CHAPTER 7

PERFORMANCE AND SCALABILITY OF APACHE FLINK OVER HADOOP MAPREDUCE
CHAPTER 7

PERFORMANCE AND SCALABILITY OF APACHE FLINK
OVER HADOOP MAPREDUCE

7.1 Introduction
7.2 Salient Features of FLINK
7.3 Experimental Setup
7.4 Algorithms Implemented
7.5 Experimental Analysis
7.6 Use Cases
    Summary

7.1 INTRODUCTION

The term Big Data is used to describe datasets so large and complex that traditional database applications prove to be inadequate to effectively and efficiently process it. Real-time applications may generate either structured or unstructured data. This in turn adds to the complexity of designing a generic real-time data processing engine. Big data is best characterized by the 4Vs [125] – Volume, Variety, Veracity and Velocity. As each dimension grows in magnitude, it becomes more and more taxing to process data. This has led to the emergence of a new data processing model – MapReduce [126].

MapReduce is a programming model for distributed computing systems. Unlike traditional data processing methods, MapReduce harnesses the power of parallel processing. The MapReduce algorithm contains two important tasks, namely Map and Reduce. The Map task tokenizes the input data set and maps each token to a relevant key. The Reduce task collects the output from a map phase and preforms an aggregate operation on the values identified by the same key.

Hadoop MapReduce [127] is a powerful and scalable data-processing engine that has been extensively used since the advent of the MapReduce paradigm. However, Hadoop MapReduce is traditionally a batch processing engine and is unsuited for real-time data processing [128]. The progression in technology heavily banks on the
success of real-time computing systems and the productivity of such systems is directly linked with its innate ability process data in real-time. Despite the presented scenario, not all applications require real-time processing. In fact, a single application is capable of generating both real-time and batch data. This leads to the need for a single data processing engine capable of handling both batch and real-time data. Also, Hadoop MapReduce fails be an effective solution in problems requiring iterative data processing (K-Means and most machine learning problems) and graph processing. Even in the case of simple batch processing, Hadoop MapReduce’s two stage model rigid model has become a bottleneck for achieving higher optimization.

Not discrediting the undeniable position held by Hadoop MapReduce in the Big Data community, the aim of this chapter is to highlight an alternative (but not a replacement) to Big Data processing and analytics. Having established the prevailing notion, we introduce Apache Flink [129], which is both a stream and batch data processor, as an alternative to Hadoop MapReduce. Flink extends the MapReduce paradigm to include several higher order operations thereby adding customizability to the otherwise rigid MapReduce model. Flink also provides a generic iterative runtime engine which is better described as a *Cyclic Data Flow Engine* [130]. It is popularly believed that Flink bears the promise to usher in a new era in Big Data processing and analytics.

### 7.2 SALIENT FEATURES OF FLINK

The following section gives a deep insight into prominent architectural advancement found in Apache Flink.

**Common Runtime – Stream and Batch Processing:** Flink offers a common runtime environment for stream and batch data processing. In fact, Flink is typically a stream data processor which treats batch data as a special case of streaming data [131]. This is in contrast to how most data processing engines treat streaming data as micro-batches. This innate ability to deal with streaming data makes Flink capable of dealing efficiently with real time data.
Active Memory Management: Several data processing engines (including Hadoop MapReduce and Flink) are implemented over Java. The major concern with any JVM based implementation is efficient management of the heap. All processor intensive tasks run in memory and hence the datasets to be processed must be present in memory before being operated upon. However, the size of the main memory is often much less than the size of the dataset leading to OutOfMemoryErrors. Another major drawback associated with JVM based engines are the stalls incurred due to garbage collection. Overhead spent in the garbage collection of several objects can be take a toll on overall system throughput. Moreover, java objects have some amount of overhead space which depletes the amount of overall memory available.

Apache Flink combats these problems associated with memory management using the concept of serialization [132]. Instead of burdening the heap, Flink serializes objects into a fixed number of pre-allocated memory segments. If data to be processed exceeds the size of the available memory, the serialized objects are spilled to the disk (Refer figure 7.1). When the need for these objects surfaces, they are de-serialized and brought back to memory. Moreover, the binary representation of objects uses far less memory. The problem associated with garbage collections is dealt with by reusing short-lived objects.

![Active Memory Management in Flink](image)

**Figure 7.1:** Active Memory Management in Flink

Program Optimizer: Program written in Flink, are not directly executed. Before execution, the job enters an intermediary Cost-Based Optimization Phase. In this phase, the Flink Optimizer, chooses the most optimum route for execution based on
the dataset and the nature of the cluster. This feature makes it possible for the programmer to focus on the code and not the execution environment or the input dataset. This ensures increased productivity of the programmer and enhanced utilization of the cluster.

Pipelined Dataflow: Pipelining is a major reason behind Flink’s low latency and high throughput [133]. The pipeline stages in a Flink are capable of consuming partial outputs. With pipelining, concurrency is achieved at each node in the cluster. Management of cluster resources to achieve this concurrency happens dynamically. Figure 7.2 depicts the pipelined dataflow in Flink. Here, X and Y represent operator acting on the datasets. Each operator is capable of receiving inputs from multiple sources.

![Pipelined Dataflow in Flink](image)

For the purpose of performance analysis, it is essential to identify a suitable benchmark for comparison. TeraSort [134] is a popular benchmark, used to measure the performance of data processing jobs. TeraSort aims to measure the duration of time required to sort a randomly distributed dataset for a given cluster using a particular data processing engine. For the purpose of generating the datasets, we make use of TeraGen, which is nothing but a MapReduce job without a reduce phase. A list of key-value pairs is generated during the map phase which is then written to HDFS. The sorting program is written in both MapReduce and Flink. The generated data is feed in as input and the sorted output is written back to HDFS.
7.3 EXPERIMENTAL SETUP

The cluster comprises of nine nodes (one master and eight slaves). Each node has the following specifications –

A) Hardware

Processor : Intel i5 4460
No. of Cores : 4 @ 3.2 GHz
RAM : 8 GB
Disk : 1 TB SATA
Cache : 6 MB

B) Software

Operating System : Ubuntu 14.04 Desktop
Apache Hadoop : 1.2.1
Apache Flink : 0.9.0
JAVA : OpenJDK 7

7.4 ALGORITHM IMPLEMENTED

1. \textit{main} ()
2. input\_file \leftarrow \text{read from HDFS}
3. map (input\_file) // 32 mappers
4. partition(map\_out)
5. reduce (partition\_out) // 32 reduces
6. HDFS \leftarrow \text{write output\_file}
7. end main
8. function \textit{map(input\_file)}
9. for each entry in input\_file
10. tokenize into key/value pairs
11. map\_out \leftarrow \text{collect key/value pairs}
12. end for
The datasets generated by TeraGen is fed as input into an environment setup with Flink and Hadoop MapReduce. The source code to achieve sorting is developed for both data processing engines. The map phase tokenizes the entries into key-value pair. Our objective is to sort by keys. The value associated with each key serves as a unique identifier. The output from the map phase is feed to a custom partitioner. The partitioner equally distributes the keys in a certain continuous range among the various reducers. This phase is responsible for the distributed nature of the sort. Each reducer sorts a certain sequential range of entries received from the partitioner and writes the output data back to HDFS.

#### 7.5 EXPERIMENTAL ANALYSIS

Let $P$ be the performance measure of the execution of a job $J$ on a cluster of $N$ nodes. The performance $P$ of a data processing engine to execute the job $J$ to completion is dependent on efficient and optimal usage of the following system resources in each node [135].

- **CPU ($\alpha$):** The processing ability of a node determines the number of parallel tasks supported by the cluster. Higher the CPU utilization, higher is the rate of execution. This can be expressed as follows
  \[
  P \propto \alpha \quad \text{(1)}
  \]

- **Memory ($\beta$):** Any currently processing job, must be in main memory. Despite having sufficient processing power, if the main memory is underutilized, CPU cycles would be expended in retrieving data from the disk (disk reads is
significantly slower than memory reads). Hence higher memory utilization is a measure of higher performance.

\[ P \propto \beta \quad (2) \]

- Disk I/O (\( \mu \)): However, it is often the case that the size of the main memory is insufficient to hold the entire dataset. In such cases, data is temporarily spilled onto the disk and read whenever the need arises. Higher optimality in performing disk operations ensure higher rate of execution.

\[ P \propto \mu \quad (3) \]

- Network (\( \lambda \)): In a distributed environment, data is transferred among intermediary nodes before reaching completion. This data transfer happens over the local network. Higher optimality and utilization of network resources results in a higher rate of job execution.

\[ P \propto \lambda \quad (4) \]

From the above observations, it can be inferred that the performance of a data processing engine running on a node is given by the following equation,

\[ P = \sum_{i=1}^{N} (\alpha C_1 + \beta C_2 + \mu C_3 + \lambda C_4) \quad (5) \]

where \( C_1, C_2, C_3 \) and \( C_4 \) are proportionality constants. If the cluster is composed of \( N \) identical nodes, the overall performance \( P' \) of the cluster is given by –

\[ P' = N (\alpha C_1 + \beta C_2 + \mu C_3 + \lambda C_4) \quad (6) \]

Also,

\[ \alpha + \beta + \mu + \lambda = 1 \quad (7) \]

This implies that we focus our analysis to strictly the aforementioned parameters. Four data-point sets have been used to test the performance of MapReduce and Flink. The results are tabulated and corresponding plots of time (in sec) against data-points (in MB) are obtained. The experiment concludes with the distributed sorting of 1 TB of data.

It is observed that Flink outperforms MapReduce over all datasets. At an outset, it can be stated that Flink performs exceptionally better on smaller datasets (Refer Plot-I in figure 7.3 and Plot-II in figure 7.4). However, as the size of the dataset grows, the
performance gap between MapReduce and Flink approaches a constant value (Refer Plot-III in figure 7.5 and Plot-IV in figure 7.6). This noticeable difference can be attributed to Flink’s active memory management component. If the dataset fits completely into memory, Flink discards the need to spill the contents on the disk. Therefore, if the size of the dataset is small, the overhead due to disk I/O is removed. Moreover, since Flink is essentially a data stream processor, the size of the stream would seldom exceed the size of the available cluster memory.

![Figure 7.3: Plot – I (Time vs. Datasets)](image1)

![Figure 7.4: Plot – II (Time vs. Datasets)](image2)

![Figure 7.5: Plot – III (Time vs. Datasets)](image3)

![Figure 7.6: Plot – IV (Time vs. Datasets)](image4)

To obtain a comprehensive conclusion, a closer inspection of the core parameters must be performed. For this purpose, we closely analyse a Terasort job for dataset of size 200 GB.
7.6 EXPERIMENTAL ANALYSIS

Flink features a pipelined processing model which reduces the need to materialize intermediate results onto the local or distributed file systems. This is evident from the spikes in the CPU utilization seen in figure 7.7. MapReduce, however, does not materialize intermediate results and consequently the CPU utilization is fairly constant (Refer Plot V in figure 7.8). A pipelined dataflow ensure concurrent CPU usage resulting in higher performance and throughput. As and when data is available at the slave nodes, worker processes are spawned resulting in localized spikes.

![Figure 7.7: Plot – V: CPU Utilization – MapReduce](image)

![Figure 7.8: Plot – VI: CPU Utilization – Flink](image)
**Analysis of Memory Utilization:** In MapReduce, a transformation takes most two phases: a map phase and a reduce phase. Between these phases, the results are spilled to disk. If multiple transformations are present, they are stacked up. Consequently, memory utilization is low (Refer Plot VII in figure 7.9).

Flink tries to combine all transformation into a single job. If a job is started, all transformations are performed in these jobs and intermediate results are kept in memory. This accounts for the higher memory utilization as seen in figure 7.10. By reducing the number of disk accesses made, CPU cycles are utilized primarily for data processing.

![Figure 7.9: Plot – VII: Memory Utilization – MapReduce](image1)

![Figure 7.10: Plot – VIII: Memory Utilization – Flink](image2)
Analysis of Network Utilization: Flink is equipped with an inline program optimizer. All jobs submitted are optimized and the most efficient route for execution is chosen. On comparing figure 7.11 and figure 7.12, it is observed that Flink shows a reduced network activity. Since the route for execution is optimized pre-execution, the network transfers are significantly reduced consequently reducing resulting overhead. Moreover, the network bottleneck existing during the shuffle phase is substantially reduced owing the in-memory transfers in Flink.

![Figure 7.11: Plot – IX: Network Utilization – MapReduce](image1)

![Figure 7.12: Plot – X: Network Utilization – Flink](image2)

7.7 USE CASES

Having described the potential of Apache Flink as prime big data analytics framework, in this section we describe plausible use cases of Apache Flink.
• **Real-Time Video Analytics:** Video analytics is a software implementation to analyze recorded footage and derive statistical and operational information. Flink being a stream data processing engine can be used to process and analyze real-time footage. Moreover, Flink’s iterative runtime and machine learning API’s, makes it possible implement learning algorithms on the top of the raw video stream libraries [136].

• **Data-Critical Systems:** In a data critical system, performance of the data processing engine is paramount. Better the performance of the data processing engine, higher is the overall throughput and consequently, factors like power consumption and heat generation reduce substantially. Flink’s in-memory capabilities coupled with its active memory management makes Flink a forerunner in data intensive applications.

• **Development:** Flink offers an extensive dataflow visualization which plays a vital role in the pre-deployment phases of a data processing job. The details of ordering of various stages in a job and the effect on data transformations immensely aid in debugging. A Flink job can be run locally on a development environment consisting of a single machine. Once the source code is ready and tested, the same job can be run on production environment spanning several hundred nodes without any changes to the source code.

**SUMMARY**

As the complexity and magnitude of data is constantly increasing, it is paramount that the technique with which data is processed also evolves. The Apache Flink is structured to enhance the manner in which data is processed. Flink extends the MapReduce paradigm and is modelled around a cyclic dataflow paradigm which provides an iterative runtime environment. Moreover, this runtime is common to both batch and streaming data. Flink is therefore more versatile in comparison to Hadoop MapReduce. Architectural advancements in Flink – an active memory management, a pipelined dataflow, an inline program optimizer, an iterative runtime routine to enhance performance of data processing and decrease the expected latency. These observations prove to make Flink a forerunner in future of data processing.