**Avoiding Small files creation in Amazon S3 using Kinesis Firehose and process using Amazon EMR running Spark**

In this modern data age, we now have large volumes of data/messages coming from various connected devices and there is a need to efficiently process them. In one of the use cases we store this large volumes of data in Amazon S3 for further downstream processing (for eg: batch processing, nightly aggregation jobs). As most of the times the message's size is very small usually in kilo bytes to couple of megabytes, we will end up creating very small files in Amazon S3 unless we merge them manually before copying into Amazon S3. It will become very difficult to process these small files in the downstream applications. To avoid this we can use Amazon Firehose to merge these messages automatically into larger size files and can use Apache spark to batch process these messages.

Traditionally these connected devices send small messages, but what if your messages you want to send are large? For example a fleet of connected cars, which need to send large messages(typically more than 1 MB) as they need to accumulate the messages on the edge device itself for a brief period of time(every 30 seconds or 1 minute) to avoid several calls to destination.

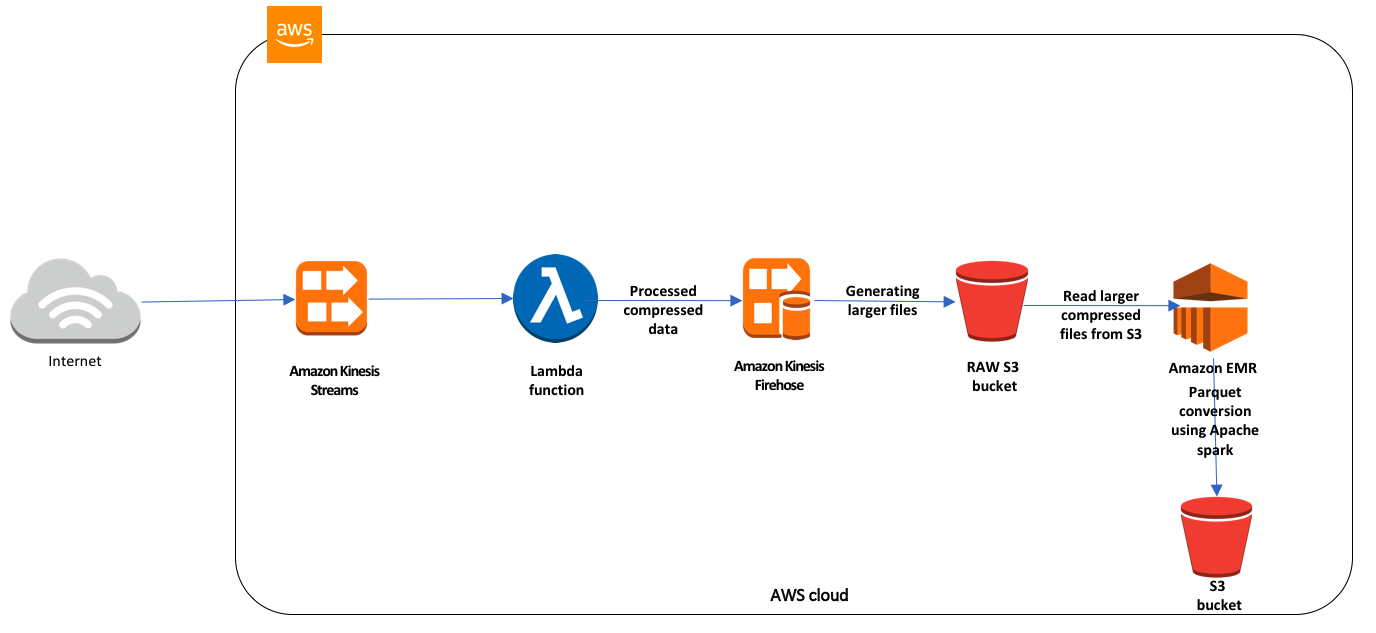
On the other hand, Amazon Kinesis Streams can only accepts messages that are less than 1MB only. We can compress these messages before sending it to Amazon Kinesis Streams to reduce the size of the incoming messages, but it will again result in small compressed files. In this blog we will show you how to send messages to Amaozn Kinesis Streams in compressed format, read them using AWS Lambda to process and add additional information to the message, compress again and send it to Amazon Firehose before sending the messages to Amazon S3. We will also see how to read the compressed files that are in Amazon S3 which does not have a proper file name extension and store back in Amazon S3 in parquet format.

# Solution Overview

At a high level, we do the following in this post:

1. Create a virtual private cloud (VPC) based on the [Amazon VPC](https://aws.amazon.com/vpc/) service and Amazon S3 bucket.
2. Provision Amazon Kinesis Stream, a lambda function to process the messages from Kinesis Stream
3. Provision Kinesis Firehose and an Amazon EMR cluster to process the data was generated from Amazon Kinesis Firehose.
4. Use custom code that was created to generate test data set from the Amazon EC2 instance.
5. Run a sample spark program from EMR cluster’s master instance to read the files from S3 source location and convert them into parquet format and write back to S3 destination location.

The following diagram explains how the services work together:



In this example, we use the Amazon services that are specified in the diagram to send and process streaming data from devices to Kinesis Streams and a lambda function will be invoked to process the data. The lambda function will read the messages and append additional data to the incoming message and compresses the data using GZip compression and then send this to Amazon Firehose (This is just to demonstrate how you can alter/modify the message before storing in Amazon S3). Amazon Firehose will buffer the incoming data and merges them into larger message (up to 128 MB or waits for 15 mins) and stores it into Amazon S3 bucket.

Once the data is stored in Amazon S3, we assume a daily job(a spark program) which reads this data from Amazon S3 and stores this back in S3 destination location in parquet format.

**Challenges with this approach**

The maximum size of a record sent to Kinesis Data Firehose is 1,000KB. If the message size is greater than this value for example couple of MBs, then Kinesis Firehose will not accept this message. To overcome the message size issue, we can compress the message before sending it to Firehose. Since we are already applying the compression on the incoming messages, it does not make sense to enable compression at the Firehose after message was sent. Firehose can effectively merges these messages into larger size upto 128MB file or buffer interval upto 900 seconds and can send the data to Amazon S3.

Since the files that were generated from Kinesis will not have any extension by default and unless we apply any compression at the firehose level, it will have the correct file extension (for eg: .gz). On the other hand Apache spark to read these compressed files, the files should have valid extension to correctly uncompress it. One of the options is to use Amazon lambda function to rename the file that as soon as it was placed by Amazon Kinesis Data Firehose. Since the objects stored in Amazon S3 are immutable, rename operation is actually a copy and delete the old file and it will be an expensive operation based on the input files that will be created in Amazon S3.

We will see how to overcome this issue in later section of the blog.

# Prerequisites and assumptions

To perform this exercise, you need the following:

* An AWS account that provides access to AWS services.
* An [AWS Identity and Access Management (IAM)](https://aws.amazon.com/iam/) user with an access key and secret access key to configure the AWS CLI.
* The templates and the code are intended to work on US-WEST-2 region only. Make sure you are deploying this solution in this region and you have enough resources available.

Additionally, be aware of the following:

* We configure all services in the same VPC to simplify networking considerations.
* **Important**: The [AWS CloudFormation](https://aws.amazon.com/cloudformation/) templates and the sample code that we provide use hard-coded user names and passwords and open security groups. These are just for testing purposes and aren't intended for production use without any modifications.

# Implementing the solution

Following, we describe implementation.

## Single-step solution deployment

You can use this downloadable template for single click deployment. This template is launched in US West (Oregon) region by default. Do not change to a different region because the template is designed to work only in US-WEST-2 region. To launch directly through the console, choose Launch Stack button.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=CF-Root-SMALLFILESBLOG&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/allsteps_cf.template)

This template takes the following parameters. Some of the parameters have default values, and you can't edit these. These predefined names are hardcoded in the code. For some of the parameters, you need to provide the values. The table gives details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| StackName | Provide Stack Name |
| Allowed CIDR | IP address range of the client that will be allowed to connect to the cluster using SSH. |
| FirehoseDeliveryStreamName | Name of the Amazon Firehose delivery stream. Default value is set to “AWSBlogs-LambdaToFireHose”. |
| InstanceType | EC2 Instance type. |
| KeyName | Name of an existing EC2 key pair to enable access to the domain controller instance. |
| Instance type | Instance type for the domain controller EC2 instance. |
| KinesisStreamName | Name of the Amazon Kinesis Stream. Default value is set to “AWS-Blog-BaseKinesisStream” |
| Region | AWS Region – By default it will be “us-west-2”. Do not change this as the scripts are developed to work in this region only. |
| S3BucketName | Name of the bucket that will be created in your account. Provide some unique name to this bucket. This bucket will be used for storing the messages and output from the spark code. |

If you use this one step solution, you can **skip to Step 7**: Generate test data set and load into Kinesis Stream.

**To create each component individually, follow the steps below.**

## 1. Use the AWS CloudFormation template to configure Amazon VPC and create an Amazon S3 Bucket

In this step, we set up a VPC, public subnet, internet gateway, route table and a security group. In addition to these resources, we will also create an Amazon S3 bucket. You can use this **[downloadable cloudformation template](http://)** to set up the above components. To launch directly through the console, choose **Lanch Stack**.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=STEP1_VPC_SETUP&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/step1_vpc.template)

This template takes the following parameters. The below table provies details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| StackName | Provide Stack Name |
| S3Bucketloc | Provide a unique S3 bucket, This bucket will be created in your account. |

After you specify the template details, choose **Next**. On the **options** page, choose **Next** again. On the **Review** page, choose **Create**.

When the stack launch is complete, it should return outputs similar to the following.

|  |  |
| --- | --- |
| **Key** | **Value** |
| StackName | Name |
| VPCID | Vpc-*xxxxxxx* |
| SubnetID | subnet-xxxxxxxx |
| SecurityGroup | sg-xxxxxxxxxx |
| S3BucketDomain | <S3\_BUCKET\_NAME>, .s3.amazonaws.com |
| S3BucketARN | arn:aws:s3:::<S3\_BUCKET\_NAME> |

When the stack launch is complete, it returns the output with information about the resources that were created. Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:  
  
$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-west-2 --query 'Stacks[0].Outputs'

## 2. Use the AWS CloudFormation template to create necessary IAM Roles

In this step, we set up two IAM roles. One of the IAM role will be used for AWS Lambda function and other IAM role will be used for AWS Firehose. You can use this **downloadable cloudformation template** to set up the above components. To launch directly through the console, choose **Lanch Stack**.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=STEP2_IAM_SETUP&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/step2_iam.template)

This template takes the following parameters. The below table provies details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| StackName | Provide Stack Name |

After you specify the template details, choose **Next**. On the **options** page, choose **Next** again. On the **Review** page, choose **Create**.

When the stack launch is complete, it should return outputs similar to the following.

|  |  |
| --- | --- |
| **Key** | **Value** |
| LambdaRoleArn | arn:aws:iam::246997141225:role/small-files-lamdarole |
| FirehoseRoleArn | arn:aws:iam::246997141225:role/small-files-firehoserole |

When the stack launch is complete, it returns the output with information about the resources that were created. Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:  
  
$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-west-2 --query 'Stacks[0].Outputs'

## 3. Use AWS Cloudformation template to configure Amazon Firehose data stream

In this step, we set up an EMR cluster with “Spark”, “Ganglia” and “Hive” applications. You can use this **downloadable cloudformation template** to set up the above components. To launch directly through the console, choose **Lanch Stack**.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=STEP3_FIREHOSE_SETUP&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/step3_firehose.template)

This template takes the following parameters. The below table provies details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| EMRClusterName | Provide name for the EMR cluster |
| ClusterSecurityGroup | Select the security group id that was created as part of the first cloudformation template. |
| ClusterSubnetID | Select the subnet id that was created as part of the first cloudformation template. |
| AllowedCIDR | IP address range of the client that will be allowed to connect to the cluster. |
| KeyName | Name of an existing EC2 Key pair to acccess the Amazon EMR cluster |

After you specify the template details, choose **Next**. On the **options** page, choose **Next** again. On the **Review** page, choose **Create**.

## 4. Use the AWS Cloudformation template to create Amazon Kinesis Stream and a Lambda Function

In this step, we set up Amazon Kinesis data stream and a Lambda function that will be invoked whenever there are messages in the stream. An event source mapping will also be created as part of this template. This will add a trigger to Lambda function for the Kinesis Stream source. You can use this **downloadable cloudformation template** to set up the above components. To launch directly through the console, choose **Lanch Stack**.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=STEP4_KINSESISSTREAM_SETUP&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/step4_kinesisstream.template)

This template takes the following parameters. The below table provies details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| StackName | Provide Stack Name |
| Role | Provide a name for IAM Role |
| SecurityGroupIds | Select the security group Id that was created from the 1st Cloudformation template. |
| SubnetIds | Select the Subnet Id that was created from the 1st Cloudformation template. |
| S3Bucket | Select Amazon S3 bucket name that was created after the 1st Cloudformation template is created. |
| S3Directory | Provide the subfolder/s3key in the provided S3 bucket. |
| LambdaMemory | Amount of memory that will be allocated for the AWS Lambda function |
| KinesisShardCount | Number of shards that will be assigned to Kinesis stream |
| KinesisPeriodHours | Number of hours that the message will stay in the Kinesis Stream |
| Region | Select the AWS region. By default it will be us-east-1. |

After you specify the template details, choose **Next**. On the **options** page, choose **Next** again. On the **Review** page, choose **Create**.

## 5. Use AWS Cloudformation template to configure Amazon EMR cluster

In this step, we set up two IAM roles. One of the IAM role will be used for AWS Lambda function and other IAM role will be used for AWS Firehose. You can use this **downloadable cloudformation template** to set up the above components. To launch directly through the console, choose **Lanch Stack**.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=STEP5_EC2_INSTANCE_SETUP&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/step5_emr.template)

This template takes the following parameters. The below table provies details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| StackName | Provide Stack Name |
| S3BucketARN | Select S3BucketARN. You can get this from the step1 cloudformation output. |

After you specify the template details, choose **Next**. On the **options** page, choose **Next** again. On the **Review** page, choose **Create**.

When the stack launch is complete, it should return outputs similar to the following.

|  |  |
| --- | --- |
| **Key** | **Value** |
| EMRClusterMaster | ssh hadoop@ec2-XX-XXX-XXX-XXX.us-west-2.compute.amazonaws.com -i <KEY\_PAIR\_NAME>.pem |

When the stack launch is complete, it returns the output with information about the resources that were created. Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:  
  
$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-west-2 --query 'Stacks[0].Outputs'

## 6. Use the AWS Cloudformation template to create Amazon EC2 Instance to generate test data

In this step, we set up Amazon EC2 isntance and will install open-jdk version 1.8. Once the instance is up and running we will run a Java program which will be already copied on to the ec2 instance’s ec2-user home directory. The file name is : “*sample-kinesis-producer-1.0-SNAPSHOT-jar-with-dependencies.jar*”. When you run this java program it will generate test data set with ~900KB message each and will be pushed to Amazon Kinesis Stream.

You can use this **downloadable cloudformation template** to set up the above components. To launch directly through the console, choose **Launch Stack**.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-west-2#/stacks/new?stackName=STEP5_EC2_INSTANCE_SETUP&templateURL=https://s3-us-west-2.amazonaws.com/skkodali-proserv-us-west-2/avoid-small-files-blog/cloudformation-templates/step6_ec2_instance.template)

This template takes the following parameters. The below table provies details.

|  |  |
| --- | --- |
| **For this parameter** | **Use this** |
| EC2SecurityGroup | Select the security group Id that was created from the 1st Cloudformation template. |
| EC2Subnet | Select the Subnet that was created from the 1st Cloudformation template. |
| InstanceType | Select the provided instance type. By default it will select r4.4xlarge instance. |
| KeyName | Name of an existing EC2 Key Pair to enable SSH access to the EC2 instance. |

After you specify the template details, choose **Next**. On the **options** page, choose **Next** again. On the **Review** page, choose **Create**.

When the stack launch is complete, it should return outputs similar to the following.

|  |  |
| --- | --- |
| **Key** | **Value** |
| EC2Instance | ssh ec2-user@<Public-IP> -i <KEY\_PAIR\_NAME>.pem |

When the stack launch is complete, it returns the output with information about the resources that were created. Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:  
  
$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-west-2 --query 'Stacks[0].Outputs'

7. Generate test data set and load into Kinesis Stream

Once the cloudformation stack is complete, login to the EC2 instance that was created as part of the Step 6, using the “ssh” command as shown in the CF template output. This template will copy the “*sample-kinesis-producer-1.0-SNAPSHOT-jar-with-dependencies.jar*” jar file which will be used to generate the test data and sends to Kinesis Stream. You can find the code corresponding to this sample kinesis producer in this git repo.

Login using the below ssh command to the EC2 instance that was created in Step 6. Make sure your EC2 instance’s security group allows ssh port 22 (Inbound) from your IP address.

* ssh ec2-user@<Public IP Address of the EC2 Instance> -i <SSH\_KEY\_PAIR\_NAME>.pem

Run the below commands to generate some test data.

*$ cd;*

*$ ls -ltra sample-kinesis-producer-1.0-SNAPSHOT-jar-with-dependencies.jar*

*-rwxr-xr-x 1 ec2-user ec2-user 27536802 Oct 29 21:19 sample-kinesis-producer-1.0-SNAPSHOT-jar-with-dependencies.jar*

*$java -Xms1024m -Xmx25600m -XX:+UseG1GC -cp sample-kinesis-producer-1.0-SNAPSHOT-jar-with-dependencies.jar com.awsblogs.smallfiles.entry.Main 10000 10*

The first parameter for this program tells how many messages that needs to be generated. In this case the number of messages is 10,000.

The second parameter for this program tells how many threads will be writing data into Kinesis Streams.

The sample JSON input message format is as below.

{    
   **"processedDate"**:"2018/10/30 19:05:19",  
   **"currentDate"**:"2018/10/30 19:05:07",  
   **"hashDeviceId"**:"0c2745e4-c2d6-4d43-8339-9c2401e80e92",  
   **"deviceId"**:"94581b5f-a117-484a-8e3c-4fcc2dbd53b7",  
   **"accelerometerSensorList"**:[    
      {    
         **"accelerometer\_Y"**:8,  
         **"gravitySensor\_X"**:5,  
         **"accelerometer\_X"**:9,  
         **"gravitySensor\_Z"**:4,  
         **"accelerometer\_Z"**:1,  
         **"gravitySensor\_Y"**:5,  
         **"linearAccelerationSensor\_Z"**:3,  
         **"linearAccelerationSensor\_Y"**:9,  
         **"linearAccelerationSensor\_X"**:9  
      },  
      {    
         **"accelerometer\_Y"**:1,  
         **"gravitySensor\_X"**:3,  
         **"accelerometer\_X"**:5,  
         **"gravitySensor\_Z"**:5,  
         **"accelerometer\_Z"**:7,  
         **"gravitySensor\_Y"**:9,  
         **"linearAccelerationSensor\_Z"**:6,  
         **"linearAccelerationSensor\_Y"**:5,  
         **"linearAccelerationSensor\_X"**:3  
      },

{

…

},

{

…

},

:

:  
   ],  
   **"tempSensorList"**:[    
      {    
         **"kelvin"**:585.4928040286752,  
         **"celsius"**:43.329574923775425,  
         **"fahrenheit"**:50.13864584530086  
      },  
      {    
         **"kelvin"**:349.95625855125814,  
         **"celsius"**:95.68423052685313,  
         **"fahrenheit"**:7.854854574219985  
      },

{

…

},

{

…

},

:

:

   ],  
   **"illuminancesSensorList"**:[    
      {    
         **"illuminance"**:44.65135784368194  
      },  
      {    
         **"illuminance"**:98.15404017082403  
      },

{

…

},

{

…

},

:

:  
   ],  
   **"gpsSensorList"**:[    
      {    
         **"altitude"**:4.38273213294682,  
         **"heading"**:7.416314616289915,  
         **"latitude"**:5.759723677991661,  
         **"longitude"**:1.4732885894731842  
      },  
      {    
         **"altitude"**:9.816473807569487,  
         **"heading"**:5.118919157684835,  
         **"latitude"**:3.581361614110458,  
         **"longitude"**:1.3699272610616127  
      },

{

…

},

{

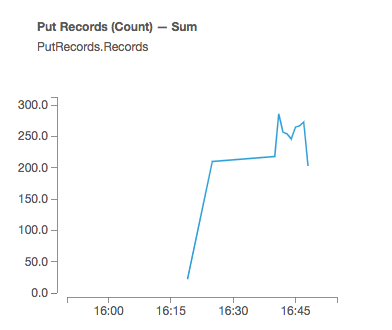
…

},

:

:  
   }

Login to Kinesis Stream UI, select the Kinesis Stream and click on the “**Monitor**” tab. You will see the graphs. Pump the data generation for atleast **15 mins** to generate enough data set.



## 8. Processing Kinesis Stream messages using AWS Lambda

As part of the setup described above , we are also creating a lambda function (name:LambdaForProcessingKinesisRecords) to process the messages from this Kinesis Stream. This Lambda function will unzip each message content, appends “additional data”(*This is just to show that the message coming from Kinesis will be processed and appends additional information to make the message size more than 1 MB*) to this incoming message and compresses it using GZIP and sends to Amazon Kinesis Firehose. As we already know each message size is more than 1 MB after uncompressing it and adding the “*additional data*” to the uncompressed message, the size will be increased. Even Amazon Kinesis Firehose can only accepts 1 MB as input size, we will be compressing it in the lambda function by using GZIP. Also we will append a “new line character” (“\n”) to each message as part of this lambda function. Note that we have set the buffer size to 128 MB and duration of the buffer is 900 seconds while creating the Kinesis Firehose. This will help to merge the incoming compressed messages into larger messages and will push it into provided Amazon S3 bucket.

The lambda function will append the below content to the original message in Kinesis Streams after reading it.

**"testAdditonalDataList"**: [  
 {  
 **"dimesnion\_X"**: 9,  
 **"dimesnion\_Y"**: 2,  
 **"dimesnion\_Z"**: 2  
 },  
 {  
 **"dimesnion\_X"**: 3,  
 **"dimesnion\_Y"**: 10,  
 **"dimesnion\_Z"**: 5  
 }  
 {  
 …  
 },  
 {  
 …  
 },  
 :  
 :  
]

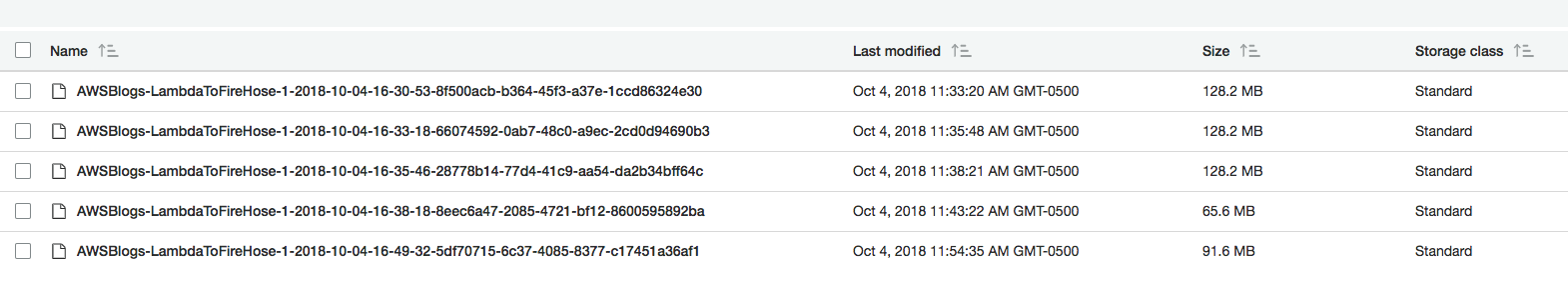
If we do not compress the message before sending to Amazon Kinesis Firehose, it will throw this error message in the cloudwatch logs.



Here is the code snippet where we are compressing the message in the AWS Lambda function. The complete code can be found in this git repository.

**private** String sendToFireHose(String mergedJsonString)  
{  
 PutRecordResult res = **null**;  
 **try** {  
 *//To Firehose -* System.*out*.println(**"MESSAGE SIZE BEFORE COMPRESSION IS : "** + mergedJsonString.toString().getBytes(*charset*).length);  
 System.*out*.println(**"MESSAGE SIZE AFTER GZIP COMPRESSION IS : "** + *compressMessage*(mergedJsonString.toString().getBytes(*charset*)).length);  
 PutRecordRequest req = **new** PutRecordRequest()  
 .withDeliveryStreamName(firehoseStreamName);  
  
 *// Without compression - Send to Firehose  
 //Record record = new Record().withData(ByteBuffer.wrap((mergedJsonString.toString() + "\r\n").getBytes()));  
  
 // With compression - send to Firehose* Record record = **new** Record().withData(ByteBuffer.*wrap*(*compressMessage*((mergedJsonString.toString() + **"\r\n"**).getBytes())));  
 req.setRecord(record);  
 res = kinesisFirehoseClient.putRecord(req);  
 }  
 **catch** (IOException ie) {  
 ie.printStackTrace();  
 }  
 **return** res.getRecordId();  
}

You can check the provided bucket to see if the messages are generating into the bucket. It will show something like below diagram.



As you note down from the above screen shot, you will see files generated from the Kinesis Firehose will not have any extension. By default Kinesis Firehose will not provide any extension to the files that are generated in Amazon S3 bucket unless you select a compression option. But in our use case, since the size of the *uncompressed* input message size is greater than 1 MB, we are compressing it before sending to Amazon Firehose delivery stream. As the message is already compressed, we are not selecting any compression option in Firehose as it will double compress the message and downstream spark application cannot process this. To avoid this double compression, we are not selecting the “compression” option in Amazon Firehose delivery stream.

## 9. Reading and converting the data into parquet format using Apache Spark program with Amazon EMR

As we noted down from the above screen shot, Kinesis Firehose by default will not generate any file extensions to the files that are written into Amazon S3 bucket. This will create a problem while reading the files using Apache Spark. Apache Spark on other hand by default will check for valid extension to the file name. In this case for GZIP compression, it looks for <filename>.gz as a valid file name to successfully uncompress it. If there is no valid file extension, sprk job will not complete successfully.

To overcome this issue, we can use S3’s AmazonS3Client class to list all the keys and use spark’s parallelize method to get the contents of the files and apply the uncopmpression using GZipInputStream class. The code snippet is below. The complete code can be found in the git repository.

**Scala code snippet:**

**val** allLinesRDD = spark.sparkContext.parallelize(s3ObjectKeys).flatMap  
{ key => Source.*fromInputStream*

(

**new** GZipInputStream(s3Client.getObject(*bucketName*, key).getObjectContent: InputStream)

).getLines

}  
  
**var** finalDF = spark.read.json(allLinesRDD).toDF()

Once the EMR cluster creation is completed successfully, execute the spark program using below spark submit command. The git hub repository for this is located at <PROVIDE GITHUB LINK>

*spark-submit --class com.aws.convert.parquet.ConvertToParquet*

*--master yarn --deploy-mode cluster --num-executors 10 --executor-cores 3 --executor-memory 10g --driver-memory 20g --driver-cores 5 --conf spark.task.cpus=4 s3://<BUCKETT>ConvertToParquet-0.0.1-SNAPSHOT.jar FROMNEWFIREHOSE/2018/08/ s3://<BUCKET>/output/*

|  |  |  |
| --- | --- | --- |
| **Argument #** | **Property** | **Value** |
| 1 | --class | com.aws.convert.parquet.ConvertToParquet |
| 2 | --master | yarn |
| 3 | --deploy-mode | cluster |
| 4 | --num-executors | 10 |
| 5 | --executor-cores | 3 |
| 6 | --executor-memory | 10g |
| 7 | --driver-memrory | 20g |
| 8 | --driver-cores | 5 |
| 9 | --conf spark.task.cpus | 4 |
| 10 |  | S3://<>/ConvertToParquet-0.0.1-SNAPSHOT.jar |
| 11 | <INPUT S3 LOCATION> | S3://<BUCKET>/FROMFIREHOSE/<year>/<mm>/ |
| 12 | <OUTPUT S3 LOCATION> | S3://<BUCKET>/output/ |

Once the program execution is complete, you can check the S3 output location to see files are written in parquet format.

# Cleaning up after the migration

After completing and testing this solution, clean up the resources by stopping your tasks and deleting the CloudFormation stacks:

# Conclusion

In this blog, we went through the process of avoiding small files creation in AWS S3 by passing the messages to Amazon Kinesis Firehose. We also went through the process of reading and stroing the data in parquet format using Apache spark with EMR cluster when there is no file extension to the input files that are generated by AWS Firehose.