

O'REILLY®

# Aerospike

## Up & Running

Developing on a Modern Operational Database  
for Globally Distributed Apps



**Early  
Release**

Raw & Unedited

Compliments of



Infinitely possible.™

V. Srinivasan  
Tim Faulkes, Albert Autin  
& Paige Roberts



# Infinitely possible.™



The database built  
for infinite scale,  
speed, and savings.



Learn more!  
[aerospike.com](https://aerospike.com)



---

# Aerospike: Up and Running

*Developing on a Modern Operational Database  
for Globally Distributed Apps*

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

*Srini V. Srinivasan, Tim Faulkes, Albert Autin, and  
Paige Roberts*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY**®

## **Aerospike: Up and Running**

by Sriniv V. Srinivasan, Tim Faulkes, Albert Autin, and Paige Roberts

Copyright © 2024 O'Reilly Media inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editors:** Andy Kwan and Gary O'Brien

**Production Editor:** Aleeya Rahman

**Interior Designer:** David Futato

**Cover Designer:** Karen Montgomery

**Illustrator:** Kate Dullea

September 2024: First Edition

### **Revision History for the Early Release**

2023-08-28: First Release

2023-11-21: Second Release

2024-03-07: Third Release

2024-05-28: Fourth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098155605> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Aerospike: Up and Running*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Aerospike. See our [statement of editorial independence](#).

978-1-098-15560-5

[LSI]

---

# Table of Contents

<b>1. Developing Your First Aerospike Application.....</b>	<b>9</b>
Installing Aerospike	9
Installing Aerospike on a Mac	10
Installing Aerospike on Windows	11
Installing Aerospike on Linux	14
Aerospike Terminology	14
Simple First Application	17
Using Aerospike Quick Look	21
Policies	24
Summary	26
<b>2. Basic Operations.....</b>	<b>27</b>
CRUD Operations	28
Data Types	28
Data Lifecycle	29
Create	30
Read	32
Update	34
Delete	35
Lightweight Operations	36
Batch	36
Wrapping Up	37
<b>3. Advanced Operations.....</b>	<b>39</b>
The Operate() command	39
Simplifying the program	41
The Operation class	42
Return value of Operate()	42

Order of Operations	43
ListOperation and MapOperation:	44
Lists	44
Maps	46
Operations	47
ListOperations and MapOperations Example	47
Inverted Flag	49
Contexts	50
Expressions	51
Filter Expressions	51
Trilean Logic	52
Read Expressions	53
Batch Operations	56
Batch Writes	56
Arbitrary Batch Operations	58
Secondary Indexes	59
Using the Secondary Index	61
Multiple Predicate Queries	62
<b>4. Architecture.....</b>	<b>65</b>
Scale Out	65
Shared Nothing Database Cluster	66
Data Distribution	67
Cluster Self-Management	69
Cluster view	70
Cluster view changes	71
Intelligent clients	71
Cluster node handling	72
Scale Up	72
Hybrid Memory Architecture	73
Multi-Core Processors	75
Memory Fragmentation	76
Data Structure Design	76
Scheduling and Prioritization	77
Parallelism	78
Distributed Transaction Consistency	79
Strong Consistency in Transactions	80
Roster	80
Split-Brain Conditions	81
Writes	84
Rack Awareness	85
Asynchronous active-active replication	86



Conclusion	87
<b>5. Data Modeling.....</b>	<b>89</b>
Classical Data Modeling	89
Aerospike Data Modeling	90
Secondary Indexes	59
Aggregating Sub-Objects into One Record	92
Aggregating Sub-Objects into Multiple Records	93
Associating Objects	101
Other Common Data Modeling Problems	107
Conclusion	87
<b>6. Administration, Tools, and Configuration.....</b>	<b>115</b>
Configuration	115
Anatomy of a config file	116
Dynamic Configuration	122
Tools	124
Asinfo	124
Asadm	128
Aerospike Quick Look	133
Asbackup and Asrestore	135
Asloglatency , asbench	137
Security	140
Recap	140
<b>7. Monitoring and Best Practices.....</b>	<b>141</b>
Monitoring	141
Application metrics	142
Aerospike database metrics	142
Prometheus	143
Aerospike exporter	147
Aerospike Exporter Service Install	147
Aerospike Exporter container	148
Metric Reference	149
Alert Rules & Dashboards	150
Alert Rules	150
Dashboards	151
Software Updates	152
Preparation	152
Upgrade	153
Quiesce	155
Troubleshooting	155



---

# Developing Your First Aerospike Application

## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

Now we know what Aerospike is, let’s start writing our first program using Aerospike as the database. Aerospike supports multiple programming languages such as Java, C, C#, Python, and Node.js. but in this book we’re going to focus on Java and Python. If you are following along, be aware that the Python client works on Mac and Linux, but not Windows. Otherwise, the concepts discussed will apply to all the programming languages Aerospike supports. For a full list of these languages, which client versions work with which Aerospike server, and what features are supported, take a look at the [Aerospike Client Matrix](#).

## Installing Aerospike

There are two components of Aerospike that you will need to install. The first is the actual Aerospike database and the second component includes the tools used to interact with the database. The database install only includes these tools for Linux.

If you are using Windows or Mac OS, and you would like the convenience of not having to call Docker first before each command, a native version of the tools for your current operating system (OS) will be required.

Since Aerospike is designed for the Linux operating system, it will not run natively on either Windows or Mac OS. However, there are simple ways to run Aerospike on these machines. Let's look at how.

## Installing Aerospike on a Mac

Running the database on a Mac requires using a virtualization layer like Docker or Virtual Box to run Linux. Docker or a similar containerized solution is arguably easier to get started, so we will focus on that. (If you're using a full virtualization layer like Virtual Box, then follow the instructions for Linux installation.)

You can use either [Docker Desktop](#), which, at the time of writing, is free for personal use, or [Docker Laptop](#). Docker Desktop is a commercial product with good support and is probably the easiest option to use so long as you comply with the license agreement. Docker Laptop is a good choice if the license agreement precludes the use of Docker Desktop or you have strong familiarity with running Docker on Linux. Either option should work fine to install a simple cluster.

### Installing the Aerospike Database Container

Once you've installed Docker, use it to pull down the latest copy of the Aerospike database and install it. There are two different versions of Aerospike: The fully open source Community Edition or the Enterprise Edition. The Enterprise Edition includes all the features of the Community Edition plus other enterprise features like encryption at rest, compression, durable deletes and so on. This edition is free to download and use for a one node database, but clustering more than one node requires a license. Multi-node clusters are free in the Community Edition but it lacks some of the features of the enterprise edition.

So that we have the complete functionality, we're going to use the Enterprise Edition for the programming part of this book, but either will work and we will draw distinctions when discussing features that are enterprise only.

To get the latest Docker image and start the Aerospike server

```
% docker run -tid --name aerospike -p 3000-3002:3000-3002 \ aerospike/aerospike-server-enterprise
```

This will pull down the latest version of the Aerospike database and start up a container. Let's look at the parameters to the Docker command:

- `run` tells Docker to launch a program

- `-tid` specifies the program should run detached and interactively, so we can attach to the process later and look at logs later if we should choose to do so
- `--name` gives the Docker process a name, so we can use this name to manage the process through Docker rather than using the container id.
- `-p 3000-3002:3000-3002` exposes the container's ports of 3000, 3001 and 3002 to the corresponding ports on the underlying host machine. This means we can attach our favorite IDE to the database without needing to worry about the database running in a container. Port 3000 is the port which the client will talk to the server on so is most important from the developer perspective.
- `aerospike/aerospike-server-enterprise` is the name of the image to download, in this case the enterprise version of Aerospike. At the time of writing, the latest version of Aerospike Enterprise Edition is 6.4.0.1.

To ensure this ran successfully, execute:

```
% docker ps
```

This should display something similar to the following:

```
CONTAINER ID   IMAGE
COMMAND          CREATED        STATUS
PORTS           NAMES
979765d925df   aerospike/aerospike-server-enterprise   "/usr/bin/as-tini-st..."
2 minutes ago   Up 2 minutes   0.0.0.0:3000-3002->3000-3002/tcp, :::3000-3002->3000-3002/tcp   aerospike
```

## Installing Tools

On a Mac, running the tools locally eliminates the need to run every command through the Docker exec, making the commands simpler. To get the latest version of the tools used to interact with Aerospike, go to <https://www.aerospike.com/download>, select Tools, then Aerospike Tools. In the Package dropdown, select the appropriate version for your Mac – either macOS x86\_64 for Intel-based Macs, or macOS arm64 for Macs based on the “M” series of processors.

Once this download is complete, run the binary to be guided through the installation process. The installation should complete quickly and you can validate that the installation was successful by launching a terminal window and executing:

```
% which asinfo
/usr/local/bin/asinfo
```

## Installing Aerospike on Windows

Windows users have two different options for installing and running Aerospike:

1. Using Docker

## 2. Using Windows Subsystem for Linux (WSL)

Let's examine each of these in turn.

### Docker Install on Windows

Docker is a better choice than WSL for serious development on Windows as it allows Aerospike to easily run as a daemon. WSL typically is not designed for daemon style programs. Similar to running on a Mac, Docker Desktop is one of the easiest ways of getting Linux running on Windows. Download and install on your system then at a command prompt execute:

```
% docker -v
```

This should give output similar to:

```
Docker version 23.0.5, build bc4487a
```

To run Aerospike, execute the command:

```
% docker run -tid --name aerospike -p 3000-3002:3000-3002 aerospike/aerospike-server-enterprise
```

For a full understanding of the parameters to this command, see the description of the same command in the section for installing on Macs. The two key parts of the command:

- `-p 3000-3002:3000-3002` exposes the container's ports of 3000, 3001 and 3002 to the corresponding ports on the underlying host machine. This means we can attach our favorite IDE to the database without worrying about the database running in a container. Port 3000 is the port on which the client will talk to the server so it is most important from the developer's perspective.
- `aerospike/aerospike-server-enterprise` is the name of the image to download, in this case the enterprise version of Aerospike.

There is no native build for Aerospike Tools on Windows, so Docker can also be used for the tools. To run Aerospike Quick Lookup (AQL) to browse the database for example, execute:

```
% docker run -ti --name aerospike-tools --rm aerospike/aerospike-tools aql -h <ip_address> --no-config-file
```

In this case `<ip_address>` should be replaced with the actual IP address of the host machine. This can be determined using the `ipconfig` command, and should not be `localhost` (127.0.0.1). This allows the two containers to communicate with each other.

## Windows Subsystem for Linux

If you're running on Windows 10 version 2004 or higher, you can install a Linux distribution like Ubuntu on your Windows machine. Full instructions can be found at <https://learn.microsoft.com/en-us/windows/wsl/install>. The process is fairly easy. Launch a command prompt with Administrator rights, then execute

```
C:\WINDOWS\system32>wsl --install
```

This will install a version of Ubuntu by default, which is a good choice for running Aerospike. Once installation is complete you will need to reboot your computer for the changes to take effect and Linux to be installed. After rebooting, you will be prompted to enter your Linux username and change your password. Then it will display a welcome message. This message should contain a line similar to:

```
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.10.102.1-microsoft-standard-WSL2 x86_64)
```

Once this appears, use a browser to navigate to <https://download.aerospike.com/artifacts/aerospike-server-enterprise/>. Here you will find a list of versions of Aerospike available for download. Select the right version for your Linux OS, then download the file corresponding to your system.

For example, after installing version 6.4.0.1 (the latest version at the time of writing), with WSL version of Ubuntu 20.04 on an Intel chipset, you would download the file `aerospike-server-enterprise_6.4.0.1_tools-8.3.0_ubuntu20.04_x86_64.tgz`.

Once you've downloaded the server, untar it. Change to the appropriate sub-directory which was created as part of the tarball extraction. Finally, run:

```
% sudo ./asinstall
```

This will install Aerospike in the appropriate folders on your system, as well as the system tools.

Each connection from a client is considered a file in Linux terminology, so increase the limit on the allowed number of files a user can open at a time. The default is typically 1,024, but it's worth bumping this up to something much higher by using:

```
% ulimit -n 100000
```

This will need to be done whenever you start a WSL process so it's worth putting this in your `~/.bashrc` file or changing `/etc/security/limits.conf` to reflect this.

To start the Aerospike daemon, invoke:

```
% sudo asd
```

This will start Aerospike. Check that it's working properly by running:

```
% asadm -e info
```

`asadm` is the Aerospike Administration tool which we will discuss in detail later

in the book, but this command should show information about your cluster. It will begin with something like:

```
/mnt/c/WINDOWS/system32$ asadm -e info
Seed:          [('127.0.0.1', 3000, None)]
Config_file:  /Users/book/.aerospike/astools.conf, /etc/aerospike/astools.conf
```

Following this, there will be information about the network configuration and the namespaces. If you see this, congratulations, your Aerospike cluster is running! If you do not see this information, try adding `--foreground` to the `asd` command, which will dump the configuration file and log information to the console so you can examine errors quickly.

## Installing Aerospike on Linux

Because Aerospike runs natively on Linux it is easy to install on this platform. Use a browser to navigate to <https://download.aerospike.com/artifacts/aerospike-server-enterprise/>. Here you will find a list of versions of Aerospike available for download. Select the right version, then download the file corresponding to your system.

For example, after installing 6.4.0.1 (the latest version at the time of writing) with WSL version of Ubuntu 20.04 on an Intel chipset, you would download the file *aerospike-server-enterprise\_6.4.0.1\_tools-8.3.0\_ubuntu20.04\_x86\_64.tgz*.

Once you've downloaded the server, untar it. Change to the appropriate sub-directory which was created as part of the tarball extraction. Finally, run:

```
% sudo ./asinstall
```

This will install Aerospike in the appropriate folders on your system, as well as the system tools.

On Linux, Aerospike is installed as a service to make maintaining the system easier. Depending on your Linux variant, either use the `system` or the `systemctl` command to check the status of the service and start it if needed:

```
% sudo systemctl status aerospike
% sudo systemctl start aerospike
```

## Aerospike Terminology

Aerospike has a number of concepts that are similar to, but not exactly the same as, other databases you might be familiar with. To understand these concepts, let's compare Aerospike to a relational database (RDBMS).



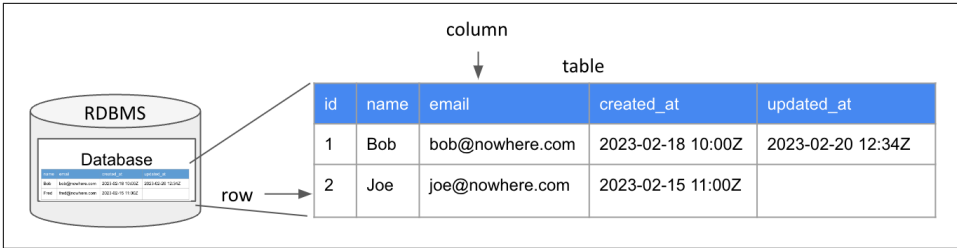


Figure 1-1. Relational Data Model

In **Figure 1-1** we can see the typical structure of a relational row-based database. The data is grouped into databases, with each database being logically isolated from one another. Each database contains tables where a table is a logical grouping of rows, aka records, such as Customer or Account.

Tables have a schema in an RDBMS and hence every row in the table has the same columns. This schema dictates what columns should exist, the types of the columns, whether null values are allowed and so on.

Rows contain the data, which is broken down into columns with each column holding a single piece of information. Since the schema enforces all rows having the same structure, if a piece of data is not needed, it still needs a null entry in the row. For example, Joe’s row in **Figure 1-1** does not have an updated\_at entry, so this will be null.

In contrast, Aerospike’s data model is shown in **Figure 1-2**.

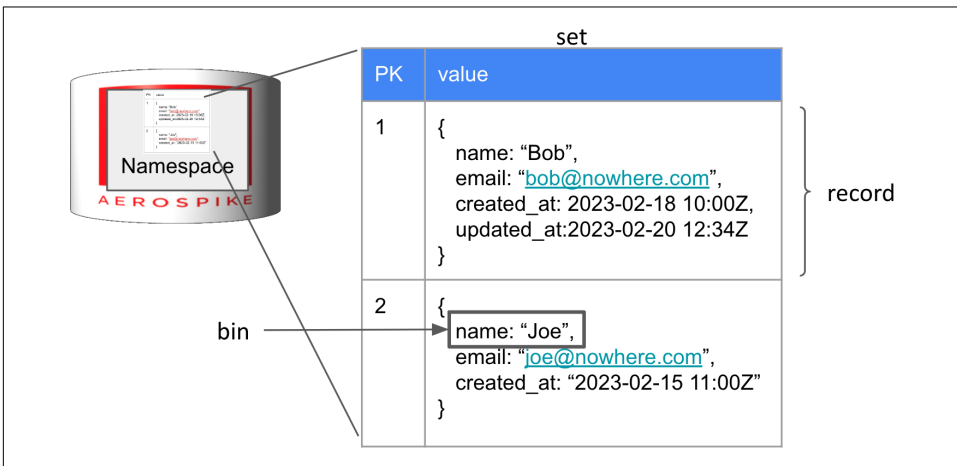


Figure 1-2. Aerospike’s Data Model

The top level storage concept is the *namespace*, a logical organization of named objects Aerospike is working with. In this case, the namespace functions similar to a

tablespace or a database in relational concepts. It defines the storage being used for the data and indices, how many copies of the data to store across the cluster and so on. Namespaces are typically isolated from each other so that if different namespaces store data on the same SSDs, they do not suffer from “noisy neighbor” problems associated with the drives.

A *set* is a logical grouping of records, similar to a table in an RDBMS. While a table has a schema which strictly defines the structure of all rows in that table, Aerospike sets are schemaless and therefore can store either structured or unstructured data.

A *record* is akin to a row in a relational database. It contains a number of discrete items of data in *bins* (similar to a column in a relational database). Since Aerospike is schemaless, each record is self-describing and so can be thought of as similar to a map data type, with the bin name being the key to the map and the value being the bin’s contents.

The value inside a bin can be a scalar like a string, integer or float, or complex like a list or map. These lists and maps (collectively called *Container Data Types* or CDTs for short) can be nested inside one another, allowing for arbitrarily complex data structures within a single bin.

Note that in the example in [Figure 1-2](#), Record 2 does not have an “updated\_at” field, similar to the RDBMS example in [Figure 1-1](#). However, unlike the RDBMS, the value in Aerospike is not null; it is just not present in the record. As each record is self-describing, different records can have different numbers or types of bins. Similarly, different records can have different types in bins with the same name. For example, one record might have a bin called “data” containing an integer and another with a bin with the same name containing a string. This is rare as the application that generates the data normally stores the same sort of data in records in the same set, hence effectively enforcing a schema on the set.

Also note that each bin inside a record has a known type. The type is set when the data is entered in that bin, and changing the contents of the bin will change the type. Consider the following Java snippet:

```
client.put(null, key, new Bin("data", 10)); // long
client.put(null, key, new Bin("data", 10.0)); // double
client.put(null, key, new Bin("data", "10")); // String
```

In this case, the same bin is being set to three different values, although the representations are similar. In the first example, we’re passing 10 (an integer) to the bin, so it will be stored as an integer. In the second example, the passed value is a double value, so Aerospike will store the value as a double, and in the last example the value will be stored as a string. Note that there is not an obvious API call to determine the type of a bin. The bin can be read and the contents examined, or the bin type can be determined using an Expression, which will be covered in Chapter 4.

It is also worth noting that Aerospike internally always stores integral values in a 64-bit integer, and floating point values in a 64-bit double value. Hence there is no difference between storing ten as an integer, a long or a short for example – Aerospike will treat them all identically.

The type of the bin can affect the operations which can be performed on that bin. For example, if a string is stored in a bin, a string append operation could be performed on it but adding an integer to it could not. If the bin value is replaced with a new value of a different type, the type of the new bin determines the allowed operations going forwards.

## Simple First Application

Now that we understand the data model, let's go and create a simple first Java program. In this case we're just going to connect to the database, insert a record into the `Item` set in the `test` namespace then read that record back and print out information contained in the record.

```
##Java
public class FirstProgram {
    public static void main(String[] args) {
        IAerospikeClient client = new AerospikeClient(null, "127.0.0.1", 3000);
        Key key = new Key("test", "item", 1);
        client.put(null, key, new Bin("name", "Stylish Couch"),
            new Bin("cost", 50000), new Bin("discount", 0.21));
        Record item = client.get(null, key);
        System.out.printf("%s costs $%.02f with a %d%% discount\n",
            item.getString("name"), item.getInt("cost")/100.0,
            (int)(100*item.getFloat("discount")));
        client.close();
    }
}
```

```
##Python
import aerospike

client = aerospike.client({'hosts':[('127.0.0.1', 3000)]}).connect()
key = ('test', 'item', 1)
bins = {
    'name': 'Stylish Couch',
    'cost': 50000,
    'discount': 0.21
}
client.put(key, bins)
(key_, meta, bins) = client.get(key);
print('Record:', bins)
client.close()
```

This program uses the AerospikeClient driver in Java so you will need to import it. How to import the library depends on the build tool you use. For example, using Apache Maven, you would add a dependency similar to:

```
<dependency>
  <groupId>com.aerospike</groupId>
  <artifactId>aerospike-client</artifactId>
  <version>6.1.10</version>
</dependency>
```

In Gradle, this would be:

```
implementation group: 'com.aerospike', name: 'aerospike-client', version: '6.1.10'
```

For specifics on other languages or other build tools or to simply download the client as a JAR file, visit <https://download.aerospike.com/download/client/> and navigate to the appropriate language.

Now, let's examine our first program.

## Establishing a Connection to the Database

The first thing we need to do is establish a connection to the database:

```
## Java
IAerospikeClient client = new AerospikeClient(null, "127.0.0.1", 3000);

## Python
client = aerospike.client({'hosts':[('127.0.0.1', 3000)]}).connect()
```

Establishing a connection to the database needs just 3 parameters:

- The `ClientPolicy` which dictates how to connect to the database. Almost all operations in Aerospike take a policy of one sort or another as the first argument and we will cover policies in detail shortly. Passing `null` here tells Aerospike to use its defaults.
- The server's host address. This can be a hostname, DNS address or IP address.
- The port on the host that Aerospike is listening on. This is port 3000 by default.

Note that in a production system this code wouldn't be great. Aerospike is a clustered database so there are typically multiple machines that hold the data with all machines being identical. This eliminates any single points of failure in the cluster. However, our code is connecting to just a single node, and if that node goes down the application cannot connect. A more usual constructor would pass multiple addresses and look more like:

```
## Java
IAerospikeClient client = new AerospikeClient(clientPolicy,
```

```

    new Host("172.17.0.2", 3000),
    new Host("172.17.0.3", 3000),
    new Host("172.17.0.4", 3000));

## Python
config = {
    'hosts': [
        ('172.17.0.2', 3000),
        ('172.17.0.3', 3000),
        ('172.17.0.4', 3000)
    ]
}

```

These nodes are known as *seed hosts*. When the `AerospikeClient` is created, `Aerospike` starts at the first seed host in the list and attempts to establish a connection to the cluster that includes that node. If it cannot connect, it moves on to the next seed host and so on. An exception is thrown if no connection can be established and all the seed hosts have been tried.

However, if one of the seed hosts resolves to a cluster, `Aerospike` uses this cluster as its database and no further seed hosts are attempted. The response back from the seed host contains information about all the nodes in the cluster. When the client receives this information, it then attempts to establish a pool of connections with each one of these nodes.

The `Aerospike` clusters are dynamic. Nodes can be added and removed at any time. Node removal can either be planned, such as taking a node down to patch its operating system, or unplanned such as a hardware failure taking the node down. Hence the client needs to continually refresh its view of what nodes are in the cluster. This process is known as *tending the cluster*, which happens once per second by default. If the client does have an inaccurate view of the cluster and sends a request to the wrong node, the cluster is smart enough to proxy that request to the node which can serve that piece of data. This will be covered in more detail later in the book.

At the end of the application the client should be closed with a call to the `close()` method to free up resources on both the client and server.

```
client.close();
```

Note that once a client has been closed, it cannot be used again. So, if, for example, you have an online program that creates an `AerospikeClient` designed to run continuously and you then occasionally run a batch job which re-uses the same client, the batch program should not call `close()` when it completes as this will affect the online application.

Once created, the client is designed to be multi-threaded. Typically, only a single client is needed in a single program; all requests to the database should use that same client. As can be seen from this description, creating a new `AerospikeClient` is

an expensive operation – multiple seed nodes are potentially tried, connection pools created and the thread that tends the cluster initialized. So creating and closing clients will slow the application down dramatically.

## Inserting and retrieving data

To insert or retrieve data from Aerospike by primary key, a Key must be created. Keys have three components:

1. The namespace
2. The set name
3. The primary key value

So, in our case, we create a key:

```
## Java
Key key = new Key("test", "item", 1);
```

```
## Python
key = ('test', 'item', 1)
```

Our namespace is “test” and we’re going to be using the record with the primary key of 1 in the “item” set.

To insert a record of data in the item set, we need to give Aeropike the policy, key, and the data we want to insert:

```
## Java
client.put(null, key, new Bin("name", "Stylish Couch"),
          new Bin("cost", 50000), new Bin("discount", 0.21));
```

```
##Python
bins = {
    'name': 'Stylish Couch',
    'cost': 50000,
    'discount': 0.21
}
client.put(key, bins)
```

This will create a new record in the database with three columns – name (a string), cost (an integer) and discount (a float).



Note in Aerospike, all operations by default are UPSERTs – if the record exists, it will be updated, if the record does not exist, it will be created. Similarly, if the set does not exist, Aerospike will automatically create that when data is inserted.

Aerospike is also case sensitive, so the set `testSet` is different to the set `TestSet` for example. However, keywords like “SELECT” are case insensitive when using tools like Aerospike Quick Look, introduced below

To retrieve a record, supply the policy and the key you want to retrieve:

```
## Java
Record item = client.get(null, key);
```

```
## Python
(key_, meta, bins) = client.get(key);
```

Note that the `Record` class being used here is `com.aerospike.client.Record`. Java 16 introduced a `java.lang.Record` which some IDEs will try to use by default, so you will need to explicitly list `com.aerospike.client.Record` in the import statements for the class.

The returned `Record` will contain all the values in the record which can be retrieved by invoking `get` methods. If the type of the data being retrieved is known, specific methods like `getString()`, `getInt()` and so on can be used. If the type of the value being returned is unknown the `getValue()` method can be used.

## Using Aerospike Quick Look

Congratulations! You’ve now written your first program to use Aerospike. We inserted a record and retrieved the record, validating that the record was correctly inserted. It is possible to view and change the records in an Aerospike database without writing code too. One of the methods of doing that is to use Aerospike Quick Look (AQL). This uses a SQL-like dialect to manipulate the data.

From a terminal, AQL can be started in interactive mode simply by executing:

```
% aql
```



`aql` will try to connect on `localhost` (127.0.0.1) by default. You can connect to a different IP address by using the `-h` option. For example, my Docker container running Aerospike might be at 172.17.0.2 and you can connect to that using

```
% aql -h 172.17.0.2
```

This should give some basic information:

```
% aql
Seed:          127.0.0.1
User:          None
Config File:   /etc/aerospike/astools.conf /Users/book/.aerospike/astools.conf
Aerospike Query Client
Version 8.1.0
C Client Version 6.3.0
Copyright 2012-2022 Aerospike. All rights reserved.
aql>
```

From there we can query the data using the Aerospike SQL utility AQL:

```
aql> select * from test.item where pk =1
```

This should give the data for this specific primary key (PK):

```
+-----+-----+-----+-----+
| PK | name           | cost | discount |
+-----+-----+-----+-----+
| 1  | "Stylish Couch" | 50000 | 0.21     |
+-----+-----+-----+-----+
1 row in set (0.000 secs)
```

OK

We can also insert a new record or update the existing record. Both use the same semantics, as there is no difference between an INSERT and an UPDATE in Aerospike. To update a record, use:

```
aql> insert into test.item(PK, cost) values (1, 60000)
OK, 1 record affected.
```

```
aql> select * from test.item where pk =1
+-----+-----+-----+-----+
| PK | name           | cost | discount |
+-----+-----+-----+-----+
| 1  | "Stylish Couch" | 60000 | 0.21     |
+-----+-----+-----+-----+
1 row in set (0.001 secs)
```

OK

As you can see, the command merged the updated bin (cost) with the existing record, preserving the existing columns which were not updated.

Inserting a new record requires providing all the bins:

```
aql> insert into test.item(PK, name, cost, discount) values (2, "Blue Chair", 7399, 0.03)
```

All records can then be viewed:



```

aql> select * from test.item
+-----+-----+-----+-----+
| PK | name          | cost | discount |
+-----+-----+-----+-----+
| 2  | "Blue Chair"  | 7399 | 0.03     |
| 1  | "Stylish Couch" | 60000 | 0.21     |
+-----+-----+-----+-----+
2 rows in set (0.164 secs)

```

AQL is useful for basic data manipulation, but it is a long way from a full SQL implementation. Some of the most common statements used in AQL include:

Table 1-1. Common AQL commands

Command	Usage
help	Displays what commands AQL supports. Note that different versions of AQL may support different commands, using help will always show the supported commands.
INSERT INTO <ns>[.<set>] (PK, <bins>) VALUES (<key>, <values>)	Insert or update the specified bins in the specified record. Since all INSERTs are really UPSERTs, there is no specific UPDATE command. Note the use of PK to refer to the primary key. For example: Insert into test.testSet(PK, name, age) values (1, 'Tim', 312)
DELETE FROM <ns>[.<set>] WHERE PK = <key>	Delete a single record. For example: delete from test.testSet where PK = 1
SELECT <bins> FROM <ns>[.<set>]	Retrieves all the records in a set and displays them. For big sets, this normally times out as the timeout default is one second. For example Select * from test.testSet
SELECT <bins> FROM <ns>[.<set>] WHERE PK = <key>	Retrieves a single record from the Aerospike Database. For example: select name, age from test.testSet where PK = 1
SELECT <bins> FROM <ns>[.<set>] WHERE <bin> = <value>	Selects records which match a given criteria. Note that a secondary index must be defined on the bin in the where clause. (Secondary indexes will be discussed in Chapter 4) For example: select * from test.testSet where name = 'Tim'
SHOW NAMESPACEs	Shows details of all the namespaces in the database
SHOW SETs	Shows details of all the sets in the database
SET OUTPUT JSON RAW  TABLE	Set the format of the output. Defaults to "table"

## Policies

Almost all of the calls we have seen so far have required a policy. Policies dictate how a particular call should behave, such as what happens if there is an issue with the network, or what to do if the record already exists. We have very conveniently passed null to these so far so the defaults were used, but let's see why policies are important. For this discussion we're going to focus on the put command we did:

```
client.put(null, key, new Bin("name", "Stylish Couch"),
          new Bin("cost", 50000), new Bin("discount", 0.21));
```

The first parameter here, which is null in the above statement, is a `WritePolicy`, which subclasses the `Policy` class. The `Policy` class controls some general parameters which are useful on the majority of calls such as networking information. Some of the most important fields in this class are shown in Table 2-2.

Table 1-2. Some of the more commonly used fields on the `Policy` class

Field	Default	Description
<code>socketTimeout</code>	30000	When sending a transaction to the server, how long to wait in milliseconds before declaring a timeout has occurred. This may trigger retries based on the settings below.
<code>maxRetries</code>	2	If a timeout occurs when sending to the server, how many retries to attempt before giving up. This defaults to 2 for reads, 0 for writes and 5 for scan/queries
<code>sleepBetweenRetries</code>	0	If a transaction can be retried, how long to wait in milliseconds between the retries.
<code>totalTimeout</code>	1000	The total time in milliseconds to wait for the transaction and all retries before giving up. Note that if this parameter is exceeded a <code>Timeout</code> exception will be thrown to the application, even if the number of retries has not been exceeded.
<code>compress</code>	false	Whether to compress data between the client and the server. For large operations this may reduce network traffic at the cost of CPU.
<code>filterExp</code>	null	Whether to filter out operations on the server based on a filter expression. Expressions will be discussed in more detail later in the book.

These network settings, in particular, are very important for production applications. They dictate how the client layer should respond if the server is not responding in time. For example, if a server dies due to a hardware failure, there is a brief period where the client and the rest of the cluster do not know that the server is down and will still route traffic to it. This can cause timeouts on the client.

The `WritePolicy` subclass itself has some other useful properties for the application. The main ones are shown in Table 2-3.

Table 1-3. Some of the more commonly used fields on the WritePolicy class

Field	Default	Description
durableDelete	false	When a record is explicitly deleted, should a tombstone marker be placed in the record or should it be quickly expunged? In some edge cases, expunged records can reappear in the database. Durable deletes are an enterprise only feature at the moment, but are recommended for data whose integrity is paramount.
expiration	0	Records in Aerospike can be automatically removed by the system. This field is the number of seconds which records should be kept for. After this time the records will automatically be removed by Aerospike. If this field is -1, the records will live forever unless explicitly deleted. If this field is 0, the default time to live specified in the namespace will be used.
generationPolicy	0	These two fields are used together to allow applications to implement thread-safe read-modify-write cycles on the database using optimistic concurrency. We'll discuss this in more detail in later chapters.
recordExistsAction	UPDATE	When an operation changes a record, how should that change be merged with the existing record if there is one.

To illustrate how these policies can be used, consider the write in our simple program:

```
client.put(null, key, new Bin("name", "Stylish Couch"),
          new Bin("cost", 50000), new Bin("discount", 0.21));
```

Since the policy is null the defaults will be used, and the default recordExistsAction is UPDATE. This will update the record if it exists and create a new record if it doesn't. Let's say that the business requirements were to only create a record with this call; if the record already exists an exception should be thrown. This corresponds to a recordExistsAction of CREATE\_ONLY. To implement this, we would change the code as follows:

```
## Java
WritePolicy writePolicy = new WritePolicy(client.getWritePolicyDefault());
writePolicy.recordExistsAction = RecordExistsAction.CREATE_ONLY;
client.put(writePolicy, key, new Bin("name", "Stylish Couch"),
          new Bin("cost", 50000), new Bin("discount", 0.21));

## Python
writePolicy = {'exists': aerospike.POLICY_EXISTS_CREATE}
client.put(key, bins, policy=writePolicy)
```

In this case we need to create a new WritePolicy. If we were to just instantiate a new WritePolicy by doing

```
WritePolicy writePolicy = new WritePolicy();
```

it would work, but the parameters' values would be those hard-coded as defaults in the Aerospike Client driver. It is quite normal for applications to use a set of timeout fields that make sense for their SLAs in their environment. These settings should be the starting point for all calls, and calls can override these settings as the specific calls dictate.

In order to do this, the `ClientPolicy` has a set of default parameters, one for each of the different policy types. Creating a new `WritePolicy` based off of the `writePolicy Default` will create a separate policy instance we can use for this specific call without affecting concurrent write calls in other threads while at the same time preserving most of the default settings, with only one change..

The code then sets the `recordExistsAction` to `CREATE_ONLY`. The Aerospike server will check to see if a record with that key already exists and it will throw an exception if it does. If no such record exists, the new record will be inserted into the database. Note that this is done atomically – there is no possibility of another thread creating the record between when the server checks for the existence of the record and when it inserts the record into the database.

## Summary

In this chapter we installed Aerospike and wrote our first program. We examined how to read and write records as well as the AQL tool to inspect and change the data. Finally, we learned about Policies and how these can affect the operations. In the next chapter we will dive more into the features of Aerospike from the client's perspective.

---

# Basic Operations

## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

In the previous chapter, we illustrated what an introductory demo application might look like. We briefly discussed establishing connections, retrieving data, and inserting data. In this chapter, we will dive into all the basic operations you’ll use when putting Aerospike to work, such as creating, reading, updating, and deleting. These operations are the heart of the day-to-day usage. You’ll use what you learn in this chapter every day you use Aerospike.

We will also discuss time-to-live (*TTL*), which helps with data life cycle management, and work more with *WritePolicy* and *Policy*, and explore various data types and how to use them. In particular, you’ll see how the Aerospike client interface handles the data types for you, in many cases, making data movement between systems easier.

# CRUD Operations

Create, Read, Update, Delete (CRUD) are the four fundamental operations of interacting with any database, including Aerospike. The CRUD operations are the fundamentals needed to get you started.

Before discussing the basic operations in depth, we should talk about the data types that can be used in Aerospike natively and the data lifecycle options available.

## Data Types

A record in Aerospike can contain one or more bins. When created, Aerospike bins are of certain data types. It is important to know the various data types so you can make informed decisions about the limitations and data size costs.

The Aerospike client driver's built-in conversion allows you to create these data types independent of the language that created the data or input it. In other words, the data types in Aerospike are language agnostic. Aerospike's native language-agnostic data types also make it easier to apply new use cases to the data later.

Table 2-1. Data types

Data Type	Description
Integer	64-bit numerical values. Can be signed or unsigned. Used for whole numbers positive and negative. Ex. 42, -12
Boolean	Holds a value of true or false.
Double	Used for storing fractional values or numbers with a decimal point. GPS coordinates, pricing, weather data such as temperatures, and weight are good examples. Ex. 37.4212,-122.0988.
String	Stores characters in an opaque byte array. Any combination of spaces, letters and numbers could be represented as a string. Ex. "Aerospike is fast," "Tim," "Pineapple42 themed backpack."
Map	A HashMap or Dictionary like data structure. Stores a relationship between a key and a value. Keys within a Map are unique, with duplicates not allowed, so all values associated with that key are in one place. Keys in the map must be a scalar data type and values can be scalar or collection types. Nesting is allowed. Ex. <pre>myMapBin: { "Aerospike": "Fast" "Product1234": "OReilly canned beans", 23: True, 0.42: [1,2,2,3] }</pre>
List	An array type data structure, similar to map in that it is a collection that can be operated on. This is distinct from the Map type in that List does not have a key to value relationship, and there can be duplicates of the data. Entries inside a list can either be scalar data types or collection data types as nesting is allowed. Some examples: a list of strings, a list of dictionaries, or even a list of lists or list of maps. Ex. <pre>pageViews: [ 1691950585, 1691953456, 1691958589, ]</pre>
Blob	Binary data, raw bytes. This data might be chosen to be stored as raw bytes due to a language-specific structure. For example, if you stored a compressed serialized dictionary, bloom filter, or in-house data type.

Data Type	Description
GeoJSON	Geometry objects, for geospatial matches. Ex. find if a device is within 42 miles of a specific store.
HyperLogLog	Probabilistic data type, used to count the members of a set or unions of a set. Ex. count of users who have an interest in "Computers" and "Furniture"

## Data Lifecycle

If your data has a limited useful lifetime and will eventually need to be deleted and cleaned up you can set a *time-to-live (TTL)* on a record by setting an *expiration* value in the *WritePolicy* while performing a write operation.

The TTL is the number of seconds until the server expires the record, deleting it automatically, and is updated when a write occurs on that record which refreshes its lifetime. A server configuration parameter called *nsup-period* controls how often Aerospike cleans up records that should be expired. The default value of this is 0 which disallows setting TTL on records. If you wish to set a positive *WritePolicy* expiration value, you'll need to change the *nsup-period* to a non-zero value as well. A good starting value for *nsup-period* is 120 seconds if you are unsure of where to start but know you need a TTL.

Setting a TTL on your records is optional and can be set per write operation, per key. There are also special flags you can use to prevent the TTL from being extended during a write. You can reference Table 3-2 for a summary of the Aerospike client driver *WritePolicy* expiration values and their effects.

Table 2-2. *WritePolicy* Expiration Value

<i>WritePolicy</i> Expiration Value	Effect
-2	Do not change the TTL while writing
-1	Never expire.
0	Use the server configuration 'default-ttl' for the namespace.
Values greater than 0	TTL in seconds, how long the record should exist in Aerospike.

Some data sets and applications may prefer this to be controlled by an external logic, via some batch process or coordinated cleanup process, for example. If you need full control over how data gets deleted, or do not wish for it to be automatically cleaned up you may set a *WritePolicy* Expiration value of -1. The reason for choosing to disable expiration, by having the default server configuration *nsup-period=0*, is to ensure that automatic cleanup, deletion, of data is an opt-in experience. No data will be deleted unless you specifically define it to be.



Setting the WritePolicy Expiration Value in the client to 0 means that data will expire in the amount of seconds defined by the *default-ttl* parameter on the server. However, setting the *default-ttl* on the server to 0 means “never expire.” These are similarly named but distinct configurations. These are separately documented in the Server Configuration reference for the *default-ttl* parameter, and in the Client API documentation for the WritePolicy .

## Create

Creating a record in Aerospike can be done with the *Put* client method. You will need an AerospikeClient object to use the *Put* method as described in Chapter 2. The AerospikeClient object is available in a number of different languages.

```
## Java
client.put( null, <Key>, <Bins>)
or
client.put( <Write Policy>, <Key>, <Bins>)
## Python
client.put( <Key>, <Bins>)
or
client.put( <Key>, <Bins>, policy=<WritePolicy>)
```

The *WritePolicy* controls much of how a write works, such as defining the data expiration value, timeout, and what happens if the record already exists. Reference your client API documentation for the full list of options.

## Data Types

When writing a record, we may want to store it using a specific data type. Using the Aerospike client driver simplifies this process. In Java and Python, Aerospike assigns the data type to the bin that matches the source and inserts the data. If the bin is assigned a *dictionary* data type, such as HashMap in Java or Dictionary in Python, the client will write that as an Aerospike Map data type. Similarly, Java or Python lists will be assigned an Aerospike list data type.

Let’s walk through an example of writing a Map data type in Aerospike, and then printing it back out. First, we define a hashmap in Java, or a dictionary in Python, and put data in it.

Then we create an Aerospike bin with no data type specified, and write the record to Aerospike. The Aerospike client driver’s built-in conversion automatically matches the data type, which allows for storage of these more complex data independent of the language that created the data or input it, aka language agnostic.

Next, we retrieve the data from the Aerospike bin, and store it back in the source database. Printing displays the data, showing that it is unchanged by the round trip.



```

## Java
// Define an example HashMap
Map<String, Object> mymap = new HashMap<>();
mymap.put("key1", "value1");
mymap.put("key2", 123);
mymap.put("key3", true);

// Convert the Map into Aerospike Bin
Bin mybin = new Bin("amap", mymap);

// Write the record to Aerospike
client.put(null, key, mybin);

// Read the record back from the database
Record record = client.get(null, key);

// Store as a local Map
Map retrievedMap = record.getMap("amap");

// Print the value of the bin
System.out.println(retrievedMap);

// Produces an output of:
// {key1=value1, key2=123, key3=true}

## Python
# Define an example dictionary
mymap = {
    "key1": "value1",
    "key2": 123,
    "key3": True
}

# Define the bins to write
record = {
    "amap": mymap
}

# Write the record to Aerospike
client.put(key, record)

# Read the record back from the database
(key_, metadata, bins) = client.get(key)

# Print
print(bins)

# Produces an output of:
# {'key1': 'value1', 'key2': 123, 'key3': True}

```

In later chapters, we will also discuss sending operations, expressions, and commands that help you interact with these data types without downloading them into the application for manipulation and inspection.

## Read

Reading a record can be done using the *Get* client method. Then we can store the result into variables to manipulate within your application. The *Policy* (*WritePolicy* is a subclass of *Policy*, related specifically to how data is written.) is optional and can be defined to control things like timeout value, retries, and consistency level. We can also specify which bins we want to be retrieved which will save us resources if only a portion of the record is needed.

Java:

```
Record record = client.get( <Policy>, <Key>, <Bins to fetch>);
```

Or to read the entire record:

```
Record record = client.get( <Policy>, <Key>);
```

Python:

```
(<Key returned by server>, <Metadata>, <Bins>) = client.get(<Key>, (<bins to fetch>, policy=<Policy>)
```

Read entire record:

```
(<Key returned by server>, <Metadata>, <Bins>) = client.get(<Key>, policy=<Policy>)
```

## Metadata Uses

When reading a record from Aerospike we receive metadata from the server about the record. This metadata shows us the *generation* of the record and the *expiration*. The *generation* is a counter on the server, and the *expiration* shows the current TTL of the record that was read.

The generation can be used for conflict resolution and a *generation sensitive* write pattern is usually used for check-and-set (CAS) writes - a method to avoid having two processes modify the same data in a multi-threaded application. The generation counter represents the number of times a record was modified, even if it was only to modify the TTL. This can be useful for ensuring a record has not been modified by another thread since it was last read.



There is an upper-bound to the generation counter which when reached, wraps around by coming back to 1.

The CAS pattern takes advantage of the metadata returned by a read, and the “generation policy” setting of the WritePolicy. Here is an example where we would fail to write, and throw an exception, if the generation (version) of the record had changed since we read it.

When we read the record with a simple get statement, we set an integer variable equal to the record’s currentGeneration property value. Then, we modify the data. Then we set the WritePolicy to only write data if the current generation is equal to what it was the last time the data was read. We set the current generation to the value in the variable. Then write the data.

```
## Java
// Read the record to be updated
Record record = client.get(null, key);

// Get the current generation of the record
int currentGeneration = record.generation;

// Modify the data based on your requirements
// ...

// Perform the write operation with an expected generation
WritePolicy writePolicy = new WritePolicy();
writePolicy.generationPolicy = GenerationPolicy.EXPECT_GEN_EQUAL;
writePolicy.generation = currentGeneration; // Set the expected generation

// Update the record with the modified data
client.put(writePolicy, key, updatedBins);
## Python
# Read the record to be updated
(_, metadata, record) = client.get(key)

# Get the current generation of the record
current_generation = metadata.generation

# Modify the data based on your requirements
# ...

# Perform the write operation with an expected generation
write_policy = {'gen': aerospike.POLICY_GEN_EQ}
write_policy['gen_value'] = current_generation # Set the expected generation

# Update the record with the modified data
client.put(key, updated_bins, policy=write_policy)
```

If the data was modified by another thread in between when we read it and when we went to write the modified data back, it will fail and throw an error. This keeps two different threads from modifying the same data unintentionally.

## Update

Updates are done using the same method as write operations, by using *Put*. When you perform a put against an existing record, the default behavior is to perform an *upsert*. This means that Aerospike creates the record if it does not exist, but if it does exist, Aerospike will keep any existing data already on the record aside from the bin we are operating on. If we have an existing record with some bins we aren't modifying, and use the default update policy, those bins remain unchanged on the record.

This behavior is controlled via the *WritePolicy.RecordExistsAction* specified when a put is called in Java. There is a similar selection in the Python client, but not under a specific enum, which can be referenced in Table 3-3 or in the [API documentation](#) under “Existence Policy Options” under the “aerospike” section. Though the specific implementations vary, this behavior modifier is present in all client drivers.

Table 2-3. *WritePolicy.RecordExistsAction*

Java	Python	Behavior
RecordExistsAction.CREATE_ONLY	aerospike.POLICY_EXISTS_CREATE	Create the record only if it does not exist. Throw an exception if it already exists.
RecordExistsAction.REPLACE	aerospike.POLICY_EXISTS_CREATE_OR_REPLACE	Replace a record completely if it exists, otherwise create it.
RecordExistsAction.UPDATE	aerospike.POLICY_EXISTS_IGNORE	Update a record if it exists, otherwise create it. This is the default behavior of all client drivers.
RecordExistsAction.REPLACE_ONLY	aerospike.POLICY_EXISTS_REPLACE	Only replace a record completely if it exists. Throw an exception if the record does not exist.
RecordExistsAction.UPDATE_ONLY	aerospike.POLICY_EXISTS_UPDATE	Only update a record if it exists. Throw an exception if it does not.

To illustrate how to use these policies and what their exceptions might look like, let's go through an example using `CREATE_ONLY`.

First, we set the policy to only create new records, and throw an error if the record already exists. In Java, this is the “`recordExistsAction.CREATE_ONLY`” setting. In

Python, it's "aerospike.POLICY\_EXISTS\_CREATE." Then, write the data. Assuming it doesn't already exist, it should simply write the data without error.

```
## Java
// Create the WritePolicy and set it to fail if the record already exists. Create
only.
WritePolicy wp = new WritePolicy();
wp.recordExistsAction = RecordExistsAction.CREATE_ONLY;

// Write the data
client.put(wp, myKey, myBin);
## Python
# Create the WritePolicy and set it to fail if the record already exists. Create
only.
wp = {
'exists': aerospike.POLICY_EXISTS_CREATE
}

# Write the data using the WritePolicy
client.put(my_key, my_bin, policy=wp)
```

If we ran this example program twice with the same key, the record would fail to be written on the second time as it already exists. It would also throw an exception. The exception will look like this:

*Exception in thread "main" com.aerospike.client.AerospikeException: Error 5,1,0,30000,1000,0,BB9020011AC4202 127.0.0.1 3000: Key already exists*



As we can see from the WritePolicy RecordExistsAction table, there is an option called Replace. This will delete any existing data on a particular key that is not being rewritten by the client. If we had 4 bins on a record and then wrote 3 bins to the record using the Replace policy, that fourth bin would be deleted since we are "Replacing" the entire record. It may help to think of replace as "Replace the entire record with this write".

## Delete

The main method to delete an entire record is using the *delete* (Java) or *remove* (Python) client methods.

```
## Java
client.delete(null, myKey);

## Python
client.remove(my_key);
```

There is an optional `writePolicy` which is used for *Durable Deletes*, an Enterprise feature which is a way to *tombstone* deletes. A tombstone marks that a record has been deleted that once existed with this key, so there is no way the records could be accidentally recovered in some event like a power outage.

## Lightweight Operations

While it is possible to always write some data to a bin to refresh the record's TTL, we can do better. If we only need to refresh a record's TTL, we do not need to rewrite the entire record. Similarly if we want to check if a record exists we do not need to fetch the entire record. These functionalities can be covered in a lighter way using the `touch` and `exists` methods the client driver provides.

The `touch` operation is used as a lightweight way of updating a record's metadata such as the *TTL or Generation*. This is primarily used to refresh a record at Read time to ensure it remains live. The `touch` operation only sends metadata and `WritePolicy` to the server making it a very lightweight operation for the client.

In this example, we extend the TTL for a day (86400 seconds):

```
## Java
WritePolicy writePolicy = new WritePolicy();
writePolicy.expiration = 86400;
client.touch(writePolicy, myKey);
## Python
client.touch(my_key, 86400)
```

The `exists` function is similar in that it only transfers metadata across the network to the client. This is used to get a boolean True or False signal on the client application to know if the record exists.

```
client.exists(null, myKey); //Java
client.exists(my_key) # Python
```

## Batch

If there is a need to read or write many records at once, the *Batch* type operations may be most suitable. To perform a batch read operation, we need to create a list or array of keys we want to fetch and then we can pass that into the `client.get` method.

In this example, we do exactly that, first creating an array of the keys that we wish to read, then reading them all at once. Note that in Python, this is a different command: `get_many`, rather than the simple `get`.

```
## Java
Key[] keys = new Key[] {
    new Key("test", "testset", "key1"),
    new Key("test", "testset", "key2"),
    new Key("test", "testset", "key3")
}
```

```
};

Records[] records = client.get(null, keys);
## Python
keys = [
('test', 'testset', 'key1'),
('test', 'testset', 'key2'),
('test', 'testset', 'key3')
]

records = client.get_many(keys)
```

When we execute this, any of the records retrieved can be located positionally in the *Records* array. That means for “key1” in this example, we would expect *Records[0]* to contain the data for it.

In Java, if the record is not found, then position 0 would not have any data while in Python, we would see a “None” entry in the list. After we execute this and find records, we can interact with the records the same way we would while doing a single record read operation.

There are also ways to batch write operations and delete operations which we’ll cover more in-depth in chapter 4.

## Wrapping Up

In this chapter, we learned how to perform all our *CRUD* operations to read, write, update, and delete. We explored how to work with the various data types, including how to create most of them, with the exception of the GeoJSON and HyperLogLog data types. These data types will require a more in-depth exploration than this beginning chapter addresses.

We learned about the behavior of write operations and how to control the behavior depending on if the record exists, as well as how to set and update the lifetime and the generation of a record. In the next chapter, we will look deeper at the more advanced operations and techniques you can use to interact with Aerospike.





---

# Advanced Operations

## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

So far you have looked at Aerospike’s basic operations, primarily reading and writing the data. You have seen that data is stored in bins and that multiple bins can be read or written in a single call. What you’re going to look at now is some more advanced operations.

First, you’ll look at a few of the many capabilities of the `Operate()` command. In particular, you’ll see how to add, append, and update data in some of the more complex data types such as lists and maps. Then, you’ll learn some of the basics of how to use expressions to create conditional writes and derived reads among other interesting data operations. You’ll be introduced to batch operations, secondary indices, multiple predicate queries and a host of other aspects of using Aerospike.

## The `Operate()` command

One of the most powerful commands on a single record is the `operate()` command. In fact, it is so powerful that all other single record commands are just syntactic sugar

that wrapper the `operate()` command. If you have been following along with the examples in this book so far, you have actually used this command, albeit wrapped in a `get()` or `put()` method.

`Operate()` allows you to perform an arbitrary complex list of operations on a single record, taking a list of `Operation` classes containing the appropriate actions. So, in a single call for example, you could insert one bin, add ten to the contents of another bin, work out the size of a map in a third bin and so on. The basic syntax for the command is:

```
public Record operate(WritePolicy policy, Key key, Operation ... operations);
```

If you're not familiar with Java the `"..."` is the *varargs* operator, allowing an arbitrary number of parameters to be passed. So one argument could be passed or a hundred, it's up to you.

Let's say you're building a shopping cart application for illustrative purposes. The record for a user contains the items in the cart, a count of the items, and the total cost of the items. For this first example, you will assume that the items in the cart are simply stored in a string.

So, you could create a record similar to:

```
Key key = new Key("test", "cart", 1);
client.put(null, key,
    new Bin("items", "shoes,"),
    new Bin("totalItems", 1),
    new Bin("cost", 59.25));
```

As you saw earlier, this would create a key which references item 1 in the "cart" set (table) in the "test" namespace. A record would be written to the key which contains three bins with the values specified. This would yield a record similar to:

```
aql> select * from test.cart;
+-----+-----+-----+
| items  | totalItems | cost |
+-----+-----+-----+
| "shoes," | 1          | 59.25 |
+-----+-----+-----+
1 row in set (0.279 secs)
```

How would you go about updating this record? Let's say you want to add another item like jeans. You could code this into a method similar to:

```
public static void addItem(Key key, String itemDescr, double cost) {
    client.operate(null, key,
        Operation.append(new Bin("items", itemDescr+",")),
        Operation.add(new Bin("totalItems", 1)),
        Operation.add(new Bin("cost", cost))
    );
}
```

```
    );  
}
```

This method does one call to the Aerospike client and one transaction on the database. This transaction contains three distinct operations:

1. It appends a string onto an existing string, in this case the item description. If the “items” bin did not exist Aerospike would automatically create that bin.
2. It adds one to the “itemCount” bin. Again, if the bin doesn’t exist it will be created with a default value of zero then have the passed value of one added.
3. It adds the cost of the items to the running cost.

This method could be called to add two new items to the cart:

```
addItem(key, "jeans", 29.95);  
addItem(key, "shirt", 19.95);
```

This would give you a database entry which looked like:

```
aql> select * from test.cart;  
+-----+-----+-----+  
| items          | totalItems | cost  |  
+-----+-----+-----+  
| "shoes,jeans,shirt," | 3          | 109.15 |  
+-----+-----+-----+  
1 row in set (0.187 secs)
```

## Simplifying the program

Your program created an initial record with hard-coded bins. However, since Aerospike will automatically create bins that are missing, this step is actually not necessary. You can simplify your program by just using the `addItem` method. If the record already has items in it, the new item will be appended. If there are no items in the record or the record does not exist, Aerospike will create the record for you, automatically creating the `items`, `totalItems` and `cost` bins.

So your program could become:

```
client = new AerospikeClient("172.17.0.2", 3000);  
Key key = new Key("test", "cart", 1);  
client.delete(null, key);  
addItem(key, "shoes", 59.25);  
addItem(key, "jeans", 29.95);  
addItem(key, "shirt", 19.95);
```

Note the addition of the line

```
client.delete(null, key);
```

This line simply ensures the record does not exist before running the program. If you didn't do this and ran the program multiple times, you would keep adding the same items to the cart.

Let's double check this through AQL after this change:

```
aql> select * from test.cart;
+-----+-----+-----+
| items          | totalItems | cost  |
+-----+-----+-----+
| "shoes,jeans,shirt," | 3          | 109.15 |
+-----+-----+-----+
1 row in set (0.187 secs)
```

## The Operation class

In the previous example, you saw that the arguments describing what work Aerospike should do on the record were on the `Operation` class. Let's take a look at the common methods on this class, as shown in Table 4-1.

Table 3-1. : Common operations on the Operation class

Method	Use
<code>add(Bin)</code>	Increments the bin by the amount in the value of the passed Bin. If the bin does not exist, it is created with the passed value. If either the value is non-numeric or the existing value in the bin on the database is non-numeric, an <code>AerospikeException</code> is thrown with a <code>resultCode</code> of <code>Result Code.BIN_TYPE_ERROR</code> .
<code>append(Bin)</code>	Adds the passed string value to the end of the string in the passed Bin. If the bin does not exist, it is created with the passed value. If either the value is not a String or the existing value in the bin on the database is not a String, an <code>AerospikeException</code> is thrown with a <code>resultCode</code> of <code>Result Code.BIN_TYPE_ERROR</code> .
<code>get()</code>	Returns the contents of all the bins in the record
<code>get(String)</code>	Returns the contents of the named bin. Returns nothing if the named bin does not exist.
<code>put(Bin)</code>	Sets the content of the bin in the database to the passed value.

## Return value of Operate()

Notice that some of these methods (the `get` methods) return a value. So how do you get access to these values? Recall that the actual signature of `operate()` is

```
public Record operate(WritePolicy policy, Key key, Operation ... operations);
```

The returned `Record` contains the values specified to be read. So let's say you want to change our `addItem` method to return the current cost of the items in the cart. You could simply add a `get("cost")` operation to the list of operations being executed:

```
public static double addItem(Key key, String itemDescr, double cost) {
    Record record = client.operate(null, key,
```

```

        Operation.append(new Bin("items", itemDescr+"")),
        Operation.add(new Bin("totalItems", 1)),
        Operation.add(new Bin("cost", cost)),
        Operation.get("cost")
    );
    return record.getDouble("cost");
}

```

Notice that this example changed the return type of the method to `double`, added the extra `Operation` at the end of the list and returned the value of the bin “cost” by invoking `getDouble("cost")` on the returned record.

## Order of Operations

The operations on a single record are applied to the record in the order they are specified to the `operate()` call. Aerospike guarantees that the operations are applied atomically and transactionally, so either all the operations happen or none of them, and no other thread can affect the record between the first operation and the last.

This order of operations can sometimes be important. For example, suppose you wanted Aerospike to return the cost of the items in the cart before you inserted the item and the cost after the item was inserted. You can change the program to do this easily enough:

```

Record record = client.operate(null, key,
    Operation.get("cost"),
    Operation.append(new Bin("items", itemDescr+"")),
    Operation.add(new Bin("totalItems", 1)),
    Operation.add(new Bin("cost", cost)),
    Operation.get("cost")
);

```

Here you’re getting the value of the “cost” bin before you update it, and again after you update it. This is valid in Aerospike, but what does it return? Thankfully the `Record` object has a useful `toString()` method, so you can inspect the returned values by adding the following straight after the `Record` is returned:

```
System.out.println(record);
```

When you do this you can see what is returned on subsequent calls:

```

(gen:1),(exp:0),(bins:(cost:59.25))
(gen:2),(exp:0),(bins:(cost:[59.25, 89.2]))
(gen:3),(exp:0),(bins:(cost:[89.2, 109.15]))

```

The first value returned is the generation count, that is how many times has this record been modified. Then the expiry of the record if a time-to-live had been set as detailed in the previous chapter. Finally the bins are returned and you can see the

cost bin is returned as a number on the first call, and as a list with two values on the two subsequent calls.

This makes sense when you think about it. On the first call, there was no “cost” bin, so Aerospike had nothing to return here. The `get("cost")` operation at the end of the list did have something to return, so Aerospike determined there was only one value to return for the cost bin, so it was returned as a single value. However, on subsequent calls, there was a valid value in the cost bin for both the `get("cost")` call at the start of the operation list and the one at the end of the list, so Aerospike has to return both values. It does this by wrapping both values in a list.

## ListOperation and MapOperation:

The operations you’ve seen so far have been fairly simple. A lot of the power of operations however comes from their ability to act on Aerospike’s Container Data Types (CDTs) – lists and maps. These were briefly introduced in Chapter 2, but let’s look at them in more detail.

### Lists

A list is an ordered collection of elements. Like all CDTs in Aerospike, the elements can be of any supported type – including other lists and maps – and there is no requirement for the elements to be of a homogeneous type. So a list might contain for example:

```
["sofa", 255.99, 2, ["black", "red", "white"]]
```

In this case the list contains a String (“sofa”), a double (255.99), a long (2) and a list.

Lists can be accessed in their entirety, or by index, value or rank. To understand these concepts, let’s look at an example. Consider the following list:

```
[1, 4, 7, 3, 9, 26, 11]
```

The index is the position of an element in the list starting with zero. The value is the value of the element identified by a specific index, and the rank is the index of an element in the list assuming the values are sorted. Table 4-2 shows the properties for this list.

*Table 3-2. List properties for a sample list*

	1	4	7	3	9	26	11
Value	1	4	7	3	9	26	11
Index	0 or -7	1 or -6	2 or -5	3 or -4	4 or -3	5 or -2	6 or -1
Rank	0 or -7	2 or -5	3 or -4	1 or -6	4 or -3	6 or -1	5 or -2

Note that both indexes and ranks can be specified with a negative value, with a negative meaning “start at the end of the list and work backwards”. They both also start with zero being the first item.

The value and index are fairly self explanatory, but let’s look at how you arrived at a rank of 3 (or -4) for the value of “7”. Remember that the rank is the value order after the list is sorted. So, sorting the list gives:

```
[1, 3, 4, 7, 9, 11, 26]
```

In this sorted list, the index (i.e., position in the list) of “7” is three, or coming from the end of the list, -4. If you’re confused as to why the “7” is position 3, remember that the first item in the list (1) is zero, so “3” is index one, “4” is index 2 and “7” is index three.

Let’s take a look at a simple program to show how we would query these:

```
public class ListMapExamples {
    private static final String LIST_BIN = "list";
    private static IAerospikeClient client
        = new AerospikeClient("localhost", 3000);
    private static Key key = new Key("test", "sample", 1);

    private static void show(Operation operation, String description) {
        Record record = client.operate(null, key, operation);
        System.out.println(description + ": " + record.getValue(LIST_BIN));
    }

    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 4, 7, 3, 9, 26, 11);
        client.put(null, key, new Bin(LIST_BIN, list));

        show(ListOperation.getByIndex(LIST_BIN, 1, ListReturnType.VALUE),
            "Index of 1");
        show(ListOperation.getByRank(LIST_BIN, 1, ListReturnType.VALUE),
            "Rank of 1");
        show(ListOperation.getByValue(LIST_BIN, Value.get(1),
            ListReturnType.VALUE), "Value of 1");
        client.close();
    }
}
```

The output of this program reflects what was mentioned above for the index, rank and value:

```
Index of 1: 4
Rank of 1: 3
Value of 1: [1]
```

Note that

Lists can be defined as ORDERED or UNORDERED. An UNORDERED list preserves the order of elements as they were inserted, and is the default. An ORDERED list however will have the elements maintained in sorted order by Aerospike, irrespective of the order they were inserted. Since the lists are maintained in sorted order, an ORDERED list has the rank being the same as the index.

## Maps

A map is a collection of items, each of which has a key and a value. Most programming languages have some variant of this, be it called a Map, a Dictionary, or so on. Both keys and elements can be of any support type inside the Aerospike database.



It is necessary to ensure the client programming language supports the format of the map being used. For example, Node.js does not support integers as map keys, so reading such a map into the Node.js client will give errors, whereas it will work fine in other languages like Java.

Maps can be accessed by key, value, index or rank. Again, this is best understood with an example. Consider the following map:

```
{a:1, b:2, c:30, y:30, z:26}
```

For this map, the properties in Table 4-3 apply.

*Table 3-3. Map properties for a sample map*

	a:1	b:2	c:30	y:30	z:26
Key	a	b	c	y	z
Value	1	2	30	30	26
Index	0 or -5	1 or -4	2 or -3	3 or -2	4 or -1
Rank	0 or -5	1 or -4	3 or -2	4 or -1	2 or -3

As can be seen in table 4-3:

The *Key* is the identifier of the element in the map

The *Value* is the value of the element identified by a specific map key

The *Index* is the key order of the value in the map

The *Rank* is the value order of the element in the map. If multiple elements have the same value (such as c:30 and y:30 in figure 4-3), their rank is based on the index order.



Maps can be `UNORDERED`, `KEY_ORDERED` or `KEY_VALUE_ORDERED`. `KEY_ORDERED` maps are the most useful, and store the maps sorted by the key. This makes operation on the keys very efficient.

## Operations

Now that you understand the basic features of an Aerospike List and Map, let's look at how you can use these. Both lists and maps are very powerful, and as you will see in Chapter 6, form the basis of many of the data modeling techniques you will use.

Similar to the `Operation` class, there are also `ListOperation` and `MapOperation` classes which contain the respective operations. There are also other operation classes such as `HllOperation` which perform HyperLogLog operations, but that is beyond the scope of this book.

## ListOperations and MapOperations Example

The shopping cart example you saw earlier in this chapter was good, but not great. The items were just a concatenated string which makes manipulating it difficult and it's not very flexible. For example, maybe you want to store a reference to the original item so if the cost changes when an item is in the cart it can be detected and a warning shown to the user.

To do this, you will turn each item in the example into a Map, and store these Maps into a List of items in the shopping cart. You will then add some operations to the cart example.

First, let's define a method that will take the information for our item and turn it into a map:

```
public Map<String, Object> createItem(String itemDescr,
    double cost, String originalItem) {
    Map<String, Object> result = new HashMap<>();
    result.put("cost", cost);
    result.put("descr", itemDescr);
    result.put("orig", originalItem);
    return result;
}
```

Then, to insert an item, you can use a simple list operation:

```
public static void addItem(IAerospikeClient client, Key key,
    Map<String, Object> item) {
    client.operate(null, key, ListOperation.append("items", Value.get(item)));
}
```

In this method, you call `client.operate` again, this time passing in a `ListOperation` which simply appends the item to the end of the list. You will notice that the item

is wrapped in a `Value.get(...)` call. This is a common pattern you will find in Aerospike, especially around `ListOperations` and `MapOperations`. Wrapping `List` and `Map` operations allows Aerospike to specify a set of supported types that it understands without needing a large set of method overloads. Taking `Object` as a parameter without the wrapper would be too generic. The wrapper prevents compile-time checking of the passed values, and the `Value` wrappers also help the Aerospike client with serialization to send the information to the server.

Now you can insert a few items into the shopping cart list like this:

```
addItem(client, key, createItem("shoes", 59.25, "/items/item1234"));
addItem(client, key, createItem("jeans", 29.95, "/items/item2378"));
addItem(client, key, createItem("shirt", 19.95, "/items/item88293"));
```

Inspecting the output in AQL shows you the new nested structures.

```
aql> select * from test.cart;
+-----+
| items
+-----+
| LIST(['{"cost":59.25, "descr":"shoes", "orig":"/items/item1234"}, {"cost"
+-----+
1 row in set (0.510 secs)
```

Note that the actual output is far longer than shown here and has been truncated for ease of reading. However, this does present a common problem. As records become larger and more complex, examining them becomes increasingly difficult. Luckily, AQL supports various display modes. The one that makes the most sense here is JSON.

```
aql> set output json
OUTPUT = JSON
aql> select * from test.cart;
[
  [
    {
      "items": [
        {
          "cost": 59.25,
          "descr": "shoes",
          "orig": "/items/item1234"
        },
        {
          "cost": 29.949999999999999,
          "descr": "jeans",
          "orig": "/items/item2378"
        },
        {
          "cost": 19.949999999999999,
```

```

        "descr": "shirt",
        "orig": "/items/item88293"
    }
  ]
},
[
  {
    "Status": 0
  }
]
]

```

You can see that our cart now comprises a list which contains 3 elements, each of which is a map.

Naturally, you can also combine multiple operations into a single API call. For example, let's say you want to remove the second item in the list and return the number of items remaining in the list. You could do this by:

```

Record record = client.operate(null, key,
    ListOperation.removeByIndex("items", index, ListReturnType.NONE),
    ListOperation.size("items"));
Int remaining = record.getInt("items");

```

In our case, this will result in `remaining` having a value of 2. As before, the operations are applied in order, so the count is taken after the item has been removed.

Notice when you invoked the `ListOperation.removeByIndex` call, you passed a parameter of `ListReturnType.NONE`? This determines what Aerospike returns from this call. `NONE` obviously specifies no return value, but there are other things you could return. For example, if you passed `ListReturnType.VALUE` here, the removed item would be returned and `ListReturnType.COUNT` would return the number of elements removed.

There are other API calls that look similar. For example, to remove two items from the list starting from the passed index you can use `removeByIndexRange`:

```

Record record = client.operate(null, key,
    ListOperation.removeByIndex("items", index, 2 ListReturnType.NONE));
int removed = record.getInt("items");

```

## Inverted Flag

All the return values for the `ListReturnType` flag, and with the corresponding `Map ReturnType` flag, are discrete - only one can be specified - with the exception of `ListReturnType.INVERTED`. This flag can be specified with any of the other return types and effectively inverts the meaning of the list command.

If you were to add this flag to the above operation, our code would become:

```
Record record = client.operate(null, key,
    ListOperation.removeByIndex("items", index, 2,
        ListReturnType.NONE | ListReturnType.INVERTED));
int removed = record.getInt("items");
```

The presence of the INVERTED flag tells Aerospike to remove all items in the list *except* the item at the passed index and the item after it. So if you started with five elements in the list or fifty, you would still end up with a list containing just two elements.

## Contexts

List and Map operations can be nested to any arbitrary depth. In our cart example, you have maps inside a list, so a nesting depth of two. This presents a problem when using list and map operations – how to affect an item which is not the top level?

The answer is that all of these APIs take a *context* parameter. This context describes the path from the top level down to the specific list or map you wish to affect. In Java this is a vararg parameter to the List or Map operation, and as such, goes at the end of each call. For example, let's say you wanted to set the price of our “shoes” item to be \$49 instead of \$59.25. You know that this is the first item in the list, so you would use this MapOperation:

```
client.operate(null, key,
    MapOperation.put(MapPolicy.Default, "items", Value.get("cost"),
        Value.get(49), CTX.listIndex(0)));
```

Let's look at what this call is doing. The first parameter to the MapOperation is the MapPolicy. In this case you're just using Default, which tells Aerospike to use an UNORDERED map. If you preferred to use a KEY\_ORDERED map, you could have specified this here. The next parameter is the bin name (“items”), which in a production system would obviously be extracted to a constant. Following this is the map key you want to set (“cost”), then the value you want to set it to (49), both wrapped in a Value.get(...) call. Finally comes the context (CTX.listIndex(0)) which says to Aerospike “the map on which I want this value set can be found as the first element in the list.”

Note that the type of the item you want to affect is a *map*, so you have to use a MapOperation even though the map is contained within a list. The path down to that map from the top (bin-level) is shown through the context. This probably would make more sense if the context were the first parameter rather than the last, but sadly in Java varargs must be the last parameter.

TODO: Common list and map operations.

# Expressions

One very powerful feature of Aerospike is the ability to use Expressions. An expression can be used to filter whether to perform a particular operation on a record, compute values on the fly, write derived quantities to a record and so on. They are very powerful and have many uses. Let's take a look at some of these.

## Filter Expressions

Filter expressions allow operations to be performed on the server only if the particular filter expression returns true. So, for example, let's say you want to update the state of an order to be COMPLETED, but only if the current state is PROCESSING. You could achieve this by reading the record, ensuring that the current state is PROCESSING and then updating the state, but another thread might have changed the record state between when you read it and when you want to write it, causing us to have to use check-and-set semantics as discussed in Chapter 3.

A better way would be to use an expression on the put operation, to create a conditional write. The expression is evaluated as part of the put transaction on the server, meaning that it is atomic and cannot be interrupted by another thread.

Let's assume that our cart has a state on it, for example by:

```
client.put(null, key, new Bin("state", "PROCESSING"));
```

Now, if you just update the state by using

```
client.put(null, key, new Bin("state", "COMPLETED"));
```

Then this will blindly overwrite the existing state without performing a check first. So you have to augment this with a filter expression.

Filter expressions are specified on the Policy of the operation. In this case you're performing a write operation, so you need to create a WritePolicy and pass this to the operation. In this case, you would pass something similar to:

```
WritePolicy wp = new WritePolicy(client.getWritePolicyDefault());
wp.filterExp =
    Exp.build(Exp.eq(Exp.stringBin("state"), Exp.val("PROCESSING")));
client.put(wp, key, new Bin("state", "COMPLETED"));
```

Here you have created a new WritePolicy based on the existing write policy defaults, and then set the filter expression. The expression itself is written in prefix (or Polish) notation, with the operation coming first then the arguments. So in this case, the expression that does the work is really:

```
Exp.eq(Exp.stringBin("state"), Exp.val("PROCESSING"))
```

The `Exp.eq` compares two expressions for equality and returns a boolean expression. The `Exp.stringBin("state")` reads a bin called “state”, which should be a string value, and `Exp.val("PROCESSING")` forms a constant expression holding the string value “PROCESSING”. The upshot of this is that the expression will return true if the bin “state” contains “PROCESSING.” In infix notation, this would be

```
stringBin("state") == "PROCESSING"
```

There are lots of different expressions that you can use to manipulate the record’s data into the expression that you want. There are comparison operators like `Exp.eq` you saw above, arithmetic operators like `Exp.add`, logical operators like `Exp.and`, control operations like `Exp.cond` which acts as an if-then-else statement and so on. Additionally, other classes like `ListExp` and `MapExp` allow manipulation of lists and maps respectively, similar to the `ListOperation` and `MapOperation` you saw earlier. Unfortunately, a full coverage of Expressions is beyond the scope of this book, but you can find details on them in the [Aerospike documentation](#).

We will touch on just a few more very useful features of expressions before we wrap up our coverage of them.

## Trilean Logic

Most programmers are familiar with boolean logic which allows only two values, true and false. There are rules which govern the behavior of boolean logic, such as “false AND anything equals false”. In Aerospike expressions, you use trilean logic which has three values: true, false and unknown.

Why is this necessary? Well, in Aerospike’s architecture there are typically two main components of a record – the metadata and the data. The metadata is typically kept in memory for efficiency, whereas the data is typically persisted on flash storage. This makes metadata access significantly faster than data access so if you can determine if an expression is either true or false based just on metadata, you can potentially skip loading the record from storage.

Consider an expression that needs to determine if the bin “state” in the record is PROCESSING and the last update to the record was within the last day. You saw how to check the bin data in the above example, so you will build on this by adding the last update time.

In pseudo code, what you want is

```
stringBin("state") == "PROCESSING" AND lastUpdateTime <= 1day
```

The expressions have a handy `sinceUpdate()` expression you can use to get the time since the record was updated in milliseconds. You can use Java’s `TimeUnits` to get the number of milliseconds in a day, so you would end up with:

```
Exp.and(
    Exp.eq(Exp.stringBin("state"), Exp.val("PROCESSING")),
    Exp.le(Exp.sinceUpdate(), Exp.val(TimeUnit.DAYS.toMillis(1)))
}
```

Aerospike keeps the last update time in the metadata but the `state` bin of the record is stored on disk. So to evaluate this, Aerospike would apply a first pass to the expression, evaluating all the metadata first and treating bin data as `UNKNOWN`. If the resulting value is `true` or `false` the expression is matched or not matched. However, if it's `UNKNOWN` the data is loaded off storage and the result re-evaluated, which will return `true` or `false`.

As an example of this, let's consider the above expression on a record updated a week ago. The first part of the expression (`Exp.eq(Exp.stringBin("state"), Exp.val("PROCESSING"))`) requires data to be loaded off storage and cannot be evaluated at this time so returns `UNKNOWN`. However the second part of the expression (`Exp.le(Exp.sinceUpdate(), Exp.val(TimeUnit.DAYS.toMillis(1)))`) can be evaluated using only metadata and returns `false` as the record was last updated a week ago. `False` and anything (including `UNKNOWN`) is `false`, so the expression can be failed without loading the data off storage.

Contrast this to a record that was last updated six hours ago. In this case the first expression clause still returns `UNKNOWN`, but the second clause returns `true`. `UNKNOWN` and `true` is `UNKNOWN`, so the first pass returns `UNKNOWN`. The data is then loaded off storage and the expression re-evaluated. The expression can be definitively evaluated, based on the contents of the `state` bin.

## Read Expressions

Sometimes it is useful to return data which is derived from the contents of a record but does not actually exist in the record. These expressions cannot be used as filter expressions as they do not return a boolean value, but rather serve to return data to the client.

For example, let's consider a use case where you have the total cost of items in the cart stored and you want to return the average cost. You also have a list of the items, so you can derive the average cost from these two pieces of information.

First, you will change your `addItem` method to keep a running sum of the cost:

```
public static void addItem(IAerospikeClient client, Key key,
    Map<String, Object> item) {

    client.operate(null, key, ListOperation.append("items", Value.get(item)),
        Operation.add(new Bin("total", (double)item.get("cost"))));
}
```

Now, when you add your items you will have a total cost:

```
addItem(client, key, createItem("shoes", 59.25, "/items/item1234"));
addItem(client, key, createItem("jeans", 29.95, "/items/item2378"));
addItem(client, key, createItem("shirt", 19.95, "/items/item88293"));
```

This can be shown through AQL:

```
aql> select total from test.cart where pk = 1;
+-----+-----+
| total | PK |
+-----+-----+
| 109.15 | 1 |
+-----+-----+
1 row in set (0.000 secs)
```

Now let's add a method which will use a read expression to get the average price back by dividing the total bin by the size of the list:

```
public static double getAverageCost(IAerospikeClient client, Key key) {
    Record record = client.operate(null, key,
        ExpOperation.read("avg", Exp.build(
            Exp.div(
                Exp.floatBin("total"),
                Exp.toFloat(ListExp.size(Exp.listBin("items")))
            )
        ), ExpReadFlags.DEFAULT));
    return record.getDouble("avg");
}
```

There's a number of things to unpack here! Let's take a closer look. The `operate()` command should be familiar to you by now, but you're using a new sort of operation here, an `ExpOperation` (short for expression operation). This has two variants: `read` and `write`. `Read` is used to return values back to the client, and `write` is used to update values in the record itself. Here you're using a `read`.

Given the purpose of a `read` operation is to return information to the client that doesn't exist in the record, you need to tell the operation how to return the data. This is the `"avg"` parameter in the call, saying "create a pseudo-bin in the returned information called 'avg'." The value of this field will be the result of the expression, which in this case is the value in the bin `"total"` divided by the number of items in the list `"items"`.

Note that the size of the list is an integer type and you cannot validly divide a float by an integer, hence you had to cast the list size to a float using `Exp.toFloat`. The other interesting thing to note is that the `ListExp.size(...)` function takes a list. This is typically a bin containing a list as in the example here:



```
ListExp.size(Exp.listBin("items"))
```

However, some of the list operations such as `getByIndexRange(...)` return a list, and this can be used as the argument to `ListExp.size(...)` and other operations. This effectively allows you to “chain” operations together. For example,

```
ListExp.size(  
    ListExp.append(ListPolicy.Default, Exp.val(10), Exp.listBin("items"))  
)
```

This will append 10 onto the list in bin “items” then return the size of the list.

It should be noted that this expression is not perfect. If there are no items in the list, `ListExp.size(...)` will return zero, and hence you will be dividing by zero, which will cause Aerospike to throw an exception. Instead of having an exception thrown, it would be better to just show the average as \$0.00.

There are a couple of ways of solving this, the easiest of which is telling Aerospike to ignore errors. You can do this by changing the flags you pass to the `ExpOperation.read(...)` method to include `ExpReadFlags.EVAL_NO_FAIL`.

So instead of

```
Record record = client.operate(null, key,  
    ExpOperation.read("avg", Exp.build(  
        Exp.div(  
            Exp.floatBin("total"),  
            Exp.toFloat(ListExp.size(Exp.listBin("items")))  
        )  
    ), ExpReadFlags.DEFAULT));
```

You can use:

```
Record record = client.operate(null, key,  
    ExpOperation.read("avg", Exp.build(  
        Exp.div(  
            Exp.floatBin("total"),  
            Exp.toFloat(ListExp.size(Exp.listBin("items")))  
        )  
    ), ExpReadFlags.DEFAULT | ExpReadFlags.EVAL_NO_FAIL));
```

In this case, Aerospike will catch the exception silently and the avg bin in the result set will be null. When you get null as an integer, Aerospike returns 0, which is exactly what you’re after.

# Batch Operations

Batch read operations were covered in Chapter 3. As a reminder, a batch read will take an array of Keys and return an array of Records identified by those keys. If a record is not found, null will be returned for that key instead.

Batch reads are probably the most useful batch, however Aerospike's batch capabilities extend beyond this. Batches in Aerospike are very efficient – the Aerospike client will work out which keys belong to which server and send all the keys to the appropriate server in one network packet. Depending on the settings in the BatchPolicy the servers may be sent the keys they need to process in parallel, allowing them to stream the results back to the client in parallel.

Let's look at a couple of other uses for batches.

## Batch Writes

Sometimes it's useful to perform the same set of operations across multiple records. These can be writes, appends, list or map operations, and so on. A batch write fulfills this need, taking an array of keys and the operations that should be applied to the records identified by those keys and efficiently performing the operation on them. For a rather contrived example, consider a set that holds people records, including the state and the tax rate for that state:

```
aql> select * from test.people
+-----+-----+-----+-----+-----+
| PK | name   | age | state | taxRate |
+-----+-----+-----+-----+
| 5  | "Alex" | 72  | "FL"  | 1.15    |
| 6  | "Lou"  | 25  | "CA"  | 2.05    |
| 0  | "Tim"  | 312 | "CO"  | 1.35    |
| 7  | "Jill" | 44  | "IA"  | 0.44    |
| 2  | "Sue"  | 43  | "FL"  | 1.15    |
| 8  | "Manish" | 49 | "CO"  | 1.35    |
| 4  | "Mary" | 33  | "NV"  | 0.95    |
| 1  | "Albert" | 29 | "CO"  | 1.35    |
| 9  | "Sunil" | 54 | "CA"  | 2.05    |
| 3  | "Joe"  | 19  | "GA"  | 1.25    |
+-----+-----+-----+-----+
10 rows in set (0.148 secs)
```

Imagine the use case called for increasing the tax rate of these ten people by 0.02%. You have all the information needed to form the keys of the records you need to update, so let's do that first:

```
Key[] keys = new Key[10];
for (int i = 0; i < 10; i++) {
```

```

    keys[i] = new Key("test", "people", i);
}

```

Now all you have to do is tell Aerospike to go and update the tax rate of these people. As the operation that needs to be applied to each record is the same, a batch write is perfect for this.

The line you need to do this is simply:

```
client.operate(null, null, keys, Operation.add(new Bin("taxRate", 0.02)));
```

This `operate` overload takes four parameters. The first is the `BatchPolicy`, a class that is common between all the batch overloads including batch reads and batch writes. Hence it contains general information such as network timeouts, number of retries and so on. It does not contain anything specific to writing the information. That's the job of the second parameter, a `BatchWritePolicy`. This contains information specific to how you want the writes to behave, such as what to do if the record does not exist. You are not using any special policies here, so passing `null` for both parameters is fine. Following this is the array of keys, then the operations you want to perform on each record, which is just increasing the tax rate by a fixed amount.

After running this code, you can see the results using AQL again:

```

aql> select * from test.people
+-----+-----+-----+-----+-----+
| PK | name   | age | state | taxRate |
+-----+-----+-----+-----+
| 2  | "Sue"  | 43  | "FL"  | 1.17    |
| 0  | "Tim"  | 312 | "CO"  | 1.37    |
| 4  | "Mary" | 33  | "NV"  | 0.97    |
| 1  | "Albert"| 29  | "CO"  | 1.37    |
| 9  | "Sunil"| 54  | "CA"  | 2.07    |
| 7  | "Jill" | 44  | "IA"  | 0.46    |
| 8  | "Manish"| 49  | "CO"  | 1.37    |
| 5  | "Alex" | 72  | "FL"  | 1.17    |
| 6  | "Lou"  | 25  | "CA"  | 2.07    |
| 3  | "Joe"  | 19  | "GA"  | 1.27    |
+-----+-----+-----+-----+
10 rows in set (0.166 secs)

```

You can see that the tax rate has increased across all the people in this set.

Note that if you put a filter expression onto a batch operation, the filter applies to each record. For example, let's say that instead of increasing everyone's tax rate by 0.02%, you only want to apply that operation to those people who live in Colorado (CO). You need to create a filter expression that matches these criteria:

```
Expression exp = Exp.build(Exp.eq(Exp.stringBin("state"), Exp.val("CO")));
```

This is simple enough; the expression will return true when the string in bin “state” is “CO”. You now need to apply this onto the filterExp on the policy. But if you look closely at the API you will see that both BatchPolicy and BatchWritePolicy take a filterExp! Which one should you use?

In this instance, you can use either. If both are specified, the more specific one (ie the one on the BatchWritePolicy) will override the one on the BatchPolicy. There are very few cases where it makes sense to specify both filterExps, so just be aware that it normally doesn’t make a difference. I’m going to set it on the BatchWritePolicy:

```
BatchWritePolicy bwp = new BatchWritePolicy();
Expression exp = Exp.build(Exp.eq(Exp.stringBin("state"), Exp.val("CO")));
bwp.filterExp = exp;
client.operate(null, bwp, keys, Operation.add(new Bin("taxRate", 0.02)));
```

## Arbitrary Batch Operations

Sometimes there is a set of operations that needs to happen to several records, but the operations are different for different records. For example, consider the data from the previous example:

```
aql> select * from test.people
+-----+-----+-----+-----+
| PK | name   | age | state | taxRate |
+-----+-----+-----+-----+
| 5 | "Alex" | 72 | "FL" | 1.15 |
| 6 | "Lou"  | 25 | "CA" | 2.05 |
| 0 | "Tim"  | 312 | "CO" | 1.35 |
| 7 | "Jill" | 44 | "IA" | 0.44 |
| 2 | "Sue"  | 43 | "FL" | 1.15 |
| 8 | "Manish" | 49 | "CO" | 1.35 |
| 4 | "Mary" | 33 | "NV" | 0.95 |
| 1 | "Albert" | 29 | "CO" | 1.35 |
| 9 | "Sunil" | 54 | "CA" | 2.05 |
| 3 | "Joe"  | 19 | "GA" | 1.25 |
+-----+-----+-----+-----+
10 rows in set (0.148 secs)
```

Let’s say you want to add one to Joe’s age, change Sue’s State to Ohio (OH), decrease Mary’s tax rate by 0.1%, read Jill’s age and taxRate, delete Alex’s record and change Tim’s name to Timothy. You could do this in several separate calls to the client, but you could just use an arbitrary batch operation. Let’s see what this would look like.

```
aKey joesKey = new Key("test", "people", 3);
Key suesKey = new Key("test", "people", 2);
Key marysKey = new Key("test", "people", 4);
Key jillsKey = new Key("test", "people", 7);
Key alexsKey = new Key("test", "people", 5);
Key timsKey = new Key("test", "people", 0);
Operation[] joesOps = new Operation[] {Operation.add(new Bin("age", 1))};
```

```

Operation[] suesOps = new Operation[] {Operation.put(new Bin("state", "OH"))};
Operation[] marysOps = new Operation[] {Operation.add(new Bin("taxRate", -0.1))};
Operation[] timsOps = new Operation[] {Operation.put(new Bin("name", "Timothy"))};
BatchRead jillsRead = new BatchRead(jillsKey, new String[] {"age", "taxRate"});
client.operate(null, Arrays.asList(
    new BatchWrite(joesKey, joesOps),
    new BatchWrite(suesKey, suesOps),
    new BatchWrite(marysKey, marysOperations),
    jillsRead,
    new BatchDelete(alexsKey),
    new BatchWrite(timsKey, timsOps)
));
System.out.println(jillsRead.record);

```

The code is really broken into three parts. The first is just defining the keys for the various people, and the second is defining the operations for the `BatchWrites`. Each of the `BatchWrites` takes an array of operations although you're just using one operation per record.

The last, and most important part is the actual call to `operate()`. This just takes a `BatchPolicy` and a list of `BatchRecords`. `BatchRecord` is a superclass of `BatchRead`, `BatchWrite`, `BatchDelete` and so on.

Most of the `BatchRecords` you're using do not need to return a result, so you can just instantiate the appropriate operation within the `operate()` call. The exception is the `BatchRead` where you read Jill's age and `taxRate`. The results for all operations are returned in the `BatchRecord` passed to the operation, so you need to save Jill's `BatchRead` in a variable. When the call succeeds, you can get the information you need from the `.record` part of the `BatchRecord`.

## Secondary Indexes

Similar to many other databases, Aerospike has secondary indexes that allow for efficient querying of records that match specific criteria. For example, "show me all the people who live in Colorado" or "get all people aged between 25 and 39." Secondary indexes are defined on a bin in a particular set and must have a type, such as numeric or `String`. Table 4-4 shows the different types of secondary indexes that are supported and the operations available on them.

*Table 3-4. Secondary Index types*

Type	Allowed Operations	Comments
NUMERIC	Equality, range comparisons	For indexing and querying integer type bins
STRING	Equality only	Aerospike stores a hash of the string to conserve space, hence only exact matches are supported. Strings are always case sensitive comparisons

GEO JSON Point-in-region, region-contains-points

GeoJSON queries are an advanced topic not covered in this book but allow efficient comparisons of geospatial data

Note that if a record contains a bin which has a secondary index defined on it, but the type in that bin does not match the secondary index type, that record will not be returned from a secondary index query. For example, a numeric index is defined on a bin “age”, but one record contains a string “40”. This record will never match the secondary index.

To create secondary indexes you will use the Aerospike Administration tool, `asadm`. System administrators use this tool extensively to monitor and affect the Aerospike cluster. You will touch on it here, but it will be covered more fully in later chapters.

To start `asadm`, simply run:

```
% asadm [-h <ip_address>]
```

Like Aerospike’s other tools, the `ip` address defaults to `localhost` (127.0.0.1) so for many local installations no parameter is needed after the command name.

This will place you in an interactive shell:

```
Admin>
```

Commands entered will have their results shown in real time. A good place to get started is entering “`help`” at this prompt. It will show a list of available commands. The shell supports `help` commands (see chapter 7 for more detail), as well as command completion, so if you know part of what you want to do you can type that in and then double-tap the `Tab` key to see what options are available.

First, you want to show what indexes there are:

```
Admin> show index
Admin>
```

You will find *index* used frequently in Aerospike documentation and tools as shorthand for “secondary index.” In your environment, at the moment, there are no secondary indexes, so you need to create one.

`Asadm` has two modes: user mode and privileged mode. User mode enables the viewing of information but doesn’t allow cluster management such as creating secondary indexes. You need to change to privileged mode which simply requires entering “`enable`.” In a production environment only users with sufficient permissions are able to enter privileged mode, but you have not enabled security on your cluster yet, so you don’t need to worry about this.

```
Admin> enable
Admin+>
```

Note the cursor changes slightly with the addition of the “+”, and the font will change to red.

Once in privileged mode you can create the index:

```
Admin+> manage index create numeric age_idx ns test set people bin age;
Use 'show index' to confirm age_idx was created successfully.
```

```
Admin+> show index
~~~Secondary Indexes (2023-08-28 13:00:08 UTC)~~~
  Index|Namespace|  Set|Bin|  Bin|  Index|State
  Name|         |     |   |   | Type|  Type|
age_idx|test     |people|age|numeric|default|RW
Number of rows: 1
```

You create the secondary index with the “manage index” command. Although it looks daunting at first glance, it is fairly readable and says ‘manage secondary indexes; create a numeric index called “age\_idx” in the namespace “test” on the set “person” and the bin “age.”’ Remembering what parameters are needed is made much easier with the help system. To work out the parameters of the above command, do this:

```
Admin+> manage index help
"manage index" is used to create and delete secondary indexes. It should be used
in conjunction with the "show index" or "info index" command.
- create:
  Usage: create <bin-type> <index-name> ns <ns> [set <set>] bin <bin-name> [in
<index-type>] [ctx <ctx-item> [. . .]]
  bin-type      - The bin type of the provided <bin-name>. Should be one of the
following values:
```

Note that the output has been truncated for the sake of brevity, but you can see the usage parameters. You can omit the last two parameters as these are used for indexing into CDTs.

## Using the Secondary Index

Now that you have your secondary index, let’s use it to query people in your set within a particular age range.

```
Statement stmt = new Statement();
stmt.setFilter(Filter.range("age", 25, 39));
stmt.setNamespace("test");
stmt.setSetName("people");
RecordSet recordSet = client.query(null, stmt);
while (recordSet.next()) {
    System.out.println(recordSet.getRecord());
}
recordSet.close();
```

The first thing you have to do is create a `Statement` object. This contains the parameters for the statement, including which namespace and set to run the query on. You specify the filter you want to use, in this case a range filter on the “age” bins with a minimum age of 25 and a maximum age of 39. Both ends of the range are inclusive, so it will pick up anyone aged 25 and 39 as well as people whose ages are in the middle of the range.

The query is then executed and returns a `RecordSet`. The `RecordSet` allows easy iteration through the matching records as shown here, with both the record and the key available once `next()` has returned `true`.

While this query looks innocuous, Aerospike does a lot of work behind the scenes for a query like this. Imagine your Aerospike cluster contains a billion records in the people set with a million of these matching our criteria. Aerospike will, by default, fire a request off to each of the servers in parallel. Each Aerospike server consults a memory tree that contains the value of the “age” bin for each record in the set. When a matching record is found, the record is streamed back to the client.

The client receives the responses from all servers in parallel. It buffers some of them but for large result sets it will rely on being able to pull more from the appropriate server as needed rather than trying to store everything in memory. The client also keeps track of which records it has processed from which servers, so if the query is interrupted by a server going down for example, the client can continue to serve records from the cluster without needing to restart the query.

It is important to close the result set once you’ve finished processing it. This will free up client-side and server-side resources.

## Multiple Predicate Queries

Aerospike only supports using one secondary index per query. So even if you had say, an index defined on the “age” bin and a different index defined on the “state” bin and you wanted to find everyone between 25 and 39 in Colorado, you would only be able to use one of the indexes, not both. Typically you would use the index with the highest selectivity in this case, that is the one that would likely return the smallest set of responses.

However, there is an easy solution for handling multiple predicates like the above. If you guessed “using expressions”, you are correct! You can put an `Expression` on a query with a secondary index. When you do this, Aerospike will use the secondary index to load the records that match and then apply the `Expression` to each of those records, returning only the records that match the expression. So, to retrieve only the people between 25 and 39 in the state of Colorado (CO), you would change the code to the following:



```
Statement stmt = new Statement();
stmt.setFilter(Filter.range("age", 25, 39));
stmt.setNamespace("test");
stmt.setSetName("people");
QueryPolicy qp = new QueryPolicy(client.getQueryPolicyDefault());
qp.filterExp = Exp.build(Exp.eq(Exp.stringBin("state"), Exp.val("CO")));
RecordSet recordSet = client.query(qp, stmt);
while (recordSet.next()) {
    System.out.println(recordSet.getRecord());
}
recordSet.close();
```

As you can see, you have created a new `QueryPolicy` based on the client's `queryPolicyDefault`, and then set the filter expression you want to use on it. You then pass the `QueryPolicy` into the `query(...)` call, and that's all there is to it!



## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

Since you are just trying to get started with Aerospike, this chapter will not dive deep into how the database is built. But knowing some key aspects of how Aerospike works under the covers will help you understand why you do certain things with it, and how to get the most out of its capabilities.

This chapter focuses on three major topics – scale out, scale up, and how transaction management with strong consistency is accomplished on this distributed system.

## Scale Out

In order to linearly scale performance latency and throughput, Aerospike must scale out on multiple commodity computers, aka nodes. To do this, Aerospike has several fundamental principles built into its heart. These foundational pillars affect how everything works in the database. The key goal of the scale-out architecture is to maintain a uniform distribution of data across data partitions and across nodes. Combined with the ability to support elasticity using dynamic addition and removal

of nodes, this architecture ensures that the system avoids hot spots (overworked nodes or network connections) while providing excellent performance (both high throughput *and* extremely low latency) with linear scalability.

## Shared Nothing Database Cluster

Aerospike is a shared nothing database. The database cluster consists of a set of commodity server nodes, each of which has CPUs, DRAM, rotational disks (HDDs) and optional flash storage units (SSDs). These nodes are connected to each other using a standard TCP/IP network. Several fundamental aspects of the database make it shared nothing:

*Every node is identical to every other node*

This is true both in terms of hardware capabilities, and software. There are no master nodes, and therefore no single points of failure.

*Data-to-node mapping is on every node*

This means that there is always only one hop from the smart client direct to the data, with no intermediary steps required.

*Data partitioning is done with distributed hash tables*

Aerospike uses an extremely random cryptographic hash on every key to spread data very evenly across the cluster.

*Long-running tasks and short tasks are prioritized in real-time*

The database dynamically and intelligently handles the interplay between long-running tasks like rebalancing data when node count changes and short-running, low-latency tasks like transactions. This protects the SLAs on the short tasks.

*Cluster management is dynamic*

Nodes can be added or dropped without interrupting transactions. Data rebalancing is done dynamically, and maintenance actions such as rolling upgrades happen without interrupting the service.

The basic idea behind all of these is to ensure that the cluster runs without operator intervention in the presence of dynamic node arrivals and departures, the data is uniformly distributed across nodes in the cluster, and the system delivers high performance and scales linearly as more client and server nodes are added to the system.

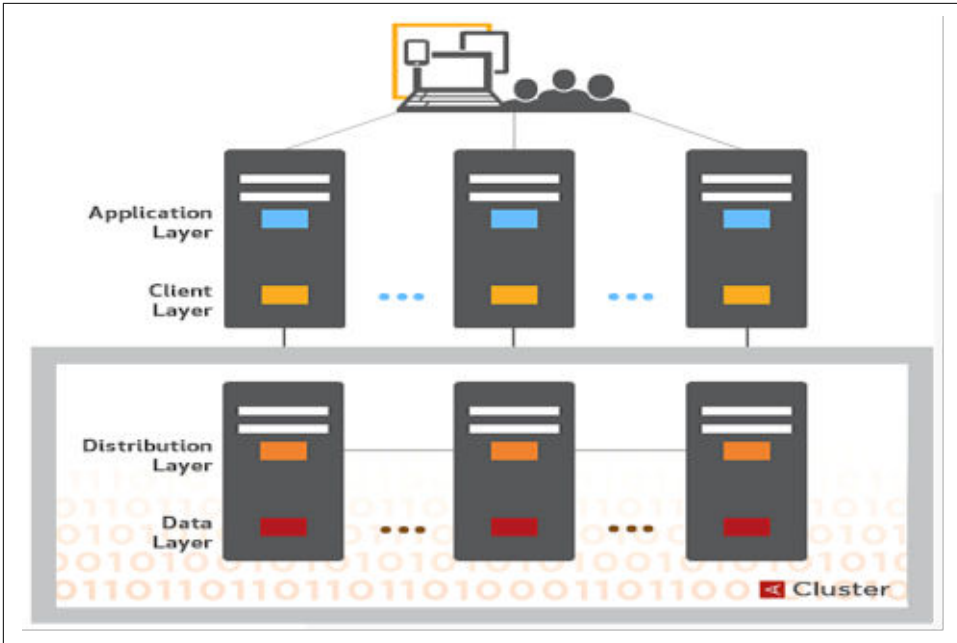


Figure 4-1. Aerospike shared-nothing database architecture

## Data Distribution

A key aspect of Aerospike is its ability to divide and conquer the problem of high scale datasets using a deterministic uniform distribution of data to minimize data migrations.



Aerospike splits partitions using the digest of the **RIPEMD algorithm** for hashing. This algorithm is very robust against collisions. The distribution of keys in the digest space and therefore in the partition space is always uniform because even if the hash is skewed, the digest of it won't be.

A partition is the primary unit of data segmentation. The random cryptographic hash on every key spreads data evenly across nodes as shown in [Figure 4-2](#). The hash splits data into 4096 partitions to then be mapped to nodes in the cluster. Partitions are assigned to nodes at random. The partition assignment algorithm objectives are:

*Be deterministic*

Each node in the distributed system can independently compute the same partition map.

### Distribute data uniformly

Distribute master partitions and replica partitions across all nodes in the cluster equally.

### Minimize data migrations

When the cluster changes, new nodes added or removed, the less data that must be moved around, the better.

Aerospike uses a uniquely modified algorithm that automatically creates uniform balancing of partitions across nodes while minimizing the data migrations, as much as possible.

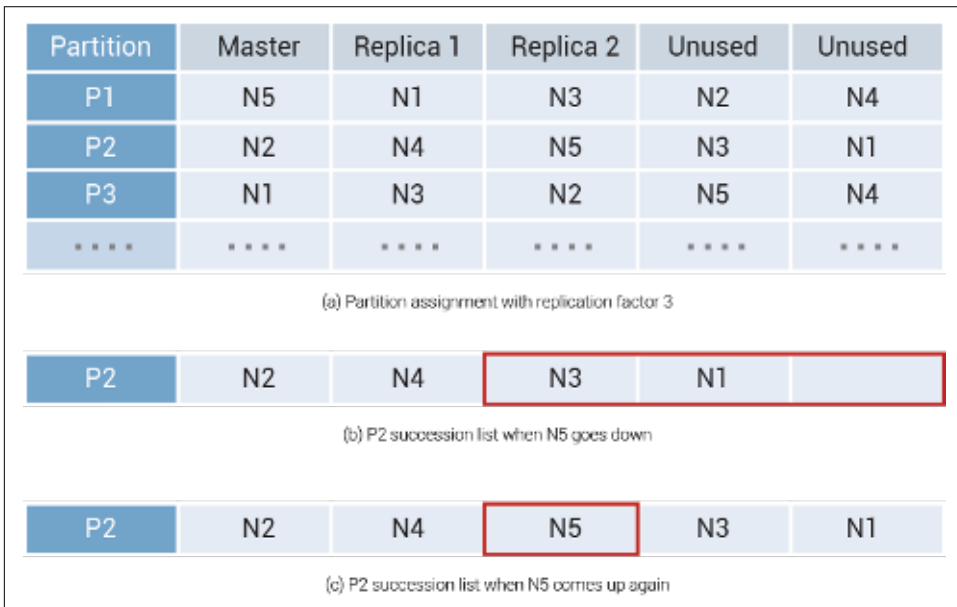


Figure 4-2. Partition Map: Partition to Node Assignment

Figure 4-2(a) shows the *partition map* for the data assignment on a 5-node cluster with a replication factor of 3. Only the number of columns equal to the replication factor are used to show where data is mapped, the first three columns in the example shown.

The last two columns in the map are not the locations of data since the data is only replicated three times. Instead, the next columns store the *succession list* of locations that data will be migrated to in case one of the nodes that has data on it fails.

Figure 4-2(b) shows what happens to the data in partition 2 if a node goes down or network connection to it is lost. In this example, node 5 goes down. Since the partition map already stores the information on where data will go next for that

partition, the data is migrated to node 3, the next node listed in the map in the column to the right.

Note that this also shifts left the next column, so if node 3 were to also go down, node 1 would be next to receive a copy of partition 2's data.

Figure 4-2(c) shows what happens when node 5 comes back up. That partition of the data is again mapped to node 5 and the other two map columns slide back to the right. They go back to showing the succession list for that partition.

Given the fixed number of partitions, 4096, and the ID's of the nodes, the partition map is computable. Every node in the cluster can compute it, so every node will have a map to where every data partition resides, without having to communicate with each other.

There is more complexity to this. In larger clusters where there are fewer partitions per node, it is necessary to sacrifice some data migration minimization to get more even data load spread. Rack awareness introduces another complicating factor as well. Feel free to dive into the docs or academic papers published by Aerospike to get more detail if you wish.

## Cluster Self-Management

The Aerospike cluster management subsystem self-manages all of this partitioning, mapping, and migration. It ensures that all the nodes come to a consensus on the current membership of the cluster.

Events such as network faults and node arrival or departure trigger cluster membership changes. These changes can be both planned and unplanned. Examples include randomly occurring network disruptions, scheduled capacity increments, and hardware/software upgrades.

The three components to clustering are:

### *The heartbeat subsystem*

Maintains adjacency list and stores latest heartbeat exchanged to keep track when nodes are added or removed.

### *The clustering subsystem*

Maintains the succession list.

### *The exchange subsystem*

Exchanges partition data and invokes partition balance.

The specific objectives of the cluster management subsystem are:

Arrive at a single consistent view of current cluster members across all nodes in the cluster.

Automatically detect new node arrival/departure and seamlessly reconfigure the cluster.

Detect network faults and be resilient to such network flakiness.

Minimize time to detect and adapt to cluster membership changes.

## Cluster view

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

`cluster_key` is a randomly generated 8-byte value that identifies an instance of the cluster view.

`succession_list` is the set of unique node identifiers that are part of the cluster.

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

Every change to the cluster view significantly affects operation latency and the performance of the entire system. This means there is a need to quickly detect node arrival/departure events for an efficient consensus mechanism to handle any changes to the cluster view.

However, migrating data when a node is not down, just has a flaky network connection for instance, would be very inefficient. So, it is important not to act until a node is down for certain.

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

The main objectives of the detection mechanism are:

To avoid declaring nodes as departed because of sporadic and momentary network glitches.

To prevent an erratic node from frequently joining and departing from the cluster. A node could behave erratically due to system level resource bottlenecks in the use of CPU, network, disk, etc.



Alternatives such as replica writes can also be used as a secondary surrogate for heartbeat messages. The cluster view is unchanged as long as a primary or secondary heartbeat message is received within the timeout interval.

Potential node failure is anticipated automatically. Every node in the cluster evaluates the health score of each of its neighboring nodes by computing the average message loss from that node.

An erratically behaving node typically has a high average message loss. If an unhealthy node is a member of the cluster, it is removed from the cluster. If it is not yet a member, it is not considered for membership until its average message loss falls within tolerable limits.

## Cluster view changes

Changes to the adjacency list trigger a run of the Aerospike clustering algorithm that arrives at the new cluster view. Aerospike works to minimize the number of transitions the cluster would undergo as an effect of a single fault event.

For example, a faulty network switch could make a subset of the cluster members unreachable. Once the network is restored, there would be a need to add all these nodes back to the cluster. To minimize cluster transitions, which are fairly expensive in terms of time and resources, nodes make cluster change decisions only at the start of fixed, configurable cluster change intervals.

The idea is to avoid reacting too quickly to node arrival and departure events, as detected by the heartbeat subsystem, and instead, process a batch of adjacent node events with a single cluster view change. Aerospike's cluster management scheme allows for multiple node additions or removals at a time without downtime.

## Intelligent clients

Databases don't exist in isolation. The full stack needs to function so that the end-to-end system scales. An intelligent client layer absorbs the complexity of managing the cluster. There are various challenges to overcome here and a few of them are addressed below.

### *Discovery*

Clients use one or more seed nodes, which tells them about every adjacent node until the client knows the role and existence of every node in the cluster. The partition map (see [Figure 4-2](#)), showing the relationship of partitions to nodes, is exchanged and cached with the clients. With that map, clients are always a single hop from the data. The map identifies the data's exact location, eliminating the need for any intermediate routing nodes.

### *Information sharing*

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

## **Cluster node handling**

For each cluster node, at the time of initialization, the client creates an in-memory structure on behalf of that node and stores its partition map. It also maintains a connection pool for that node.

When a node is considered down, the map, in-memory structure, and connection pool are all torn down. The setup and tear-down is a costly operation. If the underlying network is flaky and this repeatedly happens, it can degrade the performance of the overall system. This means, a sophisticated approach to identifying cluster node health is needed. Aerospike has a couple of systems to help.

### *Health score*

A transient network issue or other problem may make it seem like a node is down while the server node is actually up and healthy. Clients track the number of failures encountered on database operations at a specific node. The client drops a cluster node only when the failure count (a.k.a “happiness factor”) crosses a particular threshold. Any successful operation to that node will reset the failure count to 0. This scheme is implemented by default in the client libraries.

### *Cluster consultation*

Flaky networks are often tough to handle. One-way network failures (A sees B, but B does not see A) are even tougher. There can be situations where the cluster nodes can see each other but the client is unable to see some cluster nodes directly (say, X). In these cases, the client consults all the known nodes of the cluster to see if any of them has X in their neighbor list. If a node reports that X is in its neighbor list, the client does nothing. If X is not in any client-visible node’s neighbor list, the client will wait for a threshold time, and then remove the node by tearing down the data structures that reference it.

Through the years, we have found that these automatic schemes greatly improve system stability.

## **Scale Up**

Scaling out to multiple computer nodes is something that all distributed systems have in common. At Aerospike, we also focused a lot on scaling up to take advantage of

every bit of network, storage, and processing capacity that is available within each node.

In this section, we describe the Hybrid Memory Architecture (HMA) storage architecture that enables tens of Terabytes of data in Flash storage to be read and written with sub-millisecond latency. You'll also learn about system-level techniques that leverage multi-core processors to help Aerospike scale up to millions of transactions per second at sub-millisecond latencies per node.

Aerospike's ability to scale up on nodes effectively and use Flash drives means the following:

Scaling up to higher throughput levels on fewer nodes.

Better availability, since the probability of a node failure typically increases as the number of nodes in a cluster increases.

Lower total cost of ownership as storing and accessing data in real-time from Flash is a lot less expensive than storing and accessing it from memory.

Easier operational footprint. Managing a 20-node cluster versus a 200-node cluster is a huge win for operators.

## Hybrid Memory Architecture

To take full advantage of each compute node, Aerospike heavily leverages SSDs (Flash drives). The database has implemented bespoke algorithms so that it can access data on SSD with sub-millisecond latency. Indexes are stored separately in memory (DRAM or PMEM). This is the heart of the patented Hybrid Memory Architecture™ (HMA).

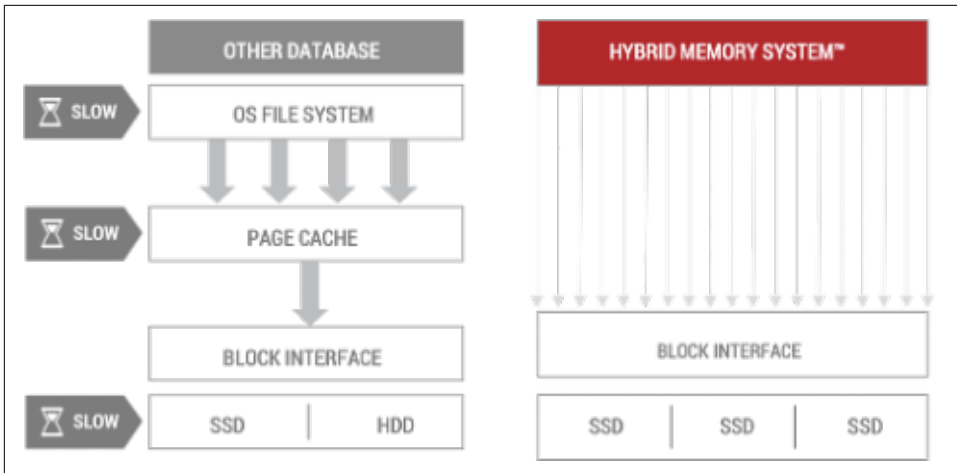


Figure 4-3. Hybrid Memory Architecture compared to standard file-system-based database architecture

Aerospike data access is written natively in C, so it bypasses the need for an operating system or file system layer between it and the data. Unlike most databases, Aerospike also does not utilize a page cache when reading, which removes another layer. The index, which is fully resident in memory, points to the actual data item on the Flash drive location allowing reads within a millisecond.

Writes can't be done directly on the SSD due to wear leveling issues where you can burn out portions of the SSD if you overwrite them too often. Aerospike implemented a log structure file system with large block writers, plus copy and write semantics, so that SSD writes are gathered in memory in large buffers, and then flushed to Flash disk. This allows Aerospike to manage very heavy read loads at high performance, even when in the presence of very heavy write loads.

Summary of SSD strategies:

Direct SSD device access – not filtered through an operating system's file system layer.

Highly parallelized – taking full advantage of multi-SSD nodes.

Large block writers to SSD – reducing wear on SSD cells to extend life of hardware.

SSD vendor-optimized – to get the best possible read and write performance.

Continuous, non-disruptive defragmentation – to prevent running out of space on disk.

The end result of all this is an increase in the amount of data you can store in a node, in proportion to the amount of SSDs on the node, not just the amount of memory. This means that the number of nodes in a large Aerospike cluster is typically an

order of magnitude less than comparable systems that store all the data in memory. For example, Aerospike would use 20 nodes to do the same work as an in-memory transactional database on 200 nodes.

## Multi-Core Processors

For this to work in practice, Aerospike also has to run 10X the transactions on each node. This requires taking advantage of multi-core processors and multi-CPU nodes.

Aerospike uses multiple strategies to make this work:

Multi-threading.

Highly efficient C code.

CPU-pinning (such as NUMA-pinning done in collaboration with Intel).

Thread binding to specific network queues (ADQ binding to CPU threads for example, optimized for particular Intel hardware cards).

Using parallel network queues to avoid bottlenecks on particular CPUs.

Running in the network listener thread to avoid costs associated with thread context switches.

And several other strategies and techniques.

All of these together allow Aerospike to drive hundreds of transactions per second through a single node.

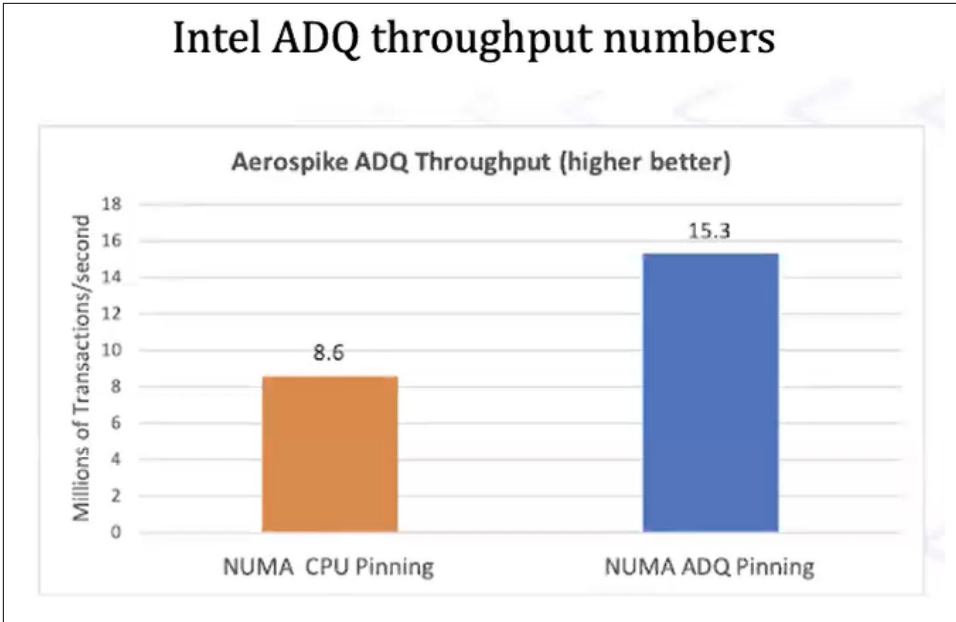


Figure 4-4. Benchmark example showing large headroom for transactions.

As benchmarks such as the one in Figure 4-4 have shown, these strategies can run as many as 8 million transactions per second, or with Intel’s ADQ (Application Data Queue), 15 million transactions per second through a single node. In practice, you are unlikely to need that many transactions on a single node, but it is useful to know that the headroom exists.

This is one strategy that helps Aerospike run very high throughput low-latency workloads at very high scale on small cluster sizes.

## Memory Fragmentation

Aerospike handles all its memory allocation natively rather than depending on the programming language or on a runtime system. Aerospike keeps the index packed into RAM, with sizes at high scale over 100 GB and high transaction rates, memory fragmentation is a major challenge. Using a specific memory allocator library ([jemalloc](#)), and other strategies like grouping objects by namespace, Aerospike optimizes the long-term object creation, access, modification and deletion pattern and minimizes fragmentation.

## Data Structure Design

For data structures like indexes and global structures which need concurrent access,

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

These structures are carefully designed to make sure that frequently and commonly accessed data has locality and falls within a single cache line in order to reduce cache misses (when the application requests data from a cache, but it isn't there) and data stalls (time spent waiting for data to be retrieved). For example, the index entry in Aerospike is exactly 64 bytes, the same size as a cache line.

In production systems like Aerospike, the functional aspects, system monitoring, and troubleshooting features need to be built in and optimized. This information is maintained in a thread-local data structure and can be pulled and aggregated together at query time.

## Scheduling and Prioritization

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

This is achieved by following three major principles.

### *Partition jobs based on their type*

Each job type is allocated its own thread pool and is prioritized across pools. Jobs of a specific type are further prioritized within their own pool.

### *Effort-based unit of work*

The basic unit of work is the effort needed to process a single record including lookup, I/O and validation. Each job is composed of multiple units of work, which defines its effort.

### *Controlled load generation*

The thread pool has a load generator, which controls rate of generation of work. It is the threads in the pool that perform the work.

Aerospike uses cooperative scheduling whereby worker threads yield CPU for other workers to finish their job after X units of work. These workers have CPU core and

partition affinity to avoid data contention when parallel workers are accessing certain data.

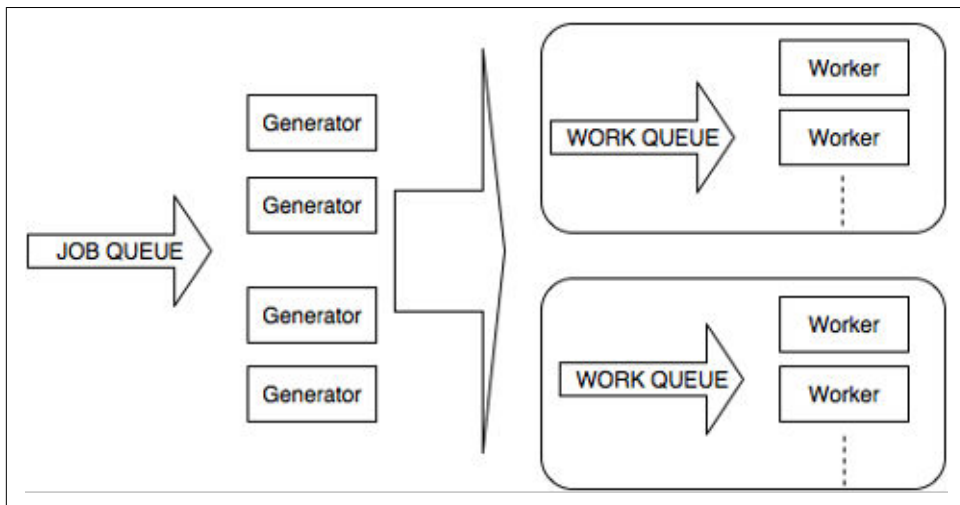


Figure 4-5. Job Management

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

## Parallelism

Aerospike's uniform data partitioning enables parallel processing using system resources in a balanced and efficient manner. Many Aerospike nodes are configured with multiple SSD drives. In addition to uniformly distributing data across nodes, Aerospike data can also be distributed randomly into these storage devices providing extremely high levels of parallelism on small clusters.



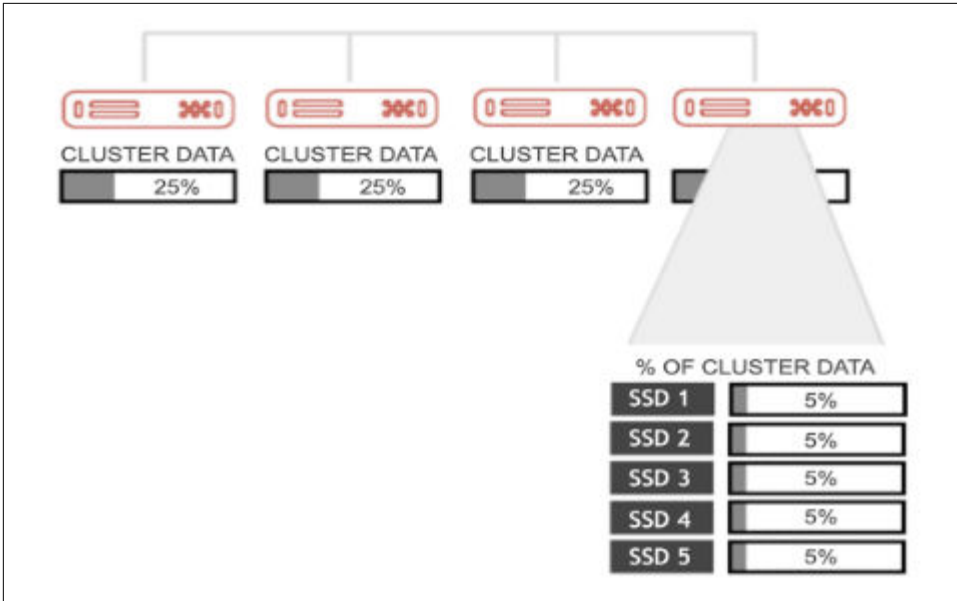


Figure 4-6. 20-way parallelism on a 4 node cluster

This provides the opportunity for even higher levels of parallelism on low numbers of compute nodes. Figure 4-6 shows a system that contains 4 nodes with 5 disks each, a small example system.

This simple example illustrates that a 100-node cluster with 16 storage devices per node can drive a 1600-way parallel execution of a high throughput workload of individual record writes in the millions of transactions per second.

With the scale up available in a hybrid memory/flash configuration using multi-threading on multi-core processor architectures with parallel network queues, up to 100TB can be stored per node resulting in database storage of 10 petabytes on that same 100-node cluster that can be processed at a very high rate of throughput with sub-millisecond read/write latency.

## Distributed Transaction Consistency

Aerospike uses all the performance its scale out and scale up capabilities provide to make transactional algorithms with strong consistency. High performance enables us to develop algorithms using strategies that are not available to other systems that are slower on the same underlying infrastructure.

This section covers the basics of how transactional consistency is guaranteed in an Aerospike cluster, and how transactions can be implemented when components are separated geographically.

## Strong Consistency in Transactions

The CAP theorem of distributed databases essentially states that while there are three important aspects to distributed databases, Consistency, Availability, and Partition tolerance, one aspect must always be sacrificed to keep the other two high. Which two are chosen determines how the database functions. Partition tolerance is necessary for distributing data and workload, so the choices are generally AP, which emphasizes availability, or CP, which emphasizes data consistency.

Many distributed databases sacrifice Consistency, dropping to a state known as eventual consistency where data across a cluster will eventually reach a state so that the data is the same on all nodes of the cluster, but there will be a period of time when this is not the case.

However, transactional databases generally require the opposite of eventual consistency, aka strong consistency, where any node can be queried, and even if a different node is queried at the same time, the result will be identical.

Aerospike can be configured in AP-mode, which prioritizes availability over consistency. But even in AP-mode, it is uncommon to violate consistency in a properly running Aerospike system, except during the following two scenarios:

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

## Roster

One key difference between Aerospike and other distributed databases is that Aerospike does not use quorums in the way other databases do. The only quorum is a simple write all and read one copy, which is possible due to Aerospike's level of write and read performance. Instead of quorum, Aerospike uses a roster-based strong consistency scheme.

When configured for strong consistency, Aerospike defines a roster as the set of nodes which are intended to be present. When all the roster nodes are present, and all

the partitions are in their correct computed location, the cluster is in its steady state and provides optimal performance.

As we described earlier, partitions are assigned to nodes in a cluster using a random assignment. For data redundancy, each partition will be copied the number of times determined by the replication factor, a configurable setting.

One of these partition copies is referred to as roster-master and the rest, roster-replica.

*roster- master*

The roster-master refers to the node that would house the primary copy of a specific data partition.

*roster- replica*

The roster-replica refers to the node or nodes that would house secondary copies of a specific data partition.



The roster and partition map is on ALL nodes, even if they are disconnected from the rest of the cluster. This means that every node knows where every partition is, even if communication between nodes is cut off.

Along with the cluster roster is the concept of a majority and supermajority of the roster.

*Majority*

More than half the total number of nodes in the roster.

*Supermajority*

More than the total number of nodes in the roster minus less than the replication setting.

For example, in a cluster with 10 nodes and a replication factor of 3, a majority would be 6 or more nodes. A supermajority would be 8 or 9 nodes, more than 10 - 3.

## Split-Brain Conditions

When network connectivity drops between nodes in a cluster, this essentially creates two functional sub-clusters that can continue to function to some extent. This is called a split-brain condition. Most systems for providing strong consistency require a minimum of three copies [[https://en.wikipedia.org/wiki/Consistency\\_model](https://en.wikipedia.org/wiki/Consistency_model)]. So, if a cluster splits, one of the two sub parts can allow writes if it has a majority (two out of three) copies of the data item. Aerospike optimizes this further by regularly storing

only two copies but using an adaptive scheme that adds more write copies on the fly in situations where they are necessary, thus optimizing the performance in most cases while incurring a small amount of overhead in edge cases that rarely occur.



Note that even a two-copy system still needs a minimum of three nodes to preserve availability. A node will never have both the master and replica of the same partition. The Aerospike replication scheme provides an equivalent level of availability with 2 copies as a traditional quorum-based system using 3 copies. This reduction continues to grow as the replication factor increases - where other systems store  $N$  replicas, Aerospike only needs to store  $(N / 2) + 1$  to achieve a similar availability level during common network failures.

The rules are in CP-Mode, which prioritizes consistency over availability, a partition is available and active for both reads and writes if:

1. A sub-cluster has *both* the roster-master and all the roster-replicas for a partition.
2. A sub-cluster contains *a majority* of nodes in the full roster, and has *either* the roster-master or a roster-replica.
3. A sub-cluster has *exactly half* of the nodes in the full roster, and it has the *roster-master* for a partition.
4. A sub-cluster has a supermajority of the nodes in the full roster (A supermajority is a greater number of nodes than full roster minus replication factor.)



Some nodes are excluded while counting the majority and supermajority conditions:

A previously departed node that rejoins the cluster with missing data (e.g., one or more empty drives).

A node that was not cleanly shut down is enabled.

These nodes will have an evade flag set until they are properly inducted into the cluster with all data.

Let's look at some examples to make this clearer. For simplicity, assume the replication factor is set to 2 in these examples. Every partition in the system will have one roster-master and one roster-replica.

Start with a simple 5 node cluster, with the roster-master of a particular partition's data on node 5 and the roster-replica on node 4. In this situation, the roster is whole and the data partition is fully active for both reads and writes.

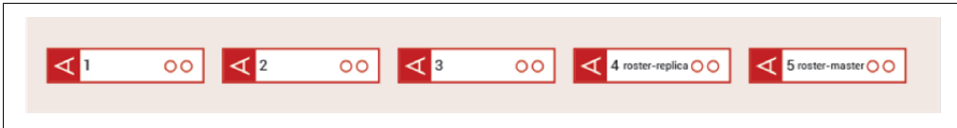


Figure 4-7. a: Whole cluster

If network communication with nodes 4 and 5 is lost, they are cut off from the rest of the cluster, creating essentially two sub-clusters as in figure 7b. Since both the roster-master and roster-replica for a partition are in one sub-cluster and no copy of that data exists in the other sub-cluster, only the smaller cluster remains available for that partition for both reads and writes.



Figure 4-8. b: Partition master and replica in same sub-cluster

Now, suppose that only node 5 is disconnected or down, possibly due to a rolling upgrade. This isolates the roster-master from the rest of the cluster, but the replica of that partition remains. Because the roster-master is in a smaller minority cluster, but a replica is in the majority cluster, the rule is that that partition of data becomes unavailable in the smaller cluster, just node 5 in the example.

Temporarily, the roster-replica will be promoted to master (alt-master) and a second replica (alt-replica) created on another node according to the succession list in the partition map. In the example in figure 5-7c, the data is replicated on node 3. Because every node has the roster and partition map, the new data location can still be computed by all nodes. This allows two writes to maintain consistency and durability, and the partition remains live for both reads and writes. In any two-cluster split brain situation, data partitions will be 100% available in either one sub-cluster or the other, although not always in both.



Figure 4-9. c: Partition replica in larger sub-cluster, master disconnected

If this was a case of a *rolling upgrade*, node 5 would become active again after being upgraded and would rejoin the cluster. Node 4 would be taken down next. Between node 5, which has all data from before it went down, and node 3, which has all changes made to the data while node 5 was down, a complete picture of the data is still available for any request. A read might be slightly slower due to having to

check both nodes to make sure the latest changes are included, but in general, there would be no disruption of reads or writes, 100% availability. (For more information on rolling upgrades, check the [Aerospike documentation](#)).

In figure 5-7d is an example of a possible split brain condition where the partition would be completely unavailable. In this situation, the master is isolated as is the replica, each in a separate sub-cluster. This is an example of the CAP theorem in action. In order to maintain strong consistency, availability would be sacrificed in this instance.



Figure 4-10. d: Master and replica both separated from rest of cluster

## Writes

Writes in Aerospike are always a *write all* mechanism, meaning that if a write is done on a database with a replication factor of 2, it will always write 2 copies of the data changes. If the replication factor is 3, it will always write 3 copies.

The example below is in a three copy system (replication factor set to 3).

1. The client initiates a write to the master node for that partition's data.
2. The master node copies the data changes to the two replicas in parallel.
3. The replica nodes respond back to the master that the data copies were successful.
4. The master commits the transaction and communicates the commit to all replicas.

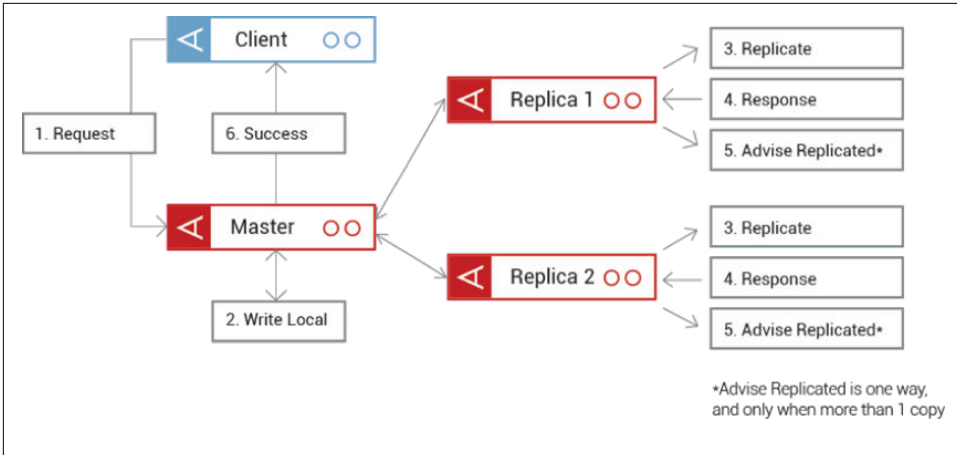


Figure 4-11. Synchronous write logic

The CAP theorem tends to be thought of as having two types of consistency, either strong consistency or eventual consistency, but there is a type of slightly lower form of strong consistency: sequential consistency also known as session consistency. For reads, Aerospike supports linearizability and sequential consistency. In the case of writes, though, they are always strongly consistent. There is no data loss tolerated.

## Rack Awareness

Rack awareness means that a distributed application is aware of which nodes are on which racks in a data center. Since Aerospike is rack aware, it will not store both the master and replicas of a partition on the same rack. This is so that if the entire rack is lost, the database can continue functioning normally on another rack.

Rack awareness has a very strong advantage as far as performance for reads. If the data is available on multiple racks, you can access the data on a single rack for reads as if it were local.

Writes, though, must always access multiple partition copies, and multiple racks in the case of a rack aware cluster.

However, since a complete copy of the data is known to be available, in some situations, Aerospike can continue to function for both reads and writes with both availability and consistency when communication between racks is down. An example of when that capability is essential is when racks are geographically separated.

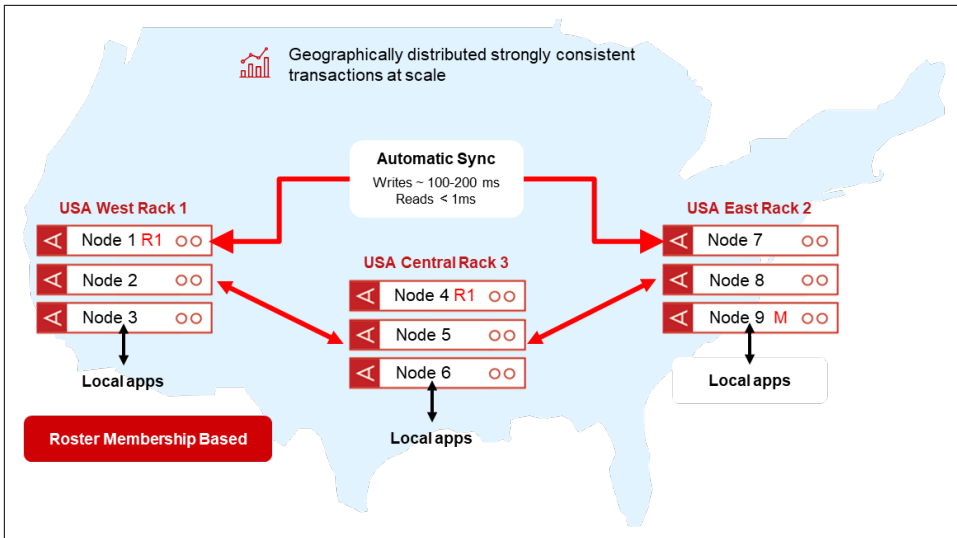


Figure 4-12. Geographically distributed database

In certain types of use cases, such as interbank money transfers, it is necessary for a single database to span multiple geographically separated locations. In this case, the tradeoff is a much higher write latency, but the data remains consistent and reliable, which is more important in this sort of use case.

The example in figure 5-9 is a 3 replica system with three racks spread across the continental United States. Each site has a full copy of the data. The rack awareness allows local client applications to read data locally at extremely low latency, under 1 millisecond. The writes will take considerably longer since they must write to all three racks and are subject to network delays, speed of light limitations, etc. These generally fall between 100 and 300 millisecond delays.

The result of this configuration is a completely synchronous active/active system. If an entire rack goes down, or connectivity to it is lost causing a split-brain situation, the other two racks can continue to run with both availability and strong consistency without causing any conflicts.

When a rack goes out or rejoins, no operator intervention is required to keep the database running smoothly. It's taken care of automatically. For you, this means very high uptime with the only tradeoff being higher write latency.

## Asynchronous active-active replication

Depending on how you configure the geo-distributed database, instead of trading write latency, you can choose to lose some data consistency. Cross-data replication (XDR) transparently and asynchronously replicates data between Aerospike clusters.



Companies often use XDR to replicate data from Aerospike-based edge systems to a centralized Aerospike system. XDR also enables companies to support continuous operations during a crisis (such as a natural disaster) that takes down an entire cluster.

Read the Aerospike documentation for more information on the various options.

## Conclusion

There is far more involved in Aerospike architecture than would make sense in a basic up and running book. Rather than drown you in details, hopefully this is enough to give you a basic understanding of how Aerospike is different at its foundation from other databases. And with luck, understanding this foundation will help your understanding of the database so you can get the most out of it.

Next, we'll dive into modeling data in Aerospike.



---

# Data Modeling

## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

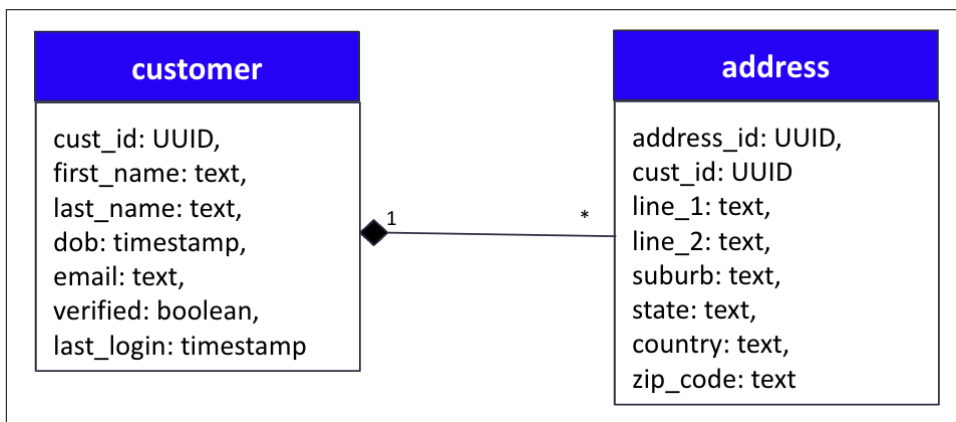
In the previous chapters, we looked at the Aerospike architecture and the facilities the database provides. In this chapter we will look at how to model data in Aerospike to solve common problems. As with most databases, there are a variety of techniques which can be used to solve the same problem so we will address some of the pros and cons of various approaches.

## Classical Data Modeling

Aerospike supports records which have a structure similar to a relational database and it supports secondary indexes similar to a relational database, so it would seem intuitive that data modeling techniques would be similar to those used in relational databases. However, this is not always the case. To illustrate this, let’s look at a classical way of aggregating data.

Suppose you want to model a Customer record. The Customer has zero or more Addresses associated with it, and the addresses have no business use if they are not associated with a customer. This is the classical case of aggregation.

This may be represented as shown in [Figure 5-1](#).



*Figure 5-1. Entity relationship model of a customer with their addresses*

In a relational model, you would create two distinct tables, customer and address, with a foreign key from the address to the customer. To preserve referential integrity, you would cascade any deletes from the customer table to the address table. This might be defined in PostgreSQL syntax similar to:

```
CREATE TABLE customer( cust_id UUID PRIMARY KEY, first_name TEXT, ...
    last_login TIMESTAMP);
CREATE TABLE address ( address_id UUID PRIMARY KEY,
    line_1 TEXT NOT NULL, line_2 TEXT, ...
    cust_id UUID REFERENCES customer(cust_id) ON DELETE CASCADE);
```

In order to get good performance when you want to load all the addresses for a customer, you should put an index on the foreign key of the address.

```
CREATE INDEX CUST_ID_IDX on ADDRESS(CUST_ID);
```

## Aerospike Data Modeling

Consider the same problem in Aerospike. You want to be able to load the addresses for a customer efficiently, and if the customer goes away you need the associated addresses to be removed too. There are a couple of ways of modeling this. Let's take a look at the pros and cons of each.



A quick reminder of some differences and equivalencies between Aerospike and most relational databases.

Relational DBs	Aerospike
Namespace	Database
Table	Set
Record	Record
Field	Bin

## Secondary Indexes

This problem could be modeled in Aerospike very similarly to the classical technique. You could have two sets, a Customer set and an Address set. The Address set could have a bin that contains the Customer id and a secondary index defined on the bin as shown in [Figure 5-2](#).

address							
address_id	cust_id	line_1	line_2	suburb	state	country	zip_code
<u>aefcb-231</u>	123bca	1 main st		Denver	CO	USA	80001

customer						
cust_id	first_name	last_name	dob	email	verified	last_login
123bca	John	Doe	12/11/71	jdoe@here.com	true	11/1/2023

Figure 5-2. Modeling an aggregation with a secondary index

There are a couple of problems with this approach:

1. Aerospike will not automatically cascade the deletion. When you remove the Customer object, the associated Address objects should be removed too. This cannot be achieved automatically with this data model. It should be fairly easy to do this with a secondary index coupled with an operation, but this would include the overhead of multiple operations. Example code would be:

```
String custToDelete = "123bca";
Statement stmt = new Statement();
stmt.setNamespace("test");
stmt.setSetName("address");
stmt.setFilter(null);
stmt.setFilter(Filter.equal("cust_id", custToDelete));
ExecuteTask task = client.execute(null, stmt, Opera-
tion.delete());
client.delete(null, new Key("test", "customer", custToDe-
```

```
lete));  
    task.waitTillComplete();
```

2. This is not a very efficient use of a secondary index. Remember that secondary index queries are performed via a “scatter gather” approach where all nodes are queried for matching records in parallel, then the results are streamed back to the client. This results in a decent amount of work for both the Aerospike client driver and the cluster. If there are only a handful of addresses and a large number of nodes in the cluster, many of the nodes will do work checking to see if they have any matching addresses without finding any.

Let’s take a look at a more efficient approach – aggregating the Addresses into the Customer

## Aggregating Sub-Objects into One Record

In this use case there are typically only a handful of addresses associated with a single Customer, and the addresses are typically small. Why not just store the addresses inside the Customer record itself?

Using this approach you would define an Addresses bin on the Customer record. This bin would contain a List and the List elements would be maps. Each map would hold one Address, with the map keys being the fields of the map (city, state, country, etc) and the values being the associated data. For example:

```
{  
  "cust_id": "123bca",  
  "first_name": "John",  
  "last_name": "Doe",  
  "dob": 1680714616850,  
  "email": "jdoe@here.com",  
  "verified": true,  
  "last_login": 1685764613895  
  "addresses": [  
    {  
      "line_1": "1 Main St",  
      "suburb": "Denver",  
      "state": "CO",  
      "country": "USA",  
      "Zip_code": 80001  
    }  
  ]  
}
```

This approach offers several advantages:

- Reading a Customer record automatically retrieves all the addresses for that Customer without needing to perform an additional query. Conversely, removing a Customer record automatically removes all the Addresses too.

- The Customer and the Addresses are accessed using Key Value operations, which are incredibly efficient in Aerospike.
- Addresses can still be directly manipulated or added / removed from the Customer using List and Map operations.
- The Address object is simplified. There's no longer any need for the Address to contain the Customer id – you must already know the Customer Id to access the Addresses. Also, there is no need for an Address Id on each Address – the place in the list could potentially serve as a “pseudo-Id” if needed.

If you're thinking that this will increase the size of the Customer record, you are correct. But the number of Addresses would normally be fairly small, meaning the overhead of this approach is typically low. In fact, the penalty for the larger objects might be smaller than you think. Remember that Aerospike typically uses SSDs to store information and most modern SSDs are “block-level” devices, meaning that data can only be read or written in blocks, rather than individual bytes. This block size is typically 4kB, so if your application asks an SSD for 1kB, the SSD will typically read 4kB and throw away 3kB. If adding the Addresses increases the size of the Customer record from say 2kB to 3kB, the cost of reading the record from storage will be exactly the same!

## Aggregating Sub-Objects into Multiple Records

The above approach works well if the number of sub-objects are known to be finitely bound to a small number. However, there are many use cases where this is not the case. Consider a credit card with the associated transactions. The number of transactions might add up to thousands per year for heavy credit card users, or significantly higher if the credit card is a corporate card.

A record for this CreditCard might look similar to this, with the primary key for the record being the Primary Account Number (PAN) which is typically a part of the credit card number:

```
{
  "cardNo": "1234123412341234",
  "cardSeqNo": 1,
  "custNo": "cust-1234",
  "expiry": 1680714616850,
  "opened": 1678912262162,
  "txns": [
    {
      "txnId": "txn1",
      "amount": 10000,
      "desc": "New car tire",
      "txnDate": 1680113831954
    },
    {
```

```

    "txnId": "txn2",
    "amount": 500,
    "desc": "Ice cream",
    "txnDate": 1699422631954
  }
]
}

```

In this case, it is not possible to store all of these transactions in a single record. As of Aerospike version 6.4, the largest record which can be stored is limited to 8MiB. Even if you could store everything in 8MiB, would you really want to? Aerospike always performs *copy on write* meaning that every time a record changes, the entire record will be re-written. So making one trivial change – such as adding a single Transaction – on an 8MiB CreditCard record means that the drive will read 8MiB and then 8MiB will be written back to the drive. This is a heavy load on the drive, and if operations such as this are common, the drives on the database nodes will likely run out of I/O capacity, slowing the database down.

A better solution to this problem is to separate the large number of children objects into finite “buckets” and store each bucket in a single record.

For the credit card example you could store the transactions in a Transaction set, and store all the transactions for a credit card separated into multiple records, one record per day. So instead of what’s shown in [Figure 5-3](#), the new Sets might look like [Figure 5-4](#).

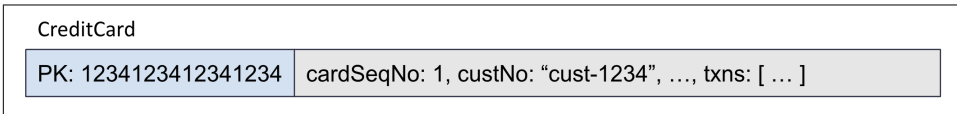


Figure 5-3. Modeling Transactions inside a CreditCard record

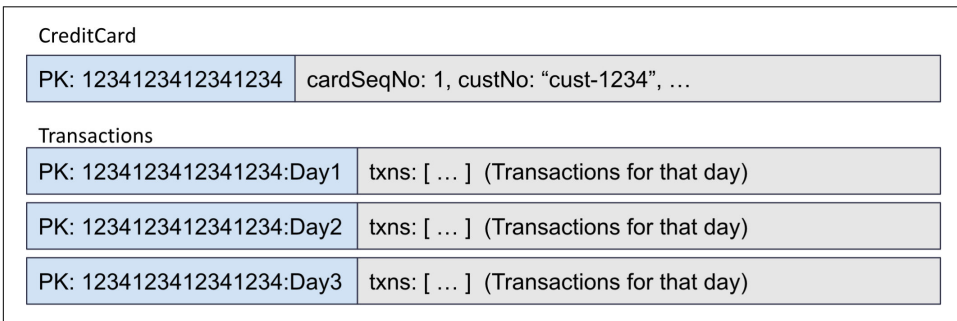


Figure 5-4. Modeling CreditCards outside of the Transaction object

In this new model, the CreditCard details become their own record. These typically don’t change very often so rewriting them whenever a transaction is added would be



inefficient, but would happen in the data model depicted in [Figure 5-3](#). Transactions have been separated into a single record per day and these records have compound keys containing both the credit card PAN and a day index from some offset.

Note that the offset day is not important so long as it's consistently used. For example, it might be set to 1/1/2010 as Day 0. The day offset can then easily be calculated as

```
(milliseconds(NOW) - milliseconds(OFFSET_DATE)) / TimeUnit.DAYS.toMillis(1);
```

Within the record for a particular day, there is a txns bin which stores the transactions for that day. There are various formats that could be used to store these, such as a List of Maps as was shown earlier in the chapter. In this case, we're going to actually change it up a bit and use a Map of Maps with the timestamp of the transaction as the map key, and the rest of the transaction contained as a map in the value. An example of this might be:

```
txns: {  
    1680113831954: {"amount": 10000, "desc": "New car tire", "txnId": "txn1"}  
    1699422631954: {"amount": 500, "desc": "Ice cream", "txnId": "txn2"}  
    1704423453345: {"amount": 2750, "desc": "Pizza", "txnId": "txn3"}  
    ...  
}
```

The reason for this change is that use cases on credit card transactions often call for operations based on time ranges, such as “retrieve all transactions between 3pm and 5pm on this date”. Having the timestamp as the transaction key makes these operations easier. Similarly, the map would likely be defined as KEY\_ORDERED so time-range based operations are more efficient. We will take a look at this in more depth later in the chapter.

Thus, a method to load the last thirty days of transactions for a credit card to run fraud detection checks could be written as:

```
private final static long EPOCH_TIME  
    = new GregorianCalendar(2010, 0, 1).getTime().getTime();  
  
private long calculateDaysSinceEpoch(Date date) {  
    return (date.getTime()- EPOCH_TIME) / TimeUnit.DAYS.toMillis(1);  
}  
  
public List<Transaction> readCreditCardTransactions(IAerospikeClient client, long  
cardId) {  
    long dayOffset = calculateDaysSinceEpoch(new Date());  
    Key[] keys = new Key[KEYS_TO_FETCH];  
    for (int i = 0; i < KEYS_TO_FETCH; i++) {  
        keys[i] = new Key(creditCardNamespace, SET_NAME,  
            "Pan-" + cardId + ":" + (dayOffset - i));  
    }  
    Record records[] = client.get(null, keys);  
    List<Transaction> txnList = new ArrayList<>();
```

```

for (int i = 0; i < DAYS_TO_FETCH; i++) {
    if (records[i] == null) {
        continue;
    }
    TreeMap<Long, Map<String, Object>> map =
        (TreeMap<Long,Map<String,Object>>)records[i].getMap(MAP_BIN);
    for (long txnDate : map.descendingKeySet()) {
        Map<String, Object> data = map.get(txnDate);
        Transaction txn = new Transaction();
        txn.setTxnId((String)data.get("txnId"));
        txn.setTxnDate(new Date(txnDate));
        txn.setAmount((long)data.get("amount"));
        txn.setDescription((String)data.get("desc"));
        txnList.add(txn);
    }
}
return txnList;
}

```

Let's take a look at what this method is doing.

First EPOCH\_TIME is defined to be the number of milliseconds since 1/1/2010. Then a method `calculateDaysSinceEpoch` simply calculates the number of days since the epoch. This uses 64-bit integer arithmetic so it's very efficient.

In the method `readCreditCardTransactions`, the first line calls the `calculateDaysSinceEpoch` method to calculate the offset of the current day. This will be a number something like 5,128, depending on the current date. Then an array of keys is formed with one key for each day in the desired range:

```

Key[] keys = new Key[DAYS_TO_FETCH];
for (int i = 0; i < DAYS_TO_FETCH; i++) {
    keys[i] = new Key(creditCardNamespace, SET_NAME,
        "Pan-" + cardId + ":" + (dayOffset - i));
}

```

Note that due to its use of constant hashing for record distribution Aerospike does not have any inbuilt support for compound keys like some other databases. However, as this example shows it is very simple to build your own, in this case a combination of a fixed string, the `cardId`, and the `dayOffset`. An example of a key generated by this section of code might be `Pan1234123412341234:5125`.

Once the array of keys has been formed, you can simply call

```
Record records[] = client.get(null, keys);
```

This call loads all the records passed in the `keys` array from the database and returns an array of records, one per key as discussed in Chapter 3. This call is typically very efficient, using parallelization across the cluster to get all the results very quickly.

Next, the program iterates through the returned records:

```
List<Transaction> txnList = new ArrayList<>();
for (int i = 0; i < DAYS_TO_FETCH; i++) {
    if (records[i] == null) {
        continue;
    }
}
```

It is possible that some of the returned records are null which will happen when no transactions were performed on a particular day. In this case there is no processing to be done so the day is just skipped.

Next, the method extracts the transaction data from the record:

```
TreeMap<Long, Map<String, Object>> map =
    (TreeMap<Long,Map<String,Object>>)records[i].getMap(MAP_BIN);
for (long txnDate : map.descendingKeySet()) {
    Map<String, Object> data = map.get(txnDate);
    Transaction txn = new Transaction();
    txn.setTxnId((String)data.get("txnId"));
}
```

In this data model, the transactions are stored in a Map on the server. However, the Map on the server is defined as a KEY\_ORDERED map so they are ordered by the transaction timestamp. Maps in Java are inherently unordered, so to preserve this order the Aerospike Java driver returns a TreeMap – an ordered Map. This allows the transactions to be processed in chronological order. This would be very convenient if the use case called for only returning the 1,000 most recent transactions over the thirty days for example.

All that remains is to unpack each transaction into a Transaction object which the application understands. This is as simple as pulling each field out of the map and setting it on the transaction object. This should be fairly familiar if you have unpacked objects using a JDBC driver in Java for a relational database for example.

### Additional operations on this model

The map structure allows the records to be queried efficiently by their key, which is conveniently a number representing a date. Aerospike's MapOperation class allows us to perform operations within the map. As a reminder, the data in the map looks like:

```
txns: {
  1680113831954: {"amount": 10000, "desc": "New car tire", "txnId": "txn1"}
  1699422631954: {"amount": 500, "desc": "Ice cream", "txnId": "txn2"}
  1704423453345: {"amount": 2750, "desc": "Pizza", "txnId": "txn3"}
  ...
}
```

The code in the section above simply gets all the elements of the map and converts them into Transaction objects. But what if you don't want the whole map? For

example, you might just want the transactions between 3pm and 5pm on that day. In other words you might want a range of keys to be returned from the map rather than the full map. This could be achieved using

```
Record record = client.operate(null, key, MapOperation.getByKeyRange(
    "txns",
    Value.get(startTime),
    Value.get(endTime),
    MapReturnType.KEY_VALUE));
```

In this case, the `startTime` would be 3pm of the day in question (converted to a `long`) and `endTime` would be 5pm on the same day. These long values need to be converted into a `Value` class in order to be passed to the `getByKeyRange` method, hence the use of the `Value.get(...)` methods. The name of the bin holding the map is “txns” and you want both the key and the value to be returned so the whole transaction can be re-assembled.

As discussed in Chapter 4, the `operate` command always returns a `Record`, but the type of the value being returned for the bin being operated on depends on the operation. For example, if you had just wanted the count of the transactions between 3pm and 5pm, you could have passed `MapReturnType.COUNT` instead of `MapReturnType.KEY_VALUE`. Then the count could have been retrieved by calling

```
int count = record.getInt("txns");
```

However, in this case you are getting both the key and the value back, so you can re-assemble the transaction. So in this case Aerospike will return a `List` of `AbstractMap.SimpleEntry`.

This might seem odd. Why would Aerospike Client return a `List` when you selected from a `Map` in the database? Well, remember that Aerospike Maps can be sorted (as in this example) and we’re selecting a range of keys from that sorted map. Java maps have no concept of ordering so the Aerospike Client returns the items in the order you requested as an ordered list.

To use this, you could do something like:

```
List<SimpleEntry<Long, Map<String, Object>>> entries =
    (List<SimpleEntry<Long, Map<String, Object>>>)record.getList(MAP_BIN);
for (SimpleEntry<Long, Map<String, Object>> simpleEntry : entries) {
    Long txnTime = simpleEntry.getKey();
    Date txnDate = new Date(txnTime);
    Map<String, Object> txnDetails = simpleEntry.getValue();
    System.out.printf("%tH:%tM:%tS:%s\n",
        txnDate, txnDate, txnDate, txnDetails);
}
```

This will output something similar to:

```
13:49:14:{amount=74, desc=desc-1, txnid=txn-74}
16:18:14:{amount=26, desc=desc-1, txnid=txn-26}
16:35:33:{amount=40, desc=desc-1, txnid=txn-40}
```

Obviously, if you're using real data instead of generated data, the transaction values will be more meaningful.

## Further refinements

The above data model is good, but we could potentially make it better. Consider one of the transactions we have stored in the map:

```
1680113831954: {"amount": 10000, "desc": "New car tire", "txnId": "txn1"}
```

Since the transaction date is the map key, you can use the `MapOperations` like `getByKeyRange`, `getByKeyList`, and so on to perform operations on the data. However, there is a lot of redundancy here as each transaction for a given day contains the same descriptions of the data: `amount`, `desc` and `txnId`. This will occupy a lot of space in the database – in fact those descriptions will be longer than the actual data being stored!

One way around this is to store the actual data in a list instead of a map. So using a data format like:

```
txns: {
  1680113831954: [10000, "New car tire", "txn1"],
  1699422631954: [500, "Ice cream", "txn2"],
  1704423453345: [2750, "Pizza", "txn3"],
  ...
}
```

The only difference between inserting the data as a list or inserting it as a map is how the value is constructed:

```
List<Object> transactionAsList = Arrays.asList(
    transaction.getAmount(),
    transaction.getDescription(),
    transaction.getTxnId());

client.operate(null, key,
    MapOperation.put(mapPolicy, MAP_BIN,
    Value.get(transaction.getTxnDate().getTime()),
    Value.get(transactionAsList)));
```

This format now saves substantial space on the drive, yielding better performance as it needs to read and write less data, send less data across the network, and more transactions can fit within the same record. However, the format is harder to read. While it's not difficult in this simple example, imagine an object stored in a list

which had twenty integer values. If these integers represent different things like dates, transaction amounts, customer ids and so on, discerning which integer means what is difficult. If however the integers are the same thing, like components of a vector, then a list is the natural structure for them.

There is one other advantage of using a List in this scenario. Lists, unlike Maps, have an inherent ordering which can be useful in some use cases.

**List ordering.** When Aerospike compares two lists to see if they're equal or one is greater than the other, it uses these rules:

1. Compare each element in the list from the first element in the list to the last. If the element being compared is greater than the corresponding element in the other list, this list is greater. If the elements are equal, move onto the next element.
2. Once one list has no more elements, look at the size of the other list. If it is greater than this list, the other list is greater. Otherwise, both lists are equal.

These rules make more sense with examples.

- List [1,2] is less than [1,3] because the first element in both lists are equal, but  $2 < 3$ .
- Similarly, [1,2,9] is less than [1,3,1] because the second element in the first list is less than the second element in the second list. The fact that the third element in the first list is greater than the third element in the second list is irrelevant.
- List [1,2,3] is greater than list [1,2] because they both contain the same first two elements, but the first list is longer than the second list.
- The other thing to remember with ordering in Aerospike is that types have a predefined order which you can see [here](#), but comparing two items of different types will always result in the same outcome, regardless of the value of that type. To illustrate with an example, Integers are always less than Strings which are always less than Lists. So 27 is less than "25" which is less than [1], simply by virtue of the types.

**Ordering by value.** Thus far, you have learned about operations that act on the keys. These operations are typically very efficient especially for maps that are defined as `KEY_ORDERED` or `KEY_VALUE_ORDERED` as you learned in chapter four. However, Aerospike also supports operations on the map values as well as the map keys.

Consider a use case where you want to be able to operate on the timestamp of the transactions, as in the earlier section, but also want to be able to get transactions that meet a certain dollar amount threshold. So you want to be able to answer "get any transactions whose dollar amount is greater than or equal to \$50,000" for example.

If you look closely at the structure of the data above, you will see that the dollar amount is the first value in the list of transaction details:

```
txns: {  
  1680113831954: [10000, "New car tire", "txn1"],  
  1699422631954: [500, "Ice cream", "txn2"],  
  1704423453345: [2750, "Pizza", "txn3"],  
  ...  
}
```

This means that the “Value” type operations will act on the transaction amount first. So to satisfy the query shown above you could do something like:

```
Record record = client.operate(null, key,  
  MapOperation.getByValueRange(  
    "txns",  
    Value.get(Arrays.asList(50000)),  
    Value.get(Arrays.asList(Long.MAX_VALUE)),  
    MapReturnType.KEY_VALUE));
```

This is very similar to the `getByKeyRange` call you saw earlier except that it’s now operating on the value instead of the key. This will return the keys and values of the records whose amount is greater than the passed values.

Did you notice that the code didn’t just pass `Value.get(50000)` as the starting value but rather `Value.get(Arrays.asList(50000))`? This is necessary because the value is a list. If you were just to pass `Value.get(50000)`, Aerospike would look at the value you passed (an integer) and compare it to its stored value (a list). Integers are always considered less than a list, so both the start and end values would be less than every list, resulting in no transactions being returned.

However, since the code wrapped the 50000 in a Java list, Aerospike will compare two lists. As mentioned in “List Ordering” above, this will cause Aerospike to look at each item in the list in turn, starting with the first one, the amount. This will yield the desired result.

## Associating Objects

We have seen the power of embedding aggregated objects into the parent object, and this is a very common use case. Another common use case is having two top-level objects which refer to one another. For example, a Customer may have multiple Accounts, and Accounts may have multiple Customers. Both are top-level entities that have business value in their own right.

The most common way of handling this is for the Customer object to have a list of Account ids, and the Accounts object to have a list of the Customer ids. [Figure 5-5](#) shows an example of this. The Customer IDs associated with that Account are placed

in the Account's customer id list, and the id of the Accounts are placed in the Customer's Account id list.

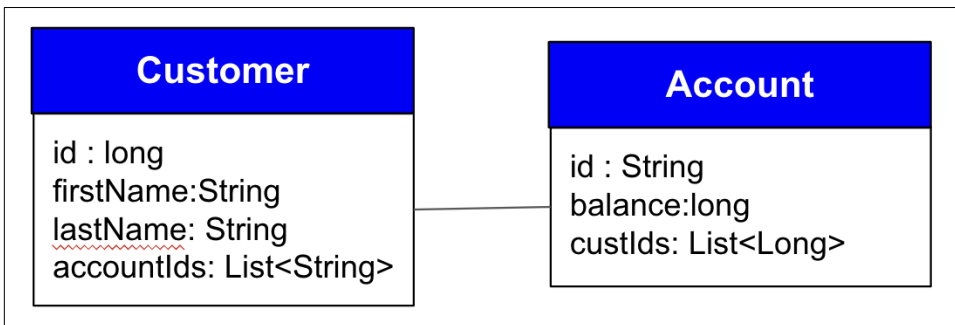


Figure 5-5. Simple relationship between Customers and Accounts

This pattern facilitates navigation from either the Account to the Customer or from the Customer to the Account. If a use case does not require this bi-directional navigation, but the Accounts are always looked up from the Customer for example, only one list needs to be maintained (the Account IDs in the Customer in this case).

### Updating relationships

Here is a simple example of creating a relationship between a Customer and an Account. This assumes that both objects have been saved into Aerospike and this code merely links between them in the database.

```
ListPolicy listPolicy = new ListPolicy(
    ListOrder.UNORDERED,
    ListWriteFlags.ADD_UNIQUE | ListWriteFlags.NO_FAIL);

// Put the ID of the account into the customer id list, ignoring duplicates
client.operate(null, new Key(NAMESPACE, CUST_SET, customer.getId()),
    ListOperation.append(listPolicy, "accountIds",
        Value.get(account.getId())));

// Put the ID of the customer into the account id list, ignoring duplicates
client.operate(null, new Key(NAMESPACE, ACCOUNT_SET, account.getId()),
    ListOperation.append(listPolicy, "custIds",
        Value.get(customer.getId())));
```

The code first sets up a `ListPolicy` to be used by the operations. It's important to ensure that the list of ids contains no duplicates, even if the same account is linked to the same customer multiple times. The `ListWriteFlags.ADD_UNIQUE` flag is used to do this, which will ensure that Aerospike will only put the item in the list if it's not already there. If the item already exists in the list an exception will be thrown. However, in this case, it's desirable to not throw an exception since that is exactly the



behavior we want. Instead, it should fail silently, so the `ListWriteFlags.NO_FAIL` flag is included.

Once this `ListPolicy` has been set up, the id of the account is placed in the `accountIds` list in the customer record. Since the behavior of the append has been set up by creating the `ListPolicy`, all you have to do is invoke `append` with the appropriate arguments.

Finally, the customer id is placed in the `custIds` list on the account object, very similar to the way the `accountIds` were.

Some sample data in the database might look like:

```
aql> select * from test.customers
+-----+-----+-----+
| firstName | lastName | accountIds |
+-----+-----+-----+
| "Fred"    | "Black"  | LIST(['ACCT-2']) |
| "Bob"     | "Smith"  | LIST(['ACCT-2', "ACCT-5", "ACCT-6"]) |
...
+-----+-----+-----+
aql> select * from test.accts
+-----+-----+
| balance | custIds |
+-----+-----+
| 600     | LIST(['5']) |
| 200     | LIST(['2, 3, 5']) |
...
+-----+-----+
```

Note that there are two distinct operations here, one on the customer and one on the account. These records are independent records so there is no transactionality across the two operations, potentially leading to an inconsistent database state if errors occur. At the time of writing, current versions of Aerospike only have transactionality on a single record, but in some cases ordering operations appropriately can work around this. In this example however, this is not possible. However, Aerospike version 7.1 is anticipated to have multi-record transactions, which will solve this problem.

### Reading related objects

Now that you've got relationships in the database between the objects, you need to be able to read the data back. Note that Aerospike does not support joins like a relational database does, so you will need to use a couple of operations to retrieve the data.

This sample will assume that you have a customer id and you want to get all the accounts associated with that customer.

```

public List<Account> getAccountsForCustomer(long customerId) {
    List<Account> results = new ArrayList<>();
    Record custRecord = client.get(null,
        new Key(NAMESPACE, CUST_SET, customer.getId(), "accountIds"));
    if (custRecord == null) {
        return results;
    }
    List<String> accountIds = (List<String>)
        custRecord.getList("accountIds");
    Key[] keys = new Key[accountIds.size()];
    for (int i = 0; i < accountIds.size(); i++) {
        keys[i] = new Key(NAMESPACE, ACCOUNT_SET, accountIds.get(i));
    }
    Record[] accounts = client.get(null, keys);
    for (int i = 0; i < accountIds.size(); i++) {
        Record account = accounts[i];
        results.add(new Account(accountIds.get(i),
            account.getLong("balance"),
            (List<Long>)account.getList("custIds")));
    }
    return results;
}

```

This code can be broken down into several steps. First, you need to get the list of account ids from the customer record by performing a simple `get` on the customer record. However, you only need the list of `accountIds` not the full customer record so this bin is listed as the only bin to be retrieved from the `get`:

```

Record custRecord = client.get(null,
    new Key(NAMESPACE, CUST_SET, customer.getId(), "accountIds"));

```

Aerospike will still read the full customer record from storage, but will transmit only the one bin that is needed back to the application.

You can omit this step if you already have the full `Customer` object loaded, as you already have the list of `accountIds`.

Next, you need to iterate through the list of `accountIds`, forming a key to the `Account` record for each of the ids:

```

Key[] keys = new Key[accountIds.size()];
for (int i = 0; i < accountIds.size(); i++) {
    keys[i] = new Key(NAMESPACE, ACCOUNT_SET, accountIds.get(i));
}

```

This gives an array of keys so all that remains is to perform a batch `get` on these keys and iterate through the results, turning each Aerospike `Record` into an appropriate `Account`:

```

Record[] accounts = client.get(null, keys);
for (int i = 0; i < accountIds.size(); i++) {

```

```
Record account = accounts[i];
results.add(new Account(accountIds.get(i),
    account.getLong("balance"),
    (List<Long>)account.getList("custIds")));
}
```

This is a very quick and easy way to effectively resolve a one-to-many relationship without needing to do a join.

### Storing of ids

If you look closely at the code where you turn the list of Aerospike records back into Account objects in the previous section, you will notice that the account balance and list of custIds come from the Aerospike record, but the id of the account comes from the list of the account ids retrieved from the Customer record. Why not just get it out of the record like the other fields? The answer is because, by default, Aerospike does not store the primary key for a record!

If you're used to relational databases, this seems like an astounding statement. Surely every record in the database needs to know its primary key, otherwise how can the database select the correct record?

Well, as was mentioned in Chapter 5, the storage and retrieval of the records in Aerospike is done by the digest, the 20 byte hash of the set name and primary key. Aerospike stores the digest for each record in lieu of the actual primary key. These digests are unique so they can serve as a primary key.

When you think about the applications you've written, how many times have you actually needed the database to tell you what the primary key of a record is, except for maybe using them in joins? Almost all the time, you know the primary key of the record and you pass this primary key to the database when reading it. Consider the example above, which starts with a customer id. How would you discover the customer id if it's not stored in the database?

Most applications would allow the user to search for a customer by name, date of birth or some other criteria. Hence the person using the application would load the Customer object by selecting from a list of customers or similar. This selection of the customer would be done using the digest, which is returned when forming the list. If the user does provide a customerId this can be used to retrieve the record directly.

In fact, the number of use cases where you actually need to store the id of a record is very small. Most of the time you already have the id to retrieve the object, as the code above did when loading the accounts.

What if you do need to store the id? Well, there are 2 options for doing so:

Manually store the id explicitly in its own bin.

Tell Aerospike to store the id for you.

The first option is the simplest, but you have to store and read an extra bin.

The second option requires setting a policy value whenever you write the record, passing `sendKey = true`. For example:

```
WritePolicy writePolicy = new WritePolicy(client.getWritePolicyDefault());
writePolicy.sendKey = true;
client.put(writePolicy, customer.getKey(),
           new Bin("firstName", customer.getFirstName()),
           new Bin("lastName", customer.getLastName()),
           new Bin("accountIds", customer.getAccountIds()));
```

This will pass the primary key as well as the digest to the Aerospike server. Aerospike will store the key associated with the record, but will also perform an additional check to ensure that the id of the value stored associated with the digest of the record is the same as the passed key. So if you're worried about hash collisions on the digest, this method safeguards against it. Note that using `sendKey = true` will force a read of the record from storage to retrieve the actual primary key, potentially introducing additional latency.

You can use `aql` to see what happens when you send the key:

```
aql> set key_send false
KEY_SEND = false
aql> insert into test.newSet(pk, value) values (1, 1)
OK, 1 record affected.
```

```
aql> select * from test.newSet
+-----+
| value |
+-----+
| 1     |
+-----+
1 row in set (0.173 secs)
```

OK

```
aql> set key_send true
KEY_SEND = true
aql> insert into test.newSet(pk, value) values (1, 1)
OK, 1 record affected.
```

```
aql> select * from test.newSet
+----+-----+
| PK | value |
+----+-----+
| 1  | 1     |
+----+-----+
1 row in set (0.154 secs)
```

OK



Aql uses `key_send` to send the key, Java uses `sendKey`- same idea but different naming. Also, Aql defaults `key_send` to `true`, but the Aerospike clients default it to `false`.

As you can see, when we set `key_send` to `false`, Aerospike can only return the value `bin` even though all columns were requested. Once `key_send` is set to `true` and the record re-inserted, Aerospike can return the key of the record.

## Other Common Data Modeling Problems

Let's turn to a set of problems that crop up regularly and look at solutions to them. As with many data modeling problems, there can be more than one correct solution and each use case tends to put its own nuances on the problem.

### External id resolution

Companies frequently share information these days and it is not uncommon for a record created with internal data to be augmented with data from external sources. This is particularly prevalent in the Advertising Technology vertical for example. These external data sources can also request the information back, augmented with information from the internal source too.

For example, Company A stores a list of topics they think a person may be interested in, called "audience segments". So they might know that Bob is interested in "sports" for example, and have a record with key 1234 holding this information. They then receive some audience segmentation data from Company B, and they work out that this data also contains audience segments for Bob. However, the key that Company B uses for Bob is 6789. Company A merges the audience segments from Company B into its record of 1234, but they have to know that when Company B asks for 6789, they have to return the record they know as 1234.

So the internal Id of 1234 sometimes needs to be discovered using an external id of 6789. The obvious way to do this is using a secondary index with a structure like:

PK	external_ID	data
1234	6789	{...}

A secondary index on `external_id` could be defined so that when the external id is provided, a secondary index query is performed, looking for the record that matches the passed id.

However, this is not a good use of secondary indexes. The external id is assumed to be unique so looking up data matching the external id will yield either no records or one record. Remember that secondary index queries are “scatter-gather” so a request will be sent to every node in the cluster, looking for matching records on that node. The client has to coordinate requests to multiple server nodes and every server node will do work looking for that external id, with potentially only one finding the record.

Worse, as the cluster scales, say from ten nodes to twenty nodes, the amount of work the cluster performs increases (doubles in this case) whilst returning exactly the same result. This is *inversely scalable* – definitely not ideal.

A better way of solving this problem is simple: have one set which maps external ids to internal ids.

PK	internal_ID
6789	1234

And another set which has the internal id mapped to the data.

PK	data
1234	{...}

When an external id comes into the system, two key-value reads are performed: one to read the external id and retrieve the internal id, and the other one to read the data associated with that id. So the code might resemble:

```
Record record = client.get(null, new Key("test", "extId", 6789));
if (record != null) {
    record = client.get(null, new Key("test", "data",
        record.getLong("internal_id")));
}
```

Key value reads are incredibly efficient in Aerospike, so these two reads will likely be finished in a millisecond or two depending on the hardware the cluster runs on.

If you're used to relational databases, performing two simple operations is typically frowned upon, preferring one more complex operation. However, with Aerospike, simpler operations are so quick and efficient that it is a very common technique.

### The very small object problem

The above technique with two key-value pairs is very efficient and solves the internal/external index problem very nicely. However, it does suffer from one drawback: Each copy of each record in Aerospike requires 64 bytes for the primary index, which is normally held in memory (DRAM). For larger records this is typically not an issue,

but here our objects contain just two ids, meaning that the data is possibly even smaller than the DRAM needed for the primary index.

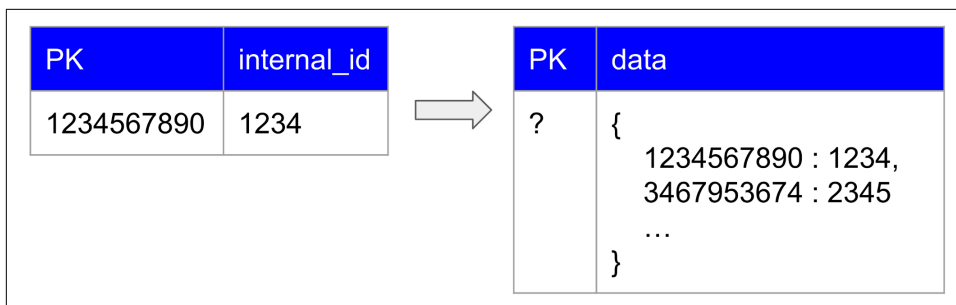
One way around this is to merge multiple records into a single record using a map. This assumes you have some knowledge about the keys. For the sake of this example, assume the external ids are 10 digit numbers, randomly distributed over the whole range, and there are a billion of these external ids.

If each external id to internal id is mapped as a single record and Aerospike has been configured to store two copies of each record, the DRAM requirements across the cluster will be:

64 bytes x 2 copies x 1,000,000,000 records  $\approx$  128GB

128GB across the cluster might not seem like much on modern machines, but remember this is just for one small component of this use case. This lowers the system resources left for other database records.

Merging multiple of these small records into one big record is easy. Diagrammatically, what you want to do is shown in [Figure 5-6](#).



*Figure 5-6. Merging internal/External ID key values into a single map bin.*

So the ten digit external id goes from being the primary key of a record to being a map key, with the internal id being the map value. There will be multiple records containing these maps as a billion entries will not fit into a single map, so the question is: given an external id, what primary key should be used to store or read that external id from the map?

A good way of working this out is deciding approximately how big you want the records holding the maps. Normally a few kilobytes is a good size; it's not too heavy on the SSDs when being read or written and gives a reasonable ratio of primary index DRAM (64 bytes) to actual storage. In this example, the combination of the external id and internal id is small, potentially about fifty bytes.

So, if the desired record size was about four kilobytes and each record is fifty bytes, that would lead to an optimal size of (4,000 / 50) entries in each map record, or

around eighty. However, it doesn't matter if it's a bit more or bit less than this; it's an approximate order of magnitude.

What if you took the external id and divided it by a thousand? Would this form a good unique primary key? The worst case is there are a thousand entries in one of the maps, but the external ids are random so this is very unlikely to occur. There are ten billion possible numbers and one billion of them have been used, so on average each map will hold around a hundred entries – one tenth of the thousand we divided the external id by.

This is right in the correct range and gives a very simple algorithm to save and retrieve these ids:

```
private Key getKey(long externalId) {
    return new Key("test", "mapping", externalId / 1000);
}

public void saveMapping(long externalId, long internalId) {
    client.operate(null, getKey(externalId),
        MapOperation.put(MapPolicy.Default, "map",
            Value.get(externalId), Value.get(internalId)));
}

public long getInternalId(long externalId) {
    Record record = client.operate(null, getKey(externalId),
        MapOperation.getByKey("map",
            Value.get(externalId), MapReturnType.VALUE));
    if (record != null) {
        return record.getLong("map");
    }
    return 0;
}
```

The key is determined by taking the external id and dividing by a thousand. To save a mapping from an external id to an internal id, the `MapOperation.put(...)` operation is used, and to retrieve the internal id from an external id the `MapOperation.get(...)` operation is used.

This algorithm works well and is simple to use and understand. There is one refinement we could make. Every map key in one map is identical except for the last three digits. We know this because the record key is the external id (the map key) divided by a thousand. Instead of storing the whole external id in the map each time, you could just store the last three digits. This would optimize space in the database, and the only changes to the above code would be changing both occurrences of `Value.get(externalId)` to `Value.get(externalId%1000)`.

What if your keys aren't numbers or their randomness is not great? Then hash the keys using a good hashing algorithm like RIPEMD160 or similar, and take a set of bits from the hash which gives a number greater than the anticipated number of keys.



## Expiry of map entries

The last problem to examine is common in several verticals: information is stored in a map and the information has an expiration date after which that information is not useful and can be removed. Aerospike supports automatic removal of records based on a time-to-live, but not automatic expiry of map entries.

To develop expiry of map entries, you will need a map which contains the information, including the expiry time of the information in the map entry. The value in the map will be a list containing this expiry time as the first item in the list followed by other pertinent information. For example, an ad-tech use case is audience segmentation, as discussed before. For a particular device (phone, tablet, etc) a list of segments (interests) is stored. These segments show topics that you may have browsed recently and hence might be interested in finding out more information about, meaning that showing you an ad related to these segments might be more impactful.

Any segments not seen in the last thirty days are considered irrelevant to your current interests, and should be removed from the map. So a sample record might look like:

```
segments: {
  "CRICKET": [1680243671492, "google.com"],
  "DATABASE": [1680243859728, "aerospike.com"],
  "KUNG_FU": [1680243656968, "shaolin.com"]
}
```

In this case there are three segments in the map for this device. Each segment has a name (the map key), an expiry time (first element in the list of values) and an originating source – the web site which reported the interest in the segment. This is obviously a simplified example but serves for illustrative purposes.

Three operations need to be performed on this data:

1. Retrieving any segments that have not expired (so the expiry date is in the future).
2. Adding a new segment with its expiry date.
3. Removing any segments that have expired.

**Retrieve any active segments.** This is easy – you want to get any elements from the map whose expiry time is in the future. The `getByValueRange` operation can be used for this, similar to how it was used earlier in the chapter to get the list of credit card transactions which were active:

```
Date now = new Date().getTime();
Record record = client.operate(writePolicy, key,
    MapOperation.getByValueRange("segments",
        Value.get(Arrays.asList(now)), Value.INFINITY,
        MapReturnType.KEY));
```

The parameters to the `getByValueRange` call are the bin name (`segments`), the start time (`now`, but as a list), the end time (`INFINITY`) and the information to be returned, in this case just the key (the segment name). If a segment is valid, the expiry time will be in the future and these parameters select these entries. For this dataset, the result will be a `List` containing [`CRICKET`, `DATABASE`, `KUNG_FU`]

**Adding new segments and removing expired segments.** Adding a new segment just requires forming the list containing the expiry date and the values and inserting the entries into the map. This is the code to do that:

```
long now = new Date().getTime();
long expiryMs = now + MS_IN_30_DAYS; // Add 30 days
List<Object> data = Arrays.asList(expiryMs, "pi.com");
client.operate(writePolicy, key,
    MapOperation.removeByValueRange("segments",
        Value.get(Arrays.asList(0)),
        Value.get(Arrays.asList(now)),
        MapReturnType.NONE),
    MapOperation.put(MapPolicy.Default, "segments",
        Value.get("ELECTRONICS"), Value.get(data)));
```

Note that there are two operations in the `operate` call: one to add the new entry (a list of `Objects`) into the map with `MapOperation.put` and another to remove any expired segments with `MapOperation.removeByValueRange`. The `removeByValueRange` is done first, throwing away any elements whose timestamp is in the range of zero to the current time. As the timestamp being stored is an expiry epoch, having an expiry time in the past means that entry is expired.

Once the expired values have been removed, the `put` places the new values into the map. Note that if the segment already exists in the map, the new value will overwrite the old ones, updating the expiry time to the new time as would be expected in this use case.

The removal of the elements being done at the same time as adding a new element is fairly efficient. The record has been read from storage and will need to be written back to storage anyway, so removing the extra elements incurs no extra storage accesses. However, the `removeByValueRange` will require the map to be sorted if it has been set up as a `KEY_ORDERED` map, which does have some CPU cost.

If you're calling this method at a moderate frequency, this extra CPU cost is unlikely to make a difference. Aerospike is normally very light on CPU so the database nodes normally have spare CPU cycles. However, if you're calling this method very quickly, this might become an issue. If it does there are two simple solutions:

1. If you're using version 7 or above of Aerospike the map can be set up as `KEY_VALUE_ORDERED` instead of `KEY_ORDERED` and an option selected which will

save the value ordered index to storage along with the record. This will remove the CPU cost of a full sort, but at the cost of extra storage space for the index.

2. Instead of performing the `removeByValue` range on every call, defer it to a background query operation running at a controlled requests per second rate, limiting the impact on the cluster. If the map contains some entries which have expired for an extra period of time, it will not impact the functionality of the other operations.

## Conclusion

Data modeling is crucial to efficiently using any database and Aerospike is no exception. You must take care to ensure the implementation of a good data model that solves the business problem. The right data model should not only perform well, it should typically also reduce the amount of hardware required to solve the problem.

Like most data modeling problems, there are multiple ways to solve the same problem. The techniques in this chapter have given you a brief introduction to some ways of solving the business problem, but they are certainly not comprehensive. A comprehensive look at data modeling in Aerospike would take a whole book all by itself, but hopefully these building blocks give you enough to go on to begin modeling for your use cases.



---

# Administration, Tools, and Configuration

## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

Now that you’re ready to bring an Aerospike cluster online, you need to know how to keep the system online and deal with changes. You should understand processes used to interact with the server so that you can manage it as part of your production environment. You will need to know how to cluster servers together, tune parameters, and manage user access. This chapter focuses on these essential aspects of administering Aerospike, exploring the tools and configurations to manage your Aerospike clusters.

## Configuration

With your Aerospike installation a configuration file should have been created at `/etc/aerospike/aerospike.conf`. This is the main configuration file and is only read when the daemon is started. The format of this file does not adhere to any specific configuration formats such as `yaml`, `json`, or even `INI`. The format is proprietary and specific to Aerospike, though it resembles `JSON`.

## Anatomy of a config file

Let's review the config file included with your installation to understand it, breaking it down into sections.

The Aerospike config file is divided into various sections known as *contexts* and *sub contexts*. Comments can be added to this file with the # symbol. You can use the [Aerospike Configuration Reference](#) to determine what each parameter does, additional notes, and properties of that configuration parameter which will be discussed soon.



We strongly recommend that you bookmark the [Aerospike Configuration reference guide](#). It's an invaluable resource for interpreting the book and will prove useful in your day-to-day activities with Aerospike, as well as for infrequent reference needs. The config reference guide will be your go-to source for navigating and understanding Aerospike's configurations effectively.

Let's start by examining a skeleton of the config file with no values (Example 7-1). This skeleton is provided directly from Aerospike's [configuration documentation](#) and briefly explains each context section. You should not copy this file as it is not a working example.

### *Example 6-1. The Aerospike server configuration file skeleton*

```
## Ini
service {} # Tuning parameters, process owner, feature
# key file

network { # Used to configure intracluster and
# application-node communications
service {} # Tools/Application communications protocol
fabric {} # Intracluster communications protocol
info {} # Administrator telnet console protocol
heartbeat {} # Cluster formation protocol
}

security { # (Optional, Enterprise Edition only) to enable
# access control on the cluster
}

logging {} # Logging configuration

xdr { # (Optional, Enterprise Edition only) Configure
# Cross-Datacenter Replication
}

namespace { # Define namespace record policies and storage
# engine
```

```

storage {} # Configure persistence or lack of persistence
set {} # (Optional) Set specific record policies
}

mod-lua { # location of UDF modules
}

```

The skeleton is useful as it helps show the overall structure of the configuration file, but since it isn't an actual configuration file, it's hard to conceptually think about what should fill out the empty sections.

Let's examine a real configuration file that ships with an Aerospike installation. For this example, the enterprise server package for Aerospike v6.3 is downloaded to target the Ubuntu operating system (See example 7-2). The config file should rarely change between Aerospike versions and Linux distributions, so this example should be virtually identical, regardless of your operating system or version.

*Example 6-2. Default Aerospike configuration file*

```

## Ini
# Aerospike database configuration file for use with systemd.

service {
    proto-fd-max 15000
}

logging {
    console {
        context any info
    }
}

network {
    service {
        address any
        port 3000
    }

    heartbeat {
        mode multicast
        multicast-group 239.1.99.222
        port 9918
        # To use unicast-mesh heartbeats, remove the 3 lines above, and see
        # aerospike_mesh.conf for alternative.
        interval 150
        timeout 10
    }

    fabric {
        port 3001
    }
}

```

```

    info {
        port 3003
    }
}

namespace test {
    replication-factor 2
    memory-size 4G
    storage-engine memory
}

namespace bar {
    replication-factor 2
    memory-size 4G
    storage-engine memory
    # To use file storage backing, comment out the line above and use the
    # following lines instead.
#    storage-engine device {
#        file /opt/aerospike/data/bar.dat
#        filesize 16G
#        data-in-memory true # Store data in memory in addition to file.
#    }
}

```

## Service Context

The first context in the config file is the ‘service’ context. The service context allows you to specify settings for high-level daemon-related runtime parameters that you can find using the configuration reference. You can safely assume the default values in the file are reasonable for most deployments to start, and change them later if needed.

```

    service {
        proto-fd-max 15000
    }

```

The first value you’ll see is a configuration parameter called ‘proto-fd-max’ having a value of 15000. proto-fd-max means the maximum number of open file descriptors opened on behalf of client connections. Refer to the [Configuration Reference guide](#) for more information.

## Logging Context

The second setting is for the vital logging context. This controls where Aerospike stores logs.

```

    logging {
        console {
            context any info
        }
    }

```



By default logging messages return to the console. Console logs can be retrieved using `docker logs` command if you are using Aerospike with docker. If not, use

```
journalctl -u aerospike.service
```

The major customizations you can change in the logging context are:

- logging to a file additionally, or instead of, the console
- using the syslog
- splitting specific messages into separate log locations

## Network Context

Following Logging is the Network context with subcontexts of Service, Heartbeat, Fabric, and Info.

```
network {
    service {
        address any
        port 3000
    }

    heartbeat {
        mode multicast
        multicast-group 239.1.99.222
        port 9918
        interval 150
        timeout 10
    }

    fabric {
        port 3001
    }

    info {
        port 3003
    }
}
```

Under the Network context, the Service context shows that Aerospike will be listening on all IP addresses on the machine, and that clients will connect on port 3000.

The Heartbeat context here is how you configure Aerospike's cluster strategy. In this example, multicast is being used where each server sends a broadcast to form a cluster within the multicast group. The broadcast goes to 239.1.99.222 on port 9918. The 'interval' and 'timeout' parameters control the threshold which Aerospike will consider a server to be unreachable (See Chapter 5 for more details on how this works.) In the example, a timeout value of 10 means that 10 heartbeats will have to fail consecutively before a server is considered to be no longer part of the cluster. The

heartbeat is sent on a schedule controlled by the interval setting, which is 150 milliseconds in this example. This means that if a server becomes unreachable for some reason, the remaining nodes in the cluster will take  $\text{timeout} \times \text{interval}$ ,  $150\text{ms} \times 10$  or about 1.5 seconds to recognize that.

In some environments, especially cloud, multicast is not available. Consider using Mesh mode for clustering on platforms where multicast is unavailable or not desired. You can follow the guide [on configuring heartbeat](#) in the Aerospike documentation. The entire heartbeat section will look similar to the following, where the mode is set to 'mesh' and a list of IPs to cluster to are provided. It is unnecessary to specify all nodes in the cluster, only enough to have a reasonable guarantee one of them will be online during start-up.

```
heartbeat {
  mode mesh
  mesh-seed-address-port 10.0.0.123 3002
  mesh-seed-address-port 10.0.0.124 3002
  mesh-seed-address-port 10.0.0.125 3002
  port 3002
  interval 150
  timeout 10
}
```

Each entry of 'mesh-seed-address-port' will tell Aerospike to try to cluster with a node at that address. So in this configuration the machine will try to cluster with machines at 10.0.0.123, 10.0.0.124, and 10.0.0.125. It is not necessary to omit the node's own IP address from this list. For example, if deployed on a machine with the IP 10.0.0.124 I do not need to omit "mesh-seed-address-port 10.0.0.124 3002" from the configuration file.

Continuing our review of Example 7-2, the Fabric subcontext controls the communication between the Aerospike servers in the cluster for replication and data redistribution, which will happen over port 3001.

```
fabric {
  port 3001
}
```

The Info context follows immediately and controls the *info protocol*.

```
info {
  port 3003
}
```

The info protocol is used by administrative tools to communicate to a running server, for things like dynamic configuration and fetching runtime statistics. This configures the info protocol to communicate over port 3002. You shouldn't need to modify many

tunables in either of the fabric and info contexts, but the available options can be found in the [Configuration Reference](#).

## Namespace Context

The Namespace contexts are another important configuration area to explore. This is where you define what namespaces the server will make available and what properties they have.

```
namespace test {
  replication-factor 2
  memory-size 4G
  storage-engine memory
}
```

The first namespace defined is called `test`. It has a replication-factor of 2, which means that Aerospike will store a replica copy of every record. You can see the memory-size parameter is specified as 4G, limiting the size of the combination of the index and data stored to less than 4GiB.



Replication-factors can be a confusing topic semantically. Think of it as ‘how many times Aerospike will store the data’ instead of using words like replication, master, and copy. Replication-factor of 2 means Aerospike will have a master and a replica. Replication-factor of 1 means Aerospike will store only a master record. Replication-factor of 3 means a master record and 2 replica copies will be stored. The highest recommended value is a replication-factor of 2 which allows the Aerospike cluster to lose 1 server (or one rack if using the rack-aware property) in a cluster with no data loss.

Finally, in the last context of Example 7-2 is the storage-engine parameter which is using the memory storage-engine for the first namespace. The stock configuration also shows a configuration in the second namespace definition “bar”.

```
namespace bar {
  replication-factor 2
  memory-size 4G
  storage-engine memory

  # To use file storage backing, comment out the line above and use the
  # following lines instead.
  # storage-engine device {
  #   file /opt/aerospike/data/bar.dat
  #   filesize 16G
  #   data-in-memory true # Store data in memory in addition to file.
  # }
}
```

The bar namespace uses memory as its storage-engine, but we can see another commented type out. Let's briefly cover the various storage-engine options more in-depth as this is another key configuration parameter.

**Storage-engine.** There are 3 storage-engines available using Aerospike; memory, device, and Intel's Persistent Memory (pmem). Aerospike's most popular storage-engine is device. Using the device storage-engine makes Aerospike store the data in a storage device while having the index in-memory. This hybrid configuration of maintaining the index in-memory and storing data on device is usually the most appealing of various trade-offs. Table 7-1 summarizes some high level trade offs made when picking a storage-engine.

Table 6-1. Storage-engine trade off matrix (Source: [Aerospike documentation](#))

Storage engine	Device (data not in memory)	All Flash	Device (data in memory)	Memory only (no persistence)	Persistent Memory***
Index In Memory	✓		✓	✓	✓
Fast Restarts	✓	✓	✓**		✓
Survive a Power Outage	✓	✓	✓		✓

As you can see in Table 7-1, the index can also be moved out of memory into pmem or flash. The 'All Flash' configuration stores the index on ssd to maximize key-space at the cost of performance.

The Fast Restarts referenced in Table 7-1 is an Aerospike Enterprise only feature. Fast Restart allows the restarting of an Aerospike daemon that uses a device backed namespace without having to rebuild the primary index. When the index is lost, Aerospike must scan the storage devices to rebuild the primary index on any persisted data.

## Dynamic Configuration

Many of the parameters in the [reference guide](#) can be changed dynamically without downtime, but some have special requirements. The configuration reference guide includes a legend shown on each configuration parameter, that indicates if it needs to be set on restart or can be set dynamically, etc. which are referred to as the *configuration parameters property*. For example, the *context* parameter is a dynamic property and if you highlight over this legend you can see it explains it can be set dynamically.

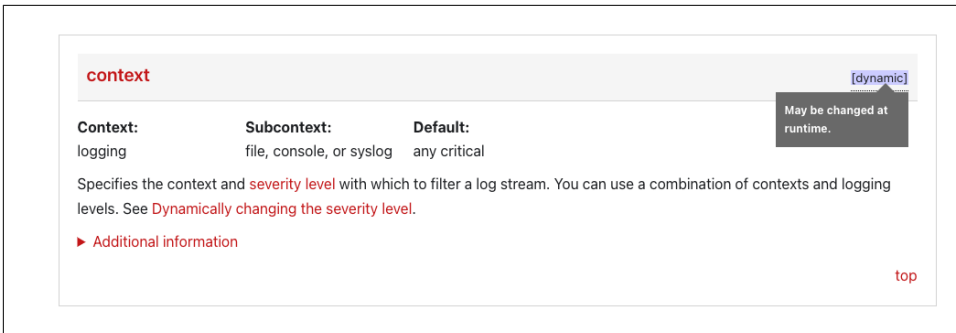


Figure 6-1. Reading the properties of a configuration parameter

By understanding what a configuration parameter does and under what circumstances it can be changed, we can manage our configuration more effectively. Table 7-2 summarizes the various properties of the configuration parameters in a table.

Table 6-2. Configuration parameter properties

Configuration Property	Meaning
dynamic	This configuration parameter can be changed on the running server without requiring a restart.
static	Static configuration parameters can only be changed with a server restart. You must update the configuration file and restart the Aerospike daemon.
unanimous	The configuration parameter must have the same value on all nodes across the cluster. Changing these values may require a total cluster downtime or special procedure to change.
enterprise	These parameters require an Aerospike Enterprise license to use.
required	Aerospike will fail to start if a required parameter is not specified. A log entry should be generated.



In some scenarios you may be restarting the daemon and it does not start or produce logs. This usually happens because the configuration file was invalid so the logs did not go to where you expected. If you cannot find log entries in your specified location you may need to consider running the `/usr/bin/asd` (location may vary) binary directly, which will print the error directly to your console.



Dynamic configuration is a powerful tool with a potentially nasty drawback: There is no way to flush the changes back to the configuration file. If you made dynamic changes to the Aerospike server while it was running, they will be lost unless you also write them to the configuration file. The configuration file is the only source of configuration during a startup. There is no method to print a diff of runtime configuration versus the static configuration file, or a way to copy runtime configuration to a file. Each time you make a dynamic change, you should manually add it to the config file if you intend for it to survive upgrades.

Now that you know how to read the reference manual for configuring Aerospike, how do you make and see those dynamic changes? You need to familiarize yourself with the tools package and how to interact with Aerospike before you get started with tuning parameters dynamically, or inspecting current runtime values.

## Tools

Your Aerospike installation should have included a tools package with it, and if not you can follow the steps outlined in Chapter 2 *Installing Tools* to get those setup. The tools package provides executable programs to manage your Aerospike installation once it's online. Let's start by going over a cursory summary of all the tools provided and a brief explanation of each in Table 7-3 before getting into the details.

Table 6-3. Tools

Tool	Description
asinfo	Used to execute 'info' commands against 1 aerospike instance. Info commands can retrieve information like latency, statistics, usage. Asinfo can also be used to set parameters.
asadm	Used to execute 'info' commands against many aerospike instances, as well as managing access, secondary indices, and <i>User Defined Functions</i> (UDFs).
aql	Aerospike quick look, AQL, is used to perform some basic CRUD operations against a server without needing to write a full program.
asbackup and asrestore	Used to take and restore full, increment, or partial backups of a running cluster.
asloglatency	Used to inspect latency of various histograms by reading the logfile.
asbench	Used to simulate various workloads against Aerospike for testing.

## Asinfo

The asinfo tool is used to change an online cluster without causing downtime, to fetch statistics of the running server, and to assist with troubleshooting. The tool's basic usage is `asinfo -v <command to run>`. There are other options you may want

to use like `-l` to split lines, `-U` and `-P` to specify user and password, and a few others. You can view `asinfo`'s argument help by running `asinfo --help`.

## Dynamic configuration

The most useful feature of `asinfo` may be that it allows you to change those dynamic configuration parameters listed in the configuration reference easily, without affecting the entire cluster. If you think your use case may benefit from modifying some tunable parameter or you're unsure about the impact of what changing a parameter might be, `asinfo` helps limit the scope a lot by allowing you to easily specify the change only be made on one host, without needing to restart a server.

Let's walk through an example of changing a dynamic configuration. If you open the [configuration reference guide](#) and find a parameter tagged as dynamic, you should be able to modify that using `asinfo`. `memory-size` is one that I frequently have to change, so let's start there.

If you look up `memory-size` in the configuration reference, you will see it is dynamic and it is under the namespace context. This shows that you can change it through `asinfo`, and the statement to change it needs to be scoped to namespace.

**memory-size** [required] [dynamic]

**Context:**  
namespace

Maximum amount of memory for the namespace, including the primary and any secondary indexes as well as any data in memory. Cannot be reduced by more than 50% of previously set value. See [Capacity Planning](#) for namespace sizing details.

Prior to 4.3.0.2, the default value was 4GiB. As of 4.3.0.2, `memory-size` is required to be explicitly configured, with a minimum of 1MiB.

► [Additional information](#)

Figure 6-2. `memory-size` configuration parameter



There is an “Additional Information” expansion on this reference which shows the command to modify this parameter for a specific namespace. The additional information is usually very helpful, and sometimes can help get you going faster. Use this to your advantage!

The command to fetch a configuration parameter starts with `get-config`. By default, this command returns the service context configuration. You want to fetch a namespace level configuration though, as this configuration reference tells you `memory-size` is a part of a namespace's configuration. You can do this by specifying the context as namespace, like so:

```
$ asinfo -v 'get-config:context=namespace'  
Error::invalid id
```

Which returns an error because you can have multiple namespaces. This specific context requires you to append “namespace” with the specific namespace you want. The namespace in this cluster is “test”, so we append “id=test”:

```
$ asinfo -v 'get-config:context=namespace;id=test'  
allow-ttl-without-nsup=false;background-query-max-rps=10000;conflict-resolution-  
policy=generation;conflict-resolve-writes=false;data-i....
```

That’s a lot of information! We can filter this down further by using the ‘-l’ option to split this into lines and then pipe into grep:

```
$ asinfo -lv 'get-config:context=namespace;id=test'|grep memory  
high-water-memory-pct=0  
memory-size=1073741824  
stop-writes-sys-memory-pct=90  
storage-engine.data-in-memory=false
```



If you can write something that succinctly and specifically fetches a parameter, then setting it is a small change. Create a command that fetches the configuration you want, then change the verb from get to set and append the particular config parameter to change. In this way, creating a command that fetches a particular parameter becomes a way to set that parameter. This will help you have confidence in building the command and also check it immediately after.

You can see the memory size here is shown in bytes. 1073741824 is 1GiB, so our current memory-size set on the namespace is 1G. We can change this to 2G by using a similar command, “set-config” and adding which parameter must be changed to which value.

```
$ asinfo -v 'set-config:context=namespace;id=test;memory-size=2G'  
ok  
$ asinfo -lv 'get-config:context=namespace;id=test'|grep memory-size  
memory-size=2147483648
```





Unless otherwise specified, all space-related parameters in Aerospike are binary representations of their unit. 1G means 1 Gibibyte, not 1 Gigabyte. A Gibibyte is 1024 Mebibytes or  $2^{30}$  bytes. This may confuse an operator because a machine's "free" command, as well as other references of space, may show units in GB. If the free command in Linux shows 500GB as available memory, that only equals 465G in Aerospike as it is 465GiB of memory. This behavior of free varies between Linux distributions, but the unit is something that you need to be mindful of. Most size related parameters support some short form of units like this, and you should consult the config reference.

Similarly, you can read and change the configuration parameters in the service context:

```
$ asinfo -lv 'get-config:context=service'|head
advertise-ipv6=false
auto-pin=none
batch-index-threads=12
batch-max-buffers-per-queue=255
batch-max-requests=5000
batch-max-unused-buffers=256
cluster-name=null
debug-allocations=none
disable-udf-execution=false
downgrading=false
```

Try changing the first configuration parameter we found, `advertise-ipv6`:

```
$ asinfo -lv 'set-config:context=service;advertise-ipv6=true'
ok
$ asinfo -lv 'get-config:context=service'|head -n1
advertise-ipv6=true
```

You can follow this pattern for many of the contexts within the configuration file. If you intend to change this on many servers, you should use `asadm` to make this change instead which we will cover later.

## Statistics, special settings, latencies

Separate from the configuration reference which we've focused on so much so far, there is also a [reference for info commands](#), which shows the other utilities `asinfo` can help execute. Of special note are the `latencies` command and `logs`. The `latencies` command allows you to report the last 10s of transactions as a histogram, useful for monitoring. The `logs` command allows you to change the verbosity of various types of messages, such as turning off a particular warning or making some code paths more verbose for troubleshooting. All of the various other ways of using `asinfo` are for more advanced users and situations/problems probably outside the scope of this book.



It should be noted the info command reference guide is a generic reference for the info protocol, and you can execute these commands using the client driver if you need to run these from an application. Aerospike provides all these utilities, but none of them are required. You can recreate the functionality of all of these tools, or some subset of their functionality in a custom written application. For example,

```
$ asinfo -v 'get-config:context=namespace;id=test'
```

Is equivalent to a Java info call:

```
Info.request(node, 'get-config:context=name-  
space;id=test');
```

## Asadm

The `asadm` utility is primarily an info protocol tool like `asinfo`, but for the whole cluster. When you need to check the health of a cluster as a whole a tool like `asadm` is helpful as it can fetch information from all members of the cluster and summarize them in a readable form. If you need to change a setting across the entire cluster `asadm` is also the first choice in executing a change like this. `Asadm` also enables you to manage access-control-lists (ACLs), quotas, secondary indices, and user-defined functions (UDFs).

There are two help commands for `asadm`. First, passing `--help` at the command line while executing `asadm` will give you commands to get connected to the cluster such as

```
$ asadm --help
```

The second describes which commands you can run from within the `asadm` shell. When you execute the `asadm` command, you will enter an `asadm` interactive session. You can tell you are inside the session because it will prefix the cursor with “Admin>”. You can run the command `help` here to see what commands are available, and the `exit` command to quit the `asadm` shell.

```
$ asadm  
Seed:      [('127.0.0.1', 3000, None)]  
Config_file: /Users/scoob512/.aerospike/astools.conf, /etc/aero-  
spike/astools.conf  
Aerospike Interactive Shell, version 2.15.0  
  
Found 1 nodes  
Online: 127.0.0.1:3000  
  
Admin>
```

Managing Aerospike will likely start with fetching information and reviewing the cluster. So let's start by covering ways to fetch and visualize information from the cluster.

## Informational commands

Once you're inside the `asadm` interactive shell, `info` is a great command to start with.

```
Admin> info
```

This command gives you a decent picture of the current state of the cluster: if migrations are pending, how many servers are in the cluster, how many records exist and replicas, how much space is being used, and a few other things. If you execute the `help info` command, you can see different modifiers and in the following sections, you can display summaries and statistics on: `dc`, `network`, `set`, `sindex`, `xdr`, `namespace`. These can be used with the `asadm info` command by adding them after the `info` command. Example:

```
Admin> info set
```

This overview of your cluster will be missing the latencies. You can find latency using the `show` command. The `help show` command will print the help menu for the available `show` commands. Some interesting `show` commands for getting started are `show latencies`, `show statistics`, and `show config`.

```
## Admin
Admin> sh latencies
~~~~~Latency (2023-08-07 00:10:19 UTC)~~~~~
Namespace|Histogram|                               Node|ops/sec|>1ms|>8ms|>64ms
test      |read      |1.0.0.127.in-addr.arpa:3000| 2658.6|0.11| 0.0| 0.0
          |          |                               | 2658.6|0.11| 0.0| 0.0
test      |write     |1.0.0.127.in-addr.arpa:3000| 2650.7|0.11| 0.0| 0.0
          |          |                               | 2650.7|0.11| 0.0| 0.0
Number of rows: 2

Admin>
```

The `show latencies` command gives us a breakdown of how many reads, writes, UDFs, and batch transactions occurred in the last 10 seconds, as well as what percentage of these transactions fell into various latency buckets.

Another very useful modifier that you can specify with the `show` command is the `like` keyword. For example,

```
## Admin
Admin> show lat like write
```



You can shorten commands at the expense of readability. Most commands require only enough characters written to disambiguate which to execute, for example if you type `s` and hit the tab key twice, you will see that the command `s` is ambiguous. It could mean either `show` or `summary`. If you type `su`, it is disambiguous enough to execute the closest matching command which is `summary`. If you find these shortened forms online in examples, and are uncertain what they stand for, you can hit tab to autocomplete their meaning. For example: `i <TAB>` will autocomplete to `info`. Similarly `sh la<TAB>` will autocomplete to `sh latencies`

This view is especially useful when combined with the `config` section. It differs from executing commands like `get-config` in `asinfo` because it can search all config sections from all nodes in the cluster.

```
## Admin
Admin> sh config like thread
~~Service Configuration (2023-08-07 00:14:31 UTC)~~~
Node |1.0.0.127.in-addr.arpa:3000
batch-index-threads |12
info-threads |16
migrate-threads |1
query-threads-limit |128
service-threads |60
sindex-builder-threads|4
Number of rows: 7

~~~~~Network Configuration (2023-08-07 00:14:31 UTC)~~~~~
Node |1.0.0.127.in-addr.arpa:3000
fabric.channel-bulk-recv-threads|4
fabric.channel-ctrl-recv-threads|4
fabric.channel-meta-recv-threads|4
fabric.channel-rw-recv-threads |16
fabric.send-threads |8
Number of rows: 6

~test Namespace Configuration (2023-08-07 00:14:31 UTC)~
Node |1.0.0.127.in-addr.arpa:3000
nsup-threads |1
single-query-threads |4
truncate-threads |4
xdr-tomb-raider-threads|1
Number of rows: 5
```

Equally useful as perusing and finding the exact configuration parameter name, is doing that with statistics.

```
## Admin
Admin> sh stat like client.*err
~~~~~Service Statistics (2023-08-07 00:17:32 UTC)~~~~~
```

```
Node |1.0.0.127.in-addr.arpa:3000
early_tsvc_client_error|0
Number of rows: 2
```

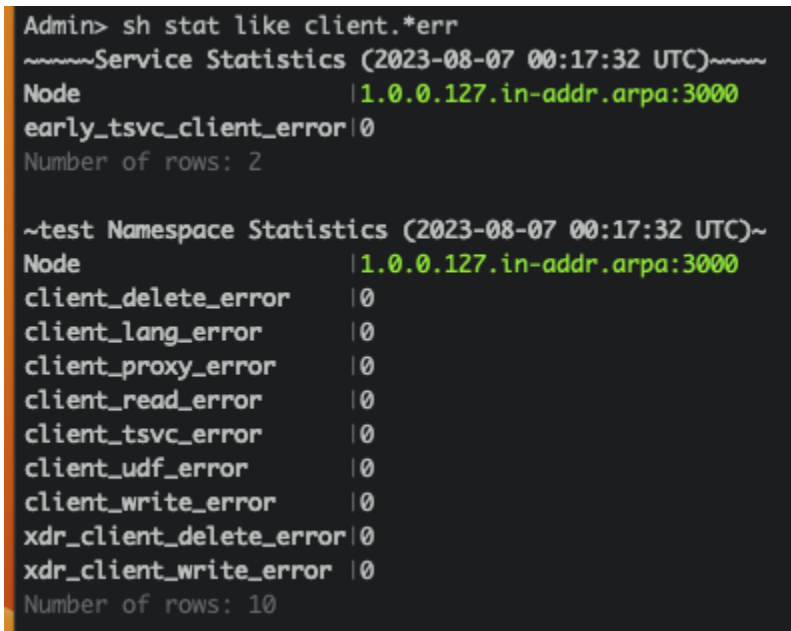
```
~test Namespace Statistics (2023-08-07 00:17:32 UTC)~
```

```
Node |1.0.0.127.in-addr.arpa:3000
client_delete_error |0
client_lang_error |0
client_proxy_error |0
client_read_error |0
client_tsvc_error |0
client_udf_error |0
client_write_error |0
xdr_client_delete_error|0
xdr_client_write_error |0
Number of rows: 10
```

```
Admin>
```

Finally, the summary command is a great way to summarize a cluster's usage without having to get too technical about all the Aerospike specific terminology.

```
Admin> summary
```



```
Admin> sh stat like client.*err
~~~~Service Statistics (2023-08-07 00:17:32 UTC)~~~~
Node |1.0.0.127.in-addr.arpa:3000
early_tsvc_client_error|0
Number of rows: 2

~test Namespace Statistics (2023-08-07 00:17:32 UTC)~
Node |1.0.0.127.in-addr.arpa:3000
client_delete_error |0
client_lang_error |0
client_proxy_error |0
client_read_error |0
client_tsvc_error |0
client_udf_error |0
client_write_error |0
xdr_client_delete_error|0
xdr_client_write_error |0
Number of rows: 10
```

## Modifications

Using the `asadm` shell, we can modify the entire cluster dynamically. This can be dangerous as it affects the entire cluster, so if you are unsure of the setting you're changing you may want to consider running this on a single node using `asinfo` or use `asadm` in 'single node' mode by passing '--single-node' to make modifications only occur on a single node.

```
$ asadm --single-node
```

Once you're in the `asadm` interactive terminal there are two main ways of changing configuration parameters. You can either execute the `asinfo` command from this shell, or use `asadm`'s "manage" commands.

There is a safety mechanism to prevent accidental changes while in `asadm`. You must first enter a privileged mode using the `enable` command. If you run a command inside the `asadm` shell that changes something without being in privileged mode, you will see an error that. `Asinfo` commands can be ran inside of `asadm` and `asadm` will always assume they may make changes. The error will look something like this:

```
## Admin
Admin> asinfo -v 'get-config:context=service'
ERROR: User must be in privileged mode to issue "asinfo" commands.
       Type "enable" to enter privileged mode.
Admin>
```

To get past this, you can follow the instructions and execute the "enable" command.

```
Admin> enable
Admin+>
```

We can see the "+" marker next to our prompt which informs you that you can execute changes and the safety is off. To stop running with 'enable' you need to exit `asadm` and open a new session.

From this enabled `asadm` session, you can now execute `asinfo` commands and make changes. You can try making the same changes from the earlier `asinfo` example by changing the memory size.

```
## Admin
Admin> sh config like memory-size
~test Namespace Configuration (2023-08-07 00:27:17 UTC)~
Memory-size|2147483648

Admin> enable

Admin+> asinfo -v 'set-config:context=namespace;id=test;memory-size=3G'
1.0.0.127.in-addr.arpa:3000 (1.2.3.4) returned:
ok
```

In a multi-node cluster, you would see a response from each server. You can press the up arrow on your keyboard to get back to the “show” command and verify our changes made it to all servers.

```
Admin+> sh config like memory-size
~test Namespace Configuration (2023-08-07 00:28:53 UTC)~
Node      |1.0.0.127.in-addr.arpa:3000
memory-size|3221225472
```

You can also make this change using the manage command in asadm. You can execute `help manage config` which leads us naturally to `help manage config namespace` as you will be changing the configuration parameter from the namespace context. This should show the following usage:

```
Usage: namespace <ns> [<subcontext>] param <parameter> to <value>
```

We can fill in the command with our real values and try it.

```
## Admin
Admin+> manage config namespace test param memory-size to 2G
~Set Namespace Param memory-size to 2G~
Node|Response
1.0.0.127.in-addr.arpa:3000|ok
Number of rows: 1

Admin+>
```

## Managing Index, ACL, UDF

Aerospike supports security features that allow fine grained or broad basic controls on users through the “manage acl” command, secondary indexes to be created on your data using the “manage index” command, and registering custom user-defined functions using “manage udfs”. These are more advanced topics that will be covered in a later section.

## Aerospike Quick Look

Aerospike Quick Look (AQL) utility is a tool for interacting with some of the data in the Aerospike cluster. Using AQL we can read, write, and query data. This can be a big time saver if we want to check if a record exists, or maybe inspect the bins of a specific record.

AQL does not have full feature parity with client drivers, so you may need to use an IDE to get full visibility into your data if `aql` is unable to display it or especially arbitrarily serialized data types like a protobufed blob.

Aside from reading and writing single keys, the most useful thing in AQL from an operator’s perspective might be that we can look at the metadata of a key or find

which node is the primary and secondary for that record. Let's get into the AQL interactive shell and explore some of this.

Unlike `asadm`, the `'--help'` menu does show us the commands we can run inside AQL.

```
$ aql --help
```

To get to the help page for connecting and entering aql shell, we execute with `"-?"` on the end of the command:

```
$ aql -?
```

The specific commands I want to show you are inspecting the metadata and the source of a record. We will need a test record to inspect, so let's create that in AQL. I'm going to copy paste the example from the help menu.

```
## AQL
aql> INSERT INTO test.demo (PK, foo, bar, baz) VALUES ('key1', 123, 'abc', true)
```

This command has created 1 record in the `"demo"` set of the `"test"` namespace. Its primary key is `"key1"` and it has 3 bins: `foo`, `bar`, and `baz` which have the values `123`, `abc`, and `true`, respectively.

We can retrieve and display this record now,

```
## AQL
aql> select * from test.demo where pk = "key1"
+-----+-----+-----+-----+
| PK    | foo | bar | baz |
+-----+-----+-----+-----+
| "key1" | 123 | "abc" | true |
+-----+-----+-----+-----+
1 row in set (0.001 secs)
OK
```



With some data types and especially long values, AQL is unable to display all or in some cases any of the data in the chosen output type. Try using the `RAW` output type in those scenarios by typing `"SET OUTPUT RAW"`. You can view the other options in the help page or by executing `"help SET"` in the AQL session.

We can see our record exists and we get a nice table showing us what's in the record. As an Admin responsible for troubleshooting, we might also want to inspect the metadata.

```
## AQL
aql> SET RECORD_PRINT_METADATA true
RECORD_PRINT_METADATA = true
aql> select * from test.demo where pk = "key1"
+-----+-----+-----+-----+-----+
| PK    | foo | bar | baz | {ttl} | {gen} |
```



```

+-----+-----+-----+-----+-----+-----+
| "key1" | 123 | "abc" | true | 2591764 | 1 |
+-----+-----+-----+-----+-----+
1 row in set (0.001 secs)
OK

```

You can change the setting to show metadata in AQL and run the select command again. This time you can see TTL, representing the seconds until the record expires, and gen, representing the generation of the record.

## Asbackup and Asrestore

The Aerospike tools package also contains utilities to make and restore backups. The backup utility works by performing a full namespace scan and for each record it finds, generating files that can be saved to disk, another machine, or cloud storage. The restore utility reads in that stored data by asbackup and for each record it can construct it performs a client write.

These utilities aren't doing block-level snapshots, write-ahead logs, or any advanced techniques. These are the same kind of calls and interactions you can perform from the client driver (scan, read, write).

Asbackup uses a scan to retrieve the data, reading records like a client and storing them somewhere. This behavior can be problematic, as you may have to worry about the data changing as you scan it in. Because there is no namespace-level or set-level locking, it's possible for some records to be taken in a backup from 10:00 am and some records aren't backed up until 11:00 am. If there is some relationship between these records, that could be problematic. The only locking mechanism during an asbackup is that the records are briefly locked while the read occurs to ensure consistency of a record.

At the same time, this behavior can be beneficial as it means you don't need to pause all client traffic while the database takes place.

The asbackup and asrestore commands each have their own help pages. Let's show how you can back up and restore using default options:

```

## Shell
$ asbackup -n test -o testfile
2023-08-13 18:39:37 UTC [INF] [29199] Starting backup of 127.0.0.1 (name-
space: test, set: [all], bins: [all], after: 1969-12-31 17:00:00 MST, before:
1969-12-31 17:00:00 MST, no ttl only: false, limit: 0) to testfile
2023-08-13 18:39:37 UTC [INF] [29199] [src/main/aerospike/as_clus-
ter.c:202][as_cluster_add_nodes_copy] Add node BB9020011AC4202 127.0.0.1:3000
2023-08-13 18:39:37 UTC [INF] [29199] Processing 1 node(s)
2023-08-13 18:39:37 UTC [INF] [29199] Node ID           Objects           Repli-
cation
2023-08-13 18:39:37 UTC [INF] [29199] BB9020011AC4202     109818             1
2023-08-13 18:39:37 UTC [INF] [29199] Namespace test contains 109818 record(s)

```

```

2023-08-13 18:39:37 UTC [ERR] [29199] Output file testfile already exists; use
-r to remove
2023-08-13 18:39:37 UTC [INF] [29199] Backed up 0 record(s), 0 secondary
index(es), 0 UDF file(s), 0 byte(s) in total (~0 B/rec)
$ ls -lad testfile
-rw-r--r-- 1 albert albert 871559458 Aug 6 19:17 testfile

```

The `asbackup` command requires at a minimum a namespace (`-n test`) and somewhere to output the data (`-o testfile`). The `asrestore` command requires even less, as the backup file has metadata on which namespace the file should be written to.

Then, to restore you just feed the file in as input:

```

$ asrestore -i testfile
...
2023-08-13 18:40:49 UTC [INF] [30088] Opened backup file testfile
2023-08-13 18:40:49 UTC [INF] [30088] Restoring records
...
2023-08-13 18:41:02 UTC [INF] [30142] 0 UDF file(s), 0 secondary index(es),
101654 record(s) (6887 rec/s, 58544 KiB/s, 8704 B/rec, retries: 0)
2023-08-13 18:41:02 UTC [INF] [30142] Expired 0 : skipped 0 : err_ignored 0 :
inserted 0: failed 101654 (existed 0 , fresher 101654)
2023-08-13 18:41:02 UTC [INF] [30142] 100% complete, ~0s remaining
$ echo $?
0

```



The `asrestore` utility cannot restore the original generation of a record. If a record is created on the server that exists in the backup file, one of those versions will be clobbered. There is no merge strategy for records' bins, though the `asrestore` write is an upsert. You may want to consider passing the `--unique` parameter described in the help page as "Skip records that already exist in the namespace; Don't touch them." or the `--replace` option which does the opposite. If you inspect the output of this command, you can see that all records *successfully failed* to be restored as the command wasn't modified to clobber the existing data in the namespace which may be more valuable to us.

We can also stream this backup data over the network, write to a mounted file system, or even write directly to an `s3` path by leveraging the stdout flags on `asbackup` output or `asrestore` output:

```
$ asbackup -o - | <do something with streaming stdout>
```

This can be helpful when streaming to proprietary storage solutions or compression software, like `zstd`:

```
$ asbackup -o - | pzstd -9 | cat > mybackup.zst
```

Similarly, `asrestore's -i -`

```
$ do something | asrestore -i -
```

Or continuing the example of restoring a compressed backup file:

```
$ cat mybackup.zst | pzstd -d | asrestore -i -
```

This makes asrestore and asbackup great tools you can use within the Linux GNU ecosystem of utilities. Aerospike did add compression to be a built-in feature under the “-z” flag though so this example isn’t very relevant today, but good to illustrate some of the ways you can pair it with GNU utilities.

An additional notable feature of asbackup is the ability to perform incremental backups using “--modified-after” or “--modified-before”. Instead of performing a full backup each time, you can perform 1 full backup and then take a backup of the things that have changed since the last backup. Then in the event of restore you will restore the full backup and then each incremental backup in the order they were taken. This greatly reduces the time to take backups of the database at the expense of complexity on restore time depending on how long you want to go between taking a full backup.

## Asloglatency , asbench

The asinfo and asadm tools shown above already show you some percentage distribution in various histogram buckets for latencies, but there is a more advanced utility called asloglatency. Asloglatency is a python utility for interpreting the histograms written in the aerospike log file. This utility can be helpful for histograms not reported by the other tools, inspecting historically logged histograms, or if you want to inspect at a higher granularity on the fly. Asloglatency allows you to inspect specific histograms and is particularly helpful if you need to enable and inspect microbenchmarks. Microbenchmarks allow a finer grain of visibility and latency tracking by adding additional histograms to aid in investigating where a problem may be.

For example, if you have read latency on your *test* namespace and are unsure what the cause is, you might want to enable read benchmarks to get extra logging on it.

```
$ asinfo -v 'set-config:context=namespace;id=test;enable-benchmarks-read=true'
```



Microbenchmarks can add additional latency and greatly increase the amount of logging generated. You should always test in a non production system, limit enabling this to one node at a time if possible, and remember to turn it off when troubleshooting is done.

This generates extra histograms in our log file, let’s find it in our log:

```
$ docker logs aerospike | grep hist | grep read|tail -n 7  
Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read  
(753 total) msec
```

```

Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
start (753 total) msec
Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
restart (0 total) msec
Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
dup-res (0 total) msec
Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
repl-ping (0 total) msec
Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
local (753 total) msec
Aug 13 2023 19:14:42 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
response (753 total) msec

```

As I am using docker, I need to export these logs to a file to be interpreted by `asloglatency`. You can skip this step if you already have your logs in a file.

```

$ docker logs aerospike > histogram.logs
$ grep "hist.c" histogram.logs | tail
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
local (753 total) msec
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:331) (00: 0000000724) (01:
0000000006) (02: 0000000015) (03: 0000000007)
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:340) (04: 0000000001)
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:320) histogram dump: {test}-read-
response (753 total) msec
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:331) (00: 0000000710) (01:
0000000036) (02: 0000000005) (03: 0000000001)
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:340) (06: 0000000001)
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:320) histogram dump: {test}-
write (774 total) msec
Aug 13 2023 19:15:32 GMT: INFO (info): (hist.c:340) (06: 0000000001) (07:
0000000667) (08: 0000000106)

```

“`hist.c`” is responsible for logging the histograms which is why you may want to use `grep` to search for that string in particular. You can see various histograms named on the line with “`histogram dump`”. The first line shows us the `test` namespace had a `read-local` histogram produced. Now that you know where your logfile, you can run `asloglatency` against it.

Let’s first look at the main histogram for reads that’s enabled by default:

```
$ asloglatency -l histogram.logs -h "{test}-read" -f head
```

And then you can drill into a specific slice of that read histogram, since you enabled these microbenchmarks:

```

$ asloglatency -l histogram.logs -h "{test}-read-local" -f head
Histogram Name : {test}-read-local
Log           : histogram.logs
Aug 13 2023 19:07:01
                %> (ms)
slice-to (sec)  1      8      64     ops/sec

```

19:20:12	10	0.00	0.00	0.00	0.0
19:20:22	10	1.88	0.87	0.00	69.1
19:20:32	10	0.59	0.00	0.00	67.9
19:20:42	10	0.95	0.00	0.00	63.3
19:20:52	10	0.47	0.00	0.00	64.4
19:21:02	10	0.48	0.00	0.00	62.3
19:21:12	10	0.48	0.00	0.00	63.0
19:21:22	10	1.01	0.00	0.00	59.3
19:21:32	10	0.83	0.00	0.00	59.9
19:21:42	10	1.25	0.00	0.00	16.0
avg		0.16	0.05	0.00	6.7
max		5.88	3.43	0.00	69.1

If that's not enough resolution, you can add *-e* and *-n* to your command to change the number (*-n*) and exponential difference (*-e*) of histogram buckets shown. Asloglatency works with all histograms found in the aerospike logfile.

Throughout this chapter we've inspected latencies on a running container, but I never had to write an application to generate a workload. For various reasons like testing new hardware, testing capacity planning theories, or even testing disaster scenarios we may need a way to generate a workload synthetically. Asbench is part of the aerospike-tools package and enables you to do that. There is an extensive number of options for customizing the workload to simulate and the behavior you want it to take, but to keep it short for the purposes of this chapter here is the command I ran to generate the above histograms:

```
$ asbench -n test -o 'B1000000' -w RU,50 -k 100 -t 120
```

This asbench command targets localhost by default, as many of the tools do. The namespace specified is test and we're using an object (record) specification to have a 1000000 byte binary blob bin. The workload is 50% reads and 50% updates. The number of keys used in this run is limited to only 100 records as I wanted to try to cause some contention. Finally, the asbench run was limited to 120 seconds of runtime.

When you run it, you'll first see the various workload stages and parameters defined being printed to you and then a ticker showing how many reads writes and failures asbench has generated.

```
## Shell
$ asbench -n test -o 'B1000000' -w RU,50 -k 100 -t 10
...
namespace:          test
set:                testset
start-key:          1
keys/records:       100
...
```

```
2023-08-13 13:32:59.944 INFO write(tps=60 (hit=60 miss=0) timeouts=0 errors=0)
read(tps=83 (hit=83 miss=0) timeouts=0 errors=0) total(tps=143 (hit=143 miss=0)
timeouts=0 errors=0)
```

You can use this tool to synthetically run a benchmark against a running cluster, with a workload that matches your own. This should help characterize how Aerospike and your hardware handle the workload.

## Security

Aerospike has various security features that enable you to run your cluster in a responsible and compliant way:

- **TLS**, to encrypt communication over the network from client to server or server to server replication.
- **ACLs**, to control specific users' access and quotas to specific namespaces and sets.
- **Hashicorp vault** integration, for secret retrieval and storage.
- **LDAP**, to manage authentication to the servers.
- **Encryption-at-rest**, to ensure data flushed to the disk is encrypted.
- **FIPS 140-2 certification**, a specialized certification that meets US federal government requirements.

All of these security features are behind an enterprise server license, so if these are features you need you can engage with the support staff to assist with any setup trouble. The public documentation available on [Aerospike's website under the security overview section](#) should contain all the information you need to set this up. As you will have an enterprise license while setting this up, you can also work with enterprise support to help with concerns and setup help so we won't go in depth on each of these features.

## Recap

In this chapter we've discussed the configuration file format and the importance of the configuration reference in setup and in daily operations work. We discussed the various tools that ship with Aerospike: `asinfo`, `asadm`, `aql`, `asbackup`, `asrestore`, `asloglatency`, `asbench`, and how they are used for daily work, troubleshooting, and benchmarking. Finally, we have a brief overview of the security features that Aerospike offers in the enterprise edition. In Chapter 8 we will build on using these utilities to perform upgrades and monitor the health of our cluster.

---

# Monitoring and Best Practices

## A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [gobrien@oreilly.com](mailto:gobrien@oreilly.com).

With every production system, you need to be able to quickly identify and respond to issues. You must ensure that you monitor Aerospike to keep your applications running smoothly in case of hardware issues, software bugs, configuration problems, and networking issues. This chapter will focus on guiding you towards a monitoring solution, vital metrics to monitor, and how to best respond to some problem scenarios. By the end of this chapter, you should have a grasp on how to monitor your Aerospike deployment, how to upgrade your servers running Aerospike, and some first steps to respond to operational demands.

## Monitoring

There are two sides of monitoring Aerospike, from the client application using it and from Aerospike itself. Both are important for a robust and complete monitoring solution. If you monitor only the Aerospike database and the metrics it reports, you won’t know if an operation fails to reach the server entirely. Similarly if you only monitor the client application you may miss out on critical information from the

cluster itself or warning systems ramping up to an incident, such as lack of storage space.

## Application metrics

Within your Aerospike client application, you will want to add monitoring around the latency of your calls, success rate, and you'll want to subscribe to the Aerospike logging interface. The latency and success rate measurements are up to you to implement by adding stop-watch timers, catching and counting exceptions and return codes.

A more obscure, frequently overlooked, built-in feature is the client logging interface. You should implement the log callback function to log information from the client. Most entries are related to node membership, reachability, and cluster events. You can read about [the logging interface and implementation here](#). This can be an instrumental component in diagnosing issues, especially ones that are hard to diagnose like intermittent partial network connectivity.

## Aerospike database metrics

The Aerospike database produces a variety of metrics. There are metrics you'd expect from a database like space remaining, latency, hit ratio, as well as deep forensic metrics you can review using Aerospike tools like `asloglatency`. You will want a system that collects these metrics to inform you if there is an issue, but you also want a system that will allow you to view trends in the data. Alerting can tell you if there is a given issue at a particular instant, and the trending allows you to determine if this is an organic growth, related to a deployment, or some other event. Trends also allow you to evaluate if a particular threshold would be appropriate, given the history and recent changes in data, or if that threshold would be too noisy.

The [Aerospike monitoring page](#) lists common monitoring options and whether or not they support alerting and trending, which I will summarize in Table 8-1.

*Table 7-1. Monitoring Options*

Tool	Documentation	Alerting	Trending
Aerospike Monitoring Stack with Prometheus Exporter	Aerospike Monitoring Stack	Yes	Yes
Aerospike Management Console	Aerospike Management Console	Yes	No
ASADM	ASADM	No	No
Aerospike Logs	Aerospike Logs	No	Yes
Collectd	<a href="#">aerospike-community/aerospike-collectd</a>	Yes*	Yes*
Graphite	<a href="#">aerospike-community/aerospike-graphite</a>	No*	Yes
Nagios	<a href="#">aerospike-community/aerospike-nagios</a>	Yes	No*
Zabbix	<a href="#">aerospike-community/aerospike-zabbix</a>	Yes	Yes
Data Dog	Data Dog Docs	Yes	Yes



\* Solution has third-party plugin for Alerting or Trending

Among these various solutions for monitoring Aerospike, a common choice is Prometheus which will be discussed in the remainder of the chapter. The thresholds, ideas, and methods apply to other solutions but the implementation will vary. For example, you could instead choose to use Nagios for alerting, paired with Graphite for trending and dashboard.

## Prometheus

Prometheus is a monitoring platform that collects metrics from various servers into a local time-series database, and can be set up with rules to aggregate and alert on these metrics. Prometheus is usually deployed as its own dedicated system, and has rules set up to tell it where to find metrics and what to do with those metrics. You usually will have an alerting system integrated with it, and maybe a dashboarding utility like Grafana. An example topography may look something like Fig 8-1.

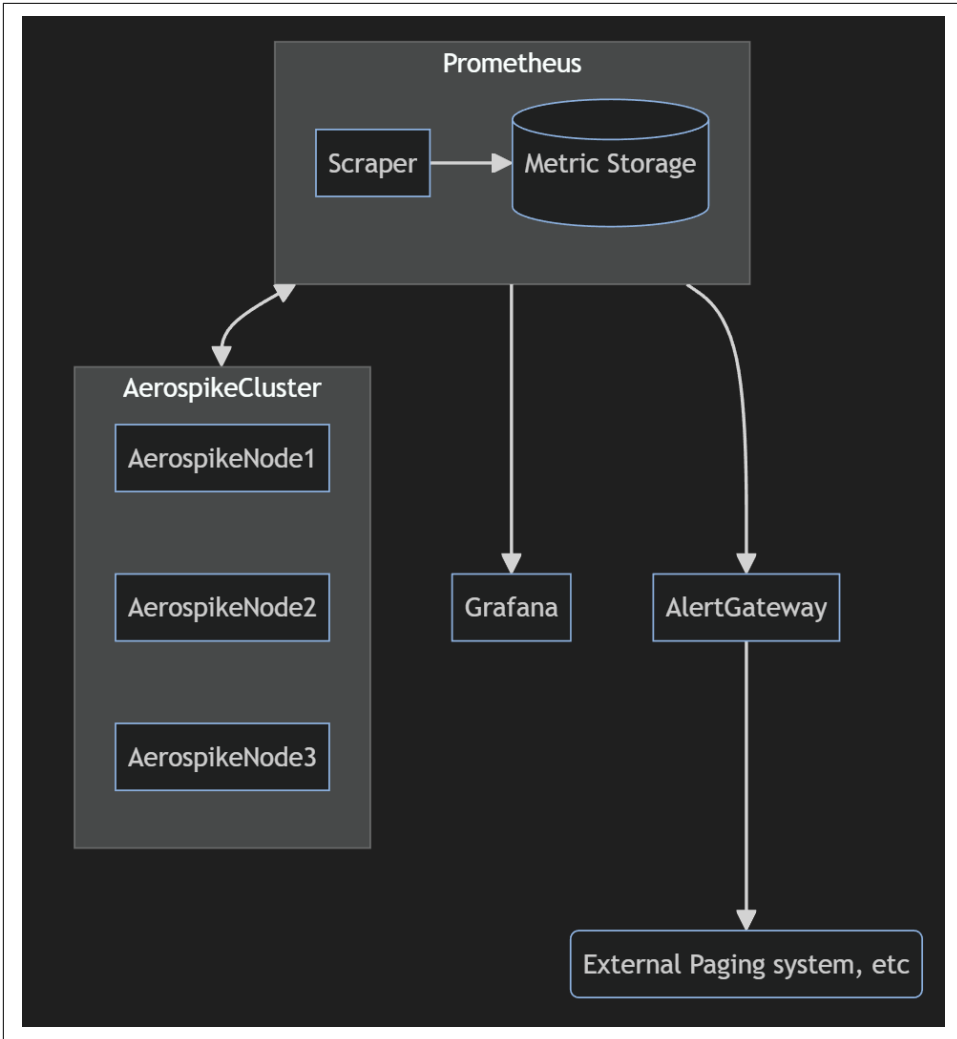


Figure 7-1. Prometheus monitoring topology example

Functionally at its core, Prometheus has a list of web addresses it needs to read. Prometheus performs a web request, GET, to each of the defined addresses and stores the metrics returned into a database. On some schedule, there is an internal system that performs queries based on rules from this data to generate alerts. The data stored in the system can also be graphed using Prometheus or a dashboarding utility like Grafana. Figure 8-1 is simplified, and the actual implementation requires us to scrape every individual Aerospike node, shown in Figure 8-2.

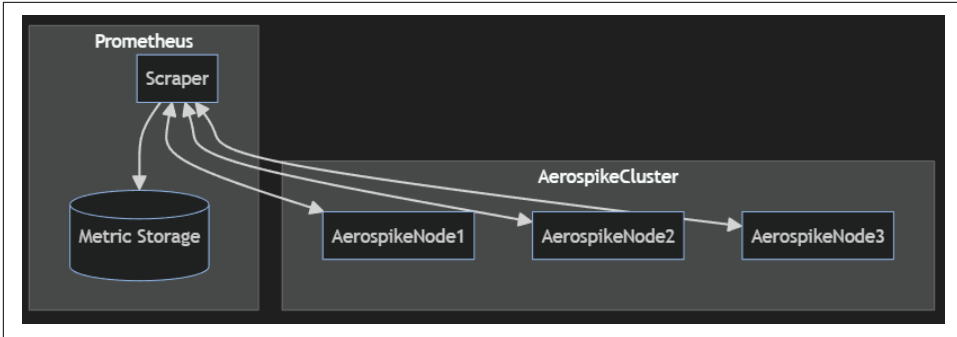
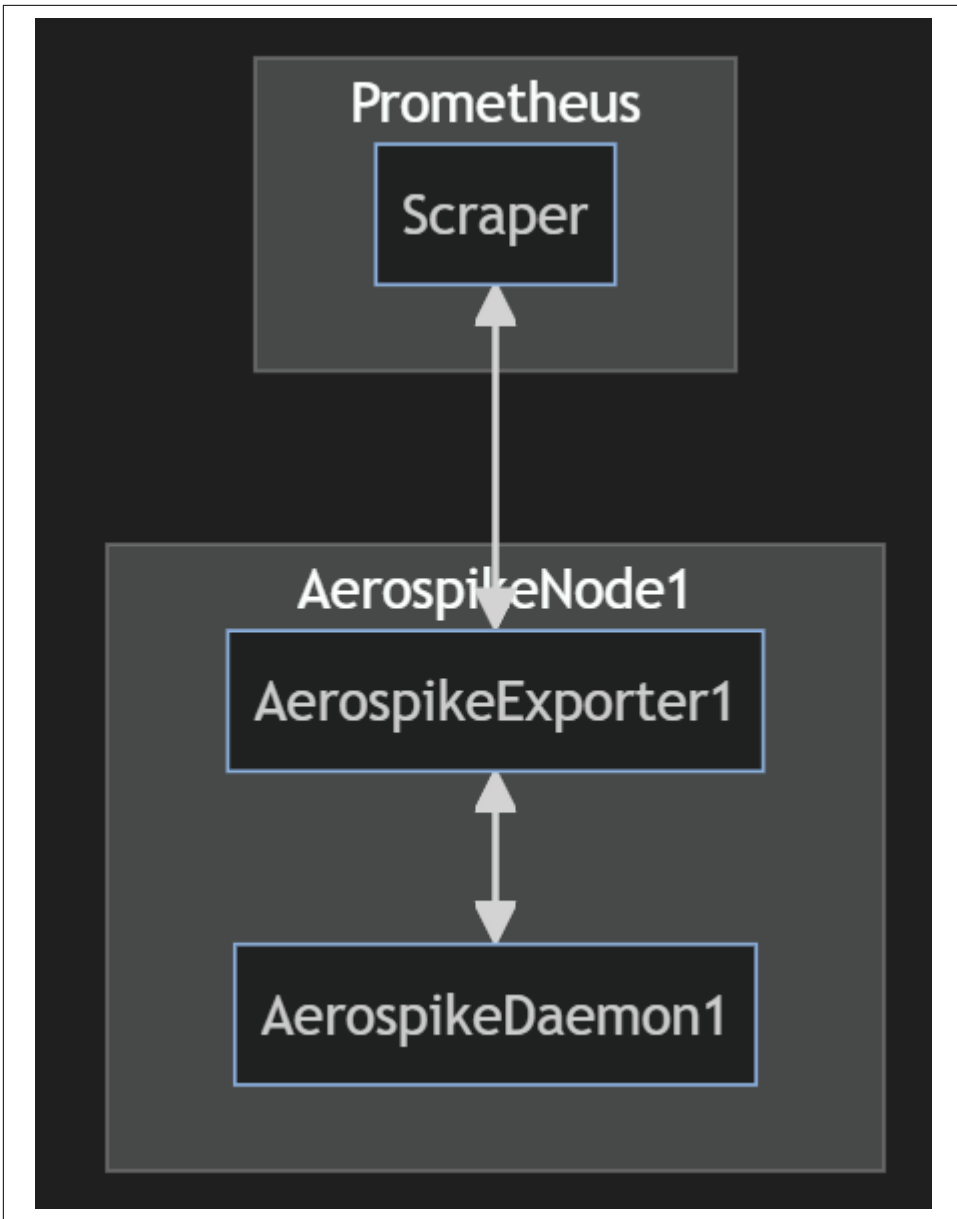


Figure 7-2. Prometheus scraping individual Aerospike instances

As Prometheus must poll every individual instance of Aerospike, you must configure each node to be set up to handle those requests. The requests Prometheus sends are web requests, which Aerospike doesn't support natively. The connecting glue that bridges the gap between an application and Prometheus is usually by means of something called an *exporter*. As shown in figure 8-3, the exporter runs alongside Aerospike to serve as a means of converting application specific signals to Prometheus style metrics, and handles the web request.



*Figure 7-3. Prometheus scrapes the Exporter instead of the Aerospike database itself*

You should be able to invoke web requests or use your browser, if the connectivity exists, to view the format of the web requests and the metrics present. Here is an example of a local exporter running, to illustrate you can simply curl the exporter's endpoint and read the result yourself.

```
$ curl -s http://localhost:9145/metrics
...
# HELP aerospike_node_up Aerospike node active status
# TYPE aerospike_node_up gauge
aerospike_node_up{build="6.4.0.10",cluster_name="null",service="172.30.0.4:3000"} 1
...
```

This book will not cover the set up of Prometheus targets and service discovery implementations, as those will vary between organizations.

## Aerospike exporter

The Aerospike exporter is typically installed as a systemd service along with the Aerospike database running on the same machine, but the exporter could also be run using docker on the machine or even a separate container within the same pod if using a kubernetes deployment.

### Aerospike Exporter Service Install

If you wish to run the Aerospike exporter as a systemd style service on a machine, you will need to be running Linux. The [Aerospike Prometheus exporter binary packages](#) can be found on the download page under observability and management, where you can choose what package you need. Then you will install using the appropriate `dpkg` or `rpm` package installation command. In the example here, the ARM64 package will be installed on a debian-like system using `dpkg --install <file.deb>` or `dpkg -i <file.deb>` for short.

```
# Download the 1.9.0 ARM64 Debian Aerospike Prometheus exporter package:
wget https://download.aerospike.com/artifacts/aerospike-prometheus-exporter/1.9.0/
aerospike-prometheus-exporter_1.9.0_arm64.deb
# Install the package
dpkg -i aerospike-prometheus-exporter_1.9.0_arm64.deb
```

With the exporter package installed, you should also have a configuration file. This config file should have been created at `/etc/aerospike-prometheus-exporter/ape.toml`. You should review this file to see what kind of customizations you can make and update it as needed referring to the [github page](#) for additional info. Some common changes to the exporter include blocklisting metrics to reduce the number of metrics Prometheus needs to ingest, adding a username and password for authentication, and customizing the port to connect to Aerospike or the port to listen on.

Now that the package is installed and you've added any necessary configuration changes to the file, you should be able to start the service using the `systemctl start aerospike-prometheus-exporter`. Once started, you can verify it remains running and read logging output using the command: `systemctl status`.

```
$ sudo systemctl status aerospike-prometheus-exporter
aerospike-prometheus-exporter.service - Aerospike Prometheus
...
Active: active (running) since Sat 2023-12-23 23:35:01 UTC; 2s ago
```

Once the exporter is up you should check that it is serving metrics and responding to web requests using a tool like `curl`.

```
$ curl -s http://localhost:9145/metrics
...
# HELP aerospike_node_ticks Counter that determines how many times the Aerospike
node was scraped for metrics.
# TYPE aerospike_node_ticks counter
aerospike_node_ticks 1
...
```

## Aerospike Exporter container

Instead of running the binary, you can choose to run the Aerospike exporter container. The container can be leveraged as part of a container running inside a pod with a Kubernetes deployment, as part of a docker compose file, or run ad-hoc using a `docker run` command which will be the fastest way to test it out. Aerospike publishes this container and provides **various environmental variables to configure the exporter**. If you run the Aerospike database as a local service and want to run the exporter in a container, the following command should work for you.

```
docker run --net=host --detach \
  -e AS_HOST=127.0.0.1 \
  -e AS_PORT=3000 \
  aerospike/aerospike-prometheus-exporter:1.14.0
```

This runs the `aerospike-prometheus-exporter` container version 1.14.0 and passes environmental variables (`-e`) to connect to the Aerospike database running at ip 127.0.0.1 on port 3000. This command also specifies it should run on the host network, `--net=host`, which allows the docker container to reach the host network instead of being stuck within the container itself. The `--detach` option is also passed here which allows docker to run this in the background.

```
$ systemctl status aerospike
...
Active: active (running) since Sat 2023-12-23 23:46:58 UTC; 4min 56s ago
...
$ sudo docker ps
...
f74a059aaaa1 aerospike/aerospike-prometheus-exporter:1.14.0 "/docker-
entrypoint..." About a minute ago Up About a minute thirsty_morse
...
```

Again you should test that it works by performing a web request against the exporter.

```
$ curl -s http://localhost:9145/metrics
...
aerospike_node_up{build="6.3.0.10",cluster_name="cakery",service="10.0.2.15:3000"} 1
```

## Metric Reference

The Aerospike exporter relies on the same protocol available to the Aerospike tools such as `asinfo` and `asadm`. The Aerospike database provides an `info` protocol and when the exporter needs to fetch information from the database it will perform those same `info` calls and then reformat them as web requests. Since the metrics from all exporters originate from the Aerospike `info` protocol, you can use the [metric reference guide](#) to help you find and understand their meaning.

With the Aerospike Prometheus exporter, the metrics are all prefixed with the name `aerospike_` followed by a metric *context* and finally the metric name and any specific labels. Picking a metric at random, the `aerospike_namespace_storage_engine_stripe_free_wblocks` metric is a good specimen to evaluate.

```
$ curl -s http://localhost:9145/metrics
...
aerospike_namespace_storage_engine_stripe_free_wblocks{cluster_name="cakery",ns="bar",service="10.0.2.15:3000",storage_engine="memory",stripe="stripe-0.0xad002000",stripe_index="0"} 63
aerospike_namespace_storage_engine_stripe_free_wblocks{cluster_name="cakery",ns="test",service="10.0.2.15:3000",storage_engine="memory",stripe="stripe-0.0xad001000",stripe_index="0"} 63
```

To look this metric up in the metric reference guide, you should be able to drop the `aerospike_` prefix and search for `storage_engine_stripe_free_wblocks` inside of the namespace context. In reality, this exercise is less than exact and requires a little guess work.

This particular metric is recorded in the metric reference as `storage-engine.device[ix].free_wblocks` which isn't the produced output! This is because of a compatibility change with some metrics – Prometheus does not support metric names with brackets, dots, or even dashes. You will find some of these metrics changed but the information is typically moved into the labels section, the area inside the braces `{..}` which is what happened with the device index `[ix]`.

Along with the complexity added with the compatibility changes, you will also find some of the fields returned by the exporter are configuration parameters that are not found in the metrics guide. For example, a commonly tuned parameter in the configuration file is the `defrag_lwm_pct`, which shows in our Prometheus metrics as `aerospike_namespace_storage_engine_defrag_lwm_pct`.

```
$ curl -s http://localhost:9145/metrics|grep ^aerospike_namespace_storage_engine_defrag_lwm_pct
```

```
aerospike_namespace_storage_engine_defrag_lwm_pct{cluster_name="cak-ery",ns="bar",service="10.0.2.15:3000",storage_engine="memory"} 50
aerospike_namespace_storage_engine_defrag_lwm_pct{cluster_name="cak-ery",ns="test",service="10.0.2.15:3000",storage_engine="memory"} 50
```

This metric reflects the configuration parameter `defrag-lwm-pct` which you can look up using the [configuration reference page](#). There is no straightforward way to identify whether the metrics are for runtime statistics or from configuration at query time, you will need to learn them or check both references.

## Alert Rules & Dashboards

Now that you have Prometheus set up to scrape all your Aerospike servers' exporters, you need to create alert rules for Prometheus and dashboards. The folks at Aerospike have created a set of rules you may want to import from [github](#). These alert rules cover a wide variety of problems, so you should experiment with the queries and understand their meaning. At a minimum every deployment should be monitoring Observability, which means the exporter is up and working, as well as Latency and Availability.

### Alert Rules

The first alert in the [github](#) repo is a good example of the syntax for creating those alert rules. There is a distinct name for the alert rule, `AerospikeExporterAgentDown`, and some more information under an indent, showing that this all belongs to that rule.

```
- alert: AerospikeExporterAgentDown
  expr: up{job="aerospike"} == 0
  for: 30s
  labels:
    severity: warn
  annotations:
    summary: "Aerospike Prometheus exporter job {{ $labels.instance }} down"
    description: "{{ $labels.instance }} has been down for more than 30s."
```

The first line will be the expression `expr` – the query to run. This expression will evaluate to True if the exporter is not responding. Scenarios where this happens are usually related to hardware failure, Prometheus target misconfiguration, or if the exporter failed to start entirely. If the expression evaluates to true for over 30s, then it will trigger the alert to generate the text with the severity defined which later can be consumed by some alerting or paging system.

If this is the first time you've seen PromQL (Prometheus Query Language), it is best to first get this data generated and inside Prometheus where you can experiment running queries while referencing the [Prometheus basic query guide](#).



The next entry is also a vital one, detecting the condition where the Aerospike exporter is online but the Aerospike database is offline or not responding. For brevity, the remaining snippets will be shortened to just the expression.

```
expr: aerospike_node_up{job="aerospike"} == 0
```

Scenarios where Aerospike is down but the exporter is not are usually caused by hardware issues, exporter misconfiguration such as not being able to authenticate to the cluster, or out-of-memory (OOM) events. Review the Aerospike server logs and system logs to determine the cause.

The next vital alerting rule you should monitor is that cluster integrity is *True* (1).

```
expr: aerospike_node_stats_cluster_integrity == 0
```

This could be 0 because the node is unable to communicate with all nodes in the cluster, or perhaps the cluster rejects the node's request to join for some reason such as time drift exceeding a threshold. You can review the Aerospike logs for reasons why this might be false, and may need to read the logs of the other nodes in the cluster to find the full picture why this may be.

Finally, the last table stakes alert rule to mention is latency. You need to ensure that every node in the cluster is responding within an appropriate time frame.

```
expr: histogram_quantile(0.99, (aerospike_latencies_read_ms_bucket{job="aerospike" }))) > 4
```

This expression will be true, and fire, if the 99th percentile latency exceeds 4ms for reads. There is a similar one for *writes* and you should customize this to fit your needs. This will fire if any particular *instance* has any namespace exceeding 4ms. You can adjust this by adding a *sum* to aggregate across labels if for example you wanted to only know that the entire cluster's 99th percentile is achieving the desired result instead of every node measured individually:

```
histogram_quantile(0.99, (sum by (ns,le)(aerospike_latencies_read_ms_bucket{job="aerospike" })))
```

## Dashboards

While you can create graphs and run queries on a Prometheus instance, you cannot create graphs across multiple Prometheus instances and multiple data sources, add custom visualizations, or save and share complex dashboards. Grafana is the tool usually paired with Prometheus for this purpose. Grafana can be connected to Prometheus, alongside other data sources, and then serve as the new front-end for running queries and dashboards. The [Aerospike monitoring repo contains dashboards](#) which can be imported and customized. By importing these dashboards, you'll be able to

track these metrics and correlate events together more easily and be up and running faster with less work building these gauges, graphs, and presenting these indicators.

## Software Updates

You must apply security and bug fix patches to the system on some cadence, for stability and security reasons. Aerospike posts [release notes on their website](#) which you should read if you think you may be encountering a bug which may already be fixed, or want to understand the changes in the newer version. The major version changes sometimes come with special upgrade requirements and procedures which you need to read and understand, such as dropping a certain feature or a requirement to sanitize your storage devices after taking the daemon down. From version 6.3, the Aerospike vendor will support and backport hotfixes for released versions for up to 2 years after the release.

You should understand the mechanical steps of a typical upgrade. The upgrade steps can vary depending on if you are using the Community Edition or Enterprise, if you allow downtime, if you have rack support, special upgrade requirements, or a special storage medium type. Reviewing various scenarios seems the most appropriate way to get started, and the first one you should know is the single rolling server restart.

## Preparation

Suppose you have a 3-node cluster, with a replication factor of 2. If you need to upgrade this cluster, you must remove 1 node from the pool which may drop your ability to handle requests by 33% ( $\frac{1}{3}$ ). With larger clusters, this number becomes smaller and less impactful but you must plan for it and understand the implications of removing a server from the cluster. Mainly, you need to make sure the other 2 instances in the server will be able to cope with the full load of requests and the replicated data size. A good rule of thumb is to apply this ratio to your various bottlenecks to determine ahead of time if you'll run into an issue.

$((n\text{-servers being removed} + \text{total-cluster-size}) / \text{total-cluster-size}) * \text{bottleneck\%}$

For example, say storage device space was your primary concern. If you have a 3-node cluster, with 34% storage space used, and needed to remove 1 node we should be able to forecast that easily as Aerospike balances things mostly linearly especially with the `prefer-uniform-balance` Enterprise feature:

$((1 + 3) / 3) * 34 = 45.34\%$

Finally, the mechanism of *migrations* will be kicking in to re-replicate and redistribute the data through the remaining 2 nodes which adds extra load. The migration process is tunable though, so you can mitigate this by [tuning migrations](#) to act slower and

even entirely stop it. However, you must be aware that this can cause some partitions to have and maintain an extra copy of data for some time period.

Now that you have your capacity checked and under watch, make sure that your cluster is stable. You should not remove servers from the cluster while the cluster is already undergoing migrations. You can utilize the `cluster-stable asinfo` command, which you can read more about in the `asinfo` command guide.

```
Admin+> asinfo -v 'cluster-stable:size=2'  
book-aerospike-1.book_default:3000 (172.28.0.3) returned:  
ERROR::cluster-not-specified-size
```

If the cluster size is not as specified, or migrations are ongoing, the `cluster-stable` command will inform you that things are not in a stable state. If migrations are ongoing, chances are there was a recent network interruption or node lost from the cluster and you should address that before proceeding.

## Upgrade

With the preparation out of the way, and your capacity being kept in check, you can now begin the upgrade. First you should understand how to upgrade a typical installation, without enterprise features and on a cluster with replication.

1. Stop Aerospike on the target node, `sudo systemctl stop aerospike`.
2. Perform any kernel or other software upgrades *yum upgrade*, *apt upgrade*, etc.
3. Install the new version of Aerospike using the appropriate OS specific command *dpkg -i*, *rpm -i*, *./asinstall*, etc.
4. Reboot if required. If kernel patches have been applied, this is usually required.
5. Start Aerospike `sudo systemctl start aerospike`.
6. Verify Aerospike has started, is healthy, and joined the cluster `sudo systemctl status aerospike`; `asinfo -v status`; `asadm -e "info"`; You should verify the server appears in `asadm` with all the other servers, and Cluster Integrity is True on all members if this is not monitored by your alerting system.
7. Wait for migrations to finish.

In a nutshell, that's the upgrade procedure. But, you should know about the pitfalls, how to avoid them, and how to utilize the enterprise license to move faster. The major points of contention and improvement are around steps 1 and 5.

Starting out with step 1, this is not a graceful handoff. When you have traffic flowing through the cluster and suddenly remove a server from the cluster, it causes a brief moment of impact. This is the same as unplugging a machine at random, which Aerospike handles quite well. The immediate impact is that in-flight transactions to

that instance fail, and for the next second or two the applications will still attempt to connect and time-out until the cluster and the applications recognize the server is gone. If you had any namespace with replication-factor of 1 this complicates the matter further. With the Enterprise Edition of Aerospike, we have the ability to quiesce a node which provides a smooth handoff of data and traffic which will be covered later.

Coming to step 5, you must be aware of potential *zombie records*. If you are using storage devices with persistence, there is a scenario where some records that have been deleted will be *resurrected* in a sense while the database rebuilds the index from the persistent device. This is because of the way Aerospike is optimized to write to the disk, where a delete won't immediately cause an operation to occur to a disk and expunge the data. The immediate result of a delete operation is that Aerospike will remove the entry from the primary index, but only later will it clean up and expunge the persisted data in a slow sweeping process called defragmentation.

This problem of zombie records can be addressed in two ways. First, you could wipe the storage devices before re-introducing the instance into the cluster and allow re-replication. This way no deleted records which should have been deleted can be re-introduced. Second, you could utilize the enterprise feature called tombstoning or durable-deletes. The durable-delete feature will write tombstones when a delete operation is performed. A tombstone is a marker that indicates the record should have been deleted and the time when. Tombstones are written to the storage to prevent the records from being reintroduced, allowing you to durably delete, but it causes the server to incur an IO operation so there is overhead.

Finally, the last couple of major ways we can improve this are through utilizing rack-awareness and rapid rebalance. With larger deployments, you can enable a feature called rack-awareness which allows you to group several systems into a single sort of failure zone or a kind of shard. The main guarantee of rack-awareness is that it guarantees the replica and master copy of a record cannot reside in the same rack. This allows you to upgrade clusters with more nodes faster, as you can take down much larger groups of servers at a time such as 10 servers at once without downtime or data loss.

Rapid rebalance is another enterprise feature that allows syncing of data faster. This is mainly useful if you are re-introducing a fast restarted Aerospike instance using the warm restart feature, or have restarted a server and are allowing it to read back from a storage device. As an instance comes back into the cluster carrying its own data, the cluster will perform comparisons on the data coming from that node to the current cluster's data. If there is a record that exists in the cluster already, and a record that exists on the node that was restarted there must be a resolution to which record to choose. Depending on the *conflict-resolution-policy* configuration specified, the cluster will choose either the existing or the incoming version of the record. With

Rapid rebalance, the cluster is able to do this in much larger batches and can bring migration time down from hours to minutes.



With migrations and this type of conflict resolution scenario, it should be noted that this is not a merge strategy. One version of the record will “win” and clobber the other version of the record. The two resolution methods are based on the generation counter or the last-update-time, usually encouraging the newer version of the record to be preferred. The cross-datacenter-replication (XDR) feature has similar guarantees with conflict resolution. If this is inappropriate for your application, you should avoid this scenario by not introducing potentially stale data into the cluster.

## Quiesce

Quiesce is an Enterprise feature that allows you to gracefully remove a server from the cluster. You can read the [full detailed guide on quiesce on the Aerospike website](#). When you quiesce a node, the cluster will immediately begin moving data off the node and the application transactions will slowly follow.

```
## Admin
Admin+> manage quiesce with <Node to remove>
Admin+> manage recluster
Admin+> sh stat like quiesce
```

And to undo this command:

```
Admin+> manage quiesce undo with <Node to undo quiesce>
Admin+> manage recluster
Admin+> sh stat like quiesce
```

Once you quiesce a node, you may need to allow the records and transactions to drain off of it. Depending on your replication and consistency requirements, this could take between a couple of minutes to a few hours. This allows smooth hand off of the data to another node, and prevents putting your cluster in a risky state. As this is an Enterprise-only feature, this strategy is something you should discuss with your Enterprise contact.

## Troubleshooting

Something eventually will go wrong, and when it does you need to be ready. This section will guide you on some of the common places to look for issues and what the appropriate response may be. Aerospike has [a troubleshooting guide published on their website](#) which may be more up-to-date than this one. There is no one prescribed solution to any particular issue, but instead you should have the tools and information at your disposal to find where the issue is.

*Chapter 7: Configuration* covers how to configure and access logging. Usually you'll be able to find your log at `/var/log/aerospike/aerospike.log` or `journalctl -u aerospike`. The log file should be your first stop for any issues related to clustering, cluster membership, or trying to understand the order of events in some problem. Along with the Aerospike logfile, you should be familiar with checking the health of the hardware and hardware logs. For physical machines this usually will be through a utility like `ipmitool` or `ras-mc-ctl`. For both physical and cloud machines, you should be able to find hardware related logs using the `dmesg` command and checking the kernel log.

Aside from looking at the log file of the problematic node, for clustering or membership related issues you should also explicitly locate and review the logs of the *principal* node. Aerospike uses a special protocol for cluster membership, called Paxos. With Paxos there is always an elected leader for the cluster which we refer to as the principal, who's additional job is to janitor and gate-keep cluster membership. It is important to be able to identify the principal node and be able to inspect it, as it will contain the logs and reasons for why membership was rejected.

The easiest method for identifying the principal node is using the `asadm info network` command and looking for the server listed in *green* with an asterisk next to it.

Once on the principal node, consider looking at clustering related messages `journalctl -u Aerospike |grep clustering|less` or non-informational messages `journalctl -u aerospike|grep -v INFO|less`.

With latency related issues, or throughput bottlenecks, you'll need to consider a whole host of potential bottlenecks. This is where supplemental monitoring and homogeneity will help you. The *microbenchmarks* referenced in Chapter 7 will also help you pinpoint.

Along with the Aerospike Prometheus exporter, you should consider running the *node* exporter. The node exporter will collect various statistics on server resources such as processor frequency, storage device latency, storage device queue depth, network throughput, and much more. With all these metrics, there are also pre-built and shared Grafana dashboards in the community to help graph and compare them.

Homogeneity across a cluster can provide invaluable insight into an issue and should not be undermined. If all the hardware in the cluster is the same, then all the latencies, cpu utilizations, disk utilizations, etc, should also be nearly the same. This is because of the proficient way that Aerospike balances traffic horizontally. If you observe that in your cluster of 10 nodes, for example, you see 1 node performing the same work but struggling much more than the other 9 nodes, it's likely you have some bad hardware there. Similarly, if you have a large deployment and see 1 *rack* struggling, you may be able to quickly identify a networking issue. This isn't

a hard requirement of Aerospike but we strongly recommend you keep the cluster homogenous if at all possible for running long-term clusters.

The Aerospike community is also developing features to help operators troubleshoot and maintain their clusters more efficiently by embedding new commands inside the `asadm` utility. The `health` command in `asadm` runs various checks across the cluster. It searches for skewed error rates and other common issues, then prints them out. The `show best-practices` command will review the configuration of the cluster and show warnings if certain parameters are not within Aerospike's recommendations.

Finally, the most important recommendation is planning. Work with your developers and have regular meetings on new features and requirements, run tests to determine the limits of your hardware, and stay ahead on capacity requests. This will help you avoid running into trouble.

## Chapter 8 final notes

You've learned about Prometheus and its exporters, how to use them to your advantage in comparing trends and monitoring live issues. You should understand how to reference metric names in the guide and find what they mean and how to create alarms on them, especially Latency, Availability, and Observability. You've learned how to upgrade Aerospike, along with pitfalls and ways to expedite the process. Common troubleshooting methods were also covered to ensure you are able to start pinpointing problems when they occur.

## About the Authors

---

**Dr. Srini V. Srinivasan** is the CTO and Founder of Aerospike. When it comes to databases, he is one of the recognized pioneers of Silicon Valley. Srini has two decades of experience designing, developing and operating highly scalable infrastructures. He also has over a dozen patents in database, web, mobile and distributed systems technologies. Srini co-founded Aerospike to solve the scaling problems he experienced with Oracle databases while he was Senior Director of Engineering at Yahoo.

**Tim Faulkes** is an enterprise application architect with over twenty years of global experience in delivering technical solutions to business problems at an enterprise level. His specialties include application architecture and design, technical team leadership, mentoring and educating developers and liaising between business and technical teams to ensure high value solutions are delivered in a timely and effective manner. Tim is currently the Chief Developer Advocate at Aerospike.

**Albert Autin** is the Lead Database Reliability Engineer at The Trade Desk. His current responsibilities include managing the data ingestion team, advising and optimizing on data structures and access patterns in the Aerospike ecosystem, and creating and testing new capacity planning methods. Albert also works with business and project leadership to plan and prioritize short and long term goals. He has over a decade of experience working with the Aerospike database.

**Paige Roberts** has worked as an engineer, trainer, support technician, technical writer, marketer, product manager, and a consultant in the last 25 years. She has built data engineering pipelines and architectures, documented and tested open source analytics implementations, spun up Hadoop clusters, picked the brains of stars in data analytics, worked in different industries, and questioned a lot of assumptions. Paige has worked for companies like Pervasive, the Bloor Group, Actian, Hortonworks, Syncsort, and Vertica. Now, she promotes understanding of Vertica, distributed data processing, open source, large-scale data engineering architecture, and how the analytics revolution is changing the world.