ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 8, August 2015

Optimized Mapreduce across Datacenter's Using Dache: a Data Aware Caching for Big-Data Applications

S.Venkata Krishna Kumar, D.Veera Bhuvaneswari

Associate Professor, Dept. of Computer Science, PSG College of Arts and Science, Coimbatore, Tamil Nadu, India

M.Phil Scholar, Dept. of Computer Science, PSG College of Arts and Science, Coimbatore, Tamil Nadu, India

ABSTRACT: The buzz word cloud computing provides a virtual area with authentication and supports the information storage and retrieval at hand. The cloud computing paradigm along with software tools such as Google's MapReduce and Apache's Hadoop MapReduce framework offer a response to the problem by distributing computations among large sets of nodes.Its a open source for big data application. In many scenarios input data is geo-distributed across data centers, and moving forwardly all data to a single data center before processing is expensive. This paper deals with executing sequences of MapReduce jobs on geo-distributed datasets.Analysis is done in all possible ways of executing such jobs, and propose data transformation graphs. Big-data refers to the very large-scale geographically distributed data processing applications that operate on exceptionally large amounts of data.The MapReduce framework generates a large amount of intermediate data. Such abundant information is thrown away after the tasks finish, because MapReduce is unable to utilize them. Dache acts as a cache memory in data centers to store the data temporarily on cloud. In this paper,Dache is proposed, which is a data-aware cache framework for big-data applications. In Dache, tasks submit their intermediate results to the cache manager. Before the computing work is being executed the cache manager is raised with queries.

KEYWORDS: Cloud Computing; Big Data; Hadoop; Data center; MapReduce; Dache; Geo-distributed.

I. INTRODUCTION

Big data analysis is one of the major challenges of our era. The limits to what can be done are often times due to how much data can be processed in a given time-frame. Big datasets inherently arise due to applications generating and retaining more information to improve operation, monitoring, or auditing; applications such as social networks support individual users in generating increasing amounts of data. Implementations of the popular MapReduce framework, such as Apache Hadoop, have become part of the standard toolkit for processing large datasets using cloud resources, and are provided by most cloud vendors. In short, MapReduce works by dividing input files into chunks and processing these in a series of parallelizable steps. As suggested by the name, mapping and reducing constitute the essential phases for a MapReduce job. In the former phase, mappers processes read respective input file chunks and produce (key,val) pairs called "intermediate data". Each reducer process atomically applies the reduction function to all values of each key assigned to it.

1.1 Geo-Distribution

As the initial hype of cloud computing is high, users are starting to see beyond the illusion computing resources and realize that these are implemented by concrete datacenters, whose locations matter. More and more applications relying on cloud platforms are geodistributed, for any (combination) of the following reasons:
(a) data is stored near its respective sources or frequently accessing entities (e.g., clients) which can be distributed, but the data is analyzed globally; (b) data is gathered and stored by different(sub)organizations, yet shared towards a common goal; (c) data is replicated across datacenters for availability, incompletely to limit the overhead of costly updates.

1.2 Job Sequences

To make matters worse, MapReduce jobs do not always come alone. Frequently,sequences of MapReduce jobs are executed on a given input by applying the first job on the given input, applying the second job on the output of the first job, and so on. An example is the handling of large Web caches by executing an algorithm such as PageRank. This algorithm constructs a graph that describes the inter-page relationships, which are refined using further MapReduce jobs. Many times distinct MapReduce jobs are applied in sequence rather than iteratively performing the same job. For instance the query performed on the geo-distributed Web cache mentioned above may have a filtering step and a content search step, which need to be executed as two consecutive jobs. Sequences also arise, indirectly, when using PigLatin/Pig to describe complex data analysis tasks from which MapReduce jobs are generated automatically. When performing a sequence of MapReduce jobs the number of possible execution paths increases dramatically.

1.3 G-MR

This paper introduces G-MR, a system for efficiently processing geo-distributed big data. G-MR is a Hadoop based framework that can efficiently perform a sequence of MapReduce jobs on a geo-distributed dataset across multiple datacenters. With current frameworks for "the cloud" operating only in single datacenters, G-MR thus metaphorically speaking acts much like the atmosphere surrounding the clouds. G-MR employs a novel algorithm nameddata transformation graph (DTG) algorithm that determines an optimized execution path for performing a sequence of MapReduce jobs based on characteristics of the dataset, MapReduce jobs, and the datacenter infrastructure. Our DTG algorithm can be used to optimize for either execution time or (monetary) cost. The optimized execution path determined by G-MR may be different from the optimum possible execution path. Determining the optimum possible execution path for a large dataset is hard since every possible data move has to be considered. G-MR provides a good compromise by determining and executing an optimized execution path that performs better than commonly used execution paths. This execution path is then enforced by G-MR through a geo-distributed chain of operations consisting of geodistributed copy operations and MapReduce executions performed with Hadoop MapReduce clusters deployed in each of the involved datacenters.

II. CACHE DESCRIPTIONS

2.1 Map phase cache description scheme Cache refers to the intermediate data that is produced by worker nodes/processes during the execution of a MapReduce task. A piece of cached data is stored in a Distributed File System (DFS). The content of a cache item is described by the original data and the operations applied. Formally, a cache item is described by a 2-tuple: Origin, Operation. Origin is the name of a file in the DFS. Operation is a linear list of available operations performed on the Origin file. For example, in the word count application, each mapper node/process emits a list of word, count tuples that record the count of each word in the file that the mapper processes. Dache stores this list to a file. This file becomes a cache item. Given an original input data file, word list 08012012.txt, the cache item is described by word list 08012012.txt, item count. Here, item refers to white-space-separated character strings. Note that the new line character is also considered as one of the white spaces, so item precisely captures the word in a text file and item count directly corresponds to the word count operation performed on the data file. The exact format of the cache description of different applications varies according to their specific semantic contexts. This could be designed and implemented by application developers who are responsible for implementing their MapReduce tasks. In our prototype, we present several supported operations:

Item Count: The count of all occurrences of each item in a text file. The items are separated by a userdefined separator.
Sort This operation sorts the records of the file. The comparison operator is defined on two items and returns the order of precedence.
Selection This operation selects an item that meets a given criterion. It could be an order in the list of items. A special selection operation involves selecting the median of a linear list of items.
Transform This operation transforms each item in the input file into a different item. The transformation is described further by the other information in the operation descriptions. This can only be specified by the application developers.
Classification This operation classifies the items in the input file into multiple groups. This could be an exact classification, where a deterministic classification criterion is applied sequentially on each item, or an approximate classification, where an iterative classification process is applied and the iteration count should be recorded.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 8, August 2015

2.2.1 Reduce cache

The file splits from the map phase are included in the cache description. Usually, the input given to the reducers is from the whole input of the MapReduce job. Therefore, we could simplify the description by using the file name together with a version number to describe the original file to the reducers. The version number of the input file is used to distinguish incremental changes. A straightforward approach is to encode the size of the input file with the file name. Since we assume that only incremental changes, i.e., appending new data at the end of the file, are allowed, the size of the file is enough to identify the changes made during different MapReduce jobs. Note that even the entire output of the input files of a MapReduce job is used in the reduce phase, the file splits can still be aggregated.

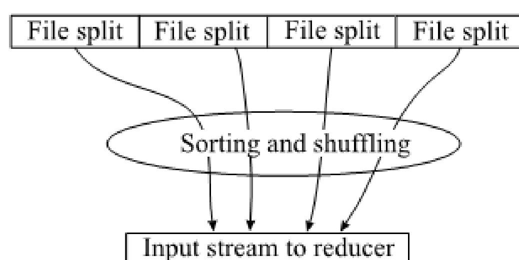

Fig. 1 The input stream to a reducer is obtained by sorting and then shuffling multiple output files of mappers.

This mapping is used to identify the input to the reducer.

As shown in Fig. 1, file splits are sorted and shuffled to generate the input for the reducers. Although this process is implicitly handled by the MapReduce framework, the users are able to specify a shuffling method by supplying a partitioner, which is implemented as a Java object in Hadoop. The partitioner examines the key of a record and determines which reducer should process this record in the reduce phase. Therefore, the cache description should be attached with the partitioner, which can be implemented as a serialized object in Hadoop. The same input file splits that are partitioned by different partitioners produce different reduce inputs, therefore cannot be treated as the same. At last, the index of the reducer assigned by the partitioner is attached. The whole description is a 3-tuple: ffile splits, partitioner, reducer indexg. The description is completed to accurately identify the input to a reducer. The reducer then appends its output with the description to produce a cache item. However, This process is automatically handled by the reducers.

III. PROTOCOL

3.1 Relationship between job types and cache organization

The partial results generated in the map and reduce phases can be utilized in different scenarios. There are two types of cache items: the map cache and the reduce cache. They have different complexities when it comes to sharing under different scenarios. Cache items in the map phase are easy to share because the operations applied are generally well-formed. When processing each file split, the cache manager reports the previous file splitting scheme used in its cache item. The new MapReduce job needs to split the files according to the same splitting scheme in order to utilize the cache items. However, if the new MapReduce job uses a different file splitting scheme, the map results cannot be used directly, unless the operations applied in the map phase are context free. By context free, we mean that the operation only generates results based on the input records, which does not consider the file split scheme. This is generally true. When considering cache sharing in the reduce phase, we identify two general situations. The first is when the reducers complete different jobs from the cached reduce cache items of the previous MapReduce jobs, as shown in Fig.2. In this case, after the mappers submit the results obtained from the cache items, the MapReduce framework uses the partitioner provided by the new MapReduce job to feed input to the reducers. The saved computation is obtained by removing the processing in the Map phase. Usually, new content is appended at the end of the input files, which requires additional mappers to process. However, this does not require additional processes other than those introduced above.

The second situation is when the reducers can actually take advantage of the previously-cached reduce cache items as illustrated in Fig. 3. Using the description scheme discussed in Section 2, the reducers determine how the output of the

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 8, August 2015

map phase is shuffled. The cache manager automatically identifies the best-matched cache item to feed each reducer, which is the one with the maximum overlap in the original input file in the Map phase.

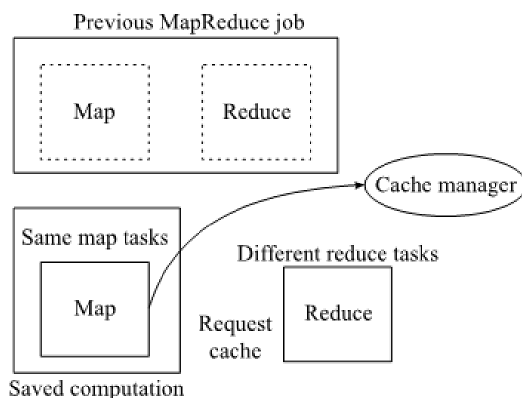

Fig. 2 The situation where two MapReduce jobs have the same map tasks, which could save a fraction of computation by requesting caches from the cache manager

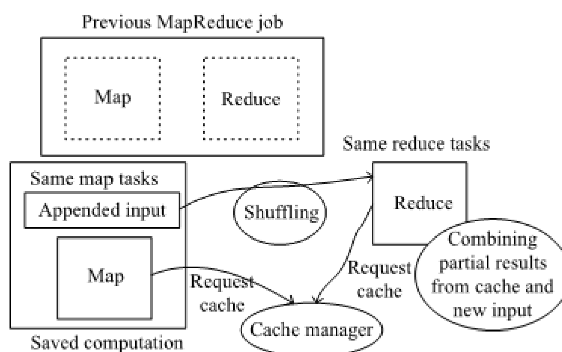

Fig. 3 The situation where two MapReduce jobs have the same map and reduce tasks. The reducers combine results from the cache items and the appended input to produce the final results.

IV. APACHE HADOOP

Hadoop is a Java-based MapReduce implementation for large clusters. It is bundled with the Hadoop Distributed File System (HDFS), which is optimized for batch workloads such as those of MapReduce. In many Hadoop applications, HDFS is used to store the input of the map phase as well as the output of the reduce phase. HDFS is, however, not used to store intermediate results such as the output of the map phase. They are stored on the individual local file systems of nodes.

The Hadoop follows a master-slave model where the master is implemented in Hadoop's JobTracker. The master is responsible for accepting jobs, dividing those into tasks which encompass mappers or reducers, and assigning those tasks to slave worker nodes. Each worker node runs a Task Tracker that manages its assigned tasks. A default split in Hadoop contains one HDFS block (64 MB), and the number of file blocks in the input data is used to determine the number of mappers. The Hadoop map phase for a given mapper consists in first reading the mapper's split and parsing it into (key1,val1) pairs.

Once the map function has been applied to each record, the TaskTracker is notified of the final output; in turn, the TaskTracker informs the JobTracker of completion. The JobTracker informs the TaskTrackers of reducers about the

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 8, August 2015

locations of the TaskTrackers of corresponding mappers. Shuffling takes place over HTTP. A reducer fetches data from a configurable number of mapper-TaskTrackers at a time, with 5 being the default number.

V. G-MR IMPLEMENTATION

This section describes implementation details of G-MR, which consists of roughly4900lines of Java code.

5.1 Datacenter and Job Configurations

A GroupManager initiates the G-MR instance by processing an XML-based datacenter configurationfile. This file describes the datacenters that may participate in the geodistributed MapReduce jobs handled by G-MR. A sample file is given in Figure 4.

An identifier is provided for each datacenter to refer to it when running MapReduce jobs. Datacenters keep their data in independent distributed file systems. Currently GMR supports Amazon Service and Apache HDFS.

```
<dcconf>
<datacenter>
<id>DC1</id>
<provider>EC2</provider>
<fstype>HDFS</fstype>
<jobtracker>
a.useast.amazonaws.com:9001
</jobtracker>
<filesystem>
hdfs://b.useast.amazonaws.com
</filesystem>
</datacenter>
</dcconf>
```

Fig. 4: Sample datacenter configuration file

Geo-distributed MapReduce job sequences are submitted to the GroupManager using an XML-based job configuration file. The GroupManager breaks the sequence down into a number of copy and MapReduce operations. This information is passed to JobManager components describing the portion of work that should be executed within their respective datacenters. A sample job configuration file is given in Figure 5.

Through a job configuration file, a user can specify sub-datasets of the considered geo-distributed input dataset, mapper and reducer classes, associativity of the job, whether DTG algorithm should optimize for cost or time etc. Additionally user has the option of manually specifying the functionsMR(), R(),M(), and A(). Size Function has functions calculateOutputSize and calculateExecutionTime, that if provided will be used to approximate the output size and execution time of the corresponding operations.

```
<jobconf>
<input>
<datacenter>DC1</datacenter>
<location>/user/ec2-user1/webinput</location>
</input>
<mapper>gmr.WordCountMapper</mapper>
...
<optimizeForTime>false</optimizeForTime>
<mrFunction>gmr.WordCountMRFunction</mrFunction>
...
</jobconf>
```

Fig. 5: Sample job configuration file

5.2 Executing Individual MapReduce Our execution algorithm supports the scenarios where mapping and reducing of a given MapReduce job are executed in two different geographical locations. To implement this, G-MR dynamically

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 3, Issue 8, August 2015

generates a straightforward mapper and a reducer for each MapReduce job. A straightforward reducer simply outputs intermediate data it receives while a straightforward mapper can read this intermediate data from a distributed file system and output the same results the actual mapper would have produced.
For each job, to determineM()functions we use the user provided mapper and the corresponding generated straightforward reducer while using the corresponding generated straightforward mapper and the reducer provided by the user to determineR()functions.

VI. CONCLUSIONS

We present the design and evaluation of a data aware cache framework that requires minimum change to the original MapReduce programming model for provisioning incremental processing for Bigdata applications using the MapReduce model. This paper presents G-MR, a MapReduce framework that can efficiently execute a sequence of MapReduce jobs on geo-distributed datasets. G-MR relies on a novel algorithm termed DTG algorithm which looks for the most suitable way to perform a job sequence, minimizing either execution time or cost. We illustrate through real MapReduce application scenarios that G-MR can substantially improve time or cost of job execution We believe that our framework is also applicable to single datacenters with non-uniform transmission characteristics, such as datacenters divided into zones or other network architectures.

REFERENCES

1. W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t Settle for Eventual: Scalable Causal Consistency for Widearea Storage with COPS,” inSOSP, 2011.
2. V. Ramasubramanian, T. Rodeheffer, D. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat, “Cimbiosys: a Platform for Content-based Partial Replication,” inNSDI, 2009.
3. “Hadoop Across Data-Centers,” http://mail-archives.apache.org/modmbox/hadoop-general/201001.mbox/%3C4B4B6AC7.
4. 6020801@lifeless.net%3E.
5. “Hadoop: The Definitive Guide,” http://oreilly.com/catalog/
6. 9780596521981.
7. “PageRank Algorithm,” http://en.wikipedia.org/wiki/PageRank.
8. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “PigLatin: A Not-so-foreign Language for Data Processing,” in SIGMOD, 2008.
9. Apache Software Foundation, “Pig,” http://pig.apache.org.
10. “Vicci Cloud Computing Testbed,” http://www.vicci.org.
11. Y. Sovran, R. Power, M. K. Aguilera, and J. Li, “Transactional Storage for Geo-replicated Systems,” inSOSP, 2011.
12. J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazi` eres, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, “The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM,” SIGOPS OSR, vol. 43, no. 4, pp. 92–105, Jan. 2010.
13. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks,” in Eurosys, 2007.
14. “Teradata,” http://www.teradata.com. C. He, D. Weitzel, D. Swanson, and Y. Lu, “HOG:Distributed
15. Hadoop MapReduce on the Grid,” inMTAGS, 2012.
16. J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd, “GridFlow: Workflow Management for Grid Computing,” inCC
17. Grid, 2003.