ISSN 2319 - 6629
Volume 3, No.4, June – July 2014
International Journal of Wireless Communications and Networking Technologies
Available Online at http://warse.org/pdfs/2014/ijwcnt01342014.pdf

A Data Aware Caching for Large scale Data Applications Using The Map-Reduce

Rupali V. Pashte[1]
Student ME Computer Engineering, SP' IOKCOE, Pune, India
r.patil0983@gmail.com

ABSTRACT

The Big-data refers to the huge scale distributed data processing applications that operate on unusually large amounts of data. Google's MapReduce and Apache's MapReduce, its open-source implementation, are the defacto software systems for Large Scale data applications. Study of the MapReduce framework is that the framework generates a large amount of intermediate data. Such existing information is thrown away after the tasks finish, because MapReduce is not able to utilize them. In this paper, we propose, a data-aware cache framework for large data applications. In this paper, tasks submit their intermediate results to the cache manager. A job queries the cache manager before executing the actual computing work. A novel cache description scheme and a cache request and reply protocols are designed. We implement Data aware caching by extending Hadoop.
Keywords: Big-data, Hadoop, JobTracker, MapReduce, TaskTracker

1. INTRODUCTION

Google MapReduce is a programming model and a software framework for Big -scale distributed Computing on large amounts of data. Figure 1 illustrates the high-level work flow of a MapReduce Task. Application developers specify the computation in terms of a map and a reduce function, and the underlying MapReduce Task scheduling system automatically parallelizes the computation across a cluster of machines. MapReduce obtain popularity for its simple programming interface and excellent Performance when implementing a large spectrum of applications. Since most such applications take a huge amount of input data, they are named as "Big-data applications".

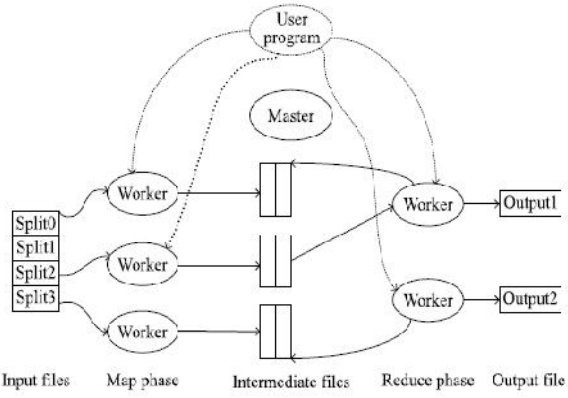

Figure 1: The MapReduce programming model architecture.

As shown in Figure 1, input data is first divided and then given to workers in the map stage. 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 given into the workers in the reduce phase. Final results are computed by multiple reducers and written back to the disk.

Hadoop is an open-source implementation of the Google MapReduce programming model. Hadoop consists of the Hadoop Common, which provides access to the file systems supported by Hadoop. Hadoop Distributed File System (HDFS) provides distributed file storage and is optimized for large unchangable blobs of data. A small Hadoop cluster will include a single master and multiple worker nodes called as slave. The master node runs multiple processes, including a TaskTracker and a Name Node. The TaskTracker is having control for managing running jobs in the Hadoop cluster. Whereas the Name Node manages the HDFS. The TaskTracker and the Name Node are usually collocated on the same physical machine. Other servers in the cluster run a Task Tracker and a Data Node processes. A MapReduce job is divided into tasks. Tasks are managed by the TaskTracker. The TaskTrackers and the DataNode are collated on the same servers to provide data locality in computation. MapReduce provides a standardized framework for implementing large-scale distributed computation, called as, the big-data applications.

Still, there is a restriction 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 produce output. There are potential duplicate computations and operations being performed in this process. However, MapReduce does not have the any technique to identify such duplicate computations and accelerate job execution. Motivated by this observation, In this paper we propose, a data-aware cache system for big-data applications using the MapReduce framework, which aims at extending the MapReduce framework and provisioning a cache layer for efficiently identifying and accessing cache items in a MapReduce job.

2. LITERATURE REVIEW

1. Large-scale Incremental Processing Using Distributed Transactions and Notifications [3]

Daniel Peng et al. proposed, a system for incrementally processing updates to a large data set, and deployed it to

create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, Auther process the same number of documents per day.

2. Design and Evaluation of Network-Leviated Merge for Hadoop Acceleration [7]
Weikuan Yu et al. proposed, Hadoop-A, an acceleration framework that optimizes Hadoop with plugin components for fast data movement, overcoming the existing limitations. A novel network-levitated merge algorithm is introduced to merge data without repetition and disk access. In addition, a full pipeline is designed to overlap the shuffle, merge and reduce phases. Our experimental results show that Hadoop-A significantly speeds up data movement in MapReduce and doubles the throughput of Hadoop.

3. Improving Mapreduce Performance through Data Placement in Heterogeneous Hadoop Cluster [5]
Jiong Xie et al. proposed that ignoring the data locality issue in heterogeneous environments can noticeably reduce the MapReduce performance. In this paper, author addresses the problem of how to place data across nodes in a way that each node has a balanced data processing load. Given a data intensive application running on a Hadoop MapReduce cluster, our data placement scheme adaptively balances the amount of data stored in each node to achieve improved data-processing performance. Experimental results on two real data-intensive applications show that our data placement strategy can always improve the MapReduce performance by rebalancing data across nodes before performing a data-intensive application in a heterogeneous Hadoop cluster.
4. Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization [6]
Zhenhua Guo et al. proposed, Benefit Aware Speculative Execution which predicts the benefit of launching new speculative tasks and greatly eliminates unnecessary runs of speculative tasks. Finally, MapReduce is mainly optimized for homogeneous environments and its inefficiency in heterogeneous network environments has been observed in their experiments. Authors investigate network heterogeneity aware scheduling of both map and reduce tasks. Overall, the goal is to enhance Hadoop to cope with significant system heterogeneity and improve resource utilization.

3. MOTIVATION

MapReduce provides a standardized framework for implementing large-scale distributed computation, called as, 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 technique to identify such duplicate computations and accelerate Task execution. Motivated by this observation, in this paper we propose, a data-aware cache system for big-data applications using the MapReduce framework, which aims at extending the MapReduce framework and provide a cache layer for efficiently identifying and accessing cache items in a MapReduce job.

4. NEED

4.1 Cache Description:
Data-aware caching requires each data object to be indexed by its content. In the context of large scale data applications, this means that the cache description scheme needs to narrate 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. The cache description scheme should provide a customizable indexing that enables content of their generated partial results. This is a nontrivial task. In the context of Hadoop, It utilizes the sterilization capability provided by the Java language to identify the object that is used by the MapReduce system to process the input data.

4.2 Cache request and reply protocol:
The size of the aggregated intermediate data can be very large. When such data is requested by other worker nodes or slave, determining how to transport this data becomes very complex. Normally for processing, program are moved to data node i.e. slave node to run the processing locally. Although, this may not always be applicable since the affinity of the worker nodes may not be easily changed. To solve Data locality problem, 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. In this paper, we present a novel cache description scheme. A high-level description is presented in Figure. 2.

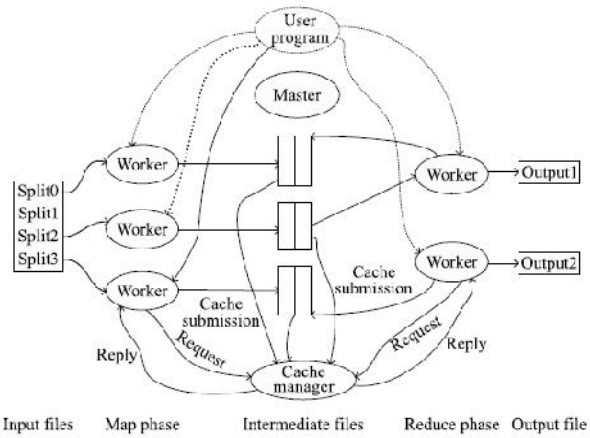

Figure 2: High-level description of the architecture of Data aware caching.

This scheme identifies the source input from which a cache item is acquired, and the operations applied on 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 technique 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 the execution of the MapReduce job. We implement data aware caching 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.

5. CACHE DESCRIPTION

5.1 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: fOrigin, 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 fword, countg tuples that record the count of each word in the file that the mapper processes. Data aware caching stores this list to a file. This file becomes a cache item. Given an original input data file, word list 01237850.txt, the cache item is described by fword list 001237850.txt, item countg. 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, I present several supported operations:

_ **Item Count**: This operation used to 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 the file. The comparison operator is defined on two items and returns the order of priority.

_ **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 format. 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 used to 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.

Cache descriptions can be recursive. For example, in sequential processing, a data file could be processed by multiple worker processes. After that a cache item, generated by the final process, could be from the intermediate result files of previous worker nodes, so its description will be stacked together to form a recursive description. Whereas on other side, this recursive description could be expanded to an iterative one by directly appending the later operations to the older ones. Still, this iterative description loses the context information about the later operations, that is, if another process is operating on a later cache item and is looking for potential cache that could save its own operations. By inspecting an iterative description, one cannot discern between a later on cache item and a previous one because the origin of the cache item is the one that was fed by the application developers. This way, the worker processes will be not able to identify the correct cache item, even if the cache item is present in cache manager.

5.2 Reduce phase cache description scheme:

The input for the reduce phase is 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 item is retrieved by storing the intermediate results of the map phase in the DFS. The performed 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 of an incremental processing, intermediate results generated in the Map phase are mixed in the shuffling phase, which causes a mismatch between the original input of the cache items and the newly generated input. A solution to this problem is apply a finer description of the original input of the cache items in the reduce stage. The description should include the original data files generated in the Map stage. 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 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. Given the above description, the input given to the reducers is not cached wholly. Only a some part of the input is same to the input of the cache items. The remaining is from the output of the incremental processing phase of the map phase. 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. This property is examined by the operations executed by the reducers.

6. HADOOP MAP-REDUCE

6.1 Apache Hadoop

Apache Hadoop is an open-source software framework for storage and processing of large-scale data-sets on clusters of commodity hardware.

'Map-Reduce' is a framework for processing parallelizable problems across large datasets using a large number of nodes, collectively referred to as a cluster or a grid . Computational processing can occur on data stored either in unstructured a file system or in a structured database. Map-Reduce can take advantage of data locality, processing it on or near the storage assets in order to reduce the distance of transmitted.

"Map" step: Input is given to the master node, which divides it into smaller sub-problems, and distributes them to worker nodes. If required A worker node may again further sub-divide it, leading to a multi-level tree structure. The worker node processes the smaller sub-problem, and gives the answer back to its master node.

"Reduce" step: The master node collects the answers of all the sub-problems and combines them to form the output which is the answer to the original problem it was trying to solve.

Map Reduce allows for distributed processing of the map and reduction operations, where mapping operation is independent of the others, all maps can be performed in parallel. Similarly, a set of 'reducers' can perform the reduction phase, if and only if all outputs of the map operation that share the same key with to the same reducer at the same time. Larger dataset is used in Map Reduce, commodity server handle peta byte of data in few hours. If one mapper or reducer fails, then rescheduled is used to assuming the input data is still available.

6.2 Task Tracker: the Map-Reduce engine

The Map-Reduce engine consists of one *JobTracker*, to which client applications submit MapReduce jobs. The JobTracker sends work request to number of TaskTracker nodes in the cluster. To achieve data locality, works are process near to worker node. A rack-aware file system is used, the JobTracker maintain information about node which contains the data, and which is nearby machines. If the work cannot be performed on the actual node where the data resides, priority is given to nodes in the same rack in rack aware file system. This reduces network traffic on the main backbone network. If a TaskTracker fails or times out then that part of the Task is rescheduled. To check the status TaskTracker, A heartbeat is sent from the TaskTracker to the JobTracker every few minutes.

6.3 Map cache:

Apache Hadoop is an open-source implementation of originally implemented by Google, is the MapReduce distributed parallel processing algorithm. In Map phase input is divided into multiple file splits which are then processed by an equal number of Map worker processes, who achieve a data-parallel processing procedure. As depicted in Figure. 3, a file splited according to user given rules. The intermediate results obtained by processing file splits are then cached. Each file split is identified by the original file name, offset, and size. This causes complications in describing cache items. Further this scheme is slightly modified to work for the general situation, In which The original field of a cache item is changed to a 3-tuple of ffile name, offset, size.

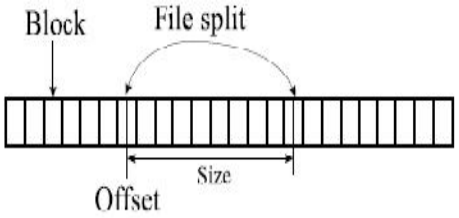

Figure 3: Map Phase A file in a DFS.

A file split cannot cross file boundaries in Hadoop MapReduce, which simplifies the description scheme of cache items. Map cache items can be aggregated by grouping file splits. Original input file generate Multiple cache items from the same original file in the DFS are grouped under the path of the original file, i.e., file name, offset, sizeg, .Using this approach it optimize the actual storage of aggregated cache items .So Map cached item could be put on a single data node in the HDFS cluster which avoid costly queries to multiple data nodes.

6.4 Reduce cache

Cache description contains the file splits from the map phase. The input given to the reducers is from the whole input of the MapReduce job. Therefore, further simplify the description by using the file name 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 to input file. A straightforward approach is to encode the size of the input file is included with the file name. Since incremental changes, appending new data at the end of the file, 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 Task is used in the reduce phase, the file splits can still be aggregated, i.e., by using the form of ffile name, split, splitg. As shown in Figure. 4, file splits are sorted and shuffled to produce the input for the reducers. Basically this process is implicitly handled by the MapReduce framework, the users specify a shuffling method by supplying a partitioner, which is implemented as a Java object in Hadoop.

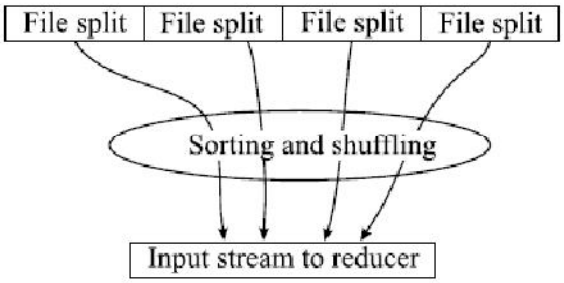

Figure 4: Architecture of Reducer

7. PROTOCOL

7.1 Relationship between Task types and cache organization

The partial results generated in the map and reduce phases can be used in different scenarios. There are two types of cache items: the map cache and the reduce cache. They have different complexities under different scenarios. Cache items in the map phase are easy to share because the operations applied are well-formed. When processing each file split, the cache manager reports the previous file splitting scheme used in its cache item. The new MapReduce Task needs to split the files according to the same splitting scheme in order to utilize the cache items. However, if the new MapReduce Taskuses 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 reducer's complete different jobs from the cached reduce cache items of the previous MapReduce jobs, as shown in Figure. 5. 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 Taskto 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 reducing cache items as illustrated in Figure. 6. Using the description scheme, 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.

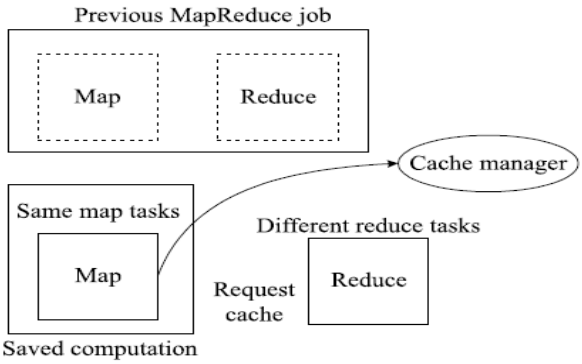

Figure 5: Map with same map task and different reduce tasks

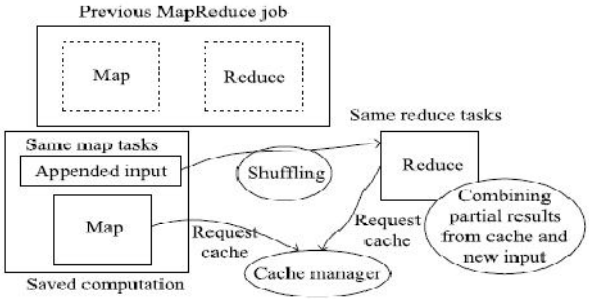

Figure 6: The situation where two MapReduce jobs have the same map and reduce tasks.

7.2 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 cache manager, which acts like a broker in the publish/subscribe paradigm. 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 accommodate fastest reply to queries. It also takes backup of 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; this will be explained in next Section. 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 are 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. One of the benefits of Data aware caching 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 Data aware caching, 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 cache manager.

7.2.1 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 no other MapReduce task utilizing the intermediate results of the cache item. There are two types of schemes 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.

7.2.2 Fixed storage quota:
Data aware caching allocates a fixed amount of storage space for storing cache items. Old cache items need to be removed out when there is no enough storage space for storing new cache items. The removal policy of old cache items can be modeled as a classic cache replacement problem. In this paper 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.

7.2.3 Optimal utility:
Increasing the storage space of cache items, a utility-based measurement can be used to determine an optimal space allocated for cache items which maximize the benefits of Data aware caching and respect the constraints of costs. This approch estimates the saved computation time, ts, by caching a cache item for a given amount of time, ta. 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 achieve this, an accurate pricing model of computational resources is required. Although traditional computing infrastructures do not offer such a model, cloud computing offer. Monetary values of computational resources are well captured in existing cloud computing services, for example, in Amazon AWS and Google Compute Engine. For many organizations that rely on a cloud service provider for their IT infrastructure, this would be a perfect model. According to the official report from Amazon AWS, the amount of organizations that are actively using their services is huge, which help them to achieve near billion dollar revenue. Therefore, this pricing model should be very useful in real-world application. On the other hand, for organizations that rely on their own private IT infrastructure, this model will be inaccurate and should only be used as a reference.

$$Expense_{ts=} P_{storage} \times S_{cache} \times t_s \quad (1)$$
$$Save_{ts} = P_{computation} \times Rduplicate \times ts \quad (2)$$

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 ts amount of time is calculated by accumulating the charged expenses of all the saved computation tasks in ts. The number of the same task that is submitted by the user in ts 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 ta amount of time. The optimal lifetime of a cache item is the maximum ta, 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.

7.3 Cache request and reply
7.3.1 Map cache:
There are several complications that are caused by the actual designs of the Hadoop MapReduce framework. The first is, when do map phase issue cache requests? As described above, map cache items are identified by the data chunk and operations performed. In order to preserve the original splitting scheme, 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 map phase tasks, so that it will not slow the entire MapReduce Task down. Their results are then combined together to form an aggregated Map cache item; to achive this nested MapReduce job is used.

7.3.2 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. 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 identifies the petitioner. The practitioner in the request and the cache item has to be identical, i.e., they should use the same partitioning algorithm and the same number of reducers. This requirement is illustrated in Figure. 7. 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.

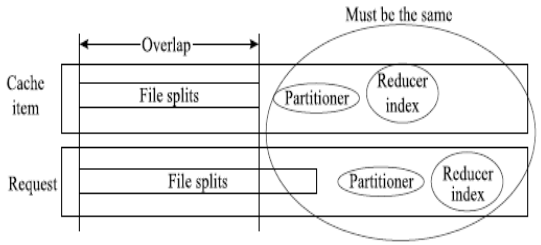

Figure 7: Working of Cache manager and reducer

8. CONCLUSIONS

This paper 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 Big data applications using the MapReduce model. In this paper propose, a data-aware cache description scheme, protocol, and architecture. In this Paper Presented method requires only a slight modification in the input format processing and task management of the MapReduce framework. As a result, application code only requires slight changes in order to utilize Data in data aware caching. This paper implements it in Hadoop by extending relevant components. In the future, we plan to adapt our framework to more general application scenarios and implement the scheme in the Hadoop project.

REFERENCES

[1] Yaxiong Zhao, Jie Wu, and Cong Liu, "Dache: A Data Aware Caching for Big- Data Applications Using the MapReduce Framework" , TSINGHUA SCIENCE AND TECHNOLOGY ISSN 1007-0214 05/10 Volume 19, Number 1, pp 39- 50, February 2014
[2] Hadoop, http://hadoop.apache.org/,2013
[3] D. Peng and f. Dabek,"Large Scale incremental Processing using distributed Transaction and notification", in Proc. of OSDI'2010, Berkeley, CA, USA, 2010
[4] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving Mapreduce performance in heterogeneous environments",in Proc. of OSDI' 2008, Berkeley, CA, USA, 2008.
[5] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, "Improving MapReduce Performance through Data Placement in Heterogeneous Hadoop Clusters", *Department of Computer Science and Software Engineering Auburn University, Auburn, AL 36849-5347*
[6] Zhenhua Guo, Geoffrey Fox "Improving MapReduce Performance in Heterogeneous Network Environments and Resource Utilization" School of Informatics and Computing Indiana University Bloomington Bloomington, IN USA
[7] Weikuan Yu, *Member, IEEE,* Yandong Wang, and Xinyu Que, "Design and Evaluation of Network-Levitated Merge for Hadoop Acceleration", IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS
[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. E. 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.