S3 Indexer – Querying S3 objects to save $$

Table of Contents

Description

AWS S3 is the one of the most popular data lake solution among the AWS Big Data users. For many customers, over a very short amount of time, this data grows rapidly. It becomes overwhelming to sieve through the billions of s3 objects just to locate that one s3 object. Listing objects is an expensive operation, especially when you have billions of them, even when the objects are organized by proper prefixes. On larger lists it is extremely difficult and/or slow to iterate/lookup using the S3 API or the AWS cli.

Use case

At HLI we are dealing with genomics data with hundreds of buckets, many of them sizing into Petabytes, and containing billions of objects, with a good number of objects in the range of few gigabytes. With several million unused objects, and potentially a lot never to be used again objects, it is necessary to have a cleaning method. It becomes imperative to identify such qualifying objects that we can add life-cycle policies to, or move to glacier, or even delete objects, so managing the objects would be cost effective and efficient. Going through these hundreds of buckets and billions of objects using the s3's list operation can be painstaking if not impossible. Writing script is a little far-fetched and can soon become un-manageable. We need a solution that is more robust, reusable and cost effective. This solution also needs to work for all the existing objects and any new objects that will be written into the buckets. The rate of newly added objects is fairly manageable and hence we break down the solution into two parts. The

first part deals with newly added objects. The second part deals with the existing objects. The first part is relatively easier to manage due to the low volume of objects being written.
In this blog, we will discuss the first part briefly from an architecture point of view. The rest of the blog is focused on second part of this situation i.e. a situation where you have several billions of existing objects across hundreds of buckets. An ideal solution should provide a query-able interface to identify the objects, to which we can apply life-cycle policies like to change storage type to IA or move to glacier or even purge if needed. A SQL/NoSQL view to all the buckets and the objects within the bucket will provide an easy access to select/filter objects that meet the criteria of qualifying objects for either IA/Glacier or deletion, or any other action we would like to take on those objects. We want to be able to answer many analytics questions. A few of such questions are:

- List objects that have been accessed more than a quarter ago
- List objects that have been accessed by an employee who no longer works with the company.
- List objects that have been accessed for a version id that is not the most recent version.
- List duplicate objects by etags, and see when was one or the other copy last accessed.

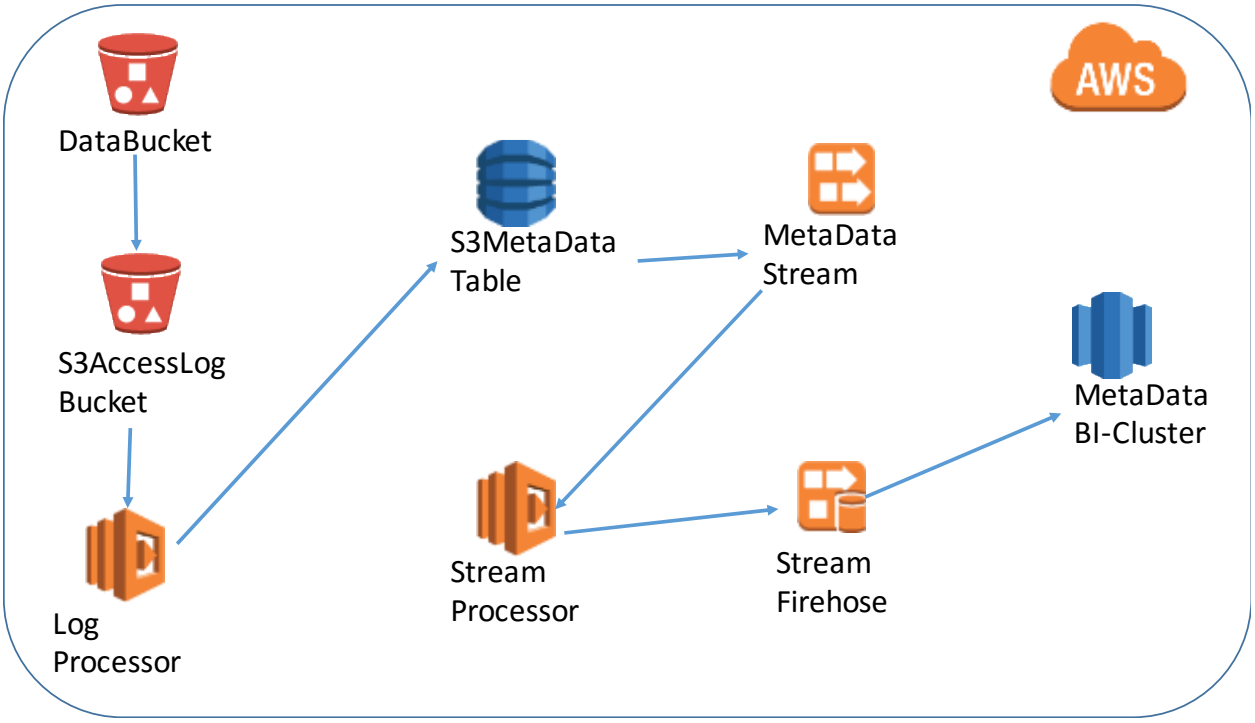
And many such queries that deal with the objects meta data (e.g. etag) and access pattern (e.g. last accessed)

Solution

As mentioned above there are two parts for the complete solution.

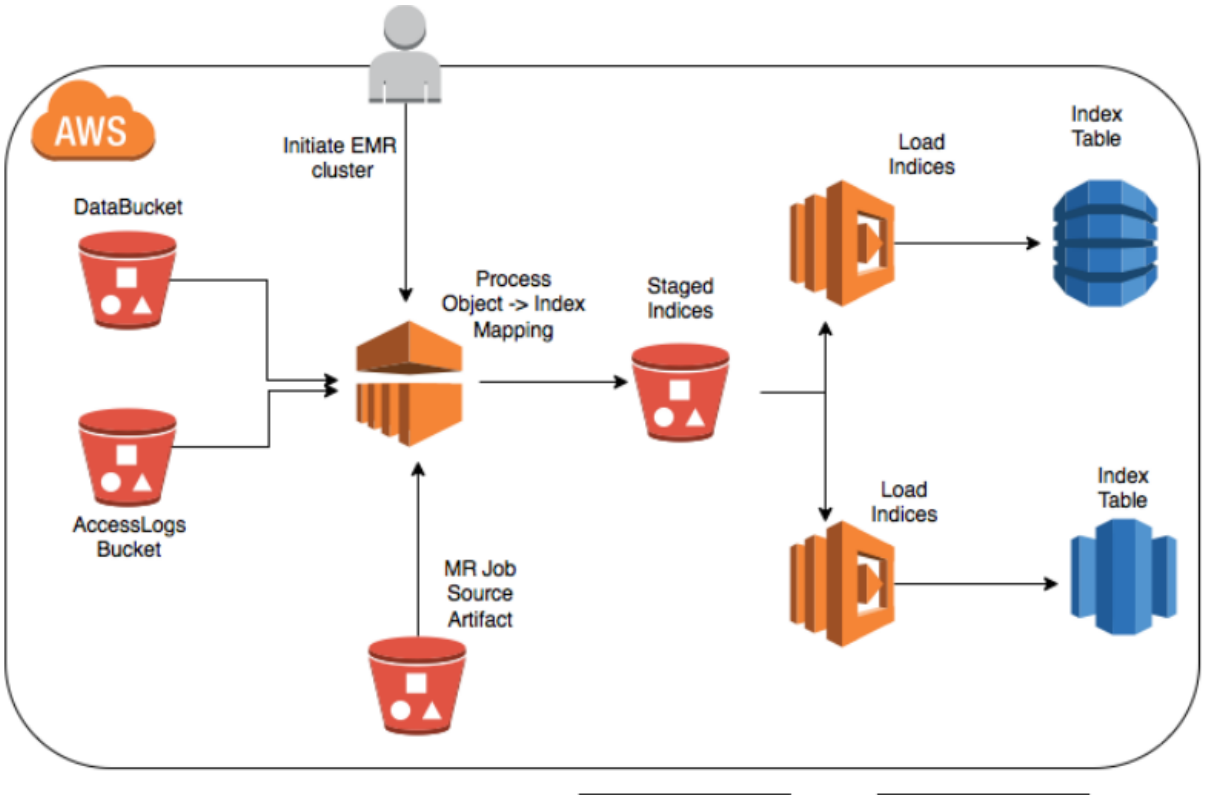
Part 1.

This first part explains how any newer objects can be processed to reflect its information in the final destination, redshift in this case. We develop a lambda function to listen to changes to the log bucket. On an event, the lambda function looks up for the access log information and the meta data information for the objects logged in the access log. The information in the log and the metadata of the object for which it is logged, is both written to a dynamoDB table, which has streams enabled. The streams is an event source for a stream processing lambda function that writes the event to firehose. Firehose can then accumulate data into batches to write it to Redshift.



Part 2.

The following architecture explains how we can use a spark EMR job to process several buckets with billions of objects and store the output into Redshift.



The architecture implements a EMR spark job to go over all the s3 buckets. The spark job first creates a RDD of the meta data of the objects in a bucket. The spark job collects all the contents of the s3 access logs (if enabled) and creates a second RDD. Using the objects metadata, and the access log information for each of these objects, the EMR job, joins these 2 data sets to present one or more rows of the wholistic image of each object. These rows include information about the metadata of the object and the access pattern of the object. This combined output can be written into s3 in a desired format as staged indices. A lambda function can use the staged indices objects as a event source and load this data into redshift. We now, can perform all sorts of analysis to identify objects that qualify for the action that we would like to take for e.g. apply life-cycle policy or move to glacier, or move to SIA.

Details

In the rest of this article, we dive deeper into the second part of the solution i.e. ingesting existing data.

Input

We need to be able to process several buckets with our job. Each of these buckets may be configure to a different s3 logging bucket. An ability to filter specific prefixes is a good to have. We might want to make our input to the job future proof. So we choose a JSON file as a input to our spark job. We create a file with one JSON object per bucket. Each object is a map of dataBucket (the S3 bucket that needs to be indexed), logBucket (the bucket which contains the access log for this data bucket), logPrefix (the prefix

within the logBucket for log file path), and excludePrefix (the prefixes within dataBucket that should be skipped). A typical input JSON object is shown below:

```
{
  "dataBucket" : "gnome-dna-data",
  "logBucket" : "gnome-access-logs",
  "logPrefix" : "logs/",
  "excludePrefix" : "test/"
}
```

The other piece of input we need is the place to write the output of the job to. Ideally we want this to be a S3 object, so we can easily load this to Redshift.

Design the code

Our next task is to setup a spark job. The spark job does the following

a. Go over each JSON object in the input
b. For each of this JSON object, list all versions of s3 objects within dataBucket, excluding excludePrefix. Create a RDD of only the metadata of these objects, keyed by the object key
c. Next, create another RDD of the corresponding logBucket, but this time with the contents of each object in the logBucket, also keyed by the object key
d. Join the 2 RDD's using the object key.

The spark job can parallelize the lookup of objects in the "dataBucket" and join the objects meta data for each version to the access entries in the access logs, on the object key. This would have to be an outer join to ensure that objects not accessed or buckets without logs are not missed out. The sample spark code below can help achieve this

First get the list of all objects from one buckets from the input JSON.

```
/**
  * list all versioned objects recursively in a bucket,
  * Filter objects for prefix if any
  * @param bucket Bucket to list all the objects
  * @param prefix the prefix if any
  * @return list of s3 versioned objects in a bucket
  */
def s3VersionedList(bucket: String, prefix: String = "")
      : java.util.List[S3VersionSummary] = {
  val s3Client = new AmazonS3Client(
                new DefaultAWSCredentialsProviderChain)

  val versionedListRequest = new ListVersionsRequest()
                        .withBucketName(bucket)
                        .withPrefix(prefix)

  var current = s3Client.listVersions(versionedListRequest)
  val keyList = current.getVersionSummaries
```

```
  while( current.isTruncated ) {
    versionedListRequest.setKeyMarker(current.getNextKeyMarker)
    versionedListRequest.setVersionIdMarker(current.getNextVersionIdMarker)
    current = s3Client.listVersions(versionedListRequest)
    keyList.addAll(current.getVersionSummaries)
  }
  keyList.addAll(current.getVersionSummaries)
  keyList
}
```

Create a RDD of contents in the logBucket/logPrefix

```
/** Read contents of access log to extract last accessed time
  * and action performed on the object, e.g. PUT
  * @param sc sparkcontext
  * @param bucket bucket name
  * @param prefix filter by prefix "logs"
  * @return rdd of action, timestamp for each s3 object
  */
def logRdd( sc: SparkContext,
            bucket: String,
            prefix: String = "") :
    org.apache.spark.rdd.RDD[(String, String)] = {

  /** Note: s3List is similar to s3VersionedList except use listObjects call */
  sc.parallelize(s3List(bucket,prefix),1000).map{ key =>
    val s3Client = new AmazonS3Client(
                  new DefaultAWSCredentialsProviderChain)
    val s3Object = s3Client.getObject(bucket, key.getKey)
    val contents: InputStream = s3Object.getObjectContent
    val lines = Source.fromInputStream(contents).getLines()

    val logOut = lines.map( line => {
      val l_rx = """([^\s"\[\]]+|\[[^\]\[]+\]|"[^"]+")""".r
      val tokens = l_rx.findAllIn(line).toArray

                /** Collect your log tokens here */

    })
    logOut
  }.flatMap(l=>l).keyBy ( str => {

                /** key your rdd e.g. by object key and version */

  })
}
```

Create a RDD of meta data of objects listed from dataBucket.

```
val sc = new SparkContext(new SparkConf().setAppName("S3Indexer"))
val summaryObjectList = sc.parallelize(s3VersionedList(dataBucket, excludePrefix))
            .map( summary =>
                    /** Collect objects metadata here */
            ).keyBy ( summary => /** key your rdd e.g. by object key and version */)
```

Join the 2 rdd's and save the output to a desired location, preferably in S3.

```
val joinedRdd = summaryObjectList.leftOuterJoin(logRdd)
joinedRdd.map( tuples =>
  /** format the joined tuples e.g csv/json */
).saveAsTextFile(s3OutPath)
```

EMR Cluster and Step

We can now setup the EMR cluster to execute our spark job. This can be done in either using CLI/Console or cloud-formation template. The spark job can be set up as a step to the EMR cluster. The EMR cluster can be set to terminate after execution. Here we show a sample cli command that adds a spark step to your emr cluster during creation.

```
aws emr create-cluster \
  --applications Name=Hadoop Name=Spark \
  .....
  .....
  --steps "[{\"Args\":[\"spark-submit\",\"--class\", \"com.aws.s3indexer.S3Index\", \"--deploy-mode\",\"cluster\",\"--
conf\",\"spark.executor.instances=4\", \"s3://aws-emr-jobs/source_code/s3-indexer.jar\", \"emr-job-
output\"],\"Type\":\"CUSTOM_JAR\",\"ActionOnFailure\":\"CONTINUE\",\"Jar\":\"command-
runner.jar\",\"Properties\":\"\",\"Name\":\"Spark application\"}]" \
  .....
```

This command should get your EMR cluster started and once it is ready, the spark step will be submitted for execution. The output would end up in aws-emr-output bucket, in this example.

Redshift Cluster

Just like the EMR cluster, a Redshift cluster can be created using your aws console/cli or a cloudformation template. We will not get into those details. You can find steps on creating your redshift cluster here.

After the cluster is created, create a table (s3index) in a database on the redshift cluster, that confirms to the schema written to in your output s3 bucket (aws-emr-output), by the spark job.

Loading to Redshift

Now that we have the output and the destination table, we can use a sql tool to execute a copy command from the s3 to redshift. This loading can easily be done using a Lambda function. The lambda function can use jdbc connector to connect to redshift

and execute a copy command. An example of the copy command is shown below

```
copy s3Index from 's3://aws-emr-output/part*'
credentials '<aws-credentials>' delimiter '\t'
region '<region>' gzip;
```

Depending on the format of your spark output, (e.g. delimited , json), you may need to modify the command accordingly.

Operational Notes

This job may be needed to run one time or occasionally, in order to get this index loaded/refreshed. One may setup additional process, similar to this, to augment the above process.
There are a couple things to note here. The access logs take time in hours to reflect any access requests. So the ongoing changes are not up to date and take time to propagate into the system.

If you use a lambda function, there is a very low possibility that the lambda does not trigger on s3 puts. Not all puts are effectively registered to the function. Lambda does have built-in retries. However, you can easily incorporate additional retry logic on failures to the copy command call. Alternatively you can track the failures in a dynamoDB table, so the failures can be tried again.

Summary

With this solution, HLI has identified and continuous to identify, a good amount of objects, very easily, that can be either

a. Moved to glacier
b. Moved to SIA
c. Deleted
d. Decide versioning

Now the query for these objects is at finger tips, with an easy to use SQL interface, that returns the objects of your criteria in a few seconds. Here is a sample of a few queries

```
-- Problem: Smaller objects that have not been used in a while (to move to Glacier)
-- Solution: Get some data for objects that have been downloaded
select distinct bucketkey, datetimestamp, operation, objectsize/1024/1024/1024 as gb
from s3index
where bucket = 'hli-client-folders'
and operation = 'REST.GET.OBJECT'
order by datetimestamp desc;


-- Problem: Potential buckets to save the most from.
-- Solution: Get total size in TB of folder or bucket.
select bucket, sum(objectsize)/1024/1024/1024/1024 as totaltb
from s3index
group by 1;
```

```
-- Problem: find the least used buckets
-- Solution: Average of frequency of access of objects per bucket
select bucket, total_access, object_count, total_access/object_count as avg_access
from (select bucket, count(*) as total_access, count(distinct bucketkey) as object_count
      from s3index group by bucket)
order by avg_access desc;
```

References

https://blogs.aws.amazon.com/bigdata/post/Tx2YRX3Y16CVQFZ/Building-and-Maintaining-an-Amazon-S3-Metadata-Index-without-Servers

http://docs.aws.amazon.com/redshift/latest/gsg/getting-started.html