Dache: A Data Aware Caching for Big-Data Applications Using The MapReduce Framework

Yaxiong Zhao† and Jie Wu‡
†Amazon.com Inc
†‡Department of Computer and Information Sciences, Temple University
zhaoyaxi@amazon.com, jiewu@temple.edu

***Abstract*—The buzz-word *big-data* (application) refers to the large-scale distributed applications that work on unprecedentedly large data sets. Google's MapReduce framework and Apache's Hadoop, its open-source implementation, are the defacto software system for big-data applications. An observation regarding these applications is that they generate a large amount of intermediate data, and these abundant information is thrown away after the processing finish. Motivated by this observation, we propose a data-aware cache framework for big-data applications, which is called Dache. In Dache, tasks submit their intermediate results to the cache manager. A task, before initiating its execution, queries the cache manager for potential matched processing results, which could accelerate its execution or even completely saves the execution. A novel cache description scheme and a cache request and reply protocol are designed. We implement Dache by extending the relevant components of the Hadoop project. Testbed experiment results demonstrate that Dache significantly improves the completion time of MapReduce jobs and saves a significant chunk of CPU execution time.**

***Index Terms*—Big-data, MapReduce, Hadoop, distributed file system, cache management.**

I. Introduction

MapReduce [5], and its open-source implementation Hadoop [4], are a software framework for large-scale distributed computing on large amounts of data. Applications specify the computation in terms of a *map* and a *reduce* function working on partitioned data items. The MapReduce framework schedules computation across a cluster of machines.

MapReduce provides a standardized framework for implementing large-scale distributed computation on unprecedentedly large-scale data set. 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. Motivated by this observation, we propose Dache, a data-aware cache system for Big-data applications using the MapReduce framework.

Dache 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: 1) Cache description scheme. Data-aware caching requires each data object to be indexed by its content. In the context of Big-data applications, this means that the cache description scheme needs to describe the application framework and the data contents. Although most big-data applications run on standardized platforms, their individual tasks perform completely different operations and generate different intermediate results. 2) Cache request and reply protocol. The size of the aggregated intermediate data can be very large. Usually the programs are moved to data nodes in order to avoid network communications. The protocol should be able to collate cache items with the worker processes potentially that need the data, so that the transmission delay and overhead are minimized.

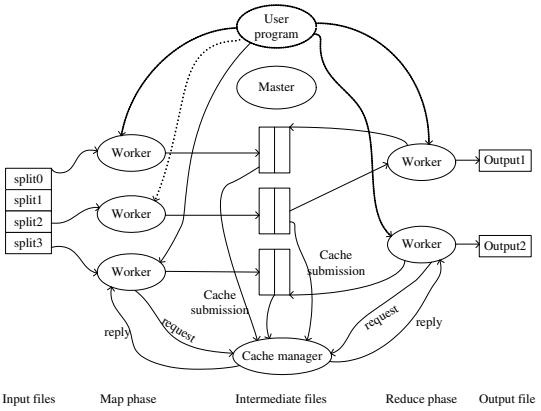

Fig. 1. High level description of the architecture of Dache.

In this paper, we present a novel cache description scheme. This scheme identifies the source input from which a cache item is obtained, and the operations applied on the input. In the reduce phase, we devise a mechanism to take into consideration the partition operations applied on the output in the map phase. We also present a method for reducers to utilize the cached results in the map phase to accelerate their execution. We implement Dache in the Hadoop project by extending the relevant components. Our implementation follows a non-intrusive approach, so it only requires minimum changes to the application code.

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

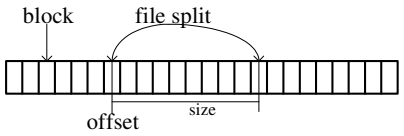

Fig. 2. The high level illustration of a file in DFS.

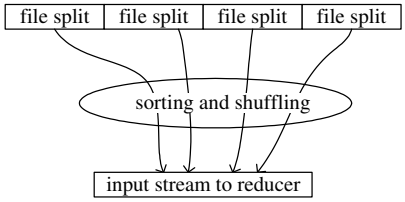

Fig. 3. The input stream to a reducer is obtained by sorting and then shuffling multiple output files of mappers.

task. A piece of cached data is stored in a *distributed file system* (DFS). The content of a cache item is described by its original data and the operations that obtained it from the original data item. Cache descriptions can also be recursive to describe sequential processing on the same data set.

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 semantics. 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. This operation collects the counts of all records in the input.
- Sort. This operation sorts the records of the file.
- Selection. This operation selects an item that meets a given criterion.
- Transform. This operation transforms each item in the input file into a different item.
- Classification. This operation splits input items into multiple groups, which is a deterministic processing that can be repeatedly applied on the same input to produce exactly same results.

B. Reduce Phase Cache Description Scheme

The input for the reduce phase is also a list of key-value pairs, where the value could be a list of values. 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-generated intermediate result from the Map phase to form the final results. For example, assume that "input3.data" is a newly generated results 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.

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. Fortunately, almost all real-world applications have this property.

III. Protocol

A. 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

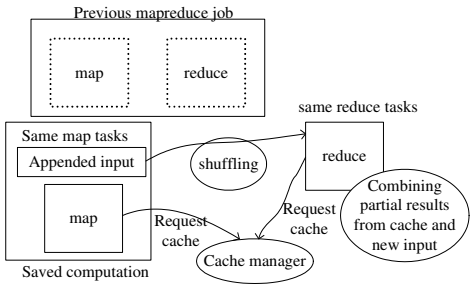

Fig. 4. The situation where two MapReduce jobs have the same map and reduce tasks.

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 than the cached reduce cache items of the previous MapReduce jobs. 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. 4. Using the description scheme discussed in Section II, 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.

B. Cache Item Submission

Mapper and reducer nodes/processes record cache items into their local storage space. When there operations are completed, the cache items are forwarded to the cache manager, which acts like a broker in the publish/subscribe paradigm [6]. The cache manager 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 cache manager 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 cache manager 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 manager. The cache manager receives this message and compares it with the stored mapping data. 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 manager 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 cache manager that it already processed the input file splits for this job. The cache manager 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, which will be explained in Section III-D.

If the proposed operations are different from the cache items in the manager's records, there are situations where the origin of the cache item is the same as the requested file, and the operations of the cache item is a strict subset of the proposed operations. The concept of a *strict super set* refers to the fact that the item is obtained by applying some additional operations on the subset item. For example, an item count operation is a strict subset operation of an item count followed by a selection operation. This fact means that if we have a cache item for the first operation, we could just add the selection operation, which guarantees the correctness of the operation.

C. Lifetime Management of Cache Item

The cache manager 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. There are two types of policies for determining the lifetime of a cache item, as listed below. The cache manager 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 manager. The cache manager still maintains the mapping between cache descriptions and the actual storage location.

1) Fixed Storage Quota: Dache allocates a fixed amount of storage space for storing cache items. Old cache items need to be evicted when there is not enough storage space for storing new cache items. The eviction policy of old cache items can be modeled as a classic cache replacement problem [2]. In our preliminary implementation, the *least recent used* (LRU) is employed. The cost of allocating a fixed storage quota could be determined by a pricing model that captures the monetary expense of using that amount of storage space. Such pricing models are available in a public Cloud service. We discuss more details about the model in Section III-C2.

2) Optimal Utility: Increasing the storage space of cache items will likely hit a plateau due to the diminishing return effect. A utility-based measurement can be used to determine the optimal space used for storing cache items in order to trade off between the benefits and the costs. This scheme estimates the saved computation time, t_s, by caching a cache item for a given amount of time, t_a. These two variables are used to derive the *monetary* gain and cost. The net profit, i.e., the difference of subtracting cost from gain, should be made positive. To accomplish this, an accurate pricing model of computational resources is required. Although conventional computing infrastructures do not offer such a model, Cloud computing does. Monetary values of computational resources are well captured in existing Cloud computing services, i.e., Amazon AWS [1] and Google Compute Engine [3].

$$Expense_{t\_s} = P_{storage} \times S_{cache} \times t\_s \quad (1)$$

$$Save_{t\_s} = P_{computation} \times R_{duplicate} \times t\_s \quad (2)$$

The equations 1 and 2 show how to compute the expense of storing cache and the corresponding saved expense in computation. The details of computing the variables introduced above are as follows. The gain of storing a cache item for t_s amount of time is calculated by accumulating the charged expenses of all the saved computation tasks in t_s. The number of the same task that is submitted by the user in t_s is approximated by an exponential distribution. The mean of this exponential distribution is obtained by sampling in history. A newly-generated cache item requires a bootstrap time to do the sampling. The cost is directly computed from the charge expense of storing the item for t_a amount of time. The optimal lifetime of a cache item is the maximum t_a, such that the profit is positive. The overall benefits of this scheme are that the user will not be charged more and at the same time the computation time is reduced, which in turn reduces the response time and increases the user satisfaction.

D. Cache Request and Reply

1) Map Cache: There are several complications that are caused by the actual designs of the Hadoop MapReduce framework. The first is, when do mappers issue cache requests? As described above, map cache items are identified by cache descriptions, which are not directly corresponding to the files in the HDFS file system. Therefore cache requests must be sent out before the file splitting phase. The jobtracker, which is the central controller that manages a MapReduce job, issues cache requests to the cache manager. The cache manager replies a list of cache descriptions. The jobtracker then splits the input file on remaining file sections that have no corresponding results in the cache items. That is, the jobtracker needs to use the same file split scheme as the one used in the cache items in order to actually utilize them. In this scenario, the new appended input file should be split among the same number of mapper tasks, so that it will not slow the entire MapReduce job down. Their results are then combined together to form an aggregated Map cache item. This could be done by a nested MapReduce job.

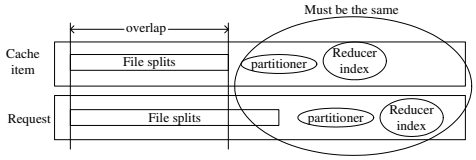

Fig. 5. In order to compare a cache description and a cache request, the cache manager needs to examine the partitioner and the reducer indexes.

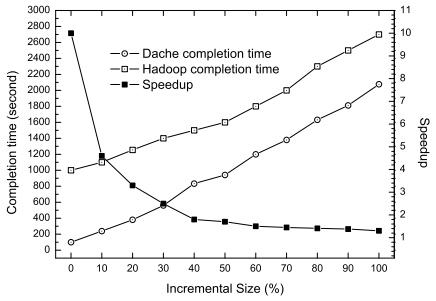

Fig. 6. The speedup of Dache over Hadoop and their completion time of word-count program.

2) Reduce Cache: The cache request process is more complicated. The first step is to compare the requested cache item with the cached items in the cache manager's database. As described in Section II-B, the cached results in the reduce phase may not be directly used due to the incremental changes. As a result, the cache manager needs to identify the overlaps of the original input files of the requested cache and stored cache. In our preliminary implementation, this is done by performing a linear scan of the stored cache items to find the one with the maximum overlap with the request. When comparing the request and cache item, the cache manager first identify the partitioner. The partitioner in the request and the cache item have to be identical, i.e., they should use the same partitioning algorithm and the same number of reducers. This requirement is illustrated in Fig. 5. The overlapped part means that a part of the processing in the reducer could be saved by obtaining the cached results for that part of the input. The incremented part, however, will need to be processed by the reducer itself. The final results are generated by combining both parts. The actual method of combining results is determined by the user.

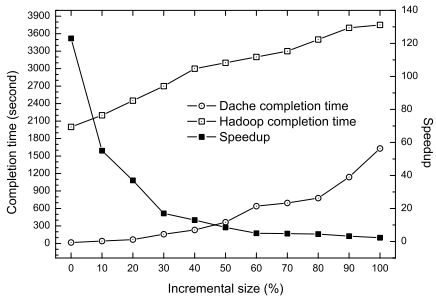

Fig. 7. The speedup of Dache over Hadoop and their completion time of tera-sort program.

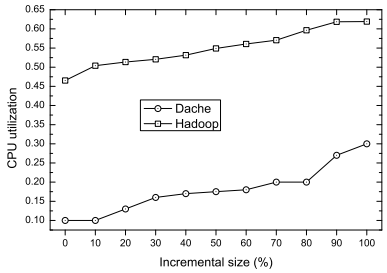

Fig. 8. CPU utlization ratio of Hadoop and Dache in the word-count program.

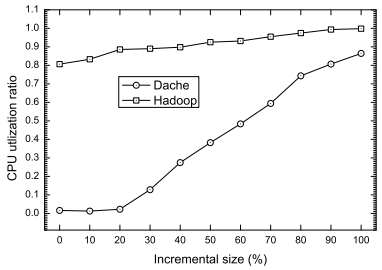

Fig. 9. CPU utlization ratio of Hadoop and Dache in the tera-sort program.

IV. Performance Evaluation

A. Experiment Settings

Hadoop is running in pseudo-distributed mode on a server that has an 8-core CPU, each core running at 3GHz, 16GB 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 Dache over Hadoop (the classic MapReduce model): *word-count* and *tera-sort*. Word-count counts the number of unique words in large input text files; tera-sort sorts key-value records based on the lexical order of the key. More details are in Hadoop manual [4]. Word-count is an IO-intensive application that requires loading and storing a sizeable amount of data during the processing. On the other hand, tera-sort uses more mixed word loads. It needs to load and store all input data and needs a computation-intensive sorting phase. The input of two applications are generated randomly, and all are 10GB in size.

B. Results

Figs. 6 and 7 present the speedup and completion time of two programs. 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 10GB. Tera-sort is more CPU-bound compared to word-count, as a result Dache can bypass computation tasks that take more time, which achieves larger speedups. The speedup decreases with the growing size of appended data, but Dache is able to complete jobs faster than Hadoop in all situations. The map phase of tera-sort does not perform much computation, which also makes it easier for Dache to work.

Figs. 8 and 9 show the CPU utilization ratio of the two programs. It is measured by averaging the CPU utilization ratio of the processes of the MapReduce jobs over time.

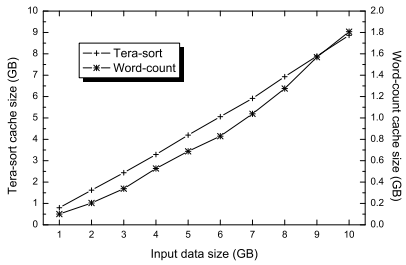

Fig. 10. Total cache size in GB of two programs.

Tera-sort consumes more CPU cycles than word-count does, which is determined by the CPU-bound nature of the sorting procedure. From the figures, it is clear that Dache saves a significant amount of CPU cycles, which is demonstrated by the much lower CPU utilization ratio. These results are consistent with Fig. 6 and 7. With a larger incremental size, the CPU utilization ratio of Dache grows significantly, too. This is because Dache needs to process the new data and cannot utilize any cached results for bypassing computation tasks. Figs. 6, 7, 8, and 9 collectively prove that Dache indeed removes redundant tasks in incremental MapReduce jobs and reduces job completion time.

Fig. 10 presents the size of all the cache items produced by a fresh run of the two programs with different input data sizes. In tera-sort, cache items should have the same size as the original input data because sorting does not remove any data from the input. The difference between the input data size and the cache size is caused by the data compression. Note also that the cache item in tera-sort is really the final output, which means that the used space is free in the sense that no extra cost is incurred in storing cache items. The word-count results are more related to the input record distribution.

V. Conclusion

We present the design and evaluation of Dache, a data-aware caching framework for MapReduce. Dache requires only a slight modification in the input format and task management of the MapReduce framework, and applications needs only slight changes in order to utilize Dache. We implement Dache in Hadoop. Testbed experiments show that it can eliminate all the duplicate tasks in incremental MapReduce jobs and does not require substantial changes to the application code.

Acknowledgment

This research was supported in part by NSF grants ECCS 1231461, ECCS 1128209, CNS 1065444, and CCF 1028167.

References

[1] Amawon web services. http://aws.amazon.com/.
[2] Cache algorithms. http://en.wikipedia.org/wiki/Cache_algorithms.
[3] Google compute engine. http://cloud.google.com/products/compute-engine.html.
[4] Hadoop. http://hadoop.apache.org/.
[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. of ACM*, 51(1):107–113, January 2008.
[6] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.