Performance Evaluation of a JSON array unhashing component in a real-time big data streaming solution

Suman Bhagavathula
MSDS University of Washington
Senior Software Engineer, Microsoft

ABSTRACT

This paper applies two separate methods for performing string-based lookup for JSON array list values in a continuous stream of blobs in a distributed system. An additional constraint is that the lookup table may not always some of the values from the JSON array list. The speedup and scaleup of each of the methods is individually measured and reported. The performance comparison of the two methods is also presented. The first method uses the concept of broadcast join, where the lookup table is distributed to all the nodes and a join is performed on the source strings from the streaming blobs to the keys in the broadcast dataset. In this approach, the lookup is attempted even for values that may not be present in the lookup table and hence may not be very efficient. In the second method, a bloom filter is used to first estimate whether the lookup value is present in the lookup table and if only then perform the lookup. This lookup is also performed via broadcast join. The lookup is skipped if the bloom filter indicates that the value is not present in the lookup.

The first section introduces the problem and gives a naïve example to illustrate the scenario. The second section describes the two approaches, the third section provides information on the experiment setup and the final section presents the results and findings.

1. INTRODUCTION

Lookups are very popular in many kinds of applications. In general, lookups are required to be very fast. But there could be situations where lookups can be slow, especially when the lookup fields are long strings and the lookup table itself is huge. When such long string-based lookups on large files are required to be used in data streaming applications, the increase in latency could be significant. Thus, it is important to measure the performance of large-scale lookup operations and to optimize them. In this paper, I study the performance of large-scale string lookups in a distributed system using two separate techniques. For this purpose, the setting I used is a complex, near-real time big data collector service used in Microsoft for several analytical applications. This system has now been open sourced as Data Accelerator. This system is extensible and supports user defined functions (UDF). One of the UDFs is a hashing module that obfuscates the values of all the attributes defined by the users. These attribute values fall into three different categories: a) the values that are public domain b) the values that are non-public domain and c) the values that are a mixture of public and non-public domain on a case to case basis. It is the third scenario that is of interest for analysts and data scientists because they want to get the plain text information for the public domain attribute values but continue to see the hashed values for the non-public domain values. Since the values are hashed in the first place, there is another UDF called unhasher that is used to replace the hashes with known public domain values, where applicable. The list of these known public values is supplied to the unhasher as a lookup table.

Illustration with a Naïve example

Here I present a very naïve example to illustrate the lookup scenario.

Consider the table below that is used for lookup. This table contains two columns, *hash* and the corresponding *clear_text*. Note that the hash algorithm is not relevant here as this lookup is used to do for the reverse operation of replacing the clear text values in place where the hash values are seen.

Sample Lookup file:

Attribute Name	**Hash**	**clear_text**
names	hash string 1	clear text 1
Names	hash string 2	clear text 2
Location	hash string 1	clear text location 1

Location	hash string 3	clear text location 2

The incoming streams are JSON blobs with thousands of different attributes, some of these attributes contain the hash values. A sample blob structure is below, here the names and location attributes have hash values:

```
{
 "id":"123",
 "names":"hash string 1",
 "location":"hash string 5",
 "date": "11/20/2019 11:32 AM"
}
```

In some cases, some of the attributes values can be arrays represented as string of values separated by some delimiter character. An example is below, here the names attribute has an array value:

```
{
 "id":"234",
 "names":"hash string 1| hash string 2| hash string 3",
 "location":"hash string 5",
 "date": "11/20/2019 10:10 AM"
}
```

Below are the different scenarios that illustrate how the unhasher is supposed to work:

Case	**input**	**output**	**Comment**

1	{
	 "id":"123",
	 "names":"hash string 1",
	 "location":"hash string 5",
	 "date": "11/20/2019 11:32 AM"
	}
		{
		 "id":"123",
		 "names":"clear text 1",
		 "location":"hash string 5",
		 "date": "11/20/2019 11:32 AM"
		}
			Match found for names, replace with clear text
			Match not found for location, leave as-is

2	{
	 "id":"123",
	 "names":"hash string 3",
	 "location":"hash string 1",
	 "date": "11/20/2019 11:32 AM"
	}
		{
		 "id":"123",
		 "names":"hash string 3",
		 "location":"clear text location 1",
		 "date": "11/20/2019 11:32 AM"
		}
			Match not found for names, leave as is
			Match found for location, replace with clear text

3	{
	 "id":"345",
	 "names":"hash string 1",
	 "location":"hash string 1",
	 "date": "11/20/2019 11:32 AM"
	}
		{
		 "id":"345",
		 "names":"clear text 1",
		 "location":"clear text location 1",
		 "date": "11/20/2019 11:32 AM"
		}
			Match found for names, replace with clear text
			Match found for location, replace with clear text

4	{
	 "id":"345",
	 "names":"hash string 3",
	 "location":"hash string 5",
	 "date": "11/20/2019 11:32 AM"
	}
		{
		 "id":"345",
		 "names":"hash string 3",
		 "location":"hash string 5",
		 "date": "11/20/2019 11:32 AM"
		}
			Match not found for both names and location, leave the hashed values as-is

5	{ "id":"567",	{ "id":"567",	Matches found for some values in names, replace those with

	"names":"hash string 1| hash string 2| hash string 3", "location":"hash string 5", "date": "11/20/2019 10:10 AM" }	"names":"clear text 1| clear text 2| hash string 3", "location":"hash string 5", "date": "11/20/2019 10:10 AM" }	clear text and leave the other values as-is

Even though the examples presented above are very naïve, in reality the lookup tables are very huge, with several tens of thousands of entries. The incoming input stream of blobs is also very rapid, often at the rate of several hundred blobs per minute and each blob is about a couple of hundred megabytes. In addition, sometimes the attribute values to be looked up are arrays, and the unhasher needs to only perform unhashing of known values within the list while retaining the unknown values as-is. This means the number of times the lookup is performed increases several folds. The performance goal is to have the latency introduced by this new feature as minimum as possible such that the near-real time characteristic of the system is maintained.

System Specification

This system is running on a 122-node Microsoft HDInsight Spark cluster with 2 head nodes (one primary, one secondary) each with 16 cores, and 120 worker nodes with a total of 1920 cores. However, not all cores are allocated to the same job since it is a multi-tenant environment. Each tenant gets about 16 executors, each with 16GB memory. For the experiments, I vary the executor and memory count in three different configurations: first uses 8 executors and 8GB memory, the second uses 12 executors and 12GB memory and the third uses 16 executors and 16GB memory. The lookup file size is also varied from 36MB to 70MB to 140 MB in the different experiments to measure the speed up and scale up as described in the objectives section below.

The system is implemented in Scala.

Objectives

The objective of the study is to perform the performance impact of the JSON array unhasher feature. There are following scenarios that I would study:

1. With Broadcast join:

    a. How does the system performance be affected when the load (the lookup values) is increased but the system capacity is left untouched?
    b. How does the system performance look like when the load is kept constant but the system capacity is increased?
    c. How does the system performance be affected when the lookup values list is increased 2-fold and 4-fold while also increasing the system capacity accordingly?

2. How does the addition of bloom filter impact the performance? Ideally this is supposed to improve the performance. Does the reality match up with the expectations?

The following configurations are used for the experiments towards studying each of the objectives:

Objective	**Dataset size**	**System Capacity (executor cores, executor memory)**
1a. Broadcast join, increased load under constant system capacity	Small (36MB)	Large – (16,16000mb)
	Large (140MB)	Large – (16,16000mb)
1b. Broadcast join, increased load under constant system capacity	Small (36MB)	Small – (8,8000mb)
	Small (36MB)	Medium – (12,12000mb)
	Small (36MB)	Large – (16,16000mb)
1c. Broadcast join, increased load and system capacity	Small (36MB)	Small – (8,8000mb)
	Medium (70MB)	Medium – (12,12000mb)
	Large (140MB)	Large – (16,16000mb)
2. Comparing between Broadcast join alone vs. Bloom filter + Broadcast join	Large (140MB)	Large – (16,16000mb)

2. REVIEW OF TWO IMPLEMENTATIONS

In the first method, the spark's native broadcasting mechanism is used to distribute the lookup table to all the nodes. The advantage of this method is that the lookup dataset is transferred only once per worker node and not per each task. And then the worker nodes use this dataset for the lookup during the execution of the individual tasks. The disadvantage of this method is that the lookup is attempted for each value in the streaming blob, irrespective of whether the hash value is present in the lookup table.

In the second method, a bloom filter is first created with the lookup values and is used prior to the actual lookup operation to infer whether the hash value is in the lookup table. The lookup is skipped if the bloom filter reports negative (which means the value is not in the lookup). The lookup is only performed if the bloom filter indicates a positive (which means the value is in the lookup). Note that since the bloom filter may have false positives, there could be some cases where we still end up doing a lookup even though in reality the value is not in the lookup table.

Recollect from the naïve example that the lookup table has three columns: Attribute Name, Hash and the Clear Text. This has been implemented as a map of maps data structure in Scala. In other words, the Attribute Name is the key for the outer map, and for each Attribute value, there is a sub map for its value with the Hash as the key and the Clear Text as the value. In this setup, there are two options for the bloom filter implementation:

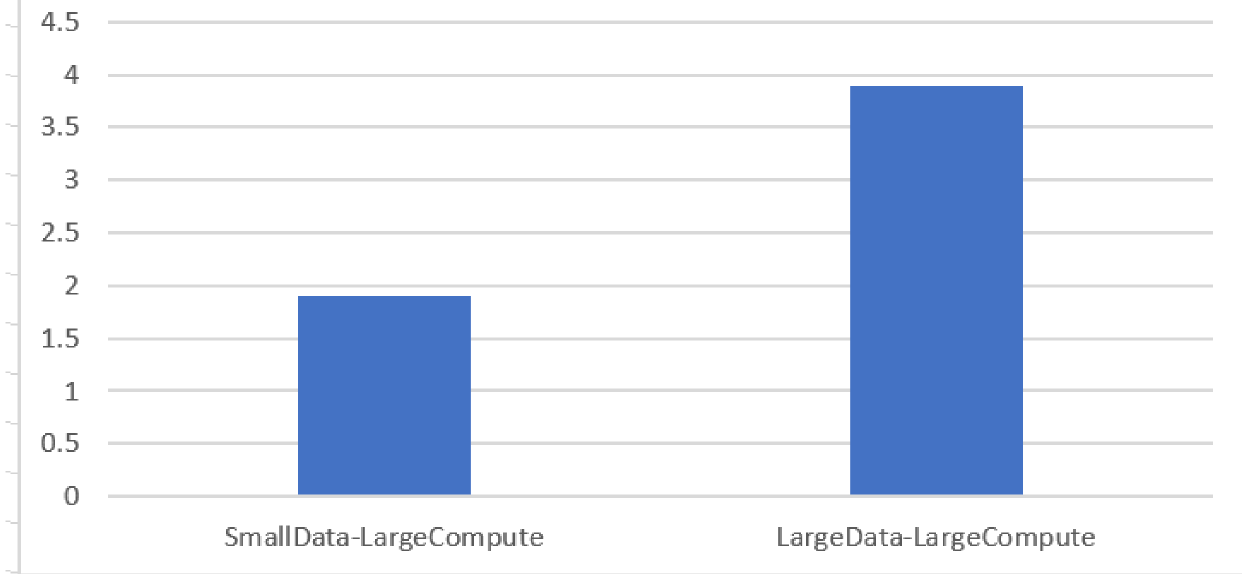
The first option is to create the bloom filter at runtime for each attribute that needs the lookup. In this case, the bloom filter is created on each of the worker nodes. This approach cannot take any advantage of the distributed system for the bloom filter creation for a specific attribute. This is not to say that the distributed system is not at all used, it is still used for processing of the different input streams. In other words, the Input streams, which are hundreds of blobs per minute, are distributed to all the available worker nodes for processing. Each worker node starts processing these blobs. When they reach the point where they need to unhash the different attribute values, they get the list of attributes that needs to be unhashed. For each attribute in this list, they create a bloom filter from the lookup table but just for the sub map of that attribute. This bloom filter is created on the specific worker node. And each worker node is creating multiples of these smaller bloom filters for the different sub maps.

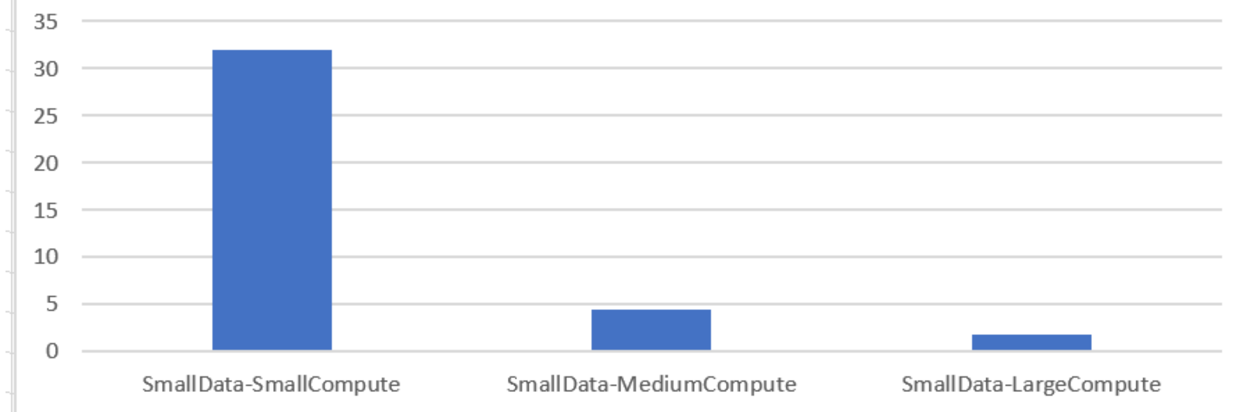
The other option is to create the bloom filter at startup time for all the attributes. The driver node can take advantage of the spark context to distribute lookup file to all the worker nodes to make the creation of the bloom filter happen in a distributed way. This bloom filter is then collected by the driver and distributed to all the worker nodes for use with the pre-lookup filtering operation.

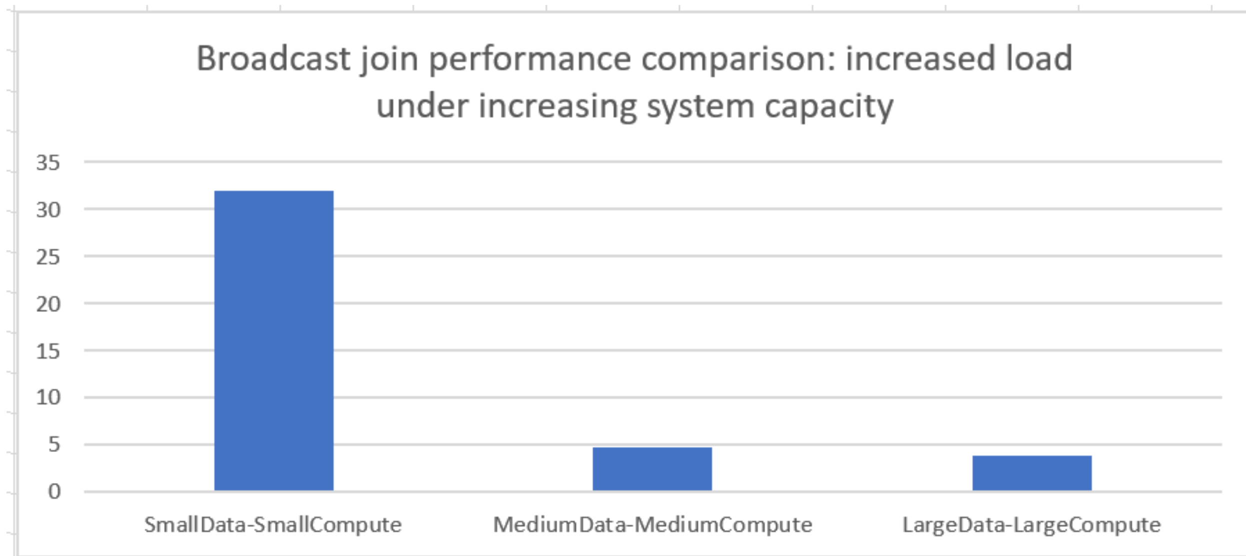
Which Bloom filter to use? There are many different implementations of bloom filter available in the open source community. There is also an option to implement the bloom filter ourselves. For the current work, for option 1 I chose the Bloom filter from the org.apache.spark.util.sketch[1] open source project. And for the second option, I chose the Bloom filter from the breeze.util[2] open source project since it supports distributed creation of bloom filter.

3. RESULTS AND ANALYSIS

For the Broadcast scale up and speed up experiments, below are the execution times observed under different data volume and compute capacities:


Broadcast join performance comparison: increased load under constant system capacity
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
SmallData-LargeCompute
LargeData-LargeCompute


Broadcast join performance comparison: constant load under increasing system capacity
35
30
25
20
15
10
5
0
SmallData-SmallCompute
SmallData-MediumCompute
SmallData-LargeCompute



Bloom + Broadcast:

With both the bloom filters-based approaches, I have observed deteriorated performance. The execution time has been consistently above 15 minutes for multiple runs with Large dataset and Large compute configuration (16 executor cores, 16000mb executor memory). This could be attributed to the following:
In the first option, since the bloom filter is being created at run time, there is additional time being taken for this operation. In the second option, even though the bloom filter is being created at the startup, it is limited by the driver resources since the driver needs to collect the entire lookup dataset to broadcast to the worker nodes for the bloom filter. In any distributed system, the driver resource allocation is typically low since the driver is only expected to perform the task coordination and nothing else.

4. FUTURE WORK

1. There is a research paper(3) I found that proposed some extensions to Bloom filters. I could explore some of these techniques to see if they improve the performance of the bloom-filter based approach.

5. REFERENCES

1. https://github.com/apache/spark/blob/master/common/sketch/src/main/java/org/apache/spark/util/sketch/BloomFilter.java
2. https://github.com/scalanlp/breeze/blob/master/math/src/main/scala/breeze/util/BloomFilter.scala
3. Song, Haoyu & Dharmapurikar, Sarang & Turner, Jonathan & Lockwood, John. (2005). Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. ACM SIGCOMM Computer Communication Review. 35. 181. 10.1145/1090191.1080114