Best Practices for Amazon EMR

Parviz Deyhim

August 2013

(Please consult <http://aws.amazon.com/whitepapers/> for the latest version of this paper)

Table of Contents

Abstract

Amazon Web Services (AWS) cloud accelerates big data analytics. It provides instant scalability and elasticity, letting you focus on analytics instead of infrastructure. Whether you are indexing large data sets or analyzing massive amounts of scientific data or processing clickstream logs, AWS provides a range of big data tools and services that you can leverage for virtually any data-intensive project.

Amazon Elastic MapReduce (EMR) is one such service that provides fully managed hosted Hadoop framework on top of Amazon Elastic Compute Cloud (EC2). In this paper, we highlight the best practices of moving data to AWS, collecting and aggregating the data, and discuss common architectural patterns for setting up and configuring Amazon EMR clusters for faster processing. We also discuss several performance and cost optimization techniques so you can process and analyze massive amounts of data at high throughput and low cost in a reliable manner.

Introduction

Big data is all about collecting, storing, processing, and visualizing massive amounts of data so that companies can distill knowledge from it, derive valuable business insights from that knowledge, and make better business decisions, all as quickly as possible. The main challenges in operating data analysis platforms include installation and operational management, dynamically allocating data processing capacity to accommodate for variable load, and aggregating data from multiple sources for holistic analysis. The Open Source Apache Hadoop and its ecosystem of tools help solve these problems because Hadoop can expand horizontally to accommodate growing data volume and can process unstructured and structured data in the same environment.

Amazon Elastic MapReduce (Amazon EMR) simplifies running Hadoop and related big data applications on AWS. It removes the cost and complexity of managing the Hadoop installation. This means any developer or business has the power to do analytics without large capital expenditures. Today, you can spin up a performance-optimized Hadoop cluster in the AWS cloud within minutes on the latest high performance computing hardware and network without making a capital investment to purchase the hardware. You have the ability to expand and shrink a running cluster on demand. This means if you need answers to your questions faster, you can immediately scale up the size of your cluster to crunch the data more quickly. You can analyze and process vast amounts of data by using Hadoop's MapReduce architecture to distribute the computational work across a cluster of virtual servers running in the AWS cloud.

In addition to processing, analyzing massive amounts of data also involves data collection, migration, and optimization.

Figure 1: Data Flow

This whitepaper explains the best practices of moving data to AWS; strategies for collecting, compressing, aggregating the data; and common architectural patterns for setting up and configuring Amazon EMR clusters for processing. It also provides examples for optimizing for cost and leverage a variety of Amazon EC2 purchase options such as Reserved and Spot Instances. This paper assumes you have a conceptual understanding and some experience with Amazon EMR and

Apache Hadoop. For an introduction to Amazon EMR, see the Amazon EMR Developer Guide.[1] For an introduction to Hadoop, see the book Hadoop: The Definitive Guide.[2]

Moving Data to AWS

A number of approaches are available for moving large amounts of data from your current storage to Amazon Simple Storage Service (Amazon S3) or from Amazon S3 to Amazon EMR and the Hadoop Distributed File System (HDFS). When doing so, however, it is critical to use the available data bandwidth strategically. With the proper optimizations, uploads of several terabytes a day may be possible. To achieve such high throughput, you can upload data into AWS in parallel from multiple clients, each using multithreading to provide concurrent uploads or employing multipart uploads for further parallelization. You can adjust TCP settings such as window scaling[3] and selective acknowledgement[4] to enhance data throughput further. The following scenarios explain three ways to optimize data migration from your current local storage location (data center) to AWS by fully utilizing your available throughput.

Scenario 1: Moving Large Amounts of Data from HDFS (Data Center) to Amazon S3

Two tools—S3DistCp and DistCp—can help you move data stored on your local (data center) HDFS storage to Amazon S3. Amazon S3 is a great permanent storage option for unstructured data files because of its high durability and enterprise class features, such as security and lifecycle management.

Using S3DistCp

S3DistCp is an extension of DistCp with optimizations to work with AWS, particularly Amazon S3. By adding S3DistCp as a step in a job flow, you can efficiently copy large amounts of data from Amazon S3 into HDFS where subsequent steps in your EMR clusters can process it. You can also use S3DistCp to copy data between Amazon S3 buckets or from HDFS to Amazon S3.

S3DistCp copies data using distributed map–reduce jobs, which is similar to DistCp. S3DistCp runs mappers to compile a list of files to copy to the destination. Once mappers finish compiling a list of files, the reducers perform the actual data copy. The main optimization that S3DistCp provides over DistCp is by having a reducer run multiple HTTP upload threads to upload the files in parallel.

To illustrate the advantage of using S3DistCP, we conducted a side-by-side comparison between S3DistCp and DistCp. In this test, we copy 50 GB of data from a Hadoop cluster running on Amazon Elastic Compute Cloud (EC2) in Virginia and copy the data to an Amazon S3 bucket in Oregon. This test provides an indication of the performance difference between S3DistCp and DistCp under certain circumstances, but your results may vary.

Method	Data Size Copied	Total Time
DistCp	50 GB	26 min
S3DistCp	50 GB	19 Min

Figure 2: DistCp and S3DistCp Performance Compared

[1] http://aws.amazon.com/documentation/elasticmapreduce/
[2] http://www.amazon.com/Hadoop-Definitive-Guide-Tom-White/dp/1449311520/
[3] http://docs.aws.amazon.com/AmazonS3/latest/dev/TCPWindowScaling.html
[4] http://docs.aws.amazon.com/AmazonS3/latest/dev/TCPSelectiveAcknowledgement.html

To copy data from your Hadoop cluster to Amazon S3 using S3DistCp

The following is an example of how to run S3DistCp on your own Hadoop installation to copy data from HDFS to Amazon S3. We've tested the following steps with: 1) Apache Hadoop 1.0.3 distribution 2) Amazon EMR AMI 2.4.1. We've not tested this process with the other Hadoop distributions and cannot guarantee that the exact same steps works beyond the Hadoop distribution mentioned here (Apache Hadoop 1.0.3).

1. Launch a small Amazon EMR cluster (a single node).

   ```
   elastic-mapreduce --create --alive --instance-count 1 --instance-type m1.small --
   ami-version 2.4.1
   ```

2. Copy the following jars from Amazon EMR's master node (/home/Hadoop/lib) to your local Hadoop master node under the /lib directory of your Hadoop installation path (For example: /usr/local/hadoop/lib). Depending on your Hadoop installation, you may or may not have these jars. The Apache Hadoop distribution does not contain these jars.

   ```
   /home/hadoop/lib/emr-s3distcp-1.0.jar
   /home/hadoop/lib/aws-java-sdk-1.3.26.jar
   /home/hadoop/lib/guava-13.0.1.jar
   /home/hadoop/lib/gson-2.1.jar
   /home/hadoop/lib/EmrMetrics-1.0.jar
   /home/hadoop/lib/protobuf-java-2.4.1.jar
   /home/hadoop/lib/httpcore-4.1.jar
   /home/hadoop/lib/httpclient-4.1.1.jar
   ```

3. Edit the `core-site.xml` file to insert your AWS credentials. Then copy the `core-site.xml` config file to all of your Hadoop cluster nodes. After copying the file, it is unnecessary to restart any services or daemons for the change to take effect.

   ```
   <property>
     <name>fs.s3.awsSecretAccessKey</name>
     <value>YOUR_SECRETACCESSKEY</value>
   </property>

   <property>
     <name>fs.s3.awsAccessKeyId</name>
     <value>YOUR_ACCESSKEY</value>
   </property>
   <property>
     <name>fs.s3n.awsSecretAccessKey</name>
     <value>YOUR_SECRETACCESSKEY</value>
   </property>

   <property>
     <name>fs.s3n.awsAccessKeyId</name>
     <value>YOUR_ACCESSKEY</value>
   </property>
   ```

4. Run s3distcp using the following example (modify HDFS_PATH, YOUR_S3_BUCKET and PATH):

   ```
   hadoop jar /usr/local/hadoop/lib/emr-s3distcp-1.0.jar -libjars
   /usr/local/hadoop/lib/gson-2.1.jar,/usr/local/hadoop/lib/guava-
   13.0.1.jar,/usr/local/hadoop/lib/aws-java-sdk-1.3.26.jar,/usr/local/hadoop/lib/emr-
   ```

```
s3distcp-1.0.jar,/usr/local/hadoop/lib/EmrMetrics-
1.0.jar,/usr/local/hadoop/lib/protobuf-java-
2.4.1.jar,/usr/local/hadoop/lib/httpcore-4.1.jar,/usr/local/hadoop/lib/httpclient-
4.1.1.jar --src HDFS_PATH --dest s3://YOUR_S3_BUCKET/PATH/ --disableMultipartUpload
```

Using DistCp

DistCp (distributed copy) is a tool used for large inter- or intra-cluster copying of data. It uses Amazon EMR to effect its distribution, error handling, and recovery, as well as reporting. It expands a list of files and directories into input to map tasks, each of which will copy a partition of the files specified in the source list.

DistCp can copy data from HDFS to Amazon S3 in a distributed manner similar to S3DistCp; however, DistCp is not as fast. DistCp uses the following algorithm to compute the number of mappers required:

*min (total_bytes / bytes.per.map, 20 * num_task_trackers)*

Usually, this formula works well, but occasionally it may not compute the right amount of mappers. If you are using DistCp and notice that the number of mappers used to copy your data is less than your cluster's total mapper capacity, you may want to increase the number of mappers that DistCp uses to copy files by specifying the `-m number_of_mappers` option.

The following is an example of DistCp command copying /data directory on HDFS to a given Amazon S3 bucket:

```
hadoop distcp hdfs:///data/ s3n://awsaccesskey:awssecrectkey@somebucket/mydata/
```

For more details and tutorials on working with DistCp, see http://hadoop.apache.org/docs/r0.19.2/distcp.html.

Scenario 2: Moving Large Amounts of Data from Local Disk (non-HDFS) to Amazon S3

Scenario 1 explained how to use distributed copy tools (DistCp and S3DistCp) to help you copy your data to AWS in parallel. The parallelism achieved in Scenario 1 was possible because the data was stored on multiple HDFS nodes and multiple nodes can copy data simultaneously. Fortunately, you have several ways to move data efficiently when you are not using HDFS.

Using the Jets3t Java Library

JetS3t is an open-source Java toolkit for developers to create powerful yet simple applications to interact with Amazon S3 or Amazon CloudFront. JetS3t provides low-level APIs but also comes with tools that let you work with Amazon S3 or Amazon CloudFront without writing Java applications.

One of the tools provided in the JetS3t toolkit is an application called Synchronize. Synchronize is a command-line application for synchronizing directories on your computer with an Amazon S3 bucket. It is ideal for performing backups or synchronizing files between different computers.

One of the benefits of Synchronize is configuration flexibility. Synchronize can be configured to open as many upload threads as possible. With this flexibility, you can saturate the available bandwidth and take full advantage of your available throughput.

To set up Synchronize

1. Download JetS3Tt from the following URL: http://jets3t.s3.amazonaws.com/downloads.html.

2. Unzip `jets3t`.

3. Create a `synchronize.properties` file and add the following parameters, replacing the values for `accesskey` and `secretkey` with your AWS access key identifiers:

```
accesskey=xxx
secretkey=yyy
upload.transformed-files-batch-size=100
httpclient.max-connections=100
storage-service.admin-max-thread-count=100
storage-service.max-thread-count=10
threaded-service.max-thread-count=15
```

4. Run Synchronize using the following command line example:

```
bin/synchronize.sh -k UP somes3bucket/data /data/ --properties
synchronize.properties
```

Using GNU Parallel

GNU parallel is a shell tool that lets you use one or more computers to execute jobs in parallel. GNU parallel runs jobs, which can be a single command or a small script to run for each of the lines in the input. Using GNU parallel, you can parallelize the process of uploading multiple files by opening multiple threads simultaneously. In general, you should open as many parallel upload threads as possible to use most of the available bandwidth.

The following is an example of how you can use GNU parallel:

1. Create a list of files that you need to upload to Amazon S3 with their current full path

2. Run GNU parallel with any Amazon S3 upload/download tool and with as many thread as possible using the following command line example:

```
ls | parallel -j0 -N2  s3cmd put {1} s3://somes3bucket/dir1/
```

The previous example copies the content of the current directly (`ls`) and runs GNU parallel with two parallel threads (`-N2`) to Amazon S3 by running the `s3cmd` command.

Using Aspera Direct-to-S3

The file transfer protocols discussed in this document use TCP; however, TCP is suboptimal with high latency paths. In these circumstances, UDP provides the potential for higher speeds and better performance.

Aspera has developed a proprietary file transfer protocol based on UDP, which provides a high-speed file transfer experience over the Internet. One of the products offered by Aspera, called Direct-to-S3, offers UDP-based file transfer protocol that would transfer large amount of data with fast speed directly to Amazon S3. If you have a large amount of data stored in your local data center and would like to move your data to Amazon S3 for later processing on AWS (Amazon EMR for example), Aspera Direct-To-S3 can help move your data to Amazon S3 faster compared to other protocols such as HTTP, FTP, SSH, or any TCP-based protocol.

For more information about Aspera cloud-based products, see Aspera at http://cloud.asperasoft.com/big-data-cloud/.

Using AWS Import/Export

AWS Import/Export accelerates moving large amounts of data into and out of AWS using portable storage devices for transport. AWS transfers your data directly to and from storage devices using Amazon's high-speed internal network and bypassing the Internet. For significant data sets, AWS Import/Export is often faster than using an Internet-based data transfer and can be more cost effective than upgrading your connectivity.

To use AWS Import/Export

1. Prepare a portable storage device from the list of supported devices. For more information, see Selecting Your Storage Device, http://aws.amazon.com/importexport/#supported_devices.

2. Submit a Create Job request to AWS that includes your Amazon S3 bucket, Amazon Elastic Block Store (EBS), or Amazon Glacier region, AWS access key ID, and return shipping address. You will receive back a unique identifier for the job, a digital signature for authenticating your device, and an AWS address to which to ship your storage device.

3. Securely identify and authenticate your device. For Amazon S3, place the signature file on the root directory of your device. For Amazon EBS or Amazon Glacier, tape the signature barcode to the exterior of the device.

4. Ship your device along with its interface connectors, and power supply to AWS.

When your package arrives, it will be processed and securely transferred to an AWS data center and attached to an AWS Import/Export station. After the data load is completed, AWS returns the device to you.

One of the common ways you can take advantage AWS Import/Export is to use this service as the initial data transfer and bulk data upload to AWS. Once that data import has been completed you can incrementally add data to the previously uploaded data using data collection and aggregation frameworks discussed later in this document.

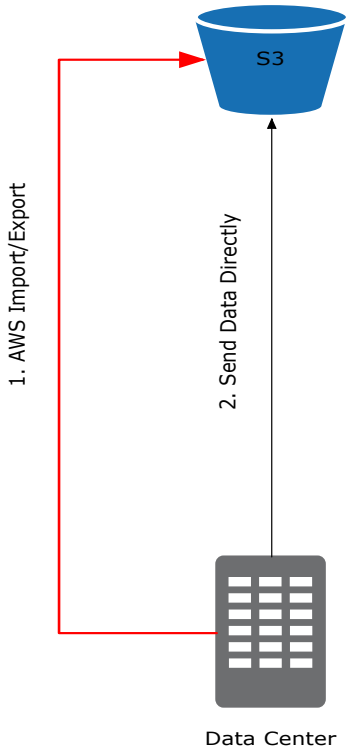

Figure 3: Moving Data to AWS Using AWS Import/Export

Using AWS Direct Connect

AWS Direct Connect makes it easy to establish a dedicated network connection from your premises to AWS. Using AWS Direct Connect, you can establish private connectivity between AWS and your data center, office, or colocation environment, which in many cases can reduce your network costs, increase bandwidth throughput, and provide a more consistent network experience than Internet-based connections.

AWS Direct Connect lets you establish a dedicated network connection between your network and one of the AWS Direct Connect locations. Using industry standard 802.1q VLANs, this dedicated connection can be partitioned into multiple virtual interfaces. This lets you use the same connection to access public resources, such as objects stored in Amazon S3 using public IP address space, and private resources, such as Amazon EC2 instances running within an Amazon Virtual Private Cloud (VPC) using private IP space, while maintaining network separation between the public and private environments. You can reconfigure virtual interfaces at any time to meet your changing needs.

When using AWS Direct Connect to process data on AWS, two architecture patterns are the most common:

1. **One-time bulk data transfer to AWS.** Once the majority of the data has been transferred to AWS, you can terminate your Direct Connect line and start using the methods discussed in the "Data Collection and Aggregation" section to continually update your previously migrated data on AWS. This approach lets you control your costs and only pay for the direct-connect link for the duration of data upload to AWS.

2. **Use AWS Direct Connect to connect your data center with AWS resources.** Once connected, you can use Amazon EMR to process your data stored in your own data center and store the results on AWS or back in your data center. This approach gives you 1 or 10 gigabit-per-second link connectivity to AWS at all time. And direct-connect outbound bandwidth costs less than public Internet outbound cost. So in cases where you expect great amount of traffic exported to your own data center, having direct connect in place can reduce your bandwidth charges.

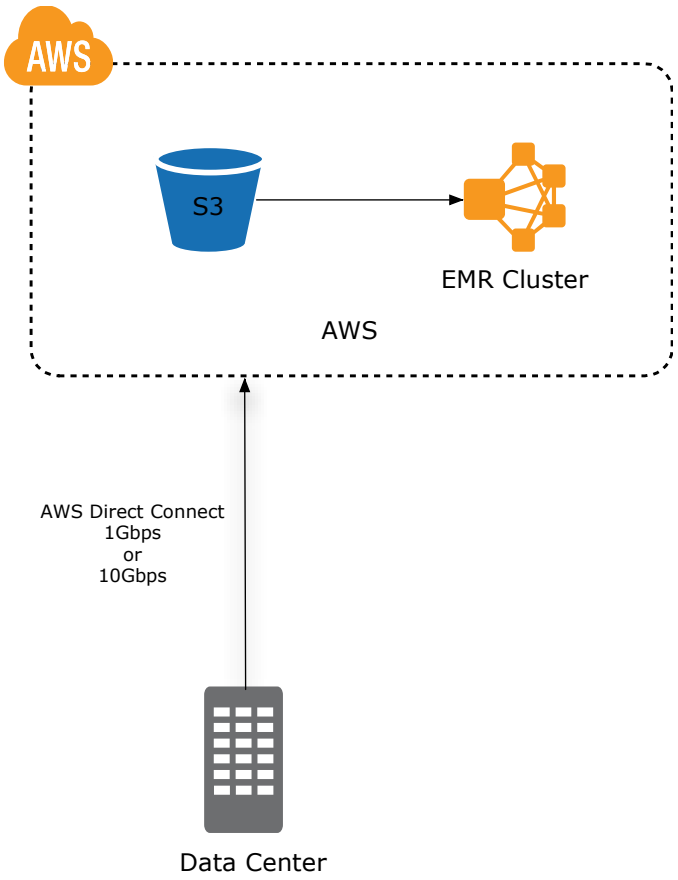

Figure 4: Moving Data to AWS Using Amazon Direct Connect

Scenario 3: Moving Large Amounts of Data from Amazon S3 to HDFS

In addition to moving data to AWS (Amazon S3 or Amazon EC2), there are cases where you need to move your data to your instances (e.g., to HDFS) from Amazon S3. We explain the details of this use case later in this document, but let us briefly cover two techniques for moving data to Amazon EC2. These techniques focus on moving data to HDFS.

Using S3DistCp

As you saw earlier, S3DistCp lets you copy large amounts of data from your data center HDFS storage to Amazon S3. But you can also use the same tool and a similar process to move data from Amazon S3 to local HDFS. If you use Amazon EMR and want to copy data to HDFS, simply run S3DistCp using the Amazon EMR command line interface (CLI) (http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-install.html). The following example demonstrates how to do this with S3DistCP:

```
elastic-mapreduce -j JOFLOWID --jar  /home/hadoop/lib/emr-s3distcp-1.0.jar
--step-name "Moving data from S3 to HDFS"
--args "--src,s3://somebucket/somedir/,--dest,hdfs:///someHDFSdir/"
```

Note: *elastic-mapreduce* is an Amazon EMR ruby client that you can download from http://aws.amazon.com/developertools/Elastic-MapReduce/2264. The above example, copies the data from `s3://somebucket/somedir/` to `hdfs:///somehdfsdir/`.

Using DistCp

You can also use DistCp to move data from Amazon S3 to HDFS. The following command-line example demonstrates how to use DistCp to move data from Amazon S3 to HDFS:

```
hadoop s3n://awsaccesskey:awssecrectkey@somebucket/mydata/ distcp hdfs:///data/
```

Since S3DistCP is optimized to move data from and to Amazon S3, we generally recommend using S3DistCp to improve your data transfer throughput.

Data Collection

One of the common challenges of processing large amount of data in Hadoop is moving data from the origin to the final collection point. In the context of the cloud where applications and compute resources generate massive amounts of data in a short period of time, collecting data in a scalable and reliable manner has an important place in big-data architecture.

Traditionally, the data generated by various parts of your architecture was stored in text files and shipped to the destination by tools such as RYSNC, SSH, FTP, or other third-party tools. It is also common for developers to write their own log-collection tools with PHP, Ruby, or other programming languages. While such attempts to write custom data collectors are important, AWS users can leverage frameworks that have already been written to provide scalable and efficient distributed data collection.

Today a variety of different applications and frameworks can help you move data from the source to a destination in a very scalable and reliable way. Using the tools mentioned here can help you load data to AWS without building your own log collections and frameworks.

Using Apache Flume

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows. It is robust and fault tolerant with tunable reliability mechanisms and many failover and recovery mechanisms.

Apache Flume has concepts such as agents, collectors, and the master. Agents are responsible for collecting data such as web logs, API logs, syslogs, or any other periodically published data. You can install Flume agents on the data source (web-servers, ad servers, or any other source that generates data) and ship data to the collectors. The responsibility of the collector is to collect the data from the agents and store them in permanent storage such as Amazon S3 or HDFS.

One of the main benefits of Flume is its simple but yet powerful hierarchical data collection model that lets you set up more agent nodes to collect data so you need fewer collectors.

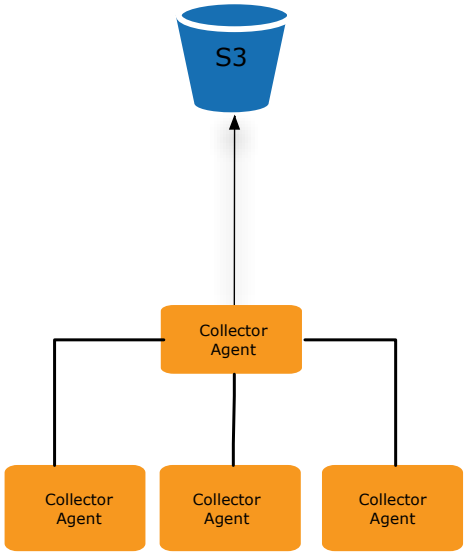

Figure 5: Data Collection Using Apache Flume

Using Fluentd

The architecture of Fluentd (Sponsored by Treasure Data) is very similar to Apache Flume or Facebook's Scribe. Fluentd is easier to install and maintain and has better documentation and support than Flume and Scribe.

Fluentd consist of three basic components:

- **Input** refers to the location where Fluentd reads logs as input. Input is extensible to support log upload formats from various sources. The officially supported input formats are HTML, JSON, tailing files, and syslog.
- **Buffer** refers to the input memory buffer. Fluentd buffers logs in memory if the output step fails.
- **Output** refers to the storage location for output data. When Fluentd passes data from buffer to output, the output writes to persistent storage. The community supports the following outputs: Amazon S3, Amazon SQS, MongoDB, Redis, and more.

Similar to Flume, Fluentd supports a hierarchical collection architecture where a set of Fluentd nodes collect and forward data to another set of Fluentd nodes for aggregation and data persistence.

Data Aggregation

Data aggregation refers to techniques for gathering individual data records (for example log records) and combining them into a large bundle of data files. For example, in web server log aggregation, a single log file records all recent visits.

Aggregating data records on Amazon EMR has multiple benefits:

- Improves data ingest scalability by reducing the number of times needed to upload data to AWS. In other words, instead of uploading many small files, you upload a smaller number of larger files.

- Reduces the number of files stored on Amazon S3 (or HDFS), which inherently helps provide better performance when processing data on Amazon EMR. Hadoop, on which Amazon EMR runs, generally performs better with fewer large files compared to many small files.

- Provides a better compression ratio. Compressing large, highly compressible files is often more effective than compressing a large number of smaller files.

Like distributed log collection, you can configure log collectors like Flume and Fluentd to aggregate data before copying it to the final destination (Amazon S3 or HDFS). If you are using a framework that does not support aggregation and you have many small files on the final storage location, you can take advantage of S3DistCp's aggregation feature. Refer to S3DistCp section of Amazon EMR guide at http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/UsingEMR_s3distcp.html.

Data Aggregation with Apache Flume

In your Flume's collector configuration, you can instruct Flume to aggregate the input data or logs before storing the data. Here is an example configuration file (`/etc/flume/conf/flume-site.xml`) that aggregates data files every 5 minutes:

```
<configuration>
  <property>
    <name>flume.master.servers</name>
    <value>ec2-master </value>
  </property>
  <property>
    <name>flume.collector.output.format</name>
    <value>raw</value>
  </property>
  <property>
    <name>flume.collector.dfs.compress.codec</name>
    <value>GzipCodec</value>
  </property>
  <property>
    <name>flume.collector.roll.millis</name>
    <value>300000</value>
  </property>
</configuration>
```

Data Aggregation Best Practices

To appreciate the suggested best practices for data aggregation, it helps to understand how Hadoop processes your data.

Before processing your data, Hadoop splits your data (files) into multiple chunks. After splitting the file(s), a single map task processes each part. If you are using HDFS as the underlying data storage, the HDFS framework has already separated the data files into multiple blocks. In addition, since your data is fragmented, Hadoop uses HDFS data blocks to assign a single map task to each HDFS block.

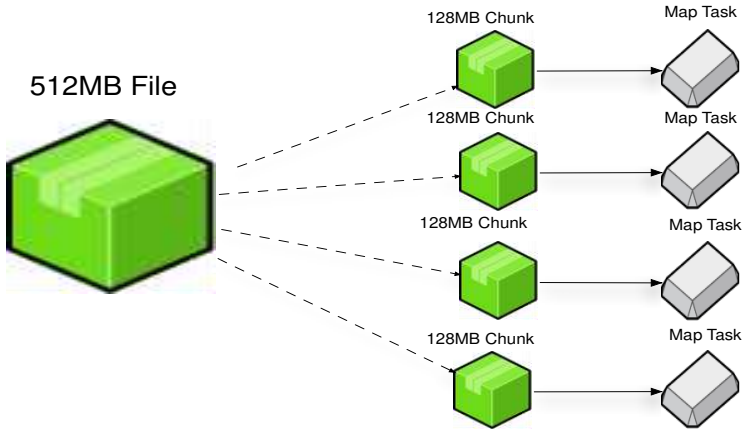

Figure 6: Hadoop Split Logic

While the same split logic applies to data stored on Amazon S3, the process is a different. Since the data on Amazon S3 is not separated into multiple parts the same way data files on HDFS are, Hadoop splits the data on Amazon S3 by reading your files in multiple HTTP range requests. This is simply a way for HTTP to request a portion of the file instead of the entire file (for example, `GET FILE X Range: byte=0-10000`). The split size that Hadoop uses to read data file from Amazon S3 varies depending on the Amazon EMR Amazon Machine Image (AMI) version you're using. The latest Amazon EMR versions have larger split size than the older ones. For example, if a single data file on Amazon S3 is about 1 GB, Hadoop reads your file from Amazon S3 by issuing 15 different HTTP requests in parallel if Amazon S3 split size is 64 MB (1 GB/64 MB = ~15).

To implement best practices about the aggregation size, it helps to understand the impact of the compression algorithm on Hadoop's data splitting logic. Regardless of where your data is stored (HDFS or Amazon S3), if your compression algorithm does not allow for splitting,[5] Hadoop will not split your data file and instead uses a single map task to process your compressed file. For example, if your data file is 1 GB of GZIP file, Hadoop processes your file with a single mapper. On the other hand, if your file can be split (in the case of text or compression that allows splitting, such as some version of LZO) Hadoop splits your file to multiple chunks and process the chunks in parallel.

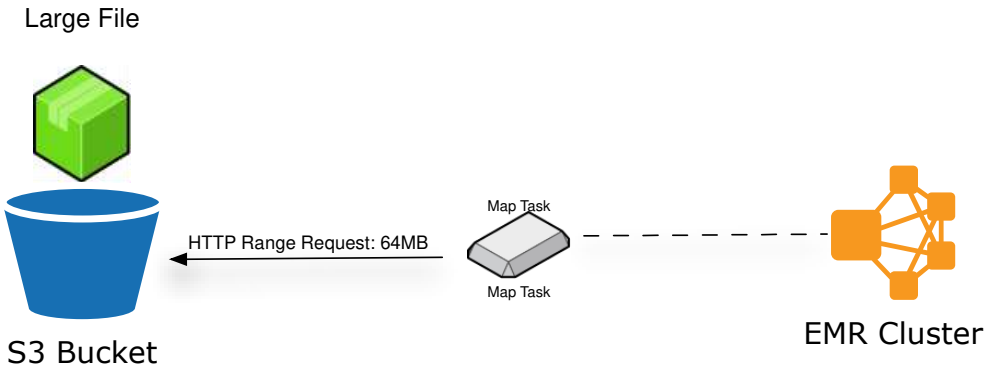

Figure 7: Amazon EMR Pulling Compressed Data From S3

[5] See "What Compression Algorithm Should I Use?" later in this document.

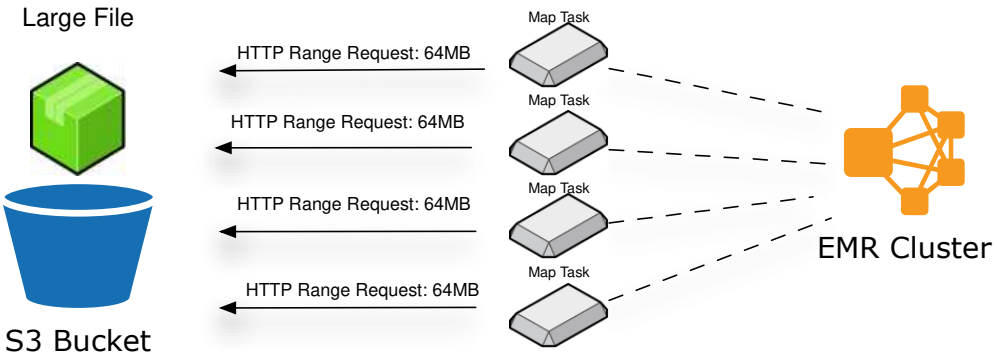

Figure 8: Amazon EMR Using HTTP Range Requests

Best Practice 1: Aggregated Data Size

The suitable aggregated data file size depends on the compression algorithm you are using. (You can find more on data compression later in this paper.) For instance, if your log files are compressed with GZIP, it is often best to keep your aggregated data file size to 1–2 GB. The reason is that since GZIP files cannot be split, Hadoop assigns a single mapper to process your data. That means that a single mapper (a single thread) is responsible for fetching the data from Amazon S3. Since a single thread is limited to how much data it can pull from Amazon S3 at any given time (throughput), the process of reading the entire file from Amazon S3 into the mapper becomes the bottleneck in your data processing workflow. On the other hand, if your data files can be split, more than a single mapper can process your file. The suitable size for such data files is between 2 GB and 4 GB.

Best Practice 2: Controlling Data Aggregation Size

An important question to ask is how do I control my data aggregation size? If you are using a distributed log collector mentioned in the previous sections, you are limited to data aggregation based on time. For example, you can configure Flume to aggregate data for five minutes before sending the aggregated files to Amazon S3. Unfortunately, with time aggregation, you do not have the flexibility to control the exact data file size created; the size of your files depend on the rate of the data being read by your distributed log collector. For example, Flume reading data at the rate of 1 MB/sec means that your files can be up to 300 MB when aggregating every 5 minutes.

At the time of writing this paper, none of the log collection frameworks can aggregate by file size, and because of that, figuring out the right time-based aggregation is mostly a process of trial and error. In the absence of file-size aggregation logic, we recommend you perform your own testing with different time values in your aggregation configuration to understand which value gives you the best file-size aggregation. For example, you may decide to configure Fluentd to aggregate your log files every 5 minutes and after further testing you may notice that 5 minute aggregation is creating 4 GB compressed GZIP files. If you recall from the previous section, storing files larger than 2 GB on Amazon S3 for Amazon EMR processing creates logjams. In that case, you might reduce your time value for aggregation down to 2 minutes or less.

Overall, since most distributed log collector frameworks are open source, you might be able to write special plugins for your chosen log collector to introduce the ability to aggregate based on file size.

Best Practice 3: Data Compression Algorithms

Depending on how large your aggregated data files are, the compression algorithm becomes an important choice. For instance, if you are aggregating your data (using the ingest tool of your choice) and the aggregated data files are

between 500 MB to 1 GB, GZIP compression is an acceptable data compression type. However, if your data aggregation creates files larger than 1 GB, its best to pick a compression algorithm that supports splitting.

Data Compression

Compressing data is important for several reasons:

1. Lower storage costs by reducing your data storage footprint.
2. Lower bandwidth costs by moving less data from the source to the destination.
3. Better data processing performance by moving less data between data storage location, mappers, and reducers.
4. Better data processing performance by compressing the data that Amazon EMR writes to disk, i.e. achieving better performance by writing to disk less frequently.

What Compression Algorithm Should I Use?

Naturally, not all compression algorithms are alike. Consider these potential advantages and disadvantages:

- As the table below suggests, some compression algorithms are faster. You need to understand your workload in order to decide if faster compressions are any use for you. For example, if your job is CPU bounded, faster compression algorithms may not give you enough performance improvement. If you decide compression speed is important, Snappy compression seems to perform faster.
- Some compressions algorithms are slower but offer better space savings, which may be important to you. However, if storage cost is not an important factor, you may want a faster algorithm instead.
- Importantly, some algorithms allow file output to be split. As discussed earlier, the ability to split your data file affects how you store your data files. If the compression algorithm does not support splitting, you may have to maintain smaller file sizes. However, if your compressed files can be chunked, you may want to store large files for Amazon EMR processing.

Compression	Extension	Splitable	Encoding/Decoding Speed (Scale 1-4)	% Remaining (Scale 1-4)
Gzip	gz	No	1	4
LZO	lzo	Yes if indexed	2	2
Bzip2	bz2	Yes	3	3
Snappy	snappy	No	4	1

Figure 9: Compression Formats Compared

Compressing mapper and reducer outputs

Compressing the input file is not the only place in Amazon EMR data processing where compression can help. Compressing the intermediate outputs from your mappers can also optimize your data processing workflow. This is data that Amazon EMR copies to reducers after the mappers are finished outputting the data. Compression helps by reducing

the amount of data to copy over the network. However, the performance boost by compressing the intermediate data completely depends on how much data must be copied to reducers (NOTE: One way to understand how much data Hadoop copies over the network is to look at Hadoop's `REDUCE_SHUFFLE)`. In order to enable mapper intermediate data compression, set `mapreduce.map.output.compress` to true and `mapreduce.map.output.compress.codec` to the compression codec of your choice (GZIP, LZO, or Snappy).

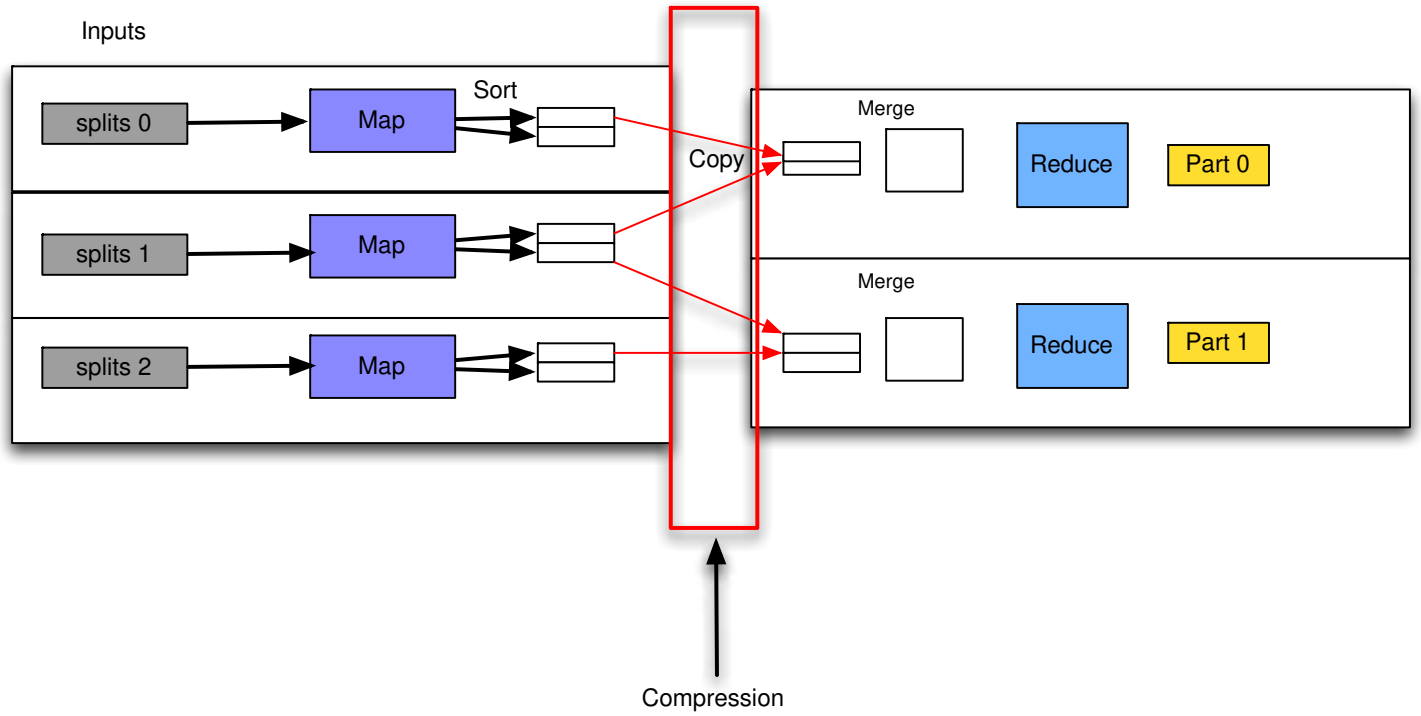

Figure 10: Compressing Mapper Outputs

The other place where you can use data compression is where data is spilled or written to disk from mappers. When mappers are processing the data, the output of mappers gets stored in a memory buffer. The size of mapper's buffer is limited (configurable) and if the input data is more than the mapper's buffer capacity, the data inside the buffer spills over to disk. With compression turned on, you reduce the amount of data to write to disk. LZO is one of the good compression candidates for compressing the mapper's output data. In order to enable compression, make sure you have the following parameter set in your Hadoop Job: `mapred.compress.map.output=true`.

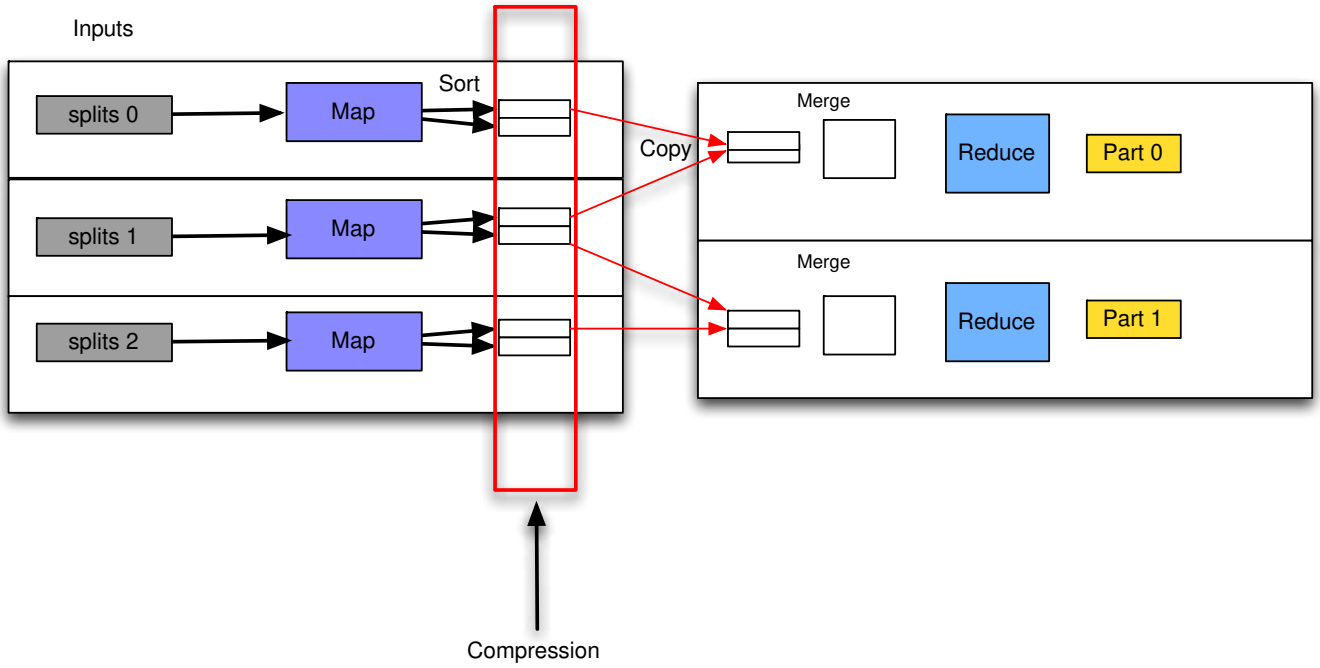

Figure 11: Compressing Mapper intermediary Spills

To summarize, the preferred compression algorithm for your workload depends on the following:

- How fast you need to compress and decompress your data files.
- How much data storage space you'd like to save.
- Whether your data files are small enough that they don't require multiple mappers for processing (data file split) or the files are large enough that you need multiple mappers to process your file simultaneously (in which case you need a compression algorithm that supports splitting).
- When possible, choose native compression libraries such as GZip, which can perform better than the Java implementations.

Best Practice 4: Data Partitioning

Data partitioning is an essential optimization to your data processing workflow. Without any data partitioning in place, your data processing job needs to read or scan all available data sets and apply additional filters in order to skip un-necessary data. Such architecture might work for a low volume of data, but scanning the entire data set is a very time consuming and expensive approach for larger data sets. Data partitioning lets you create unique buckets of data and eliminate the need for a data processing job to read the entire data set.

Three considerations determine how you partition your data:

- Data type (time series)
- Data processing frequency (per hour, per day, etc.)

- Data access and query pattern (query on time vs. query on geo location)

For instance, if you are processing a time-series data set where you need to process your data once every hour and your data-access pattern is based on time, partitioning your data based on date makes the most sense. An example of such data processing would be processing your daily logs. If you have incoming logs from variety of data sources (web servers, devices etc.), then creating partitions of data based on the hour of the day gives you a date-based partitioning scheme. The structure of such partitioning scheme will look similar to the following:

`/data/logs/YYYY-MM-DD-HH/logfiles` for this given hour, where YYYY-MM-DD-HH changes based on the current log ingest time:

```
/data/logs/2013-01-01-02/logfile1
                        .../logfile2
                        .../logfile3

/data/logs/2013-01-01-03/logfile1
                        .../logfile2
                        .../logfile3
```

Processing Data with Amazon EMR

Although Amazon EMR makes data processing extremely easy and convenient, your data processing workflow can still benefit from several considerations.

Picking the Right Instance Size

When provisioning an Amazon EMR cluster, the instance size of your nodes are important, because some workloads are CPU intensive while others are disk-I/O or memory intensive.

For memory-intensive jobs, m1.xlarge or one of the m2 family instance sizes have enough memory and CPU power to perform the work. If your job is CPU intensive, c1.xlarge, cc1.4xlarge, or cc2.8xlarge instances are the best choice. It is common to have jobs that are both memory and CPU intensive, which makes cc1.4xlarge, or cc2.8xlarge instance sizes a better choice because they have suitable memory and CPU power to handle such a workload.

In the compression section of this document, we learned that Hadoop tries to use memory as much as possible; but in cases where there is insufficient memory to process the data (as when performing a sort or group-by on large amount of data), a portion of data is spilled (written) to disk. Because disk writes are expensive and slow down the jobs, use instances with more memory to achieve better performance.

Keep in mind that one of the main benefits of Amazon EMR is that you can experiment with different instance sizes. Just switch between different instance profiles (High Memory and High CPU) by shutting down the cluster and running a new one with a different instance size that better fits your requirements.

Later in this document, you'll learn about architectural patterns that let you mix different types of Amazon EC2 instance types and expand or shrink your cluster size to meet the demand of both CPU and memory-intensive jobs. When you provision your cluster through Amazon EMR CLI or console, depending on the instance size, Amazon EMR configures each instance with the appropriate Hadoop-specific configuration settings, such as the following:

- Java memory (heap) size per daemon

- Number of mappers per instance
- Number of reducers per instance

You can change your Hadoop configuration by bootstrapping your Amazon EMR cluster (explained later), but we recommend using the preconfigured parameters that the Amazon EMR team has tested and fine tuned for you. We recommend changing the Hadoop configuration only if you are an expert in configuring Hadoop.

Picking the Right Number of Instances for Your Job

The optimal number of instances for a cluster depends on the size of your data set and how quickly Hadoop should process your data. Recall that Hadoop processes your input data files by splitting the files in chunks to process them in parallel. The larger the data set, the more splits that Hadoop creates, which in turn means that Hadoop runs more tasks to process your data. The optimal Amazon EMR cluster size is the one that has enough nodes and can process many mappers and reducers in parallel. For example, assuming that you have a data set that requires 20 mappers to process your input file (20 splits by Hadoop), an optimal Amazon EMR cluster can process all mappers in parallel. In this case, to run a very optimized Amazon EMR cluster, you need 10 m1.small instances to run all your map task jobs in parallel (each m1.small can run two map tasks in parallel).

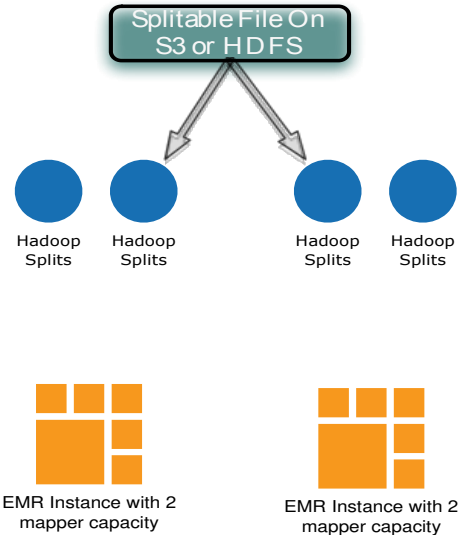

Figure 12: Hadoop's Parallel Split Processing

However, you may not need to run all mappers in parallel in the following cases:

- You have a large data set and it becomes cost prohibitive to create an Amazon EMR cluster that can run all your mapper tasks in parallel.
- You are not time constrained.

In either of the above situations, your Amazon EMR cluster might be more cost effective with fewer nodes. In the case where you do not have enough nodes and enough mapper capacity to execute all mappers in parallel, Hadoop puts the remaining mappers in a queue and as soon as the capacity becomes available, Hadoop processes the remaining mappers and continues processing the queue until all mappers execute as shown in the following diagram.

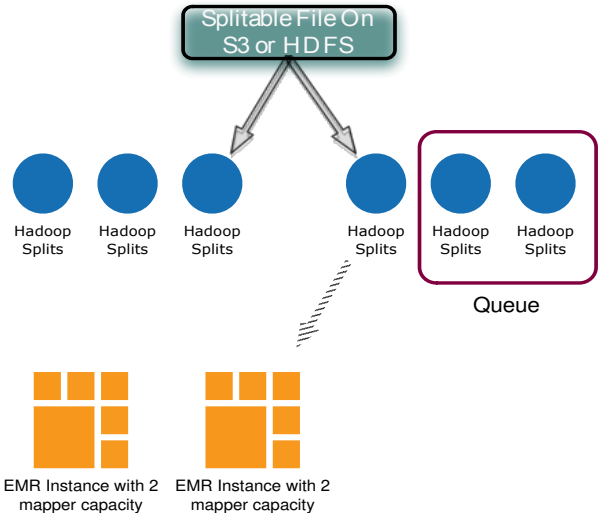

Figure 13: Hadoop's Splits in Queue

Consider the previous example (input size where Hadoop requires 20 mappers to process your data) again and assume you have decided to go with five m1.small instances instead. With five m1.small Amazon EMR nodes, you can process ten mappers in parallel and the remaining ten mappers stay in queue. Once Hadoop has process the first ten mappers, the remaining ten mappers run. Because not all mappers run in parallel, reducing the number of nodes increases your data processing time.

In summary, the optimal number of Amazon EMR instances depends on how quickly you need your data to be processed and how much you are willing to pay for your data processing–time requirement. Amazon EMR charges occur in hourly increments; in other words, running a job that finishes in less than an hour incurs a full hour EC2 usage charge. Consequently, you might choose to reduce the number of Amazon EMR instances to reduce cost, bearing in mind that the processing time will increase.

Knowing how many mappers your cluster can run in parallel depends on the Amazon EMR cluster instance sizes and number of instances. You can find the number of mappers per instance type in the Developer Guide: http://docs.amazonwebservices.com/ElasticMapReduce/latest/DeveloperGuide/TaskConfiguration_AMI2.2.html. For example, if your cluster consists of ten m1.large instances, your cluster can execute 30 mappers in parallel.

Estimating the Number of Mappers Your Job Requires

There are at least two ways to estimate for the number of mappers needed to process your input data files:

1. The number of mappers depends on the number of Hadoop splits. If your files are smaller than HDFS or Amazon S3 split size, the number of mappers is equal to the number of files. If some or all of your files are larger than HDFS or Amazon S3 split size (fs.s3.block.size) the number of mappers is equal to the sum of each file divided by the HDFS/Amazon S3 block size.

   The examples below assume 64 MB of block size (S3 or HDFS).

   Example 1: You have 100 files of 60 MB each on HDFS = 100 mappers. Since each file is less than the block size, the number of mappers equals the number of files.

Example 2: You have 100 files of 80 MB each on Amazon S3 = 200 mappers. Each data file is larger than our block size, which means each file requires two mappers to process the file. 100 files * 2 mappers each = 200 mappers

Example 3: You have two 60 MB, one 120 MB, and two 10 MB files = 6 mappers. The 60 MB files require two mappers, 120 MB file requires two mappers, and two 10 MB files require a single mapper each.

2. An easy way to estimate the number of mappers needed is to run your job on any Amazon EMR cluster and note the number of mappers calculated by Hadoop for your job. You can see this total by looking at JobTracker GUI or at the output of your job. Here is a sample of job output with the number of mappers highlighted:

```
13/01/13 01:12:30 INFO mapred.JobClient:   Total time spent by all reduces waiting after reserving slots (ms)=0
13/01/13 01:12:30 INFO mapred.JobClient:   Total time spent by all maps waiting after reserving slots (ms) =0
13/01/13 01:12:30 INFO mapred.JobClient:   Rack-local map tasks=20
13/01/13 01:12:30 INFO mapred.JobClient:   Launched map tasks=20
13/01/13 01:12:30 INFO mapred.JobClient:   SLOTS_MILLIS_REDUCES=2329458
```

Amazon EMR Cluster Types

Amazon EMR has two cluster types, transient and persistent. Each can be useful, depending on your task and system configuration.

Transient Amazon EMR Clusters

Transient clusters are clusters that shut down when the job or the steps (series of jobs) are complete. In contrast, persistent clusters continue to run after data processing is complete. If you determine that your cluster will be idle for majority of the time, it is best to use transient clusters. For example, if you have a batch-processing job that pulls your web logs from Amazon S3 and processes the data once a day, it is more cost effective to use transient clusters to process your data and shut down the nodes when the processing is complete. In summary, consider transient clusters in one or more of the following situations:

1. Your total number of Amazon EMR processing hours per day is less than 24 and you can benefit from shutting down your cluster when it's not being used.

2. You are not using HDFS as your primary data storage.

3. Your job processing is intensive, iterative data processing.

The following formula is useful to decide if you need to use transient Amazon EMR clusters:

Note: The following formulas provide general guidance only. Your circumstances may vary.

*If number of jobs per day * (time to setup cluster including Amazon S3 data load time if using Amazon S3 + data processing time) < 24 hours, consider transient Amazon EMR clusters.*

For example, if you are storing your log files on Amazon S3 and running 10 web-log processing jobs per day and each job takes about 40 minutes, your total processing time per day is less than ten hours. In this case, you should shut down your Amazon EMR cluster each time your log processing is complete.

The above calculation makes more sense for non-iterative data processing. If you are doing iterative data processing where you process the same set of data over again (as with machine learning, for example), you should experiment by moving your data to HDFS and use the following formula instead:

> *If (time to load data from Amazon S3 to HDFS + number of jobs per day) * data processing time < 24 hours, consider transient Amazon EMR clusters.*

Persistent Amazon EMR Clusters

As the name implies, persistent Amazon EMR clusters continue to run after the data processing job is complete. Similar to transient Amazon EMR clusters, persistent clusters have their own costs and benefits. Consider persistent clusters for one or more of the following situations:

- You frequently run processing jobs where it's beneficial to keep the cluster running after the previous job.
- Your processing jobs have an input-output dependency on one another. Although it is possible to share data between two independent Amazon EMR clusters, it may be beneficial to store the result of the previous job on HDFS for next job to process.
- In rare cases when it is more cost effective to store your data on HDFS instead of Amazon S3.

Similar to transient clusters, you can use the following formula to help understand if you may need to use transient or persistent clusters:

Note: The following formulas provide general guidance only. Your circumstances may vary.

> *If number of jobs per day * (time to setup cluster including Amazon S3 data load time if using Amazon S3 + data processing time) > 24 hour, you may want to use persistent Amazon EMR clusters*

Or

> *If time to load data from Amazon S3 to HDFS + number of jobs per day * data processing time > 24 hours, you may want to use persistent Amazon EMR clusters.*

Common Amazon EMR Architectures

Amazon EMR can be used in a variety of configurations of architectures, each with its own advantages and disadvantages.

Pattern 1: Amazon S3 Instead of HDFS

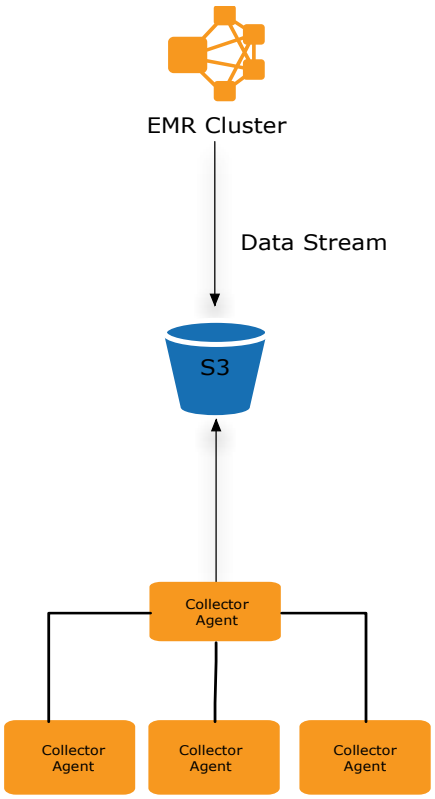

Figure 14: Using S3 Instead of HDFS

In this architecture pattern, we store data in Amazon S3 in accordance to data partitioning, data size, and compression best practices discussed earlier. For data processing, Amazon EMR nodes pull data from Amazon S3 and process it as the data downloads. It is important to understand that with this architecture, Amazon EMR does not copy the data to local disk; instead the mappers open multithreaded HTTP connections to Amazon S3, pull the data, and process them in streams.

This architecture has many benefits:

- Amazon S3 offers highly durable data storage.
- Since data resides on Amazon S3 and not HDFS, concerns around the loss of NameNode are limited.
- You can take advantage of Amazon S3's great and flexible features, such as security and lifecycle management.
- Amazon S3 scales virtually infinitely. You can generally run as many Amazon EMR nodes as needed to pull data in parallel to Amazon EMR nodes. Speeds up your data processing time without the need to scale your HDFS storage. You generally do not need to be concerned with HDFS capacity. Amazon S3 provides a highly scalable amount of storage space.
- Amazon S3 is cost effective. You only pay for the data you store on Amazon S3. In contrast, storing 100 GB on HDFS means storing 100 GB multiplied by the replication factor. For example, with 100 GB data stored on HDFS and the replication factor of two (your data replicated twice to different HDFS nodes), the actual data stored on

HDFS is 200 GB. For large data sets, this means you need to provision more instances with more HDFS storage, which can increase your Amazon EMR costs.

- This architecture offers the ability to use transient Amazon EMR clusters.
- You can run multiple jobs in multiple Amazon EMR clusters over the same data set without overloading your HDFS nodes.
- You get additional protection from data corruption: Because of HDFS data replication, HDFS helps protect your data from corruption after it stores your data. But HDFS cannot prevent data corruption that occurs *before* it stores your data. In contrast, Amazon S3 can help prevent data corruption if you enable Amazon S3 versioning on your data set.
- Amazon S3 offers lifecycle management and gives you the ability to purge and delete data based on pattern and the duration of time the content has been stored on Amazon S3. This is a great feature for controlling storage cost.

Using this architecture has at least one potential drawback:

- For iterative data processing jobs where data needs processing multiple times with multiple passes, this is not an efficient architecture. That's because the data is pulled from Amazon S3 over the network multiple times. For iterative workloads we suggest using architecture pattern #2, Amazon S3 and HDFS.

Pattern 2: Amazon S3 and HDFS

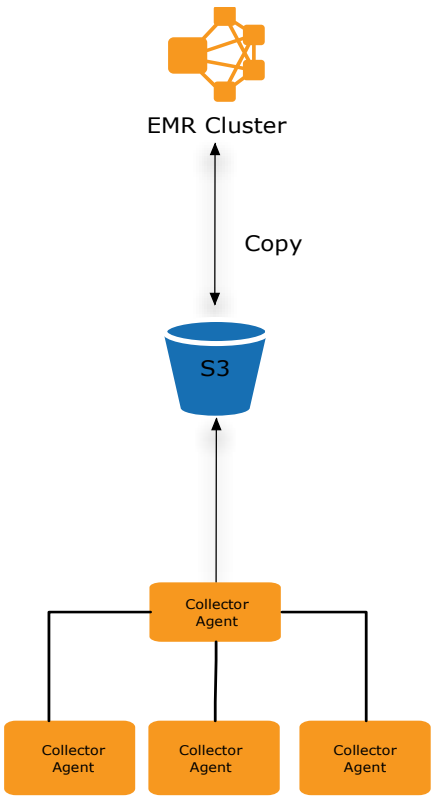

Figure 15: Using S3 and HDFS

In this architecture pattern, we store data in Amazon S3 according to data partitioning, data size, and compression best practices discussed earlier. Prior to running your data processing job, Amazon EMR copies the data to HDFS using DistCp or S3DistCp. The advantage over the previous pattern is that if your job is doing iterative work on the same data set, this pattern avoids a data copy from Amazon S3 to Amazon EMR nodes every time you process your data. For that reason, iterative workloads may reduce processing time by copying the data to local disk.

This architecture has multiple advantages:

- Amazon S3 provides highly durable data storage.
- You can take advantage of Amazon S3's great and flexible features, including the following:
  - **Bucket permission and access control** let you control who can access your data and what actions they can take.
  - **Data at rest encryption** provides server-side encryption where you data can be encrypted while at rest.
  - **Lifecycle management** lets you purge or delete data based on storage pattern and duration so you can control storage cost.
- Amazon S3's massive scale lets you run as many Amazon EMR nodes as needed to pull data in parallel using S3DistCp or DistCp.
- If data is stored on HDFS, you are required to add capacity (HDFS nodes) as the data grows. Amazon S3's massive storage capacity eliminates the need to add more storage nodes.
- This architecture lets you use transient Amazon EMR clusters. Since your data persists in Amazon S3, you can shut down your Amazon EMR cluster after the data processing job is complete.
- You can run multiple jobs in multiple Amazon EMR cluster over the same data set without overloading your HDFS nodes.

This architecture also has at least one potential drawback:

- Having to copy your data to HDFS first introduces a delay to your data processing workflow. But if you have iterative workloads, the quicker data processing can compensate for delays.

Pattern 3: HDFS and Amazon S3 as Backup Storage

In this architecture pattern, we store data directly on HDFS, Amazon EMR nodes process the data locally, and we use S3DistCp or DistCp to copy the data to Amazon S3 periodically. The main benefit of this pattern is the ability to run data processing jobs without first copying the data to the Amazon EMR nodes.

Although this architecture pattern offers data processing speed, one of the main disadvantages is data durability. Because Amazon EMR uses ephemeral disks to store data, you could lose data if an Amazon EMR EC2 instance fails. While data on HDFS replicates within the Amazon EMR cluster and HDFS can usually recover from node failures, you could still lose data if the number of lost nodes is greater than your replication factor. To avoid data loss, we recommend backing up your HDFS data to Amazon S3 in periodic phases using DistCp or S3DistCp. We also recommend making backups of your data partitions rather than your entire data set to avoid putting too much load on your existing Amazon EMR cluster.

Pattern 4: Elastic Amazon EMR Cluster (Manual)

In the manual pattern, your Amazon EMR architecture starts with enough nodes to process your daily flow of data. Since your data grows over time, you'll need to manually monitor capacity and pre-size your Amazon EMR cluster to match your daily processing needs.

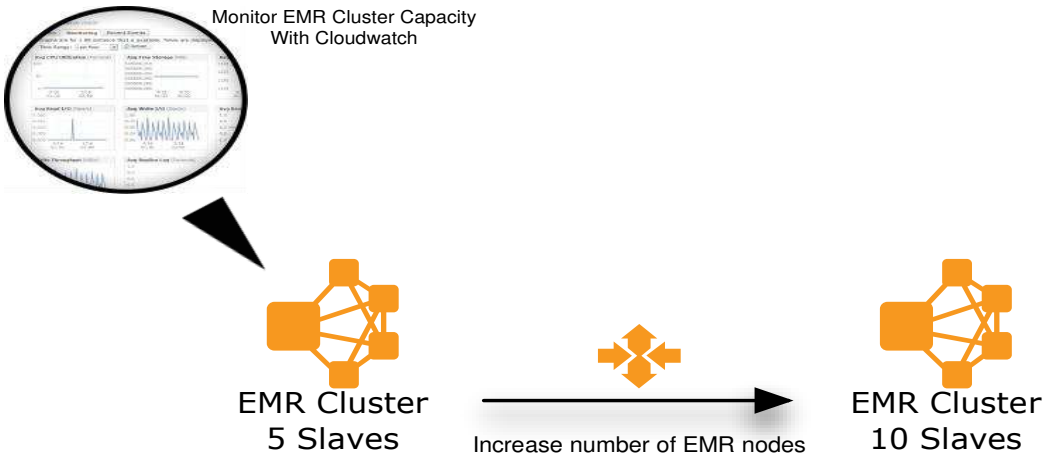

Figure 16: Monitoring Amazon EMR Capacity Using Amazon Cloudwatch

Amazon EMR utilizes several Amazon CloudWatch metrics:

- Number of mappers running
- Number of mappers outstanding
- Number of reducers running
- Number of reducers outstanding

With this architecture, your Amazon EMR clusters starts with X number of nodes to process your daily amount of data. However, after monitoring the above Amazon CloudWatch metrics, you notice that your Amazon EMR processing time has increased due to your Amazon EMR cluster's large number of outstanding mappers or reducers. In other words, you lack sufficient Amazon EMR capacity to process your entire data set.

Adapting to increasing data is a common challenge. As a solution, you can manually add more nodes using the API, CLI, or via AWS. In the next section, we will define a dynamic, elastic Amazon EMR architecture.

Pattern 5: Elastic Amazon EMR Cluster (Dynamic)

Amazon EMR consists of three node types:

- Master node, which runs JobTracker and NameNode
- Core nodes, which run TaskTracker and DataNodes

- Task nodes, which run TaskTracker only

Core nodes offer HDFS storage (DataNode) while task nodes do not (no local DataNode). Task nodes are meant for data processing only (running mappers and reducers). One of the benefits of not having HDFS on task nodes is that the Amazon EMR cluster can easily add or remove task nodes. Because HDFS is not running on these nodes, you can add task nodes to the cluster to increase computing capacity while removing unnecessary nodes.

With this pattern, the Amazon EMR cluster initially consists of a limited number of core nodes to provide minimum HDFS storage. When Amazon EMR submits data processing jobs to the cluster, you can start adding task nodes to this cluster to increase its capacity beyond what the core nodes provide. When data processing job concludes, you can start removing your Amazon EMR task nodes.

The job of adding more task nodes to an Amazon EMR cluster can be automated using Amazon EMR CloudWatch metrics. Amazon EMR can utilize several CloudWatch metrics:

- Number of mappers running or outstanding
- Number of reducers running or outstanding
- Cluster Idle
- Live Data nodes or task nodes

Amazon CloudWatch's integration with Amazon Simple Notification Service (Amazon SNS) lets you set up alarms on your Amazon CloudWatch metrics and receive notifications at each alarm threshold. Combining Amazon EMR CloudWatch metrics with SNS notification lets you automate actions like adding more task nodes when your Amazon EMR cluster is out of capacity. For example, you might have a persistent Amazon EMR cluster shared among multiple groups within your organization, with each group sending its own data processing job to Amazon EMR. By monitoring CloudWatch metrics, you notice that your Amazon EMR cluster is slowing down during business hours as more jobs are submitted, with many of them taking a long time to queue. The steps below outline an automated workflow you can implement to add Amazon EMR nodes to your cluster dynamically:

1. Amazon EMR exposes CloudWatch metrics. You can set up a CloudWatch alarm on Amazon EMR CloudWatch metrics to notify you when you reach a threshold.

2. Amazon EMR sends an SNS notification to an HTTP endpoint hosted on an EC2 instance or AWS Elastic Beanstalk. The HTTP endpoint is a simple application (i.e., Java servlet) that takes HTTP notifications from AWS SNS and triggers the appropriate action (Step 3). (**Note:** HTTP endpoint is an application developed by the customer.)

3. The HTTP endpoint application makes a request to Amazon EMR API endpoints and adds more instances to the currently running Amazon EMR cluster.

4. Amazon EMR workflow management adds more nodes to your Amazon EMR cluster.

Note: A similar workflow applies when the CloudWatch alarm triggers a low-utilization alert and the workflow starts removing Amazon EMR nodes when they are not used.

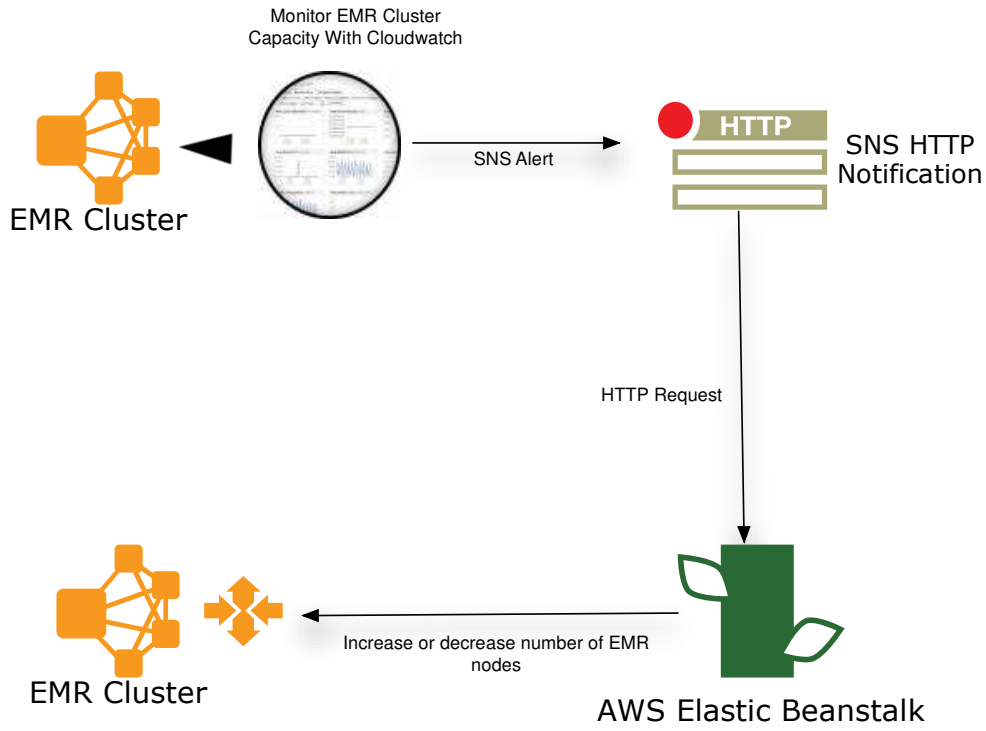

Figure 17: Automating Amazon EMR Capacity Provisioning

This architecture pattern has some important advantages:

- You can control costs by adding nodes only when more capacity is needed and removing them when they're not used.
- In most Hadoop architectures, you are limited to whatever resources the Hadoop nodes provide. By leveraging Amazon EMR, you have the ability to control and improve processing time by adding more capacity as required.

Optimizing for Cost with Amazon EMR and Amazon EC2

AWS provides a variety of ways to optimize your computing cost. In general, AWS offers three pricing models:

1. **On-Demand Instances**–With On-Demand instances, you pay for compute capacity by the hour with no required minimum commitments.

2. **Reserved Instances (RI)**–Reserved Instances give you the option to make a low, one-time payment for each instance you want to reserve and in turn receive a significant discount on the hourly usage charge for that instance.

3. **Spot Instances**–With Spot Instances, you can bid for unused Amazon EC2 capacity and bring your computing cost down by paying the EC2 hourly charge at a discount.

Depending on your Amazon EMR workload, you can optimize your costs for Amazon EMR by purchasing EC2 Reserved Instances or Spot Instances. On-Demand instances are a good option if you have transient Amazon EMR jobs or if your Amazon EMR hourly usage is less than 17% of the time. However, if your Amazon EMR hourly usage is more than 17%, Reserve Instances can save you money. For Reserved Instances, you make a low, one-time payment for each instance you want to reserve; in turn, you receive a significant discount on the hourly usage charge for that instance.

AWS offers different Reserved Instance types based on the EC2 instance utilization: Light, Medium, and Heavy. If you know the utilization of your Amazon EMR cluster, you can save even more. The Light Utilization model is a great option if you have periodic Amazon EMR workloads that run only a couple of hours a day or a few days a week. Medium Utilization Reserved Instances are the same Reserved Instances that Amazon EC2 has offered these last several years. They are a great option if you don't run Amazon EMR workloads all the time and you want the option to shut down your Amazon EMR cluster when you no longer need them (transient Amazon EMR workload). If you run consistent steady-state Amazon EMR workloads, the Heavy Utilization model is a good option.

Let us run through a few examples.

Example 1

Assume you have a steady daily Amazon EMR workload where you are running processing jobs every hour and it is beneficial to keep the cluster around (persistent) rather than shutting it down when the job is complete. The following graph represents your hypothetical daily Amazon EMR workload. The blue area represent the hourly number of instances you are utilizing to process your job. It is clear that your cluster is utilized 100% of the time. If your workload matches the characteristics mentioned previously, Heavy Utilization Reserve Instances can save you money.

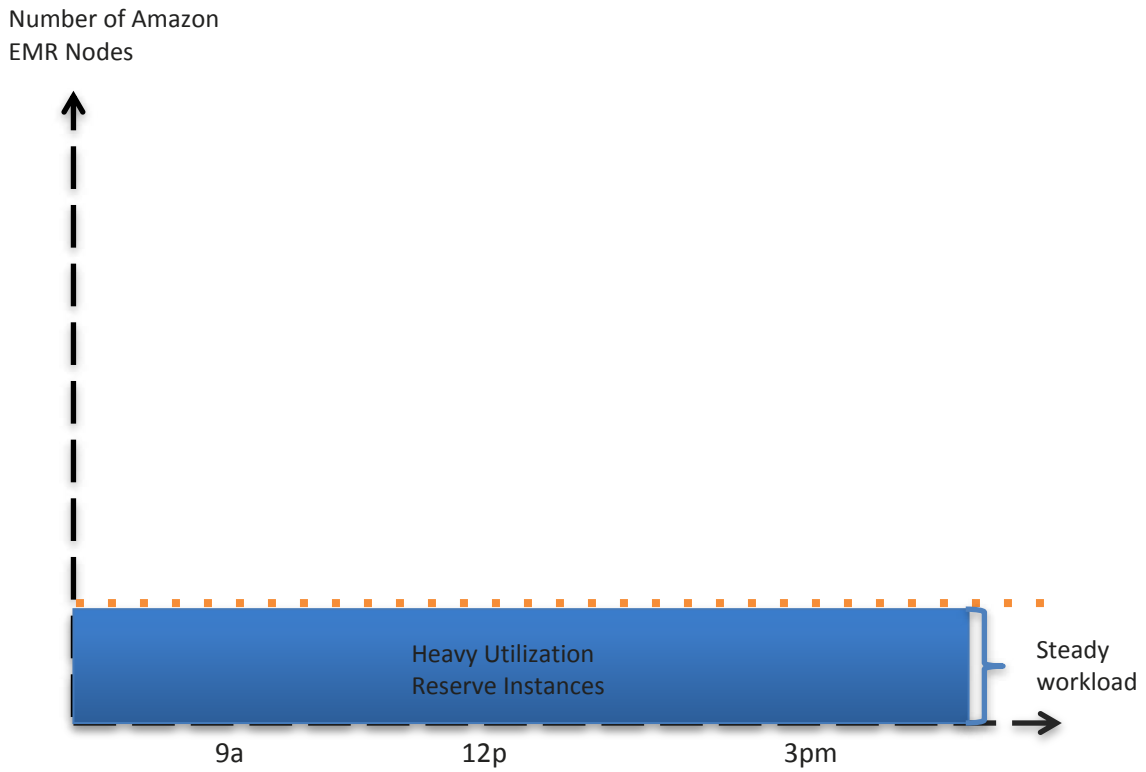

Figure 18: Cost Optimization Using Heavy Reserve Instances

Example 2

Assume you have an Amazon EMR workload in which you run your processing job a few times a day. In this scenario, since you are not using your Amazon EMR cluster enough to need a persistent cluster, you decide to shut down the Amazon EMR cluster after the processing. Because you are not utilizing your cluster 100% of the time, Heavy Utilization Reserve Instances are not a good choice. To save money, you could use Medium Utilization Reserve Instances instead. Although Medium Utilization Reserve Instances do not provide as much saving as Heavy Utilization, they are still a good option for workloads in which the processing job does not consume 100% of the time. In addition, they offer better rates compared to On-Demand instances.

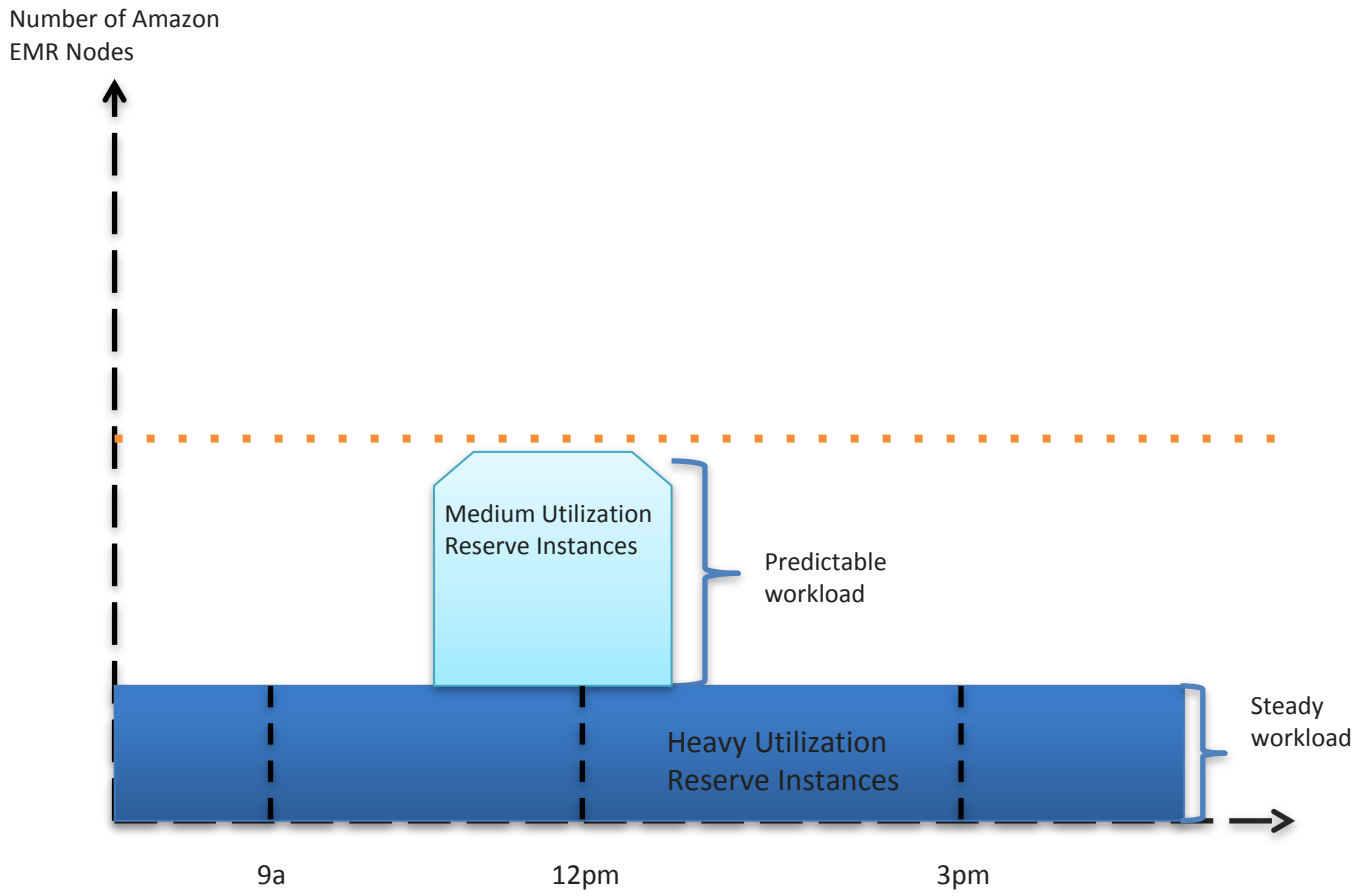

Figure 19: Cost Optimization Using Reserve Instances

Example 3

The two previous models assume that your workload is either steady (daily hourly basis) or a limited number of times per day. However, a mix of both workloads is also very common. In this scenario, you have a workload requiring your Amazon EMR cluster 100% of the time as well as transient workloads that can be either predictable or unpredictable. For unpredictable workloads, the suggested pricing model is Spot or On-Demand. To illustrate this scenario better, look at the Amazon EMR hourly usage below:

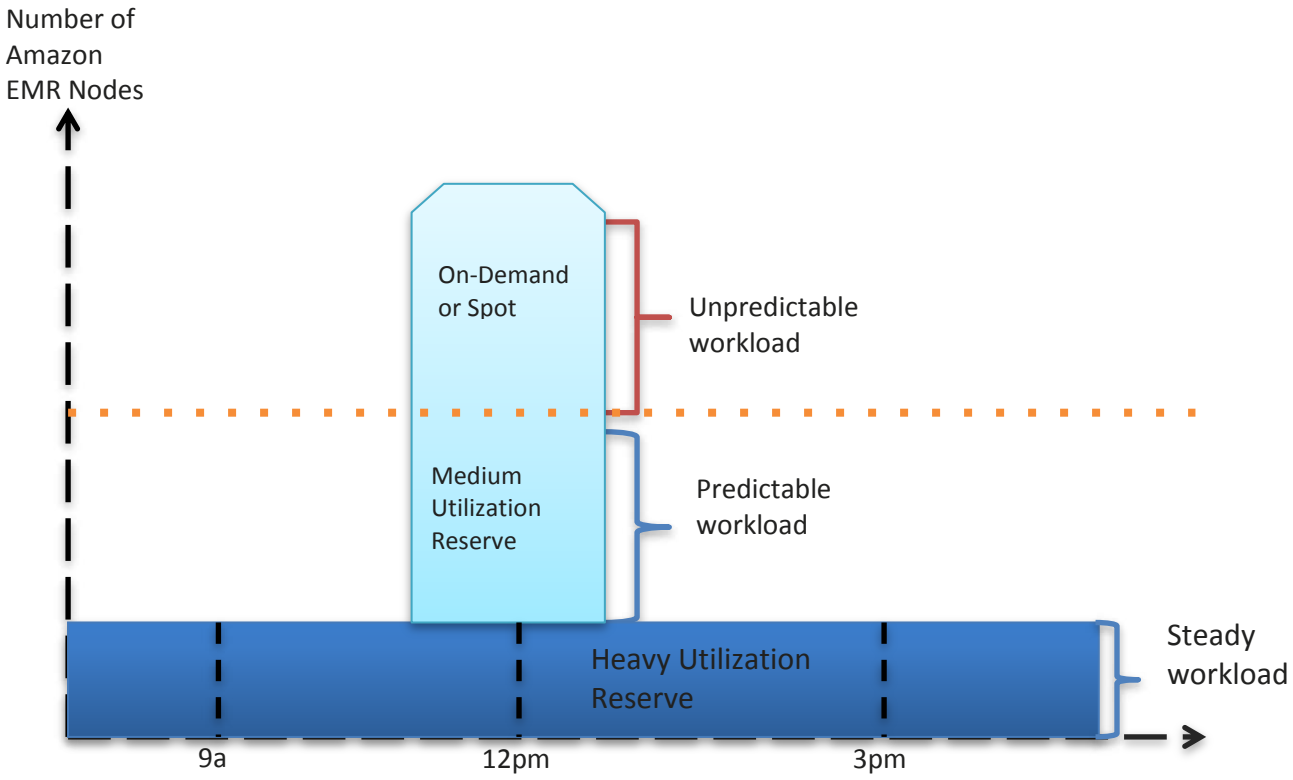

Figure 20: Cost Optimization Using On-Demand and Spot Pricing

Optimizing for Cost with EC2 Spot Instances

Spot Instances let you name your own price for Amazon EC2 computing capacity. You simply bid on spare Amazon EC2 instances and run them whenever your bid exceeds the current Spot price, which varies in real time based on supply and demand. Spot Instance pricing complements On-Demand and Reserved Instance pricing, providing potentially the most cost-effective option for obtaining compute capacity, depending on your application.

Spot Instances can significantly lower your computing costs for time-flexible, interruption-tolerant tasks. Spot prices are often significantly less than On-Demand prices for the same EC2 instance types (see current Spot prices at http://aws.amazon.com/ec2/spot-instances/#7). Amazon EMR lets you run your Amazon EMR cluster on EC2 Spot Instances. There are number of considerations and architectural patterns to consider when using EC2 Spot Instances:

1. The availability of Spot Instances depends on the bid price of the EC2 Spot Instances. If you bid too low and the market price of the EC2 instance goes above your bid price, the EC2 instance may become unavailable to you. For this reason, you should consider always using a bid price that can let you keep the EC2 instance for a longer duration. You can achieve that by looking at the history of EC2 Spot price to detect Spot pricing patterns. Another way to bid for Spot Instances is to bid close to the On-Demand prices. That helps provide not only a faster fulfillment of your EC2 Spot bid, but also increases the lifetime of the EC2 Spot Instance.

2. Amazon EMR instances fall into three categories: master, core, and task nodes. It is possible to run the entire Amazon EMR cluster on EC2 Spot Instances. However, it is not a good practice to run master nodes on Spot Instances due to fact that if you lose the master node (due to Spot market price going above your bid price); you are bound to lose the entire cluster. Similarly, you can run the core nodes on EC2 Spot Instances. However, since core nodes run HDFS (DataNode), if you lose core node(s), your Amazon EMR cluster needs to recover the lost

data and rebalance the HDFS cluster (if you're using HDFS instead of Amazon S3). Also, remember that if you lose the majority of your core nodes to the point where it is impossible to recover your HDFS data from the other available nodes, you are bound to lose the entire cluster.

Task nodes are the safest nodes to run on EC2 Spot Instances. Since task nodes do not host any data (no HDFS), losing task nodes due to the market price increase simply means that you are losing Hadoop TaskTracker nodes. Losing TaskTracker nodes causes Hadoop JobTracker to detect task-processing failures (mapper or reducer tasks) and retry the failed task on a different Amazon EMR node. Although this may slow data processing, the overall health of your Amazon EMR cluster is not affected.

3. Despite the risks of running the entire cluster on Spot Instances (master and core nodes), you can still run the entire Amazon EMR cluster on Spot Instances if your data persists on Amazon S3. In that case, if you lose your master or core nodes and eventually lose the entire Amazon EMR cluster, you can still rerun your data processing job by launching a new Amazon EMR cluster. Although this delays your job, it can be very cost effective if you can tolerate the delay.

4. If you decide to use Spot Instances in your Amazon EMR data processing jobs, consider the following architecture: Run your master node on On-Demand or Reserved Instances (if you are running persistent Amazon EMR clusters). Run a portion of your Amazon EMR cluster on core nodes using On-Demand or Reserved Instances and the rest of the cluster on task nodes using Spot Instances. For example, if you know you need 20 Amazon EMR nodes to process your job, run 10 of your nodes on core instances using On-Demand or RI instances and the remaining 10 on task nodes using Spot Instances.

Performance Optimizations (Advanced)

Before spending time on the optimizations described in this document, note that the Amazon EMR team has already performed rigorous optimization on the Hadoop configuration that you get with each instance (based on available CPU and memory resources on that instance). Also, we recommend the optimizations found in this document only for advanced Hadoop users. We do not recommend them if you have just started to explore Hadoop.

Overall, we generally suggest optimizing for cost over performance, unless improved performance can provide cost optimization. Here's why:

- The best performance optimization is to structure your data better (i.e., smart data partitioning). By structuring your data more efficiently, you can limit the amount of data that Hadoop processes, which inherently gives you better performance. Because Amazon EMR clusters already have optimizations based on the instance type, other optimizations mentioned later in this document provide only slight performance improvements. The exception is triple-digit cluster sizes where improving your processing time by a few seconds or minutes can produce a substantial time and cost saving.

- Hadoop is a batch-processing framework that measures the common processing time duration in hours to days. If you have processing time constraints, Hadoop may not be a good framework for you. In other words, if you need to improve your processing time by a few minutes to meet your data processing SLA, your requirements may be better met by other frameworks such as Twitter's Storm or Spark.

- Amazon EMR charges on hourly increments. This means once you run a cluster, you are paying for the entire hour. That's important to remember because if you are paying for a full hour of Amazon EMR cluster, improving your data processing time by matter of minutes may not be worth your time and effort.

- Don't forget that adding more nodes to increase performance is cheaper than spending time optimizing your cluster.

Once you have decided to optimize your Amazon EMR cluster, we recommend the following steps:

1. Run a benchmark test before and after your optimizations.

    a. Ensure your benchmark reflects your everyday workload.

    b. Try to eliminate as many variables as possible. For example, if you are optimizing your CPU and memory utilization, try not to use Amazon S3 as your data source and instead move data to HDFS before running your benchmark. Once you are satisfied with your CPU and memory utilizations, run a separate benchmark with data residing on Amazon S3.

    c. Run your benchmark multiple times and use the average processing time as your baseline.

2. Monitor your benchmark tests using Ganglia, an open source monitoring tool which can be installed on Amazon EMR using a bootstrap action. [6]

3. Identify your constraints by monitoring Ganglia's metrics. The common constraints can be seen in the following areas:
    - Memory
    - CPU
    - Disk I/O
    - Network I/O

4. Once you have identified the potential constraint, optimize to remove the constraint and start a new benchmark test.

Suggestions for Performance Improvement

Mappers and reducers both provide areas for optimizing performance.

Map Task Improvements

The following tips can help optimize your mapper tasks.

- **Map task lifetime**–Monitor your Hadoop mapper tasks and note how long they run on average. If they have a short lifespan (for example seconds rather than minutes), you may want to reduce the number of mappers. Since the number of mappers is a function of your input size (splits), reducing the number of mappers usually means using larger file sizes. In other words, if your map tasks are short lived, it is likely that you are processing small files and you can benefit by aggregating them into large files. Here are two approaches:

[6] http://ganglia.sourceforge.net/

i. Aggregate files by file size or by time.

ii. Aggregate smaller files into larger Hadoop archive files (HAR).

- **Compress mapper outputs**–Compression means less data written to disk, which improves disk I/O. You can monitor how much data written to disk by looking at `FILE_BYTES_WRITTEN` Hadoop metric. Compression can also help with the shuffle phase where reducers pull data. Compression can benefit your cluster HDFS data replication as well.

  Enable compression by setting `mapred.compress.map.output` to `true`. When you enable compression, you can also choose the compression algorithm. LZO has better performance and is faster to compress and decompress.

- **Avoid map task disk spill**–Avoid Map tasks spilling or writing to disk by providing enough memory for the mapper to keep data in a buffer. You can do that by increasing the `io.sort.*` parameters. The Amazon EMR team has set this value based on the instance type you are using. In most cases, you want to adjust the `io.sort.mb` parameter, which determines the size of your mapper task buffer. Monitor the Hadoop metric named `SPILLED_RECORDS` and experiment with a larger `io.sort.mb` value. Proceed with caution, because setting this number too high can limit the memory for mapper processing and can cause Java out-of-memory errors. It is common to increase mapper heap size prior to increasing `io.sort.mb`.

Reduce Task Improvements

The following tips can help optimize your reducer tasks.

- Your job should use fewer reducers than the cluster's total reducer capacity. Reducers should finish at the same time and not allow reducers to wait for execution, which happens if your job uses more reducers than the cluster's capacity. CloudWatch provides Hadoop metrics such as **Average Reduce Task Remaining** and **Average Reduce Tasks Running**. For best performance, ensure that your cluster is large enough to have all your reduce tasks in "Running" state and potentially zero in the "Remaining" state.

- Similar to configuring mappers, retain as much data in memory as possible. If your reducer job requires a small memory footprint, increase the reducer memory by adjusting the following parameters:

  - `mapred.inmem.merge.threshold` set to `0`

  - `mapred.job.reduce.input.buffer.percent` set to `1.0`

Use Ganglia for Performance Optimizations

If you use Ganglia, the following tips may help improve your data processing performance.

- **CPU**–Run your job and watch CPU utilization of your cluster. If you are not fully utilizing your CPU, you may be able to increase the task capacity (increase the number of mapper or reducer capacity) per node and potentially decrease the number of nodes in your Amazon EMR cluster. Keep in mind that this only helps if you are not fully utilizing your CPU. If CPU is at maximum (your job is CPU bounded), adding more task capacity to each node can potentially cause more CPU context switching, which degrades performance.

- **Memory**–Watch your memory utilization using Ganglia. If after running your job, you notice memory is not being fully utilized, you may benefit from increasing the amount of memory available to the task jobs (mapper or reducers). For instance, the following example decreases the amount of mappers per CC2 instance but increases the mapper/reducer memory:

```
elastic-mapreduce --create --alive --instance-group master --
instance-type c1.xlarge --instance-count 1 --instance-group core --
instance-count 25 --instance-type cc2.8xlarge --bootstrap-action
s3://elasticmapreduce/bootstrap-actions/configure-hadoop --args "-
m,mapred.tasktracker.map.tasks.maximum=14,-m,mapred.child.java.opts=-
Xmx2300m"
```

- **Network I/O**–If you are using Amazon S3 with Amazon EMR, monitor Ganglia and watch network throughput. Your goal is to maximize your NIC throughput by having enough mappers in each node to download as many input files in parallel. For instance, if you have 100 files on Amazon S3 that you need to process, your Amazon EMR cluster should have total capacity of 100 mappers to process all input files in parallel. If your cluster does not have 100 mapper capacity, two options may help:

    i. The easiest option is to increase the number of nodes in your Amazon EMR cluster.

    ii. The other option is to add more mapper capacity per node. For example, if the default mapper capacity per node is two, you could increase that to a higher number. Before adding more mappers, it is important to watch Ganglia and make sure your job is not CPU bounded. If your job is already utilizing 100% of your CPU, adding more mappers will not help. However, if you are not CPU bounded and can add more mappers per node, additional mapper tasks add parallel download threads to Amazon S3, which in turn should speed up the mapper processing time (each mapper means one thread to Amazon S3 to download data). You can potentially repeat this process until you saturate CPU or NIC throughput. The other important item to monitor is your memory. If you decide to increase the mapper capacity per node, ensure that you have enough free memory to support the new mapper added.

  There is a chance that you have enough mapper capacity in your cluster but your job is running fewer mappers than your mapper capacity. The obvious case would be when your Amazon EMR cluster has more nodes than your data requires. The other potential case is that you are using large files that cannot be split (such as GZ compressed). A potential optimization here is to reduce your data file sizes by splitting the compressed files to smaller files or use compression that supports splitting. You must strike a balance between how small or large your files should be. The best case is break up your files to a point where you are fully utilizing your cluster mapper capacity.

- **Monitoring JVM**–Monitor Ganglia JVM metrics and watch for GC (garbage collector) pauses. If you see long GC pauses, it means you are not providing enough memory to your mapper/reducer tasks. Increase JVM memory if possible. If not, add more instances to remove pressure.

- **Disk I/O**–Beware of excessive disk I/O. Disk I/O can be a huge bottleneck, so avoid writing to disk if possible. Two settings can help optimize disk I/O: Mapper spills and reducer spills (`SPILLED_RECORDS`). You can monitor both by Hadoop metrics (See the following section, “Locating Hadoop Metrics”). To reduce disk spill, try the following:

    i. Use compression for mapper output: `set mapred.compress.map.output` to `true`.

ii. Monitor Disk I/O metrics on Ganglia and increase your task memory if needed. If you've already followed the previous recommendations on compression and you see both substantial disk activity and an increase in Hadoop tasks memory, increasing Hadoop's buffer size may improve disk I/O performance. By default, the Hadoop buffer is set to a low number (4 KB). You can increase that number of changing the `io.file.buffer.size` parameter.

Locating Hadoop Metrics

The Hadoop JobTracker Web UI gives you easy access to Hadoop's internal metrics. After running each job, Hadoop provides a number of metrics that are very useful in understanding the characteristics of your job. Follow these steps to see these metrics:

1. Go to JobTracker UI interface (masternode:9100).
2. Click on any completed or running job.
3. Review the overall aggregate metrics.
4. Click on a mapper/reducer to get metrics for that map–reduce job.

Conclusion

Amazon EMR provides businesses the ability to easily deploy and operate Hadoop clusters in short amount of time. In This document we walked through some best practices and architectural patterns for moving data to Amazon Web Services (AWS) for Amazon Elastic MapReduce (EMR) processing, examined best practices on data aggregation and compression, and provided tips for optimizing your Amazon EMR cluster.

Further Reading and Next Steps

1. Getting Started With Amazon Elastic MapReduce – Video Series, http://aws.amazon.com/elasticmapreduce/training/.
2. Amazon Elastic MapReduce Developer Guide, http://aws.amazon.com/documentation/elasticmapreduce/.

Appendix A: Benefits of Amazon S3 compared to HDFS

- One of the biggest advantages of combining Amazon EMR with Amazon S3 combo is the ability to shut down your Amazon EMR cluster when you are finished processing your data. Many workloads that can take advantage of this offering.

- Persisting data on Amazon S3 lets you run your Amazon EMR workload on Spot Instances and not worry about having the Spot Instances taken away from you.

- Your data is safe from any HDFS node failures. While Hadoop's high replication factor can help, there are still cases where you may lose HDFS nodes and experience data loss. Storing data on Amazon S3 keeps you safe from any HDFS node failures.

- Amazon S3 offers features such as lifecycle, Secure Side Encryption, S3 Glacier that you can take advantage of to be more secure or reduce cost.

- If you are using a popular third-party distributed log collector such as Flume, Fluentd or Kafka, ingesting high throughput data stream to Amazon S3 is much easier than ingesting to HDFS. In other words, with Amazon S3 as the source of data, Amazon EMR can ingest a titanic amount of data.