WHITE PAPER

ALLUXIO

Accelerating On-Demand Data Analytics with Alluxio

Calvin Jia
Software Engineer at Alluxio

This whitepaper consists of two portions. The first is a high level overview of the advantages of using Alluxio as a core technology with on-demand clusters. The second portion is intended for engineers; it provides a detailed step-by-step guide to deploying an on-demand cluster with Alluxio and instructions for running a sample workload on the cluster. At the end of the paper you will have a good understanding of how to deploy this architecture and the value Alluxio brings to the stack.

- Memory speed data access.
- Efficient data sharing between applications.
- Transparent data access to storage systems.
- Reduced memory footprint.

Introduction

In the Big Data world, it is often the case that only a subset of the total data is relevant for answering the question at hand. As a result, the total cost of ownership for long running clusters for analytics is high while utilization is low, especially when adopting an architecture of co-locating compute and storage. The full compute power of the clusters is often unused, since data insights are driven by humans who query on an ad-hoc basis, unlike a data pipeline which continuously runs. The storage capacity of the cluster must also accommodate any data which may be queried against, when in reality the working set is only a small fraction of the total data. Finally, the cluster itself requires significant maintenance and management to ensure performance and isolation between groups using the cluster.

A simple and elegant solution to this problem is the concept of on-demand compute clusters paired with object storage. The proposed architecture addresses the root of the problem by decoupling persistent, long running storage and ephemeral computation. This architecture provides several benefits over the previous continuously running analytics cluster.

1. Higher storage cost-effectiveness and scalability - Object storage is cost effective and most providers can scale to arbitrary amounts of data seamlessly.
2. Higher compute cost-effectiveness and elasticity - Only provision compute resources when necessary and scale the cluster size to fit the task.
3. Lower cost of maintenance - Clusters are expendable and do not need to be maintained over long periods of time. Users also do not need to worry about interfering with the data as they are only reading a copy.

However, there is also a critical downside to this architecture.

1. Lower performance - Object storages are typically remote and not designed for high I/O throughput resulting in possibly unacceptably slow job execution times.

This is addressed by deploying Alluxio on compute nodes, bringing performance up to memory speeds without requiring a long running cluster or expensive up front costs.

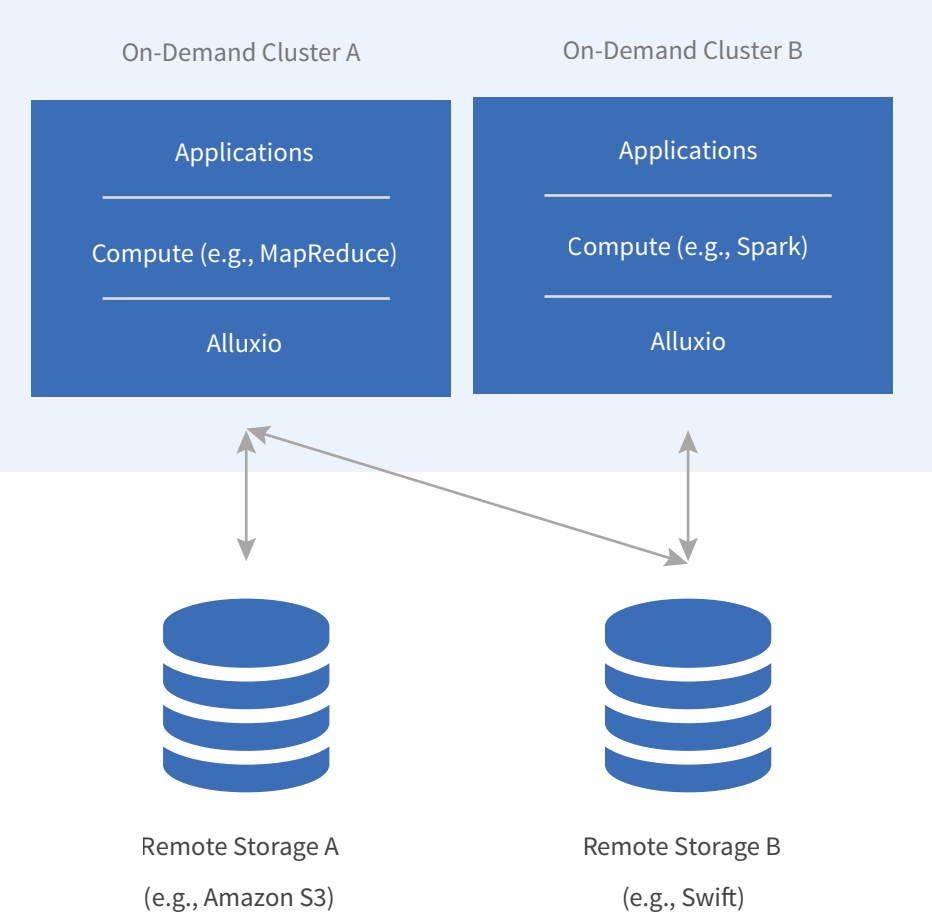
Motivation and Advantages of Alluxio

Alluxio is a memory-speed virtual distributed storage system deployed in compute clusters. Alluxio virtualizes underlying storage systems, making any applications written with Alluxio in mind automatically work with any or multiple under storage systems without any code modifications. In addition, Alluxio is designed to be a scale-out distributed system, meaning adding more machines will gracefully accommodate larger datasets and improve performance.

Deploying and leveraging Alluxio is simple and transparent to applications. Applications continue to access the data as if they were running the job against the remote storage, and Alluxio will intelligently keep hot data in memory for subsequent reads. The entire process is transparent to the application and there is no manual ETL required.

Alluxio provides flexibility and efficiency to its users. Any results or transformations which need to be persisted can be done directly through Alluxio which can synchronously propagate the data to an under storage system, ensuring no data is lost. In addition, users have the option of storing temporary or intermediate data in Alluxio memory only, allowing for memory-speed reads and writes.

Compute Datacenter



Highlight

Once the data is brought into Alluxio, it will be available for all applications of the cluster, drastically improving performance when the same data set is being used multiple times.

Example Use Case

This section will go through the life cycle of an on-demand compute cluster with Alluxio. Readers can follow the step-by-step instructions and create a three node cluster running on AWS EC2. The cluster is setup with Alluxio connected to Amazon S3 and has all the necessary applications and compute frameworks, such as Spark, installed. Sample applications like wordcount will be used as placeholders for business logic.

Deployment

The first step is to provision and deploy machines on AWS EC2. For this tutorial, we will use a deployment framework based off the one in Alluxio's open source repository. The version provided is preset with settings for deploying the cluster we want. This framework uses Vagrant and Ansible in order to provision machines and deploy Alluxio.

This tutorial may incur minor charges on your AWS account. If you use the preset settings, it will cost around $2 per hour.

AWS Prerequisites

1. You will need an AWS account. You can set one up by following the steps by going to the Amazon Web Services (AWS) page.
2. You will need an Amazon key-pair file for the us-east region. See Amazon's key-pair documentation for more details.
3. You will need an Amazon S3 Bucket. You can set one up by following the steps on Amazon's S3 Bucket documentation page.

Installing Vagrant and Ansible

1. First, download and install Vagrant.
2. Install the vagrant-aws plugin.

```
$ vagrant plugin install vagrant-aws
$ vagrant box add dummy https://github.com/mitchellh/vagrant-aws/raw/master/dummy.box
```

3. The other tool we will need is Ansible, version 1.9. It can be installed in a number of ways, including through pip.

```
$ sudo easy_install pip  // if you don't have pip installed
$ sudo pip install ansible==1.9.1
```

We recommend using Python version 2.7.

Launching the Machines

Extract the tarball provided. You can download the tarball here.

```
$ tar -xvf alluxio-deploy.tar.gz
$ cd alluxio-deploy
```

Next we will need to configure a few AWS values. First update two environment variables.

```
$ export AWS_ACCESS_KEY_ID=<Your access key>
$ export AWS_SECRET_ACCESS_KEY=<Your secret key>
```

Then create a key pair file in the us-east region called aws-east and ensure your aws key-pair file is available at ~/.ssh/aws-east.pem.

If you already have an existing key-pair you would like to use, modify conf/ec2.yml's Keypair and Key_Path fields appropriately.

Finally, open conf/ufs.yml and update the S3 Bucket to a bucket you have created. Be sure to exclude the scheme (s3-bucket and not s3://s3-bucket).

After completing the configuration steps, you are now ready to launch your cluster. This command may take a while, and the progress will be updated on your console. From the top level folder, run:

```
$ ./create 3 aws
```

A success message will be outputted at the end of the deployment with information about how to access your cluster.

Result

Virtual Cores: 8 / Node
Memory: 61 GB / Node
Network Speed: ~1 Gb / Node
Alluxio Memory: 54 GB / Node
Alluxio Version: 1.2.0
Spark Version: 2.0.0
Alluxio Master / Worker: 1/3 (default)
Spark Master / Worker: 1/3 (default)

Tip

In rare cases, the availability zone may not be applicable for your aws account. If this happens, modify conf/ec2.yml's Availability_Zone: 'us-east-1a' to another region like us-east-1b.

Using the Cluster

Congratulations! You have successfully deployed an Alluxio, Spark, and S3 stack on AWS. Now we are ready to explore the functionalities in the cluster and run some applications. Please execute the following steps on the AlluxioMaster node you deployed.

Web UI Access

You can browse your Alluxio deployment through a web browser at the Alluxio Master IP plus port 19999 (default value). The Web UI shows various information about the system. For example, you can browse the file system through the UI.

Command Line Interface

The CLI is more powerful than the web UI. You can ssh into the machines you have deployed with the following command:

```
$ vagrant ssh AlluxioMaster
```

Once you have logged into the deployed machines, you can find the Alluxio installation under /alluxio. Let's place a simple file into Alluxio which can be used later for computation.

```
$ /alluxio/bin/alluxio fs copyFromLocal /alluxio/LICENSE /file1
$ /alluxio/bin/alluxio fs ls /
```

Running an Application

Run the Spark shell.

```
$ /spark/bin/spark-shell
```

Let's do a simple line count of the file we copied into Alluxio earlier, file1.

```
scala> val file = sc.textFile("alluxio://AlluxioMaster:19998/file1")
scala> file.count()
```

Congratulations! You've run your first compute job with Spark and Alluxio on this on-demand cluster.

Tip

AlluxioMaster can be replaced with AlluxioWorker1 or AlluxioWorker2 if you wish to access the workers.

Tip

If you set your under storage bucket to a bucket which already contained data, you will discover that the data is available through Alluxio. This is because the existing metadata is automatically fetched the first time a list status operation is run on a folder. To fetch data on an old folder, use /alluxio/bin/alluxio fs ls -f /<path in Alluxio>. Data will automatically be fetched if you access the folder through the file browser in the web UI.

Tip

You can exit the spark-shell by entering :quit.

Accessing Remote Datasets

The first example used data which was on the local machine and copied to Alluxio, but in most cases you will want to do computation on data which lives in a long running persistent store. In this example, we will use Amazon S3 as our source of data.

In this example, applications in the on-demand cluster leverage Alluxio to transparently interact with data stored in Amazon S3. First, we will simulate full-scan data analytics workloads by using count. Then, we will transform the raw data into a more meaningful form, in this case we use wordcount. Finally, we will answer some questions about the processed dataset, such as what the most frequent words are. Throughout the examples, we will emulate multiple users or sessions accessing the same dataset and discover what kind of performance gains we can expect by using Alluxio.

Connecting Alluxio with Amazon S3

When you first initialized the cluster, an S3 bucket was specified as the backing store. This means any durable data would be written to the S3 bucket. Alluxio can also import durable data already in S3 into Alluxio, making it available for compute frameworks such as Spark.

The recommended way to do this is to use the mount operation. This connects a part of Alluxio's file system tree with a folder in an under storage system, allowing the user to interact with the data in the under storage system by referencing the mounted Alluxio path. In addition, any persisted data in the Alluxio path will be stored in the under storage system.

As an example, let's mount a public S3 bucket containing some sample datasets.

```
$ /alluxio/bin/alluxio fs mount -readonly /data s3a://alluxio-datasets/text
$ /alluxio/bin/alluxio fs ls -R /data
```

The dataset provided here, English Corpus, is made of randomly generated words from the English dictionary, totaling to about 150 GB of data.

If you would like to experiment with smaller datasets, you can mount s3a://alluxio-samples/datasets instead, but the performance characteristics may vary since the workload is negligible.

Tip

You may have noticed the ls command took a small amount of time. This is because it was the first access after mounting the remote data. Alluxio fetched the metadata for all the files that were in the remote bucket.

Accelerating Data Access with Alluxio

We can repeat the line count example with our 150GB English Corpus.

The first query will take a few minutes, since the data must be fetched from the remote storage.

```
scala>  val file = sc.textFile("alluxio://AlluxioMaster:19998/data/150g")
scala>  file.count()
```

Run the count query again, you will see much better performance, as the data is already stored in Alluxio and does not need to be queried from S3.

```
scala>  val file = sc.textFile("alluxio://AlluxioMaster:19998/data/150g")
scala>  file.count()
scala>  :quit
```

The previous result can also be accomplished by using Spark's persist API; however, the data will only be accessible to that particular Spark context. **Using Alluxio, if you restart the Spark shell or launch another Spark program and then access the same dataset, you will notice the speed up is still present**.

```
scala>  val file = sc.textFile("alluxio://AlluxioMaster:19998/data/150g")
scala>  file.count()
scala>  :quit
```

This is because Alluxio provides sharing between applications. Once data is in Alluxio space, Alluxio stores and manages it to optimize future data requests. In addition, **Alluxio consolidates the memory required to store the data, meaning applications (e.g. multiple Spark jobs) do not need to duplicate data in memory**. In cases where the datasets are large, this can significantly improve the performance of your workloads.

Tip

Be sure to re-evaluate the file variable so Spark gets the updated storage locations. Spark only evaluates this on RDD creation, so the original RDD will not be locality optimized to read from Alluxio. This only is a problem when RDDs are not being transformed, which is rarely the case.

Tip

You can turn on logging in spark-shell with the sc.setLogLevel("INFO") command. This will print out progress as well as timing information.

Managing Intermediate Data

Since we have a cluster up and running, we can try answering some questions about our dataset. Also, because we previously accessed the data, we will avoid any network transfers and gain the benefits of in-memory data access.

Assume we are interested in the frequency of each word in the dataset.

Although there is no need to fetch data from remote storage, the operation will still take around 10 minutes. This is because Alluxio has only stored the raw data, 150g, and in order to get the word counts, we must run word count, which is CPU bottlenecked. To avoid repeating expensive transformations like this, it is best to save the intermediate data to Alluxio.

```
scala>  val file = sc.textFile("alluxio://AlluxioMaster:19998/data/150g")
scala>  val counts = file.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _).coalesce(3)
scala>  counts.saveAsObjectFile("alluxio://AlluxioMaster:19998/intermediate/150g-counts")
```

Coalescing the resulting RDD will reduce the overhead of accessing it later. The output is very small compared to the input and will not benefit from being split into many partitions.

Now let's take a look at the top 10 most frequently occurring words and their frequencies. Note the re-evaluation of the counts RDD as intermediateData. We are now referencing the preprocessed intermediate data we previously stored in Alluxio.

```
scala>  val intermediateData = sc.objectFile[(String, Int)]("alluxio://AlluxioMaster:19998/intermediate/150g-counts")
scala>  intermediateData.map(k => (k._2, k._1)).top(10)
```

When reading an object file back from Alluxio, it is important to know what type it contains. In this case the key was the word, a String, and the value was its count, an int. You need to specify the correct type to regenerate the RDD.

Let's save those results and exit.

```
scala>  sc.makeRDD(intermediateData.map(k => (k._2, k._1)).top(10)).repartition(1).sortByKey(false).saveAsText
          File("alluxio://AlluxioMaster:19998/results/top10")
scala>  :quit
```

Imagine we need to do some more analysis on the data awhile later, or someone else would like to investigate the dataset. To simulate this, open up another spark-shell and compute the most frequently seen word starting with a.

```
scala>  val intermediateData = sc.objectFile[(String, Int)]("alluxio://AlluxioMaster:19998/intermediate/150g-counts")
scala>  intermediateData.filter(k => k._1.startsWith("a")).map(k => (k._2, k._1)).top(1)
```

Notice that even though we are using another spark-shell, the intermediate data can still be accessed. Any analysis which would be done on the wordcounts can directly start on the preprocessed data.

Spinning Down the Cluster

All on-demand clusters eventually will be shutdown. It is important to ensure you have persisted to durable storage any processed datasets you would like to keep. Alluxio provides several different mechanisms to save your data to the backing store.

Using the Persist Command

Alluxio's command line interface allows you to save a file in Alluxio to the backing under storage. In our deployment, this will save the file to the bucket you specified when you started the cluster, or if the file is under a mount point, to the mount point specified.

Let's save the top 10 results we gathered.

```
$  /alluxio/bin/alluxio fs persist /results/top10
```

Synchronous Persist

The downside of using the persist command is that the data can possibly be lost before you have a chance to persist it, for example if the cluster goes down, or Alluxio needs to free up some space to store new data.

In cases when you cannot afford to lose your data, Alluxio offers a cache-through write strategy which synchronously writes the data to the backing store. The downside of this strategy is that the write speed will be limited by the backing store's write throughput, which is often much slower than Alluxio's in-memory writes.

You can enable the cache-through strategy by updating an Alluxio property. Since Spark is the program which will invoke the Alluxio client, update the property file in Spark's configuration folder:

```
$  echo alluxio.user.file.writetype.default=CACHE_THROUGH >> /spark/conf/alluxio-site.properties
```

Then you should synchronize the files across the machines in the cluster. Alluxio provides a built in utility for this.

```
$ /alluxio/bin/alluxio copyDir /spark/conf
```

Now if you start a spark-shell or submit a new Spark job, you will use the cache-through write strategy.

Viewing Persistence

One quick way to see what has and has not been persisted is to look at the file browser in the web UI or use the command line tool to browse the file system tree. Each file will have a state associated with it, such as PERSISTED or NOT_PERSISTED.

Of course, you can also view your backing store to verify the file exists.

Shutting Down the Cluster

Depending on what kind of deployment tool you use to start the cluster, you may have different ways of shutting it down. If you used the tool presented in this paper, you can shut it down by running the destroy command on the machine you ran the create command from.

This will terminate the EC2 instances, so make sure you are really done with the machines before shutting down the cluster.

```
$ ./destroy
```

Timing Comparison

The final portion of the paper is a comparison between using Alluxio and directly using Amazon S3. The timing information is gathered from the logging in Spark. You can easily accomplish this as well by running the commands with a scheme for accessing S3 directly instead of going through Alluxio. When Alluxio is not present, Spark's MEMORY_AND_DISK persist mode is used.

Command	W/ Alluxio	Only S3
Initial count of 150GB file	451 seconds	520 seconds
Subsequent count of 150GB file	52	293
New Spark Session		
Count of 150GB file	51	528
Store wordcount data	710	831

Conclusion

Alluxio enables high performance on-demand clusters by providing the following benefits:

- Memory speed data access.
- Efficient data sharing between applications.
- Transparent data access to storage systems.
- Reduced memory footprint.

As shown, Alluxio brings up to 10x performance improvement even when using only AWS services. The performance benefits are larger in a hybrid cluster. Alluxio's mount allows for a clean and simple way to connect the compute cluster to numerous data stores. Alluxio removes key pain points from the on-demand cluster architecture, enabling this cost-effective and efficient solution to solve a wide range of problems.