Towards enabling cooperation between scheduler and storage layer to improve job performance

Mayank Pundir
University of Illinois at Urbana-Champaign
pundir2@illinois.edu

John Bellessa
University of Illinois at Urbana-Champaign
belless1@illinois.edu

Shadi A. Noghabi
University of Illinois at Urbana-Champaign
abdolla2@illinois.edu

Cristina L. Abad
University of Illinois at Urbana-Champaign
cabad@illinois.edu

Roy H. Campbell
University of Illinois at Urbana-Champaign
rhc@illinois.edu

ABSTRACT

In distributed, data-intensive computing platforms, providing fast access to data is critical. A way to do this is to maximize data locality. To improve data locality, previous research has proposed replicating and caching hot data across the nodes. However, these schemes cannot speed-up the first access to a file, since at its first access, a file has not become "hot".

We present Scheduled Caching, a technique that leverages information available to the job scheduler to improve caching. The job scheduler provides hints about the access patterns of files to the storage layer, which can then pre-fetch and cache input data right before it needs to be processed. Our initial experiments show that this technique can improve job performance by as much as 2.3x, and increase completion time by only 2.3% over the ideal (calculated when all input data is originally in memory).

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

Keywords

Hadoop, HDFS, Caching, Memory Locality, Scheduler, Datacenter

1. INTRODUCTION

Frameworks like Giraph, Pig, and Hive are commonly used in data-intensive applications. Because these frameworks are data-intensive, their efficiency is bottlenecked, to a major extent, by either disk I/O time or network transfer time. To this end, several efforts have been made by the research community to increase disk locality of the jobs running on these models. However, with the emergence of high-throughput datacenter network designs, such as full bisection network topologies, achieving disk locality is not enough and a goal of memory locality is frequently sought [1] (e.g., by caching data intelligently).

In this paper, we present *Scheduled Caching*, a technique that helps provide scheduled memory locality, i.e. memory locality at just the right time. Scheduled Caching enables collaboration between the scheduler layer and the storage layer to provide this timeliness. The hints provided by the scheduler to the storage layer facilitate pre-fetching the required files to memory right before the tasks need them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PDSW '13 Denver, CO USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

2. SCHEDULED CACHING

To exploit the use of memory locality, our goal is to enable coordination between the scheduler and the file system layer. Before running a task, the scheduler knows about the tasks that are a part of the next job, their execution location (e.g. node X in Hadoop) and the files required by these tasks. Schedulers, like those found in Pig and Hive, build an execution plan, in the form of a directed acyclic graph of jobs. Scheduled Caching takes this information and then sends hints to the file system. Specifically, we propose using two kinds of hints – but this can be generalized to any information that the scheduler is able to pass onto the file system:

1. Pre-fetch: Scheduler hints the file system to bring the files that are going to be used next into memory.
2. Do not evict: Scheduler hints the file system to not persist on disk those files which are going to be used again soon.

These hints assist the decision of which new files should be placed in the memory, as well as the removal of files that have a low probability of being accessed in the near future.

Whenever the scheduler determines which files will be required by the next task, it issues an API call to the metadata server (MDS) of the storage system (i.e., the namenode in HDFS) giving the details of the file and the task location. Next, the MDS issues the fetch command to the appropriate

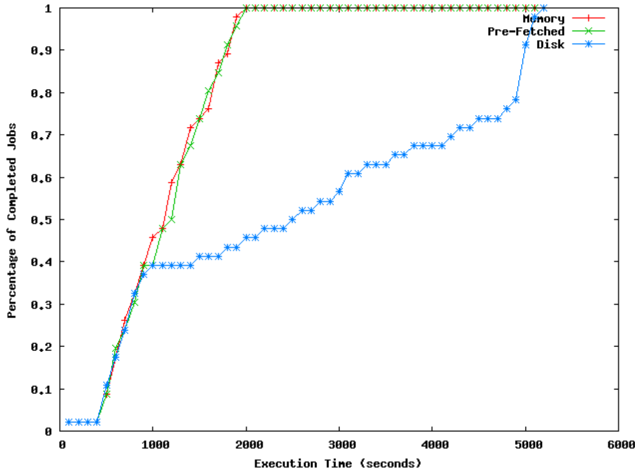

Figure 1: CDF of job completion times.

storage node (i.e., datanode in HDFS). The storage node, upon receiving the fetch request, fetches the file.

If there is not enough room in memory to accommodate the file to be fetched, additional space will be freed up by evicting some other files. To this end, we propose using an aging algorithm that increases the age of the file – based on metrics like popularity, size, last-accessed time and so on, according to a particular workload's characteristics – and removes the file with the highest age.

Finally, the scheduler issues the job execute command to actually run its next task with the files required by the task already in memory.

3. PROOF OF CONCEPT

To demonstrate the performance improvement that can be expected from Scheduled Caching, we have built a proof-of-concept prototype. The prototype is built for Hadoop scheduler and HDFS.

3.1 Implementation

For the purposes of our proof-of-concept, we have adopted a simplified version of the architecture. We use two layers of HDFS: one layer completely on disk, and another layer in memory. Memory HDFS stores its files in a directory mounted with tmpfs. Given such a design, we can simplify the problem to pre-fetching the required file from disk HDFS to memory HDFS. This setup guarantees memory presence. However, memory HDFS takes care of providing memory locality using its default locality technique.

We have extended the FileSystem API provided by HDFS to Hadoop by implementing two fetch calls. The first fetch call fetches a specified file to the destination file system. The second fetch call fetches the specified directory to destination file system. These two functionalities allow us to modify the Hadoop scheduler to pass the pre-fetch hints to the file system layer.

3.2 Evaluation

We ran our preliminary tests on a cluster with a master and 10 other slave nodes. tmpfs has been mounted with exactly half of main memory on each node i.e. the master has 5 GB in its tmpfs folder and slaves have 2.5 GB each in their tmpfs folders.

We use the SWIM benchmark [3] to evaluate our prototype. We have used 46 jobs from the benchmark with varying levels of map-heavy, shuffle-heavy and reduce-heavy jobs. The SWIM workload had to be scaled to an 11 node cluster (by doubling the input) and the same scaled up version has been used for all experiments.

Figure 1 provides a comparison of the execution times of the SWIM benchmark when performed on three setups: unmodified, on-disk HDFS; unmodified, in-memory HDFS (using tmpfs); and HDFS with pre-fetching. As shown in Figure 1, both the in-memory and pre-fetching versions complete 100% of the jobs at around 2000 seconds, while the disk version has completed less than half. With pre-fetching, we observe an average performance improvement of about 2.3x over disk, while incurring a penalty of only 2.3% over the in-memory version (average completion time of 1130.69 seconds, as compared with 1105.37 seconds).

From the figure, we also observe that roughly 40% of the tasks do not see any improvements. We hypothesize that the amount of data consumed by these tasks is relatively small and, thus, the disk I/O time is dominated by the actual computation time.

4. RELATED WORK

Recent work [1] has shown that disk locality is becoming irrelevant, partially due to the introduction of full-bisection network topologies, and that going forward, memory locality will be an increasingly more significant metric. PACMan [2] tries to ensure that all files required by a task are cached in memory so that the performance improvement of memory locality is not decreased by straggling tasks; however, it cannot improve the memory locality of a task that operates on a file that has not been processed earlier. RAMCloud [4] proposes keeping all the data in memory; however, this solution is expensive and not always feasible. Spark [5] uses the iterative nature of tasks to cache intermediate output data in memory and use the cached data for subsequent iterations; pre-fetching is not considered.

5. CONCLUSION AND FUTURE WORK

In this paper, we discussed the increasing importance of memory locality over disk locality. To this end, we present Scheduled Caching. The primary contribution of Scheduled Caching is an architecture that enables task schedulers – such as those found in Hadoop and systems built on top of Hadoop – to provide hints to underlying storage layers about future file access patterns. We then described our prototype implementation which is built on top of a dual-HDFS system. We are able to improve job performance by as much as 2.3x, and within 2.3% over the ideal (calculated when all input data is originally in memory).

Given our promising results, we have considered several possible significant extensions for future research. The most immediate is to extend the prototype to higher layers like Pig and Hive. One direction for future work is using RDMA for improving access to data across the network when memory locality is not achieved. Another potential direction is to enable MapReduce tasks to begin reading data even as it is still being loaded from disk. Finally, we have considered designing a memory-aware scheduler that schedules tasks whose input data are already in memory before the others.

6. REFERENCES

[1] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 12–12. USENIX Association, 2011.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

[3] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 390–399. IEEE, 2011.

[4] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.