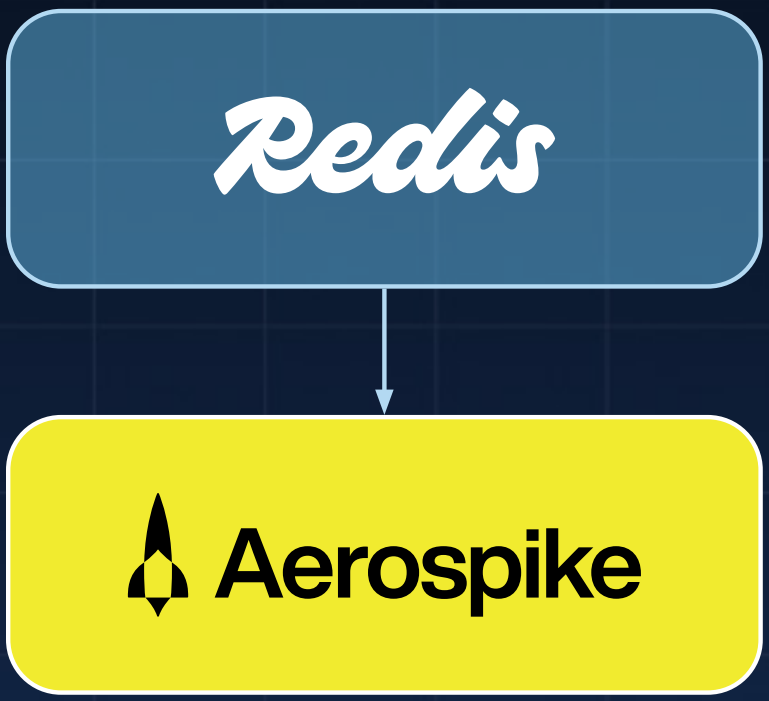
WHITE PAPER

Redis to Aerospike migration guide



Contents

Introduction

In today's fast-paced data-driven world, performance and scalability are crucial when selecting a database for your application. Redis and Aerospike are both high-performance databases that are designed to handle high throughput in distributed environments. Redis stores its data primarily in memory, limiting vertical scalability, whereas Aerospike is designed from the ground up to store data on SSDs with an architecture giving near-memory speeds across vast data volumes per node. Redis is commonly used for its versatile data types (strings, hashes, lists, sets, and more), while Aerospike is known for its strong consistency, scalability, and high availability for large-scale, real-time applications. However, Aerospike also supports versatile data types, arguably more flexible than Redis'.

This guide will walk you through the key concepts of Redis and compare them to Aerospike. It will show you which patterns and approaches to use to smoothly transition your data model and ensure minimal disruption to your application.

Why migrate from Redis to Aerospike?

Before diving into the migration process, it's important to understand why you might want to migrate from Redis to Aerospike. Here are a few reasons:

- **Scalability:** Aerospike is built to scale both vertically and horizontally across multiple nodes, providing superior scalability for larger datasets compared to Redis, which is constrained in vertical scalability by the amount of DRAM on a node.
- **Cost:** As Redis stores its data in memory for fast access, it tends to require nodes that are heavy in DRAM to store larger data sets. While Redis on Flash is an option in some installations, the integration with flash devices remains very much an afterthought and hence suffers significant performance degradation. Aerospike was designed primarily for flash devices and has very similar performance for data in memory and data on disk using its patented Hybrid Memory Architecture (HMA). This means that large data volumes up to multiple petabytes are possible cost-efficiently.
- **Consistency:** Aerospike supports both availability mode (favoring availability when cluster splits happen) and strong consistency mode, which guarantees that no acknowledged writes are lost, even in the face of cluster splits. Redis cannot guarantee strong consistency and hence must rely on another database to be the system of record.
- **Persistence:** Aerospike is designed for high persistence, offering both in-memory and disk-based storage, with HMA being the predominant use. Redis is traditionally an in-memory-only store, though it does offer persistence options with RDB snapshots, AOF logs, or Redis on Flash, although these have performance implications.
- **Rapid restart:** Aerospike uniquely offers fast restart capabilities that rapidly recover in-memory data and indexes after a node reboot, dramatically reducing downtime. Redis, by contrast, relies on slower RDB/AOF recovery.
- **Throughput:** Each shard in Redis is single-threaded to avoid locks. While this is an efficient strategy for dealing with concurrency issues, it does put an upper bound on the amount of throughput that can be achieved. Additionally, some Redis commands can take a reasonable time at scale. For example, Redis lists above a certain size are stored in linked lists, and traversing these lists takes CPU cycles. Other requests in the same shard block behind this traversal, lowering throughput and increasing latency. Aerospike is a true concurrent system with efficient use of locks, which does not have restrictions like this.

- **Advanced data model:** Aerospike supports features that Redis does not, including but not limited to:
  - Multi-record transactions across any records
  - Multiple columns in the record associated with a single key
  - Nested lists and maps allowing documents to be stored and manipulated in each column of a record
- **Durable and accurate caching:** When used as a cache instead of a database, Aerospike delivers precise eviction with per-set/table, while Redis uses approximate LRU based on sampling, often retaining stale data.

Redis has some powerful features and a rich data model that may need to be mapped into Aerospike's data model during migration. Aerospike also has a rich data model with some slight differences. A successful migration requires an understanding of how to map Redis data types onto Aerospike.

Key-value differences

Both Redis and Aerospike are key-value databases, but the implementations are different. Redis has every key being a string, and anything that is not a string gets converted into a string. Scalar values are similarly always stored as strings. It is not possible to use an integer as a key or a value, for example. Operations that operate on numeric types like INCR will convert the value to numeric, manipulate the value, and then write it back as a string. For example:

```
127.0.0.1:6379> set 1234 5678
OK
127.0.0.1:6379> set 1111 “Tim”
OK
127.0.0.1:6379> get “1234”
“5678”
127.0.0.1:6379> get “1111”
“Tim”
127.0.0.1:6379> incr 1234
(integer) 5679
127.0.0.1:6379> incr 1111
(error) ERR value is not an integer or out of range
```

Clearly, numeric operations perform string-to-integer conversions, including type checking.

Aerospike has a different structure. The “value” associated with a single key is more like a row in relational terms. It is composed of multiple values, with each value residing in its own bin (similar to a column in relational terms). So associated with a key like “Customer1234”, you might have bins for `firstName`, `lastName`, and `age`. Each bin is strongly typed to the value in that bin, so `firstName` and `lastName` would both be strings, and `age` would be an integer value. Aerospike knows the type of bins and prevents illegal operation on the bins. So you could add one to `age` (as it is stored as an integer), but not `firstName` (as it is a string). Since Aerospike knows the types of the bins, no conversion is necessary.

Note that while it is possible to store language-serialized objects into Aerospike in a BLOB field, this is not recommended. Using Aerospike's built-in types allows you to be language agnostic (writing in one programming language, reading in another), but also to take advantage of Aerospike's features like secondary indexes and bin projection.

Data grouping

People who are used to relational databases are used to having structures where they can group logically related records. For example, one MySQL instance might contain multiple databases, and each database might have multiple tables. Tables might be called "Customer," "Account," and so on to describe the data that will be stored in that table.

Relational databases also enforce schemas on the tables to ensure that all records in that table match the specified schema. A Redis data store contains no inherent groupings of records, like databases or tables. Instead, keys are standalone, and each key has a single value. However, as it is common to store multiple different business entities in Redis, application developers normally enforce a naming convention to mirror a structure. For example, keys might be "customer:1234", "account:555", and so on.

Keys are always strings in Redis, and a string can be up to 512MiB in length. Applications can choose a naming structure for keys that meets their requirements. Aerospike stores records similar to a relational database with some structure. Each key contains three distinct parts: a namespace, a set, and a key. These concepts need some explaining.

Namespace

A namespace is like a database or tablespace in relational database systems. It defines important properties like where the data is stored (in memory, on SSDs or PMEM), which SSDs it is stored on, the number of copies of data to store, and whether to favor availability or consistency in the face of a network split.

Namespaces are typically shared between different use cases. Since they rely on physical storage, they typically do not scale well as you add use cases. Technologies like Kubernetes, however, can change this sometimes as they virtualize storage. However, it's a good rule of thumb to use as you get started in Aerospike.

Sets

A set is very similar to a table in RDBMs. They are logical conglomerations of records with similar purposes, so there might be an Accounts set, a Customers set, and an Addresses set, for example. Each set belongs to a namespace, and a namespace can hold up to 4,095 sets.

Key

A key uniquely identifies a record within a set. Keys can be strings, integers, or BLOBs. This is analogous to a primary identifier in an RDBMS. So the "Tim" record might have a key of 123456 in the set Customers in the namespace of Apps.

The three parts of a key in Aerospike remove some of the naming conventions necessary to logically separate information in Redis. So a key like "Customer:123456" in Redis would translate fairly directly to a set of Customer and a key of "123456." Note that all keys in Redis are strings, whereas in Aerospike, the string "123456" is a different key from the integer 123456.

When migrating data from Redis to Aerospike, it is usual to refactor the keys with inherent groupings as shown above to use Aerospike sets and keys.

Sharding

Both Redis and Aerospike have the capability to store data distributed across a cluster of nodes, with multiple copies of each piece of data. Aerospike has this capability built-in as part of the database, and Redis through the Redis Cluster add-on. Both allow resiliency in the case of node failures and the ability to scale the cluster horizontally to store more data.

Sharding is the mechanism by which data is split into smaller, more manageable units called shards. Shards are the units which are moved between servers for resiliency.

Redis

When using more than one shard for Redis with Redis Cluster, the keys are hashed into "hash slots". A hash slot contains a collection of data and can be moved between servers. There are 16,384 hash slots, so if there are four nodes in the cluster, hash slots might be allocated like:

- Server 1: 0->4095
- Server 2: 4096->8191
- Server 3: 8912->12,287
- Server 4: 12288->16383

By default, the whole key is used as input into the hashing algorithm, so "customer:1234" might hash to 5,132 and be on server 2 in the example above, but "customer:1235" might hash to 10,453 and be on server 3.

There is a mechanism to override this for situations where it makes sense to keep multiple values on the same server. Similar to grouping data together into logical tables, as was mentioned above, this burden falls on the application developer. In this case, a part of the key can be designated as the part to hash in order to determine the slot. For Redis open source, this can be specified using {...} notation. For Redis Enterprise, there is the additional capability to specify a regular expression on the keyspace for hashing.

For example, consider a use case where it is desired to store a customer and their addresses in the same slot. The customer might have a key of "customer:1234", and the address might have a key of "customer:1234:address:1". These whole keys will be hashed independently and likely end up in different slots. To force them to the same slot, change them to be "customer:{1234}" and "customer:{1234}:address:1". With the addition of the braces, only "1234" will be used to compute the hash slot, forcing both into the same slot.

One of the primary reasons to force data into the same slot is to gain multi-record transactions. Redis supports multi-record transactions through the use of the MULTI/EXEC command, but only for records that are in the same slot. This is because each slot is effectively single-threaded, eliminating any form of race condition around the execution of the MULTI.

However, this slot mechanism can be arduous. Consider the classic case of wanting to transfer $100 between different accounts. One account must have its balance decreased by $100, and the other must have its balance increased by $100. Either both of these operations must succeed or neither of them (if failure occurs). This can only happen in Redis if they are in the same slot. As it is hard to predict which accounts will need to be involved in money transfers, all accounts would need to be placed in the same slot. Since Redis Cluster uses slots as its unit for sharding, all accounts will always exist on the same server, leading to scaling challenges.

An example of a multi-record transaction for Redis would be:

```
> MULTI
OK
> HSET user-profile:{1234} username “king foo”
QUEUED
> HSET user-session:{1234} username “king foo”
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

Note that the part of the key in braces (“1234” in this example) is the same between the different keys to force the keys into the same hash slot. If this were not the case, a CROSSSLOT exception would be thrown. For more detailed information on Redis clustering, see this link.

Aerospike

Aerospike also uses constant hashing to shard the data between servers, but unlike Redis it always uses the whole key as input to the hash. This removes the onus for the application developer to design a key that meets the requirements of transactionality.

There are 4,096 partitions in Aerospike, and to compute which partition a key falls into Aerospike hashes together the set (table) of the key, the key value and the type of the key, then takes 12 bits of the hash to determine the partition. Each partition has a master partition on a single node, typically with one or more replicas on other nodes. This algorithm means that keys are evenly distributed across all the nodes in the cluster, and moved between different nodes as the cluster size grows or shrinks.

Aerospike, from version 8.0 onwards, natively supports multi-record transactions when running in strong consistency mode. These transactions are independent of the partition the record exists in, hiding the sharding mechanism from the developer.

The same example shown for Redis might look like this in Aerospike:

```
WritePolicy wp = client.copyWritePolicyDefault();
wp.txn = new Txn();
client.put(wp, new Key(“appNs”, “user-profile”, 1234),
new Bin(“useranme”, “king foo”));
client.put(wp, new Key(“appNs”, “user-session”, 1234),
new Bin(“useranme”, “king foo”));
  client.commit(wp.txn);
```

Data expiry and eviction

Both Redis and Aerospike allow data to have an expiration date, after which the data will automatically be removed from the system. The difference between expiry and eviction is:

Expiry	Expiry is programmatically set as the desired life of the record. This is associated with the use case. For example, cookie data in an advertising use case might be kept for 30 days.
Eviction	Eviction happens when the system is low on resources, removing data automatically to make room for new writes.

There are some differences between the two systems in the handling of both expiry and eviction.

Expiry in Redis

Expiry in Redis can be set in either seconds using the EXPIRE command or milliseconds using PEXPIRE. The remaining time before a record expires can be retrieved with the TTL (Time to Live) command, with a value of -1 meaning to never expire.

```
127.0.0.1:6379> set mykey 1234
OK
127.0.0.1:6379> ttl mykey
(integer) -1
127.0.0.1:6379> expire mykey 60
(integer) 1
127.0.0.1:6379> ttl mykey
(integer) 58
127.0.0.1:6379> ttl mykey
(integer) 57
```

Anything that replaces the key's value will reset the TTL back to the default value (do not expire); however, altering the value does not reset the TTL. So, in the above example, setting mykey 12345 would reset the TTL to never expire, but incr mykey would leave the TTL unchanged.

There are a number of extra parameters that can be passed, changing when the expiry happens. These are:

- **NX:** Set expiry only when the key has no expiry
- **XX:** Set expiry only when the key has an existing expiry
- **GT:** Set expiry only when the new expiry is greater than the current one
- **LT:** Set expiry only when the new expiry is less than the current one

Additionally, there are other methods to set the expiry. One of the more useful ones is EXPIREAT, which takes the expiry timestamp as a parameter, rather than how far that is from now.

Aerospike

Aerospike's TTL is set in the expiration field of the WritePolicy passed to an update operation. The behavior is consistent irrespective of whether this is a rewrite of the record, an update to the record, or even a simple touch of the record; the new TTL will be applied by taking the passed value (in seconds) and expiring the record that many seconds in the future.

The code to set the expiration is thus co-mingled with the code to write the record:

Similarly, any time the record is read, the TTL is retrieved as part of the Record object returned:

```
Key key = new Key("test", "testSet", 1122);
WritePolicy wp = client.copyWritePolicyDefault();
wp.expiration = 60;
client.put(wp, key, new Bin("name", "Tim"));
```

The value passed in the `expiration` field has different behaviors depending on the passed value. These are:

```
Record rec = client.get(null, key);
System.out.printf("Name = %s, expiration = %d, TTL = %,ds\n",
        rec.getString("name"), rec.expiration, rec.getTimeToLive());
```

> 0	Actual TTL in seconds
0	Default to the namespace or set the level configuration variable default-ttl on the server. If both a set level and a namespace level default-ttl are set, the set level one overrides the namespace level one.
-1	Never expire the record
-2	Do not change the TTL when the record is updated.

Aerospike does not have conditional TTL settings similar to the NX, XX, GT, and LT options in Redis. However, these can be easily mimicked in many cases, although it is possible this will require an additional read to determine the current TTL. (If the read is purely to get the TTL, then the `getHeader` method should be used in preference to `get` as it avoids reading the data off storage and transmitting it.)

Also, Aerospike does not have a way of expiring the record at a particular date/time. Instead, this can be easily coded at the application level. For example:

One important thing to note about Aerospike's TTL is that it should never be reduced by programmatically setting a smaller TTL. For example, the following code is **bad:**

```
LocalDateTime time = LocalDateTime.of(2030, 12, 31, 23, 59, 59);
LocalDateTime now = LocalDateTime.now();
wp.expiration = (int)ChronoUnit.SECONDS.between(now, time);
```

The reason for this is fairly technical. If you're interested, read on, if not please skip to the next section!

```
wp.expiration = 60;
client.put(wp, key, new Bin("name", "Tim"));

wp.expiration = 1;
client.touch(wp, key);
```

Aerospike is a copy-on-write system with new versions co-existing on storage with older versions until they are cleaned up by the background defragmentation process. Rewriting the record with a shorter TTL will cause the older record to be eligible for defragmentation. When the new record with the shorter TTL expires, it will also be eligible for defragmentation. It is possible for the record with the shorter TTL to be cleaned up (defragmented) while the one with the longer TTL is still waiting to be cleaned up. If the node is restarted at this point, Aerospike will scan the disks looking for valid records. The original record will appear to be valid as itsTTL has not expired and there are no valid records which are newer than this record. Hence this record will re-appear in the database.

Eviction

Both Redis and Aerospike support removing data from storage when safety limits are breached in a process called eviction. Aerospike's algorithm effectively removes records that are closest to expiring. Records with no expiration will never be removed via eviction. There are several controls on how aggressive Aerospike is when evicting data, and the eviction process can be totally disabled if desired, or it can be disabled on a per-set basis. See aerospike.com/docs/database/manage/namespace/retention for configuration settings.

If writes continue to the database after the eviction threshold has been reached and eviction has been disabled, most of the data has no expiration, or expiration is not keeping up with the new writes, then eventually Aerospike will hit a stop-write condition, which will prevent the insertion of any new data into the database. This is obviously an undesirable situation, and proper monitoring and alerting are required to ensure that this situation is prevented.

LRU cache

One common use of Redis is as a cache. When being used as a cache, removing old records from the cache is typically done using a least-recently-used algorithm. One of the drawbacks with LRU caches, which persist data, is that the read of the data typically triggers a write, so the system knows which data was most recently used. This typically has performance implications.

Aerospike provides a very efficient mechanism for implementing this in a persistent manner, allowing true LRU semantics without having every read turn into a write. For full details about how this works, please see this article on the LRU cache capabilities.

Understanding Redis data types

Redis supports several data types, each used in different use cases. There are several of these, and knowing how to translate them into Aerospike is useful.

Strings

Strings are the basic unit of storage for scalars in Redis. If the value is a scalar type, both the key and the value must be strings. Non-string operations like INCR are supported, but Redis will convert the string to the needed datatype, perform the operation, and convert it back.

Redis Strings are limited to 512MB. This restriction applies to both the key and the value. However, very large strings can create memory fragmentation, especially if they're frequently changing, so it is common to limit size to no more than 100kB.

Aerospike also supports strings and other primitive types like integers and floats. It is recommended to store the appropriate type in a bin rather than as a string, as this allows operations on the types without needing to do type conversions.

Aerospike does have different limits. Each record has a maximum size of 8MB, so every string must be less than this size. Note that this is rarely a limitation. However, if very large strings (or other types) need to be stored, there are easy ways to break them into multiple smaller records and re-assemble them upon retrieval.

Hashes

A hash is a collection of key-value pairs that can be useful for storing multiple values associated with a single key, such as an object. Redis supports operations on hashes similar to those on individual elements. However, it is not possible to nest hashes or lists as values within a hash.

Aerospike's bins (columns) are already similar to a hash, with the bin name as the hash key and the bin value as the hash value. However, each bin in Aerospike can also be a map, very similar to a hash, and Aerospike supports complex operations on the maps. Additionally, maps can contain both lists and maps, allowing them to be nested to arbitrary depths. This means, in effect, that each bin in Aerospike can contain a document, stored as a sequence of nested lists and maps.

So, migrating a hash in Redis to Aerospike can either be done by having each hash key as a bin with the bin value being the hash value, or in a single bin with a map in it, mirroring the structure of the hash.

Lists

Redis lists are ordered collections of strings, similar to a queue or stack. Lists contain only strings and are implemented as doubly-linked lists, allowing insertion or removal of elements at either the head or tail of the list to be done in constant time (O(1)). However, accessing items in the middle of the list is done by traversing the list, taking O(n) time. Redis lists are limited to 2^32-1 (4,294,967,295) elements.

In contrast, Aerospike lists are more similar to arrays than lists; accessing any item in the array can be done in O(1) time. The items in a list can be of any supported type (strings, integers, doubles, booleans, BLOBs, lists, maps), and lists and maps can be nested inside other lists and maps to arbitrary depth. However, the lists in Aerospike form part of a record, so they are bounded by the maximum record size of 8MB.

Sets

Redis sets are unordered collections of unique strings and are useful for keeping track of unique items, such as all the unique IP addresses that have accessed a blog post. Multiple operations are available, including add to a set, determining set membership, and set operations, such as unions and intersections.

Aerospike does not natively support a "set" type like Redis, but there are two common ways to implement set-like behavior:

1. Use a list, and when inserting into the list, add the value(s) with the `ListWriteFlag` of `ADD_UNIQUE`. This will force Aerospike to keep the list containing only unique items, skipping the item if it already exists in the list, thereby providing set-like semantics. Making the `listOrder` by `ORDERED` will make the inserts and retrieval faster as it allows for binary search capabilities.
2. Utilize a map. This option is particularly useful when it is desired to store data against a set element. For example, if storing unique URLs in the set, it may be advantageous to be able to count how many times a particular URL has been encountered. This could be stored in the value associated with the map key. This option may be faster than using a list for large numbers of items in the collection, but only if storing the map in `KEY_ORDERED` or `KEY_VALUE_ORDERED` format.

Sorted sets

A sorted set in Redis stores its values in sorted order. As new entries are added to the set, they are intrinsically linked to their proper place in the list. Items are composed of a score and a member, and are sorted by the score. Various API calls exist to manipulate the sets by adding, retrieving, and deleting elements.

Aerospike does not have a concept of a sorted set; however, map operations allow similar behavior within a single record. This allows moderately complex operations analogous to those in the sorted set type, but they are constrained by the maximum size of a record (8MB).

Geospatial

Geospatial information is used to store mapping coordinates, polygons, and so on, representing points on the Earth's surface. Redis supports this primarily by storing coordinates in a geospatial index, allowing calculations primarily between the coordinates, such as GEODIST (the distance between two points in a geospatial index) and GEOSEARCH, which locates members inside an area of a box or a circle.

Aerospike also supports GeoJSON with the ability to do secondary index queries on them. Two main queries exist:

- When GeoJSON coordinates are stored in records, an arbitrary region can be searched for points in that region. This region can be complex, like a USA zip code, or as simple as a circle.
- When GeoJSON polygons or circles are stored in records, an arbitrary point can be queried for which regions contain that point.

Bitmaps

Redis and Aerospike both support direct bitmap manipulation. In Redis' case, the bitmap is on a key that can be up to 512MB in size, allowing for up to 4 billion bits to be set. Aerospike's bitmaps exist within a single record, allowing bitmaps of up to 8MB. Larger bitmaps can be accommodated by simply segmenting the bitmaps across multiple records; however, in practice, it is rare to use such large bitmaps.

Migrating code from Redis to Aerospike

When migrating the codebase from Redis to Aerospike, the first decision is whether to change records to match the more flexible format that Aerospike supports. As mentioned previously, Aerospike records are identified by a tuple consisting of the namespace, the set, and the id. In Redis, these are typically encoded in the key.

Consider an application that stores customer data, including an address. In Aerospike, you would probably store this in the customer set in an appropriate namespace, with an id appropriate for that record. There would be multiple bins, most likely one per attribute in the business model. The address is probably aggregated with the customer, so that removing the customer should remove the address.

In SQL, the customer and the address would be stored in separate tables, with a foreign key defined between them and `DELETE CASCADE` defined on the foreign key. In Aerospike, the address would likely be defined as a map inside the customer, with a structure similar to:

```
Key("test", "Customer", 1234) → {
    id: 1234,
    firstName: "John",
    lastName: "Doe",
    dateOfBirth: 478934614000,
    address: {
       line1: "123 Main St",
       city: "Denver",
       state: "CO"
       zipcode: "80221"
    }
}
```

The beauty of encapsulating this in one record is that you get all the information about the customer in a single read, and removing the record automatically removes the address.

In Redis, there are several main ways of storing the information:

1. **Using multiple keys to store individual fields**. This might look like:

```
customer:1234:firstName="John"
customer:1234:lastName="Doe"
customer:1234:dateOfBirth="478934614000"
customer:1234:adddress:line1="123 Main St"
customer:1234:adddress:city="Denver"
customer:1234:adddress:state="CO"
customer:1234:adddress:zipcode="80221"
```

   The drawbacks of this approach are that there are lots of keys to store the business object; hence, reading, changing, or deleting the items would require passing multiple keys to the Redis calls. Additionally, transactionality might be important to the use case, so that you want to be able to update `firstName` and `lastName` atomically together. This is possible in Redis using "hashtags" with `{customer:1234}` becoming the hashtag. This does require pre-planning of access patterns.

2. **Storing everything in a single string (e.g., JSON format).** In this case, there is just one key, and hence, everything can be retrieved and updated atomically. However, parsing and re-forming the string on every read and write becomes painful and potentially a maintenance issue, and there is no easy way just to change one part, such as the `firstName`. For example:

```
customer:1234='{"id":"1234","firstName":"John","lastName": "Doe","dateOfBirth":
"478934614000","address": {"line1": "123 Main St","city":"Denver", "state":"CO",
"zipcode":"80221"}}'
```

3. **Using hashes**. As discussed above, a hash is like a map or a dictionary. It contains multiple key/value pairs with both the key and the value being strings. For example:

```
customer:1234={
    "id": "1234",
    "firstName": "John",
    "lastName": "Doe",
    "dateOfBirth": "478934614000"
}
```

4. **Use a combination of the above.** The eagle-eyed readers will notice that the hash approach did not touch on the address associated with the customer. This is because hashes cannot be nested in Redis, so to store the address, it either needs to be stored as a JSON string inside the hash (option 2 above), or to have two separate keys, one for the customer and one for the address (option 1 above).

How you've used Redis will determine how you migrate that data to Aerospike, but chances are that your approach will fall into one of the above four buckets.

Connecting to the database

Both Redis and Aerospike require a connection to the database before executing requests. Surprisingly, Redis does not come with a standard driver for client languages, but there are a slew of them in the public domain. The examples here will be in Java, and the two main drivers to access Redis in Java are Jedis and Lettuce. We will use Lettuce here, but both have fairly similar capabilities. Lettuce supports synchronous, asynchronous, and reactive usage. The Aerospike client for Java supports synchronous and asynchronous usage, with reactive being a separate driver.

In your POM file, ensure you have imported the Aerospike client. The version to import depends on the version of your database and your JDK. Use these rules to determine what to import:

- If you're using a client version that precedes version 8, import `aerospike-client`. Note that if you're new to Aerospike, you should be using the latest version of both the database and the client library to ensure you get the latest features. At the time of writing, 9.0.4 is the latest version of the java client, and the latest database version is 8.0.0.7. For example, versions of the client prior to version 9 do not support ACID transactions.

```
<dependency>
    <groupId>com.aerospike</groupId>
    <artifactId>aerospike-client</artifactId>
    <version>7.2.2</version>
</dependency>
```

- If you're using a client version of 8 or later and you're using JDK 21 or later, use `aerospike-client-jdk21` version. This supports virtual threads, which offer a significant performance boost on the client-side if enabled in your JDK.

```
<dependency>
    <groupId>com.aerospike</groupId>
    <artifactId>aerospike-client-jdk21</artifactId>
    <version>9.0.4</version>
</dependency>
```

- Otherwise, use `aerospike-client-jdk8:`

```
<dependency>
    <groupId>com.aerospike</groupId>
    <artifactId>aerospike-client-jdk8</artifactId>
    <version>9.0.4</version>
</dependency>
```

In Redis, connecting to a database is simple, but requires code for connection management. For example, to connect to a cluster with no username or password, you can use:

```
RedisURI uri = RedisURI.Builder.redis(host, port).build();
RedisClient client = RedisClient.create(uri);
StatefulRedisConnection<String, String> connection = client.connect();
RedisCommands<String, String >commands = connection.sync();
```

Aerospike also supports connecting simply to the cluster and will manage the connection pooling for you. Additionally, it will monitor the state of the cluster and will automatically adjust which node requests are sent to as the cluster scales up and down. (Clustering is a primary function of the Aerospike database.):

```
this.client = new AerospikeClient(host, port);
```

This is a very simple example and probably isn't that useful in production. If the host passed to the constructor is down, the client will not be able to attach to the cluster. So, if there are five nodes in the cluster and the one you specify is down, your application cannot connect. However, if the host refers to a DNS address mapped to a group of nodes, then this is not a concern.

If you want to use security in your application or otherwise control how the client is created, you can pass an instance of a `ClientPolicy.`

```
ClientPolicy clientPolicy = new ClientPolicy();
clientPolicy.minConnsPerNode = 10;
clientPolicy.user = “admin”;
clientPolicy.password = “admin”;
Host[] hosts = new Host[] {
    new Host(host1, port1),
    new Host(host2, port2)
};
this.client = new AerospikeClient(clientPolicy, hosts);
```

The provided 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 it. 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, and if `minConnsPerNode` is set (as in the above example), it will establish multiple connections to all nodes in the cluster.

Basic operations

Some use cases for Redis simply use it to put and get data. In this case, the usage in Aerospike is very similar to that of Redis – just use the put and get APIs. (Note that Redis uses `set`, Aerospike uses `put`.) The key structure will need to be decided on, as discussed in the "Data grouping" section above.

In Redis, this would look like:

```
commands.set("test.basic.key1", "some value");
System.out.println(commands.get("test.basic.key1"));
```

Whereas Aerospike would look like:

```
Key key = new Key("test", "basicSet", "key1");
client.put(null, key, new Bin("data", "some value"));
System.out.println(client.get(null, key).getString("data"));
```

There are a few key differences to pay attention to:

- Aerospike manages connections to the cluster for you. A simple call like put or get will cause a connection to be allocated from an existing connection pool, going to the correct node, or a new connection to be allocated if there aren't enough in the pool.
- As mentioned earlier, keys in Aerospike require a namespace, a set name, and an id. In this case, the id is a string, but integers and BLOBs are supported too.
- Records in Aerospike contain multiple columns (bins). These will be created when used and do not have to be pre-defined in any form of schema. In this case, the bin holding the information is "data"
- When retrieving the information from Aerospike, a get call returns the whole record by default. The code knows the information will be in the "data" bin and the type of this bin in String, hence the call to `getString("data").` It is also possible to specify just the bins you want returned, making the last line:

```
client.get(null, key, "data").getString("data");
```

Batch operations

Redis has commands like `mget` and `mput`, which can read and write multiple keys in a single command. This can be used for bulk load operations and efficient retrieval of data. An example is:

```
commands.mset(Map.of(
    “test.basic.key1”, “some value”,
    “test.basic.key2”, “some value 2”,
    “test.basic.key3”, “12345”));
System.out.println(commands.mget(
    “test.basic.key1”,
    “test.basic.key2”,
    “test.basic.key3”));
```

Aerospike also supports batch operations. The batch get command returns an array of records as expected, but the batch write is more complex than that of Redis, as it is more powerful. Not only can it write values into bins, but any single record operations are permissible. This includes actions such as:

- Increasing and decreasing numeric values
- Adding items to a list
- Inserting items into a map
- Removing bins

Operations will be discussed below in the section on “The power of `operate`”.

There are a few flavors of batch write calls, but the most common two are:

- ```
  operate(  BatchPolicy batchPolicy,
            BatchWritePolicy writePolicy,
            Key[] keys,
            Operation... ops)
  ```

  This overload applies the same set of operations to all the keys in the list. An example of when you might want to use this would be an inventory management system. When a customer checks out with five items in their cart, the available inventory count of each item should be decremented by one.

- ```
  operate(BatchPolicy policy, List<BatchRecord> records)
  ```

  This overload allows totally different operations to be associated with different keys. The `BatchRecord` encompasses a Key, the type of operation (write, delete, etc), and possibly the operation(s) to be performed. So it could write two bins on one record, increase an integer bin on a different record, delete a third record, and so on.

An example of a batch write and batch read in Aerospike is:

```
// Batch write
Key key1 = new Key("test", "basicSet", "key1");
Key key2 = new Key("test", "basicSet", "key2");
Key key3 = new Key("test", "basicSet", "key3");
List<BatchRecord> writes = List.of(
   new BatchWrite(key1, Operation.array(
       Operation.put(new Bin("data", "some value 1")))),
   new BatchWrite(key2, Operation.array(
       Operation.put(new Bin("data", "some value 2")))),
   new BatchWrite(key3, Operation.array(
       Operation.put(new Bin("data", 12345))))
);
client.operate(null, writes);

// Batch read
Record[] records = client.get(null, new Key[] {key1,  key2, key3});
System.out.println(Arrays.asList(records));
```

List operations

Both Aerospike and Redis have lists. Redis contains explicit functions in its API, whereas Aerospike uses operations to manipulate the lists. Multiple operations can be performed on a single key atomically to form arbitrarily complex operations. Let's look at some examples.

LLEN

Returns the length of a list.

```
commands.llen(key)
```

Aerospike:

```
client.operate(null, key, ListOperation.size(binName))
```

LPUSH

Adds a new element to the head of a list.

```
commands.lpush(key, "item0")
```

Aerospike:

```
client.operate(null, key, ListOperation.insert(binName, 0, Value.get("item0")))
```

LPOP

Removes and returns an element from the head of a list.

```
commands.lpop(key)
```

Aerospike:

```
client.operate(null, key, ListOperation.removeByIndex(binName, 0, ListReturnType.VALUE))
```

Note that by default, Aerospike will return a count of the elements removed, but if you want the values, then pass a ListReturnType of VALUE. Other options exist, too, like COUNT for how many items are returned (0 or 1 on this API call).

LRANGE

Extracts a range of elements from a list.

```
commands.lrange(key, 3, 6)
```

Aerospike:

```
client.operate(null, key, ListOperation.getByIndexRange(binName, 3, 4, ListReturnType.VALUE))
```

Redis' `LRANGE` command takes the start and end index (both inclusive of the end points), whereas Aerospike's `getByIndexRange` call takes the start index (inclusive) and the number of elements to return. It's easy to calculate the length: it's just `end-start+1.` Again, the ListReturnType parameter dictates what to return, but items like `COUNT` make more sense with a range.

There is also an `INVERTED` option, which, on all calls that take a ListReturnType parameter, can return everything except what was asked for.

So, if the list is:

```
[item1, item2, item3, item4, item5, item6, item7, item8, item9, item10]
```

Then the following code will return the selected items: `[item4, item5, item6, item7]`

```
client.operate(null, key, ListOperation.getByIndexRange(binName, 3, 4, ListReturnType.VALUE |
ListReturnType.VALUE))
```

However, if the `INVERTED` flag is passed, then the result will be `[item1, item2, item3, item8, item9, item10]`

```
client.operate(null, key, ListOperation.getByIndexRange(binName, 3, 4, ListReturnType.VALUE |
ListReturnType.INVERTED))
```

LTRIM

Reduces a list to the specified range of elements.

```
commands.ltrim(key, 2, 7)
```

Aerospike:

```
client.operate(null, key, ListOperation.removeByIndexRange(binName, 2, 6, ListReturnType.VALUE |
ListReturnType.INVERTED))
```

Aerospike does not have a “remove everything except this range” operation. However, the `INVERTED` flag discussed above alters the `removeByIndexRange` function to remove everything except those items.

The power of `operate`

In the above discussions on list operations, it is readily apparent that both Redis and Aerospike have fairly similar commands, but the Redis ones use a more brief syntax. Almost everything you can do on a record is wrapped in a single verb: Operate. Why is this?

Fundamentally, Aerospike’s data model is richer than Redis’, as a single record contains multiple columns (called bins) and these can each be simple, like strings and integers, or complex, hierarchically nested lists and maps. The examples above simply show a single `Operation` being passed to the operate command, always working on the same bin. This is contrived, though. Imagine a situation where you want to keep the top 10 scores for an event, as well as the number of scores and the total of all scores (allowing you to work out the average score).

In Aerospike, this would be one atomic operation on the server on receiving each score, using code like:

```
client.operate(null, key,
       Operation.add(new Bin("count", 1)),
       Operation.add(new Bin("sum", score)),
       ListOperation.append("topScores", Value.get(score)),
       ListOperation.removeByRankRange("topScores", -10, -1,
             ListReturnType.NONE | ListReturnType.INVERTED)
   );
```

There are three bins being updated in four operations: the `count` bin gets incremented by one, the `sum` is increased by the new score, the new score is added to the list of `topScores`, and then the list of `topScores` is truncated to the 10 highest values.

In some ways operate is like doing a `MULTI` in Redis, but all the operations are specified in a single command and must all operate on the same key. All of the specified operations will be done atomically on the server in the order they are listed in the `operate` command.

Hash operations

HSET

Sets the specified fields to their respective values in the hash stored at key

```
commands.hset(key, "name", "Bob");
```

Aerospike:

```
client.operate(null, key, MapOperation.put(K_ORDERED_MAP_POLICY, binName, Value.get("name"), Value.
get("Bob")));
```

In Aerospike, you will notice that List and Map operations, in particular, quite often take a `Value` class instance. This is a wrapper to make sure the correct type can be passed to the server without requiring a large number of method overloads. Remember, in Aerospike, map keys can be integers, strings, or BLOBs, and map values can be any supported type.

HGET

Returns the value at a given field

```
commands.hget(key, "name")
```

Aerospike:

```
client.operate(null, key, MapOperation.getByKey(binName, Value.get("name"), MapReturnType.VALUE))
```

HINCRBY

Increments the value at a given field by the integer provided.

```
commands.hincrby(key, "age", 1)
```

Aerospike:

```
client.operate(null, key, MapOperation.increment(K_ORDERED_MAP_POLICY, binName, Value.get("age"),
Value.get(1)))
```

Aerospike will validate that the passed types make sense for the operation; for example, incrementing a string will result in an error, but incrementing an integer will work as expected. Note here that the number 1 is passed to `Value.get()`, passing 1 as a string would result in a server error, as incrementation requires an integer (so `Value.get("1")` is incorrect).

HMGET

Returns the values at one or more given fields.

```
commands.hmget(key, "name", "age")
```

Aerospike:

```
client.operate(null, key, MapOperation.getByKeyList(binName, List.of(Value.get("name"), Value.
get("age")), MapReturnType.KEY_VALUE))
```

Note that by default, just getting the bin containing the map will return all entries in the map, similar to `hgetall` in Redis, and is typically used more often than selecting individual keys.

Aerospike also has a very feature-rich API, which can do far more than is shown here. For example, if you need to return a set of key-value pairs whose value is between two parameters, there's an API call for that. A good example of this is storing credit card transactions in a map, with the map key being the time of the transaction, and the value containing the amount of the transaction. This allows you to easily query transactions between $500 and $1000, for example.

Set operations

Redis has built-in set operations, allowing operations like adding to set, testing for set membership, computing the intersection of two sets in the same hash shard, and so on. Aerospike has no native “set” type. However, there are easy ways to mimic this. Let’s look at some common set operations.

SADD

Adds a new member to a set

```
commands.sadd(key, “tim”);
```

Aerospike:

As mentioned above, Aerospike has no in-built “set” type. However, it is easy enough to make a list behave as a set. All we need to do is tell Aerospike that there should be only unique items in the list (`ADD_UNIQUE`), if it tries to insert an element which is already in the list just fail silently (`NO_FAIL`) and if we’re trying to insert multiple items into the set and some of them already exist then just add the unique ones (`PARTIAL`). This is all specified on a `ListPolicy` instance:

```
ListPolicy setPolicy = new ListPolicy(ListOrder.ORDERED,
    ListWriteFlags.ADD_UNIQUE |
    ListWriteFlags.NO_FAIL |
    ListWriteFlags.PARTIAL);
```

Once this is set up, just use the list as a set:

```
client.operate(null, key, ListOperation.append(setPolicy, binName, Value.get(“tim”)));
```

SREM

Removes the specified member from the set.

```
commands.srem(key, “bob”);
```

Aerospike:

```
client.operate(null, key, ListOperation.removeByValue(binName, Value.get(“bob”), ListReturnType.
NONE));
```

SISMEMBER

Tests a string for set membership.

```
commands.sismember(key, “joe”);
```

Aerospike:

```
client.operate(null, key, ListOperation.getByValue(binName, Value.get(“joe”), ListReturnType.EXISTS)
```

SCARD

Returns the size (a.k.a. cardinality) of a set.

```
commands.scard(key);
```

Aerospike:

```
client.operate(null, key, ListOperation.size(binName)).getLong(binName)
```

Sorted set operations

While Aerospike has no built-in concept of sorted sets, maps can be sorted, either by the key or the value. This makes them good for use in applications requiring sorted sets, such as leaderboards. However, where they fall down is that a single map key can only exist in the map once. Leaderboards, however, require multiple players to be able to have the same score.Aerospike:

There are a couple of approaches to this that can be used. These are:

1. The score is the map key, and the value contains a list of all the players with that score. This works reasonably well. However, some common operations, such as finding the N highest scores, become problematic.
2. Using a compound key, consisting of the score concatenated with the player id in a fixed string format. The value then becomes the player id. This allows a range of common leaderboard-style queries to be answered.

In this example, we will use the second approach. Redis’ sorted set takes the score as a double, so we will mirror those semantics. We will need two helper methods:

```
private String formCompositeKey(double number, String value) {
   String result = String.format(“%013.4f|%s”, number, value);
   return result;
}
```

This can then be used to add to our map:

```
private Operation sortSetAdd(String bin, double number, String value) {
   String compositeKey = formCompositeKey(number, value);
   return MapOperation.put(K_ORDERED_MAP_POLICY, bin, Value.get(compositeKey), Value.get(value));
}
```

Compare this to the same code in Redis:

```
client.operate(null, key, MapOperation.create("map", MapOrder.KEY_ORDERED),
       sortSetAdd("map", 10d, "Norem"),
       sortSetAdd("map", 12d, "Castilla"),
       sortSetAdd("map", 8d, "Sam-Bodden"),
       sortSetAdd("map", 10d, "Royce"),
       sortSetAdd("map", 6d, "Ford"),
       sortSetAdd("map", 3d, "Zeus"),
       sortSetAdd("map", 10d, "Zac"),
       sortSetAdd("map", 14d, "Prickett")
   );
```

Now let's look at some operations on the set:

```
commands.zadd(key,
       10d, "Norem",
       12d, "Castilla",
       8d, "Sam-Bodden",
       10d, "Royce",
       6d, "Ford",
       3d, "Zeus",
       10d, "Zac",
       14d, "Prickett");
```

ZRANGE

Returns the specified range of elements in the sorted set stored at <key>.

```
commands.zrange(key, 0, 4)
```

Aerospike:

```
client.operate(null, key,
       MapOperation.getByIndexRange("map",
               0, 5, MapReturnType.VALUE))
```

Note that in Aerospike, the end parameter of `getByIndexRange` is exclusive, whereas it is inclusive in Redis, so it becomes one bigger in Aerospike.

ZREVRANGE

Returns the specified range of elements in the sorted set stored at the key. The elements are considered to be ordered from the highest to the lowest score. This can also be done by using the `REV` parameter to `ZRANGE.`

```
commands.zrevrange(key, 0, 4)
```

Aerospike:

```
client.operate(null, key,
       MapOperation.getByIndexRange("map",
               -5, -1, MapReturnType.VALUE))
```

ZRANGEBYSCORE

Returns all the elements in the sorted set at the key with a score between passed minimum and maximum parameters (inclusive). This command can also be done by ZRANGE.

```
commands.zrangebyscore(key, 5, 10)
```

Aerospike:

```
client.operate(null, key,
       MapOperation.getByKeyRange("map",
               Value.get(formCompositeKey(5, "")),
               Value.get(formCompositeKey(10, "~")),   // Use a high character to make inclusive
               MapReturnType.VALUE)
```

ZRANK

Returns the rank of a member in the sorted set stored at key, with the scores ordered from low to high.

```
commands.zrank(key, “Sam-Bodden”)
```

Aerospike:

```
client.operate(null, key,
       MapOperation.getByValue(“map”, Value.get(“Sam-Bodden”), MapReturnType.REVERSE_RANK))
```

Unsupported operations

There are a few things that Redis does that Aerospike does not support. These fall into two broad categories:

1. **Blocking operations:** Some operations in Redis are designed to block until an item is available. For example, BLPOP removes and returns an element from the head of a list. If the list is empty, the command blocks until an element becomes available or until the specified timeout is reached.

   No operations in Aerospike are designed to block due to the limitations this can place on scalability. The only exception to this rule is if multiple operations attempt to update the same record at the same time. They will be automatically queued behind one another to ensure atomicity of updates.

   Hence, blocking operations typically require polling the database. This is simple to arrange and control at the application level.

2. **Multiple record commands.** Redis supports several commands that automatically manipulate data across multiple records. For example, LMOVE transfers items in a list from the source to the destination, where the source and destination keys can be different. In Aerospike, this would require two different operations, one to the source list and one to the destination list, but these could be done atomically using ACID transactions for consistency.

   In Redis, these commands work great in a single node instance, but with Redis Cluster you have to ensure both source and destination exist in the same slot. Otherwise, you will receive the dreaded CROSSSLOT error. This requires pre-planning of operations to be performed in your database and the use of hashtags to resolve. Sometimes these cannot be resolved without significantly impacting the scalability of Redis.

3. **Pub/Sub commands.** Redis supports a publish and subscribe mechanism similar to Kafka. Aerospike does not support this functionality inherently; however, adapters to perform change data capture (CDC) style operations with queueing systems like Kafka, JMS, Flink, etc. are available. This gives flexibility in messaging choice and allows the database to focus on being a database rather than a queueing system.

Query operations

To query data across multiple keys in Redis, there are several different options. These include:

- Using sorted sets to simulate numeric indexes across keys, with the values stored in the sorted set
- Searching through hash or JSON keys using FT.CREATE and FT.SEARCH

- Using List for creating simple iterable indexes and last N items indexes.

For a full list of the options, please see the Redis documentation on secondary indexes and the Redis Query Engine.

Aerospike supports queries through the use of secondary indexes similar to a relational database. These indexes are defined on a particular bin within a set and allow efficient matching of records with a particular criteria. There are three sorts of index queries supported:

- **Equals** comparison against string, BLOB, or numeric indexes.
- **Range** comparisons against numeric indexes. Range result sets are inclusive, that is, both specified values are included in the results.
- **Point-In-Region** or **Region-Contain-Point** comparisons against geo indexes.

Let's see this in action.

Query example

We will create a simple class of a Car which will have an id, make, and year built:

```
public record Car (int id, String make, int year) {}
```

First, test data needs to be inserted. This can just be random data over a handful of car makers at this point:

```
final var makers = List.of("Holden", "Ford", "Toyota", "Kia", "Byd");
final var rand = ThreadLocalRandom.current();
// Seed 100 sample cars
IntStream.range(0, 100)
   .mapToObj(i -> new Car(i, makers.get(rand.nextInt(makers.size())),1980+rand.nextInt(46)))
   .forEach(car -> client.put(null,new Key("test", "cars", car.id),new Bin("make", car.make()),
                  new Bin("year", car.year())));
```

The index now needs to be defined. This step can occur either before or after the data has been created. In this case, we will do it in code, but it is normal practice for indexes to be created in Aerospike's administration tool, `asadm`, by a DevOps process.

```
var task = client.createIndex(null,
        "test",       // Namespace name
        "cars",       // Set name
        "cars_idx",   // Name of this index
        "year",       // The bin to be indexes
        IndexType.NUMERIC);
task.waitTillComplete();
```

Note that the type of the index (NUMERIC) must match the information in the bin. So if a car had a year inserted as “2025” (a string) instead of 2025 (a number), then that record would never be returned from the index.

Finally, the code can be written to use the index and query the data. In this case, we’re going to retrieve all the cars made between 2010 and 2020 (inclusive):

```
var stmt = new Statement();
stmt.setNamespace(“test”);
stmt.setSetName(“cars”);
stmt.setFilter(Filter.range(“year”, 2010, 2020));
try (var results = client.query(null, stmt)) {
   while (results.next()) {
       System.out.println(results.getRecord());
   }
}
```

When you run this code, you get output similar to:

```
(gen:4),(exp:0),(bins:(make:Kia),(year:2015))
(gen:4),(exp:0),(bins:(make:Byd),(year:2010))
(gen:3),(exp:0),(bins:(make:Byd),(year:2014))
(gen:3),(exp:0),(bins:(make:Toyota),(year:2019))
...

(Your actual results will differ as the data is randomly generated.)
```

Filter expressions

Secondary indexes like the one above can only act on a single predicate (IF condition clause). In this example, the predicate is “year BETWEEN 2010 AND 2020”. But what if we wanted only Fords which were manufactured in these years, so effectively “make = ‘Ford’ AND year BETWEEN 2010 AND 2020”?

Every data operation in Aerospike has the ability to take a `filterExpression`. Filter expressions perform arbitrary calculations on the information within a record to determine whether the operation should be performed. For example, you might want to update the status field of a record to `COMPLETE`, but only if the current `status` is PENDING. To do this, you would use a put operation with a filter expression to ensure the `status` is currently `PENDING`. If the filter expression returns `false`, the operation is not performed.

In this case, we just need to attach a filter expression to the QueryPolicy object passed to the query operation, so that any records that match the secondary index query but do not match the filter expression are thrown away:

```
var qp = new QueryPolicy();
qp.filterExp = Exp.build(
        Exp.eq(
            Exp.stringBin("make"),
            Exp.val("Ford")
        ));
try (var results = client.query(qp, stmt)) {
   while (results.next()) {
       System.out.println(results.getRecord());
   }
}
```

Now the output will look like:

```
(gen:3),(exp:0),(bins:(make:Ford),(year:2014))
(gen:3),(exp:0),(bins:(make:Ford),(year:2011))
(gen:4),(exp:0),(bins:(make:Ford),(year:2018))
```

Filter expressions are a very powerful feature, effectively allowing every data operation to have an IF conditional on it.

Extra Aerospike operations

One largely overlooked feature of Aerospike is its ability to perform filtering on almost all operations. Consider an example where a customer has accounts, and it is desired to get a list of all accounts for a particular customer that have a balance of over $1,000. This would require a read of the customer to get a list of the ids of their accounts, then a batch read of the accounts using the keys. However, rather than getting the accounts back to the client and applying the filtering logic at the client, Aerospike Expressions can perform this filtering server-side, minimizing the amount of client-server network traffic and client complexity.

Data migration

The other component that needs consideration is migrating data from Redis to Aerospike. This may or may not be an option depending on the durability of the data. For example, if Redis is being used as a cache for transient data such as session data, then this data will seamlessly re-create itself over time; it's just a matter of doing dual writes from the application to both Aerospike and Redis until Aerospike has all the data in Redis.

In other scenarios, data migration is more significant. If the data currently in Redis needs to be preserved, it can be migrated over to Aerospike. As noted earlier in this guide, it will need to be transformed to use the (namespace, set, id) tuple as a key. This typically isn't a big issue as data is either so simple that it's structureless, or the keys impose some kind of structure.

To help with the data migration, various tools, ranging from very simple to very advanced, can be used. Some of these allow pulling from Redis directly and pushing data after migration with a CDC pipeline, but most are just "one-shot" migrations from a backup.

A simple tool called the Redis Data Migrator has been created to help with this process. The aim is to take a Redis backup and import the records into Aerospike, altering the data model to suit Aerospike's style in the process. As with all tools, it is likely to get enhanced and change over time, so please consult the tool itself for the latest documentation.

Populate data in Redis

Create a Redis database, which can be as simple as running `redis-server` at the command line if you're just running a single instance, or more complex if you're running Redis Cluster. Then run `redis-cli` to be ableto enter commands.

For this simple example, there will be two types of objects: customers and accounts. Customers will be implemented as a series of keys, since we will want a set of the account ids for each customer, and Redis hashes do not support lists as values. The accounts we will implement as hashes. This gives us a good use case to see two different sorts of object mappings in Redis, as discussed earlier.

First, make sure the database is empty:

```
127.0.0.1:6379> dbsize
(integer) 0
```

Go ahead and create four simple customers using individual keys:

```
mset customer:1:firstName Tim customer:1:lastName Smith customer:1:heightInCm 185.5 customer:1:age 28
mset customer:2:firstName Bob customer:2:lastName Jones customer:2:heightInCm 167 customer:2:age 53
mset customer:3:firstName Sue customer:3:lastName Williams customer:3:heightInCm 154 customer:3:age 33
mset customer:4:firstName Theresa customer:4:lastName Ng customer:4:heightInCm 192 customer:4:age 25
```

So each of the four people has `firstName`, `lastName`, `heightInCm`, and `age` attributes, with the keys set up to be in the format `objectType:id:attribute`. (Here, `objectType` is always customer.)

Create six accounts, but using hashes this time:

```
hset account:1 name “Savings 1” type Savings balance 300
hset account:2 name “Checking 1” type Checking balance 1000
hset account:3 name “Checking 2” type Checking balance 5000
hset account:4 name “Checking 3” type Checking balance 240
hset account:5 name “Savings 2” type Savings balance 1200
hset account:6 name “Savings 3” type Savings balance 10
```

Let's associate accounts with customers. There are six accounts and four customers, so let's do:

Account Id	Name	Owners(Ids)
1	Savings 1	1, 4
2	Checking 1	2, 3
3	Checking 2	2
4	Checking 3	4
5	Savings 2	1, 3, 4
6	Savings 2	1

In Redis, we will store these as sets associated with the customer.

```
sadd customer:1:accounts 1 5 6
sadd customer:2:accounts 1 4 5
sadd customer:3:accounts 2 5
sadd customer:4:accounts 4 5
```

That's it! We now have a couple of different formats of objects.

Launching Aerospike

Launching Aerospike is pretty easy. There's a free version (called Community Edition), a free one-node Docker container of the Enterprise Edition, a free Enterprise Edition trial, and so on. You can read information about getting started with Docker here.

Assuming you have Aeropike installed and started, you'll need a namespace to store the data. By default, all Aerospike installations come with a namespace called "test," so we will use this. If your installation has a different namespace, feel free to use that.

Inside the namespace, store the information inside sets (very similar to tables in RDBS systems). These will automatically be created when they're used; there is no explicit creation process. Let's call them "Customer" and "Account," respectively.

Creating the mapping file

Before we can import the data into Aerospike, we need to export it from Redis. This is easy:

```
% redis-cli save
```

This will create a file called dump.rdb in the current directory.

The next step is to create a mapping file to describe how to map the data from Redis to Aerospike. This will be in a YAML file, and ours will look like:

```
mappings:
- key: account:(\d+)
  namespace: test
  set: Account
  id: $1

- key: customer:(\d+):(\w+)
  namespace: test
  set: Customer
  id: $1
  path: $.$2
```

Consider the `account` mapping first. The key is the key as it appears in Redis. These were set up similar to `hset account:1 name "Savings 1" type Savings balance 300`. So the key is `account:1`, and it will store the key/value pairs in a hash. In the mapping file, we have `account:(\d+)`, a regular expression which will match the "account" part of the key, then capture the decimal part of the key in a regular expression group. In this example, the id will be 1.

So, this record will be stored in the "test" namespace (hard-coded in the mapping file), in the "Account" set. The id will be whatever was captured in the first group ($1), in this case "1".

Note that if we wanted to have the key as an integer instead of a string, which seems reasonable in this case, we could augment the definition with `type: INTEGER`, so:

```
- key: account:(\d+)
  namespace: test
  set: account
  id: $1
  type: INTEGER
```

The data in the hash associated with this key will be automatically imported into the database, with the hash key forming the bin name and the hash value forming the bin value.

The customer is a bit more complex. Remember that these were set similarly to:

```
mset customer:3:firstName Sue customer:3:lastName Williams customer:3:heightInCm 154 customer:3:age 33
```

With the mapping definition being:

```
- key: customer:(\d+):(\w+)
  namespace: test
  set: Customer
  id: $1
  path: $.$2
```

So, each key contains the object type (`customer`), the id of the customer in question (3), then the attribute of the customer (`firstName`, etc). There will be multiple entries like this for each customer.

The key in the mapping file is specified as `key: customer:(\d+):(\w+)` Again, this is a regular expression, matching the word customer in the key first, then `(\d+)` will capture the number coming after the first colon in group 1 (3 in this case), and finally `(\w+)` will capture any word characters coming after the second colon (`firstName`, etc) into the second group.

This will store the customer in the hard-coded "test" namespace in the set "Customer" and with the id of the first captured group, 3 in this case. However, in this case, the bin name cannot be implied from the data contents, like the hash values could. Instead, the last part of the key forms the bin name. This was captured as the second group, so we just need to specify `path: $.$2` to use this as the bin. Note that paths use JsonPath style syntax – "`$.`" denotes the root of the record, so `$.myBin` would be a bin called "`myBin`" directly under the root. All mapping must be able to determine the bin. In this case, the name of the bin comes from the second captured group, so "`$.$2`"

Importing the data

Finally, all is ready to import the data. If you have downloaded the Redis Data Migrator from the link above, you can build it from the root directory using:

```
% mvn clean package
```

To execute it, use the following command:

```
java -jar target/redis-data-migrator-0.9-full.jar -m mapping.yaml \
    -i dump.rdb -h localhost:3000
```

This provides the program with the mapping file, the Redis saved file, and the host to connect to Aerospike. This should churn for a moment and then give a summary:

```
Execution completed in 1,007ms. 26 records imported successfully, 0 records failed.
```

Looks promising! Let's use AQL to check the records.

Looks like it worked! Note that the row count (26 rows) from the execution of the tool resulted in only 10 rows in Aerospike since

```
aql> select * from test.Customer
+-----+------+-------------------------+------------+------------+-----------+
| PK  | age  | accounts                | lastName   | heightInCm | firstName |
+-----+------+-------------------------+------------+------------+-----------+
| "2" | "53" | LIST('["1", "4", "5"]') | "Jones"    | "167"      | "Bob"     |
| "3" | "33" | LIST('["2", "5"]')      | "Williams" | "154"      | "Sue"     |
| "1" | "28" | LIST('["1", "5", "6"]') | "Smith"    | "185.5"    | "Tim"     |
| "4" | "25" | LIST('["4", "5"]')      | "Ng"       | "192"      | "Theresa" |
+-----+------+-------------------------+------------+------------+-----------+
4 rows in set (0.016 secs)

OK

aql> select * from test.Account
+-----+--------------+------------+---------+
| PK  | name         | type       | balance |
+-----+--------------+------------+---------+
| "2" | "Checking 1" | "Checking" | "1000"  |
| "1" | "Savings 1"  | "Savings"  | "300"   |
| "4" | "Checking 3" | "Checking" | "240"   |
| "6" | "Savings 3"  | "Savings"  | "10"    |
| "3" | "Checking 2" | "Checking" | "5000"  |
| "5" | "Savings 2"  | "Savings"  | "1200"  |
+-----+--------------+------------+---------+
6 rows in set (0.015 secs)

OK
```

the discrete keys forming the customer business object became a single row in Aerospike.

Enhancing the mapping file

The mapping file used in the import is good, but not great. Redis stores everything as strings, and Aerospike has no context to know what the correct type for data is. For example, the `age` field in the `Customer` set is stored as a string when it's likely to

need to be an integer. Similarly, `heightInCm` should be a floating point type. This is easy to fix. Change the mapping file for the customer to be:

```
- key: customer:(\d+):(.+)
  namespace: test
  set: Customer
  id: $1
  type: INTEGER
  path: $.$2
  translate:
  - path: "$.age"
    type: INTEGER
```

The `translate` section defines the type of fields, and the item in this section tells the importer, "When you write the bin `age`, make it an `INTEGER`." The path is fairly flexible, as we will see soon.

Notice that there is now also a "`type: INTEGER`" under the main section. This allows you to define the type of the key. In this case, we probably want the keys on both sets to be integers, so it's defined here. Re-importing and selecting the customer set now shows:

```
aql> select * from test.Customer
+----+-----------+------------+------------+-------------------------+-----+
| PK | firstName | heightInCm | lastName   | accounts                | age |
+----+-----------+------------+------------+-------------------------+-----+
| 3  | "Sue"     | "154"      | "Williams" | LIST('["2", "5"]')      | 33  |
| 2  | "Bob"     | "167"      | "Jones"    | LIST('["1", "4", "5"]') | 53  |
| 4  | "Theresa" | "192"      | "Ng"       | LIST('["4", "5"]')      | 25  |
| 1  | "Tim"     | "185.5"    | "Smith"    | LIST('["1", "5", "6"]') | 28  |
+----+-----------+------------+------------+-------------------------+-----+
4 rows in set (0.012 secs)
```

Note that you might see duplicate records in your database. We previously had records with string keys ("1", "2", etc.), and now they have numeric keys (1, 2, etc.), so these are different keys and hence duplicate records. This can be solved by truncating the set before doing the import.

There are a couple of other changes we make to the customer set. The height in centimetres should probably be a `double` type, and the name is a little long. Instead of storing it in `heightInCm`, let's store it in the `height` bin. Go and add the following

under the `translate` section:

```
-- path: “**.heightInCm”
    type: DOUBLE
    name: height
```

Note the use of “**” at the start of the path? This means “match 0 or more path parts”. So, anything called `heightInCm` would have this rule applied to it. If there were friends of the customer stored in a list of maps in a `friends` bin, then this translation rule would also affect their heights! If you use these wildcards, ensure you put the translation string in quotes, or you will confuse the YAML parser.

Adding this rule in and reimporting gives:

```
aql> select * from test.Customer
+----+------------+-------------------------+-----+--------+-----------+
| PK | lastName   | accounts                | age | height | firstName |
+----+------------+-------------------------+-----+--------+-----------+
| 4  | “Ng”       | LIST(‘[“4”, “5”]’)      | 25  | 192    | “Theresa” |
| 1  | “Smith”    | LIST(‘[“1”, “5”, “6”]’) | 28  | 185.5  | “Tim”     |
| 3  | “Williams” | LIST(‘[“2”, “5”]’)      | 33  | 154    | “Sue”     |
| 2  | “Jones”    | LIST(‘[“1”, “4”, “5”]’) | 53  | 167    | “Bob”     |
+----+------------+-------------------------+-----+--------+-----------+
4 rows in set (0.015 secs)
```

Both the `age` and `height` bins are now as expected. The only other change we probably should make is the account ids. If we’re changing the account ids to integers, we need to change the ids in the list. This is another translation rule:

```
- path: “$.accounts[*]”
   type: INTEGER
```

The `path` attribute will match any items in a list in the `accounts` bin, thanks to the use of the wildcard. If the path had just referred to `“$.accounts[1]”` for example, then only the second element in the list would have been converted to an integer type (second as the lists have a zero offset).

You may notice that there are two distinct wildcards: * and **. The difference between these is:

Wildcard	Usage
*	Matches exactly one item. So `$.customer.*.name` would match `$.customer.account.name`, but not `$.customer.name` nor `$.customer.account.subaccount.name`
**	Matches zero or more items. So `$.customer.**.name` would match `$.customer.account.name, $.customer.name`, and `$.customer.account.subaccount.name`

Let's look at the full mapping file and the results after these changes:

```
mappings:
- key: account:(\d+)
  namespace: test
  set: Account
  id: $1
  type: INTEGER
  translate:
  - path: $.balance
    type: DOUBLE

- key: customer:(\d+):(.+)
  namespace: test
  set: Customer
  id: $1
  type: INTEGER
  path: $.$2
  translate:
  - path: “$.age”
    type: INTEGER
  - path: “**.heightInCm”
    type: DOUBLE
    name: height
  - path: “$.accounts[*]”
    type: INTEGER
```

Once imported with this mapping, the results should be:

```
aql> select * from test.Customer
+----+-----------+-----+-------------------+--------+------------+
| PK | firstName | age | accounts          | height | lastName   |
+----+-----------+-----+-------------------+--------+------------+
| 1  | "Tim"     | 28  | LIST('[1, 5, 6]') | 185.5  | "Smith"    |
| 3  | "Sue"     | 33  | LIST('[2, 5]')    | 154    | "Williams" |
| 4  | "Theresa" | 25  | LIST('[4, 5]')    | 192    | "Ng"       |
| 2  | "Bob"     | 53  | LIST('[1, 4, 5]') | 167    | "Jones"    |
+----+-----------+-----+-------------------+--------+------------+
4 rows in set (0.012 secs)

OK

aql> select * from test.Account
+----+---------+--------------+------------+
| PK | balance | name         | type       |
+----+---------+--------------+------------+
| 5  | 1200    | "Savings 2"  | "Savings"  |
| 2  | 1000    | "Checking 1" | "Checking" |
| 6  | 10      | "Savings 3"  | "Savings"  |
| 1  | 300     | "Savings 1"  | "Savings"  |
| 3  | 5000    | "Checking 2" | "Checking" |
| 4  | 240     | "Checking 3" | "Checking" |
+----+---------+--------------+------------+
6 rows in set (0.015 secs)

OK
```

Perfect!

Redis to Aerospike: Mission accomplished

Successfully migrating from Redis to Aerospike enables your systems to move beyond the constraints of memory-bound architectures and embrace a high-performance, flash-optimized platform built for scale, consistency, and operational predictability. This guide has walked through Redis data model equivalencies, nuances in command behavior, and the architectural considerations that underpin an effective migration.

From type-safe, multi-bin records to native support for strongly consistent multi-record transactions, Aerospike offers a powerful and expressive data model that simplifies complex application logic. Its sharding, indexing, and I/O paths are optimized to deliver predictable low-latency access patterns, even across petabyte-scale datasets.

While the included Redis Data Migrator provides a solid foundation for data transformation and ingestion, production-grade migrations should include validation workflows, end-to-end consistency checks, and iterative tuning of record sizes, bin types, TTL policies, and client-side concurrency settings.

To go further, we recommend reviewing:

- Database Configuration
- Monitoring
- Expressions
- Aerospike Support Portal

With a solid understanding of Aerospike's data model and operational behavior, you're well-positioned to take full advantage of its performance characteristics and operational simplicity in production.

2440 W. El Camino Real, Suite 100, Mountain View, CA 94040 | (408) 462-2376 | aerospike.com