ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)
Vol. 3, Issue 3 (July - Sept. 2015)
International Journal of Advanced Research in Computer Science & Technology (IJARCST 2015)

An Approach for Data Processing in Big Data Application using Distributed and Centralized Cache (D-Cache)

[I]Lakesh, [II]A.K.P.Kumar
[I]Student, Dept. of Computer Science, SEAGI, India
[II]Asst. Professor, Dept. of Computer Science, SEAGI, India

Abstract
The big data refers to the large-scale distributed applications that work on unprecedentedly large data sets. Map reduce framework are user specified computation, for parallelizing the computation across large scale clusters of machines. An observation regarding these applications is that they generate a large amount of intermediate key/value pair and these are thrown away after the process is finished. The information which is thrown can be used for the next cycle of execution for particular application and can decrease the time of execution for application. There are potential duplicate computations being performed in this process. The motivation over this system is to use intermediate results for further execution of the applications, where we don't want to execute results which are previously processed. Thus in this system we proposed a Centralized and Distributed cache strategy which keeps previously processed data in cache memory at local and centralized. So there are two cache memories will be used for faster execution of applications, one will be local cache and another will be centralized distributed cache. So that, the size and transactional capacity of data will be increased.

Keywords
Big data; Map Reduce; Distributed and Centralized cache.

I. Introduction

Google Map Reduce[1] is a programming model and a software framework for large-scale distributedcomputing on large amounts of data.Fig.1 illustrates the high-level work flow of a Map Reduce job. Application developers specify the computation in terms of a map and a reduce function, and the underlying MapReduce job scheduling system automatically parallelizes the computation across a cluster of machines. MapReduce gains popularity for its simple programming interface and excellent performance when implementing a large spectrum of applications. Since most such applications take a large amount of input data, they are nicknamed "Big-data applications". As shown in Fig. 1, input data is first split and then feed to workers in the map phase. Individual data items are called records. The MapReduce system parses the input splits to each worker and produces records. After the map phase, intermediate results generated in the map phase are shuffled and sorted by the MapReduce system and are then fed into the workers in the reduce phase. Final results are computed by multiple reducers and written to the disk.

MapReduce provides a standardized framework for implementing large-scale distributed computation, namely, the big-data applications. However, there is a limitation of the system, i.e., the inefficiency in incremental processing. Incremental processing refers to the applications that incrementally grow the input data and continuously apply computations on the input in order to generate output. There are potential duplicate computations being performed in this process. However, MapReduce does not have the mechanism to identify such duplicate computations and accelerate job execution. Motivated by this observation, we propose Distributed and Centralized cache system for big-data applications using the MapReduce framework. This aims at extending the MapReduce framework and provisioning a cache layer for efficiently identifying and accessing cache items in a MapReduce job. The following technical challenges need to be addressed before implementing this proposal.

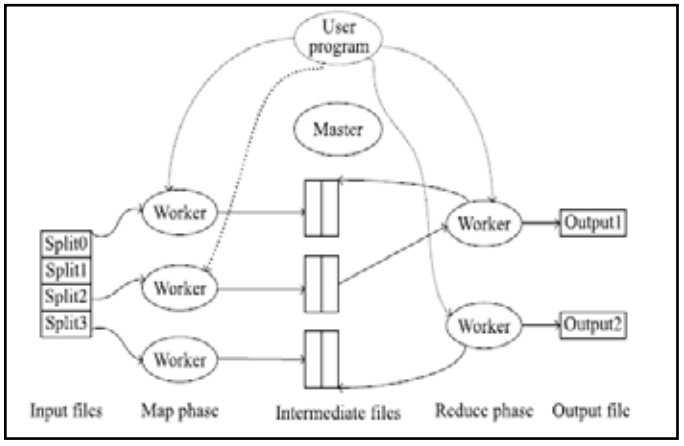

Fig.1 : A high-level illustration of the Map Reduce architecture.

Cache description scheme.In the context of big-data applications, this means that the cache description scheme needs to describe the application framework and the datacontents. Although most big-data applications run on standardized platforms, their individual tasks perform completely different operations and generate different intermediate results. The cache description scheme should provide a customizable indexing that enables the applications to describe their operations and the content of their generated partial results. This is a non-trivial task. In the context of Hadoop, we utilize the sterilization capability provided by the Java[3] language to identify the object that is used by the Map Reduces system to process the input data.

In this paper, we present a cache description scheme on which Distributed and Centralized cache is presented in Fig.2. This scheme identifies the source input from which a cache item is obtained, and the operations applied on the input, so that a cache item produced by the workers in the map phase is indexed properly. Here we are distributing the data to local cache. Each local cache is managed by Distributed cache and it is centralized. So that reduce phase utilize the data item from the centralized cache. We also present a method for reducers to utilize the cached results in the map phase to accelerate the execution of the MapReduce job.

ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

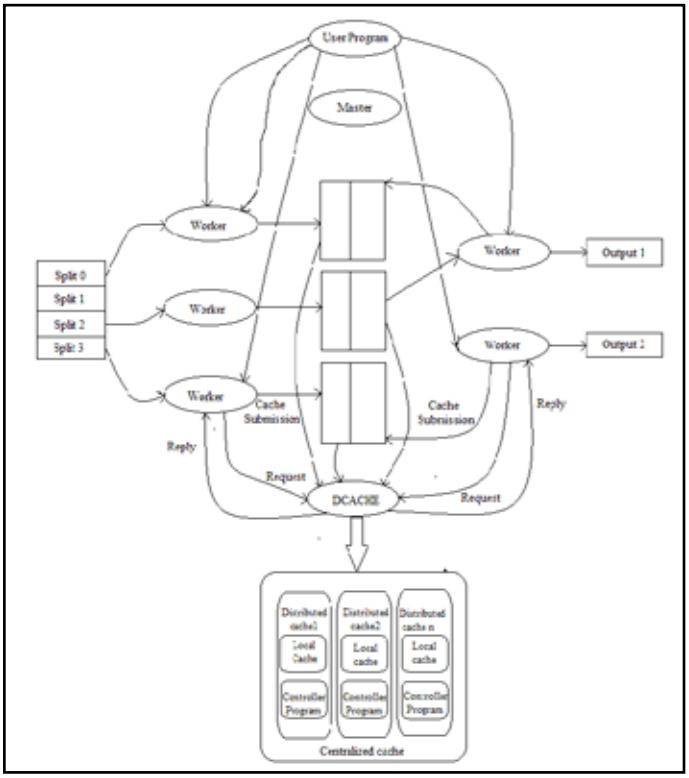

Fig. 2 : High-level illustration of D-Cache Approach.

II. Cache Description

A. 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. DCache 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 countoperation 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 user-defined separator.

Sort. This operation sorts the records of thefile. 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.

B. Reduce Phase Cache Description Scheme

The input for the reduce phase is also a list ofkey-value pairs, where the value could be a list ofvalues. Much like the scheme used for the map phase cache description, the original input and the applied operations are required. The original input is obtained by storing the intermediate results of the map phase in the DFS. The applied operations are identified by unique IDs that are specified by the user. The cached results, unlike those generated in the Map phase, cannot be directly used as the final output. This is because, in incremental processing, intermediate results generated in the Map phase are likely mixed in the shuffling phase, which causes a mismatch between the original input of the cache items and the newly generated input.

A remedy is to apply a finer description of the original input of the cache items in the reduce phase. The description should include the original data files generated in the Map phase. For example, two data files,“file1.data” and “file2.data”, are shuffled to produce two input files,“input1.data” and “input2.data”, for two reducers. “input1.data” and “input2.data” should include “file1.data” and “file2.data” as its shuffling source. As a result, new intermediate data files of the Map phase are generated during incremental processing; the shuffling input will be identified in a similar way. The reducers can identify new inputs from the shuffling sources by shuffling the newly-generatedintermediate result from the Map phase to form thefinal results. For example, assume that “input3.data” is a newly generated result from Map phase; the shuffling results “file1.data” and “file2.data” include a new shuffling source, “input3.data”. A reducer can identify the input “file1.data” as the result of shuffling “input1.data”, “input2.data”, and “input3.data”. The final results of shuffling the output of “input1.data” and “input2.data” are obtained by querying the cache manager. The added shuffling output of “input3.data” is then added to get the new results.

Given the above description, the input given to the reducers is not cached exactly. Only a part of the input is identical to the input of the cache items. The rest is from the output of the incremental processing phase of the mappers. If a reducer could combine the cached partial results with the results obtained from the new inputs and substantially reduce the overall computation time, reducers should cache partial results. Actually, this property is determined by the operations executed by the reducers.

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. 3. In this case, after the mappers submit the results obtained from the cache items, the MapReduce framework uses the practitioner 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.

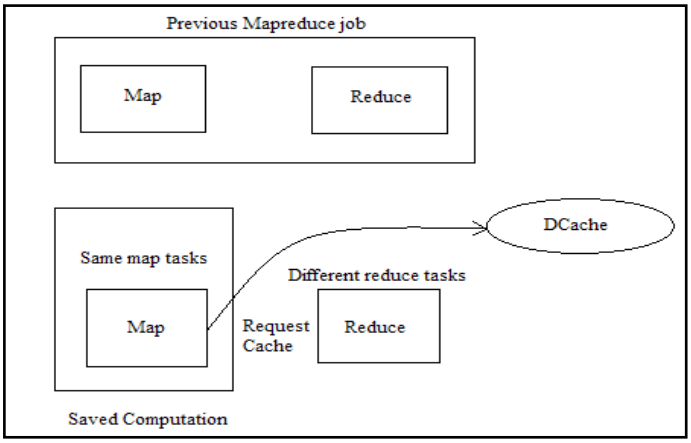

Fig. 3 : The situation where two MapReduce jobs have the same map tasks, which could save a fraction of computation by requesting caches from the DCache.

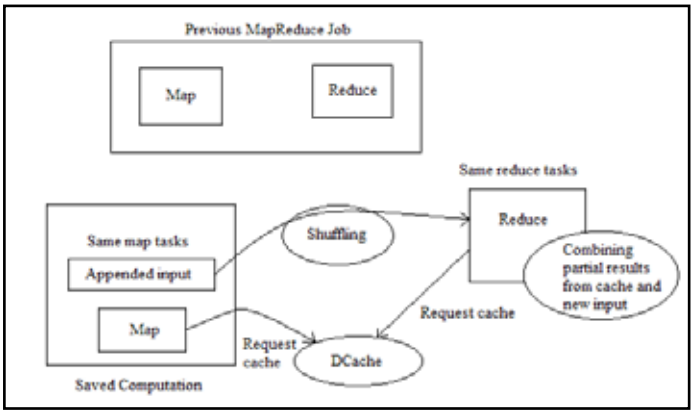

Fig. 4 : The situation where two MapReduce jobs have the same map and reduce tasks.

The second situation is when the reducers can actually take advantage of the previously-cached reduce cache items as illustrated in Fig. 4. Using the description scheme discussed in Section 2, the reducers determine how the output of the 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.

C. Cache Item Submission

Mapper and reducer nodes/processes record cache items into their local storage space. When these operations are completed, the cache items are forwarded to the DCache, which acts like a broker in the publish/subscribe paradigm[4]. The DCache records the description and the file name of the cache item in the DFS. The cache item should be put on the same machine as the worker process that generates it. This requirement improves data locality. The DCache maintains a copy of the mapping between the cache descriptions and the file names of the cache items in its main memory to accelerate queries. It also flushes the mapping file into the disk periodically to avoid permanently losing data.

A worker node/process contacts the DCache each time before it begins processing an input data file. The worker process sends the file name and the operations that it plans to apply to the file to the cache. The Centralized cache which has distributed cache receives this message and compares it with the stored mapping data in local cache. If there is a exact match to a cache item, i.e., its origin is the same as the file name of the request and its operations are the same as the proposed operations that will be performed on the data file, then the DCache will send back a reply containing the tentative description of the cache item to the worker process.

The worker process receives the tentative description and fetches the cache item. For further processing, the worker needs to send the file to the next-stage worker processes. The mapper needs to inform the DCache that it already processed the input file splits for this job. The cache then reports these results to the next phase reducers. If the reducers do not utilize the cache service, the output in the map phase could be directly shuffled to form the input for the reducers. Otherwise, a more complicated process is executed to obtain the required cache items.

One of the benefits of DCache is that it automatically supports incremental processing. Incremental processing means that we have an input that is partially different or only has a small amount of additional data. To perform a previous operation on this new input data is troublesome in conventional MapReduce, because MapReduce does not provide the tools for readily expressing such incremental operations. Usually the operation needs to be performed again on the new input data, or the application developers need to manually cache the stored intermediate data and pick them up in the incremental processing. In DCache, this process is standardized and formalized. Application developers have the ability to express their intentions and operations by using cache description and to request intermediate results through the dispatching service of the centralized cache.

III. Lifetime Management Of Cache Item

The cache needs to determine how much time a cache item can be kept in the DFS. Holding a cache item for an indefinite amount of time will waste storage space when there is no other MapReduce task utilizing the intermediate results of the cache item. The DCache also can promote a cache item to a permanent file and store it in the DFS, which happens when the cache item is used as the final result of a MapReduce task. In this case, the lifetime of the cache item is no longer managed by the cache. The centralized cache still maintains the mapping between cache descriptions and the actual storage location.

IV. Performance Evaluation

A. Implementation

We extend Hadoop to implement DCache. Hadoop is a collection of libraries and tools for DFS and MapReduce computing. The complexity of the entire package is beyond our control, so we take a non-intrusive approach to implement DCache in Hadoop and try not to hack the Hadoop framework itself, but implement DCache by changing the components that are open to application developers. Basically, the centralized cache is implemented as an independent server. It communicates with task trackers in distributed cache which has local cache and provides cache items on receiving requests. The cache stands outside of the Hadoop MapReduce framework.

In order to access cache items, the mapperand reducer tasks first send requests to the DCache. However, this cannot be implemented in mapper and Reducer classes. Hadoop framework fixes the interface of Mapper and Reducer classes to only accept key-value pairs as the input. They cannot identify the file split they are working on; therefore, cache requests cannot be sent from mappers

ISSN : 2347 - 8446 (Online)
ISSN : 2347 - 9817 (Print)

or reducers. We alter two components of Hadoop to implement this function. The first component is Input Format class, an open-accessed component that allows applicationdevelopers to modify. It is responsible for splittingthe input files of the MapReduce job to multiple filesplits and parse data to key-value pairs. Input Format class should query the cache to fetch the splitting scheme of the cache item, if they are the same Map tasks that were being executed previously. It then splits the input files in the same way as the cache item and puts the incremental parts into new file splits. The second component that needs to be altered is the Task Tracker, which is the class responsible for managing tasks. Task Tracker is able to understand file split and bypass the execution of mapper classes entirely. Task Tracker also manages reducer tasks. Similarly, it could bypass reducer tasks by utilizing the cached results. Additionally, application developers must implement a different reduce interface, which takes as input a cache item and a list of key-value pairs and produces the final results.

B. Experiment Settings and Results

Hadoop is run in pseudo-distributed mode on a serverthat has an 8-core CPU, each core running at 3 GHz, 16 GB memory, and a SATA disk. The number of mappers is 16 in all experiments, the reducers' count varies. We use two applications to benchmark the speedup of DCache over Hadoop (the classic MapReduce model): word-count and sort. Word-count counts the number of unique words in large input text files; sorts key-value records based on the lexical order of the key. More details are in Hadoop manual[2] . Word-count is an IO-intensive application that requires loading and storing a sizeable amount of data during the processing. On the other hand, sort uses more mixed word loads. It needs to load and store all input data and needs a computation-intensive sorting phase. The inputs of two applications are generated randomly, and all are 10 GB in size.

The completion time and the speedup are put together. Data is appended to the input file. The size of the appended data varies and is represented as a percentage number to the original input file size, which is 10 GB. sort is more CPU-bound compared to word-count, as a result DCache can bypass computation tasks that take more time, which achieves larger speedups. The speedup decreaseswith the growing size of appended data, but DCache is able to complete jobs faster than Hadoop in all situations. The map phase of sort does not perform much computation, which also makes it easier for DCache to work.

V. Conclusion

We present an approach of a distributed and centralized cache framework that requires minimum change to the original MapReduce programming model for provisioning incremental processing for Big-data applications using the MapReduce model.

References

[1] *J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. of ACM, vol. 51, no. 1, pp. 107-113, 2008.*
[2] *Hadoop, http://hadoop.apache.org/, 2013.*
[3] *Java programming language, http://www.java.com/, 2013.*
[4] *P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput.Surv., vol. 35, no. 2, pp. 114-131, 2003.*