EARNCache: Self-adaptive Incremental Big Data Caching on Non-Big Clusters

Junshi Guo
Fudan University

Yifeng Luo
Fudan University

Eric Lo
The Chinese University of Hong Kong

Shuigeng Zhou
Fudan University

Abstract

Memory caching plays a crucial role in determining whether people's requirements on (quasi-)real-time processing of exploding data on big-data clusters could be satisfied. As big data clusters usually are concurrently shared by multiple computing frames, applications or end users, there exists intense competition for memory cache resources, especially on small clusters which are supposed to process big datasets comparable to large clusters yet with tightly limited resources. Applying existing on-demand caching strategies on small shared clusters inevitably results in frequent cache thrashings when the conflicts of simultaneous cache resource demands are not mediated, and then the running efficiency of the whole big data cluster will be deteriorated.

In this paper, we propose a novel self-adaptive incremental big data caching mechanism, called EARNCache, to improve the cache efficiency for shared big data clusters, especially for small clusters where cache thrashings are more likely to occur. EARNCache self-adaptively adjusts resource allocation strategies according to the competition condition of cache resources: evolving to incremental caching to depress competition when resources are in deficit, devolving to traditional on-demand-caching to expedite data caching-in when resources are in surplus. Our experimental evaluation shows that EARNCache's elasticity could enhance the cache efficiency for shared big data clusters, with improved resource utilization.

1 Introduction

Memory caching plays a crucial role in bridging the performance gap between storage subsystems and computing frameworks, which gradually becomes the determinant of whether processing units of big data platforms could work at wire speed to satisfy people's growing computing requirements. As more and more time-critical applications commence employing memory to cache datasets, big data clusters are usually shared by multiple computing frameworks, applications or end users, and there exists intense competition for memory cache resources, especially on small clusters which are supposed to process comparably big datasets as large clusters do, yet with tightly limited resources. Consequently implementing effective and adaptive management on memory cache resources becomes increasingly important for the efficient running of big data clusters, especially for small/medium enterprises who could only afford to build non-big clusters. Applying existing on-demand caching strategies on small shared clusters inevitably results in frequent cache thrashings when the conflicts of simultaneous cache resource demands are not mediated, leading to deterioration in overall cluster efficiency. The principal reason is that aggressively caching massive number of data blocks of big datasets on demand causes constant block replacement in cache, and thus conversely exacerbates resource competition.

Web or traditional OLTP database applications usually produces vast difference in range or block "hotness" regarding access recency and frequency. Though in big data applications, input files are usually scanned as a whole for data processing and all blocks reveal almost equal "hotness". On the other hand, traditional system-level or database-level data caching is executed on small data units (i.e. 8KB-sized pages), while big data caching is executed on much larger units(i.e. 256MB-sized blocks). So the cost of caching in/out a data unit in big data scenarios far exceeds that of traditional data caching. Accordingly, traditional caching may have millions of cache slots, and this makes hotter data less likely cached out by colder data; while big data caching may only have thousands of slots, which shows almost no preference for particular slots.

Targeting the caching problem existing on non-big clusters, we propose an adaptive cache mechanism, which is named as EARNCache (from s**E**lf-**A**daptive inc**R**eme**N**tal **Cache**), to coordinate concurrent cache resource demands to prevent exacerbated cache efficiency ~~in this paper~~, when intense competitions for memory cache resources occur. As big data applications usually access their input data in Write-Once-Read-Many (WORM) fashion, we only consider read caching in this paper. Major contributions of this paper include: 1) proposing an incremental caching mechanism which could self-adaptively adjusts cache allocation strategies according to the competition condition of cache resources; 2) formulating and solving the cache resource allocation and replacement problem as an optimization problem; 3) implementing a prototype of the proposed method, and performing extensive experiments to evaluate the effectiveness of the proposed mechanism.

In the rest of this paper, we first illustrate the system design and implementation details of EARNCache in Sec. 2. We provide empirical results in Sec. 3, and present the related work in Sec. 4. We finally conclude the paper in Sec. 5.

2 System Framework

We mainly illustrate how EARNCache works in this section. Firstly we present the overview about the cache-earning mechanism of EARNCache, and then illustrate its architecture design, and finally explains the incremental cache-earning policy and its implementation.

2.1 Overview

On shared non-big clusters with relatively limited cache capacity, cache resource conflicts ~~on non-big clusters~~ would be norm rather than exception. When only quite few users are using a non-big cluster and the competition for cache resources is mild, applying on-demand caching could expedite hotter blocks taking over cache resources from colder blocks, and hotter data is less likely to be cached out by colder data. When more concurrent users are using the cluster and the competition for cache resources gets wild, on-demand caching leaves concurrently cache resource demands unmediated, which makes hotter data more vulnerable to being cached out. Files which are frequently accessed recently sometimes could be totally cached out by files which would rarely be accessed for the second time in the near future, then these hot cached-out files need caching in soon as their next access should occur in the incoming future. We consequently need to revisit existing on-demand caching mechanisms and strategies, and propose more effective measures to improve the efficiency of data caching on non-big clusters.

We believe that a good caching strategy for non-big clusters should be self-adaptive to resource competition conditions to depress competitions and avoid thrashings when resources are in desperate deficit. Obviously caching big data files on demand as a whole could not provide such self-adaptivity. Intuitively we should not cache files entirely ~~when caching them entirely causes tension.~~ Not compulsorily caching files entirely provides the elasticity of tuning the amount of cache resources allocated for different files, according to the files' access recency and frequency.

Ideally, more recently frequently accessed files should be assigned with more cache resources, and less recently frequently accessed files should be assigned with less cache resources. However, it's not possible to know in advance what files would be frequently accessed in the upcoming future and we could only make predictions based on historical file access patterns, especially the most recent information. Based on files' historical access information, EARNCache implements an incremental caching strategy, where a user should earn resources to cache files from other concurrent users via accessing these files. Cache resources are incrementally allocated to files becoming frequently accessed, which gradually takes over cache resources from files getting less frequently accessed, until all blocks of the file have been cached in. The more a file is accessed, the more cache resources it takes over. The incremental caching strategy ensures that cached hot files being frequently accessed recently will not be flushed out by other massive less hot files, which may only be accessed occasionally or randomly.

2.2 Architecture

Cached files originally are stored in the under distributed file system (i.e. Hadoop File System), and EARNCache ~~globally~~ coordinately cache them across the whole cluster. EARNCache's architecture consists of a central *master* and a set of *workers* residing on storage nodes of the cluster(see Fig. 1). The master's main role is: 1) to determine how many cache resources should be allocated to a file, based on the resource competitions; 2) to inform workers of cache resource allocation plans via heartbeats; 3) to keep track of cache metadata about which storage node a cached block resides on; 4) to answer clients' queries on cache metadata. And the worker's main role is: 1) to receive resource allocation plans from the master; 2) to calculate how many resources a file should contribute to compose the allocated resources; 3) to cache in/out blocks according to the calculated resource composition plans; 4) to inform the master of cached blocks via heartbeats; 5) serve clients with cached blocks.

The procedures of a client accessing a block are: 1) the client queries the master where the block is located; 2) the master tells the client which worker it should contact to access the block; 3) the client contacts the worker to access the block; 4) the worker serves the client with the block data from cache. One thing worth noting here is that: the client will not contact any worker to access a block if the block has been cache out, as the master only keeps track of cached blocks, and could not provide the client with information about cached-out blocks, then the client has to turn to the under file system for this block.

2.3 Incremental Caching

As we prefer to letting frequently accessed files recently incrementally take over resources from less frequently accessed files recently, "recently" should be defined quantitatively before we could design the incremental caching

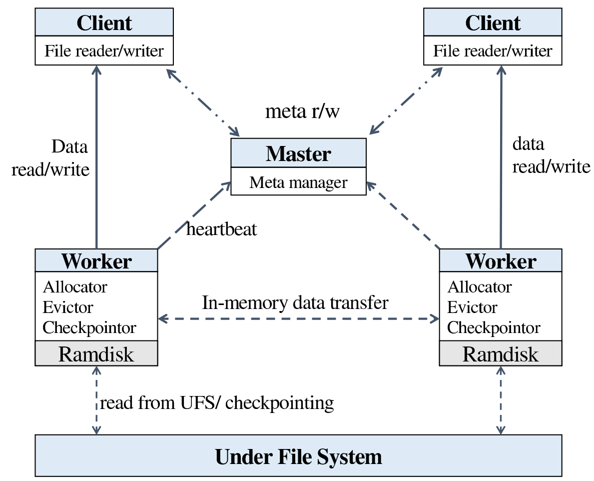


Figure 1: EARNCache's Architecture.

strategy, and other related elements should also be defined. Tab. 1 presents all definitions of notations involved in our incremental caching strategy.

Table 1: Notation definitions

Notation	Definition
W	predefined window size of the most recently accessed data for observing files falling within
a, b	scan time per unit data from memory(a) and hdd(b)
N	total number of files falling in the observation window
d_i	data size of the ith file
D	total data size of N files
M	cache capacity of the whole cluster
f_i	access frequency of the ith file
F	total access frequency of N files
x_i	percentage of data cached for the ith file
$h_i(x_i)$	the ith file's profit gain with x_i data cached

We define a function $h_i(x_i)$ to denote the profit gain of the ith file to instruct how cached files should contribute resources. Then we attempt to instantiate $h_i(x_i)$ and to maximize total profit gain of all files falling in the observation window, just as Equ.1 shows.

$$\sum_{i=1}^{N} f_i \cdot h_i(x_i) \tag{1}$$

According to definitions in Tab.1, we can assume that the time it takes to scan the ith file is:

$$time(x_i) = [a \cdot x_i + b \cdot (1 - x_i)] \cdot d_i \tag{2}$$

As mentioned above, we use h_i to indicate the ith file's profit gain with x_i data cached. If we take saved scan time as a file's profit gain, then we can define h_i's deviation at x_i as its gain change over δx_i, which could be further defined as the percentage of increased saving of the file's scan time with increased cache share at x_i over the total saved scan time at x_i, compared to zero cache share, just formulized as:

$$\frac{\delta h_i}{\delta x_i} = \frac{time(x_i) - time(x_i + \delta x_i)}{time(0) - time(x_i)} = \frac{\delta x_i}{x_i} \tag{3}$$

Thus we can derive that $h_i(x_i) = \ln x_i$, and now our optimization goal becomes

$$\sum_{i=1}^{N} f_i \cdot \ln x_i \tag{4}$$

subjected to

$$\sum_{i=1}^{N} x_i \cdot d_i \leq M \tag{5}$$

Note at any given time, x_i is the only variant contained in the optimization goal, and $f_i \cdot \ln x_i$ is a convex function. After applying Lagrange multiplier method, our maximizing goal turns to:

$$L = \sum_{i=1}^{N} f_i \cdot \ln x_i - \lambda(\sum_{i=1}^{N} x_i \cdot d_i - M) \tag{6}$$

Let $\frac{\delta L}{\delta x_i}$ be 0, then we get

$$x_i \cdot d_i = \frac{f_i}{F} \cdot M \tag{7}$$

The above result shows that the amount of memory resources allocated to a file is linear to f_i at a given moment, as all files' access frequency is determined at that given moment, which exactly responds to our original intention of incremental caching. One more thing worth noting is that: if the overall size of files filling in the whole observation window is less than the cache capacity, which means that there are cache resources being occupied by files falling out of the observation window, EARNCache will collect resources from those obsolete files falling out of the observation window by LRU when a file needs caching, and the file needs caching could cache in its blocks once and for all, rather than gradually taking over resources from files falling within the observation window. EARNCache thereby could adaptively devolve to traditional on-demand-caching so as to expedite the process of collecting cache resources for files that need caching when the contention for cache resources is minute, and evolve to incremental caching to depress competition when resources are in deficit.

2.4 Implementation Details

We implemented EARNCache by implanting our incremental caching mechanism into the modified Tachyon[4]. In EARNCache, we first evenly re-distribute a file's cached data blocks across the whole cluster, so that almost the same amount of blocks are hosted in cache on each cluster node, and all workers can manage their cache resources independently yet still in concert. As uneven data distribution will drag down completion of the whole job, evenly distributing cached data blocks guarantee that tasks running on each node could ideally finish almost simultaneously.

When the ith file needs caching, EARNCache preallocate f_i/F fraction of cache resources on each node to the file based on Equ.7. If the resources pre-allocated to the file is larger than its aggregate resource demand, EARNCache has other files in need of cache resources fairly share the resources beyond the file's actual need. Each worker checks its remaining free cache resources, and allocates as many resources as necessary to them directly if enough free resources are available, which could make full use of cache resources. When there are not enough remaining resources, the worker calls *BlocksToEvict()*, which implements the eviction algorithm with incremental caching, to determine which blocks should be cached out. As all blocks are cached in from the underlying file system, cached-out blocks need no more backup and workers could discard them directly from cache. After blocks being cached in/out, workers informs the master of cached-in/out blocks, and then the master updates the metadata about caching.

Alg. 1 describes the process of caching out blocks. EARNCache first checks whether the file requesting cache resources has used up its resource share in Line 1∼3. In the while loop, Line 7∼14 mainly selects files whose occupied cache resources exceed the most than their computed shares. If no such file exists, EARNCache will reject the cache resource request (Line 15∼17). Otherwise, blocks of these selected files are added to the candidate cached-out blocks until enough cache resources have been collected(Line 18∼24). As the same recency and frequency of all blocks within a file are identical, there exists no difference between blocks of the same file for EARNCache workers when choosing cached-out blocks, and workers could cache out any in-cache block of the file.

3 Experiment

We deploy an HDFS cluster to evaluate EarnCache's performance, on which Spark and EarnCache are deployed as the upper-level application tier and the middle-level caching tier respectively. The cluster consists of five Amazon EC2 m4.2xlarge nodes, one of which serves as the masters of Spark, EarnCache and the underlying HDFS, four of which serves as slaves of Spark, EarnCache and HDFS. Each cluster node has 32GB of memory, 12GB reserved as working memory and the remaining 20GB of memory employed as cache resources, summing up to 80GB of cache in total.

We mainly evaluated EarnCache's performance by issuing jobs from Spark to scan cached files parallelly ~~while~~ without any further processing, and compare the performance of EarnCache incremental caching with LRU, LFU on-demand caching and with the MAX-MIN fair caching. We set the observation window size of EarnCache to 1000GB by default. For each experiment, we issue file scanning jobs on three cached files, denoted as FILE-1, FILE-2 and FILE-3, with the following three various frequency patterns, denoted as ROUND, ONE and TWO respectively.

Algorithm 1 Eviction Algorithm: BlocksToEvict()

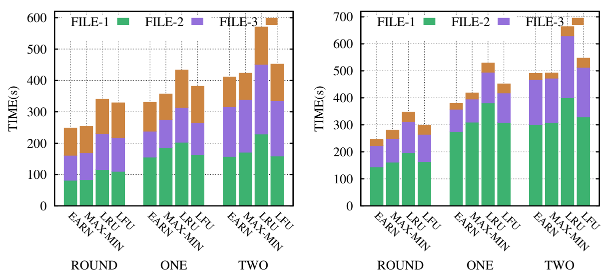
Input: *s*, requested cache resources; *r*, the requesting file id; *A*=$\{a_1, a_2...a_N\}$, a list of files' pre-allocated memory bytes; *C*=$\{c_1, c_2...c_N\}$, a list of current consumed memory bytes in local node; *M*, memory capacity of local node

Output: a list of candidate blocks to evict

1: **if** $c_r \geq a_r$ **then**
2:     algorithm ends as file r has already consumed all its allocated memory
3: **end if**
4: $candidate \leftarrow \{\}$ ▷ *candidate cached out blocks*
5: $mem \leftarrow 0$ ▷ *free resources obtained from evicting candidates*
6: **while** $mem < s$ **do**
7:     $j \leftarrow -1$
8:     $over_j \leftarrow 0$
9:     **for** a_i in A and $i \neq r$ **do**
10:         **if** $c_i - a_i > over_j$ **then**
11:             $j \leftarrow i$
12:             $over_j \leftarrow c_i - a_i$
13:         **end if**
14:     **end for**
15:     **if** $j = -1$ **then**
16:         return as request failure
17:     **end if**
18:     find b_j as a block of file j and not in $candidate$
19:     $candidate \leftarrow candidate + b_j$
20:     $mem \leftarrow mem + sizeof(b_j)$
21:     $c_j \leftarrow c_j - sizeof(b_j)$
22:     **if** $mem \geq s$ **then**
23:         return $candidate$
24:     **end if**
25: **end while**

- ROUND Three files are accessed in pattern: FILE-1, FILE-2,FILE-3 ..., where three files are accessed with equal frequency.
- ONE Three files are accessed in pattern: FILE-1, FILE-2, FILE-1, FILE-3 ..., where one file is accessed more frequently than other two files.
- TWO Three files are accessed in pattern: FILE-1, FILE-2, FILE-1, FILE-2, FILE-3 ..., where two files are accessed more frequently than the other file.

We respectively set the size of FILE-1, FILE-2 and FILE-3 equally to 40GB and unequally to 70GB, 40GB and 10GB, and then evaluate EarnCache with different caching strategies and different frequency patterns. Fig. 2(a) and Fig. 2(b) shows the averaged overall running time of jobs scanning various number of equally and unequally sized files within one period of ROUND, ONE and TWO frequency patterns, including 3, 4, and 5 file scans each. We can see that EarnCache provides the best file scanning performance,



(a) Files with equal size

(b) Files with unequal size

Figure 2: Running time of file scanning jobs.

More specifically, we statistically analyze the distribution of blocks accessed from local cache, from remote cache, and from the under distributed file system respectively, and the results are shown in Fig. 3. We can see that EarnCache has the largest number of blocks accessed from cache, locally or remotely, which means that it yields the highest memory efficiency than other caching strategies, and this self-evidently explains why EarnCache yields the best file scanning performance. However, we can see that EarnCache is the strategy which has the largest number of blocks accessed from remote cache among the four evaluated strategies. The reason is that the Spark task scheduler is not optimized to recognize blocks in exclude cache and under file system in seperate. and this also means EarnCache has the largest potential of performance improvement. If cache-aware task scheduling can be injected into the upper-level task scheduler, more tasks could access their input data blocks from local cache, and EarnCache could obtain much better overall performance than other caching strategies.

We can also see that the performance of EarnCache is

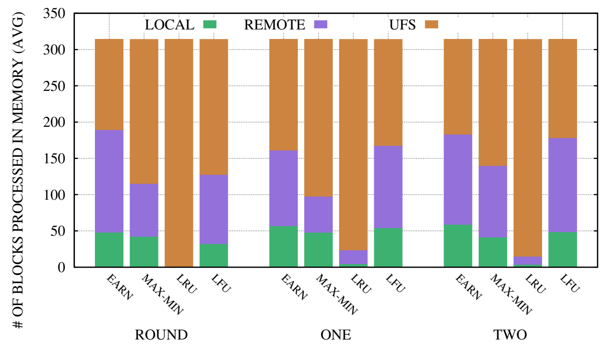
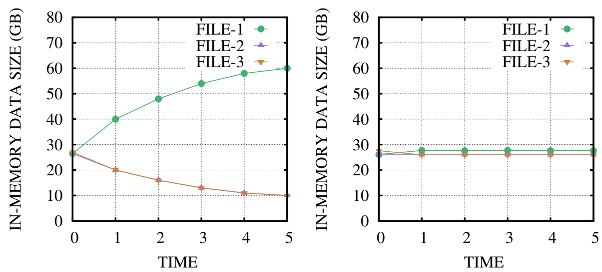


Figure 3: Distribution of blocks accessed in local cache, remote cache and under file system.

only slightly better than the MAX-MIN caching strategy, and sometimes they achieve similar performance. This is because that file receives moderately divergent amounts of cache resources with these two caching strategies, as far as our experimental settings are concerned. However, the MAX-MIN caching is unable to re-allocate resources properly when there exist files not receiving any further accesses. To illustrate this, we comparatively present the process of resource re-allocation of EarnCache and the MAX-MIN caching in Fig. 4(a) and Fig. 4(b), when two out of three equal-sized files stop receiving further accesses. We can see that the file remaining accessed gradually takes over cache resources from those obsolete files as time goes on, while each file with MAX-MIN caching still equally holds 1/3 cache resources even though two files receives no further accesses. Correspondingly, the running time of the file scanning job gradually decreases with EarnCache, while remains stable with MAX-MIN caching.

Finally, we experimentally analyze the impact of the predefined observation window size on the average running time of file scanning jobs, and the results are presented in Fig. 5. We can see that when the observation window size is set too small, the time performance will degrade.



(a) EARNCache (b) MAX-MIN

Figure 4: Dynamic resource re-allocation with two files receiving no further accesses.

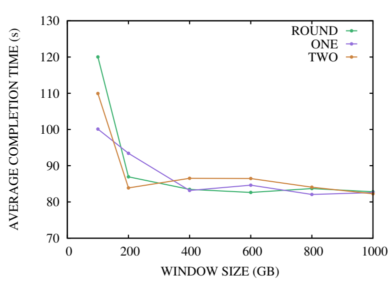


Figure 5: Impact of the observation window size.

4 Related Work

More and more time-critical applications [16, 20] are leveraging massive memory resources to store or cache data to gain improved data access performance, such as J. Ousterhout proposed RAMCloud [2] to keep data entirely stored in memory for large-scale Web applications, and Spark [19, 9] enables in-memory MapReduce [3]-style parallel computing by leveraging memory to store and cache distributed (intermediate) datasets. While caching on distributed parallel systems is tremendously different from traditional centralized page-based file system or database caching, and directly applying centralized caching on distributed parallel systems usually does not help much to improve and sometimes even hurts cache efficiency and performance.

Some previous work focuses on implementing an additional layer on existing distributed file system, which makes applications able to transparently cache distributed datasets from the underlying distributed file system. J. Zhang, et al. [1] and Y. Luo, et al. [11] respectively proposed HDCache and RCSS distributed cache system based on HDFS [6, 17], which manages cached data just as HDFS manages disk data. H. Li, et al. [4] further implemented a distributed memory file system for data caching by checkpointing data to the underlying file system. EARNCache imbeds the incremental caching into Tachyon [4] to coordinate resource competitions and avoid cache thrashings, which improves cache efficiency and resource fairness to a certain degree.

Some work focuses on optimizing data caching for specific frameworks or goal. S. Zhang, et al. [10] proposed to cache MapReduce intermediate data to speed up MapReduce applications. G. Ananthanarayanan, et al.[7] found the important All-or-Nothing property, which implies all or none input data blocks of tasks within the same wave should be cached, and then proposed PACMan to coordinate memory caching for parallel jobs, including LIFE and LFU-F the two block cache eviction policies, where LIFE aims to minimize average completion time of jobs by favoring input files with smaller waves, and LFU-F aims to maximize cluster efficiency by favoring files that are accessed more frequently. Y. Li, et al.[5], S. Tang, et al. [14] and A. Ghodsi, et al. [15] respectively proposed dynamic resource partition strategies to improve fairness, and maximize the overall performance in the meanwhile. Q. Pu, et al. [8] extended the MAX-MIN fairness [12, 13] with probabilistic blocking, and proposed FairRide to avoid cheating and improve fairness for shared cache resources.

5 Conclusion

In this paper, we propose the EarnCache incremental big data caching mechanism for non-big clusters, which self-adaptively adjusts resource allocation strategies according to the competition condition of cache resources: evolving to incremental caching to depress competition when resources are in deficit, falling back to traditional on-demand-caching to expedite data caching-in when resources are in surplus. With EarnCache, files are not cached as a whole on demand, and applications or end users incrementally take over cache resources from others by accessing their datasets. EarnCache achieves improved resource utilization and performance.

References

[1] J. Zhang, G. Wu, X. Hu, et al. A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services. In Proceedings of GRID, 2012, pages 12-21.

[2] J. Ousterhout, P. Agrawal, D. Erickson, et al. The Case for RAMCloud. Communications of the ACM. Vol.54 No.7(2011), pages 121-130.

[3] J. Dean, S. Ghemawat, et al. MapReduce: Simplified data processing on large cluster. In Proceedings of OSDI, 2004, pages 137-150.

[4] H. Li, A. Ghodsi, M. Zaharia, et al. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In Proceedings of SOCC, 2014, pages 6:1-6:15.

[5] Y. Li, D. Feng, Z. Shi. Enhancing Both Fairness and Performance Using Rate-aware Dynamic Storage Cache Partitioning. In Proceedings of DISCS, 2013, pages 31-36.

[6] K. Shvachko, H. Kuang, S. Radia, et al. The Hadoop Distributed File System. In Proceedings of MSST, 2010, pages 121-134.

[7] G. Ananthanarayanan, A. Ghodsi, A. Warfield, et al. PACMan: Coordinated Memory Caching for Parallel Jobs. In Proceedings of NSDI, 2012, pages 267-280.

[8] Q. Pu, H. Li, M. Zaharia, et al. FairRide: Near-Optimal, Fair Cache Sharing. In Proceedings of NSDI, 2016, pages 393-406.

[9] M. Zaharia, M. Chowdhury, T. Das, et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Proceedings of NSDI, 2012, pages 15-28.

[10] S. Zhang, J. Han, Z. Liu, et al. Accelerating MapReduce with Distributed Memory Cache. In Proceedings of ICPADS, 2009, pages 472-478.

[11] Y. Luo, S. Luo, J. Guan, et al. A RAMCloud Storage System based on HDFS: Architecture, Implementation and Evaluation. Journal of Systems and Software. Vol.86 No.3(2013), pages 744-750.

[12] Q. Ma, P. Steenkiste, H. Zhang. Routing High-Bandwidth Traffic in Max-Min Fair Share Networks. In Proceedings of SIGCOMM, 1996, pages 206-217.

[13] Z. Cao, W. Zegura. Utility Max-Min: An Application-Oriented Bandwidth Allocation Scheme. In Proceedings of INFOCOM, 1999, pages 793-801.

[14] S. Tang, B. Lee, B. He, et al. Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems. In Proceedings of ICS, 2014, pages 251-260.

[15] A. Ghodsi, M. Zaharia, B. Hindman, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In Proceedings of NSDI, 2011, pages 323-336.

[16] Redis. http://redis.io.

[17] HDFS. http://hadoop.apache.org/hdfs.

[18] Hadoop. http://hadoop.apache.org.

[19] Spark. http://spark.apache.org.

[20] Memcached. http://danga.com/memcached.