EclipseMR: Distributed and Parallel Task Processing with Consistent Hashing

Vicente A.B. Sanchez[§], Wonbae Kim[§], Youngmoon Eom[§†],
Kibeom Jin, Moohyeon Nam, Deukyeon Hwang, Jik-Soo Kim[‡], Beomseok Nam
UNIST (Ulsan National Institute of Science and Technology), [†]Korea Military Academy, [‡]Myongji University

***Abstract*—We present *EclipseMR*, a novel MapReduce framework prototype that efficiently utilizes a large distributed memory in cluster environments. EclipseMR consists of double-layered consistent hash rings - a decentralized DHT-based file system and an in-memory key-value store that employs consistent hashing. The in-memory key-value store in EclipseMR is designed not only to cache local data but also remote data as well so that globally popular data can be distributed across cluster servers and found by consistent hashing.**

In order to leverage large distributed memories and increase the cache hit ratio, we propose a *locality-aware fair* (LAF) job scheduler that works as the load balancer for the distributed in-memory caches. Based on hash keys, the LAF job scheduler predicts which servers have reusable data, and assigns tasks to the servers so that they can be reused. The LAF job scheduler makes its best efforts to strike a balance between data locality and load balance, which often conflict with each other. We evaluate EclipseMR by quantifying the performance effect of each component using several representative MapReduce applications and show EclipseMR is faster than Hadoop and Spark by a large margin for various applications.

***Index Terms*—MapReduce; Distributed Caching; Consistent Hashing; Distributed Hash Table;**

I. Introduction

In the past decade, a significant amount of effort has been made to improve the effectiveness of the MapReduce frameworks in terms of resolving the load imbalance problem [20], [26], implementing a global resource manager [31], and employing in-memory caching to reuse previously computed data [28], [35], [8], [19], just to name a few.

Although one of the major new features introduced in the latest Hadoop is the in-memory caching in HDFS, Hadoop is still far from satisfactory since the in-memory caching is used only for caching local input data. The input data caching alone does not significantly improve the job execution time especially for compute-intensive applications. In the database systems field, considerable research has been conducted to show semantic caching successfully reduces query response time and improves system throughput by exploiting sub-expression commonality across multiple queries. If multiple MapReduce jobs have some common sub-computations, caching not only the input data but also the intermediate results for the sub-computations can greatly speed up subsequent tasks.

Another well known important limitation of Hadoop is the *skew* problem, which occurs both in a record level and an input block level. For both skew types, Hadoop is unable to achieve good load balance [7], [20], [26]. That is, if input data blocks are unevenly distributed, some servers process a larger number of blocks and it aggravates the straggler problem. Even if input data blocks are evenly assigned to servers, some map tasks may take longer to complete than other map tasks if certain input data blocks require more computations. `page rank` is an application of this type that suffers from an uneven distribution of computations.

In this work, we propose *EclipseMR*, a novel MapReduce framework prototype integrated with the distributed in-memory cache that leverages large distributed memories. Key components of EclipseMR are i) a decentralized DHT file system, ii) distributed in-memory caching with consistent hashing, and iii) a *locality-aware fair* (LAF) job scheduler that makes scheduling decisions based on the hash key ranges of the distributed in-memory caches. In EclipseMR, applications can choose to tag and store intermediate results from map tasks or job outputs for future reuse so that applications can leverage those cached objects by either reusing them directly or by applying user-defined data transformations to them.

In distributed systems, there is a trade off between load balance and data reuse. In other words, forwarding a MapReduce task to an idle worker server may turn out to be detrimental to overall performance since other busy worker servers may have cached data objects that can considerably speed up the job processing throughput. Similarly, forwarding a MapReduce task to a worker server that has cached data objects may hurt load balance when thousands of tasks need to access the same cached data object. As both the cache hit ratio and load balancing are critical performance factors that can hurt each other, a scheduling algorithm which yields high cache hit ratio without hurting load balance is essential to enforce high system throughput. Although Hadoop's job scheduling algorithms and Spark's delay scheduling algorithm consider both input data locality and load balance to some extent, they do not strike a balance between load balance and cache hits very well since the delay scheduling makes scheduling decisions by considering only cache hits first and then considering load balance after waiting some amount of time in the queue, i.e., they give up data locality if load balancing can not be achieved.

The LAF scheduling algorithm we propose for EclipseMR makes predictive scheduling decisions based on the probability of reusing cached data. The LAF scheduler receives job requests from clients, and determines the frequency of cached data accesses. Based on the recent data access pattern, the LAF scheduler makes its best efforts to assign an equal number of tasks to each worker server to achieve load balance. In addition to load balancing, the LAF job scheduler takes advantage of consistent hashing, i.e., the LAF job scheduler maps objects that have the same hash key to the same worker server, thus improving the cache hit ratio. The consistent hashing also helps map tasks to proactively shuffle intermediate results and applications to find cached objects without searching the central directory.

The main contributions of this work are as follows.

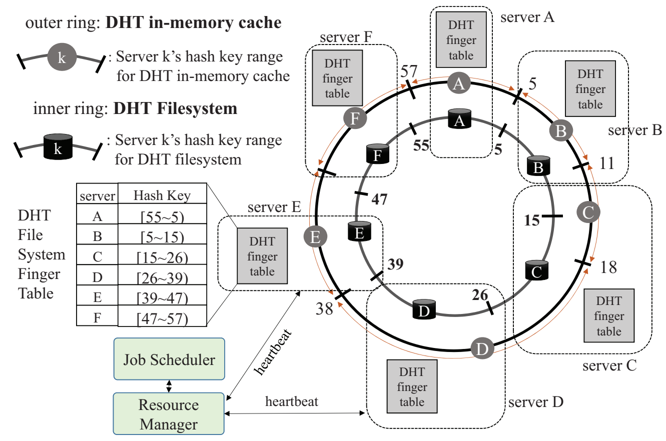
- **Integration of consistent hashing into MapReduce framework**
  EclipseMR replaces distributed file systems (e.g., GFS or HDFS) with a decentralized DHT file system. Unlike HDFS or GFS, the DHT file system spreads the file metadata service evenly across the cluster and eliminates the bottleneck of the central directory manager (e.g., NameNode in HDFS). As each worker server in EclipseMR has its own DHT routing table, it can directly access data objects in remote peer servers without looking up a central directory.
- **Distributed in-memory key-value caching**
  EclipseMR employs a distributed in-memory key-value store that caches data objects independent of their input data locations but based on their hash keys. EclipseMR allows applications to choose whether they cache input file blocks or intermediate results in the distributed in-memory caches. As in the DHT file system, the cached data objects can be found by their hash keys.
- **Locality-aware fair job scheduling policy**
  We propose a *locality-aware fair (LAF)* job scheduling policy that dynamically adjusts the hash key ranges of the distributed in-memory key-value store to balance the system load. By adjusting the hash key ranges of consistent hashing, an equal number of incoming tasks can be assigned to each server while preserving input data locality.

Through extensive experiments, we show that each design and implementation of EclipseMR components contributes to improving the performance of distributed job processing.

The rest of the paper is organized as follows: In section II, we present our design and implementation of the EclipseMR framework. In section III, we evaluate the performance of EclipseMR against Hadoop and Spark. In section IV, we discuss other research efforts related to the improvement of the MapReduce framework. In section V, we conclude the paper.

II. EclipseMR

EclipseMR consists of a job scheduler, a resource manager, and double-layered consistent hash ring structures - a DHT file system and a distributed in-memory key-value store as shown in Figure 1. The job scheduler is responsible for assigning incoming queries, including MapReduce tasks, to back-end worker servers, and the resource manager is responsible for server join, leave, failure recovery, and file upload.


outer ring: DHT in-memory cache
k : Server k's hash key range for DHT in-memory cache
inner ring: DHT Filesystem
k : Server k's hash key range for DHT filesystem
DHT File System Finger Table
server A
server B
server C
server D
server E
server F
DHT finger table
A
B
C
D
E
F
57
5
11
18
38
55
5
15
26
39
47
Job Scheduler
Resource Manager
heartbeat
heartbeat

server	Hash Key
A	[55~5)
B	[5~15)
C	[15~26)
D	[26~39)
E	[39~47)
F	[47~57)

Fig. 1. *Double-layered Chord ring in EclipseMR. The outer layer is the distributed in-memory cache layer and the inner layer is the distributed file system layer.*

The distributed in-memory cache and the DHT file system are completely decentralized components leveraging consistent hash rings. EclipseMR requires the job scheduler and resource manager to act as coordinators, but any worker server can take on the responsibility regardless. Hence, the job scheduler and the resource manager are selected by a distributed election algorithm.

A. DHT File System

As in HDFS, DHT file system in EclipseMR partitions an input data file into fixed-sized blocks, but the partitioned data blocks are distributed across the servers according to their hash keys. Since the location of the partitioned blocks can be determined by hash functions, the DHT file system does not need a centralized directory service that manages the location of each block. Instead, we store metadata about a file including file name, owner, file size, and partitioning information in a decentralized manner. For example, if a user uploads a file, we generate a hash key using the file name, and store the metadata about the file in the server (*file metadata owner*) whose hash key range includes the file's hash key. At the same time, the partitioned file blocks are distributed across servers based on their hash keys.

Later, when an application wants to access a file, it obtains the hash key of the file using its file name, and accesses the file metadata owner in order to check the access permission, file size, hash keys of the partitioned blocks, etc. Once the applications reads the file metadata, it multicasts the block read requests to remote servers. Suppose the hash key ranges of the DHT file system are as shown in Figure 1. If a file's hash key is in the range of $[5, 15)$, its file metadata will be stored in server B. To resolve the input data block skew problem, we distribute the partitioned file blocks across the ring using their hash keys. We discuss the data skew problem and consistent hashing in more details in Section II-E.

We make DHT file system fault tolerant by replicating the file metadata as well as file blocks in predecessors and successors. When a worker server fails, either a predecessor or a successor will take over the faulty server and utilize the replicated blocks and metadata. Hence, unless a server fails along with its predecessor and successor at the same time, the DHT file system can tolerate system failures. If a resource manager or a scheduler fails, the rest of the worker servers execute an election algorithm to choose a new resource manager and a scheduler.

In the DHT file system, each server manages its own routing table, called finger table, containing m peer servers' information. m can be determined by system administrators but it should be chosen so that $2^m - 1 > S$, where S is the number of servers in the hash ring. Unless a cluster has more than thousands of servers, as in large scale peer-to-peer file sharing systems, we set m to the total number of servers to enable the *one hop DHT* routing [13]. When m is smaller, file IO requests can be redirected and the IO performance can be degraded. Because most distributed query processing systems are more stationary than dynamic peer-to-peer file sharing systems and the number of servers is usually less than a couple thousand, storing complete routing information for entire servers in a DHT routing table does not hurt the scalability of the system but improves data access performance [13].

Since the local DHT routing table is very small, the table lookup places minimal overhead on each server. When a server receives a file block access request from a remote server, it checks if the hash key of the file block is within its own hash key range. If so, it looks up its local disks and serves the data access request. Otherwise, i.e., if zero hop routing is not enabled, it routes the request to another server that owns the hash key as in the classic DHT routing algorithm [29].

The DHT routing table is stationary so that it updates neighbor information including successor and predecessor only when a participating server joins, leaves, or fails. Each server exchanges heartbeat messages with direct neighbors to detect server failures, and the resource manager and job scheduler are notified when a server failure is detected. If a server fails, the resource manager reconstructs the lost file blocks in a *take-over* server using the replicated data blocks.

B. Distributed In-Memory Cache

In data-intensive computing, it is common for same applications to submit jobs that share the same input data. For example, database queries often access the same tables. There exist several prior works [14], [11] that report more than 30% of MapReduce jobs are repeatedly submitted in a production environment. Over the past decades, there have been a large number of works that exploit sub-expression commonality across multiple queries and incremental computation [8], [21], [22], [23], [24], [30]. The incremental computation significantly increases the chances of data reuse, reduces the job response time, and improves the system throughput.

On top of the DHT file system, EclipseMR deploys a distributed in-memory cache layer to exploit the incremental computation. The distributed in-memory cache consists of two partitions - *iCache* and *oCache*.

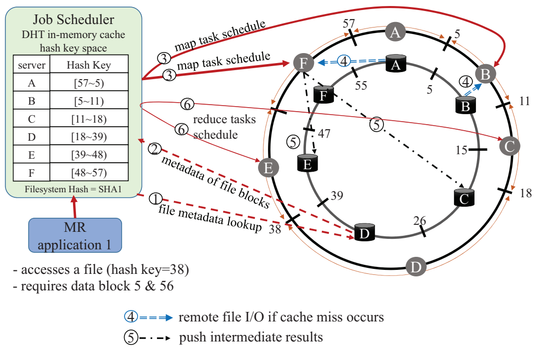


Fig. 2. *MapReduce Job Scheduling in EclipseMR*

iCache is where input data blocks are implicitly cached. The latest HDFS also implemented in-memory caching, but HDFS in-memory caching stores only local input data blocks. Since data skew problem occurs not only in a record level but also in an input block level, HDFS in-memory caching does not mitigate the skew problem of input blocks. To resolve this problem, we let *iCache* allow input data blocks to be cached in peer servers according to their hash keys. In an extreme case, if all the submitted map tasks need input data blocks stored in a particular single server's local storage, the popular input data blocks will be cached across the entire cluster by a heuristic job scheduling algorithm that we propose in section II-E.

oCache is where intermediate results of map tasks and outputs of iterative jobs are explicitly cached by user applications. EclipseMR tags the cached data with their metadata (application ID, user-assigned ID for cached data). *oCache* helps avoid redundant computations by sharing the intermediate results among multiple jobs. *oCache* is similar to RDDS in Spark, but intermediate results or outputs for iterative jobs in EclipseMR are cached according to their hash keys, so it evenly distributes frequently accessed cached data objects across the entire distributed memories. The cached intermediate results and outputs of iterative jobs are also persistently stored in the DHT file system according to their hash keys so that long running jobs can survive faults and restart from the point of failure.

The hash key ranges of in-memory caches are determined by a job scheduler based on workload pattern so that popular hash key ranges can use more distributed memories. However, the hash key ranges of DHT file system are statically determined by consistent hashing and do not change unless servers join or fail. Therefore, the hash key ranges of the distributed in-memory cache layer can be misaligned with the hash key ranges of the DHT file system.

C. MapReduce Processing

Figure 2 illustrates how EclipseMR processes a MapReduce job on a double layered ring structure. Suppose a job is submitted to the job scheduler. The job scheduler runs a hash function with the input file name to find out which server is the file metadata owner. Suppose server D is the file metadata owner in the example (①). Then, server D checks the file access permission and replies how the file is partitioned (step ②) and what are their hash keys.

If the input file is partitioned into two blocks and their hash keys are 6 and 56, the two blocks are stored in server A and B's local disks, respectively. Given the hash keys of the two blocks, the job scheduler searches the hash key ranges of the distributed in-memory caches, and assigns a map task to each of server B and server F (step ③). Note that a map task is scheduled in server F instead of A even though an input file block 56 is stored in server A's local disk. This is because each worker server's hash key range in the job scheduler's hash key table are misaligned with the hash key ranges of the DHT file system.

If server F has the input file block in its *iCache*, it reuses it. Otherwise, server F reads the input file block from the DHT file system, i.e., looks up its DHT routing table to find out the file block 56 exists in remote server A's local disk. After reading the block from A's local disks, server F stores the block in its *iCache* (step ④), and runs map tasks.

While map tasks are running, EclipseMR forwards the intermediate results generated by the map tasks to other servers according to the hash keys of the intermediate results so that they are persistently stored in the DHT file system (step ⑤). EclipseMR stores the intermediate results in persistent file systems as in Hadoop so that it can restart failed tasks and reuse the intermediate results of the previous failed tasks. Although we store the intermediate results on disk, they can be cached in *oCache* for future reuse. Note that we store the intermediate results on the reducer side, not on the mapper side. The stored intermediate results are invalidated by time-to-live (TTL) which can be set by applications, and they are not replicated by default.

While map tasks are generating intermediate results, they notify the scheduler with their hash keys. With the given hash keys, the scheduler schedules reduce tasks where the intermediate results are stored. Reduce tasks read these intermediate results from *oCache* or the DHT file system using the hash keys (step ⑥).

If a user application specifies it can reuse intermediate results and they are available in *oCache* or the DHT file system, the map tasks skip computation and reducer tasks can immediately reuse the cached data. If intermediate results are not available, the map tasks search *iCache* for input data blocks to reuse. If input data blocks are not available either, they read input data blocks from the DHT file system. There exist certain applications such as `k-means` that can not reuse intermediate results between map tasks and reduce tasks, but they need the results of reduce tasks from each iteration. For such applications, the EclipseMR allows applications to store the iteration outputs in *oCache* or the DHT file system instead of intermediate results.

D. Proactive Shuffling

Hadoop stores the intermediate results in the local disks of the server where the map tasks run. The shuffle phase in Hadoop sorts, splits, and sends the intermediate results to reducers. It is known that the shuffle phase of MapReduce is network intensive and the shuffle phase can constitute a bottleneck. Hadoop tries to pipeline map, shuffle, and reduce phases by starting reduce tasks as soon as intermediate result files are available, but Hadoop pipelining is far from satisfactory, and there have been several previous works that try to aggressively overlap the shuffle phase with the map phase and decouple from the reduce phase [2], [3], [6], [12], [32].

Unlike Hadoop or Spark, EclipseMR determines where to run reduce tasks based on the hash keys of the intermediate results. Therefore, the shuffle phase in EclipseMR does not have to wait until map tasks finish. Instead, EclipseMR lets each mapper pipeline the intermediate results to the DHT file system in a decentralized fashion while they are being generated. Based on the hash keys of the intermediate results, each map task stores the intermediate results in a memory buffer for each hash key range. When the size of this buffer reaches a certain threshold specified by the application, EclipseMR spills the buffered results to the DHT file system so that they can be accessed by reducers.

E. Locality-aware Fair Job Scheduling

Skew is a well-known problem in the MapReduce framework. Input block level skew problem occurs due to uneven distribution of input data blocks [7], [20]. Several prior works investigated to mitigate the input block skew problem by implementing speculative scheduling policies [7], [15], or by repartitioning unprocessed input data and utilizing idle servers in the cluster [20]. In this work, we propose a *locality-aware fair* (LAF) job scheduling algorithm for EclipseMR that proactively resolves the skew problem.

The LAF job scheduling algorithm assigns map tasks considering the data locality in distributed in-memory caches in order to leverage a large amount of distributed memory. It also evenly spreads given workloads across multiple worker servers in order to achieve good load balance.

Planning load-balanced schedules while maintaining a high cache hit ratio is a hard problem because cached data objects are frequently replaced and having the job scheduler keep track of a large number of distributed cached data objects is not very scalable. In Spark, the metadata of the cached data objects are managed by a central component. If the number of cached data objects grows, such centralized cache management can become a potential performance bottleneck. Instead, EclipseMR resorts to a heuristic algorithm.

LAF scheduling is a statistical prediction algorithm that uses *box kernel density estimation* and a moving average. In LAF job scheduling, a job scheduler estimates the hash key distribution of file block accesses and predicts the moving trend of the incoming data access pattern. The LAF scheduler computes the moving average of hash key accesses because it is not practical to keep all the historical data accesses in

Algorithm 1 Locality-Aware Fair Job Scheduling

```
1:  hkey = getHashKey(task.input_data);
2:  while true do
3:      server = selectAvailableServer();
4:      if $hkey$ is within $server.hashKeyRange$ then
5:          assignTaskToSlot(task, server);
6:          break;
7:      end if
8:  end while
9:  /* add a new hash key to hash key distribution */
10: distr = add2Distribution(distr, hkey);
11: if $distr.size == N$ then
12:     /* merge the new distribution to the old one */
13:     for $b \leftarrow 0$ to $NumOfBins$ do
14:         /*compute moving averaged hash key distribution.*/
15:         maDistr[b] = $\alpha$*distr[b] + maDistr[b]*(1-$\alpha$);
16:     end for
17:     cdf = constructCDF(maDistr);
18:     for $i \leftarrow 0$ to $NumServers - 1$ do
19:         /* assign equally-probable hash key range */
20:         servers[i].hashKeyRange = partitionCDF(cdf, i);
21:     end for
22:     initializeDistribution(distr);
23:     distr.size = 0;
24: end if
```

memory. As shown in Algorithm 1, a job scheduler constructs a histogram of hash keys accessed by the previous N tasks to determine recent data access patterns and predict which servers have possibly cached what input data blocks.

The job scheduler partitions the hash key space into a large number of fine-grained histogram bins, and it increases the counter of multiple adjacent k bins for each input data block access by $1/k$, where k is a bandwidth parameter in box kernel density estimation. As k becomes larger, the curve of probability distribution function (PDF) becomes smoother.

The data access patterns can be considered as time series data, thus we smooth out short-term fluctuations of the recent data access patterns and attenuate historic data access patterns by convolving the recent data access patterns with old data access patterns using a moving average equation. That is, the job scheduler constructs a hash key access PDF for a predefined number of recent tasks, attenuates the historic PDF by multiplying a weight factor $1-\alpha$, and computes the moving average (line 15 in Algorithm 1). Since each worker server caches only a certain number of recently accessed data objects using the LRU cache replacement policy, ideally the moving average should reflect the hash key distribution of data objects cached in distributed in-memory caches. If the weight factor is set to 1, the LAF job scheduler does not consider the past hash key distribution but just the hash key distribution of the current workload. If the weight factor is set to 0, the LAF job scheduler makes scheduling decisions based on the fixed static hash key ranges, which is perfectly aligned with the hash keys of the DHT file system.

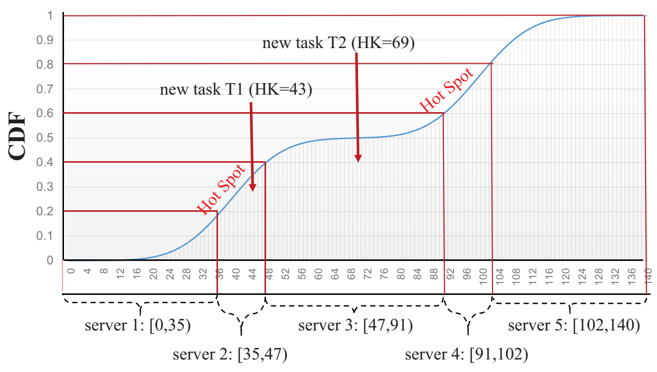


Fig. 3. *Each hash key range has an equal probability to serve incoming tasks. T1 and T2 will be scheduled in server 2 and 3.*

With the smoothed-out hash key histogram (PDF), the LAF scheduling policy partitions the DHT hash key space to equally-probable hash key ranges and assigns each hash key range to a worker server. For example, suppose there are five servers and the hash key space is $[0, 140)$. If a MapReduce job accesses input file blocks in a uniform distribution, the servers will be assigned [0,28),[28,56),[56,84), [84,112), and [112,140). If a task that needs to access a file block whose hash key is 40 is submitted, the LAF scheduling policy will assign the task to the second worker server whose hash key range ([28,56)) covers 40. If another task that accesses the same file block is submitted, it will again be scheduled to the same worker server so that the task can benefit from data locality and it can read the file block from the in-memory cache.

Later, suppose hash keys near 40 and 90 become popular and more frequently accessed than other hash keys. Then Algorithm 1 will gradually adjust the hash key distribution. Using the updated hash key distribution, the $partitionCDF()$ function in Algorithm 1 will re-partition the hash key space into 5 ranges - $[0, 35)$, $[35, 47)$, $[47, 91)$, $[91, 102)$, and $[102, 140)$ as shown in Figure 3.

Each of these hash key ranges have the same probability (20%) of task assignments according to the hash key distribution. Note that servers 2 and 4's hash key ranges are narrower than the other servers' hash key ranges because their ranges include popular hash keys. If hash keys near 40 and 90 become even more popular, The LAF scheduling policy will further narrow down the ranges of servers 2 and 4 so that fewer incoming map tasks are assigned to them.

In an extreme case, if the single hash key 40 becomes the one and only hot spot and all the incoming tasks access no other hash keys but 40, $partitionCDF()$ in Algorithm 1 will divide the hash key space into $[0, 40]$, $[40, 40)$, $[40, 40)$, $[40, 140)$ so that all the worker servers will eventually read the same hot data 40 from the DHT file system and replicate it in their distributed in-memory caches.

If the distribution of hash keys of accessed input data changes over time, some cached data objects can be misplaced in the servers whose hash key ranges do not cover the cached

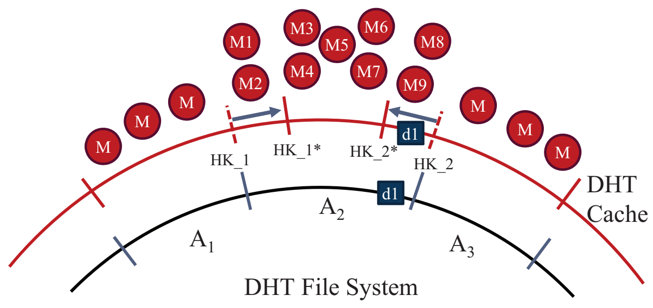
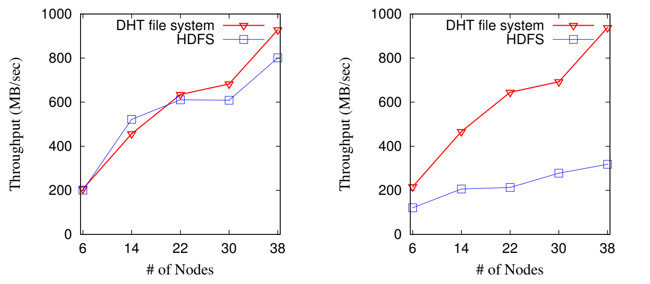


Fig. 4. *LAF achieves load balancing while considering data locality.*



(a) AVG IO Throughput (Bytes/Map Task Exec. Time)

(b) AVG IO Throughput (Bytes/Job Exec. Time)

Fig. 5. IO throughput with varying the number of data nodes

data objects. For example, as shown in Figure 4, if input file blocks located in worker server $A2$ become popular, the LAF scheduling algorithm will change the hash key range of $A2$ from $[HK\_1, HK\_2)$ to $[HK\_1*, HK\_2*)$ so that tasks $M1$,$M2$,$M8$,and $M9$ are scheduled in $A1$ or $A3$. Then a file block cached in $A2$ according to the old hash key ranges ($d1$ in the example) can not be reused by subsequent tasks since the LAF job scheduler will assign incoming tasks that need $d1$ to neighbor worker server $A3$. Since $A3$ does not have $d1$ in its in-memory cache, $A3$ has to read $d1$ from $A2$'s disks. An easy solution to the *misplaced cached data* problem is to migrate the cached data objects to a neighbor worker server whose new hash key range covers the misplaced cached data. EclipseMR provides an option to check if a left or a right neighbor worker server has cached data objects, and to migrate the cached data if either one has. A better way to resolve the misplaced cached data problem is to read replicated file blocks. As discussed in section II-A, each input file block is replicated in the predecessor and successor. Hence, unless the hash key ranges of the distributed in-memory cache are seriously misaligned with the DHT file system hash key ranges, EclipseMR can avoid remote disk IOs. In our experiments, we observed few misplaced cached data objects are needed by neighbor worker servers, hence we disable this data migration option for the experiments presented in section III.

The time complexity of the LAF scheduling algorithm is $O(n)$ where n is the number of worker servers as shown in Algorithm 1. Thus the LAF scheduling algorithm is very lightweight.

F. Delay Job Scheduling

In EclipseMR, we implemented a variant of the *delay scheduling* algorithm [34] to compare it against LAF job scheduling algorithm.

Since the EclipseMR job scheduler does not keep track of which server has what cached data objects but it just manages the hash key range of distributed in-memory caches, our variant of the delay scheduling algorithm schedules a task to a worker server that is likely to have reusable cached data based on the hash key ranges of distributed in-memory caches. If the selected server is busy and the task can not start after a specified time (5 seconds in Spark), the task is reassigned to another server as in Spark's delay scheduling. Note that the delay scheduling algorithm does not adjust the hash key ranges of in-memory caches, but they are fixed and aligned with the hash key ranges of the DHT file system.

III. Evaluation

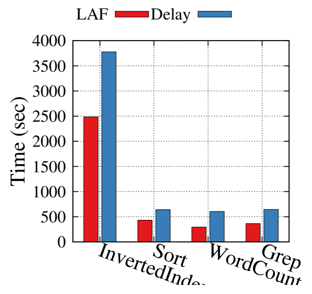
In this section, we first evaluate the performance of EclipseMR by quantifying the impact of each feature we presented. We then compare the performance of EclipseMR against Hadoop 2.5 and Spark 1.2.0. We have implemented the core modules of EclipseMR prototype in about 17,000 lines of C++ code. The source code that we used for the experiments is available at http://github.com/DICL/Eclipse.

We run the experiments on a 40-nodes (640 vCPUs) Linux cluster that runs CentOS 5.5. Each node has dual Intel Xeon Quad-core E5506 processors, 20 GBytes DDR3 ECC memory, and a 5400rpm 256GB HDD for OS and a single 7200rpm 2TB HDD for HDFS and the DHT file system. 20 nodes are connected via a 1G Ethernet switch, the other 20 nodes are connected via another 1G Ethernet switch, and another 1G Ethernet switch forms the two level network hierarchy. We set both the number of map task slots and the number of reduce task slots to 8 (total 640 slots).

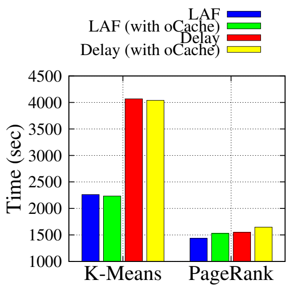
We use HiBench [1] to generate 250 GB text input datasets for the `word count`, `inverted index`, `grep`, and `sort` applications, 15 GB graph input datasets for `page rank`, and 250 GB `kmeans` datasets. In [4], [9], they report the median input sizes for the majority of data analytics jobs in Microsoft and Yahoo datacenters are under 14 GBytes. Hence, we also evaluate the performance of EclipseMR with small 15 GB text input datasets that we collect from Wikipedia and 15 GB k-means datasets that we synthetically generate with varying distributions.

A. IO Throughput

In the experiments shown in Figure 5(a), we measure the read throughput (total bytes/map task execution time) of the DHT file system and HDFS using HDFS DFSIO benchmark while varying the number of servers. As can be seen in Figure 5(a), HDFS and DHT file system show similar IO throughput. Note that this metric does not include the overhead of NameNode directory lookup and job scheduling, but it measures the read latency of local disks. In Figure 5(b), we



(a) Non-Iterative MapReduce Jobs



(b) Iterative MapReduce Jobs

Fig. 6. *Job Execution Time with LAF and Delay Scheduler*

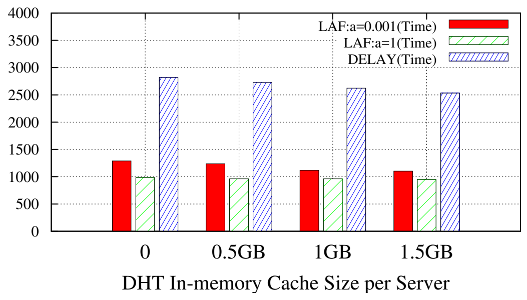
measure the read throughput in a different way, i.e., total bytes/job execution time. While the DHT file system has negligible overhead in decentralized directory lookup and job scheduling, Hadoop suffers from various overheads including NameNode lookup, container initialization, and job scheduling.

In order to evaluate the scalability of the DHT routing table and HDFS name node, we submitted multiple concurrent DFSIO jobs in an experiment that we do not show due to the page limit, and we observed that the IO throughput of HDFS degrades at a much faster rate than the DHT file system.

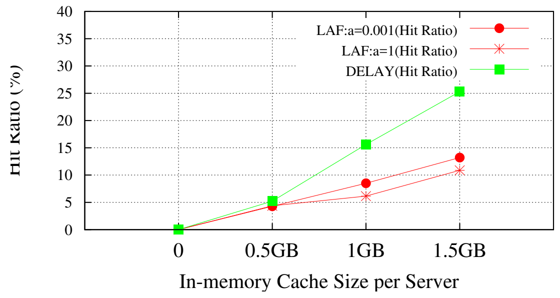
B. LAF Scheduling vs. Delay Scheduling

Figure 6 shows the job execution times of EclipseMR applications using either the LAF scheduling algorithm or the delay scheduling algorithm. Before each job submission, we empty the OS page cache as well as the distributed in-memory caches in EclipseMR. We set the size of buffers for intermediate results to 32 MB so that map tasks spill the buffered results to the DHT file system in discrete 32 MB chunks. Since we submit a single MapReduce job with cold cache, the non-iterative jobs do not benefit from cached data reuse. The LAF scheduling algorithm consistently outperforms the delay scheduling algorithm for all non-iterative MapReduce jobs because it does not make tasks wait for 5 seconds to read a 128 MB input block from cache and it achieves better load balancing. Unlike the LAF job scheduling, the delay scheduling algorithm assigns incoming tasks based on fixed hash key ranges of the DHT file system. Thus the delay scheduling algorithm often fails to adapt to uneven distribution of input hash keys and waits in the queue for a predefined time (5 sec as suggested by [35], [34]) even when there are idle slots in other worker servers.

As for the iterative jobs shown in Figure 6(b), we empty the OS page cache but enable the distributed in-memory caches so that the subsequent iterations can benefit from in-memory caching. We set the size of distributed in-memory cache per server to 1 GB, which is large enough to hold all iteration outputs. We make both `page rank` and `kmeans` iterate 5 times. For comparison, we run experiments with the in-memory caching disabled for the iteration outputs. The output of each iteration in the `kmeans` application is just a set of cluster center points. Hence the size of the iteration output



(a) Job Execution Time



(b) Cache Hit Ratio

Fig. 7. *Performance with Varying LAF Weight Factor*

(1.7 KB) is much smaller than the input file size (250 GB). However, the output of each iteration in the `page rank` application is the pairs of document ids and updated ranks, thus the size of the iteration output in `page rank` is much larger (about 15 GB).

Interestingly, the in-memory caching for iteration outputs does not help improve the performance. This is because when EclipseMR stores the iteration outputs in the DHT file system, they are stored in OS page caches as well. The LAF scheduling algorithm performs better than the delay scheduling algorithm in both iterative applications. But the performance gap in `kmeans` is larger than that in `page rank` mainly because the input file size is larger. With 250 GB input datasets for `kmeans`, we run 4000 mappers. But 15 GB input datasets for `page rank` need only 240 mappers. Since our testbed cluster has 304 slots for map tasks, `page rank` does not have a load balancing issue in both scheduling policies.

C. Load Balancing and Data Locality

In the experiments shown in Figure 7, we synthetically generated `grep` workloads that access input data blocks that are distributed across the DHT file system in a non-uniform distribution. That is, we synthetically merge two normal distributions that have different average hash keys as shown in Figure 3. We submit 24 jobs that run a total of 6410 map tasks reading a total of 90 GB of data blocks.

Figure 7(b) shows that delay scheduling yields a higher cache hit ratio since the hash key ranges of in-memory caches are static and it waits in the queue to reuse cached data even if the server is busy. But the total job execution time of the delay scheduling is up to 2.86x slower than the LAF job scheduling, which achieves better load balancing especially

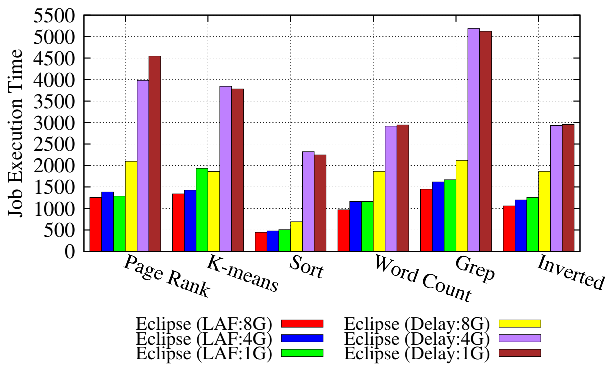


Fig. 8. *Execution time of each application with various cache size when multiple jobs run concurrently*

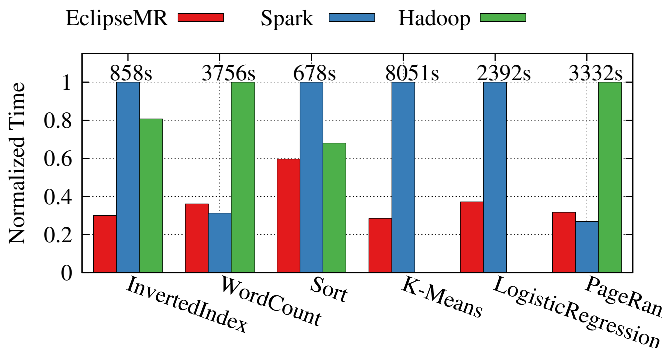


Fig. 9. Performance comparison against Hadoop and Spark

when the accessed hash keys are not uniformly distributed. In the experiments, the standard deviation of the number of processed tasks per slot is just 4.07 for the LAF job scheduling policy but the standard deviation of the delay job scheduling is 13.07, which can result in 3.25x higher job response time.

As the cache size increases, the cache hit ratios of both job scheduling policies increase and the job execution times decrease linearly. For the LAF job scheduling, we vary the weight factor α from 0.001 to 1 and measure the performance spectrum. When α is 1, the LAF scheduling evenly distributes the current workload across servers, i.e., it achieves the perfect load balance. When α is 0, it behaves similar to the delay scheduling algorithm but its waiting time is unlimited. When the α is set to 1, its cache hit ratio is lower than when α is 0.001 ($\sim$10.8% vs. $\sim$13.2%) but its job execution time is faster due to better load balancing behavior.

In general, we observe that a small α such as 0.001 exhibits good performance for various applications especially when a large number of subsequent jobs are submitted as in time series. But for other types of applications which can more benefit from cache hits, a different weight factor α can be used. We fix α to 0.001 for the rest of the experiments.

D. Multiple Concurrent Jobs

Figure 8 shows the execution time of each application when multiple concurrent jobs run and compete for resources. We submit a batch of 7 jobs (2 `grep`, 2 `word count`, 1 `page rank`, 1 `sort`, and 1 `k-means` jobs) at the same time but with smaller input files. `word count` and `grep` jobs share the same 15 GB input data, and the other applications access their own 15 GB input datasets. We vary the size of distributed in-memory cache per worker server up to 8 GB, and set the payload buffer size to 32MB.

As we discussed earlier, Figure 8 shows that a large cache size helps increase the number of available slots and improves job processing throughput. When the cache size is small (1 GB), the overall cache hit ratio of the LAF scheduling policy is 14% while the delay scheduling shows 8% cache hit ratio. When the cache size is 4 GB, the hit ratio of the LAF scheduling policy is 15% and that of the delay scheduling policy is 10%. When the cache size is 8 GB, both scheduling policies show about 69% cache hit ratio. Since we submit a large number of tasks that access various input files, the hash keys are not skewed unlike the experiments shown in Figure 7. Thus, the cache hit ratio with LAF scheduling is not lower than the delay scheduling policy. As a result, the LAF scheduling policy consistently outperforms the delay scheduling policy due to its superior load balancing behavior.

When the hash key boundaries are static, some worker servers can be more overloaded than others in the delay scheduling policy. If the cache capacity is inadequate to hold the working set, the waiting tasks in the overloaded server should share the cache while fewer tasks in less loaded severs can utilize a larger cache space, which explains the low cache hit ratio and poor performance of the delay scheduling policy with small cache sizes in the experiments. On the contrary, the LAF scheduling policy evenly distributes the tasks across multiple servers. Hence, even when the size of the in-memory cache is small, the LAF scheduling policy utilizes the entire distributed in-memory caches well, and it results in a high cache hit ratio, good load balancing, and faster job execution times.

E. Comparison with Hadoop and Spark

Finally we compare the performance of EclipseMR against Hadoop and Spark. Since all three frameworks provide different *levers* for tuning, we performed Hadoop and Spark tunings to our best efforts according to various performance tuning guides available on the web.

However, it is hard to quantify which design of EclipseMR contributes to the performance differences because EclipseMR does not share any component with Hadoop software stack. Moreover, Hadoop and Spark are full featured frameworks that provide various generic functionalities that are usually followed by significant overhead. For an example, Hadoop tasks run in Yarn containers and each Yarn container spends more than 7 seconds for initialization and authentication [17]. This overhead becomes significant because the container initialization and authentication repeats for every task. I.e., Hadoop spends 7 seconds for every 128 MB block [16]. Compared to Hadoop and Spark, EclipseMR is a lightweight prototype framework that does not provide any other functionalities than what we present in this paper.

We use the default *fair* scheduling in Hadoop, and the delay scheduling in Spark. Again we submit a single application that accesses 250 GB datasets (or 15 GB datasets for `page rank`) at a time after emptying the OS buffer cache and distributed in-memory caches for non-iterative MapReduce applications. For `page rank`, `kmeans`, and `logistic regression` applications, we enable distributed in-memory caches and set the size of the cache to 1 GB per server.

Figure 9 shows the normalized execution time to the slowest result. For the non-iterative MapReduce applications, Spark often shows slightly worse performance than Hadoop, which we believe is because Spark is specifically tailored for iterative jobs such as `page rank`, `kmeans`, and `logistic regression`, not for non-iterative ETL jobs such as `inverted index`. For non-iterative jobs, all three frameworks do not benefit from caching. Therefore, the performance differences are mainly caused by scheduling decisions.

The performance of `sort` in Figure 9 shows how efficiently each framework performs the shuffle phase. Spark is known to perform worse than Hadoop for `sort`, and our experiments also confirm it. Spark claims it has improved `sort` since version 1.1, but our experiments with version 1.6 show that Spark is still outperformed by Hadoop and EclipseMR.

For iterative applications, we set the number of iterations to 5 for the `kmeans` application, 2 for the `page rank` application, and 10 for the `logistic regression` application. Since Hadoop is an order of magnitude slower than the other two frameworks, we omit the performance of Hadoop `kmeans` and `logistic regression`.

For `kmeans`, EclipseMR is about 3.5x faster than Spark. This result is consistent with the results we presented earlier in Figure 6(b). The LAF scheduling algorithm yields about 2x faster job execution time than the delay scheduling algorithm for `kmeans` application. For `logistic regression`, EclipseMR is about 2.5x faster than Spark. Note that our faster C++ implementations of `kmeans` and `logistic regression` contributed to the performance improvement, but there are other performance factors that are not out of scope of this this research, i.e., Java heap management, container overhead, and some engineering issues that make Spark tasks unstable also need to be investigated.

For `page rank` application, Spark is about 15% faster than EclipseMR. This result is also consistent with the results presented in Figure 6(b); the LAF scheduling algorithm shows similar performance to the delay scheduling algorithm for the `page rank` application. This is because the size of the input file in `page rank` is small and our cluster has a large enough number of slots to run all the mappers concurrently. So, there's no load balancing issues. Moreover, `page rank` generates a very large output for each iteration; the size of iteration outputs in `page rank` is often similar to that of input data. While Spark does not store the intermediate outputs in file systems, EclipseMR writes the large iteration outputs to the persistent DHT file systems to provide fault tolerance. Therefore, if the size of intermediate results is large, the performance gap between EclipseMR and Spark decreases and EclipseMR is outperformed by Spark.

F. Iterative Applications

In the experiments shown in Figure 10 we further analyze the performance of EclipseMR and Spark for iterative applications - `k-means`, `logistic regression`, and `page rank`. Spark runs the first iteration of the iterative applications much slower than subsequent iterations because it constructs RDDs that can be used by subsequent iterations. For subsequent iterations of `kmeans` and `logistic regression`, EclipseMR runs 3x faster than Spark because it does not wait to be scheduled on the servers that has the iteration outputs in their caches, but it immediately starts running in a remote server and accesses remote cached data.

Similar to `kmeans` and `logistic regression`, `page rank` also runs subsequent iterations faster than the first iteration by taking advantage of input data caching. Unlike `kmeans`, EclipseMR is outperformed by Spark for subsequent `page rank` iterations mainly because EclipseMR writes large iteration outputs to the DHT file system. However, even if EclipseMR writes to slow disks, EclipseMR is at most 30% slower than Spark. With a small 30% IO overhead, EclipseMR can restart from the iteration if system crashes.

Note that Spark runs `page rank` slower than EclipseMR in the last iteration because Spark writes its final outputs to disk storage. The last iterations of `kmeans` and `logistic regression` are not slower than the previous iterations because the outputs of these applications are not as large as `page rank`.

IV. Related Work

As the demand for large-scale data analysis frameworks grew in the high performance computing community in the late '90s, several distributed and parallel data analysis frameworks such as *Active Data Repository* [18], which supports the MapReduce programming paradigm and *DataCutter* [5], which supports generic DAG workflows, were developed for large scale scientific datasets. A few years later, industry had a growing demand for large-scale data processing applications and Google developed Google File System [10] and the MapReduce framework [7]. Since then, there has been a great amount of effort to extend and improve distributed job processing frameworks for various data-intensive applications [6], [23], [27], [28], [35].

Spark [35], [34] shares the same goal as our framework in that it reuses a working set of data across multiple parallel operations. *Resilient Distributed Datasets* (RDDs) in Spark are read-only in-memory data objects that can be reused for subsequent MapReduce tasks. Spark addresses the conflict between job scheduling fairness with data locality by delaying a job for a small amount of time if the job can not launch a local task [33]. Our EclipseMR job scheduling is different from Spark in that EclipseMR employs consistent hashing to

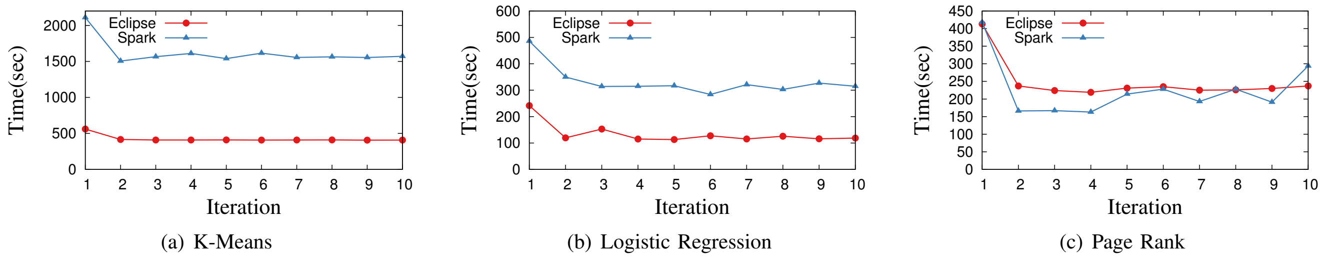


Fig. 10. *Execution Time of Iterative Jobs*

determine where to store and access the cached data objects. Based on the consistent hashing, EclipseMR strikes a balacne between load balancing and data locality. *Dache* is another MapReduce framework where a central cache manager uses its best efforts to reuse the cached results of previous jobs [36]. Compared to Dache, EclipseMR is more scalable as it does not have a central directory to keep the list of cached data objects that can change dynamically at a very fast pace.

Main Memory MapReduce (*M3R*) proposed by Shinnar et al. [28] is a MapReduce framework that performs in-memory shuffle by simply storing the intermediate results of map tasks in main memory instead of the block device storage. They show in-memory shuffle significantly improves a certain type of applications. However, M3R can not be used if workloads are large and do not fit in main memory or applications require *resilience* because the in-memory framework is not fault tolerant. Moreover, it is questionable if MapReduce is the right programming paradigm for their target application - sparse matrix vector multiplication. Rahman et al. [25] proposed *HOMR* - a hybrid approach to achieve the maximum possible overlapping across map, shuffle, and reduce phases. Our work is similar to that in the sense that EclipseMR aggressively overlap three phases by proactive shuffling. The in-memory caching layer of EclipseMR is similar to *Tachyon*, which is the cache layer that sits on top of HDFS and acts as a distributed cache for Spark. The difference between Tachyon and EclipseMR in-memory caching is that EclipseMR caching evenly distributes popular cached objects via LAF algorithm.

DryadInc [23] is an incremental computation framework that allows computations to reuse partial results of the computations from previous runs. Tiwari et al. [30] proposed the *MapReuse* delta engine as an in-memory MapReduce framework that detects input data similarity, reuses available cached intermediate results and computes only for the new portion of input data. They show the reuse of intermediate results significantly improves job execution time. *ReStore* [8] is another framework that stores intermediate results generated by map tasks in HDFS so that they can be reused by subsequent jobs. Their works are similar to ours but the conventional fair job scheduling policies they use do not guarantee balancing the workloads if requested intermediate results are available only on a small number of overloaded servers.

The *MRShare* framework proposed by Nykiel et al. [22] merges a batch of MapReduce queries into a single query so that it takes the benefits of sharing input and output data across multiple queries. The multiple query optimization problem has been extensively studied in the past, and it has been proven to be an NP problem. Nevertheless, extensive research has been conducted to minimize query processing time through data and computation reuse using heuristics or probabilistic efforts [21], [24], [23], [8]. These multiple query optimization studies are complementary to our work.

V. Conclusion

In this work, we present *EclipseMR*, a distributed in-memory caching and job processing framework for data-intensive applications. i) EclipseMR replaces HDFS with a robust and scalable DHT file system. ii) EclipseMR employs a distributed in-memory cache layer that maximizes the chances of data reuse, which also helps solve data skew problem. iii) We design and implement a novel locality-aware fair job scheduling policy that employs consistent hashing. In a large distributed query processing systems with semantic caches, striking the balance between cache hit ratio and load balancing plays a critical role in determining the degree of performance improvement. Our proposed scheduling algorithm considers load balancing and cache hit ratio at the same time. iv) By employing consistent hashing, we enable the proactive shuffling that allows map tasks to generate and send intermediate results to reduce tasks as soon as the results are generated.

Our experimental study show that each design and implementation of EclipseMR components contributes to improving the performance of distributed MapReduce processing. We show EclipseMR outperforms Hadoop and Spark for several representative benchmark applications including iterative applications.

As a future work, we plan to decompose EclipseMR into a standalone DHT file system, a distributed in-memory key-value store that employs LAF algorithm, and a generic distributed job processing framework implemented in C++ that can easily adapt high performance computing technologies.

VI. Acknowledgement

§ The first three authors contributed equally in this work. This research was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-SRFC-IT1501-04.

References

[1] HiBench. *https://github.com/intel-hadoop/HiBench*.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *USENIX Annual Technical Conference*, 2014.

[4] R. Appuswamy, C. Gkantsidis, D. Narayana, O. Hodson, and A. Rowstron. Scale-up vs scale-out for Hadoop: Time to rethink? In *4th annual Symposium on Cloud Computing (SOCC)*, 2013.

[5] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133, Mar. 2000.

[6] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce Online. In *the 7th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[8] I. Elghandour and A. Aboulnaga. ReStore: Reusing results of MapReduce jobs. *Proceedings of the VLDB Endowment (PVLDB)*, 5(6):586–597, 2012.

[9] K. Elmeleegy. Piranha: Optimizing short jobs in Hadoop. *Proceedings of the VLDB Endowment (PVLDB)*, 6(11):985–996, 2013.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[11] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, pages 75–88, 2010.

[12] Y. Guo, J. Rao, and X. Zhou. iShuffle: Improving hadoop performance with shuffle-on-write. In *10th USENIX International Conference on Autonomic Computing (ICAC)*, pages 107–117, 2013.

[13] A. Gupta, B. Liskov, and R. Rodrigues. One hop lookups for peer-to-peer overlays. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS)*, pages 7–12, Lihue, Hawaii, May 2003.

[14] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query co-processing on graphics processors. *ACM Transactions on Database Systems*, 34(4):21–39, 2009.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *2nd ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys)*, 2007.

[16] W. Kim, Y. ri Choi, and B. Nam. Coalescing HDFS blocks to avoid recurring yarn container overhead. In *10th International Conference on Cloud Computing (IEEE Cloud)*, 2017.

[17] W. Kim, Y. ri Choi, and B. Nam. Mitigating YARN container overhead with input splits. In *17th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2017.

[18] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. In *Proceedings of the ACM/IEEE SC1999 Conference*, 1999.

[19] J. Kwak, E. Hwang, T. kyung Yoo, B. Nam, and Y. ri Choi. In-memory caching orchestration for Hadoop. In *Proceedings of the 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[20] Y. Kwon, M. Balaziniska, B. Howe, and J. Rolia. SkewTune: mitigating skew in mapreduce applications. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, 2012.

[21] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):546–585, 1998.

[22] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Sharing across multiple MapReduce jobs. *ACM Transactions on Database Systems*, 39(2):12:1–12:46, 2014.

[23] L. Popa, M. Budiu, Y. Yu, and M. Isard. Dryadinc: Reusing work in large-scale computations. In *Proceedings of the 2009 USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[24] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1989.

[25] M. W. Rahman, X. Lu, N. S. Islam, and D. K. Panda. HOMR: A hybrid approach to exploit maximum overlapping in MapReduce over high performnace interconnects. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*, 2014.

[26] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2012.

[27] S. Sakr, A. Liu, and A. G. Fayoumi. The family of MapReduce and large-scale data processing systems. *ACM Computing Surveys*, 46(1):11:1–11:44, 2013.

[28] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. *Proceedings of the VLDB Endowment (PVLDB)*, 5(12):1736–1747, 2012.

[29] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[30] D. Tiwari and Y. Solihin. MapReuse: Reusing computation in an in-memory MapReduce system. In *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2013.

[32] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of the ACM/IEEE SC2011 Conference*, 2011.

[33] 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 (EuroSys)*, 2010.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2010 USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[36] Y. Zhao and J. Wu. Dache: A data aware caching for big-data applications using the MapReduce framework. In *Proceedings of INFOCOM*, pages 271–282, 2013.