Aerospike Logo

Enterprise Data Migrator (EDM)

APPLICATION DOCUMENT

Table of Contents

[EDM Overview 2](#_Toc513117212)

[Different Source Data Types 4](#_Toc513117213)

[Cassandra – 4](#_Toc513117214)

[MongoDB 5](#_Toc513117215)

[Couchbase 6](#_Toc513117216)

[RedisLab 7](#_Toc513117217)

[DynamoDB 8](#_Toc513117218)

[MySQL 8](#_Toc513117219)

[MS Excel and CSV Data Files 9](#_Toc513117220)

[Destination 10](#_Toc513117221)

[Aerospike 10](#_Toc513117222)

[Sysout 11](#_Toc513117223)

[JSON Output File 11](#_Toc513117224)

[Channels 12](#_Toc513117225)

[EDM Application Flow – 13](#_Toc513117226)

[Entry point 13](#_Toc513117227)

[Source Data Implementation Classes 16](#_Toc513117228)

[Destination Implementation Classes 16](#_Toc513117229)

[Abstract Classes 16](#_Toc513117230)

[Common Implementation Classes 17](#_Toc513117231)

[Admin Classes for Telnet Service 17](#_Toc513117232)

[EDM Basic Application Flow 18](#_Toc513117233)

# EDM Overview

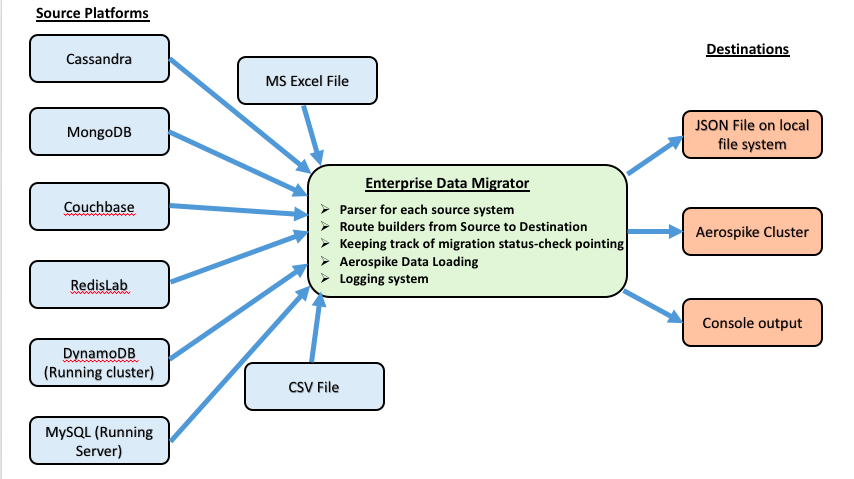
Enterprise Data Migrator utility is for transferring data from any and/or each of the following different sources onto Aerospike cluster.

Presently it supports following platforms as data sources in addition to CSV and Excel formatted data–

1. Cassandra
2. MongoDB
3. Couchbase
4. RedisLab
5. DynamoDB
6. MySQL

Above six platforms, and CSV and Excel formatted files are referred as “Source” systems and Aerospike platform is the only “Destination” platform for this utility application. Along with writing data onto Aerospike, users can also configure this utility to display data on console/standard out and also into a JSON formatted file on local file system.

Out of above mentioned 6 systems, EDM expects data dumps for the first 4 platforms in their respective formats. However, for DynamoDB and MySQL, EDM expects to be connected to the running Dynamo cluster and MySQL server.



Picture 1

In addition to migrating data from source to destination, EDM also keeps track of migration status by implementing it’s own check-pointing mechanism. It also has the capability for generating logs in different granularity conforming to Java Log4J specifications.

EDM also integrates additional two things –

1. Graphite Reporting service
2. Telnet service

JSON formatted configuration file of Data Migrator lets users add following configurations for extending EDM runtime performance metrics to Graphite Reporting server –

"graphiteReporterInterval" : 10,

"graphiteReporterHost" : "127.0.0.1",

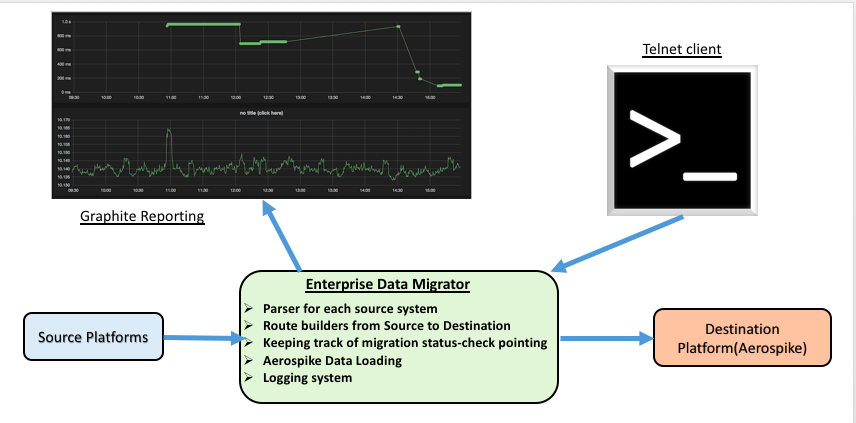
"graphiteReporterPort" : 22003,

"graphiteReporterPrefix" : "j",

"graphitePickeled" : false,

"graphitePickeledBatchSize" : 1,

It also allows users to provide a tcp port in it’s configuration file to be used for opening a telnet service associated with that port. By implementing this service, EDM extends it’s runtime monitoring to remote users. EDM applications with these 2 additional extensions is described pictorially below.



Picture 2

# Different Source Data Types

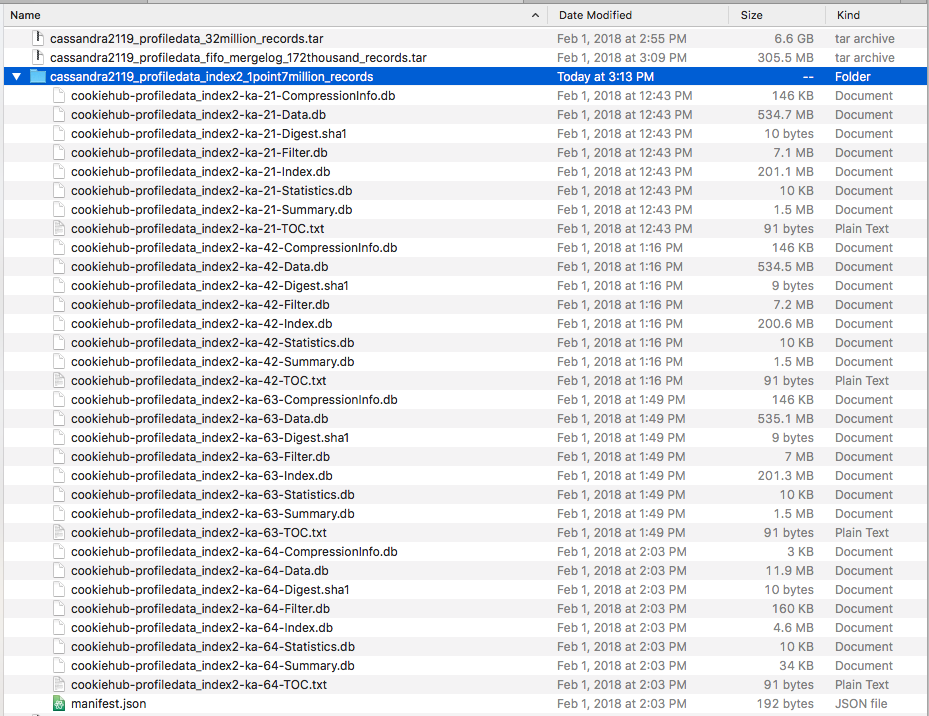
## Cassandra –

EDM expects Cassandra data in SSTable format. User has to take a Cassandra snapshot backup using nodetool utility, command for this is 🡺 nodetool -h <host/ip address> -p 7199 snapshot <keyspace name>.

Snapshot backup is created inside Cassandra keyspace data directory with snapshot name inside snapshots directory. If not specifically mentioned in Cassandra.yaml file as data location, Cassandra data directory is created as “$CASSANDRA\_HOME/data/data/<keyspace name>/<cf name>”. By default for a particular column family (cf/table), snapshot backup is created in “$CASSANDRA\_HOME/data/data/<keyspace name>/<cf name>/snaphots/<snaphot name>”.

For a Cassandra installation in “/Users/saikat/cassandra/dsc-cassandra-3.0.9” directory, with keyspace name as “cookiehub” and column family name “profiledata”, snapshot backup location will be something like as follows –

“/Users/saikat/cassandra/dsc-cassandra-3.0.9/data/data/cookiehub/profiledata-7e5a73b0fd4411e79b97c918e04e3955/snapshots/1525455575653”



Picture 3

Above directory listing is an example of Cassandra snapshot for a keyspace “cookiehub” and table “profiledata\_index2”. All the file names shown above ending with “-Data.db” are actual data dumps while other files are related metadata for these data files.

Cassandra nodetool snapshot utility also generates the metadata/schema file for each column family in the snapshot directory with a name as “schema.cql”. Snapshot backup location and schema file location – both need to be provided in EDM Cassandra source configuration.

User has to provide following information for Cassandra as a data source in EDM –

{

"id": "sstable",

"impl": "com.aerospike.cs.edm.cassandra.CassandraSSTableSource",

"url": "file:///Users/rashidmayes/Documents/workspace/EnterpriseDataMigrator/examples/cassandra",

"sortField" : "DATE",

"sortDirection" : "ASC",

"sourceType": "cassandra",

"properties": {

"keyspace" : "cookiehub",

"partitioner" : "org.apache.cassandra.dht.Murmur3Partitioner",

"schema" : "file:///Users/rashidmayes/Documents/workspace/EnterpriseDataMigrator/examples/cassandra/schema.cql"

}

}

Parameter “impl” identifies the Java class within this same package which does the parsing of SSTable data.

“url” identifies location of Cassandra SSTable data.

Parameters “sortField” and “sortDirection” are provided to inform EDM the order of parsing and loading data in case of availability of multiple SSTable db backup files.

Users has to tell EDM about Cassandra “keyspace” and “[partitioner](https://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architecturePartitionerAbout_c.html)” information in the format as shown above.

Finally EDM also needs to know the data model or schema for the Cassandra tables which are included in the Cassandra snapshot backup. That information is provided by "schema" parameter as the location of the corresponding CQL file.

## MongoDB

EDM accepts MongoDB data backup in binary BSON format. For migrating data from a small deployment of MongoDB, users may use mongodump/mongorestore utilities to generate BSON files. For a very large and distributed MongoDB cluster users should adopt to other appropriate [backup methods](https://docs.mongodb.com/manual/core/backups/).

Similar to Cassandra, following configuration section is for Mongo BSON data source to be used by EDM –

{

"id": "mongodb",

"impl": "com.aerospike.cs.edm.mongo.MongoSource",

"url" : "file:///Users/saikat/Documents/MongoDB",

"sortField" : "DATE",

"sortDirection" : "DESC",

"sourceType": "mongodb",

"properties": {

"mongodb.maxscannerspersource": 1

}

}

Parameter “impl” identifies the class within EDM package which scans, reads and parses MongoDB data from BSON files.

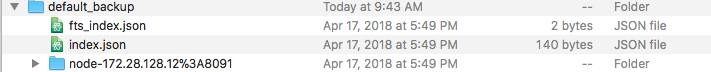
“url” is the location of BSON files on local file system.

In case of multiple BSON files in the above mentioned file system location, “sortField” and “sortDirection” parameter values specify the ordering for parsing those files.

The parameter “maxscannerspersource” determine the number of allocated threads in the thread pool for scanning, reading and parsing Mongo data backup from BSON files.

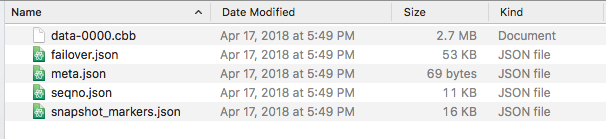
## Couchbase

Data Migration application requires Couchbase data in it’s backup form for loading onto Aerospike. A typical Couchbase backup structure looks as follows –



Picture 4

Node folder as shown above contains actual data backup in .cbb format, which is parsed by EDM.



Picture 5

Source configuration section for Couchbase is as follows -

{

"id": "couchbase",

"impl": "com.aerospike.cs.edm.couchbase.CouchbaseSource",

"url" : "file:///Users/saikat/Documents/Couchbase/default\_backup/",

"sortField" : "DATE",

"sortDirection" : "DESC",

"sourceType": "couchbase",

"properties": {

"couchbase.maxscannerspersource": 1

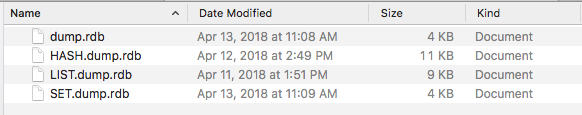
}

}

Exactly same as MongoDB. Parameters “impl”,”url” identify the class file within this EDM package responsible for scanning, reading and parsing Couchbase backup files. “sortField” and “sortDirection” determine the order of processing in case of multiple cbb backup files are present.

## RedisLab

EDM expects RedisLab data dumps in “.rdb” format. Users can use “redis-cli” interface and run “save” command to create “dump.rdb” for any particular Redis database. Unlike Couchbase, Redis doesn’t have any particular directory structure for it’s backup files. Users can put just the “.rdb” files in any location and point that location in sample.json file’s Redis source configuration location. Any filename with “.rdb” at the end will be picked up for parsing and loading onto Aerospike.



Picture 6

Source configuration section of sample.json for Redis looks as follows –

{

"id": "rdb",

"impl": "com.aerospike.cs.edm.redis.RedisRDBSource",

"url" : "file:///Users/saikat/Documents/Redis/dataTypes",

"sortField" : "NAME",

"sortDirection" : "ASC",

"sourceType": "redis",

"properties": {

"redis.maxscannerspersource": 1

}

}

Parameters are exactly similar and indicate the class name for processing Redis data backup, location for Redis backup files’ location, ordering strategy for scanning and parsing multiple Redis backup files and finally number of allocated threads for the executor pool for scanning, reading and parsing Redis data backup file for loading onto Aerospike cluster.

## 

## DynamoDB

While EDM expects backup dump files for all the above data source types for extracting data to load onto Aerospike cluster, for DynamoDB data source, Data migration tool establishes live connection with the source cluster on Amazon cloud.

Following is the source configuration section of Dynamo in the EDM configuration –

{

"id": "dynamodb",

"impl": "com.aerospike.cs.edm.dynamodb.DynamoDBSource",

"url" : "http://US\_EAST\_1/example2#Id",

"sortField" : "DATE",

"sortDirection" : "DESC",

"sourceType": "couchbase",

"properties": {

"aws.accesskey" : "XXXXXXXXXXXXXXXXX",

"aws.secretkey" : "YYYYYYYYYYYYYYYYYYY",

"dynamodb.ratelimit" : 100,

"dynamodb.scanthreads" : 4,

"dynamodb.consumerthreads" : 8

}

}

While “impl” parameter indicates the class name within the package responsible for extracting DynamoDB data for loading onto Aerospike cluster, “url” parameter here is actually the HTTP location identifying the Dynamo database storage onto AWS cloud.

“aws.accesskey” and “aws.secretkey” are required login credentials for establishing connection with DynamoDB cluster. EDM utilizes DynamoDB Bootstrap Worker and parallel scanner for reading and parsing data from DynamoDB. Following 3 parameters like “dynamodb.ratelimit”, “dynamodb.scanthreads” and “dynamodb.consumerthreads” are essential for instantiating parallel scanners for DynamoDB.

## MySQL

EDM extracts data (ResultSet) from MySQL server making a JDBC connection and running query, instead of dealing with database backup or dump files.

From connection metadata, EDM extracts table information, and their metadata definition. Then extracts data from each of the tables using “select \*” statement and parsing result set by table’s metadata definition and converts that to a key-value pair for further processing to load into Aerospike cluster.

## MS Excel and CSV Data Files

EDM can also scan and parse Excel spreadsheets and CSV or Delimited files for loading on to Aerospike cluster. Source configuration for these data types are shown below –

{

"id": "xlsx",

"impl": "com.aerospike.cs.edm.xls.ExcelSource",

"url" : "file:///Users/saikat/Documents/Streamsets-Datacollector/data/xlsx",

"sortField" : "DATE",

"sortDirection" : "ASC",

"sourceType": "xlsx",

"properties": {

"excel.maxscannerspersource": 1

}

},

{

"id": "csv",

"impl": "com.aerospike.cs.edm.csv.CSVSource",

"url" : "file:///Users/saikat/Documents/Streamsets-Datacollector/data/csv/store",

"sortField" : "DATE",

"sortDirection" : "ASC",

"sourceType": "csv",

"properties": {

"csv.maxscannerspersource": 1

}

}

“impl” parameter values for each section identify the class file names for respective file type scanners and parsers. “url” value tells EDM location of excel, csv or other delimited files, “sortField” and “sortDirection” parameter values determine order of processing when multiple data source files are present. And finally “maxscannerspersource” indicate number of allocated threads in the pool of scanners for respective data source types.

# Destination

## Aerospike

Purpose of this EDM application is to ingest data from different NoSQL, SQL and/or Spreadsheets/CSV type data sources and load that onto Aerospike cluster on the other side. So from target or destination perspective it is only Aerospike cluster definition and looks as follows in the configuration file –

"aerospike.host" : "172.28.128.4",

"aerospike.port" : 3000

"destinations" : [

{

"id": "aerospike",

"impl": "com.aerospike.cs.edm.out.AerospikeDestination",

"buffer": 200000,

"concurrency": 20,

"rateLimit" : 0,

"properties": {

"aerospike.namespace" : "test",

"set" : "hash",

"aerospike.userkeybinnmae" : "\_userKey",

"aerospike.clientPolicy" : "{ \"maxConnsPerNode\" : 300, \"writePolicyDefault\" : { \"timeout\" : 5000, \"expiration\" : 0 }}"

}

}

],

This destination configuration for Aerospike allows user to specify cluster host address and port number. “impl” parameter value identifies the class file responsible for handling data load on to Aerospike cluster.

“buffer” parameter value signifies size (number of items) of the memory queue ([LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/LinkedBlockingQueue.html)) where source records are stored for processing and loading onto Aerospike. Depth of the queue (number of items stored) depends on the difference of throughputs for data reading and data writing (on to Aerospike cluster). If data writing is faster than data reading rate, queue depth is minimum. Similarly, when Aerospike writing is slower than the rate data is being read from source systems, queue depth increases and may reach closer to the limiting value. However, this [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/LinkedBlockingQueue.html) protects queue overflow or data loss.

“concurrency” and “rateLimit” determine the parallelism and throughput for data transfer.

EDM configuration file offers flexibility to users for specifying Aerospike “namespace” name, “set” name, “user\_key\_bin\_name” and set of Aerospike client policies. Set name is generally derived from the table/CF/Document name mentioned in the source backup files and it over-rides the set name mentioned in Aerospike destination configuration.

Whenever any Aerospike client object will be instantiated, it will include these set of policies. However, within the code, additional client policies can be added too. But to make those effective, codes will have to be recompiled.

## 

## Sysout

During the development phase or for testing this application, a destination “sysout” is available too. If the application is run with destination set as “sysout” only, console writer will take the data from buffer queue and display on console record by record.

{

"id": "sysout",

"impl": "com.aerospike.cs.edm.out.SystemDotOutDestination",

"buffer": 100000,

"concurrency": 150,

"rateLimit" : 0,

"properties": {

"useErr" : true

}

},

Again, “impl” identifies the class file responsible for console writing. Parameters “buffer”, “concurrency”, “rateLimit” determine the parallelism and throughput for writing on the console. Finally, user can control the flag “useErr” for selecting the output for display as “standard out” or “standard error” destination for the program as configured by user.

## JSON Output File

EDM also has an implementation class (JSONOutputDestination) for storing data, extracted from source data backup, in a JSON formatted file on local file-system. It utilizes rolling file appending with a configurable file size.

# Channels

Finally, at the concluding section of the configuration file, users select one or more data-flow-maps for informing EDM data migration route(s) for a particular run.

It is users’ choice to decide whether to migrate all data corresponding to different migration routes in a single run of EDM or in different runs. Accordingly user has to configure channels in EDM configuration document. Following example configuration shows multiple channels in a single run –

"channels" : {

"sstable" : [ "aerospike" ],

"xlsx" : [ "aerospike" ],

"csv" : [ "aerospike" ],

"rdb" : [ "aerospike" ],

"mongodb" : [ "aerospike" ],

"couchbase" : [ "aerospike" ],

"dynamodb" : [ "aerospike" ]

}

Above channels configuration informs EDM to migrate data –

1. From Cassandra (source) snapshot data backup onto Aerospike (destination) cluster following their configuration details
2. From MS Excel Spreadsheet (source) onto Aerospike(destination) cluster
3. From CSV/delimited (source) flat files onto Aerospike (destination) cluster
4. From Redis Lab (source) database dump files onto Aerospike (destination) cluster
5. From MongoDB (source) BSON data files onto Aerospike (destination) cluster
6. From Couchbase (source) backup directories onto Aerospike (destination) cluster
7. From DynamoDB (source) cluster on Amazon Cloud onto Aerospike (destination) cluster

In a single run of EDM.

For testing purpose, just to display data (e.g. Cassandra snapshot backup data) on console, following channel configuration will be appropriate –

"channels" : {

"sstable" : [ "sysout" ]

}

Similarly, for migrating data from Couchbase backup .cbb files onto Aerospike after displaying data on console, following channel configuration will be perfect –

"channels" : {

"couchbase" : [ "sysout" , “aerospike” ]

}

# EDM Application Flow –

## Entry point

* “com.aerospike.cs.edm.App” is the entry point for this application package.
* This class parses configuration file (~config/sample.json) and extracts all source information, destination information and channel/route information

**public** **static** **void** main(String[] args) {

String fileName = ( args.length == 0 ) ? "config/sample.json" : args[0];

File file = **new** File(fileName);

**if** ( file.canRead() ) {

**try** {

ObjectMapper objectMapper = **new** ObjectMapper();

Route route = objectMapper.readValue(file, Route.**class**);

ThreadLocalData.*DefaultBufferSize* = (8192\*2);

App context = **new** App(route);

Logs.*configure*(context);

context.execute();

} **catch** (Exception e) {

Logs.***system***.error(file.getAbsolutePath(),e);

}

} **else** {

System.***err***.println("Unable to read " + file.getAbsolutePath());

System.*exit*(0);

}

System.*exit*(0);

}

* It also verifies checkpoint status for determining already migrated source data files and skipping them

File checkpointFile = **new** File( (mRoute.checkpointFile == **null**) ? "jncheckpoint.json" : mRoute.checkpointFile);

**if** ( checkpointFile.exists() ) {

mCheckpoint = mObjectMapper.readValue(checkpointFile, Checkpoint.**class**);

} **else** {

mCheckpoint = **new** Checkpoint();

}

Logs.***system***.info(mObjectWriter.writeValueAsString(mCheckpoint));

When a source data backup file is read, it’s record count is determined by respective source data reader class file, which also keeps track of the record count being handed over to the destination writer class (Aerospike writer in most cases). When record count reaches the same value as in source data backup file, check pointing marks this file as completed migration. Next time, EDM runs with same source configured, it skips the files which completed migration in previous runs.

* Initializes logging and reporting metrics for console as well as Graphite reporting

**private** **void** initMetrics() {

**if** ( mRoute.consoleReporterInterval > 0 ) {

mConsoleReporter = ConsoleReporter.*forRegistry*(***metrics***)

.convertRatesTo(TimeUnit.***SECONDS***)

.convertDurationsTo(TimeUnit.***MILLISECONDS***).build();

mConsoleReporter.start(mRoute.consoleReporterInterval, TimeUnit.***SECONDS***);

}

**if** ( mRoute.graphiteReporterInterval > 0 ) {

**if** ( mRoute.graphitePickeled ) {

PickledGraphite pickledGraphite = **new** PickledGraphite(**new** InetSocketAddress(mRoute.graphiteReporterHost, mRoute.graphiteReporterPort), mRoute.graphitePickeledBatchSize);

mGraphiteReporter = GraphiteReporter.*forRegistry*(***metrics***)

.prefixedWith(mRoute.graphiteReporterPrefix)

.convertRatesTo(TimeUnit.***SECONDS***).convertDurationsTo(TimeUnit.***MILLISECONDS***).filter(MetricFilter.***ALL***)

.build(pickledGraphite);

} **else** {

Graphite graphite = **new** Graphite(**new** InetSocketAddress(mRoute.graphiteReporterHost, mRoute.graphiteReporterPort));

mGraphiteReporter = GraphiteReporter.*forRegistry*(***metrics***)

.prefixedWith(mRoute.graphiteReporterPrefix)

.convertRatesTo(TimeUnit.***SECONDS***).convertDurationsTo(TimeUnit.***MILLISECONDS***).filter(MetricFilter.***ALL***)

.build(graphite);

}

mGraphiteReporter.start(mRoute.graphiteReporterInterval, TimeUnit.***SECONDS***);

}

}

* Defines decimal number format and

NumberFormat numberFormat = NumberFormat.*getNumberInstance*();

numberFormat.setMaximumFractionDigits(3);

PeriodFormatter formatter = **new** PeriodFormatterBuilder().printZeroAlways().appendDays().appendSuffix("d ")

.printZeroAlways().appendHours().appendSuffix("h ")

.printZeroAlways().appendMinutes().appendSuffix("m ")

.printZeroAlways().appendSeconds().appendSuffix("s ")

.printZeroAlways().appendMillis().appendSuffix("ms").toFormatter();

* Also launch a Telnet service

TelnetServer telnetServer = **null**;

**try** {

**if** ( mRoute.telnetPort > 0 ) {

telnetServer = **new** TelnetServer(**this**);

telnetServer.bind(mRoute.telnetPort);

}

* Then it starts configuring actual data source objects and for different data source types, corresponding configure() methods of respective data source type implementation classes are invoked. Following is for data source MongoDB-

**public** **void** configure(App context, Route.SourceConfig sourceConfig, Listener listener, Checkpoint checkpoint) **throws** Exception {

**super**.configure(context, sourceConfig, listener, checkpoint);

**this**.mPruneCompleted = sourceConfig.getBoolean(context, "mongodb.prunecompleted", **false**);

**this**.mMaxScannersPerSource = sourceConfig.getInt(context, "mongodb.maxscannerspersource", 2);

}

* It also configures destination objects invoking configure() methods of respective destination classes. Following is for Aerospike-

**public** **void** configure(App context, DestinationConfig destinationConfig) **throws** Exception {

**super**.configure(context, destinationConfig);

mPort = destinationConfig.getInt(context, "aerospike.port", 3000);

mHost = destinationConfig.getString(context, "aerospike.host", **null**);

String policyJson = destinationConfig.getString(context, "aerospike.clientPolicy", **null**);

**if** ( policyJson != **null** ) {

**try** {

**abstract** **class** MixIn {@JsonIgnore TlsPolicy tlsPolicy;}

***MAPPER***.getDeserializationConfig().addMixInAnnotations(ClientPolicy.**class**, MixIn.**class**);

mClientPolicy = ***MAPPER***.readValue(policyJson, ClientPolicy.**class**);

Logs.***system***.info(***MAPPER***.writerWithDefaultPrettyPrinter().writeValueAsString(mClientPolicy));

} **catch** (Exception e) {

Logs.***system***.error("Cannot read client policy", e);

}

}

mKeyBufferLength = destinationConfig.getInt(context, "aerospike.maxkeylength", -1);

mNamespace = destinationConfig.getString("aerospike.namespace", **null**);

mSet = destinationConfig.getString("aerospike.set", **null**);

mUserKeyBinName = destinationConfig.getString("aerospike.userkeybinnmae", **null**);

mMapPolicy = **new** MapPolicy(MapOrder.***KEY\_VALUE\_ORDERED***, MapWriteMode.***UPDATE***);

}

* Once entire migration data flow routes are built, it instantiates multiple thread-pools with pre-configured number of threads for reading source data

//build channels

mChannels = **new** LinkedHashMap<Source, List<Destination>>();

List<Destination> list;

**for** ( String sId : mRoute.channels.keySet() ) {

source = mSources.get(sId);

list = **new** ArrayList<Destination>();

**for** ( String dId : mRoute.channels.get(sId) ) {

**if** ( mDestinations.containsKey(dId) ) {

list.add(mDestinations.get(dId));

} **else** **if** ( mRoute.failOnInvalidChannel ) {

Logs.***system***.error(sId + " bad channel destination > " + dId);

System.*exit*(0);

}

}

mChannels.put(source, list);

}

//processing sources, enable parallel processing

**int** threadCount = Math.*max*(1, mRoute.sourceConcurrency);

ExecutorService executor = Executors.*newFixedThreadPool*(threadCount);

Runnable r;

**for** (Source source : mChannels.keySet()) {

r = **new** Runnable() {

**public** **void** run() {

source.writeTo(mChannels.get(source));

}

};

executor.execute(r);

}

* Depending on the actual source data type, corresponding implementation classes will instantiate table scanners and data readers and read chunks of data and put them in data buffer for destination writer to consume.

## Source Data Implementation Classes

|  |  |  |
| --- | --- | --- |
| Sl# | Source Data Type | Source Data Configuration and Reader Implementation Class Name |
| 1 | Cassandra | Com.aerospike.cs.edm.cassandra.CassandraSSTableSource |
| 2 | Redis Database | Com.aerospike.cs.edm.redis.RedisRDBSource |
| 3 | Couchbase | Com.aerospike.cs.edm.couchbase.CouchbaseSource |
| 4 | MongoDB | Com.aerospike.cs.edm.mongo.MongoSource |
| 5 | DynamoDB | Com.aerospike.cs.edm.dynamodb.DynamoDBSource |
| 6 | MySQL | Com.aerospike.cs.edm.mysql.MySQLSource |
| 7 | Excel | Com.aerospike.cs.edm.xls.ExcelSource |
| 8 | CSV | Com.aerospike.cs.edm.csv.CSVSource |

## Destination Implementation Classes

|  |  |  |
| --- | --- | --- |
| Sl# | Destination Type | Destination Data Configuration and Writer Implementation Class Name |
| 1 | Aerospike | Com.aerospike.cs.edm.out.AerospikeDestination |
| 2 | sysout | Com.aerospike.cs.edm.out.SystemDotOutDestination |
| 3 | JSON output | Com.aerospike.cs.edm.out.JSONOutputDestination |

## Abstract Classes

This package includes some generic abstract classes to be extended by specific source and destination implementation classes. They are as follows –

1. Com.aerospike.cs.edm.Source
2. Com.aerospike.cs.edm.Destination
3. Com.aerospike.cs.edm.Writable

## Common Implementation Classes

“com.aerospike.cs.edm” includes another a few common general purpose implementation classes as follows –

1. Com.aerospike.cs.edm.Checkpoint
2. Com.aerospike.cs.edm.Constants
3. Com.aerospike.cs.edm.Log
4. Com.arospike.cs.edm.Route
5. Com.aerospike.cs.edm.ScannerDetail
6. Com.aerospike.cs.edm.SimpleWriteListener
7. Com.aerospike.cs.edm.RejectedWriteExecutionHandler

## Admin Classes for Telnet Service

There are 3 implementation classes in “com.aerospike.cs.edm.admin” sub-package for providing Telnet service and related commands for the clients accessing this service provided by EDM. They are –

1. DefaultCommandHandler
2. TelnetCommandHandler
3. TelnetServer

# EDM Basic Application Flow

Read configuration file and cache source information, destination information and channel information

Read check point status file for determining already migrated source data files

Initialize metrics –

Set console report interval

Set Graphite report interval

Define number format for decimal expression

Launch Telnet service at configured port

Build actual dataflow maps (channel configuration) as routes

Starts with configuring sources listed in configuration file. For each type of data sources, configuration defines it’s respective implementation class. Actual methods for configuration of data sources , scanners, readers of different source data types are implemented in the respective source reader classes :

Once migration data flow routes are built, Java multi-threaded, concurrent Executor thread pools are instantiated with thread counts as defined for source concurrency

Different source data implementation classes extract following information from backup data files –

* Identifies actual data backup files (eg. “\*-Data.db”)
* Parses metadata files to extract metadata
* Determines data structure either from schema files or from metadata
* Instantiate Data readers and table scanners
* Read chunks of data from source back up files and put it in buffers
* Invokes destination writer to read and process these source data chunks
* Also deduces various metrics for source data for reporting and logging

Different thread-pools with specified number of threads start extracting data from different data sources

* Aerospike destination writer invokes performWrite() method and finally handleWrite() method to write source data onto Aerospike cluster
* It checks for any plugin configuration for source data filtering before it writes on to Aerospike cluster
* This class derives PK for writing records on to Aerospike using namespace configured, set name configured or set name metadata of the source data record and source data record key
* Finally commits write for the values of different bins as read from source data record on to Aerospike cluster using Aerospike client object

Similarly, for “sysout” destination, it’s implementation class “SystemDotOutDestination” has it’s own writeTask() and performWrite() methods for handling console display for the source data records