Caching in the Multiverse

Mania Abdi[⋆], *Amin Mosayyebzadeh*[⋄], *Mohammad Hossein Hajkazemi*[⋆]
Ata Turk[‡], *Orran Krieger*[⋄], *Peter Desnoyers*[⋆]
[⋆]Northeastern University, [⋄]Boston University, [‡]*State Street*

Abstract

To get good performance for data stored in Object storage services like S3, data analysis clusters need to cache data locally. Recently these caches have started taking into account higher-level information from analysis framework, allowing prefetching based on predictions of future data accesses. There is, however, a broader opportunity; rather than using this information to predict one future, we can use it to select a future that is best for caching. This paper provides preliminary evidence that we can exploit the directed acyclic graph (DAG) of inter-task dependencies used by data-parallel frameworks such as Spark, PIG and Hive to improve application performance, by optimizing caching for the critical path through the DAG for the application. We present experimental results for PIG running TPC-H queries, showing completion time improvements of up to 23% vs our implementation of MRD, a state-of-the-art DAG-based prefetching system, and improvements of up to 2.5x vs LRU caching. We then discuss the broader opportunity for building a system based on this opportunity.

1 Introduction

Modern data analytics platforms (e.g. Spark [37], PIG [20], or Hive [27]) are often coupled with external data storage on services such as Amazon S3 [3] and Azure Data Lake Store [17], resulting in storage bottlenecks [4, 15, 18, 23].

Multiple caching solutions have been developed to address this storage bottleneck, e.g. Solutions such as Pacman [5], Tachyon/Alluxio [2, 16] and Apache:Ignite [6] allow datasets to be cached within the local cluster. However given finite cache, and often even more limited bandwidth for fetching data into the cache, the performance of this cache depends on its caching policy, and recent studies show that traditional caching policies (e.g. LRU) for this workload perform poorly relative to task-specific ones [5, 9, 15, 21].

Higher-level analysis frameworks such as PIG [20], Hive [27] and SPARK [37] compile user programs into an execution plan consisting of multiple, for example, MapReduce [11], or Tez [24] jobs, and a directed acyclic graph (DAG) of dependencies between these jobs. Jobs are then scheduled in parallel, within the constraints set by these dependencies. Jobs can take minutes to even hours [9], resulting in execution plans which identify data accesses far into the future. Exploiting this knowledge of future access patterns results in significant improvements in caching performance vs. LRU and other history-based algorithms, as shown by works such as MemTune [33], LRC [36] and [15].

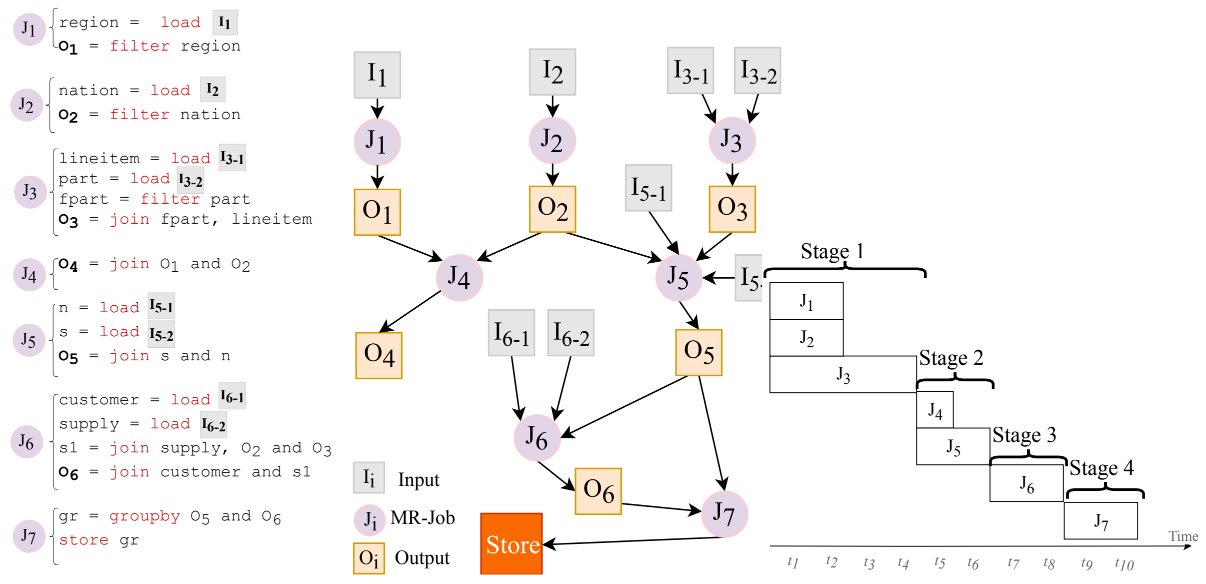
These existing efforts use application DAG information to *predict* future data accesses, and then prefetch data into the cache and manage the cache contents based on those predictions. In doing so, they are not taking advantage of a fundamental opportunity. Rather than caching data given a prediction of task execution, can we exploit the information provided by the DAG to influence the order of task execution to enable more effective caching? That is, rather than managing/prefetching the cache based on one prediction of the future universe, can we select a universe for which caching will be more effective?

This paper provides preliminary evidence that the answer is yes. In a simple, semi automated experiment, we show that by caching can be used to optimize the critical path through the DAG, and present experimental results showing completion time improvements for TPC-H queries of as much as 2.5x over LRU and 23% over MRD [22], the state-of-the-art DAG-based approach (and in all cases no worse than MRD).

We next provide more background on the opportunity, present our initial evidence, and then discuss the research challenges and effort to exploit this opportunity in a systematic way.

2 Background and Motivation

To explain DAG-guided caching, we consider the PIG workflow management framework, which compiles the user's query into a directed acyclic graph (DAG) of MapReduce [11], Spark [37] or Tez [24] jobs. In Figure 1a, we see TPC-H



```
J1 { region =  load I1
     O1 = filter region

J2 { nation = load I2
     O2 = filter nation

J3 { lineitem = load I3-1
     part = load I3-2
     fpart = filter part
     O3 = join fpart, lineitem

J4 { O4 = join O1 and O2

J5 { n = load I5-1
     s = load I5-2
     O5 = join s and n

J6 { customer = load I6-1
     supply = load I6-2
     s1 = join supply, O2 and O3
     O6 = join customer and s1

J7 { gr = groupby O5 and O6
     store gr
```

(a) PIG Latin script (b) DAG of MapReduce jobs (c) Scheduled jobs

Figure 1: Execution of Query #8 from TPC-H benchmark using Pig framework. In (a) we see the query script with jobs and inputs identified in purple circles and grey squares, respectively. (b) shows the directed acyclic graph (DAG) produced by the Pig compiler, with the corresponding jobs and inputs; (c) is the Pig schedule of the DAG to the MapReduce framework, with jobs executed in breadth-first stages.

Query 8 in PIG Latin [25], which is compiled by PIG into the execution plan DAG in Figure 1b. Tasks are then sorted by dependency into *stages*, and submitted for execution resulting in a timeline such as is seen in Figure 1c.

In this environment, caching policy can use not only information from previous requests, but knowledge of future requests derived from this execution plan[1], including i) the dependency graph, ii) job type, and iii) job input datasets and sizes. In addition, to predict the execution timeline, we need to iv) predict individual job execution times, which may be done with data from past executions of the same job type (sort/join/etc. in PIG; application executable in some systems), and v) network and storage system bandwidth, which may be known or can be measured.

In Table 1 we see the jobs created by the PIG compiler ($J_1, J_2, .., J_7$) as well as the inputs to each job ($I_1, I_2, ..., I_{6-2}$) with their respective sizes. Runtime for each job (in arbitrary units) is predicted without cached input ("baseline runtime") as well as for the case where each input dataset is cached (expressed as runtime improvement over baseline). For this example we assume that speedups are additive, and that they are all-or-nothing; i.e. if one block of input for J_2 is fetched (I_2 is 2 units in size), there is no speedup, while it completes in 1 time unit if both units of I_2 are fetched. (We take this simplifying assumption from Pacman [5])

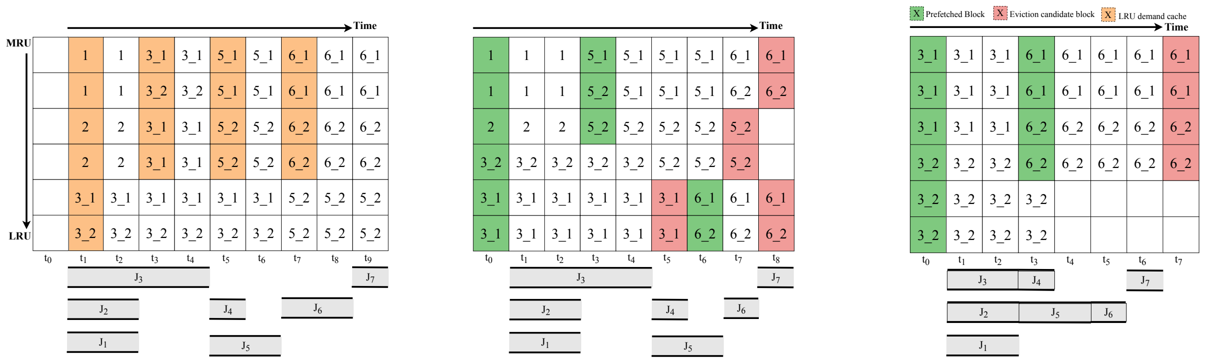
Figure 2(a) shows the execution plan without prefetching and with LRU cache management; completion time is 9 units as there is no input data re-use, and thus all inputs are read from remote storage (purple); at the bottom of the figure we see a timeline of jobs running at each unit of time.

MRD uses prior runtime information to predict the order and timing of data requests, prefetching data (green blocks) and evicting other data (orange) to improve performance. Prefetching allows job J_6 to run faster, resulting in a completion time of 8 units rather than 9.

At each point in time, MRD fetches the dataset which will be requested the soonest in the future; ties are broken arbitrarily. Stage 1 requires 10 units of input, but we have 6 units of cache; we show inputs for J_1 and J_2 being prefetched, but only part of the two inputs to J_3, I_{3-1}, so stage 1 ends at t_4 as before. Prefetching of inputs to J_5 begins at t_3, when the inputs become more valuable than data already in the cache; however input I_{5-2} cannot be completely loaded as inputs to J_3, which is still running, are occupying 3 units of cache, and prefetching I_{5-1} gives a speedup of 0 to J_5. Finally at t_6 we prefetch I_{6-1} but not I_{6-2}, giving a speedup of 1 to job J_6 and completing the entire workflow one unit sooner.

By taking speedup information into account, we can do much better as shown in Figure 2c(c). For each stage of job execution, we prefetch the set of inputs (subject to available cache space) which will result in the largest decrease in overall execution time, or nothing if no decrease is possible. Thus inputs to J_1 and J_2 are ignored, as they cannot cause stage 1 to complete in less than 2 time units. However the inputs to

[1] We restrict our discussion to PIG; however the same approach may be used with Spark and other systems which expose internal dependencies.



(a) LRU (b) MRD (c) NOC

Figure 2: Different cache managements decisions over Query #8 from TPC-H benchmark. Job J_i takes input datasets j (or i_j, where j is the j^{th} input for job i). LRU completes in t_9, MRD completes in t_8, Near Optimum Cache (NOC) completes in t_6

Job	Start	Baseline Runtime	Inputs	Size	Runtime Improvement
J_1	1	2	I_1	2	1
J_2	1	2	I_2	2	1
J_3	1	4	I_{3-1} I_{3-2}	3 3	1 1
J_4	5	1	{}		
J_5	5	2	I_{5-1} I_{5-2}	2 2	0 1
J_6	7	2	I_{6-1} I_{6-2}	2 2	1 0
J_7	9	1	{}		

Table 1: Job characteristics and predicted runtime improvement Query #8.

job J_3 are fetched in their entirety, giving a 2-unit speedup and finishing the job (and thus the stage) by t_2. There is not enough cache space to prefetch for J_5, so none of its inputs are prefetched; as a result there is sufficient room to fetch all inputs to J_6 at the beginning of stage 2 (t_3), giving a 1-unit speedup and completing the workflow in 6 units of time.

Given the above information and assumptions, we can (a) determine the feasibility of any prefetching/eviction schedule (e.g. does it fit in cache) and (b) estimate its completion time. From this, in turn, we can (in theory) determine the optimal prefetching schedule. In theory this problem is no doubt NP-complete; however for practical job schedules (especially with PIG stage-based scheduling) it is likely that good approximations may be found.

This approach leads to our suggested caching/pre-fetching strategy, which we call Near-Optimal Caching (NOC). NOC pulls DAG information along with job type (e.g. filter, sort, etc.) and input data sets for each job with the DAG and extracts their size from the storage. In addition, we estimate the job performance under two scenario: (1) when data is in the cache, (2) when data is not in the cache. We assume a staged execution model such as used by PIG, where all jobs complete before a stage ends. We use the scheduling information of the Pig execution framework and for each stage, we select the input datasets that could decrease the stage runtime the most by being in cache.

Based on prior measurements of external storage throughput, for each to-be-prefetched dataset, we calculate its read latency from backend. This allows us to begin prefetching a dataset in time to have it in the cache when it is required.

3 Evaluation

We put together a simple experimental testbed to get a feeling for the performance gains that could be obtained with NOC. The experimental environment included a four node compute cluster and a four node storage cluster. The compute cluster nodes have 2x Intel Xeon CPU E5-2660 2.20GHz (16 total cores, 32 threads), 128GB RAM and 2x 10GbE NICs and the storage cluster nodes have 2x Intel Xeon E5-2660 CPU (28 total cores, 56 threads), 256GB RAM, 12 x 2 TB 7.2K SATA Seagate HDDs and 2x10GbE NICs. On the storage cluster we deployed CEPH [32] 12.2.7. On the compute nodes we deployed PIG [20] 0.17.0 on top of Hadoop [11] 2.8.4 and Alluxio [2] 1.8.0 as an in-memory cache layer. We dedicated 6GB of memory on each node to Alluxio for a total of 24GB of cache. We restricted Hadoop to 50GB of memory on each node.

For evaluation, we used a subset of the *TPC-H* [28] benchmark transformed into PIG Latin and run on a dataset of 32GB. We ran queries sequentially based on their order and before each query we clear the cache. Before the experiment, we

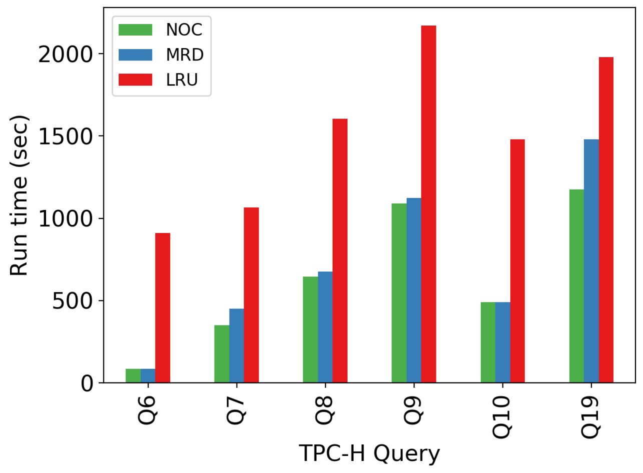


Figure 3: Effect of NOC data prefetching on query execution end-to-end latency.

executed all queries with and without caching and recorded the time of execution for each job within the query. In other words, we ran query and gather statistics when query's input is in the cache and when it is out of the cache.

Also, we modified Pig to output the execution plan (DAG of MapReduce jobs) before launching the query. Following the procedure explained in section 2, we created a I/O plan for which dataset should be prefetched/cached and when the prefetching request should be scheduled. We also modified Pig to provide timing and staging information as the query is executed by the framework[2].

We processed the execution plan to create scripts that would run in parallel to the execution of the query. These scripts issue `load` requests to Alluxio to prefetch data and `free` requests to evict data from the cache. A separate script was generated that emulated the NOC and MRD algorithms. We compare the performance of these algorithms to LRU, implemented natively by Alluxio. The results of this experiment are shown in Figure 3. With this simple experiment, we see that NOC has substantial gains over LRU, achieves at least as good performance as our emulation of MRD and in the best case (i.e., Q19) NOC outperforms MRD by 22.8%. In the queries where NOC and MRD have identical performance, they choose the same datasets to evict and prefetch, while in other cases NOC chooses datasets based on future stages as shown in Figure 2 for query 9.

This experiment suggests that a NOC implementation may well offer significant performance advantages. These results are, we believe, quite pessimistic. NOC will offer significantly more advantages in an environment where the bandwidth to storage is limited. The scheduling by Pig into stages, where jobs whose dependencies are met are not scheduled until all jobs in the previous stage have completed, greatly limits our opportunity to optimize in the current experimental environment. We also believe the degrees of freedom available with NOC will be more valuable when caching intermediate data sets (in these experiments we only cached input data sets) and when multiple tasks are being executed concurrently on the cluster and the cache resources are being shared.

[2]This required a 300 line patch to the PIG framework.

4 Challenges and Future Work

What are the challenges in applying near-optimal caching to realistic systems, and what are the longer-term questions in pursuing this approach?

Runtime and speedup estimation: how can we estimate the speedup due to prefetching an input data set? Inferring speedup is a straightforward extension of the methods (e.g. regression) already in use to estimate job runtime based on job type and input data sets. More information may in fact be available from this data than can be used by current algorithms: e.g. if jobs were found to have partial speedups with partial data (instead of all-or-nothing), or if speedups for multiple inputs were not additive. It is an open research question as to whether we can actually schedule based on these more realistic models, however. Estimating job runtime with different levels of accuracy has been studied extensively [10, 14, 30]. Future work will compute the confidence interval to limit the consequences of inaccurate prediction. If insufficient information is available to predict the runtime accurately, the confidence interval will be wide, the strategy degrades to be the same as MRD.

Data Locality: Execution engines favor data locality when allocating resources for job execution; cache management decisions (i.e. where to cache a prefetched data item) thus impact scheduler job placement. Is this interaction a problem, and if so how can the execution scheduler and cache manager cooperate?

Running Concurrent Queries: Workflow management frameworks run multiple queries to reach higher resource utilization. Considering one query at a time does not necessarily lead to an optimum decision or even might degrade the performance of the other queries. A cache management needs to take into account other jobs from other queries and decide which datasets should be fetched to improve overall performance of the system (even though some queries' performance may deteriorate). Since query runtimes can be predicted by NOC, we propose using this information to perform shortest-job-first (SJF) scheduling across multiple queries [31].

Query priorities: Many analysis systems handle both interactive and batch queries. Prefetching for interactive queries is difficult, yet even interactive use often has many repeated patterns, leading to the question of whether we can predict and prefetch for future queries, thus improving interactive performance.

Writing to Cache: In the discussion above, all data sets have been assumed to be inputs. Writeback caching violates this assumption: intermediate results are kept in cache [2, 16] before being written back to external storage. The size of these results is not known, but must be estimated based on job parameters, and write data must be prioritized in the cache until it can be written back.

Other platforms: It should be possible to apply this strategy to other DAG-based frameworks, such as HIVE, Spark, etc.

Other job types: Some fraction of jobs (20% in the Alibaba [1] traces) will not use frameworks with DAG information that can be used for prefetching. How can we handle those jobs and the cache they use, without either starving them or the DAG-based jobs? Can we infer dependencies and cache size utilizations for them?

5 Related work

Cache management policies and prefetching has been studied extensively in different areas of computer system design [7, 12, 13, 19, 26, 29]. Cache management policies for data analytics have been studied extensively. Traditionally, such work [15, 34] exploits job execution history to speculate dataset future access pattern. However, there is also a small number of recent work that attempts to use future information extracted from higher level framework [8, 22, 35, 36] for dataset caching and prefetching.

Mithril [34] proposes a prefetching layer that applies data mining algorithms on block access history to speculate what to prefetch at time. Netco [15] takes advantage of dataset access history to prefetch the best candidate based on available cache size and network bandwidth to meet deadline SLOs such as maximizing the number of finished jobs. Our approach is different from the enumerated methods as it relies on the future information.

Among work relying on future information, both SADP [8] and MRD [22] rely on information from higher-level framework to find and prefetch the nearest dataset used in future. LRC [36] prioritizes datasets to cache which has the most number of dependent jobs and evict datasets with least number of dependencies. Although all of them take advantage of the future information, they only consider a single dimension i.e., job dependency; however we take into account other dimensions e.g., dataset contribution in total runtime.

6 Conclusion

Locally caching data sets is critical for compute clusters that use storage from object storage services. Researchers have started using scheduling information (i.e. job DAGs) from existing analytic environments to manage these caches. The fundamental insight of this work is that, rather than prefetching based on a prediction of job execution based on this information, we can use this information to influence the execution order in order to enable more effective caching. We developed a simple experimental environment that used DAGs extracted from PIG to prefetch data along the critical path of execution through the DAG, and evicted cached data that was no longer required. Even with this very simple experimental environment we obtained up to 22.8% improved performance, providing strong preliminary evidence of the power of this approach.

7 Discussion Topics

- What are the other analytic systems that we can use? What changes we need to apply to popular systems such as SPARK to take advantage of NOC?
- How feasbile is it to apply the same idea to the frameworks such as Hive?
- What are the other workflow management frameworks that this idea can be applied to?
- Whe consider the network bandwidth to the backend storage is unlimited, however, in practice the network bandwidth resource is limited. What changes we need to make to deal with this limitation?
- How can we generate a more realistic workloads (e.g., with concurrent queries) to emphasize the propose idea?

Acknowledgment

We would like to acknowledge the feedback of the anonymous reviewers and our shepherd, Yu Hua, as well as contributions by Larry Rudolph. This work was supported in part by the Mass Open Cloud (massopen.cloud) and its industrial partners.

References

[1] Alibaba Cloud Traces". `https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018`.

[2] Alluxio - Open Source Memory Speed Virtual Distributed Storage. `https://www.alluxio.org`.

[3] Inc. Amazon Web Services. Amazon Simple Storage Service (S3) — Cloud Storage — AWS. available at aws.amazon.com/s3/.

[4] Amazon EMR. https://aws.amazon.com/emr/.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX.

[6] Apache Ignite. https://ignite.apache.org/.

[7] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[8] C. Chen, T. Hsia, Y. Huang, and S. Kuo. Scheduling-Aware Data Prefetching for Data Processing Services in Cloud. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 835–842, March 2017.

[9] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, August 2012.

[10] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. CAST: Tiering Storage for Data Analytics in the Cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 45–56, New York, NY, USA, 2015. ACM.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[12] H. Falahati, M. Abdi, A. Baniasadi, and S. Hessabi. ISP: Using idle SMs in hardware-based prefetching. In *The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013)*, pages 3–8, Oct 2013.

[13] Falahati H., Hessabi S., Abdi M., and Baniasadi A. Power-efficient prefetching on GPGPUs. *The Journal of Supercomputing*, 71(8):2808–2829, Aug 2015.

[14] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the Tenant-provider Gap in Cloud Services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 10:1–10:14, New York, NY, USA, 2012. ACM.

[15] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph Seffi Naor, and Sriram Rao. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 186–198, New York, NY, USA, 2018. ACM.

[16] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[17] Microsoft Datalake. http://azure.microsoft.com/en-us/solutions/data-lake.

[18] Microsoft Azure HDInsight. https://azure.microsoft.com/en-us/services/hdinsight/.

[19] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[20] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[21] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[22] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. Reference-distance Eviction and Prefetching for Cache Management in Spark. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 88:1–88:10, New York, NY, USA, 2018. ACM.

[23] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard

Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, New York, NY, USA, 2017. ACM.

[24] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[25] Savvas Savvides. TPCH-PIG. https://github.com/ssavvides/tpch-pig.

[26] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, Berkeley, CA, USA, 2015. USENIX Association.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005, March 2010.

[28] TPCH benchmark. http://www.tpc.org/tpch/.

[29] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[30] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016. USENIX Association.

[31] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. CLARINET: WAN-Aware Optimization for Analytics Queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 435–450, Savannah, GA, 2016. USENIX Association.

[32] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[33] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 383–392, May 2016.

[34] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: Mining Sporadic Associations for Cache Prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 66–79, New York, NY, USA, 2017. ACM.

[35] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. LERC: Coordinated Cache Management for Data-Parallel Systems. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, Dec 2017.

[36] Y. Yu, W. Wang, J. Zhang, and K. Ben Letaief. LRC: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, 2010.