JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 27, 1137-1152 (2011)

Short Paper

Improving MapReduce Performance by Exploiting Input Redundancy*

SHIN-GYU KIM, HYUCK HAN, HYUNGSOO JUNG+,
HYEONSANG EOM AND HEON Y. YEOM
School of Computer Science and Engineering
Seoul National University
Seoul, 151-744, Korea
+*School of Information Technologies*
University of Sydney
NSW 2006, Australia

The proliferation of data parallel programming on large clusters has set a new research avenue: accommodating numerous types of data-intensive applications with a feasible plan. Behind the many research efforts, we can observe that there exists a nontrivial amount of redundant I/O in the execution of data-intensive applications. This redundancy problem arises as an emerging issue in the recent literature because even the locality-aware scheduling policy in a MapReduce framework is not effective in a cluster environment where storage nodes cannot provide a computation service. In this article, we introduce SplitCache for improving the performance of data-intensive OLAP-style applications by reducing redundant I/O in a MapReduce framework. The key strategy to achieve the goal is to eliminate such I/O redundancy especially when different applications read common input data within an overlapped time period; SplitCache caches the first input stream in the computing nodes and reuses them for future demands. We also design a cache-aware task scheduler that plays an important role in achieving high cache utilization. In execution of the TPC-H benchmark, we achieved 64.3% faster execution and 83.48% reduction in network traffic in average.

Keywords: mapreduce, I/O redundancy, task scheduling, distributed system, cloud computing

1. INTRODUCTION

MapReduce [11] is Google's programming model that allows developers to easily build scalable and high-performance applications that process very large data on clusters of computers; it has proven to be very attractive for parallel processing of arbitrary data. MapReduce, upon the request for large computation, separates large computation into small tasks that can be executed in parallel on multiple machines, and that scales easily to clusters of commodity computers. For example, MapReduce can run data analysis work in parallel on data centers despite failures during the computation. The MapReduce

Received November 30, 2009; revised February 26 & July 1, 2010; accepted August 5, 2010.
Communicated by Chung-Ta King.
* This work was supported by the National Research Foundation (NRF) grant funded by the Korea government (MEST) (No. 2010-0014387). The ICT at Seoul National University provided research facilities for this study.
+ Corresponding author.

frame-work is initially conceived to improve data analysis of multi terabyte index data at Google, and was adopted by various fields of data analysis such as data mining and log analysis.

A data warehouse is an online repository for decision-support applications that answer business queries quickly. The current computing trend of data parallel programming on large clusters attempts to change the base of data warehouses from traditional databases to the MapReduce framework. In [19], the authors said that MapReduce offers dramatic performance gains in analytic application areas that still require great performance speed-up, and this feature made Greenplum [20] and Asterdata [21] to integrate MapReduce into their SQL MPP data warehouse products. The biggest DBMS company, Oracle, also provides a way to implement MapReduce programs within their Oracle database system [22]. In addition to this, there exists open source data warehouse software built upon Hadoop, such as Hive [23] and Cloudbase [24].

Along with these paradigm shifts in computing and data analysis, the size of a data warehouse also tends to grow enormously. The common schema model of data warehouses is usually composed of a few fact tables (commonly only one) that reference dimension tables. Since a size of the fact table is very big, it is very important to reduce I/O upon processing a data warehouse workload (*e.g.*, On-Line Analytical Processing (OLAP)). In the trace of [9], redundant I/O in scanning the input data contributes to around 33% of the total I/O, and the most frequently accessed input file is accessed more than 200 times on average per day. In OLAP, this type of redundancy is more critical because a central fact table is accessed by most of queries.

In MapReduce frameworks, network bandwidth is an important design factor for a task scheduler. Despite the locality-aware scheduling policy of MapReduce, reading the same input files by different MapReduce jobs inevitably necessitates the I/O redundancy in the current architecture. Besides, redundant I/O leads to performance degradation especially when computing nodes are separated from storage nodes (*e.g.*, Amazon EC2 [3] and S3 [4]). To achieve high performance by reducing redundant I/O, MapRedue frameworks should provide a mechanism that caches and reuses a small part of input data downloaded from distributed storages. While achieving this task, the new mechanism should preserve the data consistency by means of a proper cache invalidation method.

In this article, we propose SplitCache to mitigate the I/O redundancy noticed in the execution of data-intensive OLAP-style applications in a cluster environment. The fundamental idea is that SplitCache caches input data when it first appears among applications, and reuses the cached data when other applications request the same data in near future. This leads to faster execution of each application, higher cluster throughput and less overhead of distributed storage services. SplitCache runs in computing nodes and is independent of the type of an underlying distributed file system. SplitCache is distinguished from the cache of traditional distributed file systems, such as Coda [15] and AFS [10], in that SplitCache cooperates with the cache-aware task scheduler. In a MapReduce framework, tasks can be dispatched toward the data. This feature changed the key factor that should be considered for the cache hit ratio; we should emphasize more on data itself rather than on a data access pattern of programs. Traditional cache policies, such as LRU and LFU, have tried to predict a future data access pattern of programs as much as possible. However, the change of the key factor affecting the good hit ratio renders many traditional cache policies ineffective. Thus, we proposed SplitCache as a new cache struc-

ture for a MapReduce framework. The preliminary version [16] of this article proposed a prototype of SplitCache. This article extends the previous work substantially by applying numerous optimization techniques, and shows evaluation results measured in various experimental environments.

The remainder of this article is organized as follows. Section 2 presents background and related work. We then explain the details of SplitCache in section 3, and evaluate it with the TPC-H benchmark in section 4. Finally, we conclude this article in section 5.

2. BACKGROUND AND RELATED WORK

One of the most successful applications of cloud computing is the analysis of very large data sets. Recently, several cloud computing service providers announced database systems on their own cloud platform. Amazon has SimpleDB [6] and Relational Database Service (RDS) [7]. SimpleDB is not a relational database but key-value stores. Key-value stores are very useful especially in the cloud environment [1]. Couch DB [30], Cassandra [31] and Hbase [32] are also included in this category as well. RDS is a relational database system based on MySQL. Microsoft also has a relational database system, SQL Data Services (SDS) [29], based on its cloud platform Azure [28]. In near future, many of database applications and infrastructures, including data warehouses, will be available on these database services based on the cloud infrastructure. It, therefore, is very obvious that reducing redundant disk I/O and network traffic is important for resource utilization and low cost of cloud computing services.

He *et al.* [9] introduced the wave model exposing redundancy among queries of data-intensive distributed applications. They identified two types of redundant operations: input data scan and common sub-query computation. In the study on a query trace, they found 33% of total I/O and 30% of sub-queries were redundant. Due to the redundancy, the current state of data-intensive distributed computing is far from reaching the ideal. Popa *et al.* [13] proposed DryadInc that reuses identical computation that is performed over common input data, and DryadInc computes just with the newly appended data. DryadInc is a solution to the problem of redundant computation in the Dryad [2]. SplitCache is a solution to the problem of redundant I/O in MapReduce.

There have been numerous research attempts aiming at the performance improvement of MapReduce. Zhang *et al.* [25] introduced a distributed memory cache between map and reduce phases. Map workers write intermediate data in the distributed memory cache and reduce workers read the corresponding data from the distributed memory cache. This work is similar to SplitCache in terms of using cache. But, Zhang's work focuses on time-intensive applications on small scale clusters, while SplitCache targets at OLAP-style applications on large-scale clusters such as data warehouse.

Zaharia *et al.* [26] suggested a new job scheduler for a heterogeneous environment. Because a MapReduce framework assumes a homogeneous environment, its performance has a problem in a heterogeneous environment. The authors proposed a new scheduling algorithm, Longest Approximate Time to End (LATE), to cope with the problem in a heterogeneous environment. Seo *et al.* [27] proposed the D-LATE (data-aware LATE) scheduling algorithm that extends the LATE algorithm. The D-LATE improves performance by prefetching the rear part of an assigned split and pre-shuffles input data be-

fore entering the map phase. Zaharia *et al.* [17] also proposed a simple delay scheduling technique to improve data locality. This simple technique just keeps a task tracker waiting for a small amount of time when it cannot launch a local task. It tries to place tasks on nodes that contain their input data as much as possible. These proposals are similar to SplitCache in that all of them extended the current simple job scheduler. But, they do not focus on the redundancy of input data rather than focus on the heterogeneous environment, and have no caching mechanism.

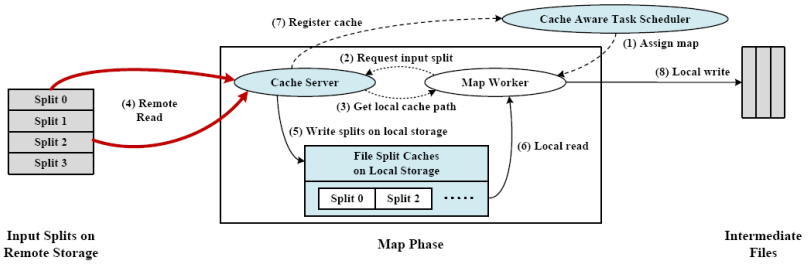
3. SYSTEM ARCHITECTURE

In this section, we present the design of SplitCache in detail. SplitCache is composed of two components: (1) a cache server, a local service daemon which caches common data for later use, and (2) a cache-aware task scheduler, an extension of the default locality-aware scheduler. Upon a job request, the cache-aware task scheduler determines which input split can be reused by a new map task and dispatches the task to a node that contains the split in its local cache. We describe the two different execution flows of SplitCache in section 3.1, and the cache server and the cache-aware task scheduler in sections 3.2 and 3.3, respectively. In section 3.4 we discuss about several issues related with SplitCache.

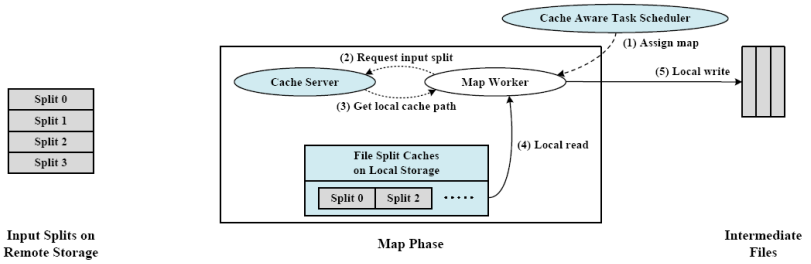
3.1 Overall Execution Flows

Fig. 1 (a) shows the execution steps when requested input data is not cached yet. First, a cache-aware task scheduler assigns a map task to a node ("(1) Assign map" in Fig. 1 (a)). The map worker requests an input data by invoking the **`open`** remote method call ("(2) Request input slit" in Fig. 1 (a)). At this time, the map worker also sends the start position and the length of the input split. Though the cache server has not cached the requested input split yet, the cache server creates an empty cache file and returns the location for the empty file ("(3) Get local cache path" in Fig. 1 (a)), and starts to read data from remote storage ("(4) Remote read" in Fig. 1 (a)). As the input data is written into a cache file ("(5) Write splits on local storage" in Fig. 1 (a)), the map worker can read data up to the latest written position without waiting for the completion of reading the input split to the last position ("(6) Local read" in Fig. 1 (a)). After completion of reading the input split from the remote storage, the cache server registers its ownership of the input split to the central cache-aware task scheduler for later use ("(7) Register cache" in Fig. 1 (a)). Finally, a map worker passes the result to the next reduce phase ("(8) Local write" in Fig. 1 (a)).

Fig. 1 (b) is when a cache server cached the input data already. In this case, SplitCache has the same execution steps until the third step ("(3) Get local cache path" in Fig. 1 (b)) as was in the previous case. The difference is that a cache server can provide data from its local storage when the same data is requested from any map task ("(4) Local read" in Fig. 1 (b)). Reusing caches greatly contributes for reducing network traffic (red arrows of "(4) Local read" in Fig. 1 (a)), and boosts up overall system performance. Following steps are same as the previous case. A cache-aware task scheduler plays a key role in increasing cache utilization in this case. The mobility of map tasks makes many



(a) Read from remote storage.



(b) Read from local cache.

Fig. 1. Cache server caches input data (a) when the data is first requested and (b) reuses the cached data when any subsequent map task requests the same data in near future. Recycling cached data saves the costly operation of fetching data from remote storage.

well-known cache replacement algorithms ineffective. The cache utilization of SplitCache is explained in the following section.

3.2 Cache Server

A cache server has three roles: (1) giving map workers the location information of caches, (2) reading and caching the remotely located input splits in its local storage, and (3) notifying the central cache-aware task scheduler of the ownership of cached input splits. Note that the cache server does not take part in reading caches, and it runs per node, not per task.

3.2.1 Space for cache files

We use one of modern high speed non-volatile storage systems (*e.g.*, magnetic or flash memory disks) as a storage system for cache files instead of small volatile memory. Because the size of the input data for a MapReduce application is very large, main memory cannot provide enough space for SplitCache. Besides, due to the finite space of disks, allocating cache space is a bit restricted. In many cases, cloud computing service providers usually assign relatively small local disk space per virtual compute instance. As we aforementioned earlier, traditional cache replacement policies are not effective in a MapReduce environment without the assistance of a task scheduler. The main cause of the ineffectiveness is due to the presence of a task scheduler. However, the choice of cache replacement policies for managing local caches does not affect the total cache utilization.

In each compute node, the cache server uses the LRU or LFU policy to limit the total size of cache data. The experimental result is shown in section 4.1.

3.2.2 Caching from neighbor cache servers

To reduce network traffic further, we added an optimization technique by which a cache server can read data from a neighbor cache server instead of remote storage. This technique is effective when a task cannot be dispatched to a compute node that has a cache of the desired data. This situation happens when a compute node having cache is busy. In this case, the cache-aware task scheduler assigns the task to another compute node that is the closest available compute node to the node that cached the desired data. This technique can help reducing long-range network traffic.

3.2.3 Consistency

In an append-only distributed file system, the modifiable region is only the last split, and accordingly the length of the modified file should be extended. When a map worker requests a modified input split (specifically the last input split), the cache server fails to look up a cache that covers the newly extended range and starts to create a new cache for it. If a file "A" is deleted and a new file "B" is created with different contents at the same location as the file "A", the above statement "the modifiable region is only the last split" is no longer valid. To manage this case, the cache server also keeps the creation time of each cache file. Before the cache server provides data with a map task, it compares the creation time stored at the cache with the creation time at the remote storage. If the two times are different, the cache server updates the stale cache file with new data.

3.3 Cache-Aware Task Scheduler

The cache-aware task scheduler assigns a map task to a node that has a matched cache. If the scheduler cannot dispatch a task to a compute node which has a matched cache, the task is assigned to the closest available node to the node which has the cache. Because MapReduce knows the network topology of all compute and storage nodes, the distance among compute nodes can be measured by the network topological distance.

Management of cache locations is one of the key roles of the cache-aware task scheduler. The cache-aware task scheduler supports two operations for managing cache locations; registering and searching cache blocks. Cache locations are managed by using two data structures: an interval tree and a hash table (Fig. 2). We make use of these data structures so that the interval tree is responsible for managing locations of cache blocks of each file, and the hash table is responsible for retrieving an interval tree of each file. The search operation is to find compute nodes which have caches that cover the desired "range". If the size of data block is fixed, the hash table is the best choice for searching caches. However, the size of data block is flexible. For this reason, we chose the interval tree for managing cache locations instead of the hash table. Every node of an interval tree has range values ($cache_{range}$) where a cache block covers within a file, and a list of compute nodes ($list_{node}$) that have the cache block.

When a cache server of SplitCache registers a new cache block to the cache-aware

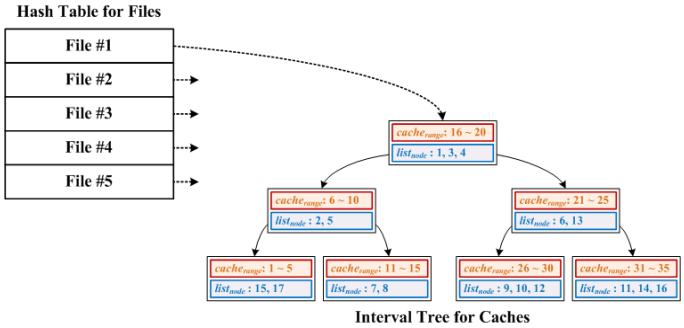


Fig. 2. Data structure for management of cache locations.

task scheduler, the scheduler checks whether an entry for the file path for the new cache block exists in the hash table first. If the scheduler fails to find the hash entry, the scheduler inserts a new entry for the file path in the hash table and creates an interval tree for the file. Otherwise, the scheduler retrieves an interval tree for the file from the hash table. After an interval tree for the new cache block is ready, the scheduler searches the interval tree to find a node which has same range value. If the scheduler fails to find the target node, it creates a new tree node which has a range value same as the new cache block. If the scheduler succeeds the searching, it adds the host address of the cache block to the $list_{node}$ of the corresponding node in the interval tree. When a cache-aware task scheduler searches the list of nodes which have a cache block that covers a desired range of a file, the scheduler retrieves an interval tree for the file from the hash table, and gets the list of nodes from the interval tree.

We used a simple hash function for implementing the hash table for files as below.

$$h(s) = s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1],$$

where $s[i]$ is the ith character of the string, n is the length of the string. We implemented the interval tree by using the augmented Red-Black tree algorithm [18]. We had considered three balanced binary search tree algorithms, which are Red-Black tree, AVL tree, and Splay tree, as a base data structure for the interval tree. Most of MapReduce applications have a sequential read access pattern. Pfaff [14] showed that the Splay tree has better performance than the AVL tree and the Red-Black tree in the case of a sorted data access pattern. However, the Splay tree can be changed even when it is accessed in a 'read-only' manner (*i.e.* by search operations). This complicates the use of the Splay tree in our multi-threaded cache-aware task scheduler. Specifically, extra management is needed if multiple threads are allowed to perform search operations concurrently. The AVL tree and the Red-Black tree have the same time complexity both in the average and the worst cases. However, if the assigned capacity for storing cache blocks becomes lower, the cache server should replace old cache blocks with new ones more frequently. This leads to many updates in the interval tree, and the more relaxed Red-Black balancing rule results in less rebalancing overhead in this case. Thus, we chose the Red-Black tree for implementing the interval tree. The interval tree, which is based on the augmented Red-Black tree algorithm, has running time of $O(\log n)$ for insertion and search,

where n is the number of cache blocks. The insertion and search operations for the hash table run in $O(\log n)$ and $O(1)$ respectively. Thus, the two operations for managing cache locations have running time of $O(\log n)$.

3.4 Discussions

3.4.1 Cache utilization

For many decades, numerous cache replacement algorithms have been designed to increase the hit ratio in a computing environment where a task cannot be moved to other nodes. In this case, designing an efficient cache replacement policy is the most important ingredient in terms of cache utilization. However, in SplitCache where map tasks are dispatched toward caches by a cache-aware task scheduler, many well-known cache replacement policies, such as LRU and LFU, are ineffective due to the mobility of a task. In this case, designing an efficient task scheduling policy is the most important factor for higher cache utilization.

SplitCache can guarantee full utilization of reusable cache by rearranging the execution order of tasks. Thus, the cache hit ratio of current job ($hit_{current}$) can be determined by the size of cached data which is required by current job ($size_{redundancy}$). For example, cache has data which covers file 'A' from 10 to 50. Job 'B' requires file 'A' of which the size is 50. In this case, $size_{redundancy}$ is (50 – 10) = 40. The cache hit ratio of current job is

$$hit_{current} = \frac{size_{redundancy}}{size_{input}}.$$

If all data files are read at least once and the size of cache space ($size_{cache}$) is unlimited, $hit_{current}$ will eventually converge to 100%. But, if $size_{cache}$ is limited, $hit_{current}$ is affected by input data of former jobs. In this situation, $size_{redundancy}$ cannot exceed $size_{cache}$. In addition to this, $size_{redundancy}$ might be much less than $size_{cache}$. Thus, the larger $size_{redundancy}$ is, the higher $hit_{current}$ achieves.

3.4.2 Load balancing between compute nodes

MapReduce splits a large job for large data into many *map* and *reduce* tasks by dividing the data, and executes the tasks on commodity clusters. The default job scheduling model of Hadoop exploits the Master-Worker and PULL interaction model. When a worker process (TaskTracker) is available, it requests a new task from the master (JobTracker). The master then assigns a task to the worker and the worker starts the task. Under this model, tasks are naturally distributed in a balanced way. SplitCache extends the Hadoop's default scheduler and also follows the Master-Worker and PULL interaction model. And the scheduler does not reorder queued jobs, and does not lead to the priority inversion of jobs and tasks. From this perspective, SplitCache does not have the load balancing issue.

3.4.3 Increasing the level of replication vs. SplitCache

In the viewpoint of the distributed file systems, cache files in computing nodes

could be regarded as replicas of original files. Increasing the level of replication of those files might be easier in the distributed file systems. But the cache files of SplitCache are not managed by the distributed file systems. As mentioned before, we assume an environment that data storage nodes are separated from computing nodes. Furthermore, increasing the level of replication cannot reflect temporal locality dynamically. Thus, SplitCache is designed to be independent from the underlying distributed file systems.

3.4.4 Overhead of managing cache blocks at the cache-aware task scheduler

The cache-aware task scheduler has as many interval trees as the number of files. Therefore, if the total size of cache blocks is not limited, the space and processing overhead for managing cache locations will increase as the size of data grows. However, SplitCache can set a quota on the total size of cache blocks. We evaluated SplitCache with varied quotas on the total size of cache blocks, and the results showed that the cache size of 4GB per each compute node shows comparable performance with unlimited cache size. Therefore, if the total size of cache blocks is limited at an appropriate level, the cache-aware task scheduler will not be a bottleneck as the data increases. The detailed evaluation results will be presented in section 4.2.

4. PERFORMANCE EVALUATION

We implemented the cache server and the cache-aware scheduler for Hadoop that is a very popular Map-Reduce implementation. Hadoop also provides an implementation similar to the Google File System (GFS) [8]. For the experiment, we used HDFS [5] to store input and output data.

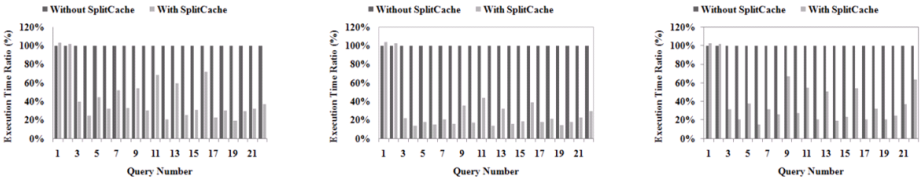
4.1 Experimental Environments

The experiment was performed on both private cluster and cloud computing environments. For the cluster environment, we used 10-node cluster in which each node is equipped with Intel Core2 Quad CPU 2.83GHz and 8 GB RAM running Ubuntu Linux with 2.6.30.5 kernel. Each node is configured to have three map tasks and one reduce task. All nodes are connected through a switched 1 Gbps Ethernet LAN. For the cloud environment, we used 16 virtual nodes of the icube cloud testbed [12], and each virtual node is equipped with two 2.00GHz virtual cores and 4 GB RAM running CentOS Linux with 2.6.18 kernel. All virtual nodes are connected through a switched 1 Gbps Ethernet LAN. We configured HDFS to have a single replica per block and the block size was 128 MB, and we used the default settings of HDFS for other configurations. The default split replacement policy is LRU, if not specified.

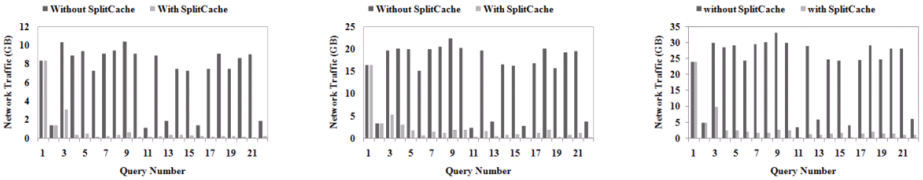
We assessed the performance of our system by running all queries of the TPC-H benchmark, a set of programs and queries designed to measure the performance of data warehouses. The schema of the benchmark is composed of eight tables (**`LINEITEM`**, **`ORDERS`**, **`CUSTOMER`**, **`SUPPLIER`**, **`PART`**, **`NATION`**, **`REGION`** and **`PARTSUPP`**). Each table is generated by the **`dbgen`** program of TPC-H and stored in HDFS as a form of a text file. Among the eight tables, the sizes of two tables (**`LINEITEM`** and **`ORDERS`**) are

much larger than those of other tables. All TPC-H queries evaluated in our experiments involve at least one of the two tables.

We conducted experiments by varying the scale factor. The scale factor represents the size of data in the TPC-H benchmark. Scale factor "1" means 1GB of the database size. Performance metrics include total running time, network traffic, and cache utilization. The total running time is the total running time of a MapReduce job. It includes initialization, map, and reduce phase. The network traffic is the total amount of transferred data between compute nodes and remote storage. The cache utilization is calculated as the ratio of the size of data read from caches to the total size of requested data. If all map workers read data only from caches, the cache utilization is 100%. We added logs to measure these metrics and the logging overhead was negligible.



(a) Normalized running time. (10 SF) (b) Normalized running time. (20 SF) (c) Normalized running time. (30 SF)



(d) Amount of network traffic. (10 SF) (e) Amount of network traffic. (20 SF) (f) Amount of network traffic. (30 SF)

(g) Normalized running time. (10 SF, random order) (h) Amount of network traffic. (20 SF, random order)

Fig. 3. Normalized running time and amount of network traffic in the cluster computing environment; (a)-(f) show the results when scale factors are 10, 20 and 30 respectively, and the arrival sequence of queries is sequential; (g) and (h) show the results when scale factor is 10, and the arrival sequence of queries is random.

4.2 Experimental Results

From Figs. 3 (a)-(c), we can see that SplitCache helps MapReduce-based TPC-H queries be processed faster. All queries are executed in the increasing order of the query number. In the case of query 1, the benchmark program reads only **`LINEITEM`** of which the size is 70% of the total data. Since the status of SplitCache is cold, the SplitCache case processes the query 1 slightly slower than the case without SplitCache due to caching overhead. The benchmark reads **`PARTSUPP`**, **`SUPPLIER`**, **`NATION`**, **`PART`** and **`RE-`**

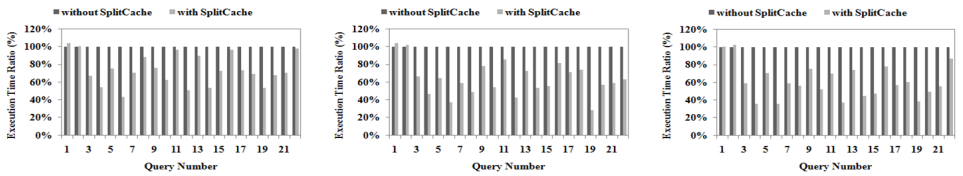
GION of which the size is 13% of total data when query 2 is executed. Since data for query 2 is not overlapped with data for query 1 at all, the SplitCache case evaluates query 2 slightly slower than the case without SplitCache due to caching overhead. **CUSTOMER**, **ORDERS** and **LINEITEM** are processed when query 3 is evaluated. The data of **CUSTOMER** and **ORDERS** (17% of total data) are not cached by SplitCache and this incurs trivial overhead. However, since data of **LINEITEM** (70% of total data) is cached, the performance benefit is great on processing query 3. After queries 1, 2 and 3 are processed, all data are cached. This fact leads to great performance benefits on processing the rest of the queries. Moreover, similar patterns of performance benefits between both cases of scale factor 10 and 20 arise.

Figs. 3 (d)-(f) show the amount of network traffic that is caused by IO when all queries are evaluated in the increasing order of query number. From the figures, we can see that SplitCache helps the network to be used efficiently in fields of analytic works or batch processing. Due to the caching effect, SplitCache reduces network traffic greatly on processing all queries except queries 1 and 2. During execution of all queries, SplitCache avoids about 83% of the data traffic as shown in Table 1.

For more realistic execution environment, Figs. 3 (g) and (h) show the normalized running time and the amount of network traffic when the arrival sequence of queries is random. We conducted 10 experiments with different sequence each, and the results are averaged over 10 experiments. As the above graphs show, SplitCache also accomplished greatly reduced running time and network traffic. In this situation, the cache utilization was 86.52% (Table 1).

Table 1. Average ratio of cache utilization.

Scale Factor	10 (sequential)	20 (sequential)	30 (sequential)	10 (random)
Cache Utilization	84.70%	83.16%	82.57%	86.52%



(a) 10 SF. (b) 20 SF. (c) 30 SF.

Fig. 4. Normalized running time in the cloud computing environment (SF: Scale Factor) reduces high authorotion, and evaluated with various environments including cloud coputing environment.

Fig. 4 shows results of SplitCache in the cloud testbed. From the figure, we can also see that SplitCache helps MapReduce-based TPC-H queries be processed faster even in the cloud environment. The pattern of performance benefits in the cloud testbed is similar to that in the cluster environment. It is noted that results of IO are skipped since results of the cloud testbed are the same as those of the local cluster.

Fig. 5 shows results of SplitCache when varying the cache size and the split replacement policy. In cases of query 1 and 2, running time is similar since the status of

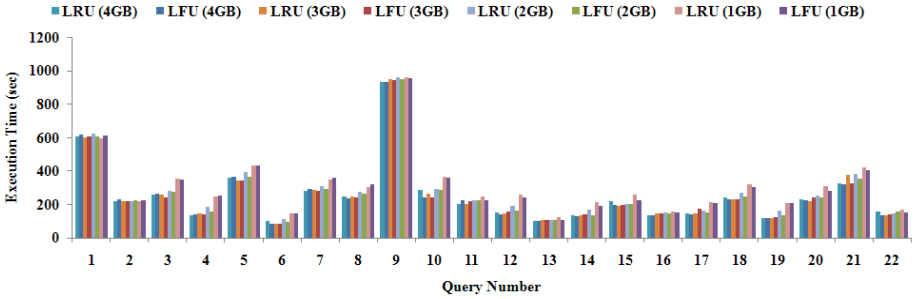


Fig. 5. Running time of TPC-H benchmark with varying the cache size and split replacement policy (20 SF, LRU: Least Recently Used, LFU: Least Frequently Used).

Table 2. Average performance improvement ratio with varying the cache size and split replacement policy (20 SF).

Split Replacement Policy	Performance Improvement Ratio			
	1GB	2GB	3GB	4GB
LRU	51.25%	56.61%	59.89%	59.94%
LFU	52.84%	58.14%	59.71%	60.12%

Table 3. Average cache utilization with varying the cache size and split replacement policy (20 SF).

Split Replacement Policy	Cache Utilization			
	1GB	2GB	3GB	4GB
LRU	14.96%	46.10%	67.74%	78.47%
LFU	17.61%	45.92%	67.73%	76.78%

SplitCache is cold. From the figure, we can see that the split replacement policy does not affect the running time of queries since the cache-aware task scheduler of SplitCache moves computation toward cached split, maximizing data locality. Table 2 summarizes Fig. 5 into the average performance improvement ratio. It shows that the performance benefit of SplitCache does not depend on the choice of split replacement policies.

Table 3 verifies that the cache utilization of both LRU and LFU policies is almost the same. As shown in the Table 3, the utilization increases as the cache size increases, and this leads to better performance results in cases of bigger cache size in Fig. 5. As shown in Table 1, the cache utilization converges into 83-84% when the size of cache space becomes larger. The cache with more than 4 GB does not affect the performance of TPC-H query evaluation in large part.

We evaluate the average cache utilization with varying split size from 4MB to 128MB by doubling the split size. As shown in Table 4, the cache utilization decreases as the split size increases. This is because the effect of one cache miss to the cache utilization becomes larger as the split size increases. If all data are read once by map tasks, following map tasks can read all of required input data from the local cache. However, if a compute node that has a cache block is busy, another compute node launches the task, and subsequently a cache miss happens. In this case, one cache miss with the split size of

128MB generates data transfer of 128MB from a remote storage node. But, one cache miss with the split size of 4MB generates data transfer of only 4MB. This is why cache utilization decreases as the split size grows.

Table 4. Average cache utilization with varying split size (10 SF).

Split Size	4MB	8MB	16MB	32MB	64MB	128MB
Cache Utilization	89.40%	88.91%	88.91%	86.07%	84.46%	84.97%

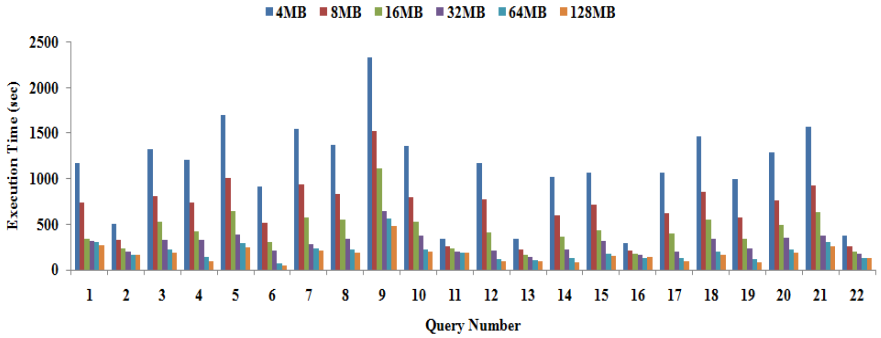


Fig. 6. Running time of TPC-H benchmark with varying the split size (10 SF).

Small split size, however, causes severe performance degradation. Fig. 6 shows the running time of TPC-H with varying the split size from 4MB to 128MB by doubling the split size. The total running time with the split size of 4MB is over 6.5x slower than the case of 128MB. This performance degradation comes from the initializing and finalizing overhead of each task in Hadoop. Because Hadoop launches map tasks as many as the number of splits, the overhead grows as the size of split becomes smaller. Thus, when the split size is small, the cache utilization is high meanwhile the overall performance becomes low.

5. CONCLUSION

We proposed SplitCache as a solution to the problem of significant redundant I/O which is common in the execution of data-intensive applications. Our study targets an environment in which data storage nodes are separate from computing nodes. This configuration is popular in many public cloud computing sites. Unfortunately, current locality-aware task scheduler in a MapReduce framework is not effective in this cluster environment. SplitCache caches common input splits in computing nodes for later use, and assigns map tasks to the nodes that have caches for the requested input splits. The cooperation of the cache server and the cache-aware task scheduler reduced redundant network traffic and disk I/O significantly. We also showed by the TPC-H benchmark that SplitCache achieved 64.3% faster execution and 83.48% reduction of network traffic on average.

REFERENCES

1. "Key value stores: usefulness in cloud environment," http://www.infosysblogs.com/cloudcomputing/2010/02/key_value_stores_usefulness_in.html.
2. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007, pp. 59-72.
3. Amazon, EC2 service, http://aws.amazon.com/ec2.
4. Amazon, S3 service, https://s3.amazonaws.com/.
5. Apache, Hadoop, http://hadoop.apache.org/.
6. Amazon, SimpleDB, http://aws.amazon.com/simpledb/.
7. Amazon, Relational Database Service (RDS), http://aws.amazon.com/rds/.
8. S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," *SIGOPS Operation Systems Review*, Vol. 37, 2003, pp. 29-43.
9. B. He, M. Yang, Z. Guo, R. Chen, W. Lin, B. Su, H. Wang, and L. Zhou, "Wave computing in the cloud," in *Proceedings of Workshop on Hot Topics in Operating Systems*, 2009, pp. 5-9.
10. J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, Vol. 6, 1988, pp. 51-81.
11. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004, pp. 10-22.
12. NexR, iCube Cloud testbed, http://www.icubecloud.com.
13. L. Popa, M. Budiu, Y. Yu, and M. Isard, "DryadInc: Reusing work in large-scale computations," in *Proceedings of Workshop on Hot Topics in Cloud Computing*, 2009, pp. 21-25.
14. B. Pfaff, "Performance analysis of BSTs in system software," *ACM SIGMETRICS Performance Evaluation Review*, Vol. 32, 2004, pp. 410-411.
15. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, Vol. 39, 1990, pp. 447-459.
16. S. Kim, H. Han, H. Jung, H. Eom, and H. Y. Yeom, "Harnessing input redundancy in a mapreduce framework," in *Proceedings of the 25th Annual ACM Symposium on Applied Computing*, 2010, pp. 362-366.
17. M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, 2010, pp. 265-278.
18. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., The MIT Press, Cambridge, pp. 348-353.
19. "Why mapreduce matters to SQL data warehousing," http://www.dbms2.com/2008/08/26/why-mapreduce-matters-to-sql-data-warehousing.
20. Greenplum, http://www.greenplum.com/.
21. Asterdata, http://www.asterdata.com/.
22. "In-database mapreduce (map-reduce)," http://blogs.oracle.com/datawarehousing/2009

/10/in-database_map-reduce.html.
23. Hive, http://hadoop.apache.org/hive/.
24. CloudBase, http://sourceforge.net/projects/cloudbase/.
25. S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Accelerating map-reduce with distributed memory cache," in *Proceedings of the 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 472-478.
26. M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 29-42.
27. S. Seo, I. Jang, K. Woo, I. Kim, J. S. Kim, and S. Maeng, "HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment," in *Proceedings of the 11th IEEE International Conference on Cluster Computing*, 2009, pp. 1-8.
28. Microsoft, Azure, http://www.microsoft.com/windowsazure/.
29. Microsoft, SQL Data Services (SDS), http://msdn.microsoft.com/en-us/sqlserver/dataservices/default.aspx.
30. Couch DB, http://couchdb.apache.org/.
31. Cassandra, http://cassandra.apache.org/.
32. Hbase, http://hadoop.apache.org/hbase/.

Shin-Gyu Kim received his B.S. and M.S. degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea in 2006 and 2008, respectively. Currently, he is a Ph.D. candidate at Seoul National University. His current research area includes distributed computing and massive data processing.

Hyuck Han received his B.S. and M.S. degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003 and 2006, respectively. Currently, he is a Ph.D. candidate at Seoul National University. His research interests are distributed computing systems, parallel computing, and database systems.

Hyungsoo Jung is a postdoctoral researcher in the School of Information Technologies, the University of Sydney, Australia. He received the B.S. degree in Mechanical Engineering from Korea University, Seoul, Korea, in 2002; and the M.S. and the Ph.D. degrees in Computer Science from Seoul National University, Seoul, Korea in 2004 and 2009, respectively. His research interests are in the areas of distributed systems, database systems, and transaction processing.

Hyeonsang Eom is an Assistant Professor with the School of Computer Science and Engineering, Seoul National University. He received his B.S. degree in Computer Science from Seoul National University in 1992 and his M.S. and Ph.D. degrees in Computer Science from University of Maryland at College Park in 1996 and 2003 respectively. From 2003 to 2005, he worked with Samsung Electronics as a Senior Researcher.

He joined the Department of Computer Science, Seoul National University in 2005, where he currently teaches and researches on distributed systems, mobile application, and software performance engineering.

Heon Y. Yeom is a Professor with the Department of Computer Science and Engineering, Seoul National University He received his B.S. degree in Computer Science from Seoul National University in 1984 and received the M.S. and Ph.D. degree in Computer Science from Texas A&M University in 1986 and 1992, respectively. From 1992 to 1993, he was with Samsung Data Systems as a Research Scientist. He joined the Department of Computer Science, Seoul National University in 1993, where he currently teaches and researches on distributed systems, multimedia systems and transaction processing, *etc.*