IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 Issue 3, March 2015.
www.ijiset.com
ISSN 2348 – 7968

Pursuit of Data Aware Caching for Big-Data using MapReduce Framework Proposal

Rudresh.M.S[1], Harish.K.V[2], Shashikala.S.V[3]

[1] P.G. Scholar, Dept. of CSE, B.G.S Institute of Technology, B.G.Nagar, Karnataka, India

[2] Assistant Professor, Dept of CSE, B.G.S Institute of Technology, B.G.Nagar, Karnataka, India

[3] Professor and Head, Dept. of CSE, B.G.S Institute of Technology, B.G.Nagar, Karnataka, India

Abstract
The buzz-word big-data refers to the large-scale distributed data processing applications that operate on exceptionally large amounts of data volumes. For case in point, big data is commonly unstructured and require more real-time analysis strategy. This evaluation calls for new system architectures for data acquisition, storage, transmission and large-scale data processing gadgets. Google incorporates MapReduce concepts and Apache's Hadoop, due to its open-source implementations; it becomes a principal and motives software systems for big-data applications. An inspection of the MapReduce framework is that the framework generates a large amount of intermediate data. In this prodigy, we insinuate, a data-aware caching framework for big-data applications, also the tasks submit their intermediate results to the cache manager. Hence a task queries the cache manager before executing the actual computing work. A narrative cache description scheme and a cache request and reply protocols are designed and adopted. We implemented a data aware cache technique by extending Hadoop. Pragmatic experiment results reveals that data aware cache significantly improves the finishing point time of MapReduce jobs.

Keywords:*Big data paradigm,MapReduce,Hadoop platform applications,Cache techniques.*

1. Introduction

The Google MapReduce [1] is a programming standard and a software framework for large-scale distributed computing on large amounts of data. Fig- 1 presents the high-level work flow of a MapReduce jobs. MapReduce gains popularity for its simple programming interface and excellent performance when implementing a large spectrum of programs. Since most such applications take a large amount of input data, they are nicknamed "Big-data apps". As shown in Fig. 1, input data is first split and then feed to workers in the map phase. Particular data items are called records. The Interrelated system parses the input splits to each worker and produces records. Later the map phase, transitional results generated in the map phase are shuffled and sorted by the MapReduce system and are then fed into the workers in the reduce segment. Absolute results are computed by multiple reducers and written to the disk.

Hadoop [2] is an open-source implementation of the Google MapReduce programming model. It mainly consists of the Hadoop Common GUI, provides access to the file systems supported by Hadoop. Particularly, the Hadoop Distributed File System (HDFS) provides distributed file storage and is optimized for large immutable blobs of data. Typical Hadoop cluster will include a single master and multiple worker segments. The master node runs multiple processes, including a JobTracker and a NameNode. The JobTracker is responsible for managing running jobs in the Hadoop cluster. The NameNode, on the other medium, manages the HDFS.

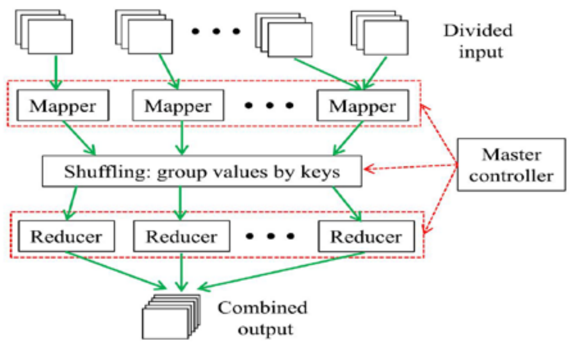

Fig. 1 MapReduce programming model and the underlying implementation architecture.

A MapReduce job is divided into tasks. These are managed by the TaskTracker. Both the TaskTrackers and the DataNode are collated on the same servers to provide data locality in computation. Mapping platform provides a standardized framework for implementing large-scale distributed computation. However, MapReduce does not have the mechanism to identify such duplicate computations and accelerate job execution procedures. Motivated by this observation, we propose a data-aware cache structure for big-data applications using the MapReduce platform. It aims at extending the MapReduce framework and provisioning a cache layer for efficiently identifying and accessing cache items in a MapReduce work series. The following technical challenges need to be addressed before implementing this proposal.

Cache description scheme:
Data-aware caching requires each data object to be indexed by its content. In terms of big-data applications, this means that the cache skeleton description scheme needs to describe the application framework and the data contents. It should provide a customizable indexing that enables the applications to describe their operations and the content of their generated partial results. Using Hadoop source, we utilize the sterilization capability provided by the Java [3] language to identify the object that is used by the MapReduce system to process the input data.

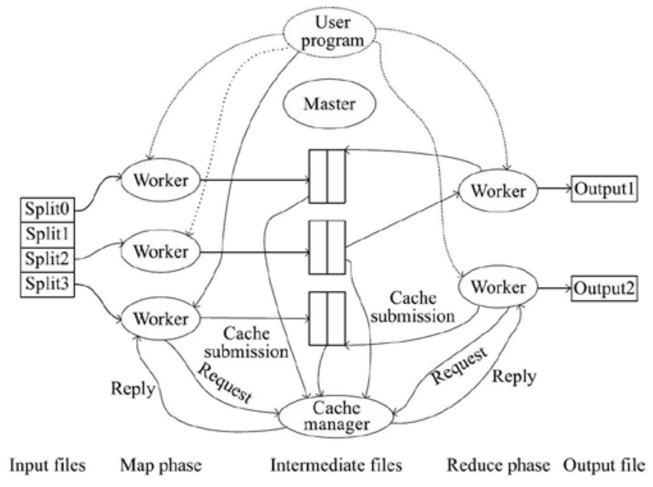

Fig. 2 High-level Architecture of data aware cache.

Cache request and reply protocol:
The size of the aggregated intermediate data can be very large. In such scenarios data is requested by other worker nodes, Estimating how to transport this data becomes complex. Data locality is another concern in this case. The protocol should be able to collate cache items with the worker processes potentially that need the data mods, Hence the transmission delay and overhead are minimized. In this research work, we present a novel cache description scheme. It is depicted in Fig. 2. The scheme identifies the source input from which a cache item is obtained, and the several sequential operations applied on to the input, so that a cache item produced by the workers in the map phase is indexed properly. 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 presented a method for reducers to utilize the cached results in the map phase to accelerate the execution of the MapReduce jobs.

2. Cache Description on Big data Environment

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 function. A part of cached data is stored in a Distributed File System (DFS). Regularly, a cache item is described by a 2-tuple: {Origin, Operation}. Origin is the name of a file in the DFS. Hence the transaction is a linear list of available operations performed on the Origin file. Consider a criteria, in the word count application, every mapper node/process emits a list of {word, count} tuples that record the count of each word in the file that the mapper processes.

The exact format of the cache description of different applications varies according to their specific semantic environments. This can be designed and implemented by application developers who are responsible for implementing their MapReduce tasks. Cache descriptions can be recursive. For an instance, in sequential processing, part of data file could be processed by multiple worker nodes/processes. In such scenarios, a cache item, generated by the final process, it should be from the intermediate result files of a previous worker; hence its description will be stacked together to form a recursive description.

In some cases, this recursive description could be expanded to an iterative one by directly appending the later operations to the older ones. By inquiring an iterative description, one cannot discern between a later cache item and a previous one because the origin of the cache item is the one that was fed by the application developers. By this way, the worker processes will be unable to precisely identify the correct cache item, even if the cache item is readily available.

2.2 Reduce phase cache description scheme

The input for the reduce phase is also a list of key-value combination; here the value could be a list of values. The genuine input is obtained by storing the intermediate results of the map phase in the DFS module. The tested operations are identified by unique IDs that are specified by the user point of view. Hence all the cached results, those are generated in the Map scenario, and it cannot be directly used as the final output. 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 methods. 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. 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. Also, this property is determined by the operations executed by the reducer's node. Successfully, almost all real-world applications have this property.

2.3 Case study with Hadoop MapReduce

A. Map cache

Apache Hadoop [2] (HDMR) is an open-source implementation of the MapReduce distributed parallel processing algorithm originally designed by Google. Map phase is a data-parallel processing procedure in which the input is split into multiple file splits which are then processed by an equal number of Map worker processes. As depicted in Fig. 3, a file split divides one or more input files based on user-supplied rules. The intermediate results obtained by processing file splits should be cached. Each file split is identified by the original file name, offset, and size. This identification scheme causes complications in describing cache items. In reality, such a situation is seldom the case.

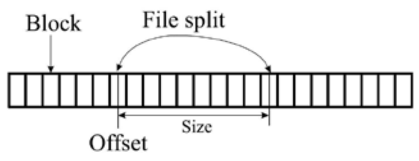

Fig. 3 A file in a DFS, stored as multiple blocks, which are fixed-size data blocks with File split and offset size segments.

This identification scheme causes complications in describing cache items. In reality, such a situation is seldom the case. This scheme is slightly modified to work for the general situation. The original field of a cache item is changed to a 3-tuple of {file name, offset, size}. Note that a file split cannot cross file boundaries in Hadoop paradigm, which mainly simplifies the description scheme of cache items.The operation field required by the cache description is described by a serialized Java object. This field is read by the Java program and tested against known Java class definitions to determine what operations are used.

B. Reduce cache

Cache description in the reduce phase follows the designs in Section 2(b). 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. Note that even the entire output of the input files of a MapReduce job is used in the reduce phase, Hence the file splits can still be aggregated as described in Section 2(a), i.e., by using the form of {file name, split, … , split}.

As shown in Fig. 4, 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 would be implemented as a Java object in Hadoop.

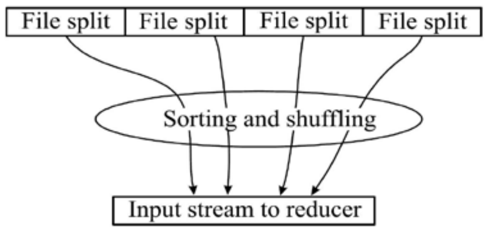

Fig. 4 The input stream to a reducer is obtained by sorting and then shuffling multiple output files of mappers.

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. At last, the index of the reducer assigned by the partitioner is attached. The whole description is a 3-tuple: {file splits, partitioner, reducer index}. The description is completed to accurately identify the input to a reducer. However, this process is automatically handled by the reducers.

3. Interactive Protocols in Cache Organization

3.1 Relationship between job types and cache organization

The partial results generated in the map and reduce phases can be utilized in different cases. There will be two types of cache items: the map cache and the reduce cache. Cache items in the map phase are easy to share because the operations applied are generally well-formed. While processing each file split operation, cache manager reports the previous file splitting scheme used in its cache item. When considering cache sharing in the reduce phase scenario, we come across two general situations. First thing is when the reducers' complete different jobs from the cached reduce cache items of the previous MapReduce jobs, as shown in Fig. 5. In this time period, after the mappers submit the results obtained from the cache items queue, the MapReduce frames uses the partitioner provided by the new MapReduce job to feed input to the reducers.

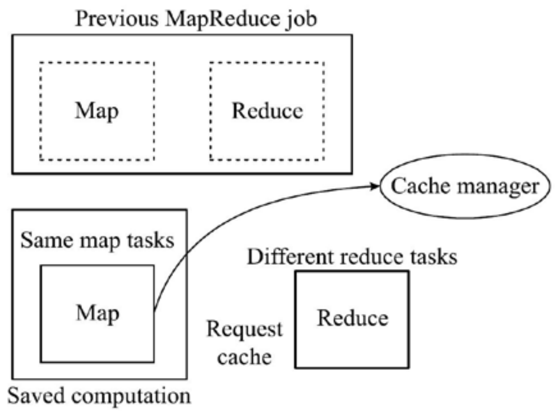

Fig. 5 The situation where two MapReduce jobs have the same map tasks, which mainly saves a fraction of computation by requesting caches from the cache manager.

The second situation is when the reducers can actually take advantage of the previously-cached reducing cache items as illustrated in Fig. 6. Using the description scheme discussed in Section 2, the reducers determine how the output of the map phase is shuffled.

3.2 Cache item submission

Mapper and reducer nodes/processes record cache items into their local storage space. When this type of operations are get terminated, remaining cache items are forwarded to the cache manager area, which operates like a broker in the publish/subscribe paradigm [4]. The cache manager records the description and the file name of the cache item in the DFS.A worker node/process contacts the cache manager each time before it begins processing an each input data file for further execution. The worker process sends the file name and the operations that it plans to apply to the file to the cache manager. It mainly receives this message and compares it with the stored mapping data.

Later the worker process receives the tentative description and fetches the cache item. The mapper needs to inform the cache manager that it already processed the input file splits for this job execution. Than the cache manager then reports these results to the next phase reducers. If the proposed operations are different from the cache items in the manager's records, in some situations where the origin of the cache item is the same as the requested file, and similarly the operations of the cache item are a strict subset of the proposed operations.

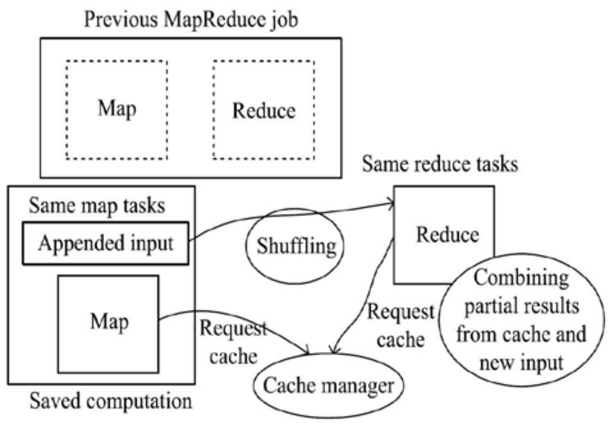

Fig. 6 The situation where two MapReduce jobs have the same map and reduce tasks. The reducers combine results from the cache items and the appended input to produce the final results.

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 e.g, each item count operation is a strict subset operation of an item count followed by a selection operation. One of the benefits of data aware caching structure is that it automatically supports incremental processing. It means that we have an input that is partially different or only has a small amount of additional data items. To perform a last operation on this new input data is troublesome in conventional MapReduce techniques, because it does not provide the tools for readily expressing such incremental operations.

3.3 Lifetime management of cache item

The cache manager needs to determine how much time a cache item can be kept in the DFS boundary. 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 execution. Here the two types of policies for determining the lifetime of a cache item, as mentioned below.

a) Fixed storage quota

Data aware cache feature allocates a fixed amount of storage space for storing cache items queue. Older cache items need to be evicted when there is no enough storage space for storing new cache items. In our preliminary implementation, the Least Recent Used (LRU) is implied. Hence 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 segments. Such valuable models are available in a public Cloud service.

b) Optimal utility

A utility-based measurement can be used to determine an optimal space allocated for cache items which maximizes the benefits of data aware caching and respect the constraints of costs. This scheme estimates the saved processing time interval, t_s, by caching a cache item for a given amount of period, t_a. Hence two variables are used to derive the monetary gain and cost prices. The net profit, i.e., the difference of subtracting cost from gain, should be made positive. To establish this, exact pricing models of computational resources are needed. Monetary values of computational resources are well captured in existing cloud computing services, for example, in Amazon AWS [6] and Google Compute Engine [7]. On the other hand, for organizations those rely on their own private IT paradigm, this model structure will be inaccurate and should only be used as a reference.

$$\text{Expense}_{\text{ts}} = P_{\text{storage}} \times S_{\text{cache}} \times t_{\text{s}} \quad (1)$$

$$\text{Save}_{\text{ts}} = P_{\text{computation}} \times R_{\text{duplicate}} \times t_{\text{s}} \quad (2)$$

Equations (1) and (2) show how to compute the expense of storing cache and the corresponding saved expense in computation procedure. The steps for computing the variables introduced above are as follows. Since 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 interval of time. The number of the same task that is submitted by the user in t_s is approximated by an exponential distribution. The cost is directly computed from the charge expense of storing the item for t_a amount of time interval. The optimal lifetime of a cache item is the maximum t_a, such that the profit is positive in nature. Hence overall benefits of this scheme are that the user will not be charged more and at the same time the computation time is minimized, which mainly reduces the response time and increases the user satisfaction.

4. Performance Evaluation in Hadoop Policy

A. Platform Implementation

We extend Hadoop to implement Data aware cache concepts. 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 nonintrusive approach to implement this in Hadoop and try not to hack the Hadoop framework itself, but implement Data aware cache by changing the components that are open to application developers. Basically, the cache manager is implemented as an independent server. It communicates with task trackers and provides cache items on receiving requests. The cache manager stands outside of the Hadoop MapReduce framework. The cache manager uses HDFS, the DFS component of Hadoop, to manage the storage of cache items.

In order to access cache items, the mapper and reducer tasks first send requests to the cache manager. 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. We alter two components of Hadoop to implement this function. The first component is InputFormat class, an open-accessed component that allows application developers to modify. It is responsible for splitting the input files of the MapReduce job to multiple file splits and parse data to key-value pairs. The second component that needs to be altered is the TaskTracker, which is the class responsible for managing tasks. TaskTracker is able to understand filesplit and bypass the execution of mapper classes entirely. 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. Experimental setups

Hadoop is run in pseudo-distributed mode on a server that has an 8-core CPU, each core running at 3 GHz, 16GB memory, and a SATA disk. The number of mappers is 16 in all experimental activities, the reducers' count varies. We mainly use two applications to benchmark the speedup of data aware caching over Hadoop (the classic MapReduce model): word-count and tera-sort fuctions. Here 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 [2]. Word-count is an IO-intensive application that requires loading and storing a sizeable amount of data during the processing time. In another way, tera-sort uses more mixed word loads in its operations. It mainly needs to load and store all input data and needs a computation-intensive sorting phase. The inputs of two applications are generated dynamically, and all are 10GB in size.

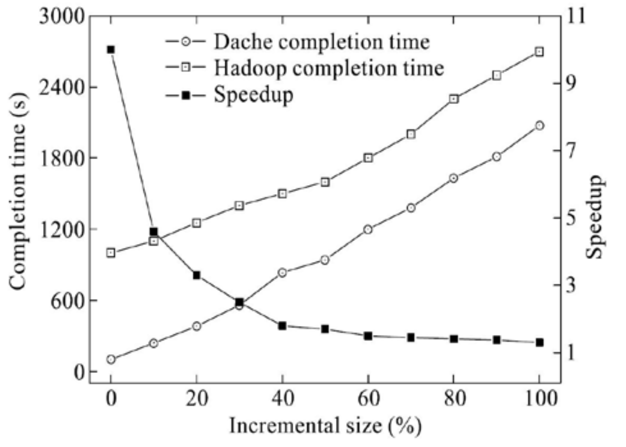

Fig. 8 The speedup of Dache over Hadoop and their completion time of word-count program.

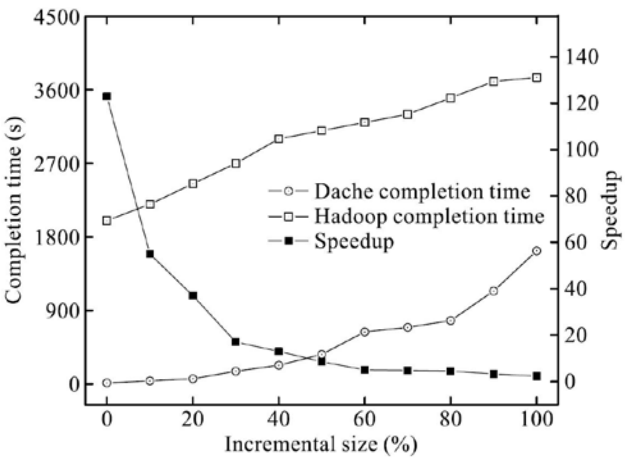

Fig. 9 The speedup of Dache over Hadoop and their completion time of tera-sort program.

C. Results

Figures 8 and 9 present the speedup and completion time of two programs execution. The completion time and the speedup are put combinable. Data is appended to the input file storage space,and The size of the appended data varies and is represented as a percentage number to the original input file size occupied, which is 10 GB. Tera-sort is more CPUbound compared to word-count; as a result Dache can bypass computation tasks that take more time to process, which achieves larger speedups in execution. The speedup decreases with the growing size of appended data in a processing queue, but Dache is able to complete jobs faster than Hadoop in all criteria’s. The map phase of tera-sort does not perform much computation process, which also makes it easier for Dache to work in a consistent environment.

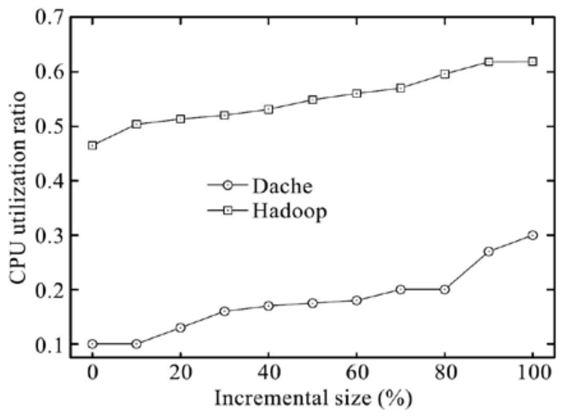

Fig. 10 CPU utilization ratio of Hadoop and Dache in the word-count program.

Figures 10 and 11 shows the CPU utilization ratio of the two programs.

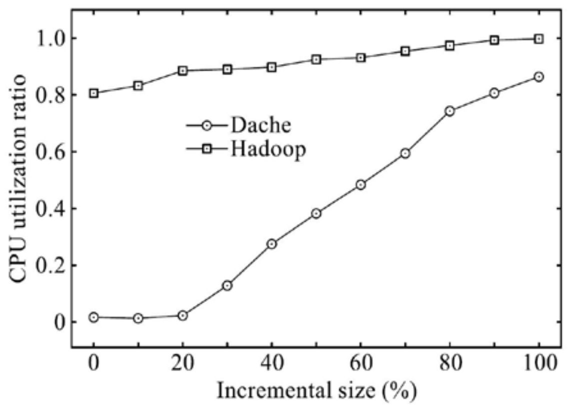

Fig. 11 CPU utilization ratio of Hadoop and Dache in the tera-sort program.

From the figures, it is clear that Dache saves a significant amount of CPU cycles calculated for their Performance, which is demonstrated by the much lower CPU utilization ratio. These results are consistent with Figs. 8 and 9. With a larger incremental size, Hence the CPU utilization ratio of Dache grows significantly,This is due to Dache needs to process the new data and cannot utilize any cached results for bypassing computation tasks. Figures 8-11 collectively prove that Dache indeed removes redundant tasks in incremental MapReduce jobs and reduces job completion time.

Figure 12 presents the size of all the cache items produced by a fresh run of the two programs with different input data sizes in the cache area. 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 queue. The differentiation of input data size and the cache size is caused by the data compression process. Observe that the cache item in tera-sort is really the final output results, which tells that the used space is free in the sense that no extra cost is incurred in storing cache items in the Dache memory

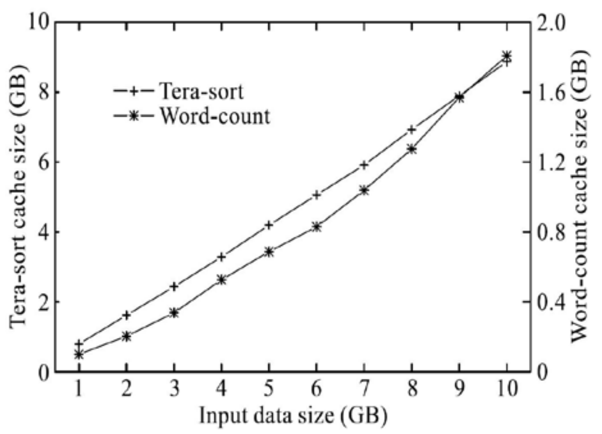

Fig. 12 Total cache size in GB of two programs.

5. Conclusions

We finally 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. We propose Dache, a data-aware cache description scheme, protocol, and architecture. Our method requires only a slight modification in the input format processing and task management of the MapReduce frames.Hence, application code only requires slight changes in order to utilize Dache. We implement Dache in Hadoop by extending relevant components. 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. In the future, we plan to adapt our framework to more general application scenarios and implement the scheme in the Hadoop project.

Acknowledgments

This dedicated research is a combined effort of me and our Guide/HOD of BGS Institute of Technology. I would like to gratefully acknowledge my organization and my guide Dr.Ravikumar.G.K for his valuable suggestion and support.

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.
[5] Cache algorithms, http://en.wikipedia.org/wiki/Cache algorithms, 2013.
[6] Amawon web services, http://aws.amazon.com/, 2013.
[7] Google compute engine, http://cloud.google.com/products/computeengine.html, 2013.
[8] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation, in Proc. of POPL '93, New York, NY, USA, 1993.
[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data, in Proc. of OSDI'2006, Berkeley, CA, USA, 2006.
[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, The google file system, SIGOPS Oper. Syst. Rev., vol. 37, no. 5, pp. 29-43, 2003.
[11] D. Peng and F. Dabek, Largescale incremental processing using distributed transactions and notifications, in Proc. Of OSDI' 2010, Berkeley, CA, USA, 2010.
[12] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazi‘eres, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, Stratmann, and R. Stutsman, The case for ramcloud, Commun. of ACM, vol. 54, no. 7, pp. 121-130, 2011.
[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, SIGOPS Oper. Syst. Rev., vol. 41, no. 3, pp. 59-72, 2007.
[14] L. Popa, M. Budiu, Y. Yu, and M. Isard, Dryadinc: Reusing work in large-scale computations, in Proc. Of HotCloud'09, Berkeley, CA, USA, 2009.
[15] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, Nova: Continuous pig/hadoop workflows, in Proc. of SIGMOD'2011, New York, NY, USA, 2011.
[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, Pig latin: A not-so-foreign language for data processing, in Proc. of SIGMOD'2008, New York, NY,USA, 2008.

About Authors:

Rudresh.M.S received the B.E degree in Information Science from Visvesvaraya Technological University Karnataka in 2011, also M.S degree in Computing Technologies and Virtualization from Manipal University in 2013, and currently he is a post graduate student pursuing M.Tech in Computer Science and Engineering from B.G.S Institute of Technology under Visvesvaraya Technological University Karnataka.

He has presented 11 papers in National level conferences and also presented 2 papers in international conference, He has published 4 papers in International Journals. His main research interests include cloud computing and Big data as well as computer networks and virtualization. He is currently doing his research project in Big data with distributed sensor networks.

Harish.K.V received the B.E degree in Computer Science from Visvesvaraya Technological University Karnataka in 2011, also M.Tech degree in Networking and Internet from VTU in 2013, and currently he is working as an Assistant Professor in the department of computer science at B.G.S Institute of Technology under Visvesvaraya Technological University Karnataka.

He has presented 3 papers in National level conferences and he has published 1 paper in International Journal. He is having a totally 2 years of teaching experience. His main research interests include computer networks and internet security as well as cloud computing and big data.

Shashikala.S.V received her B.E degree in Computer Science from Mysore University, Karnataka in 1990 and M. Tech in Computer Science and Engineering from Visvesvaraya Technological University in 2005; she is currently working towards her PhD degree in the area of Big data over a multipath TCP.
She is a Professor and Head in the department of computer science at B.G.S Institute of Technology; B.G.Nagar Mandya and having totally 22 years of teaching experience. She has presented 5 papers in National level conferences and also presented 2 papers in International conferences, she has published 4 papers in International Journals. Her main research interests include computer networks, Big data and data mining.