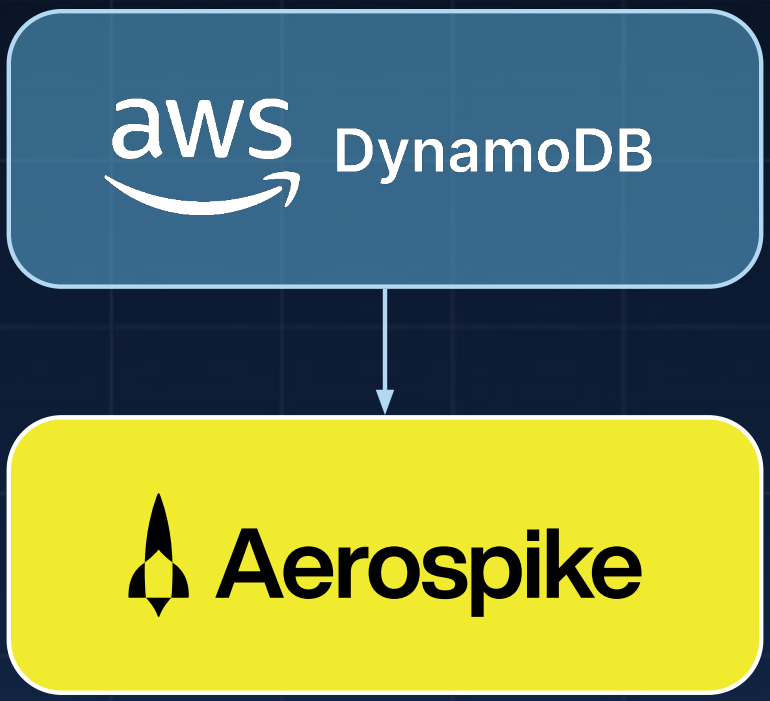
WHITE PAPER

DynamoDB to Aerospike migration guide



Contents

Introduction

DynamoDB and Aerospike are both high-performance databases that are designed to handle large amounts of throughput in clustered environments. DynamoDB is a database service offered and fully managed by Amazon Web Services (AWS). It is a NoSQL cloud database platform supporting document and key-value data storage models. DynamoDB is designed to handle large-scale applications, providing seamless integration with other AWS services. Aerospike is particularly known for its ability to handle high-throughput workloads with sub-1ms latency and petabytes of data volume, making it suitable for real-time applications.

Aerospike is designed from the ground up to store data on high-performance SSDs (such as NVMe), while keeping indexes in memory, enabling low-latency access at scale. DynamoDB is commonly used for its versatile data types (Scalar: Number, String, Binary, Boolean, and Null; Multi-valued: String Set, Number Set, and Binary Set; Document: List and Map), 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 DynamoDB.

This guide will walk you through the key concepts of DynamoDB and compare them to Aerospike. It also illustrates which patterns and approaches to smoothly transition your data model during migration, ensuring minimal disruption to your application.

Why migrate from DynamoDB to Aerospike?

Before diving into the migration process, it's important to understand why you might want to migrate from DynamoDB 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. DynamoDB leverages auto scaling, which can be a common problem for AWS customers, as the default reaction time to scaling up is slow (10–15 mins). Also, DynamoDB can not scale sufficiently to maintain a 70% utilization level.
- **Cost:** The pricing for Amazon DynamoDB is calculated based on the data's reads, writes, and storage consumption, where any additional feature is charged separately. Aerospike, by contrast, does not charge by transaction but by data volume instead.
- **On-demand capacity mode:** This pricing option is better suited for applications with less predictable database traffic, as you are charged based on the application traffic. This pricing mode follows the pay-as-you-go model, allowing the application workload to scale up and down automatically as necessary.
- **Provisioned capacity mode:** On the other hand, the provisioned capacity mode is used if the application traffic is consistent and predictable. Here, the user has to specify the number of data reads and writes per second required by the application.
- **Consistency:** Aerospike supports availability mode (favoring availability when cluster splits happen), strong consistency mode (which guarantees that no acknowledged writes are lost, even in the face of cluster splits), and can execute distributed ACID transactions. DynamoDB 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 the latter (Hybrid Memory Architecture) being the predominant use. DynamoDB is traditionally an SSD-only store, though it does offer additional options with multi-availability zone support via global tables (which can have performance implications), and DynamoDB Accelerator (DAX), which is intended for increased performance.
- **Throughput:** Each shard in DynamoDB is single OS-level threaded to avoid locks. Whilst 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 DynamoDB commands can take excessive time at scale to execute. For example, DynamoDB lists above a certain size get stored in linked lists, and traversing these consumes extra CPU cycles. Other requests in the same shard then become blocked, 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 DynamoDB does not, including but not limited to:
  - Distributed ACID transactions across multiple records
  - Multiple columns in the record associated with a single key
  - Nested lists and maps allowing for documents to be stored and manipulated in each column of a record

DynamoDB 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 DynamoDB data types onto Aerospike.

Key-value differences

Both DynamoDB and Aerospike are key-value databases, but the implementations are different. DynamoDB has every key being sent across the network as 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, only String, Binary, and Number can be used as the key attribute type. 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.

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 value "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.

Data grouping

Those familiar with relational databases often rely on structured schemas to organize related data. For instance, a single MySQL instance can host multiple databases, each containing several tables such as Customer or Account, clearly defining the type of data stored within each. Relational databases also enforce schemas on the tables to ensure that all records in that table match the specified schema.

A DynamoDB database contains no inherent structure 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 DynamoDB, 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 String, Binary, or Numeric in DynamoDB, and can be up to 2048 bytes 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 databases. 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 sometimes change this 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 an 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 DynamoDB. So a key like "Customer:123456" in DynamoDB would translate fairly directly to a set of Customer and a key of "123456". Note that in Aerospike, the string "123456" is a different key from the integer 123456.

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

Sharding in DynamoDB

Sharding in DynamoDB involves distributing a table's data across multiple partitions, with each partition hosted on a separate server. This horizontal scaling approach enhances the performance, capacity, and resilience of the table by balancing workloads across infrastructure.

Performance implications of sharding

- **Increased throughput:** By spreading data across multiple partitions, DynamoDB can handle a greater number of simultaneous read and write requests, resulting in improved performance and higher throughput.
- **Improved data locality:** With data distributed across servers, read operations can be serviced more efficiently, as the load is not concentrated on a single machine. This reduces bottlenecks and supports faster response times for localized data.
- **Enhanced scalability:** Each partition operates independently, allowing DynamoDB to scale effortlessly. Reads and writes can occur in parallel across partitions, improving both speed and reliability as demand grows.
- **Potential for increased latency:** While sharding boosts scalability, it can also introduce latency. When a request is made, DynamoDB must identify the correct partition holding the desired data, which can take extra time, especially if the key distribution is uneven or the table is accessed inefficiently.

Considerations

Sharding significantly enhances the capacity and performance of DynamoDB tables, especially for large-scale applications. However, it introduces additional complexity, such as the need for careful partition key design to avoid hotspots and maintain even distribution.

Before sharding, it's essential to evaluate your access patterns and determine whether thebenefits outweigh the operational overhead. For applications with predictable and high-volume traffic, sharding is often well worth the tradeoff.

Sharding in Aerospike

Aerospike uses constant hashing to shard the data between servers, but unlike DynamoDB, 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 distributed ACID 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 DynamoDB might look like this in Aerospike:

```
WritePolicy wp = client.copyWritePolicyDefault();
wp.txn = new Txn();
client.put(wp, new Key(“app”, “user-profile”, 1234),

          new Bin(“useranme”, “king foo”));

client.put(wp, new Key(“app”, “user-session”, 1234),

          new Bin(“useranme”, “king foo”));

client.commit(wp.txn);
```

Understanding DynamoDB data types

DynamoDB 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

In DynamoDB, the String data type is used to store textual data, such as names, descriptions identifiers, or any sequence of Unicode characters. Strings can be up to 400KB in size, making them suitable for a wide range of use cases. As with all DynamoDB data types, strings are case-sensitive and must be UTF-8 encoded. They are commonly used as attribute values and can also serve as partition keys or sort keys. When used as keys, the choice of string values plays a crucial role in ensuring even data distribution across partitions. DynamoDB supports rich querying and indexing on string attributes, including filtering, pattern matching, and lexicographic sorting, which enables flexible access patterns in applications.

Aerospike also supports strings as well as other primitive types like integers and floats. It is recommended to store the appropriate type in a bin rather than storing it as a string, as it allows operations on the types without needing to do t ype 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, and if very large strings (or other types), there are easy ways to break this large data into multiple smaller records.

Hashes

In DynamoDB, the term “hash” typically refers to the partition key in a table’s primary key schema, which determines how data is distributed across partitions. This “hash key” is used by DynamoDB’s internal hashing algorithm to map each item to a specific partition, ensuring scalability and consistent performance. Unlike traditional hash data structures (like dictionaries or maps), DynamoDB’s use of hashes is tied to data partitioning rather than in-memory key- value access. Each partition key must be unique if there is no sort key, or it can be combined with a sort key to allow multiple items with the same hash key but different sort keys. Effective hash key design is crucial; poor distribution can lead to hot partitions, where too much load is concentrated on a single node, impacting throughput and latency.

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 DynamoDB 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

In DynamoDB, the List data type allows you to store an ordered collection of values, similar to arrays in programming languages. Lists can contain multiple elements, and each element can be of any DynamoDB-supported type, including nested lists or maps. This makes them highly flexible for modeling complex or hierarchical data structures within a single item. Lists preserve the order of insertion, which can be important for scenarios such as activity feeds, preference rankings, or configuration sequences. However, since DynamoDB treats the entire list as a single attribute, updating specific elements within a list may require read-modify-write logic, especially if conditional updates or partial changes are needed. Lists are ideal when your data model benefits from maintaining an ordered group of related values under a single attribute.

Aerospike lists, in contrast, 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

In DynamoDB, the Set data type allows you to store multiple unique values in a single attribute. There are three types of sets: String Set (SS), Number Set (NS), and Binary Set (BS), each storing values of their respective types. Sets are useful when you want to represent collections of unique items, such as tags, user IDs, or access permissions. Unlike lists, sets do not preserve order and do not allow duplicate values. DynamoDB automatically ensures uniqueness within a set, which simplifies application logic. However, sets cannot contain mixed data types or nested structures. They’re efficient for membership testing and scenarios where you frequently add or remove elements without concern for order, but updates typically require replacing the entire set unless handled at the application level.

Aerospike does not natively support a “set” type like DynamoDB, 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

DynamoDB does not have a native Sorted Set data type like Redis. However, similar functionality can be implemented using a combination of a partition key and a sort key. In this pattern, the partition key groups related items (e.g., a leaderboard or timeline), while the sort key represents the value to sort by, such as a score or timestamp. This approach enables efficient querying of sorted data within a partition, including range queries, pagination, and ordering. To maintain sorted order across multiple partitions, application-side logic or additional indexing mechanisms are required. While this setup provides flexibility and scalability, it requires careful schema design.

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. DynamoDB does not natively support geospatial data types or geospatial queries such as distance calculations or bounding box searches. However, geospatial functionality can be implemented by combining DynamoDB with other AWS services like Amazon Location Service or by using geohashing techniques. A common approach is to encode geographic coordinates (latitude and longitude) into a geohash string, which is then stored as a partition or sort key in DynamoDB. This allows for proximity-based lookups by searching for geohash prefixes within a certain range. While this method provides basic spatial indexing and querying, it requires custom logic for encoding, decoding, and managing geographic boundaries. For applications requiring robust geospatial features, such as nearest- neighbor queries or complex polygon searches, it’s often better to pair DynamoDB with a dedicated geospatial engine like Amazon OpenSearch or PostGIS.

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 GeoJON polygons or circles are stored in records, an arbitrary point can be queried for which regions contain that point.

Bitmaps

DynamoDB does not offer native support for bitmaps as a dedicated data type, unlike some other databases, such as Redis or Aerospike. However, bitmap-like functionality can be emulated by storing binary data using DynamoDB's Binary (B) data type. This allows you to store a binary-encoded string or blob representing a bitmap, which can then be manipulated within your application logic. While this approach is suitable for compactly representing boolean states or flags (e.g., feature toggles or availability slots), all bit-level operations, such as setting, clearing, or querying specific bits, must be handled on the client side. This means you must read the binary value from DynamoDB, modify it locally, and then write it back, which can be less efficient and more complex than using a database with native bitmap operations.

Aerospike supports direct bitmap manipulation. 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 DynamoDB to Aerospike

When migrating the codebase from DynamoDB to Aerospike, the first decision is whether to change records to match the more flexible format that Aerospike supports. As mentioned previously, Aerospike keys are identified by a tuple consisting of the namespace, the set, and the id. In DynamoDB, 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 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 DynamoDB, 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 DynamoDB calls. Additionally, transactionality might be important to the use case, so that you want to be able to update `firstName` and `lastName` atomically together. In the context of DynamoDB, the term hashtags does not refer to a built-in feature or data type like in social media platforms or even in Redis (where hashtags are used to control key placement in clusters). However, developers sometimes use the concept of "hashtags" informally when designing composite keys to help group related items or control data locality. For instance, a key like `{customer:1234}` uses a structured string format that mimics hashtags to encode hierarchical relationships into a single partition or sort key. This pattern allows for efficient querying and sorting within a partition by using begins_with, between, or range queries. While not a native feature, this approach—often called key design or key modeling—is crucial in DynamoDB for optimizing access patterns, simulating relational joins, and improving query performance. 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 DynamoDB, so to store the address, it either needs to be stored as a JSON string inside the hash (option two above), or to have two separate keys, one for the customer and one for the address (option one above).

How you've used DynamoDB 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 DynamoDB and Aerospike require a connection to the database before executing requests. Connecting to DynamoDB is straightforward, especially when using the AWS SDKs, which provide native support across popular programming languages like Java, Python, Node.js, and Go. Since DynamoDB is a fully managed service hosted by AWS, there's no need to manage servers or set up traditional database connections. Instead, applications connect via HTTPS using the DynamoDB API, and the SDK handles authentication, request signing, and retries. You'll typically configure the SDK with your AWS credentials (via environment variables, IAM roles, or AWS profiles) and specify the AWS region where your DynamoDB tables reside.

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 prior to 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 (9.0.4 at the time of writing) to ensure you get the latest features. For example, versions of the Java 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>
```

Connecting with DynamoDB

In DynamoDB, connecting to a database is simple, but requires code for connection management. For example, to connect via a Java client, make sure AWS credentials are configured via environment variables, AWS CLI, or IAM roles if running in AWS, you can use:

```
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
import software.amazon.awssdk.services.dynamodb.model.*;
public class DynamoDBExample {
    public static void main(String[] args) {
        // Create client
        DynamoDbClient ddb = DynamoDbClient.builder()
                .region(software.amazon.awssdk.regions.Region.US_WEST_2)
                .build();
        // Get item from table
        GetItemRequest request = GetItemRequest.builder()
                .tableName("MyTable")
                .key(Map.of("id",AttributeValue.builder().s("123").build()))
                .build();
       GetItemResponse response = ddb.getItem(request);
        System.out.println("Item: " + response.item());
    }
}
```

Connecting with Aerospike

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

In DynamoDB, this would look like:

```
GetItemRequest getRequest = GetItemRequest.builder()
        .tableName("MyTable")
        .key(key)
        .build();

GetItemResponse response = ddb.getItem(getRequest);
System.out.println("Retrieved item: " + response.item());
```

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 is 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");
```

Here's the updated section without bullet points:

List and map operations

Both DynamoDB and Aerospike support rich list and map data types, enabling developers to model complex, nested data structures. However, Aerospike provides more extensive and efficient server-side capabilities for manipulating these structures.

List operations in Aerospike

Aerospike lists are strongly typed, ordered collections that support a wide range of operations such as appending, inserting, trimming, removing by index or value, and slicing ranges, all performed atomically on the server. Lists can contain any supported type and can be nested with maps or other lists. They support constant-time access by index, range queries, filtered reads, uniqueness constraints via list policies, and complex multi-operation transactions on a single key. This makes them ideal for modeling features like user activity history, ordered preferences, or scoreboards with high performance and strong consistency.

Map operations in Aerospike

Aerospike maps function like dictionaries, with keys and values of any type. They support inserting, updating, deleting fields, querying by key, value, index, or rank, and maintaining order either by key or by value. Maps can be nested within other maps or lists and allow partial reads, conditional logic, and server-side filtering. These capabilities make maps well-suited for storing user profiles, counters, session metadata, or JSON-style objects.

Comparison with DynamoDB

In DynamoDB, lists and maps are supported, but manipulation is limited. Most operations require reading the item, modifying it in the application code, and writing it back. There is no native support for querying within lists or maps, and filtering must be done on the client. Additionally, the 400KB item size limit restricts how large these structures can grow.

In contrast, Aerospike performs all list and map operations on the server, which minimizes latency and reduces client-side complexity. The operate() API allows atomic chaining of multiple operations, and the 8MB record size limit enables much larger and more flexible data structures.

The power of `Operate`

In the above discussions on list operations, it is readily apparent that both DynamoDB and Aerospike have very different support for List operations. Why is this?

Fundamentally, Aerospike's data model is richer than DynamoDB's, 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: the count 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 `TransactWriteItems API` in DynamoDB, 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.

Unsupported operations

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

1. **Blocking operations:** DynamoDB does not support blocking operations because it is designed for high availability and scalability using non-blocking, stateless API calls.

   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 of the database. This is simple to arrange and control at the application level.

2. **Multiple record commands.** DynamoDB doesn't have a native command to move an element from one list to another, but you can simulate this behavior using an atomic transaction with two `UpdateItem` actions. In Aerospike, this operation would also involve two separate actions: one to remove the element from the source list and another to add it to the destination list, but both can be executed atomically using ACID transactions to maintain consistency.

3. **Pub/Sub commands.** DynamoDB supports Streams, a publish and subscribe mechanism similar to Kafka using AWS Lambda + SNS or SQS. Aerospike does not support this functionality inherently. However, adapters to perform change data capture (CDC) style operations with queueing systems like Kafka, JMS, Flink, Spark, etc., are available. This gives flexibility on your messaging choice and allows the database to focus on being a database rather than a queueing system.

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: Conclusion and best practices

Migrating from Amazon DynamoDB to Aerospike involves careful planning and execution to ensure data integrity, application compatibility, and optimal performance. Below are best practices to guide you through this process:

1. **Understand the data models**
   - **DynamoDB:** Utilizes a schemaless design with tables, items, and attributes. It supports key-value and document data models, with each item identified by a primary key.

- **Aerospike:** Employs a schemaless, key-value store with a set-based architecture. Data is organized into namespaces, sets, and records, with each record identified by a unique key.

Action: Analyze your DynamoDB data model to determine how tables and items will map to Aerospike's namespaces, sets, and records.

2. **Plan the schema migration**
   - **Data types:** Ensure that data types used in DynamoDB have corresponding types in Aerospike.
   - **Indexes:** Identify primary keys and secondary indexes in DynamoDB and plan their implementation in Aerospike. Aerospike supports primary indexes by default and allows the creation of secondary indexes for efficient querying.

Action: Create a detailed schema mapping document to guide the migration process.

3. **Extract and transform data**
   - **Data extraction:** Use AWS services like AWS Data Pipeline or AWS Glue to export data from DynamoDB.
   - **Data transformation:** Format the extracted data to align with Aerospike's data model. This may involve restructuring JSON documents or modifying attribute names and data types.

Action: Develop scripts or use ETL tools to automate the extraction and transformation processes.

4. **Load data into Aerospike**
   - **Data import:** Utilize Aerospike's tools such as asloader or client libraries (e.g., Java, Python) to import data into Aerospike. Ensure that the data is distributed evenly across the cluster to prevent hotspots.

Action: Perform test imports with sample datasets to validate the process before full-scale data loading.

5. **Update application code**
   - **Client integration:** Replace DynamoDB client libraries with Aerospike client libraries in your application code.
   - **Query modification:** Adapt queries and data access patterns to use Aerospike's API and query mechanisms. Aerospike supports complex queries through secondary indexes and operates with a different query syntax compared to DynamoDB.

Action: Refactor and test application code to ensure compatibility with Aerospike's APIs and query language.

6. **Ensure data consistency and integrity**
   - **Validation:** After migration, perform thorough data validation to ensure that all records have been accurately transferred and that data integrity is maintained.
   - **Consistency models:** Understand the consistency models of both databases. DynamoDB offers eventual and strong consistency options, while Aerospike provides tunable consistency settings. Configure Aerospike to meet your application's consistency requirements.

Action: Implement validation scripts to compare data between DynamoDB and Aerospike, ensuring completeness and accuracy.

7. **Optimize performance**
   - **Configuration tuning:** Adjust Aerospike configurations such as `migrate-sleep`, `migrate-threads`, and `migrate-max-num-incoming` to optimize migration performance without impacting cluster stability.

- **Resource provisioning:** Ensure that the Aerospike cluster is adequately provisioned with CPU, memory, and storage resources to handle the incoming data and workload.

Action: Monitor system performance during and after migration, making necessary adjustments to configurations and resources.

8. **Implement backup and recovery strategies**
    - **Backups:** Establish regular backup procedures for Aerospike to protect against data loss. Aerospike provides tools and mechanisms for creating backups of your data.
    - **Disaster recovery:** Develop a disaster recovery plan that includes data replication and failover strategies to ensure high availability and data durability.

Action: Regularly test backup and recovery procedures to confirm their effectiveness.

9. **Monitor and maintain**
    - **Monitoring:** Utilize Aerospike's monitoring tools and integrate them with your existing monitoring systems to track cluster health, performance metrics, and potential issues.
    - **Maintenance:** Schedule regular maintenance tasks such as defragmentation, rolling upgrades, and configuration reviews to maintain optimal performance and stability.

Action: Establish a monitoring and maintenance schedule, assigning responsibilities to relevant team members.

10. **Engage with the community and support**
    - **Community resources:** Leverage Aerospike's community forums, documentation, and knowledge base for insights and assistance.
    - **Professional support:** Consider engaging Aerospike's professional services for guidance, especially for complex migration scenarios or performance tuning.

Action: Join Aerospike's community forums and consider professional support options to enhance your migration experience.

By following these best practices, you can facilitate a smooth transition from DynamoDB to Aerospike, ensuring that your application continues to function effectively with improved performance and scalability.

Appendix: Detailed comparison: DynamoDB vs. Aerospike sharding

Sharding mechanism

Feature	DynamoDB	Aerospike
Sharding strategy	Hash-based on partition key	Hash-based on a combination of namespace, set, and key
Partition count	Dynamic – created as data grows	Fixed – 4,096 partitions per namespace
Shard assignment	Managed by AWS, not visible to users	Managed by Aerospike, fully transparent
Data placement control	Requires careful key design to avoid hot partitions	Automatic and uniform via consistent hashing

Operational behavior

Feature	DynamoDB	Aerospike
Hot key risk	High if partition key distribution is poor	Very Low – consistent hash spreads load evenly
Load balancing	Dynamic – created as data grows	Built-in via automatic partition hashing
Cluster expansion	AWS handles, but repartitioning is opaque	Aerospike rebalances partitions across nodes automatically
Consistency across shards	Not supported for multi- item ACID transactions across partitions	Supported via strong consistency mode and multi-record transactions
Concurrency scaling	Read/Write capacity can be throttled per partition	Supports true parallelism, locks only at record level

Monitoring and debugging

Feature	DynamoDB	Aerospike
Shard visibility	No insight into shard boundaries or count	Full visibility via Aerospike tools like AMC or CLI
Hotspot detection	Requires CloudWatch metrics and key access tracking	Detailed introspection of partition load and latency is available
Resharding transparency	Not visible; can introduce temporary throttling	Fully automated and transparent partition reassignment

Practical challenges

Feature	DynamoDB	Aerospike
Multi-record operations	Requires complicated logic using batch operations or transactions	Native support for atomic multi-record ops in SC mode
Time-series data	Needs care in key design to avoid bursts	Can naturally spread across partitions
High-volume writes	Prone to hot partitions unless keys are designed properly	Effortlessly scales via partition distribution

Summary

Feature	DynamoDB	Aerospike
Developer control	High (manual partition key logic)	Low (automated and reliable)
Sharding transparency	Low	High
Scalability	Horizontal, cloud-native	Horizontal, on-prem/cloud/hybrid
Rebalancing	Opaque	Transparent and automatic
ACID transactions	Single-item only (multi-item with limitations)	Full multi-record support (in SC mode)

About Aerospike

Aerospike is the real-time database for mission-critical use cases and workloads, including machine learning, generative, and agentic AI. Aerospike powers millions of transactions per second with millisecond latency, at a fraction of the cost of other databases.

Global leaders, including Adobe, Airtel, Barclays, Criteo, DBS Bank, Experian, Grab, HDFC Bank, PayPal, Sony Interactive Entertainment, The Trade Desk, and Wayfair, rely on Aerospike for customer 360, fraud detection, real-time bidding, profile stores, recommendation engines, and other use cases.

Try Aerospike for free: aerospike.com/try-now

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