ALLUXIO

Effective Spark Resilient Distributed Datasets (RDDs) with Alluxio

Gene Pang
Software Engineer at Alluxio

What's Inside

1 / Introduction

2 / Alluxio and Spark RDD Cache

3 / Saving RDDs

4 / Reading "Saved" RDDs

5 / Sharing "Saved" RDDs

6 / Conclusion

ARTICLE

1 / Introduction

Organizations like Baidu and Barclays have deployed Alluxio with Spark in their architecture, and have achieved impressive benefits and gains. Recently, Qunar deployed Alluxio with Spark in production and found that Alluxio enables Spark streaming jobs to run 15x to 300x faster. In their blog post, they described how Alluxio improved their system architecture, and mentioned that some existing Spark jobs would slow down or would never finish because they would run out of memory. Aer using Alluxio, those jobs were able to finish, because the data could be stored in Alluxio, instead of within Spark. In this blog, we investigate how Alluxio can make Spark more eective, and discuss various ways to use Alluxio with Spark. Alluxio helps Spark perform faster, and enables multiple Spark jobs to share the same, memory-speed data. We conducted a few simple and controlled experiments with Spark and Alluxio. For these experiments we used Spark version 2.0.0, and Alluxio 1.2.0.

In this article, we show by saving RDDs in Alluxio, Alluxio can keep larger data sets in-memory for faster Spark applications, as well as enable sharing of RDDs across separate Spark applications.

2 / Alluxio and Spark RDD Cache

A common way Spark users have greatly increased performance of their computations is to use the Spark RDD cache() API. This Spark API stores the RDD data in the Spark executors, so that the next time the RDD is accessed, the data can be served straight from memory. However, sometimes, storing the RDD data in Spark may not leave enough memory for the computation, since data can be very large and the amount of memory to allocate for data cannot always be accurately predicted. For example, in a previous blog, Qunar experienced some Spark jobs could not finish in a timely manner because they could never fit the data in memory. In addition, if jobs crash, the data saved in Spark will not persist, so the next access of the data will no longer be in memory.

An alternative is to store the RDD data in Alluxio. Spark jobs do not need to configure extra memory to store data, but only need enough memory to perform the computations on the data. Since Alluxio leverages memory for data storage, the RDD will still be in memory (in Alluxio). Also, if for whatever reason the job crashes, the data would still be in Alluxio memory, and available for subsequent access by the job.

Storing RDDs in Alluxio memory is very simple, and only requires saving the RDD as a file to Alluxio. Two common ways to save RDDs as files, `saveAsTextFile` and `saveAsObjectFile`, can be used with Alluxio. The saved RDDs in Alluxio can be read again (from memory) by using `sc.textFile` or `sc.objectFile`.

In order to understand how saving RDDs to Alluxio compares with using Spark cache, we ran a simple experiment. We used a single r3.2xlarge Amazon EC2 instance, with 61 GB of memory, and 8 cores. We ran both Spark and Alluxio in standalone mode on the node. For the experiment, we tried dierent ways of caching Spark RDDs within Spark, and dierent ways of saving RDDs in Alluxio, and measured how the various techniques aect performance. We also varied the size of the RDD to show how data size aects performance.

3 / Saving RDDs

There are several dierent ways to “save” or “cache” a Spark RDD. Spark provides the `persist()` API to save the RDD on dierent storage mediums. For these experiments we used:

`MEMORY_ONLY`: stores Java objects in the Spark JVM memory.

`MEMORY_ONLY_SER`: stores serialized java objects in the Spark JVM memory.

`DISK_ONLY`: stores the data on the local disk.

Here is a code example of saving an RDD using the `persist()` API.

```
rdd.persist(MEMORY_ONLY)
rdd.count()
```

In addition to the persist API, an alternative way to save RDDs is to write them out as files to Alluxio. The common APIs available are:

`saveAsTextFile`: writes the RDD as a text file, where each element is a line in the file.

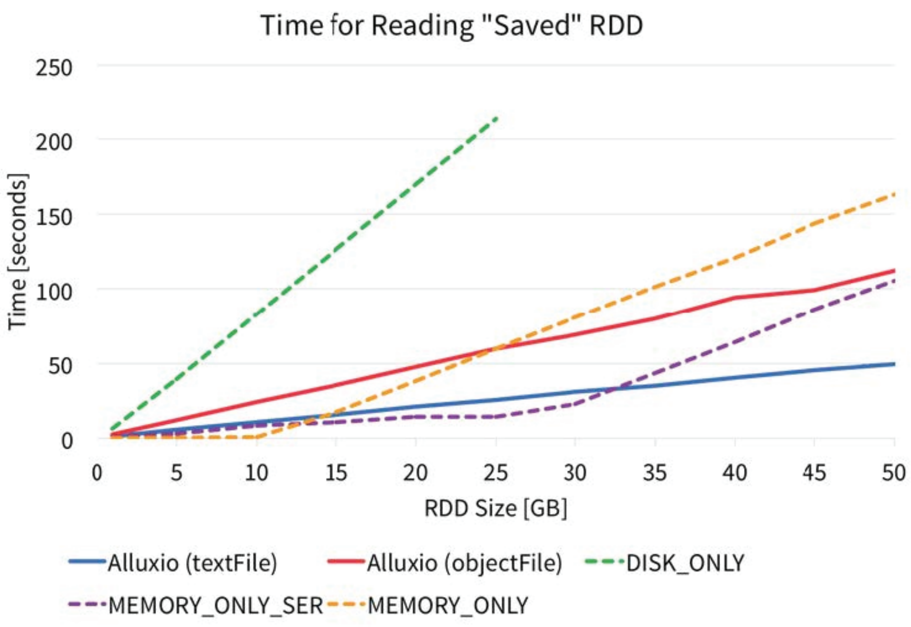
`saveAsObjectFile`: writes the RDD out to a file, by using Java serialization on each element.

Here is a code example of saving RDDs as files in Alluxio:

```
rdd.saveAsTextFile(alluxioPath)
rdd = sc.textFile(alluxioPath)
rdd.count()
```

4 / Reading “Saved” RDDs

Aer RDD are saved, they can be read to do subsequent computation. In our simple experiments we ran a `count()` on the cached or saved RDD. We measured the time it took to perform the `count()` on the previously saved RDD, and the figure below shows the time it took to complete the operation.



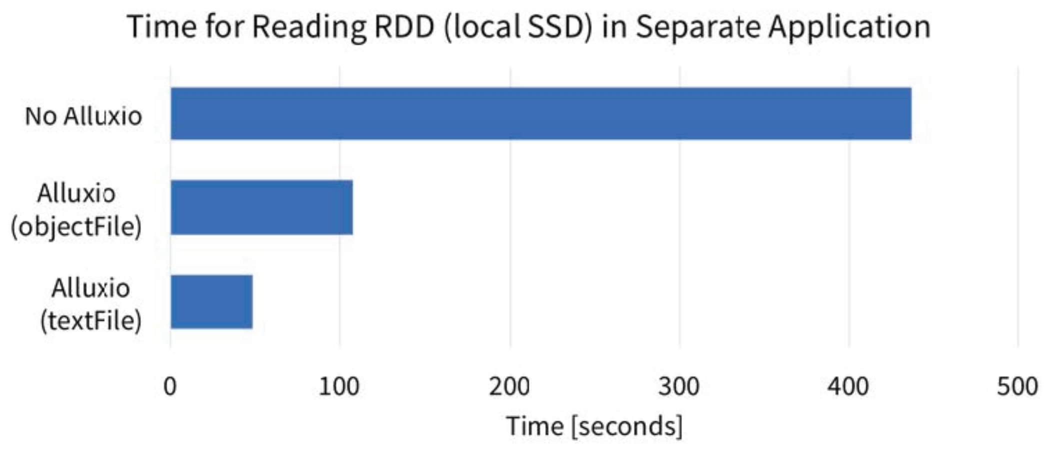
From the figure, it is clear that reading from an RDD saved in Alluxio results in very stable, and predictable performance. However, when persisting data within Spark, the performance is high for smaller data set sizes, but large data set sizes causes a significant decrease in performance. For instance, when using `persist(MEMORY_ONLY)` on a machine with 61GB of memory, when the data set size exceeds 10GB, the data no longer can fully fit in Spark memory, and the runtime slows down.

This figure also shows that when using `saveAsTextFile` with files in Alluxio memory, the performance is slower than using the Spark cache directly for smaller data set sizes. However, for larger data set sizes, reading the RDD from Alluxio files performs significantly better, because it scales linearly with the data size. Therefore, for a given size of memory for a node, Alluxio enables applications to process more data at memory speeds.

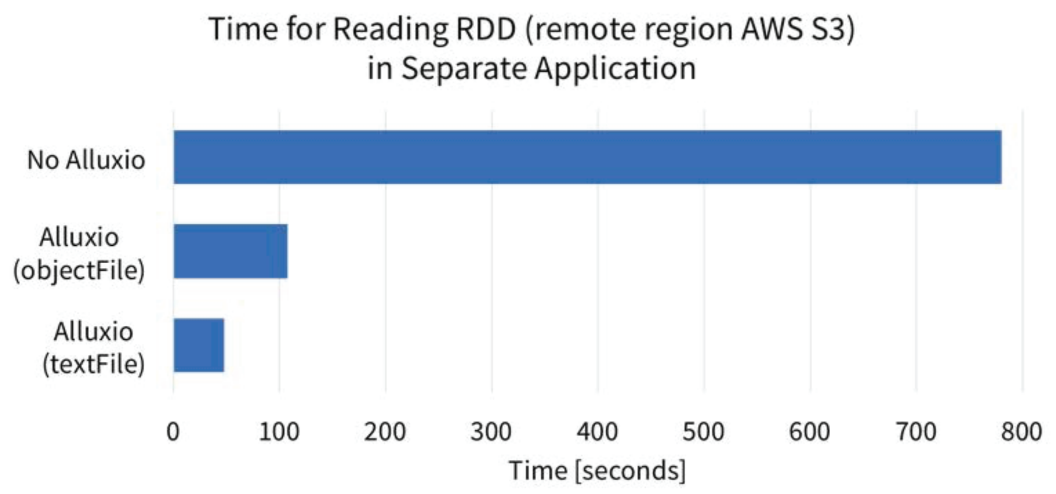
5 / Sharing Saved RDDs

Another benefit of Alluxio is the sharing of data in-memory. When saving files in Alluxio, the RDDs can be shared across dierent jobs and contexts, via Alluxio's memory. That means if an RDD is accessed frequently by many applications, they can all read it from the Alluxio file, and do not have to re-compute or re-fetch the data. Any Spark application can compute on the RDD from Alluxio memory.

To highlight the benefit of sharing the RDD via memory, we ran a simple experiment with the same setup as described earlier. With the 50GB data size, we ran a separate Spark job to compute on the RDD again. For Spark, this means the RDD has to read from the source again, and for this case, the source was the local SSD. However, when using Spark with Alluxio, the source of the RDD is a file in Alluxio, which is in Alluxio memory. Below we show the time it took for the second Spark application to run count() on the same RDD.



The results show that without Alluxio, Spark has to read the RDD from the source again, which is local SSD in this case. However, if the RDD was from a slower or remote source, it is likely that this is more expensive. When using Spark with Alluxio, the data is already in Alluxio memory, so Spark can operate on the RDD quickly. When Spark reads from Alluxio, it can process the RDD up to 4x faster. If the RDD is from a remote source, the performance improvement with Alluxio will be even more prominent. Below is is a figure showing the runtimes for when the RDD is in a remote Amazon AWS S3 bucket.



In this situation, the RDD data is not close to the computation, so takes longer to read the data again. However, when using Alluxio, the data is still in Alluxio memory, so the computation can complete quickly. In this example, using Alluxio sped up the RDD processing over 16x.

These experiments show that sharing RDDs via Alluxio can increase performance for multiple Spark applications that read the same data.

6 / Conclusion

This article presents the following values of running Alluxio with Spark. Saving RDDs as in-memory files in Alluxio enables Spark applications to complete in a predictable and performant manner. Alluxio allows larger data sets to be kept in-memory, resulting in faster Spark applications. Alluxio also enables sharing the same data sets in-memory with multiple Spark applications, which can improve the performance of the entire cluster.