International Journal of Electrical, Electronics and Computer Engineering **3**(2): 13-16(2014)

ISSN No. (Online): 2277-2626

Map Reduce algorithm for Data Compression

Anindita Khade* and Dr. Subhash K. Shinde**

**Research Scholar, Department of Computer Science Engineering,*
Lokmanya Tilak College of Engineering, Koparkhairane, Navi Mumbai, (MS), India
***Professor, Department of Computer Science Engineering,*
Lokmanya Tilak College of Engineering, Koparkhairane, Navi Mumbai, (MS), India

(Corresponding author: Anindita Khade)
(Received 05 May, 2014 Accepted 28 July, 2014)

ABSTRACT: File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network, or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop. For data intensive workloads, I/O operation and network data transfer will take considerable time to complete. When using Hadoop, there are many challenges in dealing with large data sets. Regardless of whether you store your data in HDFS or ASV, the fundamental challenge is that large volumes can easily cause network and I/O bottlenecks. One of the best known techniques of data compression is dictionary encoding. MapReduce, many instances of "map" steps process individual blocks of an input file to produce one or more outputs; these outputs are passed to "reduce" steps where they are combined to produce a single end result. In this paper we propose a MapReduce algorithm that efficiently compresses and decompresses a large amount of data. For testing purpose we use Hadoop Framework.

Keywords: Data Compression, Hadoop, HDFS, MapReduce

I. INTRODUCTION

A concise, contemporary definition of big data from Gartner defines it as "high-volume, -velocity and -variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making". So, big data can comprise structured and unstructured data, it exists in high volumes and undergoes high rates of change. The key reason behind the rise of big data is its use to provide actionable insights. Typically, organisations use analytics applications to extract information that would otherwise be invisible, or impossible to derive using existing methods.

Industries such as petrochemicals and financial services have been using data warehousing techniques to process very large data sets for decades, but this is not what most understand as big data today. The key difference is that today's big data sets include unstructured data and allow for extracting results from a variety of data types, such as emails, log files, social media, business transactions and a host of others. For example, sales figures of a particular item in a chain of retail stores exist in a database and accessing them is not a big data problem .But, if the business wants to cross-reference sales of a particular item with weather conditions at time of sale, or with various customer details, and to retrieve that information quickly, this would require intense processing and would be an application of big data technology.

A. Big Data Storage

One of the key characteristics of big data applications is that they demand real-time or near real-time responses. If a police officer stops a car they need data on that car and its occupants as quickly as possible. Likewise, a financial application needs to pull data from a variety of sources quickly to present traders with correlated information that allows them to make buy or sell decisions ahead of the competition. Data volumes are growing very quickly - especially unstructured data - at a rate typically of around 50% annually. As we move forward, this will only likely increase, with data augmented by that from growing numbers and types of machine sensors as well as by mobile data, social media and so on. All of which means that big data infrastructures tend to demand high processing/IOPS performance and very large capacity. The methodology selected to store big data should reflect the application and its usage patterns. Traditional data warehousing operations mined relatively homogenous data sets, often supported by fairly monolithic storage infrastructures in a way that today would be considered less than optimal in terms of the ability to add processing or storage capacity. By contrast, a contemporary web analytics workload demands low-latency access to very large numbers of small files, where scale-out storage - consisting of a number of compute/storage elements where capacity and performance can be added in relatively small increments - is more appropriate. That implies a number of storage approaches. Firstly, there is scale-out NAS. This is file level access storage in which storage nodes can be daisy-chained together and storage capacity or processing power can be increased as nodes are added. Meanwhile, the presence of parallel file systems that scale to billions of files and peta bytes of capacity allow for truly big data sets that can be linked together across locations and interrogated.

Major scale out NAS products for big data include: EMC Isilon with its OneFS distributed file system; Hitachi Data Systems' Cloudera Hadoop Distribution Cluster reference architecture; Data Direct Networks hScaler Hadoop NAS platform; IBM SONAS; HP X9000, and NetApp, which has now reached version 8.2 of its DATA Ontap scale-out operating system. Another possible approach that allows to very large sets of data is object storage. This sees the replacement of the traditional tree-like file system with a flat data structure in which files are located by unique IDs, something like the DNS system on the internet. This potentially makes the handling of very large numbers of objects less taxing than is the case with a hierarchical structure. Object storage products are increasingly able to work with big data analytics environments and products include Scality's RING architecture, Dell DX, EMC's Atmos platforms.

B. Need for compression

When using Hadoop, there are many challenges in dealing with large data sets. Moreover, the internal MapReduce "shuffle" process is also under huge I/O pressure, and must often "spill out" intermediate data to local disks before processing can advance from the Map phase to the Reduce phase. Disk I/O and network bandwidth is a precious resource for any Hadoop cluster. Therefore, compressing files for storage can not only save disk space, but also speed up data transfer across the network. More importantly, when processing large volumes of data in a MapReduce job, the combination of data compression and decreased network load can sometimes bring significant performance improvements, due to the reduced I/O and network resource consumption throughout the MapReduce process pipeline. Enabling compression is typically a trade off between I/O and speed of computation. Compression will typically reduce I/O and decrease network usage. Compression can happen when the MapReduce code reads the data or when it writes it out. When a MapReduce job is run against compressed data, CPU utilization is increased, because data must be de-compressed before files can be processed by the Map and Reduce tasks. Therefore, decompression usually increases the time of the job. However, we have found that, in most cases, the overall job performance will be improved by enabling compression in multiple phases of the job configuration.

C. Dictionary Encoding

A dictionary coder, also sometimes known as a substitution coder, is a class of lossless data compression algorithms which operate by searching for matches between the text to be compressed and a set of strings contained in a data structure (called the 'dictionary') maintained by the encoder. When the encoder finds such a match, it substitutes a reference to the string's position in the data structure.

II. LITERATURE SURVEY

Single machine dictionary encoding is used in RDF storage engines like Hexastore, 3Store and Sesame to store the information more efficiently [10-12]. Inference engines like Reasoning-Hadoop [3] and OWLIM [13] also compress the data with this technique. Dictionary encoding is not only used within the Semantic Web but also in several other domains. In [15] dictionary encoding is used for image compression. In [14] the authors present some parallel techniques to compress data using an pre-existing dictionary. In some domains, the dictionary is small enough to be kept in main memory. A good comparison between the performance of different in-memory data structures is given in [16]. From the comparison, it is clear that the hash table is the fastest data structure. [17] proposes a new data structure, called burst trie which maintains the strings in sorted, or near- sorted order, and has performance comparable to the one of a tree.

III. APACHE HADOOP AND MAP REDUCE

Hadoop is an open source software framework that supports data-intensive distributed applications. Hadoop is licensed under the Apache v2 license. It is therefore generally known as Apache Hadoop. Hadoop has been developed, based on a paper originally written by Google on Map Reduce system and applies concepts of functional programming. Hadoop is written in the Java programming language and is the highest-level Apache project being constructed and used by a global community of contributors. Hadoop was developed by Doug Cutting and Michael J. Cafarella. It is made up of 2 parts : HDFS and Hadoop MapReduce.

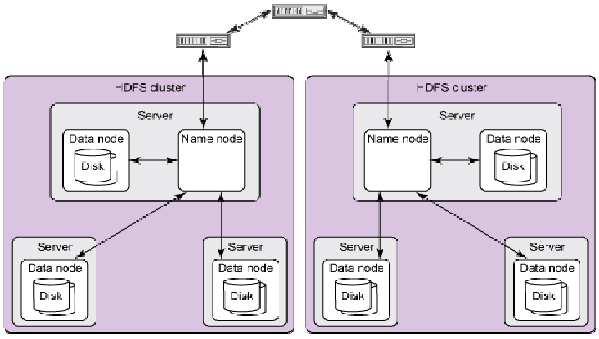

Fig.1. HDFS Architecture.

A. Working with HDFS

The file content is split into large blocks (typically 128 megabytes), and each block of the file is independently replicated at multiple Data Nodes. The blocks are stored on the local file system on the datanodes. The Namenode actively monitors the number of replicas of a block. When a replica of a block is lost due to a DataNode failure or disk failure, the NameNode creates another replica of the block. The NameNode maintains the namespace tree and the mapping of blocks to DataNodes, holding the entire namespace image in RAM.

The NameNode does not directly send requests to DataNodes. It sends instructions to the DataNodes by replying to heartbeats sent by those DataNodes. The instructions include commands to: replicate blocks to other nodes, remove local block replicas, re-register and send an immediate block report, or shut down the node.

B. Map Reduce Programming model

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel distributed algorithm on a cluster. A MapReduce program is composed of a **Map()** procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a **Reduce()** procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). The "MapReduce System" (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and tolerance. The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as in their original forms. The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine once. As such, a single-threaded implementation of MapReduce will usually not be faster than a traditional implementation. Only when the optimized distributed shuffle operation (which reduces network communication cost) and fault tolerance features of the MapReduce framework come into play, is the use of this model beneficial.

MapReduce libraries have been written in many programming languages, with different levels of optimization. A popular open-source implementation is Apache Hadoop. The name MapReduce originally referred to the proprietary Google technology but has since been generalized.

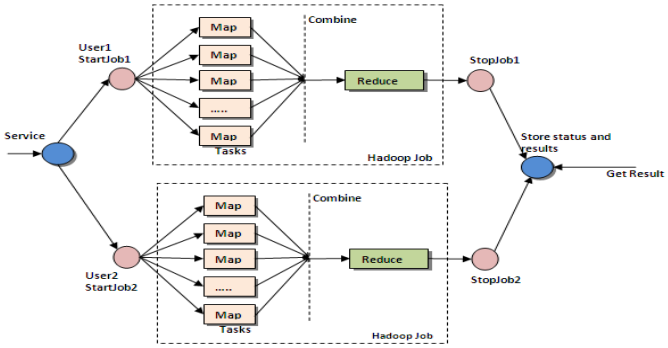

Fig. 2. Map Reduce Programming Model.

VI. MAP REDUCE BASED DICTIONARY ENCODING ALGORITHM

MapReduce can be either used alone or in combination with an external DBMS. If we would use an external DBMS to store the dictionary table, we can minimize the number of queries exploiting the sorting ability of MapReduce. In this case, we could write an appropriate map function so that all the statements that share the same term in the same position would be grouped together. The reduce function could query the DBMS once per group and replace at most one term in each statement. The advantage of this approach compared to the naive one is that here we need to query the DBMS only once per term, and not at every occurrence. However, the performance will suffer from load balancing problems because the statements are grouped on the single terms, and some of them will generate groups that are too large to be processed by a single machine. We propose an alternative algorithm which does not use an external dictionary, but instead builds it internally. We do not execute one job for each part of the statements, but instead we first deconstruct the statements, replace the terms with the numerical IDs, and finally reconstruct them. It internally consists of 3 Map Reduce tasks. The first one creates the dictionary. The second encodes the data with the dictionary algorithm. The third decodes the encoded file back to normal.

V. RESULTS AND OBSERVATIONS

We have implemented this algorithm using 1 master and 2 slaves. The size of dataset (for experimental purpose) we took is of 1.4 KB. We have also compared our algorithm with 2 known text compression algorithms. The results are as follows:

Table 1 : Comparison of results.

Algorithm Used	Rate of compression	Size of file after compression	Comp. time	Decomp. time
MapReduce algorithm	1.93	724Bytes	50 secs	70 secs
GZip	1.85	756Bytes	7 secs	10 secs Approx.
BZip2	1.86	751Bytes	10 secs	10 secs Approx.

VI. CONCLUSIONS AND FUTURE WORK

Thus a map reduce based encoding algorithm for text compression has been implemented. We have used Hadoop framework for our testing purpose. The results show that in a distributed environment, our Map reduce algorithm is slower as compared to the other two, but size of file after compression is less as compared to them. Hence we conclude that this algorithm can be used when we need to compress large files for storage purpose. We also conclude that the time required to decompress files using this algorithm is higher as compared to the compression algorithm. Future work can be suggested as to use this compression algorithm for multimedia compression. The work we have presented can be extended in many different ways. For example, the current algorithm can compress the data only once because the dictionary is built from scratch. Future work could aim to expand the algorithm to deal with incremental updates without recompressing the entire input every time.

REFERENCES

[1]. Jacopo Urbani, Jason Maassen, Henri Bal .Massive Semantic Web data compression with MapReduce, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010.

[2]. J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters. *In proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages **137**(147), 2004.

[3]. J. Urbani, S. Kotoulas, E. Oren, and F.van Harmelen. Scalable distributed reasoning using mapreduce, In Proceedings of the ISWC '09, 2009.

[4]. H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map reduce merge: simplified relational data processing on large clusters.

[5]. Makho Ngazimbi, Data clustering using Mapreduce.

[6]. Nascif A. Abousalh-Neto, Sumeyye Kazgan, Bigdata Exploration through visual analytics.

[7]. Jens Dittrich Jorge-Arnulfo Quian´e-Ruiz, Efficient Big Data Processing in Hadoop MapReduce.

[8]. Tom White, Hadoop-The Definitive Guide.

[9]. Compression in Hadoop, Microsoft White paper.

[10]. D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. *In Proceedings of the 33rd international conference on Very large data bases*, pages 411-422. VLDB Endowment, 2007.

[11]. J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential,* page 197, 2003.

[12]. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment archive, **1**(1): 1008-1019, 2008.

[13]. A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM a pragmatic semantic repository for OWL. *In Proceedings of the Conference on Web Information Systems Engineering (WISE) Workshops,* pages 182-192, 2005.

[14]. H. Nagumo, M. Lu, and K. Watson. Parallel algorithms for the static dictionary compression. *In Proc. IEEE Data Compression Conf*, pages 162-171,1995.

[15]. Y. Ye and P. Cosman. Dictionary design for text image compression with JBIG 2. *IEEE Transactions on Image Processing,* **10**(6): 818-828, 2001.

[16]. J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, **80**: 2001.

[17]. S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, effcient data structure for string keys. *ACM Transactions on Information Systems*, **20**:192-223, 2002.