and Reduction steps. The simple use case is counts and aggregate sums inside the database.



One of the main differences from other systems is that the

aggregation can optionally specify an index, essentially making an indexed MapReduce system. The performance can be very high with the use of an index.

The aggregation system is implemented using User Defined Functions (UDFs) written in Lua. Functionality written in C can be called from Lua.

A client sends the aggregation request to all the servers in the cluster, who processes results independently, then each server individually send results back to the requesting client in parallel. The client then runs a final reduce phase, also in Lua, to collate the results.

Example Use Case

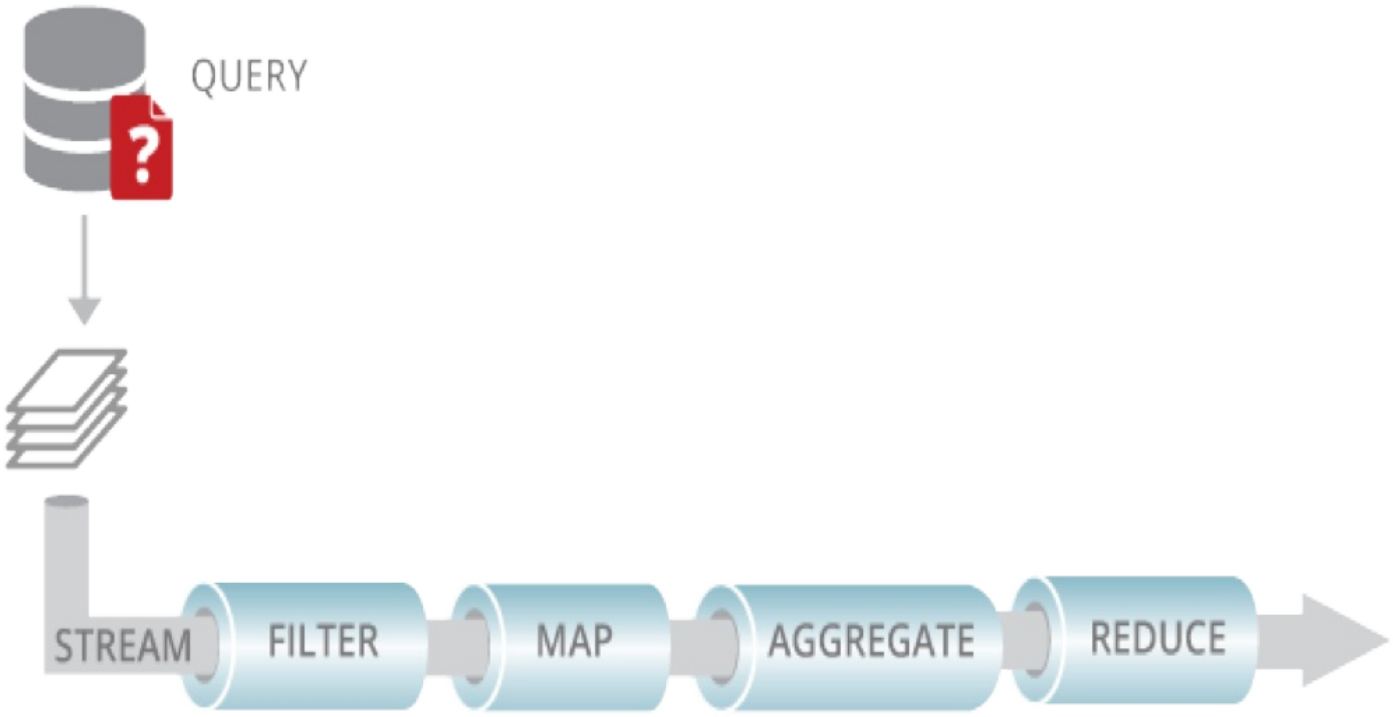
The Aerospike Labs example "Real-time analytics with Aerospike" shows code, and a real-world dataset, which determines which airline had late flights in January 2012.

https://github.com/aerospike/flights-analytics

Using a secondary index on a Bin containing the update timestamp, an Aggregation can can touch fewer records and quickly gather statistics on records that have recently changed. Compared to standard MapReduce systems, which act on an entire dataset without indexes.

Stream processing

The concept of stream processing is the the output of a query as tuples (records) flowing in a stream. The stream is READ-ONLY.



The contents of the stream could simply flow to the client as a standard result of the query, but by adding any number of aggregation functions the stream can be processed by one or more of the following function stereotypes:

Function stereotyopes

Function stereotypes	Description
Filter	One or more filters applied to “filter out” required. A filter function decides if the tu

	continue in the stream, or be removed fro
Map	One or more map function(s) are used to a tuple.
Aggregate	The aggregate function acts as an accumu elements in the stream info an intermedia List of top 10, etc.
Reduce	Reducing is the gathering of intermediary them into the final output results. The red each node, reducing the output from that reduce is executed on the client, reducing each node

You can have more that one kind of Aggregation stereotype function in the stream and they can be configured in any order that is semantically correct for your application.

Aerospike provides a library of Lua types that coincide with the types supported by the database. These are the basic type that can be used as parameters and return values.

Types in the stream

Type	Description
Stream	This is the stream its self and it is used to confi functions that process the stream
String	UTF-8 string
Integer	8 Byte unsigned integer
Bytes	The bytes type provides the ability to build a by using bytes and Integer - This type coincides wi type in the database.

Record	Represents database records, including Bins – (value) pairs – and metadata. **Note:** a record car final return type of the stream.
List	A list is data structure that represents a sequer values. A list can nest other lists and maps.
Map	A collection of (key, value) pairs, in which a ke appear once in the collection. A map can nest o and maps.

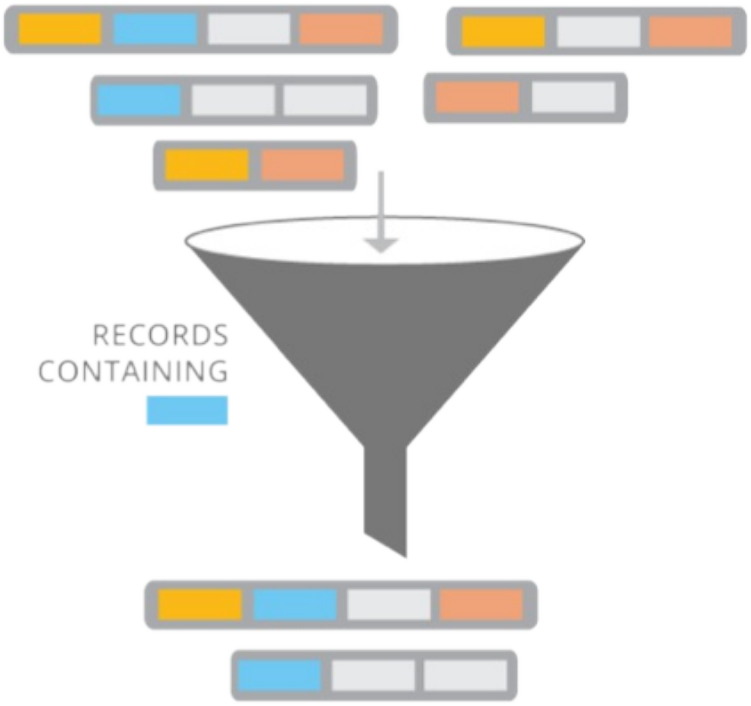
Aggregation functions

Aerospike uses UDFs to implement the stream operations. There are four basic function stereotypes:

- Filter
- Map
- Aggregate
- Reduce

Filter Operation

The filter operation is a classic pass filter implemented in code. The filter operation will filter values in the stream, either allowing them to pass through to the next step, or removing them from the scream.



The Filter function, written in Lua, accepts the current value from the stream and should return true or false, where true indicates the value should continue down the stream.

An example filter function is below. A query will feed records in to the stream. The filter operation will apply the filter function for each record in the stream. In this example, the filter function allows records containing "males" older than "18" years to be passed down the stream.

```
local function my_filter_fn(record)
     if record['age'] > 18 and record['sex'] == 'male' then
         return true
     else
         return false
     end
end
```

Zero or more filter functions can be configured to process the stream. This allows a modular, and generalized, filter design. You can construct a library of filters and reuse them.

Map operation

The map operation transforms values in the stream.

The map function accepts a value from the stream, and returns a value which will be passed to the next function in the processing chain. The type of the return value must be one of those supported by the database

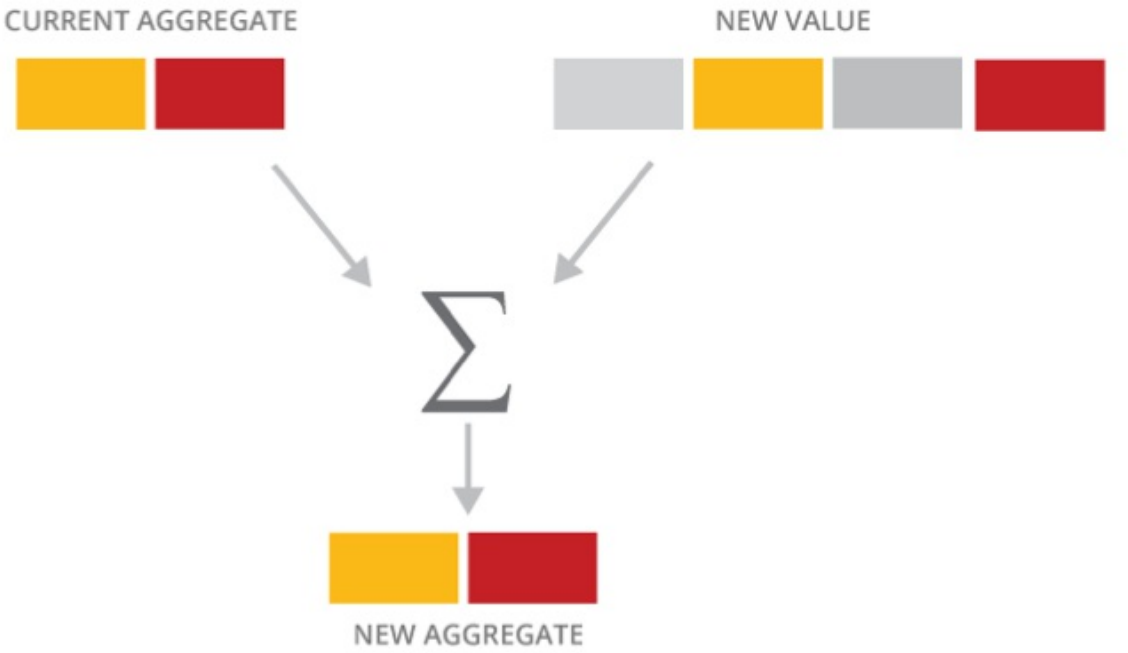
In this example map function a Record is transformed to a Map

```
local function my_map_fn(rec)
     local result = map()
     result['name'] = rec['name']
     result['city'] = rec['city']
     return result
end
```

An empty Map is created and the value of the Bins “name” and “city” are stored in the Map, and the Map is returned as the new element in the stream.

Aggregate Operation

Aggregate operation aggregates a stream of data into a single aggregate value. The aggregate operation function two arguments and returns one value.



The aggregate function arguments are:

the aggregate value

the next value from the input stream.

The function accumulates a stream of data into a single aggregate value by taking current aggregate, and the next value from the stream, to produce a composite aggregate.

The aggregate value and the return value should be of same type. The most efficient approach is to aggregate value passed in and combine it with the next value from the input stream.

A typical aggregate function looks as below. This example is doing a “group-by” type of operation where it is calculating the number of citizens in each city.

```
local function my_aggregation_fn(aggregate, nextitem)
```

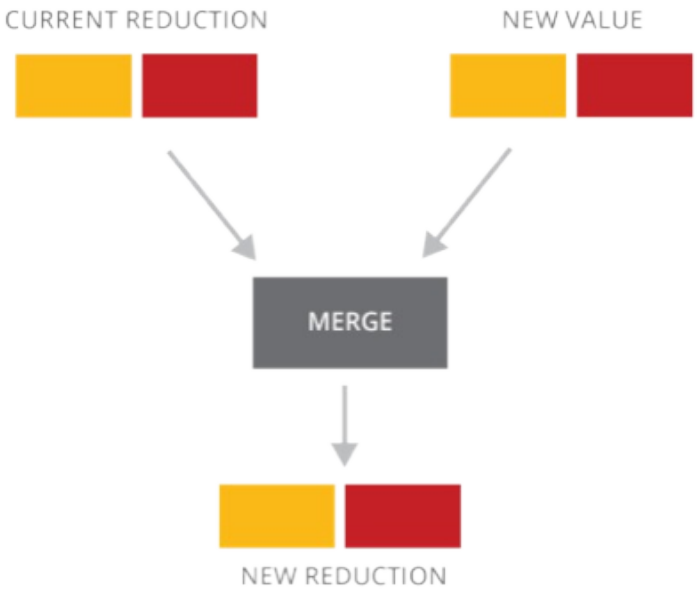
```
    -- If the count for the city does not exist, initialize it with 1
    -- Else increment the existing counter for that city
    aggregate[nextitem['city']] = (aggregate[nextitem['city']] or 0) + 1
    return aggregate
end
```

TIP: The aggregate function takes a collection elements from its input stream (across multiple invocations) and will return only one element to its output stream. There is little benefit putting two aggregate functions in a row, because the output of the first aggregate function will only emit single element which can be consumed by the next aggregate function.

The accumulated aggregate value can grow quite large. Make sure you have sufficient memory to accommodate the accumulated value.

Reduce Operation

The reduce operation will reduce values in the stream to a single value.



The reduce function accepts two values from the stream and

returns a single value that is fed back into the function as the first argument. The reduce function should be cumulative for the best performance. The two arguments of the reduce function and the return value should be of the same type. The type of the return value must be one of those supported by the database. The Reduce function executes on both cluster side and client side

The example function merges 2 Maps into a single Map.

```
local function my_merge_fn(val1, val2)
     return val1 + val2
end

local function my_reduce_fn(global_agg_value, next_agg_value)
     -- map.merge is a library function will call the my_merge_fn
     -- for each key and returns a new map.
     -- my_merge_fn will be passed the values of the key in two maps as arguments.
     -- It stores the result of merge function against the key in the result map
     return map.merge(global_agg_value, next_agg_value, my_merge_fn)
end
```

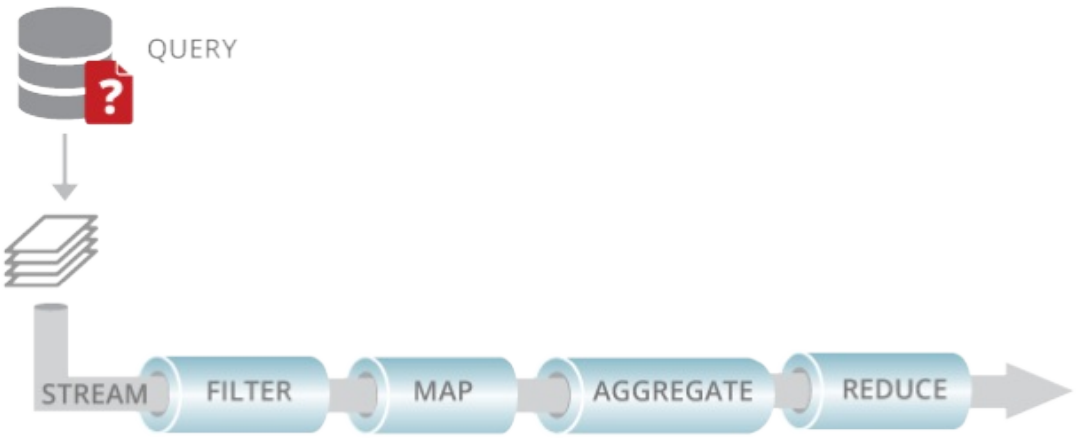
Note the use of an additional function the merges a single element of a Map.

One main characteristic of reduce function is that it executes both on the server nodes as well as the client side (in application instance). Each node first runs the data stream through the functions defined in the stream definition. The end result of this is sent to the application instance. The application gets results from all the nodes in the cluster. The client layer in it does the final reduce using the reduce function specified in the stream. So, the reduce function should be able to accumulate the intermediate values (coming form the cluster nodes).

If there is no reduce function, the client layer is passes all the data coming from the nodes to the application.

Stream UDF

A Stream UDF configures the stream processing, by specifying one or more of the stereotype functions and the order that they will be invoked in the stream. It needs to be registered with the cluster in a similar way to a Record UDF before it is available for use.



Consider this example.

```
local function aggregate_stats(map,rec)
  if rec.region == 'n' then
      map['n'] = map['n'] + 1
  elseif rec.region == 's' then
      map['s'] = map['s'] + 1
  elseif rec.region == 'e' then
      map['e'] = map['e'] + 1
  elseif rec.region == 'w' then
      map['w'] = map['w'] + 1
  end
  return map
end

local function reduce_stats(a,b)
  a.n = a.n +
  a.s = a.s + b.s
  a.e = a.e + b.e
  a.w = a.w + b.w
  return a
end
```

```
function sum(stream)
   return stream : aggregate(map{n=0,s=0,e=0,w=0},aggregate_stats) :
reduce(reduce_stats)
end
```

The Stream UDF is `sum()`, it has a single parameter `stream`, and this is the function that you execute from your application using the Aerospike client. In this example, there are no filter or map functions, only a single aggregate and reduce function.

The `aggregate()` method is invoked on the `stream` object, passing in 2 parameters:

an initial accumulation value - a map with 4 elements, n,s,e,w, each initialized to 0

a function pointer to the `aggregate_stats` function

The `reduce()` method is invoked on the `stream` object, passing in 1 parameter that is a function pointer to the `reduce_stats` function.

The following table show where and how frequent the functions are invoked

Function	Where	When
Filter	On same cluster node as the element	Each element in the strean
Map	On same cluster node as the element	Each element in the strean
Aggregate	On same cluster node as the element	Each element in the strean

Reduce	On each cluster node and on the client	On each node for each ele On the client for each noc

Stream functions are invoked on the node where the stream element is produced. The Filter, Map and Aggregate functions are executed on the same node as the stream element, and as the element is produced (from a query or previous stage of the stream). The Reduce function is invoked on each cluster node, for each element in the stream, and on the client for each node in the cluster.

Executing an Aggregation

Executing an aggregation from your application is very similar to executing a query. The difference is that you specify the Stream UDF module and function name that will process the stream of tuples flowing from the query.

Prepare and execute a query

A query is prepared and executed using a Statement. A ResultSet is returned from the invocation and it is very similar to a RecordSet containing a blocking queue and a protocol connection to the query jobs running on the cluster nodes. A ResultSet will contain one or more of the type that you return from your Stream UDF.

Example code:aggregation execution

```
// Java prepare and execute
String[] bins = { "username", "tweetcount", "gender", "region" };
Statement stmt = new Statement();
stmt.setNamespace("test");
stmt.setSetName("users");
stmt.setIndexName("tweetcount_index");
stmt.setBinNames(bins);
stmt.setFilters(Filter.range("tweetcount", min, max));

ResultSet rs = client.queryAggregate(null, stmt, "aggregation_region", "sum");

// PHP execute
$where = $db->predicateBetween("tweetcount ", $min, $max);
$status = $db->aggregate("test", "users", $where, "aggregation_region", "sum",
       array("username", "tweetcount", "gender", "region"), $result);
if($status !== Aerospike::OK) {
 . . .
```

Processing the Results

Results are returned in a ResultSet collection described above.

They are processed by iterating over the collection. When you have finished processing the ResultSet, make sure you close it as this will signal the query jobs, running on each node, to terminate. Valid types in a ResultSet:

String

Integer

List

Map

Bytes

Example code:ResultSet processing

```
// C# process results
ResultSet rs = client.QueryAggregate(null, stmt, "aggregation_region", "sum");
if (rs.Next())
{
       Dictionary<object, object> result = (Dictionary<object, object>)rs.Object;
       Console.WriteLine("Total Users in North: " + result["n"]);
       Console.WriteLine("Total Users in South: " + result["s"]);
       Console.WriteLine("Total Users in East: " + result["e"]);
       Console.WriteLine("Total Users in West: " + result["w"]);
}

// Java process results
ResultSet rs = client.queryAggregate(null, stmt, "aggregation_region", "sum");
if (rs.next()) {
 Map<Object, Object> result = (Map<Object, Object>) rs
       .getObject();
 console.printf("\nTotal Users in North: " + result.get("n") + "\n");
 console.printf("Total Users in South: " + result.get("s") + "\n");
 console.printf("Total Users in East: " + result.get("e") + "\n");
 console.printf("Total Users in West: " + result.get("w") + "\n");
}
```

Example: Average

In this simple Average example we are averaging the the total number of tweets between a range, by getting the sum and count.

```
// Java averages example
Statement stmt = new Statement();
```

```
stmt.setNamespace("test");
stmt.setSetName("users");
stmt.setFilters(Filter.range("l1", 0, 1000));
ResultSet rs = client.queryAggregate(null, stmt, "average_example", "average");
```

```
-- Lua averages module
function average(s)

    local function accumulate(out, rec)
        out['sum'] = (out['sum'] or 0) + (rec['l1'] or 0)
        out['count'] = (out['count'] or 0) + 1
        return out
    end

    local function reducer(a, b)
        local out = map()
        out['sum'] = a['sum'] + b['sum']
        out['count'] = a['count'] + b['count']
        return out
    end

    return s : aggregate(map{sum = 0, count = 0}, accumulate) : reduce(reducer)
end
```

The stream function “average” operates on the Bin “l1”. The function is configured with a:

Aggregate operation implemented by the function `accumulate`. It accumulates the `sum` value and the `count` value in a Map. Note: this map is created once and add to on each invocation of the `accumulate` function.

Reduce operation implemented by `reducer`, simply adds the `sum` values and `count` values then returns a Map containing these two values.

The Map is returned to the application where it can the be used to perform a simple average calculation.

Remember: Make your functions lean. The Aggregate operation is invoked once for every record in the output of the query. The Reduce operation is invoked for: every data partition, every node

in the cluster, and once on the client.

UDF

User Defined function

Lua

Lua is a powerful, fast, lightweight, embeddable scripting language.

Module

A module is a collection of Lua functions grouped together in a single file

RecordSet

A RecordSet manages record retrieval from queries. Multiple threads will retrieve records from the server nodes and put these records on the queue. The single user thread consumes these records from the queue.

Namespace

Namespaces are the top level containers for data and define storage and default policies.

Set

A Set is similar to a table as it is a collection of records, but it has no schema. It is a logical collection of records

Record

A record is the basic unit of storage and is a row that is the value pointed to by the primary key.

String

A string is a database type stored as UTF-8

Integer

Integers are 8 byte unsigned

Key

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Digest

The primary key value is hashed using th RIPEMD160 algorithm to produce a 20 byte (160 bit) digest.

Metadata

Data stored with each record comprising of Generation and Expiry (Time to live)

ResultSet

A ResultSet is a iterable collection that is returned from an aggregation execution.

Aql

aql provides an SQL-like command line interface for database, UDF and index management. Aerospike does not

support SQL as a query or management language, but instead provides aql to provide an interface similar to tools that utilize SQL.

Bin

Bins are the equivalent of columns or fields in a conventional RDBMS. A Bin consists of a name and a value.

Bytes

Bytes are byte arrays of a specific size. This allows any binary data of any type to be stored.

List

List is a collections of values ordered in the insert order. A list may contain values of any of the supported data types, including other Lists and Maps.

Map

Map is a collection of key-value pairs, such that each key may only appear once in the collection and is associated with a value. The key and value of a map may be of any of the supported data types, including other Lists and Maps.

ResultCode

The result code is the value returned for each operation from the cluster

Associative arrays

associative array, map, symbol table, or dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

Record UDF

A Record UDF operates on a single record.

Stream UDF

A Stream UDF operates on a stream of records flowing as the results of a Query

Statement

A Statement is a data structure that provides parameters to the query, in object oriented languages, it is an object instance.

Query

An Aerospike query provides value-based look up through the use of secondary indexes.

Aggregation

Aggregations are a programmatic framework that is similar to a MapReduce system, in that an initial query emits a stream of results in a highly parallel fashion.

Double

Double-precision floating-point format is a computer number format which occupies 8 bytes (64 bits) in computer memory and represents a wide, dynamic range of values by using a floating point. -- Wikipedia

UDF

User Defined function

Related Glossary Terms

Aggregation, Record UDF, Stream UDF

Index

Lua

Lua is a powerful, fast, lightweight, embeddable scripting language.

Index

Module

A module is a collection of Lua functions grouped together in a single file

Index

RecordSet

A RecordSet manages record retrieval from queries. Multiple threads will retrieve records from the server nodes and put these records on the queue. The single user thread consumes these records from the queue.

Related Glossary Terms

Query, Record

Index

Namespace

Namespaces are the top level containers for data and define storage and default policies.

Index

Set

A Set is similar to a table as it is a collection of records, but it has no schema. It is a logical collection of records

Index

Record

A record is the basic unit of storage and is a row that is the value pointed to by the primary key.

Related Glossary Terms

RecordSet

Index

Chapter 2 - Data Hierarchy
Chapter 4 - Read operations
Chapter 6 - Executing a Query
Chapter 6 - Executing a Query

String

A string is a database type stored as UTF-8

Index

Integer

Integers are 8 byte unsigned

Index

[Chapter 2 - Data Types]

Key

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

Index

Digest

The primary key value is hashed using th RIPEMD160 algorithm to produce a 20 byte (160 bit) digest.

Index

Metadata

Data stored with each record comprising of Generation and Expiry (Time to live)

Index

[Chapter 2 - Data Hierarchy]

ResultSet

A ResultSet is a iterable collection that is returned from an aggregation execution.

Index

Aql

aql provides an SQL-like command line interface for database, UDF and index management. Aerospike does not support SQL as a query or management language, but instead provides aql to provide an interface similar to tools that utilize SQL.

Index

Bin

Bins are the equivalent of columns or fields in a conventional RDBMS. A Bin consists of a name and a value.

Index

Bytes

Bytes are byte arrays of a specific size. This allows any binary data of any type to be stored.

Index

[Chapter 2 - Data Types]

List

List is a collections of values ordered in the insert order. A list may contain values of any of the supported data types, including other Lists and Maps.

Index

[Chapter 2 - Data Types]

Map

Map is a collection of key-value pairs, such that each key may only appear once in the collection and is associated with a value. The key and value of a map may be of any of the supported data types, including other Lists and Maps.

Index

[Chapter 2 - Data Types]

ResultCode

The result code is the value returned for each operation from the cluster

Index

[Chapter 4 - Errors and Result Codes]

Associative arrays

associative array, map, symbol table, or dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

Index

[Chapter 5 - Lua]

Record UDF

A Record **UDF** operates on a single record.

Related Glossary Terms

Stream UDF, UDF

Index

Stream UDF

A Stream UDF operates on a stream of records flowing as the results of a Query

Related Glossary Terms

Aggregation, Record UDF, UDF

Index

Statement

A Statement is a data structure that provides parameters to the query, in object oriented languages, it is an object instance.

Index

Query

An Aerospike query provides value-based look up through the use of secondary indexes.

Related Glossary Terms

RecordSet

Index

Chapter 6 - Queries

Aggregation

Aggregations are a programmatic framework that is similar to a MapReduce system, in that an initial query emits a stream of results in a highly parallel fashion.

Related Glossary Terms

Stream UDF, UDF

Index

Chapter 7 - Aggregations

Double

Double-precision floating-point format is a computer number format which occupies 8 bytes (64 bits) in computer memory and represents a wide, dynamic range of values by using a floating point. -- Wikipedia

Index

[Chapter 2 - Data Types]