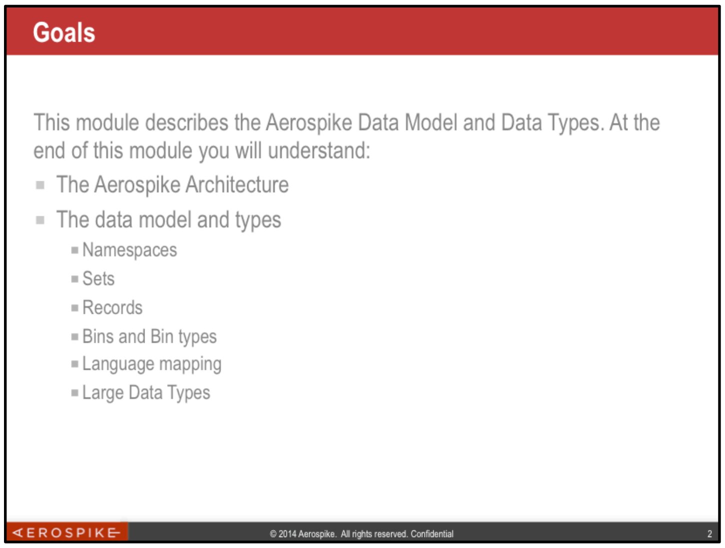
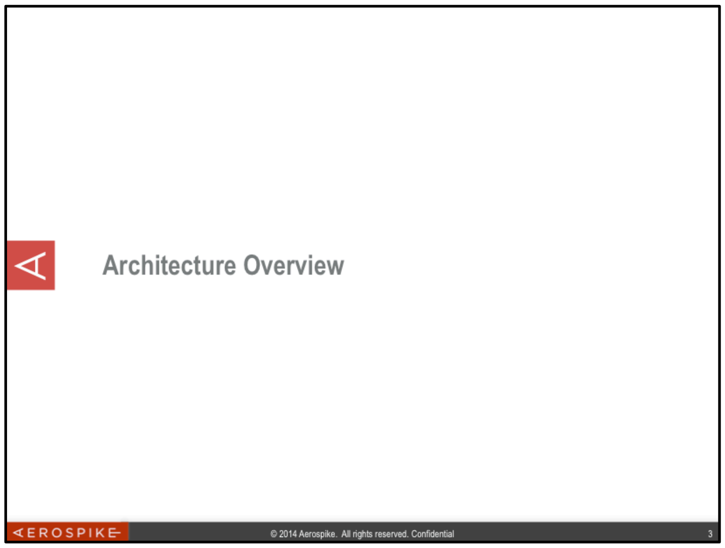
AEROSPIKE

Architecture, Data Model and

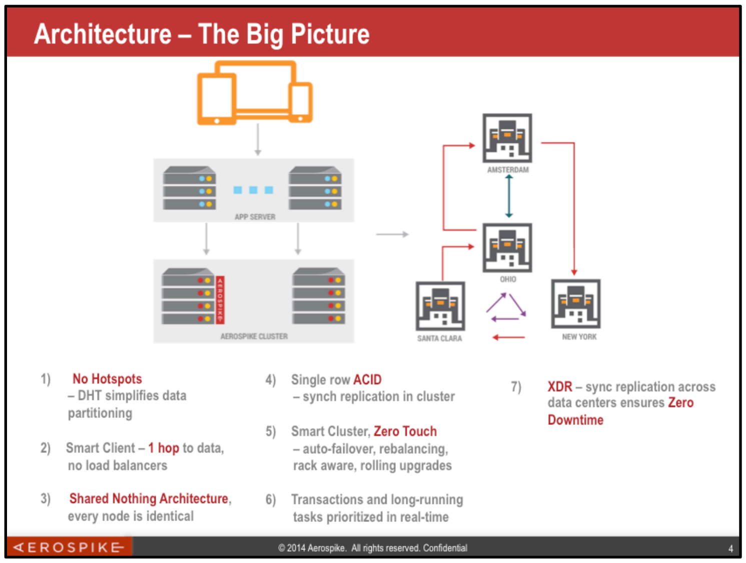

Goals

This module describes the Aerospike Data Model and Data Types. At the end of this module you will understand:

- The Aerospike Architecture
- The data model and types
  - Namespaces
  - Sets
  - Records
  - Bins and Bin types
  - Language mapping
  - Large Data Types



Architecture Overview


APP SERVER
AEROSPIKE CLUSTER
AMSTERDAM
OHIO
SANTA CLARA
NEW YORK

Architecture – The Big Picture

1) **No Hotspots** – DHT simplifies data partitioning
2) Smart Client – **1 hop** to data, no load balancers
3) **Shared Nothing Architecture**, every node is identical
4) Single row **ACID** – synch replication in cluster
5) Smart Cluster, **Zero Touch** – auto-failover, rebalancing, rack aware, rolling upgrades
6) Transactions and long-running tasks prioritized in real-time
7) **XDR** – sync replication across data centers ensures **Zero Downtime**

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 4

The Big Picture

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

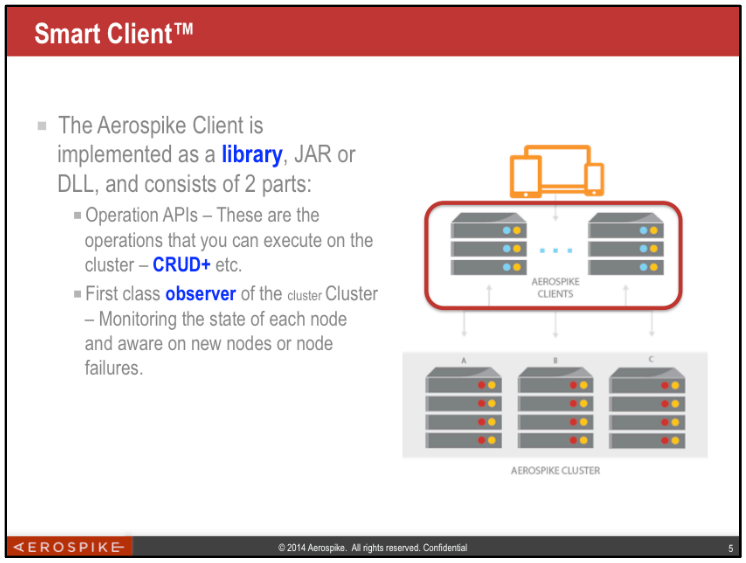
- To create a flexible, scalable platform that would meet the needs of today's web-scale applications
- To provide the robustness and reliability (ie, ACID) expected from traditional databases.
- To provide operational efficiency (minimal manual involvement)

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

1. The cluster-aware Client Layer includes open source client libraries that implement Aerospike APIs, track nodes and know where data reside in the cluster.
2. The self-managing Clustering and Data Distribution Layer oversees cluster communications and automates fail-over, replication, cross data center synchronization and intelligent re-balancing and data migration.
3. The flash-optimized Data Storage Layer reliably stores data in RAM and Flash.

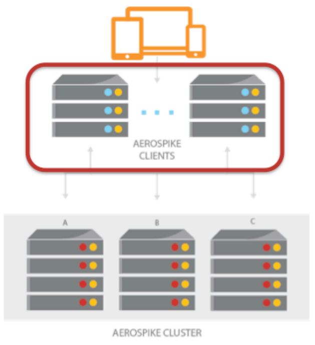
Aerospike scales up and out on commodity servers.
1) Each multi-threaded, multi-core, multi-cpu server is fast
2) Each server scales up to manage up to 16TB of data, with parallel access to up to 20 SSDs
3) Aerospike scales out linearly with each identical server added, to parallel process 100TB+ across the cluster
(AppNexus was in production with a 50 node cluster that has now scaled up by increasing SSD capacity per node and reducing node count to 12, in preparation for scaling back up on node counts and SSD capacity)
- see http://www.aerospike.com/wp-content/uploads/2013/12/Aerospike-AppNexus-SSD-Case-Study_Final.pdf

SYNC replication within cluster for immediate consistency, #replicas from 1 to #nodes in cluster, typical deployment has 2 or 3 copies of data; Can support ASYNC within cluster. ASYNC replication across clusters using Cross Data Center Replication (XDR). Writes automatically replicated across one or more locations. Fine grained control - individual sets or namespaces can be replicated across clusters in master/master/slave complex ring and star topologies.



Smart Client™

- The Aerospike Client is implemented as a **library**, JAR or DLL, and consists of 2 parts:
  - Operation APIs – These are the operations that you can execute on the cluster – **CRUD+** etc.
  - First class **observer** of the cluster Cluster – Monitoring the state of each node and aware on new nodes or node failures.

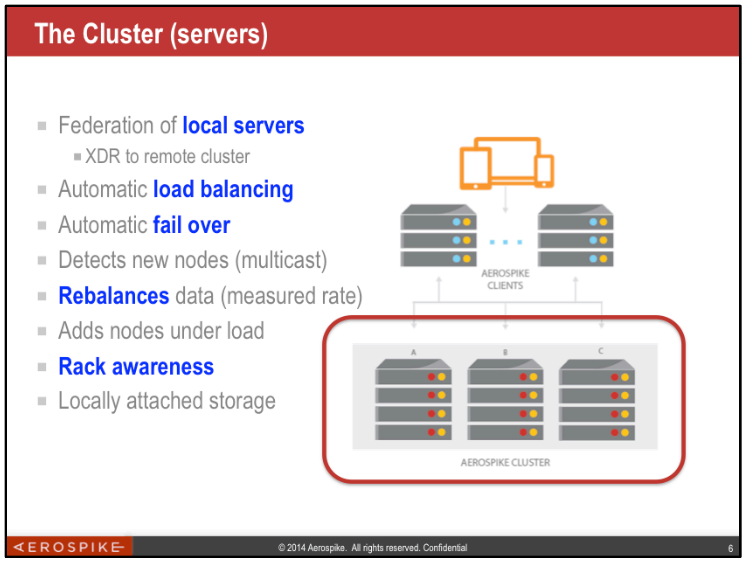


Client Layer

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

- Implements the Aerospike API, the client-server protocol and talks directly to the cluster.
- Tracks nodes and knows where data is stored, instantly learning of changes to cluster configuration or when nodes go up or down.
- Implements its own TCP/IP connection pool for efficiency. Also detects transaction failures that have not risen to the level of node failures in the cluster and re-routes those transactions to nodes with copies of the data.
- Transparently sends requests directly to the node with the data and re-tries or re-routes requests as needed. One example is during cluster re-configurations.

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



The Cluster (servers)

- Federation of **local servers**
  - XDR to remote cluster
- Automatic **load balancing**
- Automatic **fail over**
- Detects new nodes (multicast)
- **Rebalances** data (measured rate)
- Adds nodes under load
- **Rack awareness**
- Locally attached storage

AEROSPIKE CLIENTS
A
B
C
AEROSPIKE CLUSTER

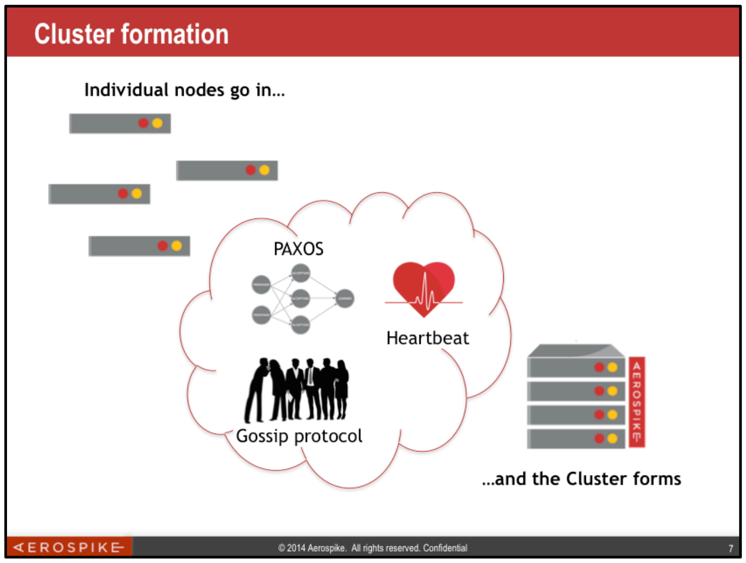
AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 6

Cluster Layer

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

- The Cluster Management Module tracks nodes in the cluster. The key algorithm is a Paxos-like consensus voting process which determines which nodes are considered part of the cluster. Aerospike implement special heartbeat (active and passive) to monitor inter-node connectivity.
- When a node is added or removed and cluster membership is ascertained, each node uses distributed hash algorithm to divide the primary index space into data 'slices' and assign owners. Data Migration Module then intelligently balances the distribution of data across nodes in the cluster, and ensures that each piece of data is duplicated across nodes and across data centers, as specified by the system's configured replication factor.
- Division is purely algorithmic, the system scales without a master and eliminates the need for additional configuration that is required in a sharded environment.
- The Transaction Processing Module reads and writes data as requested and provides many of the consistency and isolation guarantees. This module is responsible for
  1. Sync/Async Replication : For writes with immediate consistency, it propagates changes to all replicas before committing the data and returning the result to the client.
  2. Proxy : In rare cases during cluster re-configurations when the Client Layer may be briefly out of date, it transparently proxies the request to another node.
  3. Duplicate Resolution : when a cluster is recovering from being partitioned, it resolves any conflicts that may have occurred between different copies of data. Resolution can be configured to be
     - Automatic, in which case the data with the latest timestamp is canonical
     - User driven, in which case both copies of the data can be returned to the application for resolution at that higher level.

Once you have the first cluster up, you can optionally install additional clusters in other data centers and setup cross data-center replication – this ensures that if your data center goes down, the remote cluster can take over the workload with minimal or no interruption to users.



Clustering

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

The distributed, shared-nothing architecture means that nodes can self-manage and coordinate to ensure that there are no outages to the user, while at the same time being easy to expand your cluster as traffic increases.

Heartbeat

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

- Multicast in this case multicast IP:PORT is used to broadcast heartbeat message
- Mesh in this case address of one Aerospike server is used to join cluster

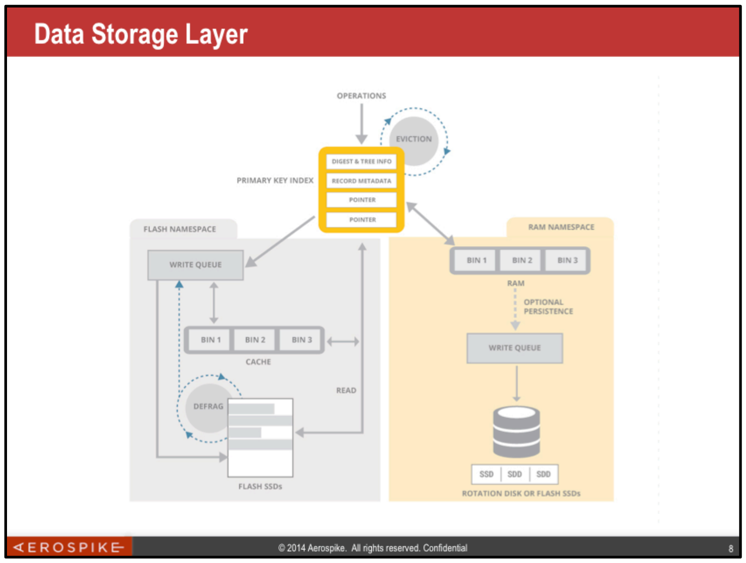
High speed distributed consensus

Aerospike's 24/7 reliability guarantee starts with its ability to detect cluster failure and quickly recovering from that situation to reform the cluster.

Once the cluster change is discovered, all surviving nodes use a Paxos-like consensus voting process to determine which nodes form the cluster and Aerospike Smart Partitions™ algorithm to automatically re-allocate partitions and re-balance. The hashing algorithm is deterministic – that is, it always maps a given record to the same partition. Data records stay in the same partition for their entire life although partitions may move from one server to another.

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

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



Data Storage Layer

© 2014 Aerospike. All rights reserved. Confidential 8

Data Storage Layer

Aerospike is a key-value store with schema-less data model. Data is organized into policy containers called namespaces, semantically similar to databases in an RDBMS system. Within a namespace, data is subdivided into sets (similar to tables) and records (similar to rows). Each record has an indexed key that is unique in the set, and one or more named bins (similar to columns) that hold values associated with the record.

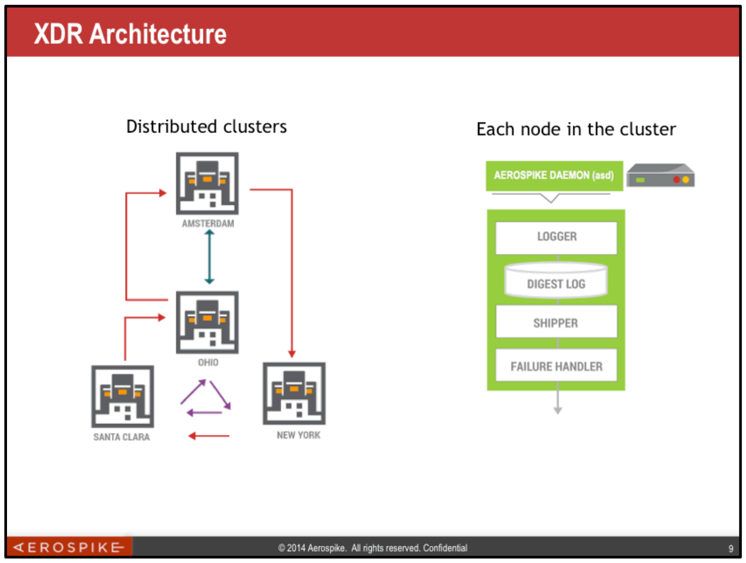
Sets and bins do not need to be defined up front, but can be added during run-time for maximum flexibility.

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

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

The Data Layer was particularly designed for speed and a dramatic reduction in hardware costs. It can operate all in-memory, eliminating the need for a caching layer or it can take advantage of unique optimizations for flash storage. In either case, data is never lost.

- 100 Million keys take up only 6.4GB. Although keys have no size limitations, each key is efficiently stored in just 64 bytes.
- Native, multi-threaded, multi-core Flash I/O and an Aerospike log structured file system take advantage of low level SSD read and write patterns. In addition, writes to disk are performed in large blocks to minimize latency. This mechanism bypasses the standard file system, historically tuned to rotational disks.
- Also built-in are a Smart Defragmenter and Intelligent Evictor . These processes work together to ensure that there is space in RAM and that data is never lost and safely written to disk.
    - The Defragmenter tracks the number of active records in each block and reclaims blocks that fall below a minimum level of use.
    - The Evictor removes expired records and reclaims memory if the system gets beyond a set high water mark. Expiration times are configured per namespace, the age of a record is calculated from the last time it was modified, the application can override the default lifetime at any time



XDR Architecture

XDR - Cross Datacenter Replication - is an Aerospike feature which allows one cluster to synchronize with another cluster over a longer delay link. This replication is asynchronous, but delay times are often under a second. Each write is logged, and that log is used to replicate to remote clusters. Each master for a partition is responsible for replicating that partition's writes, and the current write status is replicated to the local cluster's partition replica, so when that replica is promoted, it can take over where the previous master left off.

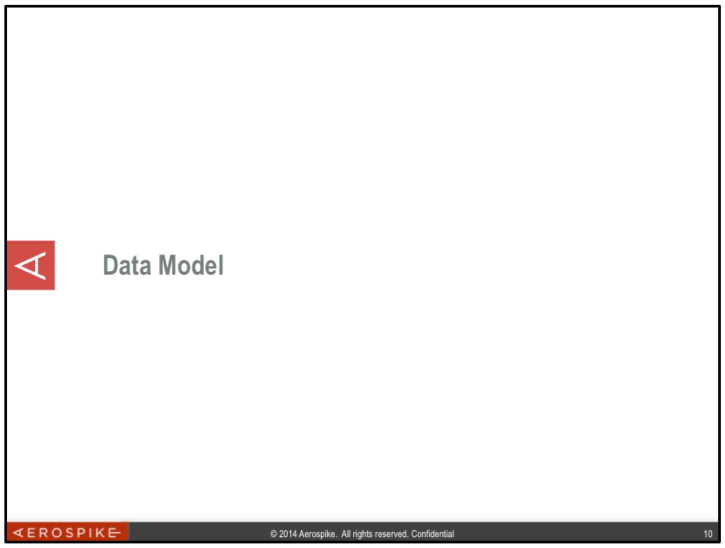
An XDR process runs on every node of the cluster along side the Aerospike process. The main task of the XDR process is to monitor database updates on the "local" cluster and send copies of all write requests to one or more "remote" destination clusters. XDR determines the remote cluster details from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

Each node runs:

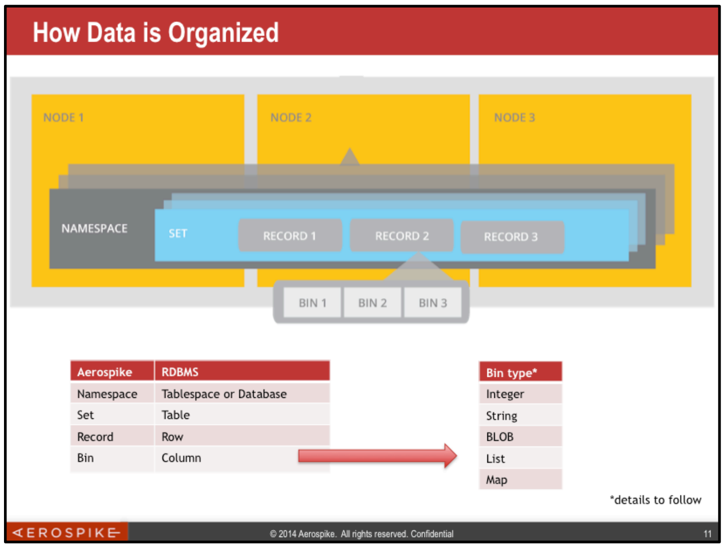
- Aerospike daemon process (asd)
- XDR daemon process (xdr), which is composed of the following:
    - Logger module
    - Shipper module
    - Failure Handler module

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

The details of this are that writes and updates by the main database system are communicated with its XDR process over a named pipe. This communication is received by the Logger module which stores this information in the digest log. To keep the digest log file small, only minimal information (the key digest) is stored in the digest log – just enough to ship the actual records at a later time. The Digest Log is a ring buffer file. i.e the most recent data will potentially overwrite the oldest data (if the buffer is full). This allows us to keep a check on this file size. But this also introduces a capacity planning issue. You need to configure a big enough digest log file to avoid the loss of any pending data to be shipped. You should plan



Data Model



How Data is Organized

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

Bin type*
Integer
String
BLOB
List
Map

*details to follow

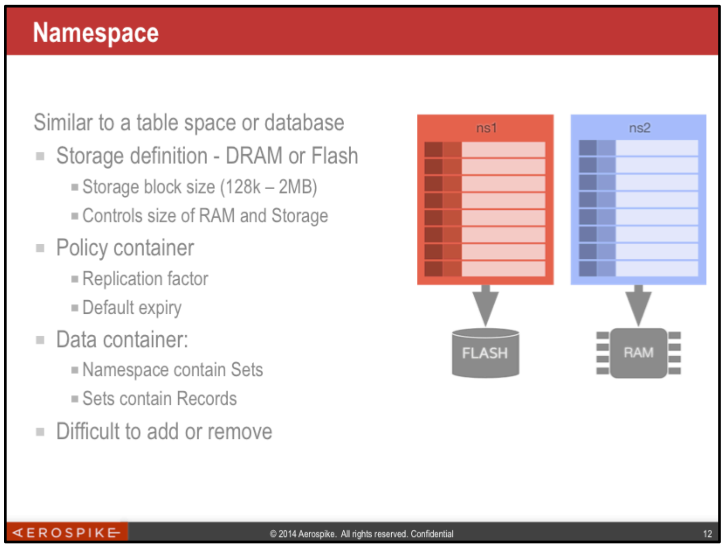
ᗉEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 11

Data Hierarchy

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

- A Namespace (like a tablespace or database), a name space can contain zero or mode Sets
- A Set is like a Table and contains a collection of records. A set has no schema.
- A Record, sometimes called an object, is a row, it contains Bins.
- Bins are like a column or field. A bin has a name and a type.
    - Bin types (more details to follow):
        - Integer
        - String
        - BLOB
        - List
        - Map

Details of each component are on the following pages.



Namespace

Similar to a table space or database

- Storage definition - DRAM or Flash
  - Storage block size (128k – 2MB)
  - Controls size of RAM and Storage
- Policy container
  - Replication factor
  - Default expiry
- Data container:
  - Namespace contain Sets
  - Sets contain Records
- Difficult to add or remove

ns1
ns2
FLASH
RAM

Namespaces

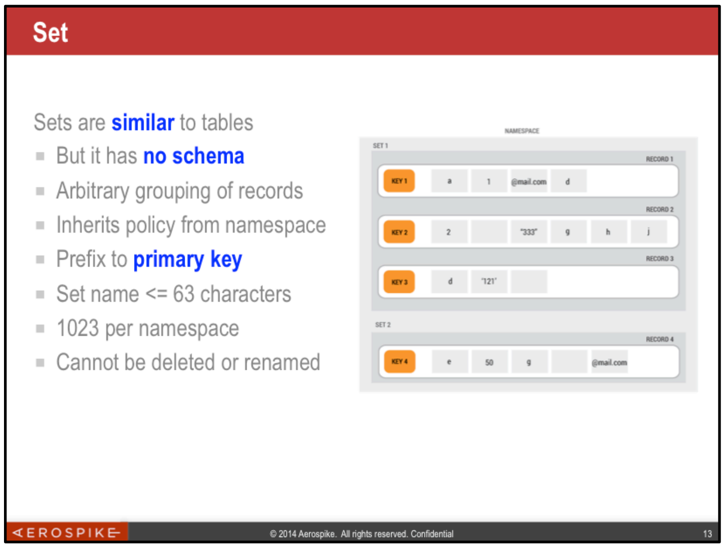
Namespaces are the top level containers for data. A namespace can actually be a part of a database or it can be a group of databases as you would think of them in a standard RDBMS – the reason you collect data into a namespace relates to how the data is going to be stored and managed.

A namespace contains records, indexes and policies. A policy dictates the behavior of the namespace, including:

- How data is stored: DRAM or disk
- How many replicas should exists for a record.
- When records should expire.
- Consistency

Namespaces can only be defined with a cluster wide restart. This makes them difficult to add or remove.

A database may specify multiple namespaces, each with differing policies, depending on the applications' needs. A namespace is considered a physical container, because it binds the data to a storage device, whether a segment of RAM, a disk or file.



Set

Sets are **similar** to tables

- But it has **no schema**
- Arbitrary grouping of records
- Inherits policy from namespace
- Prefix to **primary key**
- Set name <= 63 characters
- 1023 per namespace
- Cannot be deleted or renamed

NAMESPACE
SET 1
RECORD 1
KEY 1
a
1
@mail.com
d
RECORD 2
KEY 2
2
"333"
g
h
j
RECORD 3
KEY 3
d
'121'
SET 2
RECORD 4
KEY 4
e
50
g
@mail.com

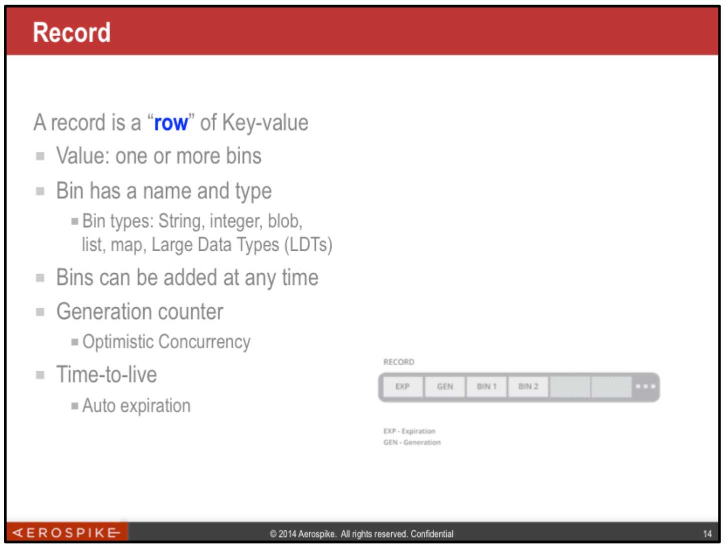
AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 13

Set

Within a namespace, records can belong to a logical container called set. A set provides applications the ability to group records in collections. A Set is similar to a table as it is a collection of records, but it has no schema. Sets use the policy defined by the namespace to which they belong, such as storage, replication and consistency. A Set may define additional policies specific to the set, such as a secondary index definition. Sets names are like a prefix to the primary key, and cannot be deleted or renamed.

The total number of sets that can be created in a namespace is 1023. This is 1023 "named" sets and 1 unnamed set (or null set).

Note: Moving a record from one Set to another may move it from one server to another. Sets can be iterated using the Scan operation.



Record

A record is a "**row**" of Key-value

- Value: one or more bins
- Bin has a name and type
  - Bin types: String, integer, blob, list, map, Large Data Types (LDTs)
- Bins can be added at any time
- Generation counter
  - Optimistic Concurrency
- Time-to-live
  - Auto expiration

RECORD
EXP
GEN
BIN 1
BIN 2
EXP - Expiration
GEN - Generation

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 14

Records

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

A record is composed of the following:

Component	Description
Key	A record is addressable via a hash of if its key, called a Digest
metadata	The metadata provides information about the version of the record (generation) and the time-to-live (TTL).
Bins	Bins are the equivalent of columns or fields in an conventional RDBMS.

Key / Digest

In the application, each record will have a key associated with it. This key is what the application will use to read or write the record.

However, when the key is sent to the database, the key is hashed into a 160-bit digest. Within the database, the digest is used address the record for all operations.

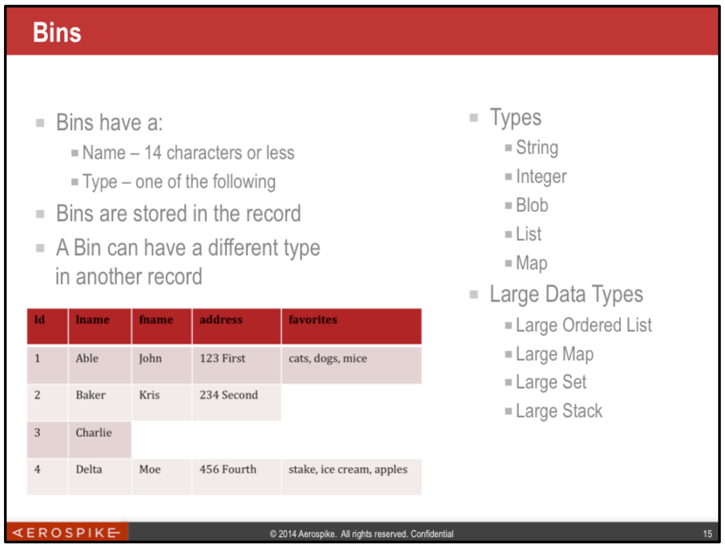
The key is used primarily in the application, while the digest is primarily used for addressing the record in the database.

The key maybe either an Integer, String, or Bytes value. For details on these values, see Data Types.

Metadata

Each record is stored with metadata about the record itself, including the following:

- generation reflects how many times the record has been modified. This number is handed back to the application on a read, and may be used to make sure that data that is being written has not been modified since the last read.
- time-to-live (TTL) specifies how long the record will exist. Aerospike will automatically expire records based on their TTL. A record's TTL is incremented each time a write-operation is executed on the object.



Bins

- Bins have a:
  - Name – 14 characters or less
  - Type – one of the following
- Bins are stored in the record
- A Bin can have a different type in another record

Id	lname	fname	address	favorites
1	Able	John	123 First	cats, dogs, mice
2	Baker	Kris	234 Second	
3	Charlie			
4	Delta	Moe	456 Fourth	stake, ice cream, apples

- Types
  - String
  - Integer
  - Blob
  - List
  - Map
- Large Data Types
  - Large Ordered List
  - Large Map
  - Large Set
  - Large Stack

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 15

Bins

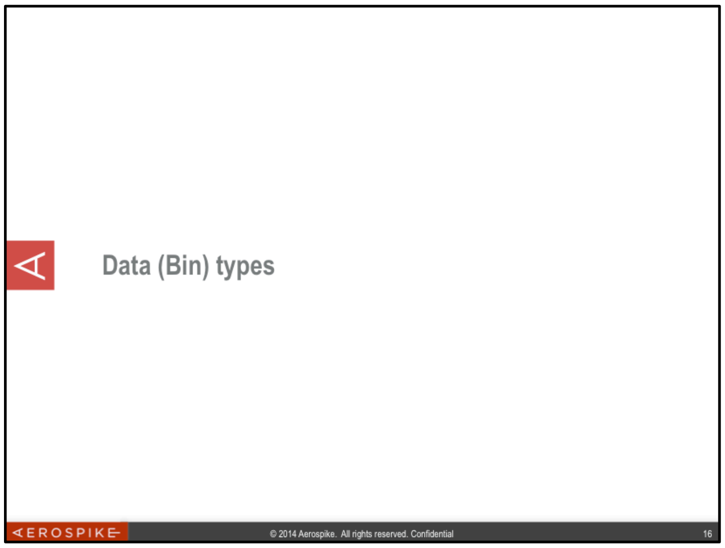
Within a record, data is stored in one or many bins. A bin consists of a name and a value. A bin does not specify the type, instead the type is defined by the value contained in the bin.

This dynamic typing provides much flexibility in the data model. For example, a record may contain a bin named "id" with a string value of "bob". The value of the bin can always change to a different string value, but to a value of a different type, such as the integer 72.

Also, records within a namespace or set maybe composed of very different collection of bins. There is no schema for the records, so it is possible for each record to have completely different set of bins. Also, bins can be added or removed at any point in the lifetime of a record.

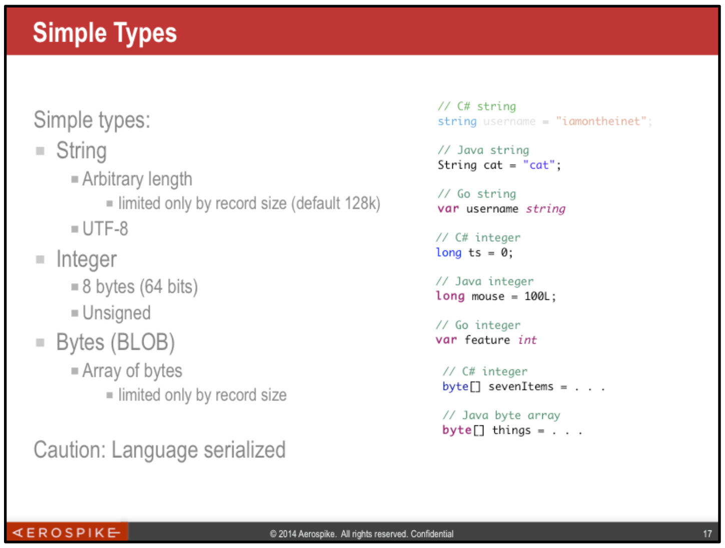
There is a limit on the number of bin names currently in use within a namespace, due to an optimized string-table implementation. That limit is 32K unique bin names per namespace. If more bin names are required, consider using a map. With a map, you can store an arbitrary set of key-value pairs, and access those values in a UDF for efficiency.

A bin name is limited to 14 characters.



Data (Bin) types

AEROSPIKE



Simple Types

Simple types:

- String
  - Arbitrary length
    - limited only by record size (default 128k)
  - UTF-8
- Integer
  - 8 bytes (64 bits)
  - Unsigned
- Bytes (BLOB)
  - Array of bytes
    - limited only by record size

Caution: Language serialized

```
// C# string
string username = "iamontheinet";

// Java string
String cat = "cat";

// Go string
var username string

// C# integer
long ts = 0;

// Java integer
long mouse = 100L;

// Go integer
var feature int

 // C# integer
 byte[] sevenItems = . . .

 // Java byte array
 byte[] things = . . .
```

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 17

String

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

Integer

Integers are stored as 8 byte quantities, which limits integer values in the current version. At the client, the integer is presented as a language specific “long”. Integers are useful for: Timestamps, counters and any numeric value.

Bytes – BLOB

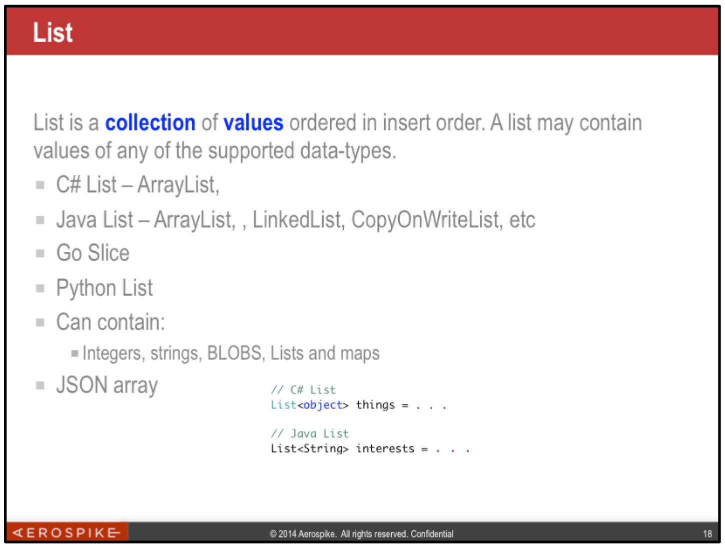
Bytes are byte arrays of a specific size. This allows any binary data of any type to be stored. Note these are not NULL terminated. The greatest efficiency can be used with binary objects (blobs). These are limited in size only by the size of the record in total. Many deployments will use their own serializer, and possibly compress that object, and store that object directly. Doing so means that data can't easily be accessed through a UDF.

Language serialize objects

If you pass a complex language type - like your own class in Java - into a data call, the Aerospike client will use the language's native serialization system. That data will be stored with a "blob type" specific to the language. This allows a client of the same language to read data with clean code, but most language's default serializers are poor.

Floats

There is no native float type currently. Consider using “fixed point” arithmetic, where a real number is represented with a fixed number of digits before, and after, the radix point (decimal point).



List

List is a **collection** of **values** ordered in insert order. A list may contain values of any of the supported data-types.

- C# List – ArrayList,
- Java List – ArrayList, , LinkedList, CopyOnWriteList, etc
- Go Slice
- Python List
- Can contain:
  - Integers, strings, BLOBS, Lists and maps
- JSON array

```
// C# List
List<object> things = . . .

// Java List
List<String> interests = . . .
```

18

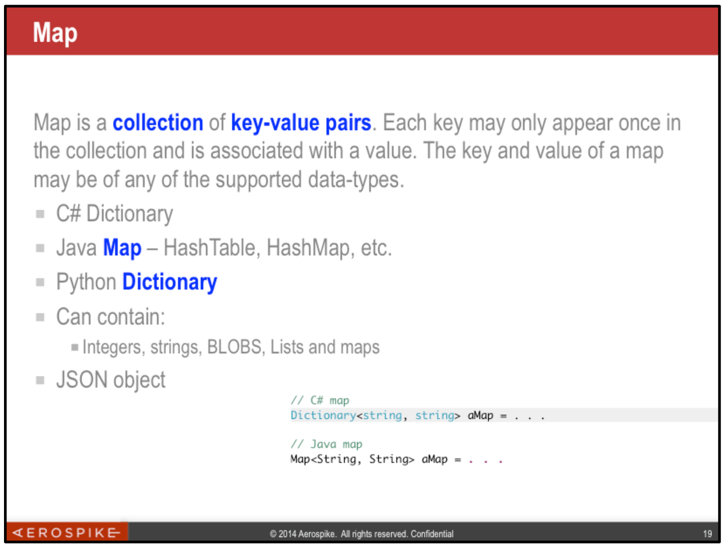
Lists

List is a collections of values ordered in insert order. A list may contain values of any of the supported data-types, including other Lists and Maps. The size is limited only by the maximum record size (default of 128k)

A list is ideally suited to storing a JSON Array and is store internally in a language neutral way. So a list can be written in C# and read in Python with no issues. A combination of Lists and Maps allows you to store JSON documents. It is better to use a List rather than a JSON Array as a string.

Unless you use a UDF (described in a later module) to manipulate elements of the list, you store and retrieve the whole list from the client API. So do consider the network cost when using sizable lists.

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



Map

Map is a **collection** of **key-value pairs**. Each key may only appear once in the collection and is associated with a value. The key and value of a map may be of any of the supported data-types.

- C# Dictionary
- Java **Map** – HashTable, HashMap, etc.
- Python **Dictionary**
- Can contain:
  - Integers, strings, BLOBS, Lists and maps
- JSON object

```
// C# map
Dictionary<string, string> aMap = . . .

// Java map
Map<String, String> aMap = . . .
```

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 19

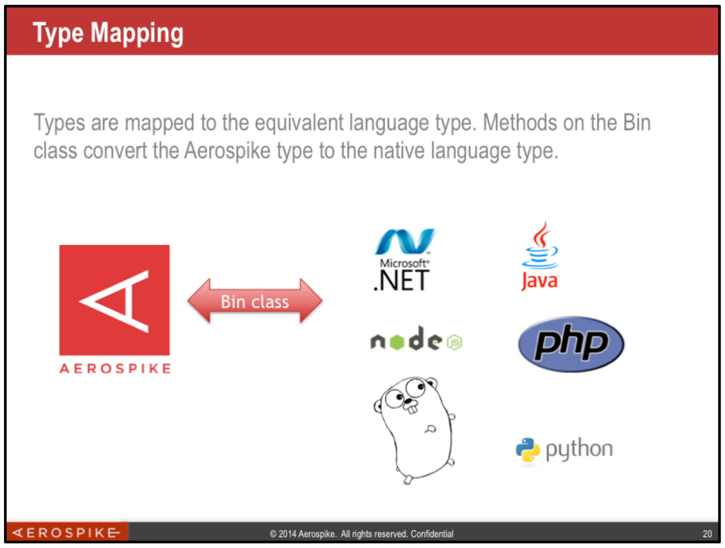
Map

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

A Map is ideally suited to storing a JSON Object and is store internally in a language neutral way. So a Map can be written in Java and read in C# with no issues. A combination of Lists and Maps allows you to store JSON documents. It is better to use a Map rather than a JSON document as a string.

Unless you use a UDF (described in a later module) to manipulate elements of the map, you store and retrieve the whole map from the client API. So do consider the network cost when using sizable Maps.

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



Type Mapping

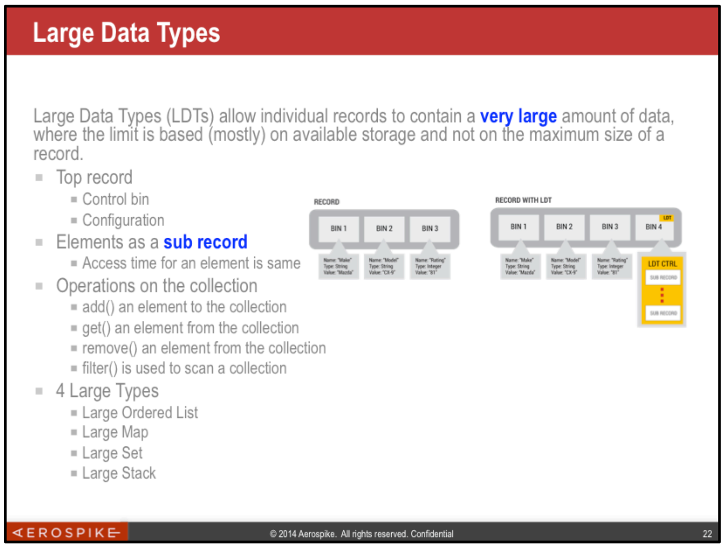
Types are mapped to the equivalent language type. Methods on the Bin class convert the Aerospike type to the native language type.

Bin class
AEROSPIKE
Microsoft .NET
Java
node
php
python

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

Aerospike	C#	Java
Integer (8 bytes unsigned)	Long	Long or long
String	string	String
BLOB	Byte[]	Byte[]
List	List	List
Map	Dictionary	Map

Large Data Types



Large Data Types

Large Data Types (LDTs) allow individual records to contain a **very large** amount of data, where the limit is based (mostly) on available storage and not on the maximum size of a record.

- Top record
  - Control bin
  - Configuration
- Elements as a **sub record**
  - Access time for an element is same
- Operations on the collection
  - add() an element to the collection
  - get() an element from the collection
  - remove() an element from the collection
  - filter() is used to scan a collection
- 4 Large Types
  - Large Ordered List
  - Large Map
  - Large Set
  - Large Stack

RECORD
BIN 1
BIN 2
BIN 3
Name: "Make" Type: String Value: "Mazda"
Name: "Model" Type: String Value: "CX-9"
Name: "Rating" Type: Integer Value: "81"
RECORD WITH LDT
BIN 1
BIN 2
BIN 3
BIN 4
LDT
Name: "Make" Type: String Value: "Mazda"
Name: "Model" Type: String Value: "CX-9"
Name: "Rating" Type: Integer Value: "81"
LDT CTRL
SUB RECORD
SUB RECORD

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 22

Large Data Types

Large Data Types (LDTs) allow individual records to contain a very large amount of data, where the limit is based (mostly) on available storage and not on the maximum size of a record.

Elements in an LDT are stored in Sub-records. Sub-records are very similar to regular Aerospike records, with the main exception that they are linked to a parent record. They share the same partition address and internal record lock as the parent record, so they move with their parent record during migrations and they are protected under the same isolation mechanism as their parent record.

An LDT bin contains a complex control structure (called ldtCtrl) that defines the layout and configuration of an LDT instance. The LDT control Bin describes the configuration and storage containers for an LDT collection.

The storage container is a new type of record, called a "sub-record". Sub-records are very similar to regular Aerospike records, with the main exception that they are linked to a parent record. Sub-records share the same partition address as their parents as well as other resources, such as internal record locks. Because they share locks, they are protected under the same isolation mechanism as their parent record. Sub-records move with their parent record during migrations. Thus, LDT objects take advantage of Aerospike's robust replication, re-balancing and migration mechanisms that ensure immediate consistency and high availability. LDT objects are processed in-database, on the server, using client side APIs.

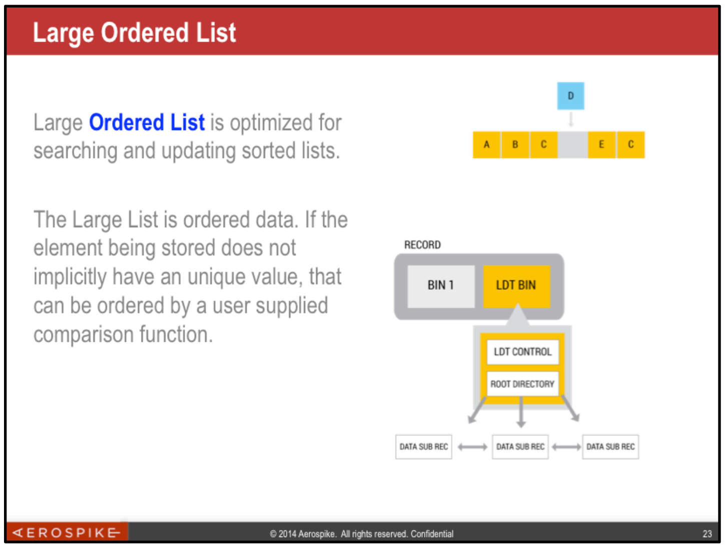
An LDT object comprises multiple sub-records (with sizes ranging roughly from 2kb to 1mb. The sub-records are indexed and linked together, and managed in-database. The use of sub-records means that access to an LDT instance typically affects only a single sub-record, rather than the entire record bin value.

Sub-Record

A sub-record is a record which is linked to a parent record. A parent record is a regular record, which applications interact with to store and retrieve data. Unlike a regular record, the sub-record is not directly accessible. Instead, the sub-record is only accessed via the regular (parent) record.

Using parent records and sub-records, linked data structures can be created to store large amounts of data. If you look at it as a tree, then the parent record would be the root node of the tree, and each sub-record is either a branch or leaf node of the tree. In order to access any branch or leaf, you would need to begin from the root.

Although sub-records are not stored contiguously with the parent record, they are stored on the same partition as the parent record thus allowing the sub-records to be replicated and migrated with the parent record.



Large Ordered List

Large **Ordered List** is optimized for searching and updating sorted lists.

The Large List is ordered data. If the element being stored does not implicitly have an unique value, that can be ordered by a user supplied comparison function.

D
A
B
C
E
C
RECORD
BIN 1
LDT BIN
LDT CONTROL
ROOT DIRECTORY
DATA SUB REC
DATA SUB REC
DATA SUB REC

Large Ordered List

Large Ordered List (llist) is optimized for searching and updating sorted lists. It can access data at any point in the collection, while still being capable of growing the collection to virtually any size.

The Large List is particularly suited for storing any type of ordered data, either simple values (numbers, string) or complex objects (lists, maps, documents). If the object being stored does not implicitly have an atomic value that can be compared (and thus ordered), then the user must supply a function that extracts a simple (atomic) value from the object that can be used for comparison.

Features

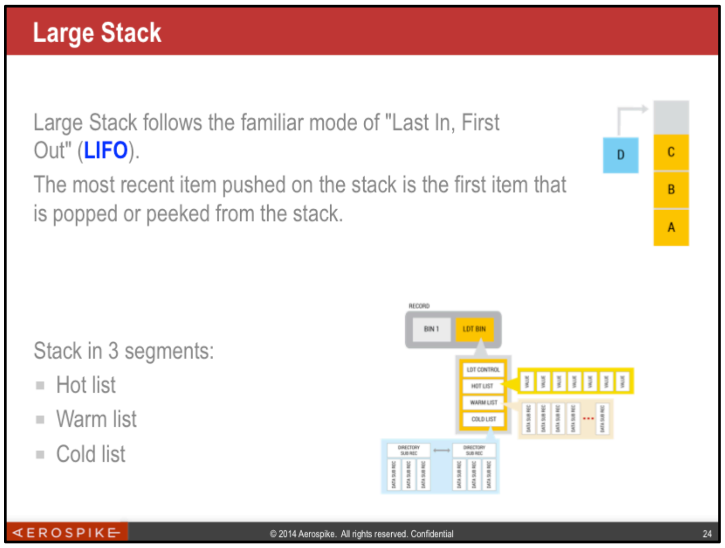
Large Ordered List has the following capabilities:

- Atomic or Complex object management
- Infinite Storage
- UDF Predicate Filters, UDF Transformations
- Single-item or multi-item insert
- Single-value search, with optional UDF filter
- Range value search, with optional UDF filter
- Full scan, with optional UDF filter
- Single-item Update
- Min and Max search
- Single-Value Delete

Use Cases

Large ordered lists are suited for storing any type of ordered data. The data can be simple atomic values, such as numbers or strings, or it can be complex values, such as lists, maps or documents. In the case where the objects are atomic, the large ordered list can provide simple list management (e.g. name or number retrieval). In the case where the objects are complex, the large ordered list can provide indexing services, such as a primary or secondary index, over the set of objects.

A common use case would be to employ LLIST as a primary index, where there is a collection of complex objects (e.g. maps). LLIST would provide fast search and update capabilities to the objects. Note that LLIST can be used in a manner similar to LMAP, with lookup of a key value that corresponds to an object. However the main difference between LLIST and LMAP is that LLIST supports range queries. So, with LLIST, it would be possible to search for all names beginning with "L", or for all names from "L" to "T". In the



Large Stack

Large Stack follows the familiar mode of "Last In, First Out" (**LIFO**).

The most recent item pushed on the stack is the first item that is popped or peeked from the stack.

D
C
B
A

Stack in 3 segments:

- Hot list
- Warm list
- Cold list

RECORD
BIN 1
LDT BIN
LDT CONTROL
HOT LIST
WARM LIST
COLD LIST
DIRECTORY SUB REC

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 24

Large Stack

A Large Stack follows the familiar mode of "Last In, First Out" (LIFO). That means that the most recent item pushed on the stack is the first item that is popped or peeked from the stack. Although not strictly time-based, the insert order of the items on a stack typically denote the order that the items were created or discovered.

Although directed by a client-side LStack function calls, the operations are performed in the database server. Significant amounts of data can be fetched and filtered on the server before it is returned to the client.

The Large Stack data storage is organized in a tiered manner. To achieve the best possible performance, access to the top of the stack is fastest. The "Warm Data" is one I/O indirection away. The "Cold Data" is two I/O indirections away

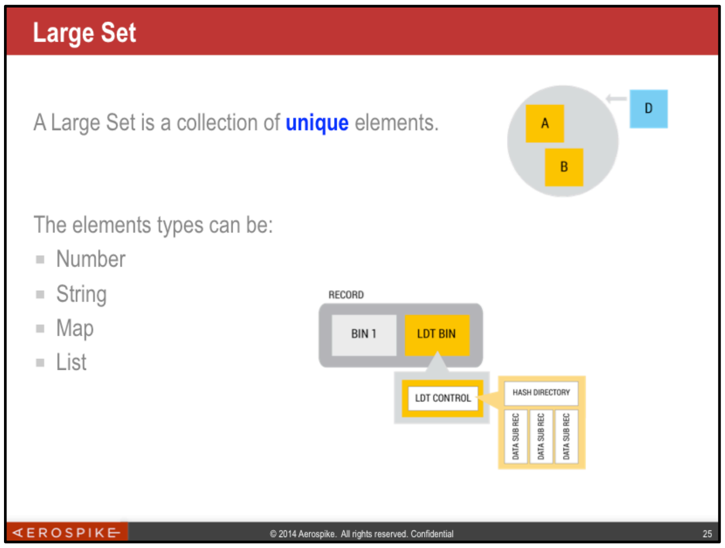
Since the "Hot List" data is stored directly in the parent record, access to the hot list data immediate; there is no additional I/O involved for reading or writing. The sizes are configurable to match user needs, but the default Hot List size is 100 objects.

The parent record also includes a "Warm List", which is a list of sub-record pointers. The default Warm List size is also 100, and the default sub-record capacity is 100, which gives the Warm List a default capacity of 10,000 objects. As is the case with the Hot List, all of these parameters can be set by the administrator for optimal tuning.

Finally the parent record contains a pointer to the cold data directory – a linked list of sub-records containing directories, where each directory contains pointers to data sub-records. Each Cold List directory node is basically the equivalent of a Warm List. Stack objects naturally migrate from the Hot List to the Warm List and then to the Cold List over time. This tiered organization ensures that access to the Hot List incurs no additional I/O, access to the Warm List incurs only 1 additional I/O, and access to the Cold List incurs 2 or more I/Os.

Since the size of the Cold List is not bounded, a large stack instance is not limited to any size. Also, since data naturally migrates from hot to warm to cold, the size of a large stack can be capped at a certain size. Data that migrates beyond the cap is implicitly discarded and the sub-record storage is reclaimed.

Contents of an lstack are stored in a tiers: hot, warm and cold. The "hot" tier are the data items which were most recently written. Once the "hot" tier is full, the older items will be pushed into the "warm" tier. Once the "warm" tier is full, the older items will be pushed into the "cold" tier.



Large Set

A Large Set is a collection of **unique** elements.

The elements types can be:

- Number
- String
- Map
- List

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 25

Large Set

A Large Set (lset) is a collection of unique elements. The elements can simple types (e.g. number, string) or complex (e.g. map, list). The lset maintains an index for fast and efficient searches to determine whether an element exists.

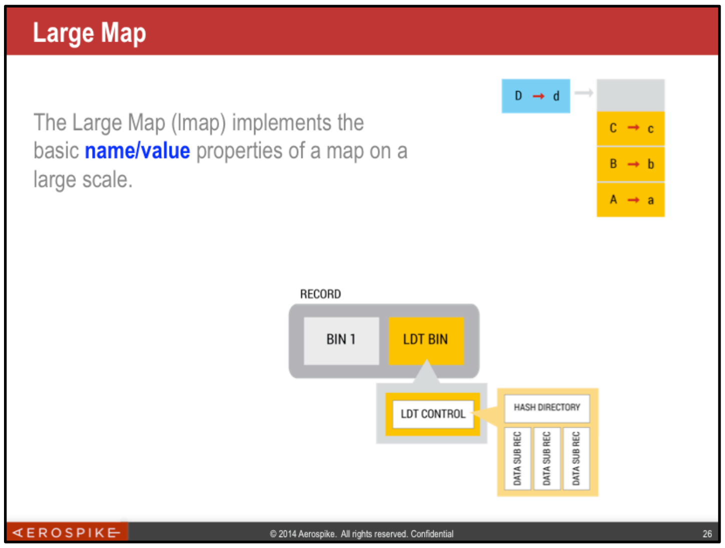
The LSET control Bin contains the Large-Map-specific configuration settings and the Large Map Hash Table. The hash table uses the linear hashing algorithm, where the hash table grows with the size of the set.

The hash table contains cells, where each cell status shows one of the following:

- Empty
- Short List
- Single Sub-Record
- Radix Tree of multiple Sub-Records

Since the maximum size of the large set is dependent upon the size of the hash table that resides in the LDT bin of the top record, a large set is implicitly bound a size no greater than 2 gigabytes, and possibly less than that if other large values reside in the record.

The Large Set values are stored in an internal hash table. The hash is computed either on the entire object (if it is an atomic type, such as number or string); or the hash is computed on the entire "stringified" object, if object is complex (e.g. list or map); or the hash is computed on a sub-set of the complex object, if the user has supplied an appropriate "unique_identifier()" function.



Large Map

The Large Map (lmap) implements the basic **name/value** properties of a map on a large scale.

D → d
C → c
B → b
A → a
RECORD
BIN 1
LDT BIN
LDT CONTROL
HASH DIRECTORY
DATA SUB REC
DATA SUB REC
DATA SUB REC
AEROSPIKE
© 2014 Aerospike. All rights reserved. Confidential
26

Large Map

The Large Map (lmap) implement the basic name/value properties of a map, on a large scale.

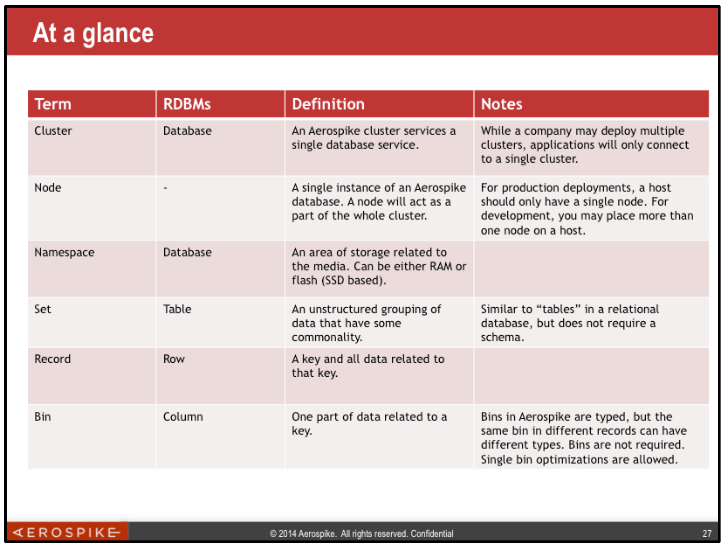
Large Maps are suited for storing dictionaries, where there is an simple name value (number, string) and a document or object value.

Similar to Large Sets, the LMAP LDT control Bin contains the Large-Map-specific configuration settings and the Large Map Hash Table. The hash table uses the linear hashing algorithm, where the hash table grows with the size of the set (Figure 4b).

The hash table contains cells, where each cell status shows one of the following:

- Empty
- Short List
- Single Sub-Record
- Radix Tree of multiple Sub-Records

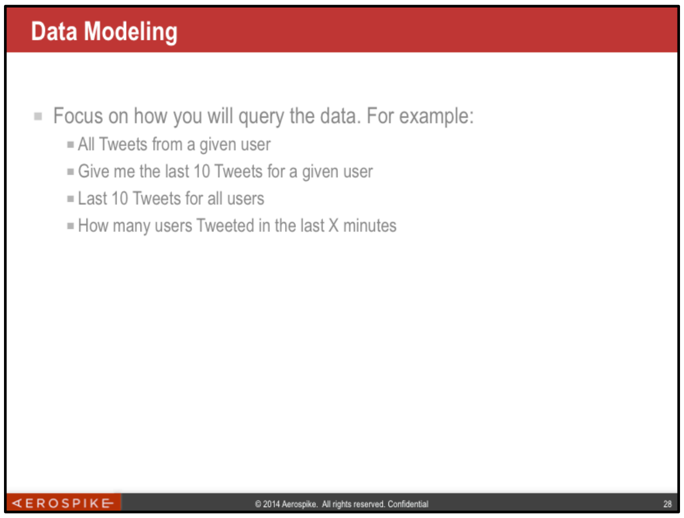
Since the maximum size of the large map is dependent upon the size of the hash table that resides in the LDT bin of the top record, a large map is implicitly bound a size no greater than 2 gigabytes, and possibly less than that if other large values reside in the record.



At a glance

Term	RDBMs	Definition	Notes
Cluster	Database	An Aerospike cluster services a single database service.	While a company may deploy multiple clusters, applications will only connect to a single cluster.
Node	-	A single instance of an Aerospike database. A node will act as a part of the whole cluster.	For production deployments, a host should only have a single node. For development, you may place more than one node on a host.
Namespace	Database	An area of storage related to the media. Can be either RAM or flash (SSD based).	
Set	Table	An unstructured grouping of data that have some commonality.	Similar to "tables" in a relational database, but does not require a schema.
Record	Row	A key and all data related to that key.	
Bin	Column	One part of data related to a key.	Bins in Aerospike are typed, but the same bin in different records can have different types. Bins are not required. Single bin optimizations are allowed.

AEROSPIKE



Data Modeling

- Focus on how you will query the data. For example:
  - All Tweets from a given user
  - Give me the last 10 Tweets for a given user
  - Last 10 Tweets for all users
  - How many users Tweeted in the last X minutes

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 28

De-normalized data is OK

De-normalization is a time-space trade-off. Normalized data takes less space, but may require join to construct the desired result set, hence more time. If it's de-normalized, data is replicated in several places. It then takes more space, but the view of the data is immediately available.

The basic idea of de-normalization is that you'll add redundant data, or group some, to be able to get that data more easily -- which is better for performances.

Historical data, data that is written once and kept forever, is a perfect candidate. Audit trail entries, click streams, or buy/sell transactions in a market, are historical and never change.

Focus on how you will query the data

Consider a ”User" and a ”Tweets" table, for a twitter like application, where the

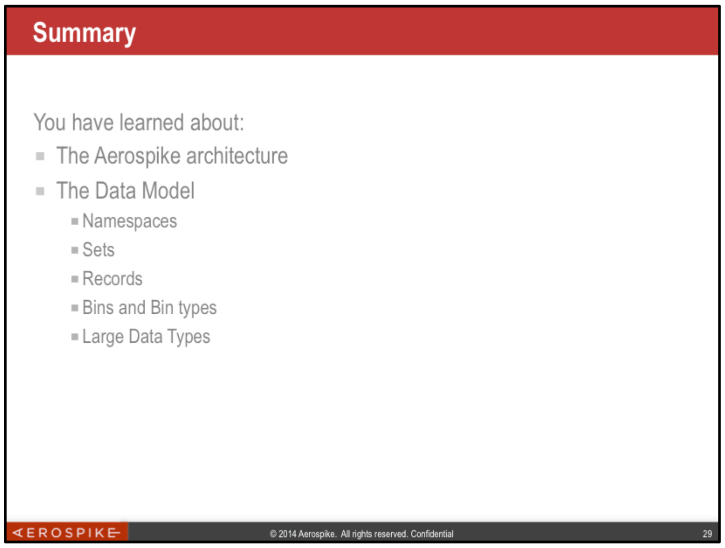
user records has:

- id = “peter-1”
- Count of tweers
  - For each User, you'll have several entries in the ”Tweet" table
  - This means that, to display a list of Tweets for a User, you'll have to :
    - Do one query on the User
    - Do one query on the Tweets related to the User
    - (Yes, those can be merged into only one, but they are still 2 sub queries)

Alternately, if you formed the Tweet primary key by concatenating it to the user id:

“peter-1:1”, “peter-1:2”, “peter-1:3”, etc

you can retrieve the tweets in a single “batch read”. You only need one operation.



Summary

You have learned about:

- The Aerospike architecture
- The Data Model
  - Namespaces
  - Sets
  - Records
  - Bins and Bin types
  - Large Data Types

AEROSPIKE © 2014 Aerospike. All rights reserved. Confidential 29

AEROSPIKE