We did one migration to AWS

---greenplum

DynamoDB(NoSQL Store,Can’t join)

Any conf change in AWS-DMS

a.> Migrating from MongoDB to DynamoDB

MongoDB DynamoDB

Collection Table

Document Item

Field Attribute

Secondary index Secondary index

DynamoDB supports two kinds of primary keys:

Partition key – A simple primary key, composed of one attribute known as the partition key.

Partition key and sort key – Referred to as a composite primary key, this type of key is composed of two attributes.

The first attribute is the partition key, and the second attribute is the sort key.

DynamoDB provides fast access to items in a table by specifying primary key values.

However, to allow efficient access to data with attributes other than the primary key,

many applications might benefit from having one or more secondary (or alternate) keys available.

MongoDB

{

"\_id": ObjectId("5a03f1d6c029d5af14264744"),

"Year": 2017,

"Month": 10,

"DayofMonth": 7,

"DayOfWeek": 3,

"DepTime": 946,

"CRSDepTime": 915,

"ArrTime": 1037,

"CRSArrTime": 1001,

"UniqueCarrier": "PS",

"FlightNum": 1451,

"ActualElapsedTime": 51,

"CRSElapsedTime": 46,

"ArrDelay": 36,

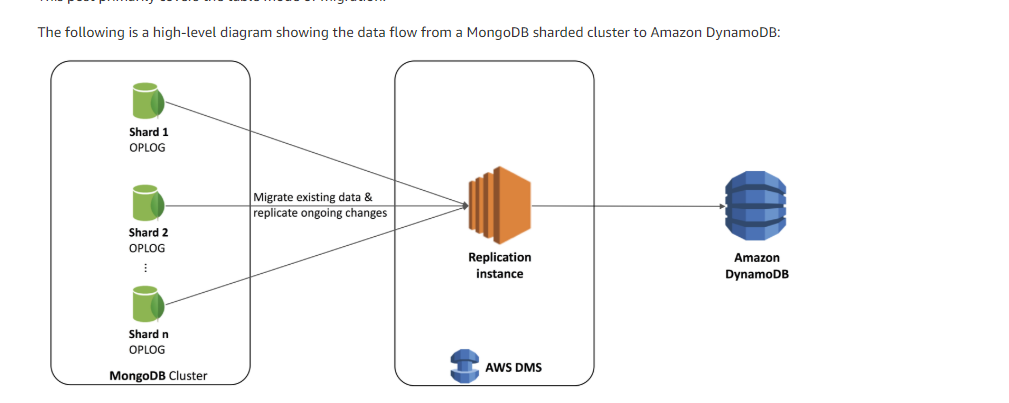
"DepDelay": 31,

"Origin": "SFO",

"Dest": "SAN",

"Distance": 192

}



Migration approach using AWS DMS

AWS DMS supports migration from a MongoDB collection as a source to a DynamoDB table as a target. AWS DMS supports the MongoDB migration in two modes:

Document mode: In this mode, AWS DMS migrates all the JSON data into a single column named “\_doc” in the target DynamoDB table.

Table mode: In this mode, AWS DMS scans a specified number of documents in the MongoDB database and creates a sample schema with all the

keys and their types. During migration, you can use the object mapping feature in AWS DMS to transform the original data from MongoDB to the desired structure in DynamoDB.

Create tasks with a table mapping rule

Both MongoDB and Amazon DynamoDB allow you to store JSON data with a dynamic schema.

DynamoDB requires a unique primary key—either a partition key or a combination of a partition and a sort key.

You need to restructure the fields to create the desired primary key structure in DynamoDB.

Use a composite primary key with a partition key that is a combination of the same fields as in the MongoDB shard key

(the Origin and Year attributes), and a sort key that is a combination of the DayofMonth (day of travel), Month, CRSDepTime

(scheduled departure time), UniqueCarrier and FlightNum attributes.

{

"rules": [

{

"rule-type": "selection",

"rule-id": "1",

"rule-name": "1",

"object-locator": {

"schema-name": "airlinedb",

"table-name": "airline"

},

"rule-action": "include"

},

{

"rule-type": "object-mapping",

"rule-id": "2",

"rule-name": "2",

"rule-action": "map-record-to-record",

"object-locator": {

"schema-name": "airlinedb",

"table-name": "airline"

},

"target-table-name": "airlinedata",

"mapping-parameters": {

"partition-key-name": "depCitybyYear",

"sort-key-name": "depTimeByFlightNum",

"exclude-columns": [

"CancellationCode",

"Diverted"

],

"attribute-mappings": [

{

"target-attribute-name": "depCitybyYear",

"attribute-type": "scalar",

"attribute-sub-type": "string",

"value": "${Origin}-${Year}"

},

{

"target-attribute-name": "depTimeByFlightNum",

"attribute-type": "scalar",

"attribute-sub-type": "string",

"value": "${DayofMonth}-${Month}:${CRSDepTime}|${UniqueCarrier}-${FlightNum}"

}

]

}

}

]

}

Amazon CloudWatch to see all the details---UI

Query patterns

The following are examples of querying data from a MongoDB collection and from a DynamoDB table.

MongoDB:

The following code snippets demonstrate querying data from a MongoDB collection.

Scenario 1: Retrieve all flight details with a delayed departure in San Diego International Airport for the year 2017:

MongoCollection<Document> collection = database.getCollection("airline");

Document filterCondition = new Document("Origin", "SAN")

.append("Year", 2017)

.append("DepDelay", new Document("$gte",0));

// Retrieve documents and display

Block<Document> readData = new Block<Document>() {

@Override

public void apply(final Document document) {

System.out.println(document.toJson());

}

};

// Query all documents matching the filter condition

collection.find(filterCondition).forEach(readData);

// Get count of matching documents

System.out.println(collection.count(filterCondition));

Scenario 2: Retrieve flight status for PS-1451 scheduled to depart on October 14, 2017, at 07:30 AM from San Diego International Airport:

MongoCollection<Document> collection = database.getCollection("airline");

Document filterCondition = new Document("Origin", "SAN")

.append("Year", 2017)

.append("Origin", "SAN")

.append("UniqueCarrier", "PS")

.append("FlightNum", 1451)

.append("Month", 10)

.append("DayofMonth", 14)

.append("CRSDepTime", 730);

// Retrieve documents and display

Block<Document> readData = new Block<Document>() {

@Override

public void apply(final Document document) {

System.out.println(document.toJson());

}

};

// Query all documents matching the filter condition

collection.find(filterCondition).forEach(readData);

DynamoDB:

The following code snippets demonstrate querying data from a DynamoDB table.

Scenario 1: Retrieve details of all flights with a delayed departure from San Diego International Airport for the year 2017:

Table table = dynamoDB.getTable("airlineData");

// Query condition

QuerySpec querySpec = new QuerySpec()

.withHashKey("depCitybyYear" , "SAN-2017") // hashkey name and its value

.withQueryFilters(new QueryFilter("DepDelay").ge(0));

// Query all items matching the filter condition

ItemCollection<QueryOutcome> items = table.query(querySpec);

// Retrieve items and display

items.forEach(System.out::println);

// Get count of matching items

System.out.println(items.getAccumulatedItemCount());

Scenario 2: Retrieve the flight status for PS-1451 scheduled to depart on October 14, 2017, at 07:30 AM from San Diego International Airport:

Table table = dynamoDB.getTable("airlineData");

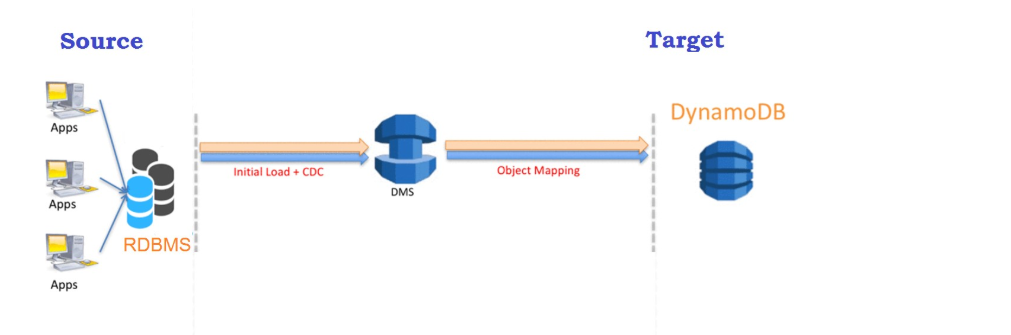
// Retrieve item by its primary key

Item item = table.getItem("depCitybyYear", "SAN-2017", "depTimeByFlightNum", "14-10:730|PS-1451");

// Display retrieved item

System.out.println(item);

b.> RDBMS to DynamoDB



ParallelLoadThreads

* Default Value: 0
* Range: 2–32

The ParallelLoadThreads parameter helps increase the migration speeds with regards to the number of threads to migrate the data. Just be aware that setting a higher value can cause a load on the source instance. A large number of threads can have an adverse effect on database performance because a separate connection is required for each thread. Additionally, there is a provision where the value of this parameter can be increased to more than 32. To do that, you must raise a case with the AWS Support team.

ParallelLoadBufferSize

* Default Value: 50
* Range: 50–1000

The ParallelLoadBufferSize setting specifies the number of data records to store in the buffer. The default is 50 records. This has no impact on large objects (LOBs) for Amazon DynamoDB. Furthermore, the DynamoDB connector only supports character large objects (CLOBs) and national character large objects (NCLOBs), which are just treated as string data types on the DynamoDB target.

CREATE TABLE "FDRGIIT"."BIG\_TABLE"

(

"ID" NUMBER,

"OWNER" VARCHAR2(30 BYTE) NOT NULL ENABLE,

"OBJECT\_NAME" VARCHAR2(30 BYTE) NOT NULL ENABLE,

"SUBOBJECT\_NAME" VARCHAR2(30 BYTE),

"OBJECT\_ID" NUMBER NOT NULL ENABLE,

"DATA\_OBJECT\_ID" NUMBER,

"OBJECT\_TYPE" VARCHAR2(19 BYTE),

"CREATED" DATE NOT NULL ENABLE,

"LAST\_DDL\_TIME" DATE NOT NULL ENABLE,

"TIMESTAMP" VARCHAR2(19 BYTE),

"STATUS" VARCHAR2(7 BYTE),

"TEMPORARY" VARCHAR2(1 BYTE),

"GENERATED" VARCHAR2(1 BYTE),

"SECONDARY" VARCHAR2(1 BYTE),

CONSTRAINT "BIG\_TABLE\_PK" PRIMARY KEY ("ID")

);

REDSHIFT(SQL-on top of PostGreSQL)

REGEXP\_SPLIT\_TO\_ARRAY()---in place of regexp\_substr

Both the databases viz. Redshift and Amazon Dynamo DB works in distributed environment where many nodes are connected to form the cluster and the task or responsibility are distributed on all the nodes. However, both have different usage

**Architecture -**

Dynamo DB is Key Value pair database where each object or single row is identified by the hash value

Amazon RedShift is columnar database where the data is stored in column format and it is built on top of PostgreSQL

**Usage -**

Dynamo DB is used more for real-time transaction where many read and write operations has to be performed.

Amazon Redshift is used for big data analysis

The Reason why dynamoDB is not used for big data analysis is it’s incapability of performing JOIN operations

I'm guessing probably not DynamoDB.  It's a type of NoSQL architecture based on key-value pairs.  You will get sub-second response times on it for reading and writing single keys at a time.  If you need to scan large amounts of data (ie a lot of keys all in one query), you won't like the performance.  
  
Redshift is a column oriented relational ACID compliant and ANSI SQL compliant database.  It mainly shines for doing massive reads of large volumes of data (and crunching it down into something actionable).  
  
Depending on what your cleansing/wrangling rules are, I would say either Redshift or neither.  If you just need to do some basic lookups and mass transformations, Redshift will probably work.  If you need to do something more complex like deduplication based on complex survivability rules, you should look to use something like EMR if you must stay within AWS and then load into Redshift.

# Using the COPY Command to Load from Amazon S3 to Redshift

copy venue

from 's3://mybucket/data/venue\_fw.txt'

iam\_role 'arn:aws:iam::0123456789012:role/MyRedshiftRole'

fixedwidth 'venueid:3,venuename:25,venuecity:12,venuestate:2,venueseats:6';

delimiter '|';

venue.txt: ASCII English text

Amazon RDS vs Redshift vs DynamoDB vs SimpleDB

