iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 5 **Issue:** IX **Month of publication:** September 2017

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor:6.887
Volume 5 Issue IX, September 2017- Available at www.ijraset.com

A Study on Speeding up Spark with Big Data Compression through Xeon/FPGA in VMware Virtualization Communication Model

R.Prem Kumar[1], Padmavathi.S[2], Boopathy.P[3]
[1]*Research Scholar of computer science, Maruthupandiyar College.*
[2]*Head of the Department, Computer Science, Maruthupandiyar College.*
[3] *Curriculum Developer, Panacea Research Midway.*

Abstract: ***Cloud computing provides a promising platform for big sensing data processing and storage as it provides a flexible stack of massive computing, storage, and software services in a scalable manner. Due to the high volume and velocity of big sensing data, traditional data compression techniques lack sufficient efficiency and scalability for data processing. Based on specific on-Cloud data compression requirements, In addition to different compression technologies and methodologies, selection of a good data compression tool is most important. There is a complete range of different data compression techniques available both online and offline working such that it becomes really difficult to choose which technique serves the best.***
In this survey, we characterize the Spark framework and also discuss the open issues and challenges raised on parallel data compression analysis with Spark. We propose a novel scalable Apache Spark with Xeon/FPGA data compression approach in VMware virtualization communication model.
Keywords: ***Data compression, Xeon/FPGA, VMware model, Spark, Hadoop.***

I. INTRODUCTION

One technique to use our storage more optimally is to compress the files. By taking advantage of redundancy or patterns, we may be able to "abbreviate" the contents in such a way to take up less space yet maintain the ability to reconstruct a full version of the original when needed. Such compression could be useful when trying to cram more things on a disk or to shorten the time needed to copy/send a file over a network.

A. Apache Spark

In order to store, process and analyze terabytes and even petabytes of such information, one needs to put into use big data frameworks. In this blog, offering an insight and analogy between two such very popular big data technology - Apache Spark [22].
Spark was optimized to run in memory, helping process data far more quickly than alternative approaches like Hadoop's MapReduce,which tends to write data to and from computer hard drives between each stage of processing. Its proponents claim that Spark running in memory can be 100 times faster than Hadoop MapReduce, but also 10 times faster when processing disk-based data in a similar way to Hadoop MapReduce itself. This comparison is not entirely fair, not least because raw speed tends to be more important to Spark's typical use cases than it is to batch processing, at which MapReduce-like solutions still excel.
Spark is a multi-stage RAM-capable compute framework with libraries for machine learning, interactive queries and graph analytics. Spark is a general-purpose data processing engine [23], suitable for use in a wide range of circumstances. Interactive queries across large data sets, processing of streaming data from sensors or financial systems, and machine learning tasks tend to be most frequently associated with Spark. Spark is often used alongside Hadoop's data storage module, HDFS, but can also integrate equally well with other popular data storage subsystems such as HBase, Cassandra, MapR-DB, MongoDB MapR File System, Cassandra, OpenStack Swift, Amazon S3, Kudu, and Amazon's S3.

II. LITRATURE REVIEW

Some techniques have been proposed to process big data with traditional data processing tools such as database, traditional compression, machine learning, or parallel and distributed system.

Shannon – Fano Coding: Shannon – Fano is very simple to implement but may not give the best compression ratio. Basically, it takes the source messages a(i) and their probabilities P(a(i)) and lists them in the order of decreasing probability. This list is then divided into two groups with nearly equal total probabilities. The messages in first group are given 0 as prefix code and
the messages in the second half of the list are given 1 as the prefix code. Each of the subgroups are iteratively divided according to the same criteria and the prefix codes are appended to them until each subset contains only one message. Thus, Shannon – Fano yields a minimal prefix code.

A. Static Huffman Coding

Huffman compression [21] is a primitive data compression scheme invented by Huffman in 1952. The Huffman compression algorithm is named after its inventor, David Huffman, formerly a professor at MIT. This code has become a favorite of mathematicians and academics, resulting in volumes of intellectual double-talk. Huffman's patent has long since expired and no license is required. There are however many variations of this method still being patented. The code can easily be implemented in very high speed compression systems. The Huffman code assumes "prior knowledge" of the relative character frequencies stored in a table or library. A secret table made available only to authorized users can be used for data encryption. A more sophisticated and efficient lossless compression technique is known as "Huffman coding", in which the characters in a data file are converted to a binary code, where the most common characters in the file have the shortest binary codes, and the least common have the longest.

B. Apache Hadoop

Hadoop is an open-source framework that allows to store and process big data in a distributed environment across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. This brief tutorial provides a quick introduction to Big Data, MapReduce algorithm, and Hadoop Distributed File System, hadoop performance shown in fig no [1] flowingly.

C. Audience

This tutorial has been prepared for professionals aspiring to learn the basics of Big Data Analytics using Hadoop Framework and become a Hadoop Developer. Software Professionals, Analytics Professionals, and ETL developers are the key beneficiaries of this course.

D. Prerequisites

Before you start proceeding with this tutorial, we assume that you have prior exposure to Core Java, database concepts, and any of the Linux operating system flavors. Due to the advent of new technologies, devices, and communication means like social networking sites, the amount of data produced by mankind is growing rapidly every year. The amount of data produced by us from the beginning of time till 2003 was 5 billion gigabytes. If you pile up the data in the form of disks it may fill an entire football field. The same amount was created in every two days in 2011, and in every ten minutes in 2013. This rate is still growing enormously. Though all this information produced is meaningful and can be useful when processed, it is being neglected.

E. Algorithm

1) Generally MapReduce paradigm is based on sending the computer to where the data resides!
2) MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
a) *Map stage:* The map or mapper's job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.

b) *Reduce stage:* This stage is the combination of the Shuffle stage and the Reduce stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.
3) During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
4) The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the node
5) Most of the computing takes place on nodes with data on local disks that reduces the network traffic.

6) After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.

II. DATA COMPRESION USING APACHE SPARK THROUGH XEON/FPGA

Spark is a top-level project of the Apache Software Foundation, designed to be used with a range of programming languages and on a variety of architectures. Spark's speed, simplicity, and broad support for existing development environments and storage systems make it increasingly popular with a wide range of developers, and relatively accessible to those learning to work with it for the first time.

A. Big Data Compression Through Apache Spark

Apache Spark can be considered as an integrated solution for processing on all architecture layers. It contains Spark Core that includes high-level API and an optimized engine that supports general execution graphs, Spark SQL for SQL and structured data processing, and Spark Streaming that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Definitely, batch processing using Spark might be quite expensive and might not fit for all scenarios and data volumes.

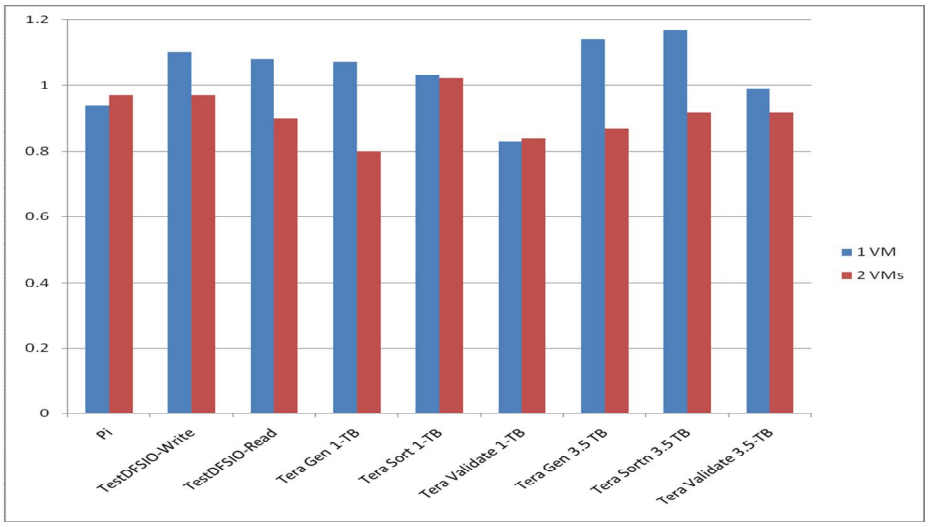


Fig 1: Performance of Hadoop for several workloads

For example: The data files are text files with one record per line in a custom format. Files contain from 10 to 40 million records. In the tests that follow, I used a 14.4 GB file containing 40 million records.The files are received from an external system, meaning we can ask to be sent a compressed file but not more complex formats (Parquet, Avro…).Finally, Spark is used on a standalone cluster (i.e. not on top of Hadoop) on Amazon AWS. Amazon S3 is usually used to store files.

B. Loading data into spark dataframes

The first case walks through loading and querying tabular data. This example is a foundational construct of loading data in Spark. This will enable to understand how Spark gets data from disk, as well as how to inspect the data and run queries of varying complexity.

First, we will import some packages and instantiate a sqlContext, which is the entry point for working with structured data (rows and columns) in Spark and allows the creation of DataFrame objects.

C. Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor:6.887
Volume 5 Issue IX, September 2017- Available at www.ijraset.com

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.
Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

D. Create An Rdd

First, we have to read the input file using Spark-Scala API and create an RDD as shown in fig no [2]. The following command is used for reading a file from given location. Here, new RDD is created with the name of input file.

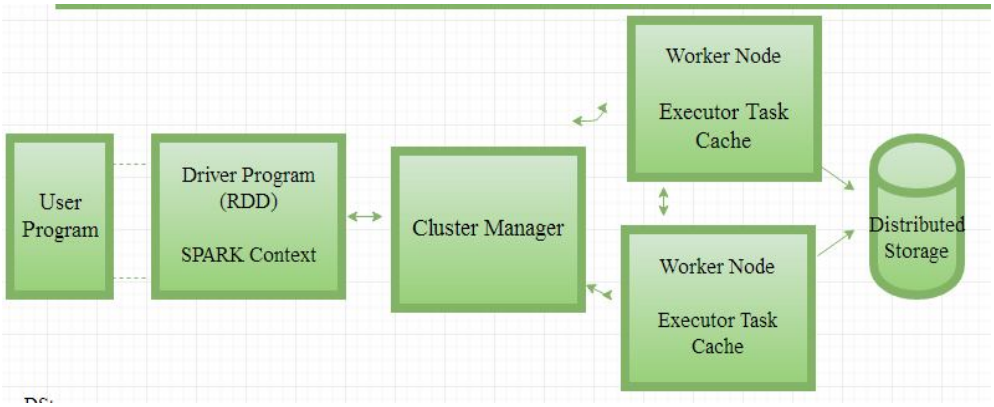


Fig 2: Spark Driver program with RDD work

The String which is given as an argument in the text File (“”) method is absolute path for the input file name. However, if only the file name is given, then it means that the input file is in the current location. ***Scala> Val inputfile = sc.textFile ("input.txt").***

E. Data compression coding using spark:

```
public class ExampleJob {
private static JavaSparkContext sc;
public ExampleJob(JavaSparkContext sc){
this.sc = sc;
   }
public static final PairFunction<Tuple2<Integer, Optional<String>>, Integer, String> KEY_VALUE_PAIRER =
   new PairFunction<Tuple2<Integer, Optional<String>>, Integer, String>() {       public Tuple2<Integer, String> call(
        Tuple2<Integer, Optional<String>> a) throws Exception {
    // a._2.isPresent()
      return new Tuple2<Integer, String>(a._1, a._2.get());       }   };
 public static JavaRDD<Tuple2<Integer,Optional<String>>> joinData(JavaPairRDD<Integer, Integer> t, JavaPairRDD<Integer, St
ring> u)
{
JavaRDD<Tuple2<Integer,Optional<String>>>leftJoinOutput=leftOuterJoin(u).values().distinct();
return leftJoinOutput;
}
publicstaticJavaPairRDD<Integer,String>modifyData(JavaRDD<Tuple2<Integer,Optional< String>>> d)
{
   return d.mapToPair(KEY_VALUE_PAIRER);
}
public static Map<Integer, Object> countData(JavaPairRDD<Integer, String> d){ Map<Integer, Object> result = d.countByKey();
   return result;
}
public static JavaPairRDD<String, String> run(String t, String u)
```

```
{
JavaRDD<String> transactionInputFile = sc.textFile(t);
JavaPairRDD<Integer, Integer> transactionPairs = transactionInputFile.mapToPair(new PairFunction<String, Integer, Integer>()
{
public Tuple2<Integer, Integer> call(String s)
{
 String[] transactionSplit = s.split("\t");
return new Tuple2<Integer, Integer>(Integer.valueOf(transactionSplit[2]), Integer.valueOf(transactionSplit[1]));
} });
JavaRDD<String> customerInputFile = sc.textFile(u);
JavaPairRDD<Integer, String> customerPairs = customerInputFile.mapToPair(new PairFunction<String, Integer, String>()
{
public Tuple2<Integer, String> call(String s)
{
      String[] customerSplit = s.split("\t");
      return new Tuple2<Integer, String>(Integer.valueOf(customerSplit[0]), customerSplit[3]);
}});
Map<Integer, Object> result = countData(modifyData(joinData(transactionPairs, customerPairs)));
List<Tuple2<String, String>> output = new ArrayList<>();
 for (Entry<Integer, Object> entry : result.entrySet()){
output.add(new Tuple2<>(entry.getKey().toString(), String.valueOf((long)entry.getValue())));
}
JavaPairRDD<String, String> output_rdd = sc.parallelizePairs(output);
return output_rdd;
}
public static void main(String[] args) throws Exception
{
JavaSparkContext sc = new JavaSparkContext(new SparkConf().setAppName("SparkJoins").setMaster("local"));
ExampleJob job = new ExampleJob(sc);
JavaPairRDD<String, String> output_rdd = job.run(args[0], args[1]);
output_rdd.saveAsHadoopFile(args[2], String.class, String.class, TextOutputFormat.class);
sc.close();
}}
```

F. Rdd Operations

Operations on RDDs are divided into several groups:

1) Transformations
2) apply user function to every element in a partition (or to the whole partition)
3) apply aggregation function to the whole dataset (groupBy, sortBy)
4) introduce dependencies between RDDs to form DAG
5) provide functionality for repartitioning (repartition, partitionBy)
6) Actions
7) trigger job execution
8) used to materialize computation results
9) Extra: persistence
10) explicitly store RDDs in memory, on disk or off-heap (cache, persist
11) checpointing for truncating RDD lineage

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor:6.887
Volume 5 Issue IX, September 2017- Available at www.ijraset.com

Here's a quick recap on the execution workflow before digging deeper into details: user code containing RDD transformations forms Direct Acyclic Graph which is then split into stages of tasks by DAGScheduler is described by the following figure No [3].
So basically any data processing workflow could be defined as reading the data source, applying set of transformations and materializing the result in different ways. Transformations create dependencies between RDDs and here we can see different types of them.

G. Splitting Dag Into Stages

1) RDD operations with "narrow" dependencies, like map() and filter(), are pipelined together into one set of tasks in each stage operations with shuffle dependencies require multiple stages (one to write a set of map output files, and another to read those files after a barrier).
2) In the end, every stage will have only shuffle dependencies on other stages, and may compute multiple operations inside it. The actual pipelining of these operationshappens in the RDD.compute() functions of various RDDs

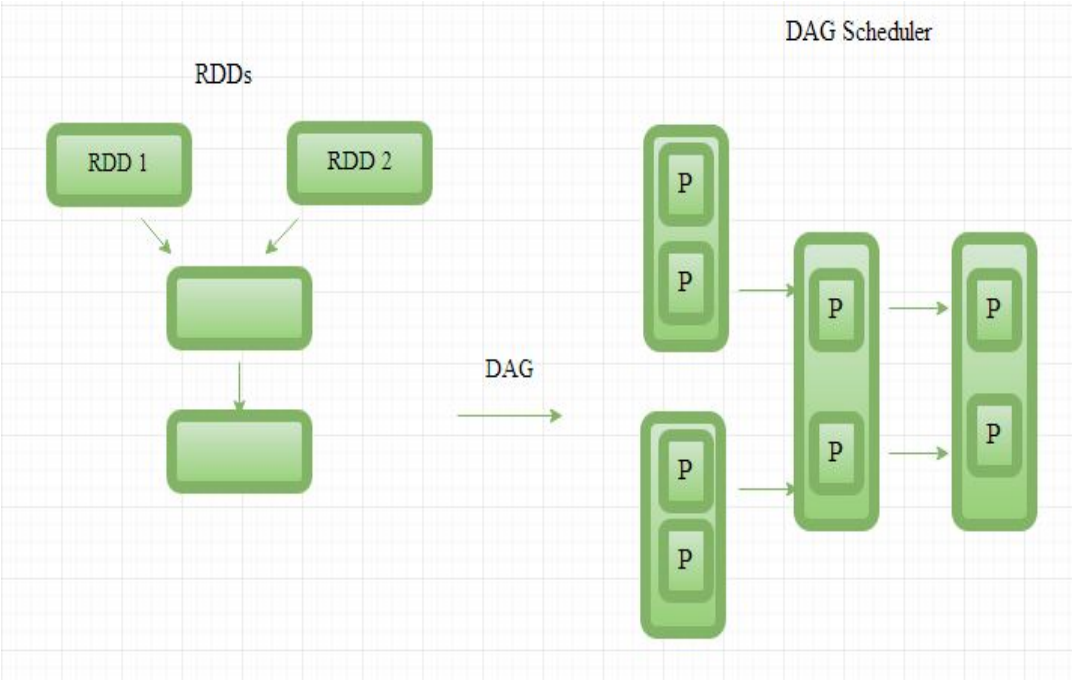


Fig No 3: RDD data transformation into DAG scheduler

H. Shuffle

During the shuffle ShuffleMapTask writes blocks to local drive, and then the task in the next stages fetches these blocks over the netwok.

1) Shuffle Write
a) redistributes data among partitions and writes files to disk
b) each *hash shuffle* task creates one file per “reduce” task (total = MxR)
c) sort shuffle task creates one file with regions assigned to reducer
d) sort shuffle uses in-memory sorting with spillover to disk to get final result
2) *Shuffle Read*
a) fetches the files and applies reduce() logic
b) if data ordeing is needed then it is sorted on “reducer” side for any type of shuffle

In Spark Sort Shuffle is the default one since 1.2, but Hash Shuffle is available too.

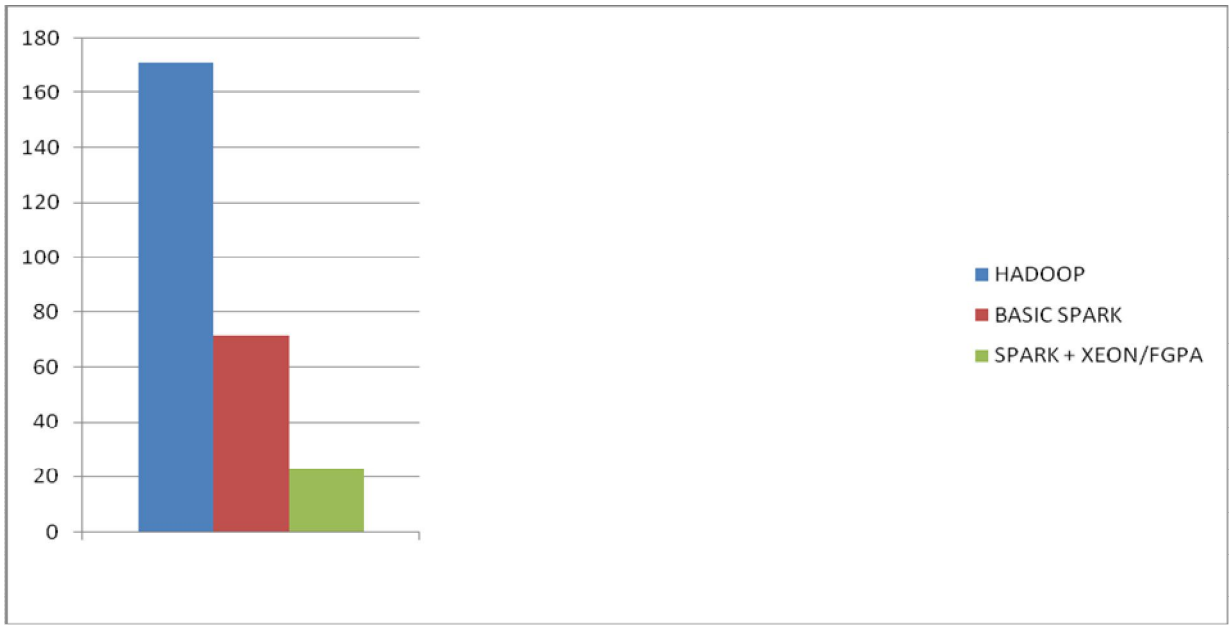


Fig No 4: The performance Comparison of Hadoop and Apache Spark

I. Spark With Xeon/Fpga For Improved Spark Performance

In the Spark database operation primitive, sorting requires efficient implementation and high performance in terms of latency, throughput, and energy consumption.

More work is being performed to enable user-defined time extraction functions. This will enable developers to check event time against events already processed. Work in this area is expected in a future release of Spark.

We develop a hybrid design for data compression associate with Spark for get better performance and achieve the function by desired time and speed. Recently accelerating sorting using Xeon/FPGA has been of growing in data compression process.

1) Enhanced Performance: Accelerators compliment CPU cores to meet market needs for performance of diverse workloads in the

2) Data Center: Enhance single thread performance with tightly coupled accelerators or compliment multi-core performance with loosely coupled accelerators via PCIe or QPI attach.

*3) Move to Heterogeneous Computing***:** Moore's Law continues but demands radical changes in architecture and software.

a) Architectures will go beyond homogeneous parallelism, embrace heterogeneity, and exploit the bounty of transistors to incorporate application-customized hardware.

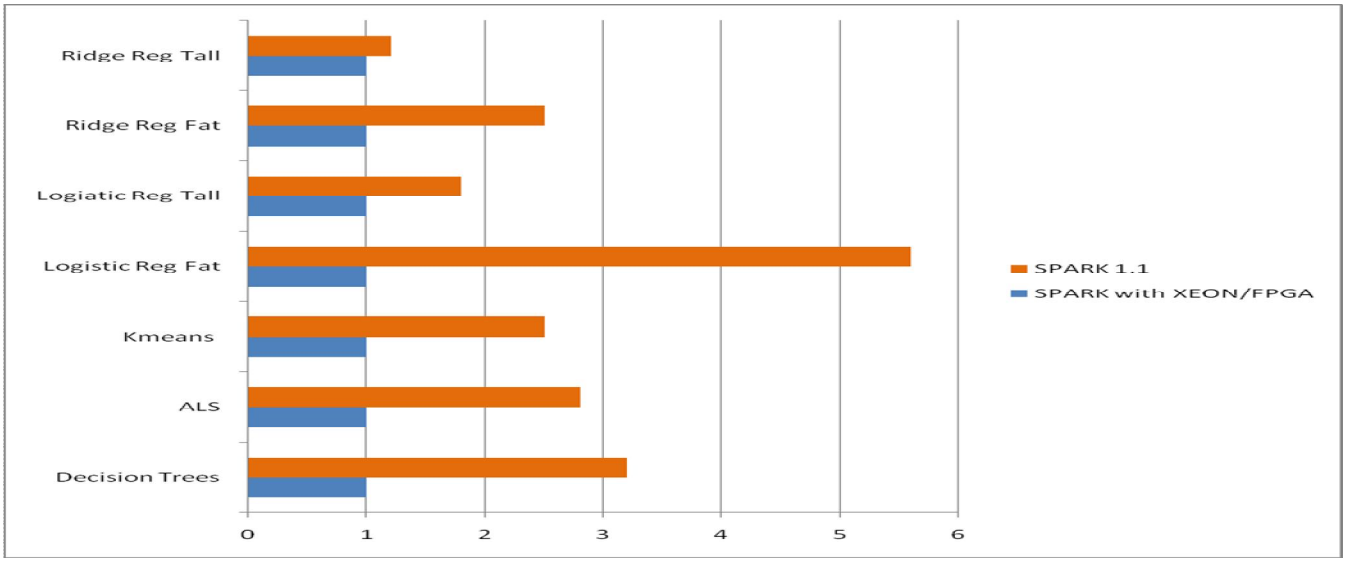


Fig No5: Performance Comparision of Hadoop and Apache Spark with Xeon/FPGA

III. CONCLUSION AND FUTURE WORK

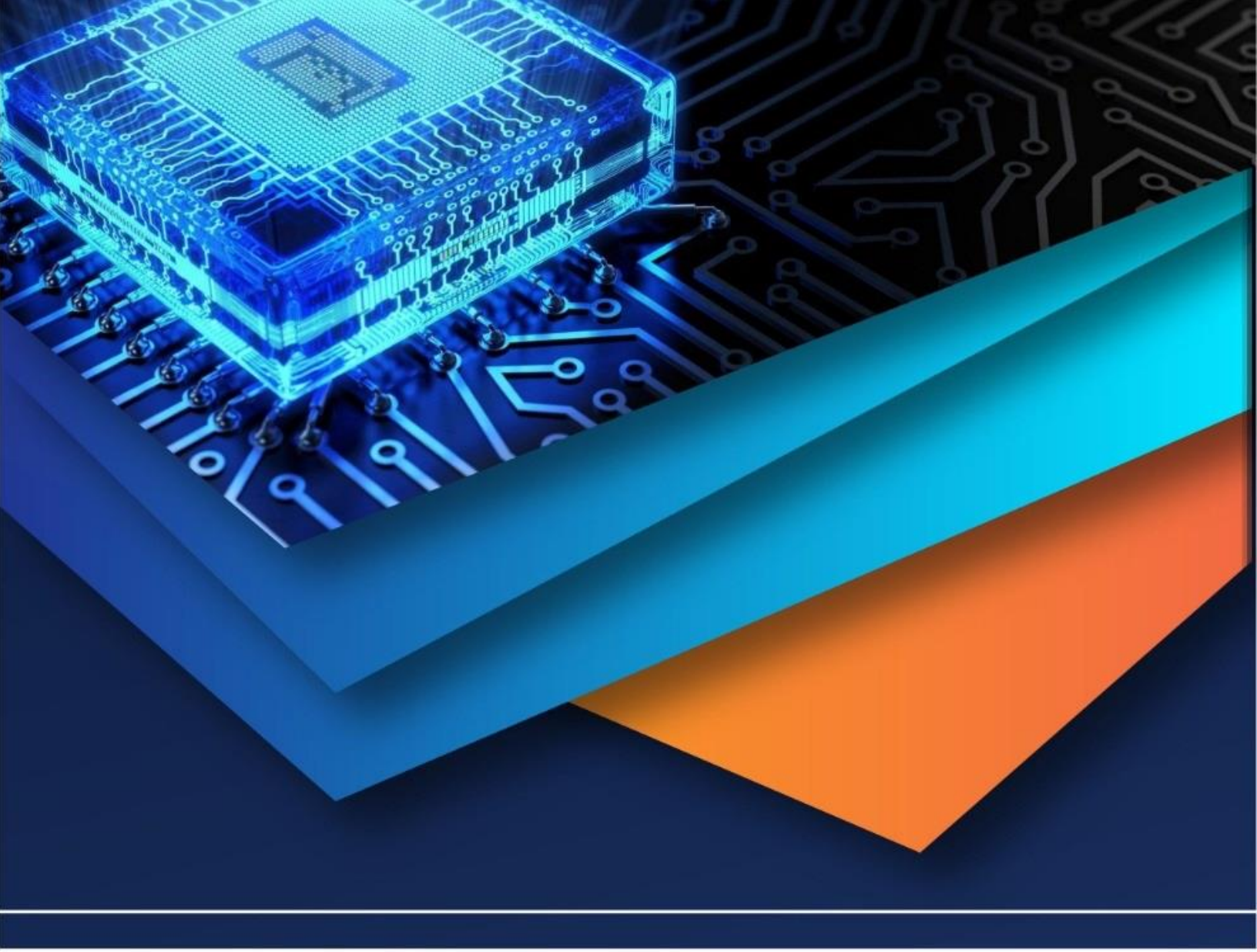
A similarity model was developed to generate the standard data chunks for compressing big data sets. Instead of compression over basic data units, the compression was conducted over partitioned data chunks.

Spark is simple but provides speed in memory and run in disk, so reduce number of read/write operations. The Spark offers good scalability and fault-tolerance for massive data processing. However, that Spark complements RDD with scalable and flexible parallel processing for various data analysis such as scientific data processing. Nonetheless, efficiency, especially I/O costs of Spark still need to be addressed for successful implications.

With the popularity of Spark and its specialty in processing streaming big data set, in future we will explore the way to implement our Spark data compression techniques with Xeon/FPGA to compress the multimedia files for better data processing achievements.

REFERENCE

[1] R.S. Brar and B.Singh, "A survey on different compression techniques and bit reduction algorithm for compression of text data" International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE) Volume 3, Issue 3, March 2013
[2] S. Porwal, Y. Chaudhary, J. Joshi and M. Jain , " Data Compression Methodologies for Lossless Data and Comparison between Algorithms" International Journal of Engineering Science and Innovative Technology (IJESIT) Volume 2, Issue 2, March 2013
[3] S. Shanmugasundaram and R. Lourdusamy, "A Comparative Study of Text Compression Algorithms" International Journal of Wisdom Based Computing, Vol. 1 (3), December 2011
[4] I. M.A.D. Suarjaya, "A New Algorithm for Data Compression Optimization", (IJACSA) International Journal of Advanced Computer Science and S. Kapoor and A. Chopra, "A Review of Lempel Ziv Compression Techniques" IJCST Vol. 4, Issue 2, April - June 2013
[5] Applications, Vol. 3, No. 8, 2012, pp.14-17
[6] S.R. Kodituwakku and U. S. Amarasinghe , "Comparison Of Lossless Data Compression Algorithms For Text Data" Indian Journal of Computer Science and Engineering Vol1No 4 416-425R. Kaur and M. Goyal, "An Algorithm for Lossless Text Data Compression" International Journal of Engineering Research & Technology (IJERT), Vol. 2 Issue 7, July - 2013
[7] H. Altarawneh and M. Altarawneh, " Data Compression Techniques on Text Files: A Comparison Study " International Journal of Computer Applications, Volume 26– No.5,July 2011
[8] U. Khurana and A. Koul, "Text Compression And Superfast Searching" Thapar Institute Of Engineering and Technology, Patiala, Punjab, India-147004
[9] A. Singh and Y. Bhatnagar, " Enhancement of data compression using Incremental Encoding" International Journal of Scientific & Engineering Research, Volume 3, Issue 5, May-2012
[10] A.J Mann, "Analysis and Comparison of Algorithms for Lossless Data Compression"International Journal of Information and Computation Technology, ISSN 0974-2239 Volume 3, Number 3 (2013), pp. 139-146
[11] K. Rastogi, K. Sengar, "Analysis and Performance Comparison of Lossless Compression Techniques for Text Data" International Journal of Engineering Technology and Computer Research (IJETCR) 2 (1) 2014, 16-19
[12] M. Sharma, "Compression using Huffman Coding " IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.5, May 2010
[13] S. Shanmugasundaram and R. Lourdusamy, " IIDBE: A Lossless Text Transform for Better Compression " International Journal of Wisdom Based Computing, Vol. 1 (2), August 2011
[14] P. Kumar and A.K Varshney, "Double Huffman Coding " International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE) Volume 2, Issue 8, August 2012
[15] R. Gupta, A. Gupta, S. Agarwal, "A Novel Data Compression Algorithm For Dynamic Data" IEEE REGION 8 SIBIRCON
[16] A. Kattan, "Universal Intelligent Data Compression Systems: A Review" 2010 IEEE[18] M. H Btoush, J. Siddiqi and B. Akhgar, "Observations on Compressing Text Files of Varying Length " Fifth International Conference on Information Technology: New Generations, 2008 IEEE 2012, pp.1-6
[17] M. H Btoush, J. Siddiqi and B. Akhgar, "Observations on Compressing Text Files of Varying Length " Fifth International Conference on Information Technology: New Generations, 2008 IEEE.
[18] "Huffman Encoding and Data Compression" CS106B Spring 2012 Julie Zelenski with minor edits by Keith Schwarz.
[19] "A Review of Various Data Compression Techniques to form a New Technique for Text Data Compression" Imperial Journal of Interdisciplinary Research (IJIR) Vol-1, Issue.5, 2015.
[20] "Simultaneous Data Compression and Encryption", T. Subhamastan Rao et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (5), 2011, 2369-2374.
[21] http://www.evontech.com/what-we-are-saying/entry/apache-hadoop-vs-apache-spark-two-popular-big-data-frameworks-compared.html
[22] "Getting started with apache spark" james a scott, Copyright © 2015 James A. Scott and MapR Technologies, Inc. All rights reserved.
[23] http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/
[24] A Scalable Data Chunk Similarity Based Compression Approach for Efficient Big Sensing Data Processing on Cloud IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 29, NO. 6, JUNE 2017.
[25] http://www.nallatech.com/solutions/fpga-accelerated-computing/opencl-integrated-platforms/



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY