A Community Cache with Complete Information

Mania Abdi[⋆], *Amin Mosayyebzadeh*[◇], *Mohammad Hossein Hajkazemi*[⋆], *Emine Ugur Kaynar*[◇], *Ata Turk*[‡], *Larry Rudolph*[†], *Orran Krieger*[◇], *Peter Desnoyers*[⋆]

[‡]*State Street,* [†]*TwoSigma,* [◇]Boston University, [⋆]Northeastern University

Abstract

Kariz is a new architecture for caching data from datalakes accessed, potentially concurrently, by multiple analytic platforms. It integrates rich information from analytics platforms with global knowledge about demand and resource availability to enable sophisticated cache management and prefetching strategies that, for example, combine historical run time information with job dependency graphs (DAGs), information about the cache state and sharing across compute clusters. Our prototype supports multiple analytic frameworks (Pig/Hadoop and Spark), and we show that the required changes are modest. We have implemented three algorithms in Kariz for optimizing the caching of individual queries (one from the literature, and two novel to our platform) and three policies for optimizing across queries from, potentially, multiple different clusters. With an algorithm that fully exploits the rich information available from Kariz, we demonstrate major speedups (as much as 3×) for TPC-H and TPC-DS.

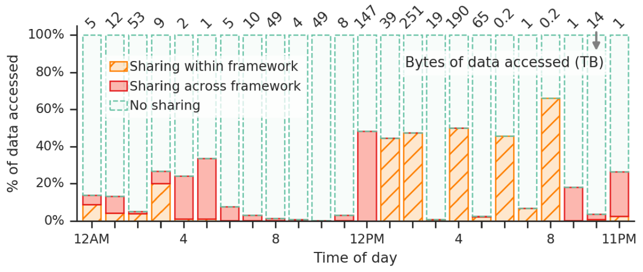


Figure 1: *The value at the top shows the total amount of data accesses in each hour. Green is the proportion of the accesses to data that was not accessed previously in the hour. Orange is the accesses that go to data previously accessed in the same hour by the same analytic framework. Red is accesses to data that was last accessed by another framework in that hour. This analysis suggests that caching can be effective both to capture repeated access to the same data from a framework, and access by different frameworks to the same data.*

1 Introduction

Large-scale data-flow oriented analytic frameworks, such as Spark [72], Hive [62], and Pig [56], are broadly used in many public and private cloud environments. Today, cloud deployments commonly use centralized "data lakes" [3, 9, 10, 42, 58] such as Amazon S3 [4], Azure Data Lake Store [11], and Ceph [67] that are used by all the frameworks running in the cloud. Although such dis-aggregation of storage offers many benefits, it also carries major performance costs [61].

Caching and prefetching, which move frequently-used datasets close to the analytic frameworks, are standard techniques for improving performance [20, 50]. Data-flow oriented analytical frameworks share a number of features that provide the opportunity to explore caching strategies that differ from prior work on CPU, page-based, and variable-sized (e.g. web) caching:

- they expose the input objects and inter-job dependency with Directed Acyclic Graphs (data-flow DAGs), where complex DAGs providing a detailed view into future I/O behavior;
- units of data access and computation are large, taking many seconds to access or run, allowing complex strategies not feasible in many other caching domains; and
- recurring jobs, where the same code runs on different data, are common [19, 24, 48], allowing accurate prediction of execution timing and characteristics [21, 40, 46, 66],

To illustrate these features, we consider confidential traces shared with us by an industrial partner recording 4 months of usage from a mid-sized (>100 nodes) cluster in production use which is running Hive/Hadoop, Spark, native MapReduce, and streaming jobs. Although the majority of jobs were small (1-4 node DAGs), 90% of input data was read by complex DAG-based jobs with 5 nodes or more, with some DAGs involving over 50 nodes. Individual DAG stages averaged 5 minutes with some stages taking as 6 hours. Over 90% of the jobs seen over the four months were jobs that repeated many times, and more than 90% of object reads were to objects repeatedly read. Clearly, there is an enormous opportunity to optimize performance for repeating I/O intensive jobs that provide complete visibility in future accesses and where minutes are available to compute strategies. In fact, a number of groups have started exploiting these characteristics to develop more sophisticated caching and prefetching strategies within analytics clusters [17, 20, 37, 44, 57, 69, 71].

Kariz is a platform designed to enable different strategies for caching and prefetching at the storage level. It collects DAG information from analytic platforms and the execution time that stages of the DAG take to execute. It also maintains global information about what data is cached and resource availability (e.g. storage bandwidth). This information is used by Kariz to accurately (§3.3) predict stage run-time. We have implemented a number of strategies, including the previously published strategy MRD [57] (Most Reference Distance) that exploit DAGs to maximize cache hit rate, and two new strategies, we call CP (Critical Path) and CMR (Cache for Minimizing Runtime). CP exploits the stage run-time prediction to cache data in order to reduce the critical path of stages in the DAGs. CMR relies on support in Kariz for partial file caching, prefetch partial data from each stage of the DAG to improve performance beyond a single critical path.

The focus in Kariz for optimizing the critical path is not primarily to improve the latency. Analytic platforms like Spark typically reserve resources until the entire DAG has completed. By optimizing the critical path, the resources reserved for a DAG can be freed earlier, improving throughput; potentially huge savings for complicated DAGs where the critical path takes much longer than most of the work.

A fundamental difference between Kariz and previous work [57,69,71,72] is that Kariz implements a *community cache*, where multiple clusters, potentially running different analytic platforms, can concurrently share a single instance of Kariz. A community cache is based on the idea that the near past of a community of clusters data accesses may be a good predictor of the near future data access pattern of new jobs from a cluster.

For example, within some financial services companies (e.g. the employers of two of the authors) different groups often create individual clusters with varied frameworks accessing the same shared data-lake, with reasons for this fragmentation including regulatory issues, security, organizational structure, or preference. When new datasets arrive (e.g. recent market data), many jobs are submitted independently by members of different groups, resulting in heavy access across most or all frameworks and clusters. Anecdotal reports (e.g. as described by the authors of Quiver [48]) indicate similar behavior by data scientists, who often use company-wide datasets for joins or training ML models. Finally, we see similar behavior in the confidential traces, e.g. in Figure 1 at 3AM the 692 jobs are distributed across frameworks (11% Spark, 20% Oozie, 16% MapReduce, 47% Hive/MapReduce); 36% of Spark jobs share objects (42% of total data) with Hive, while tables accessed by Oozie were a subset of Hive accesses.

In contrast to previous works [57,69,71], that focused on optimizing individual queries, Kariz also supports strategies for optimizing multiple concurrent queries coming from potentially different analytic platforms to the community cache. We have implemented two such strategies: 1) Shortest Job First (SJF) that focuses cache resources to accelerate the fastest predicted query to free resources as quickly as possible. 2) Cache for Minimizing Runtime of multiple DAGs (CMR-M), that additionally takes into account storage bandwidth and the sharing of data across multiple queries in selecting data to be cached and prefetched.

We have adapted two analytic platforms to exploit Kariz (PIG/Hadoop and SPARK) and found that while significant effort was required to understand the platform, the end changes were less than 100 LOC in each platform.

We experimentally evaluated our system on a 16-node bare-metal cluster. We use the characteristics of the confidential trace (including query submission rate, DAG structure, data accessed, data reuse) to run a mix of synthetic workloads from TPCH and TPC-DS. We demonstrate experimentally that the new algorithms enabled by the rich information collected by Kariz (and its support for partial caching) result in major performance advantages with our synthetic workload on both Pig/Hadoop (mean across all queries of 1.25x and maximum 2x) and Spark (mean 1.8x and 3x). Through simulation, we show that there are significant advantages to a community cache; e.g., an improvement of up to 1.5x over separate per-analytics platform caches.

Key contributions of this work are: 1) demonstrating the value of partial over than full file caching, 2) a high-accuracy run-time prediction based on the amount of cached state and available storage bandwidth, 3) demonstrate that it is possible, and in fact simple, to extract the information needed for optimizing performance from multiple analytics platforms, 4) showing that strategies can be developed and effective that target the direct improvement of predicted run-time by optimizing critical paths rather than implicit characteristics such as hit rate, and finally, 5) a new architecture that demonstrates that a community caching layer for analytics platforms is feasible and offers value.

2 Background and Related Work

We begin by describing the data analytics frameworks targeted by Kariz in further detail, providing an example to motivate our approach, and then briefly survey related work.

DAG-based frameworks: One way in which frameworks such as Spark [72], Hive [62], Impala [47] and Pig [56] differ from traditional large-scale applications is in their use of Directed Acyclic Graphs (DAGs) of operations (vertices) and their input/output dependencies (edges). Before a user query is executed, a query planner parses it, generating an unoptimized logical plan with resolved tables and columns. The optimizer performs optimizations such as predicate pushdown, prunes columns and partitions, and may even remove data from the logical plan. The planner transforms the optimized logical plan to a corresponding physical plan—e.g. the logical operation *TableScan* is transformed to *JsonScan* or *ParquetScan*, and byte ranges within input objects are assigned to each physical operator before the physical execution plan is sent to the execution engine. It is this physical execution plan which Kariz uses for intelligent prefetching and caching.

Definitions: As there are differences in terminology used by each framework, in this paper we use the following terms: A *task*, t, is the smallest unit of computation; tasks are scheduled by the lower-level framework scheduler (e.g. Yarn [65]). A stage, s, is a set of parallel tasks that execute the same code and are submitted for execution simultaneously (e.g. mappers); a higher-level scheduler (e.g. Spark's DAG-scheduler) decides when to submit a stage of tasks to the lower-level framework scheduler. Data-flow oriented frameworks have different terms for a set of stages linked by dependencies; we will follow common usage and use the term *DAG* for them. For each stage there is a set I_s of input objects and their byte ranges for that stage. Finally, some data-flow oriented frameworks (e.g. Pig [56]) divide *DAGs* into *stage-sets*, sets of stages in the DAG without inter-dependencies, which may run in parallel but must complete before the next stage starts.

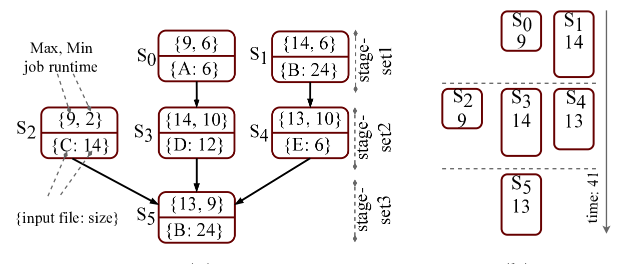


Figure 2: *(a) 6-stage DAG divided into 3 stage-sets. Each stage is labeled with run-time (max=no input in cache, min=all input in cache), input file name and size. (b) Execution schedule with no prefetching or caching for gang-scheduling.*

Disaggregated Storage: These frameworks typically default to using HDFS [61], which stores data locally on compute nodes in a cluster. Yet in modern practice, storage is often disaggregated from the systems performing analysis [42,52,54,58]. Data is typically stored on remote object stores such as S3 [4], Azure Blob Store [23], or on-premises equivalents such as Ceph [8], and accessed directly via e.g. the S3A connector [5]. This enables, for example, elastic analytics clusters [3] and serverless data analytics [59], but with a performance penalty, e.g. in 2018 the NetCo researchers [42] measured object storage speeds of less than 50MB/s per VM (or 4MB/core) on widely-used cloud services.

Motivational example: Kariz is a caching and prefetching system to accelerate computation in these frameworks; we provide a simple example to give intuition about our approach and how it contrasts with previous approaches. Figure 2 shows a DAG made up of 6 stages divided into three stages-sets (sets of stages in the DAG without inter-dependencies, which may run in parallel but must complete before the next stage starts), with the stages executing from the top to bottom. For each stage, the DAG specifies the input files and size; we have labeled each stage with its input file (A/B/...) and size, and run-time with (a) no data is in the cache, and (b) all inputs are cached or prefetched by Kariz. We assume Kariz can predict stage run-time from prior execution times, and that it can perform fine-grained *partial caching* with proportional speedup.

In stage-set 1 (top), we can see that there is no value in prefetching input B for S_0 (uncached duration 9) until we have addressed S_1 (duration 14), which determines the stage completion time. If we cache all of the input to S_1, however (see Figure 2a), reducing its run-time to 6, some of that cache space (and remote bandwidth) will be wasted, as S_0 will now determine stage-set completion time. Kariz instead caches "just enough" of each input to minimize stage-set run-time—e.g. caching 15 units of B will bring S_1 duration down to 9, and additional prefetching will be applied to both S_0 and S_1.

2.1 Related work

We focus on recent work on *informed* (rather than *history-based*) caching and prefetching for analytics frameworks, and omit the vast literature on disk/file [32,33,38,39,60], web [18], and CPU/memory caching [29,30,34,64].

Caching and prefetching rely on the existence of patterns and correlations in real workloads; Jockey [31] and Corral [41] show that the data access patterns of analytics frameworks are highly repetitive and predictable. In Ernest [66] we see that in real world deployments job run-times are predictable as well, based on factors such as input size and jobs DAG structure.

Unlike Kariz, MC^2 [70] and CD-LDS [27] are targeted to caching/prefetching (of files and memory respectively) for general applications, using OS and compiler hints rather than the job DAG available in our more specific scenario. Unlike Kariz which extracts the exact I/O access patterns from the analytic applications, Quiver [48] builds a deep learning specific cache layer and exploits the predictability of accesses in these applications for cache management. MRD [57] makes prefetch and eviction decisions based on graph distance, with the goal of maximizing cache hit rate; accesses at the next stage in the graph are prioritized over those farther in the future. LRC [71] does not prefetch, but makes eviction decisions based on *reference count*, i.e. the number of references to input in stages not yet executed. Dagon [69] ties caching with the DAG scheduler, using the stage scheduling priorities to evict/prefetch data. MemTune [68] manages RAM-based caching in Spark, evicting/prefetching data using only information from the currently runnable tasks.

Alluxio [2] (based on Tachyon [50]) and Apache Ignite [7] are widely used for caching in analytics frameworks; the caching component of Kariz is similar to these, although with extensions for partial caching of objects. A number of replacement and prefetching policies (as opposed to systems) specific to analysis frameworks have been developed, as well: PacMan [20] attempts to minimize run-time by considering MapReduce job wave widths; while NetCo [42] and MRD implement approximations to Belady's MIN algorithm based on predicted job execution order.

Kariz differs from these prior works in several ways: (1) it makes use of *partial caching*, which gives significant gains when the cache size is not large compared to input object sizes, and (2) it explicitly tries to minimize predicted DAG completion time, rather than e.g. cache hit rate.

3 Kariz design

Kariz is a cache management and prefetching system that controls admission/eviction to/from a storage cache, and calculates prefetching schedules for data-flow oriented frameworks. We show the Kariz architecture (§3.1), its partial caching (§3.2), runtime estimation (§3.3) and availability and scalability strategies(§3.4).

3.1 Architecture

As shown in Figure 3, Kariz implements a *community cache*, where multiple clusters, potentially running different analytic platforms, can concurrently share a single cache. Kariz interfaces with the frameworks to obtain data-flow DAGs, execution states, and scheduling events, and collects historical

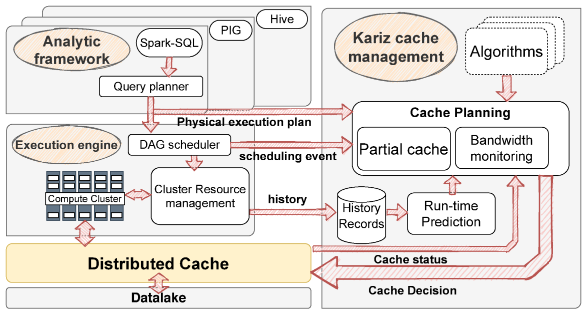


Figure 3: *Kariz architecture. The Kariz components extract DAGS, scheduler and run-time information from the analytic framework. The run-time predictor uses historical run-time information to predict stage execution time. The Kariz cache management algorithms generate caching/prefetching plans for DAGs by exploiting the predicted execution time, partial caching, and data sharing across DAGs. The cache planning makes caching, prefetching, and eviction decisions based on the generated plans to minimize the DAGs runtime while taking account of data sharing for efficiency.*

information (logs) to use in predicting the run-time of future jobs. The cache controller maintains information about what data is cached and estimates bandwidth available to the storage cluster. The run-time prediction component estimates stage completion time based on prior runtimes and current state. The algorithms make up the core of the system, where the task of a single-DAG planner is to come up with a plan for an individual DAG that is then combined into an overall plan by the multi-DAG planner, which issues cache, prefetch, and eviction commands to the cache controller and cache.

Interaction with analytics framework: The framework interface notifies Kariz of DAG submission, providing the physical query execution plan detailing input objects, sizes, formats, operation (e.g. map, filter, join), and parallel task count. Kariz is also notified when a stage-set (Pig) or stage (Spark) begins or finishes execution. In addition, Kariz needs job (DAG) history information (i.e. logs) for prediction; typically this is available through existing framework interfaces.

Storage bandwidth: Prefetching is constrained not only by cache capacity but also by the effective bandwidth from back-end storage, limited by the speed of the network or the storage system itself. Kariz schedules operations to fit within this bandwidth, which is currently configured based on measurements but could be estimated dynamically as well.

Planners: In Kariz, scheduling of cache capacity and storage bandwidth is performed in two stages. The single DAG planner examines individual DAGs and stages to determine *caching candidates*—sets of data from one or more objects which can be prefetched or retained in the cache to speed DAG execution. The multi-DAG planner, in turn, determines which of these caching candidates to put into effect within constraints of cache size and backend bandwidth, prioritizing completion time within a DAG and throughput across DAGs, taking account of data sharing for efficiency.

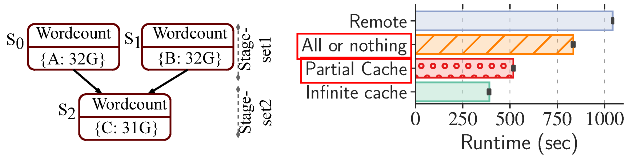


Figure 4: *Physical experiment to illustrate partial caching: DAG with 3 wordcount jobs with 32 GiB inputs and 63 GiB total cache capacity.*

Cache control: The cache controller provides an abstract cache interface to the Kariz planner. It is responsible for tracking currently cached data, and managing the prefetching, retention, and eviction processes; as a distributed cache scales [44] this component will be replicated. Its interface to the cache has methods to prefetch data on a fine-grained basis, "pin" it in cache or release it, and enumerate cache contents (e.g. on startup).

3.2 Partial caching

Hadoop and Spark are sensitive to *stragglers* [49], longer-running tasks which delay completion of a computational stage. Prior prefetching work [20, 42, 57, 71] assumes partial caching of stage inputs will lead to such stragglers, giving no benefit. This will occur when done naively, as some tasks will find their entire input in the cache, while others will fetch their full input from remote storage. We instead assume fine-grained control over prefetching and cache retention/eviction, allowing data to be cached in *strides* much smaller than the input to a single task.

We see the utility of partial caching for real workloads with limited caching in Figure 4. We define an artificial DAG with three wordcount Mapreduce stages, dependent on the other two; each with 32 GiB input. With all-or-nothing caching we can only cache the input to S_2, minimizing stage 2 runtime, but cannot speed up stage 1. With partial caching we still cache the entire input to S_2, yielding the highest runtime reduction per unit of caching, but can distribute the remaining cache across S_0 and S_1, reducing runtime closer to the fully-cached minimum.

Kariz architecture explicity supports column-oriented formats like Parquet [51] and Arrow [6]. It relies on physical query plans to identify object ranges, rather than entire objects; prefetch decisions would then be made within these ranges.

3.3 Run-time prediction

Recent studies from in-production clusters at Microsoft (e.g. Graphene [36], NetCo [42], and others [24, 28, 40, 43]) show that examined jobs are recurring—similar computations are repeatedly executed on different datasets. The same studies show that tasks extensively share common operands, and that most user-defined operations are not custom programs, but widely-used shared libraries (Cloudview Figure-(3) and Figure-4(a & d) [43]). Recent studies, such as Ernest [66], CherryPick [19], and Selecta [45] have shown good accuracy when predicting run-time for such recurring workloads, as a function of input file size, size of cluster and DAG structure [66].

In Kariz, we extend the runtime prediction model proposed by Ernest to incorporate caching and bandwidth differences between cache and remote storage. Similarly to Ernest, we assume that computation time scales linearly with the input size [66] and the communication patterns among stages of a DAG could be represented as sequential, aggregate, and shuffle operations. Unlike Ernest that assumes that builds a runtime prediction model for the entire DAG, Kariz predicts the performance of stages according to the operands executed on that stage and the communication pattern with the previous stage.

We predict stage time T given total input size S, with $f \cdot S$ in cache, bandwidth r_s and r_c to storage bandwidth and cache bandwidth, T tasks (e.g. mappers)[1] and N executors, fitting the following equation:

$$T = \theta_0 + \theta_1 \frac{(1-f)S}{r_s} + \theta_2 \frac{fS}{r_c} + \theta_3 \frac{T}{N} + \theta_4 \log(N) + \theta_5 N \qquad (1)$$

where terms represent fixed startup time (θ_0), data fetch from backend storage and cache (θ_1, θ_2); following Ernest we also incorporate terms for sequential (θ_3), aggregate (θ_4), and shuffle θ_5) cross-stage communication. We use Lasso regression [63] with non-negative coefficients and cross-validation to be resilient to overfitting when training on limited data.

At each scheduling event, Kariz identifies the available backend storage[2] and cache, iterates over the future stages to find longest and "slack" paths. Kariz does this by predicting the runtime of each stage in two cases—data in cache and data needing to be fetched from the remote and uses Bellman-Ford [22] with negative weights to identify the order of the longest paths. We discuss the accuracy of this model further in Section 6.3.

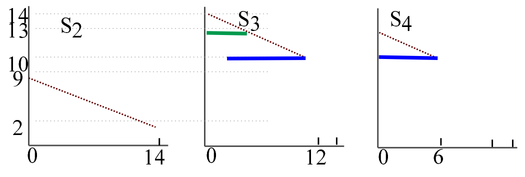
3.4 Availability and Scaling

Kariz takes a simple approach to availability, based on the principle that prefetching and sophisticated cache control are *optional*—DAG-based frameworks and associated caches (e.g. Alluxio) are widely used today with no prefetching or cross-DAG scheduling. If Kariz crashes, the cluster continues to operate, and the caches fall back to LRU after the current commands have completed; on restart Kariz can fetch all needed states (DAG queue, cache contents, execution history) from other components and resume.

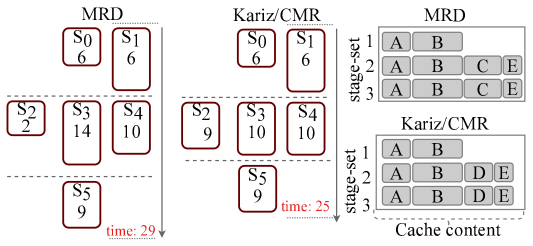
Most computation in Kariz occurs in the cache controller, which is responsible for block-level caching and prefetching commands; this scales by adding additional controllers, each responsible for some set of caches. The central planning algorithm is not currently scalable, and the cluster size which can be controlled by a single Kariz instance is limited by its

[1]This is predicted by the analytic framework during query planning

[2] Currently, Kariz splits bandwidth equally between all running DAGs in a cluster.



(a) *Stage runtime vs. data in cache*



(b) *Dag schedule and Cache status*

Figure 5: *(a) Job runtime vs. data in cache at job start for DAG stage 2 (S_2, S_3, S_4). Green represents caching needed to reduce S_3 runtime to that of S_4; blue is caching to reduce S_3, S_4 to their minimum runtime. Caching for S_2 will never reduce stage runtime, (b) With cache size 50, Kariz/CMR finishes before MRD (25 vs 29) despite a lower hit rate (83% vs 86%)*

speed. In §6.8, we show that current unoptimized performance should scale to a cluster of thousands of nodes.

4 Planners

We implement three planners in Kariz for scheduling caching and prefetching: MRD [57], Critical Path (CP) [17], and our new algorithm, Caching for Minimizing Runtime (CMR).

MRD (Most Reference Distance) is based on topological distance, i.e. the number of DAG stages between two accesses to a file, evicting data with the longest distance until future reference, and prefetching data with the shortest distance. CP, described in an earlier workshop paper, prioritizes prefetching and caching for jobs on the DAG critical path. We refer readers to the respective publications for a more detailed description.

CMR considers the analytic framework DAG scheduling schema and makes full use of the runtime estimation, partial caching, and bandwidth measurement features provided by Kariz. To minimize DAG runtime, it jointly schedules cache space and backend storage bandwidth. When multiple DAGs are active it divides cache space and prefetching opportunities across DAGs using a heuristic that attempts to maximize the throughput—i.e. prioritizing shared data which will speed up multiple DAGs.

4.1 CMR Overview

To explain the intuition behind CMR, we again use the 6-stage DAG from Algorithm 2, scheduled with gang scheduling, examining stage-set 2 (stages 2, 3, and 4) in more detail. In Figure 5a, we assume a graph of runtime vs. amount of input in the cache for these stages. For the sake of simplicity of the discussion, it shows a piece-wise linear function for stage time as a function of prefetched/retained data. In reality, we use runtime prediction (Section 3.3) to predict the required cache size to achieve certain improvement. In Figure 5a, we see S_3 runtime decreases from 14 to 10 as its 12 units of data are cached; S_4 from 13 to 10 with 6 units of caching, and S_2 from 9 to 2 with 14 units of caching.

In making prefetch/retention decisions CMR examines *candidate caching sets*—sets of input objects and their ranges which speed up one or more stages in a stage-set. The first candidate here, shown as a green horizontal bar, represents the fraction of input to S_3 needed to reduce its runtime to that of S_4. The next candidate, shown in blue, corresponds to the remaining input to S_3 and S_4, reducing their runtime to a minimum. Note that the input to S_2 is not part of any candidate set, as S_2 will always complete before S_3 and S_4 and never affect stage-set completion time.

The core of CMR consists of enumerating these candidate sets and pick them in decreasing order of runtime improvement and prefetching or retaining all sets which fit within cache size and bandwidth constraints. We see CMR compared to MRD in Figure 5b, with a cache size of 50. Although MRD achieves a higher hit rate (86%) than CMR (83%), CMR's achieves a lower runtime (25 vs 29) by ignoring stages like S_2 with "slack" in their schedule and focusing on only ones determining stage-set runtimes.

In our simplified example, there is no need to compare caching candidates against each other—once we decide not to cache input to S_2, there is enough cache space for all remaining input. With fewer resources, however, we must choose between e.g. retaining data for one future stage vs. prefetching for a different one.

We do this based on *marginal utility*, i.e. the ratio of completion time saved by caching a candidate set to its size. This is similar to the fractional knapsack problem, i.e. achieving maximum reduction of run-time given a fixed cache capacity, hence the use of cost:benefit in comparisons. This allows comparing candidates across DAG stages, for example, to determine whether cache space and storage system bandwidth in stage-set 1 would be better spent prefetching for stages in stage-set 2 or stage-set 3.

4.2 CMR

The CMR planner runs upon receiving the stage-set scheduling events from the analytic framework, identifying the prefetching and cache pinning/unpinning operations to be executed *during* that stage for execution in *following* stages; prefetching is scheduled so that it will complete by the beginning of the stage in which the data will be used. As with prior work, we assume the existence of a "stage 0" before the DAG begins execution; in practice, this would correspond to the last stage of the previously-executed DAG. We describe CMR operation in the case of a single DAG in two parts: enumeration of caching candidates, in Algorithm 1, and candidate selection and execution, in Algorithm 2.

Algorithm 1 Caching candidate set enumeration

```
Input:
    $T_1, T_2, \ldots$        no-cache job completion times, longest first
    $\alpha_1, \alpha_2, \ldots$      per-job time improvement per unit cached
    $I_1, I_2, \ldots$        job inputs

Output:
    $c_1, c_2, \ldots$        caching candidates
    $c_i$ specifies data to be cached from $I_1 \ldots I_i$, and has value
    (i.e. time saved) = $T_i - T_{i+1}$

procedure CANDIDATES(stage $i$)
    $T_{min} = T_1 - \alpha_1 \cdot |I_1|$
    $t = (T_1 - T_2),\ c_1 = \{I_1 : \frac{t}{\alpha_1}\}$
    $t = (T_2 - T_3),\ c_2 = \{I_1 : \frac{t}{\alpha_1}, I_2 : \frac{t}{\alpha_2}\}$
    etc. while $T > T_{min}$
end procedure
```

The candidate enumeration algorithm in Algorithm 1 examines stages from longest to shortest within a stage-set, enumerating candidates in decreasing order of benefit (completion time saved) to cost (size). The first candidate will be from the input to the longest stage, of sufficient size to reduce its runtime to that of the next-longest—i.e. the green segment from Figure 5a. The second candidate in Figure 5a corresponds to the blue segments, reducing the runtime of S_3 and S_4. Candidate enumeration stops when no more candidates can be enumerated, e.g. in this case where S_3 and S_4 runtimes are reduced to their minimum.

Candidate selection is performed by Algorithm 2; we describe this first for the case of a single active DAG, before discussing its operation across multiple DAGs.

After enumerating caching candidates for all future stage-sets in decreasing benefit:cost order, we compute the "slack" back-end storage bandwidth available in each stage-set based on the current estimated stage-set completion time and measured storage access rate. Candidates which are "too early" are eliminated; these are ones that may be safely deferred to a later stage-set and still complete by the time of the stage-set in which they are needed.

We then consider the remaining candidates—if a candidate "fits" into the remaining cache space and bandwidth, we schedule the prefetching operation (if needed) and "pin" the candidate in cache until the end of the stage in which it is needed. In the next step, CMR updates available cache space and slack bandwidth (prefetch bandwidth), as well as adjusting stage completion time estimations to account for the speedup. If we run out of prefetch bandwidth before cache space (omitted for clarity in Algorithm 2), we continue examining in-cache candidates until we run out of cache space.

This strategy not only calculates a set of data to prefetch but implicitly calculates evictions as well. Data currently in the cache which is valuable for reducing the runtime of a later DAG stage will be part of one of the selected candidate sets, and will be pinned through the end of its scheduled use; the remaining cache contents are unpinned and may be evicted

Algorithm 2 Cache candidate selection: schedule prefetching/pinning for later stages

```
    Initial Conditions:
 2:     $t_0 = 0, t_i = t_{i-1} + max(T_{i,*})$        ▷ Estimated stage completion times
        $fetch_i = sum(|I_{i,*}|)$                  ▷ committed bandwidth by stage
 4:
    Input:
 6:     candidate lists for each active DAG

 8: Output:
        prefetch, pin, and unpin operations
10:
    procedure PLAN($lists$)
12:     while slack$_{bw}$ and cache available do
            $lists \leftarrow sort\left( \frac{T_{list}}{\sum_{b \in frags(list)} \left(\frac{1}{n_b}\right)} \text{ for } list \text{ in } lists \right)$   ▷
    $n_b : N_{dags}$ shared b.
14:         link $l_1$ and $l_2$ if $l_1$ share blocks with $l_2$ for each $l_1$ and $l_2$ in $lists$
            slack$_{bw}(j) \leftarrow r \cdot (t_j - t_{j-1})$        ▷ bandwidth slack in stage $j$
16:         $c = best(head(list)$ for $list$ in $lists)$        ▷ best candidate
            skip c if stage(c)>now and c fits in slack$_{bw}$(future)
18:         $s \leftarrow stage(c)$
            $fetch_s \leftarrow fetch_s - |c|$        ▷ $|c|$ no longer demand-fetched
20:         if $c$ not in cache then
                $fetch_{now} \leftarrow fetch_{now} + |c|$
22:             prefetch($c$)
            end if
24:         adjust $t_s$,... for $c$ speedup
            pin $c$ until end of $s$
26:         cache used += $|c|$
        end while
28: end procedure
```

(e.g. in LRU order) if necessary to make room for new data.

Event Scheduling: in most analytic frameworks that implement the event-based scheduling schema, e.g. Spark, the root stages of the DAG (those with no-dependency) are responsible for fetching DAG input data. In this case, CMR, for each root stage, predicts the longest path to the leaf stages of the DAG (those with that produce output). Then, it sorts these paths according to the predicted run-time and recursively identifies the cache candidates for them.

4.3 Multi-DAG Scheduling

Next, we describe the algorithm that prefetches for multiple DAGs simultaneously, *Cache for Minimizing Runtime of Multiple DAGs* (CMR-M). It attempts to maximize throughput by minimizing the time to completion of sequences of DAGs.

CMR-M assumes the use of *static partitioning* (default setup) in Spark and Pig, where resources are allocated to a DAG for the entire period of execution [14].

We enumerate caching candidates using Algorithm 1, then we choose to execute candidates in Algorithm 2, as before; selecting the "best" cache candidates from competing per-DAG lists via a heuristic score for data sharing, *sharing-aware weight*. The *sharing-aware weight* is calculated per cache candidate and gives preference to candidates that are shared by other running DAGs, as the throughput increase from prefetching will be higher than that indicated by the single-DAG value.

The *sharing-aware weight* is similar to the resident set size (RSS) calculation [35] from 'ps': data shared by multiple DAGs is split equally among them before calculating cost. We calculate the *sharing-aware weight* on a block-per-block basis, counting the number of DAGs n_b sharing any block b (Equation 2). We then calculate a "unique file size", U_c, counting each block in C shared between n_b DAGs as having size $\frac{1}{n_b}$ (line 12 – 14 in Algorithm 1). We then re-compute the marginal utility using this weight and use this utility to compare candidates across DAGs. Finally, we find the "partners" to the selected candidate, i.e. those sharing blocks with it, and select those for prefetching/caching as well.

$$U_s = \sum_{b \in frags(C)} \left(\frac{1}{n_b} \right) \qquad (2)$$

5 Implementation

The Kariz implementation combines a Kariz service with a our previously developed [44] caching layer embedded within the Ceph Rados Gateway (RGW [8]). We modified this caching layer for Kariz by integrating fine-grained prefetching and pinning (~100 C++ LOC). We have also modified both Pig/Hadoop [56] (~100 Java LOC) and Spark [72] (30 Scala LOC) to work with Kariz. We discuss each of these components in turn.

5.1 Kariz Service

Our Kariz prototype is about 5000 lines of Python[3], including the runtime predictor, the MRD, CP and CMR DAG planners, and the CMR-M multi-DAG planner. We use the `sklearn` package for runtime prediction, and `graph-tool` for graph traversal. The Kariz service also includes a cache coordinator that translates high-level operations from the planner into individual block operations on the cache.

Interfaces between the analytic frameworks and Kariz are listed in Table 1. The *newDAG*, *stageStart*, and *completeDAG* notifications from the framework trigger DAG planning activities, and carry information (e.g. annotated DAGs) needed for planning. The *prefetch*, *pin*, and *unpin* requests to the cache controller, in turn, translates high-level requests from the planner (specifying object and stride) into requests to the cache to fetch, pin/unpin, or evict individual blocks.

5.2 Caching layer

We build Kariz by extending the multi-tenant cooperative caching architecture recently added to RGW [44] This RGW cache layer expands to multiple clusters and allows different frameworks such as Hadoop MapReduce [26] and Apache Spark [72] to cache and share their inputs. Our extension to support Kariz involved around 100 C++ LOC. This involved

[3]http://github.com/maniaabdi/Kariz

Table 1: Interface from Analytics framework (e.g. Pig or Spark) to Kariz and between Kariz and the Cache

	API	Description
→ Kariz	`newDAG(ID, DAG)`	new DAG started
	`stageStart(ID, stage)`	Jobs in `stage` scheduled for execution
	`completeDAG(ID)`	DAG completed.
→ Cache	`prefetch(blocks)`	Asynchronously fetch blocks into cache
	`pin(blocks)`	Lock blocks in cache.
	`unpin(blocks)`	Release blocks to be replaced as space is needed

adding the operations to prefetch pin and unpin lists of $4MB$ chunks of datasets, as depicted in Table 1. The small changes required is evidence that we will be able to integrate Kariz with other caching services. Kariz could integrate with other distributed caching systems such as Alluxio [2, 50] and coordinates their caches. In the case of Alluxio minor modifications are needed to support partial prefetching.

5.3 Analytical frameworks modifications

To exploit Kariz, an analytic framework must provide some interface that Kariz can use to extract run time interface and notify Kariz of new DAGs, the start of stages, and DAG completion using the interface in Table 1. We have found it relatively easy to develop adaptors for two frameworks, Pig [56] and Spark [72]; suggesting that framework developers will find it easy to add the required functionality to take advantage of Kariz.

Pig modifications: Modifications to Pig are 100 Java LoC in the following functions: (1) *compile()* in MapReduceLauncher.java to extract the DAG, annotate it, and invoke *newDAG*. (2) *launchPig()* in MapReduceLauncher.java to extract the stage and invoke *stageStart*. (3) *dumpStats()* in MRPigStatsUtil.java to invoke *completeDAG*; Kariz then request detailed statistics from Hadoop history server.

Spark modifications: Modifications to Spark are 50 Scala LoC in the following functions: (1) the *constructor* in SQLExecutionRDD.scala, and *toRdd()* in QueryExecution.scala to annotate the RDD DAG, (2) *runJob()* in SparkContext.scala to extract DAG and invoke *newDAG*, where the ID is based on a UUID and spark application ID, (3) *submitStage()* in DagScheduler.scala to extract the stage and invoke *stageStart* (4) When the SparkContext shuts down, it invokes *completeDAG*; Kariz then request detailed statistics from Spark history server.

6 Evaluation

We use a combination of experimental evaluation on our prototype and simulation to evaluate Kariz. After describing the experimental infrastructure and simulator (§6.1), we **experimentally** demonstrate the value of partial caching (§6.2), examine the accuracy of our run-time prediction (§6.3), and then show results for the different DAG planners with both PIG and Spark (§6.4). The remainder of the evaluation uses **simulation** to evaluate the single DAG planners for a larger set of queries (§6.5), explore the multi-DAG planners for both queries from separate PIG and Spark clusters, and when Kariz is simultaneously used by both PIG and Spark clusters (§6.6) and finally perform sensitivity (§6.7) and scalability (§6.8) analysis.

Table 2: Hardware configuration

	Compute Server	Cache Server
CPU	1x Intel E5-2650	2x Intel E5-2699v3
RAM	128 GB	128 GB
Disk	1x 500 GB HDDs 5400 RPM	2x Intel P3600 1.6 TB NVMe SSDs (RAID0)
Network	10Gb/s	40Gb/s

Table 3: Software configuration

	Hadoop	Pig	Spark
Version	2.8.4	0.17.0	2.4.5

6.1 Setup

Infrastructure: The physical experiments with Pig/Hadoop and Spark are run on a 16 node cluster with the hardware and software configuration in Table 2 and Table 3. We provisioned the compute nodes via diskless provisioning [53] and use the local disks of the compute nodes to deploy local HDFS. We use the NVMe SSDs of the cache servers to build the cache layer.

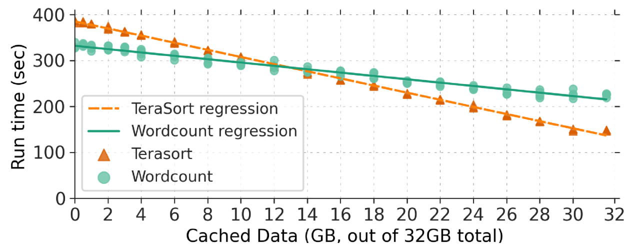
Simulator: We implement a simulated execution framework and cache, allowing additional experiments not possible on the physical cluster[4]. Execution time for each job to be simulated was determined by the run-time prediction model trained for each operation in Pig/Hadoop and Spark with different cache sizes. A random term was added to the runtime, with standard deviation taken from measured run-time. Additional simulator logic mimics the Kariz extensions to the framework scheduler, allowing the same Kariz code to be used in physical experiments and simulations.

6.2 Partial Caching

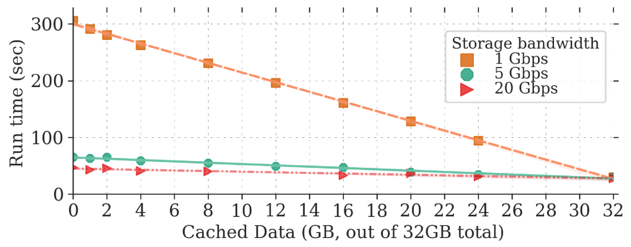
We evaluate a key premise of Kariz: that straggler-resistant partial caching can reduce runtimes. In Figure 6a, we see experimental results for `Wordcount` and `TeraSort` on a 16-node Hadoop cluster with 128 mappers and 32 GiB input files. In Figure 6b, we see the `Wordcount` benchmark on a 16-node Spark cluster with 64 GiB input files, and vary bandwidth to remote storage.

With both Hadoop and Spark we see a linear relationship between run time and cached data. While these are trivial applications, given that the platform partitions the data across the mappers, we believe this is good evidence that partial caching can be effective. In contrast, random choice of 4 MiB blocks was found to produce little or no speedup when less than 60% of the input was cached, and (as expected) caching a strict prefix of the file produced no improvement until the entire file was in cache.

[4]As well as timely reproduction after algorithm changes.



(a) *Hadoop: WordCount and TeraSort, cached data vs. runtime, 128 mappers, 1Gbit backend bandwidth.*



(b) *Spark: cached data vs. runtime for WordCount and different backend bandwidth on 32 GiB input, 128 partitions.*

Figure 6: *Partial caching*

We also see that the slope varies with storage bandwidth and application, motivating our design to build per-operation runtime prediction model that incorporates storage bandwidth.

6.3 Runtime prediction

Training model: to train the runtime prediction models, we ran Lasso regression with $\alpha = 0.001$ on each category. On our test cluster, we run all 44 queries from Pig-TPCH [13] and Spark-TPCH [16], 13 times each, with different configuration: (1) input datasets randomly selected from 8GB to 80GB, (2) number of cached blocks randomly selected from 0% to 100%, (3) the bandwidth to backend storage restricted randomly from 1Gbps to 40GBps, and (4) the number of executors per query was configured randomly from the 2, 4, 8, 16. In total, we captured statistics for 286 queries for Pig/Hadoop and 286 queries for Spark. We use the 80%/20% split to train and test each model. Table 4 shows the average run time of the test set, the root mean square error (RMSE) of the runtime prediction model per operation and average absolute error.

Prediction accuracy: The caching and prefetching planners depend on the Kariz runtime predictor being accurate enough to predict the correct paths (longest, the 2^{nd}-longest, etc.) to cache for. We ran the 22 queries from the Spark TPCH benchmark, with random input sizes, on a 16-node cluster with no caching. We predict the run-time using the trained models: for 20 queries out of 22 the order for 1^{st}, the 2^{nd}, and 3^{rd} longest path were identified correctly. For query Q11, it mis-predicts the 1^{st} longest path, and for query Q21, the order of the 2^{nd} and 3^{nd} longest paths were reversed. In these two cases, the errors were in paths differing by less than 3s; in almost all cases either both or neither would be cached, and the misprediction impact would be minor.

In Figure 7, we see the ratio of the actual to predicted longest path. The maximum error was 27% and on average 7%. We annotate each bar with the bandwidth, the input size, and the actual runtime of the longest path for that query. For Q6, where we had the maximum relative error, the actual runtime was short.

6.4 Experimental evaluation

We compare CMR with two DAG-informed policies: (1) MRD [57], which caches and prefetches in breadth-first order to increase hit ratio, and (2) our CP [17] which caches and prefetches for jobs on the DAG critical path.

Workloads: We use the characteristics of the confidential trace (including query submission rate, DAG structure, data accessed, data reuse) to construct a mix of synthetic workload using standard analytic benchmarks (TPC-H and TPC-DS) because of the lack of public workloads. The synthetic benchmark represents an hour of data processing. We scale down the size of our cluster and the DAG submission rate by a factor of 10; in the confidential traces, the job submission rate follows the Poisson distribution with distribution parameter(λ) of 0.2 (on average

Table 4: Accuracy of runtime prediction per operation

(a) Pig operations

Operation	$\overline{runtime}$ (s)	*RMSE* (s)	$\overline{Absolute}$ (s)
Co-group	330	63.27	61.14
Map only	5.8	0.33	0.26
Groupby	14.6	1.8	1.6
Combiner	23	8.3	2.8
Hash join	99.3	37	25
Replicated join	251	38.5	55
Order by	10.6	0.39	0.49
Sampler	10.74	0.58	0.54

(b) Spark operations

Operation	$\overline{runtime}$ (s)	*RMSE* (s)	$\overline{Absolute}$ (s)
Hash aggregate (HA)	1.3	0.66	0.52
Scan	12	3.3	2
Scan, Filter	20.2	6.28	3.27
Scan, Filter & HA	30.6	5.2	3.97
Filter & HA	2	0.66	0.63
Sort merge join	3.51	1.5	0.94
Sort merge join & HA	3.53	0.75	0.53

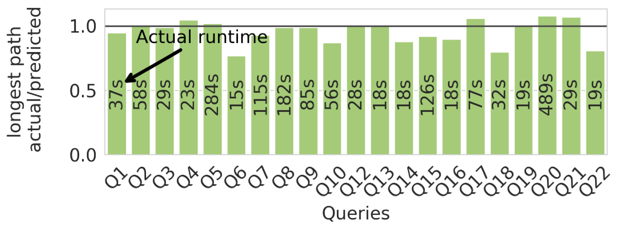


Figure 7: *Ratio of actual to predicted longest path on different Spark queries. See text for Q11.*

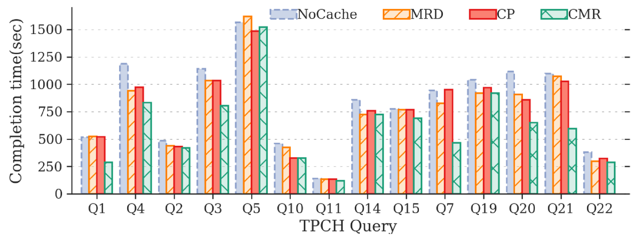


Figure 8: *Pig-MapReduce performance for selected TPC-H queries - small DAGs (1,4), long sequential (2,3,5), tree-like (10,11,14), aggregate (15), large/complex (7,19-22). 64 GiB data set, 40 GiB cache, cold start.*

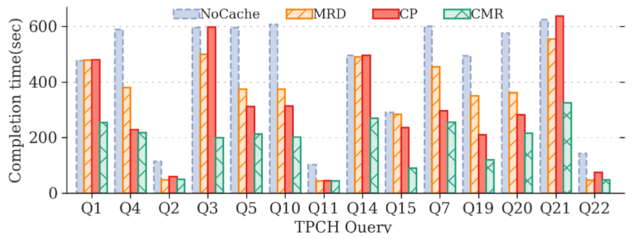


Figure 9: *TPC-H query performance using Spark contrasting CMR, CP, MRD and no caching.*

702 queries were submitted per hour). Thus, we use Poisson distribution with λ of 0.02 to generate query submission events (result in 67 queries). Then, at each event, a query is drawn from a pool of 22 TPC-H [13], and 19 TPC-DS [12] queries translated into Pig Latin [56]. To select queries from the pool, we categorize queries according to their DAG structure for a sampled hour. We map each query in the pool to a category with the maximum DAG similarity (maximum common subgraph [55]). To be consistent, for Spark, we use the same set of queries from Spark TPCH [16] and Spark TPC-DS benchmarks [15].

For the sampled hour, 30% of the objects were accessed at least twice. This is similar to the ZipF distribution with $a = 1.125$. Accordingly, we assign each query a dataset selected from this distribution, giving a trace of 496 accesses over 357 unique input objects, with 78 accesses to the most used input object, consistent with our evaluated traces. Then, we associate the input sizes to the datasets by randomly choosing from 4 GB, to 256 GB. Finally, we use the standard TPC-H [13] and TPC-DS [12] input generators to create CSV datasets.

The cache size is set to 40GB and the dataset size is 64GB. The RGW cache network to datalake is throttled to 10Gbps. For each query, we assume 5 seconds queue time, the minimum observed queue time in the evaluated traces, before submitting the first stage for execution to the execution engine. We take advantage of the queue time to start prefetching for the DAG and we clear the cache before each run. The reported numbers are the average of three runs.

Performance evaluation: Figure 8 and Figure 9 shows runtime of selected TPC-H queries in, respectively, Pig-Latin and Spark comparing CMR, CP, MRD, and no caching.

Our evaluations show that relative to the no caching case, CMR can improve query performance by up to 2 times for Pig-MapReduce and up to 3.2x on Spark, with mean speedups of 1.3x and 1.8x respectively. Running on the Pig-MapReduce framework and comparing to MRD and CP, CMR can improve the runtime by up to 2x and 1.8x and in average by 1.3x and 1.3x respectively. Our experiments using LRU shows similar behavior to no-prefetching (gray bar).

With the Spark framework, CMR can improve the runtime compared to MRD and CP by up to 3.1x and 3x and in average by 1.8x and 1.7x respectively. Spark shows more sensitivity to the backend storage bandwidth than MapReduce, Since MapReduce imposes extra overheads such as JVM start up([45, 46]). This results in sharper speed up slope for CMR on the Spark framework than the Pig-MapReduce framework and better cache space utilization compared to MRD and CP.

Q1 and Q4 represent small-sequential queries; with reads only at the begining of the job in the graph. Due to the partial caching strategy implemented by CMR, it has better performance compared to MRD, CP on both Pig/MapReduce and Spark frameworks. The table shows the average speedup:

Q1 and Q4	MRD	CP	no caching
Pig/MapReduce	1.5x	1.5x	1.6x
Spark	1.8x	1.4x	2.2x

For Q2, Q3, Q5, Q10, Q11, Q14, and Q15, the structure of DAGs generated by Pig and Spark is different, which leads to different caching decisions. On Pig-MapReduce, Q2, Q3, and Q5 are long sequential queries with small reads in the first stages and large reads in the following one. Here, the CMR ranking mechanism makes it possible to prioritize prefetching plans that have more effect on the sequential DAGs. Q10, Q11, and Q14 are long sequential graphs and Q15 is a large aggregated graph. For these, the combinations of stage oriented decisions and partial caching leads to performance improvement.

Q7 and Q19 to Q22 have large complicated DAGs on both Pig-MapReduce and Spark. The excellent relative performance of CMR over the other options for these queries (see table below) is encouraging, as our analysis of real-world traces in §1 showed over 90% of data read by complex queries like these.

Q7 and Q19-Q22		Pig/MapReduce	Spark
Average	MRD	1.4x	1.8x
	CP	1.5x	1.5x
	no caching	1.6x	2.8x
Maximum	MRD	1.8x	2.9x
	CP	2x	2x
	no caching	2x	3.2x

6.5 Simulated evaluation - Single DAGs

We evaluate CMR across the synthetic workload (67 TPC-H and TPC-DS queries) using our simulated cache and Pig framework. In Figure 10 we see CMR, CP, and MRD performance

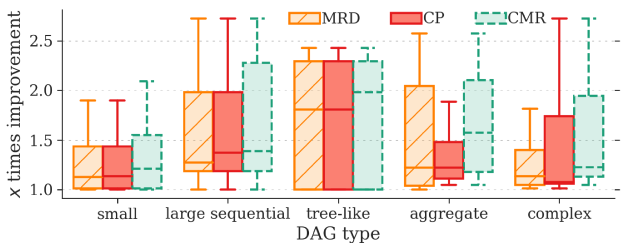


Figure 10: *Runtime by DAG class for 67 queries; 128 GiB cache, 10 Gb/s backend bandwidth.*

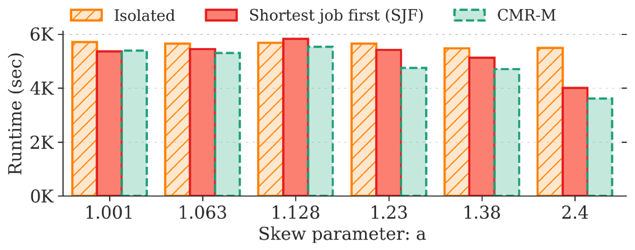


Figure 11: *Performance of different caching strategies when different level of data sharing exist within the cluster. (simulated)*

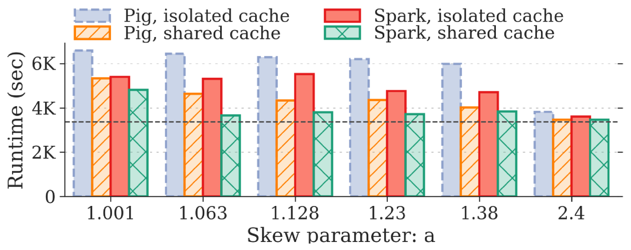


Figure 12: *Performance of different frameworks with shared cache vs isolated cache per framework. Each cache has 1 TB cache, 25 Gb/s storage-bandwidth, and 100 Gb/s cache bandwidth. (simulated)*

relative to no cache, with a simulated cache size of 128 GiB, grouped by query type.

Median (center line) and 75th-percentile (top of the box) performance are seen to be higher than MRD or CP in all cases. CMR performance is much higher (1.6x vs 1.2x for MRD and CP) for aggregation queries, although the top quartile of queries achieved relatively similar speedups for CMR and MRD. For complex queries CMR outperformed MRD by large amounts, with a 75th-percentile speedup higher than the maximum MRD speedup.

6.6 CMR Performance on Multiple DAGs

Kariz is a *community cache* that considers data sharing between DAGs running on one or several analytic frameworks. We simulate Kariz to measure the performance under two scenarios:
Multiple DAGs in a single analytic cluster: We compare the performance of three multi-DAG strategies: CMR-M, shortest-job-first (SJF) (prefetches/caches for DAGs with the shortest remaining runtime), and Isolated (static isolated cache partitioning for each DAG) [25]. In all three cases, CMR is used to manage within-DAG caching/prefetching decisions.

We simulate 1 TB of cache, 25 Gbit/s network bandwidth, and 100Gbit/s cache bandwidth. We generate a workload that consists of 200 randomly-chosen Spark TPC-H [16] queries with a dataset size of 164GB in a cluster that can handle 10 simultaneous queries. For the Isolated strategy, we allocate 128 GiB of cache space to each query. To produce different sharing patterns, we generate 6 traces of 200 datasets generated by changing the ZipF distribution parameter (a: 1.001-2.4)– e.g. a = 1.001 giving a trace with 192 unique dataset accesses. Finally, we map each dataset in every trace to one query.

In Figure 11, we see the end-to-end runtime of all 200 TPC-H Spark queries with 6 traces, when we increase the reuse/sharing of datasets within the trace. As seen, CMR-M outperforms both isolated cache and shortest job first by up to 1.51× (a = 1.23) and 1.14× (a = 1.38), respectively. As depicted, the SJF policy can degrade performance compared to isolated cache. The reason is SJF favors DAGs with smaller predicted runtime. This results in DAGs with longer runtime deprived of the cache and therefore to read most of their data from the backend. For the Spark cluster, the runtime for the base case (all data remote) is 13000 seconds; for a = 0.001, i.e. almost no data sharing, the performance gain over that comes from data prefetching.
Two analytic clusters: We compare the performance of Kariz with multiple analytic clusters (Pig/MapReduce and Spark) sharing a cache vs the case where the cache is statically partitioned, using CMR-M with the CMR single-DAG planner. We use the same 200 Spark TPC-H queries, combined with 100 Pig TPC-H queries randomly selected from the same distribution, generating 6 traces with 300 datasets each as described above.

As shown in Figure 12, by increasing the data sharing across analytic clusters, both clusters have seen runtime improvement vs the statically-partitioned case. Pig cluster runtime improves by up to 1.5× (a = 1.38), with a mean of 1.3×. The Spark cluster benefits from both prefetching and caching, improving on average by 1.27× and up to 1.52× (a = 1.128). The dashed horizontal line in the Figure 12 shows the extreme case when one dataset is shared by all the queries in both clusters.

6.7 Sensitivity Analysis

We analyze sensitivity to cache size and prediction errors.
Cache size: Figure 13 shows the average speed up of all DAGs from the mixed workloads (§6.1) as we vary the cache size from 16GB to 400GB (the size of the dataset) with the network bandwidth to the backend set to 10Gbps. CMR achieves substantially higher performance compared to MRD and CP until the entire data set fits in the cache. For example, when the cache size is 64GB, CMR outperforms MRD and CP by up to 51% and on average 10% and 8% respectively.
Impact of runtime mis-prediction: To see the effect of runtime misprediction we introduce a multiplicative error factor R_{error}. We simulate 27 queries with a total of 310 jobs from the

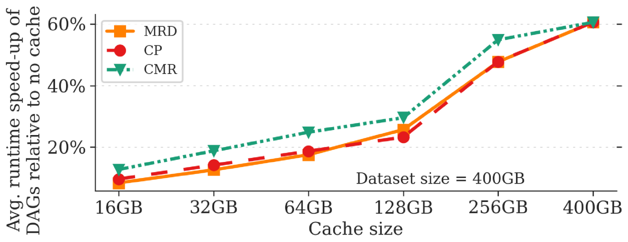


Figure 13: *Mean runtime across all queries vs. cache size, normalized to uncached runtime; 10 Gb/s bandwidth. (simulated)*

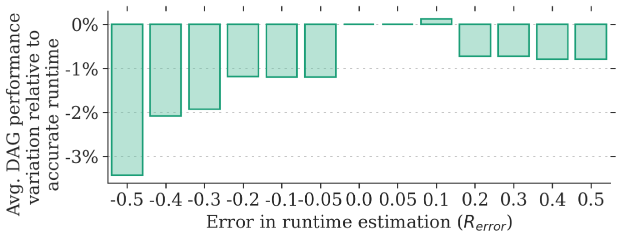


Figure 14: *Sensitivity to misprediction error: introduced error vs. performance degradation - 30% of predications adjusted by factor of* $(1+R_{error})$. *(simulated)*

mixed workloads in single-DAG mode, with 60% of jobs recurring, and randomly pick 130 jobs (~42%) to adjust by R_{error}.

In Figure 14 we see normalized change in runtime, relative to no mis-estimation, for values of R_error between 0.5 and 1.5. CMR performance drops when the runtime is mispredicted, especially in the negative direction, but when only a fraction of jobs are mispredicted the effect is small.

6.8 Scalability

To analyze CMR scalability, we run (in simulation) a pool of 67 queries(DAGs) from TPC-H and TPC-DS benchmarks. Using timing from the Alibaba traces [1] (80% of DAG stages complete in less than one minute) we assign a random execution time between 1s to 60s to each stage. We submit DAGs at a rate of 90 per minute and measure the execution time for CMR planning.

Figure 15 shows CMR planner runtime vs a number of currently executing DAGs. Execution time (in unoptimized Python) is seen to be under six seconds in all cases, with up to 160 concurrent DAGs. Based on Alibaba statistics this would allow scaling a single controller to a cluster of 1500 to 2000 servers, with minimal delay in issuing prefetch commands.

7 Conclusion

Kariz is a cache management system for analytic frameworks that makes possible cache algorithms informed by DAGs, historical run time information, current cache state, and storage bandwidth. We have implemented multiple algorithms using Kariz, including a new CMR algorithm that achieves dramatic performance improvements by exploiting all this information.

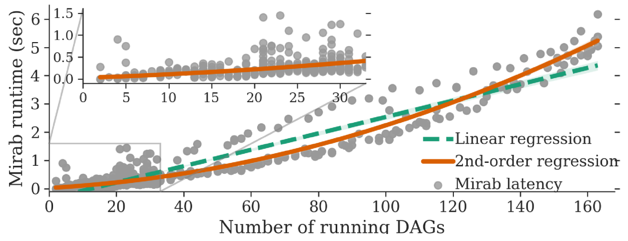


Figure 15: *CMR-M scaling - planner runtime vs running DAGs. (simulated)*

We demonstrate that new analytics frameworks (100 LOC for PIG/Hadoop, and 30 LOC for SPARK) and cache systems (100 LOC for the cache we used) can easily be integrated. Our work is the first to: 1) support multiple concurrent DAGs, 2) employ more than one of bandwidth, runtime prediction, and DAGs, 3) explore a cooperative caching model, and 4) employ straggler resistant partial caching.

Acknowledgment

We thank our shepherd, Nitin Agrawal, and our anonymous FAST reviewers for their valuable feedback and suggestions. We thank Shankar Pasupathy, Art Harkin, Peter Macko, and Xing Lin of NetApp, Matt Benjamin and Ali Maredia of Red Hat for their support and contribution. We would like to acknowledge the support of our commercial partners in Mass Open Cloud and Open Infra Labs, which include Red Hat, Two Sigma, Intel, IBM, Brocade, Cisco, and Lenovo. Partial support for this work was provided by the National Science Foundation award CNS-1910327, CNS-1414119, and a NetApp faculty fellowship.

References

[1] Alibaba trace data. `http://github.com/alibaba/clusterdata`, 2019.

[2] Alluxio. `http://www.alluxio.org`, 2019.

[3] Amazon EMR. `http://aws.amazon.com/emr/`, 2019.

[4] Amazon S3. `http://aws.amazon.com/s3/`, 2018.

[5] Anatomy of the S3A filesystem client. `http://redhat.com/en/blog/anatomy-s3a-filesystem-client`, 2018.

[6] Apache Arrow. `http://arrow.apache.org/`, 2019.

[7] Apache Ignite. `http://ignite.apache.org/`, 2019.

[8] Ceph Object Gateway. `http://docs.ceph.com/docs/master/radosgw/`, 2019.

[9] Dave Wells. The Future of the Data Warehouse. `http://eckerson.com`, 2017.

[10] Microsoft Azure HDInsight. `http://azure.microsoft.com/services/hdinsight/`, 2019.

[11] Microsoft Datalake. `http://azure.microsoft.com/en-us/solutions/data-lake`, 2019.

[12] Pig TPC-DS queries. `http://github.com/ssavvides/tpcds-pig`, 2019.

[13] Pig TPC-H queries. `http://github.com/ssavvides/tpch-pig`, 2019.

[14] Scheduling Spark cluster. `http://spark.apache.org/docs/latest/job-scheduling`, 2019.

[15] Spark TPC-DS queries. `http://github.com/databricks/spark-sql-perf`, 2019.

[16] Spark TPC-H queries. `http://github.com/ssavvides/tpch-spark`, 2019.

[17] Mania Abdi, Amin Mosayyebzadeh, Mohammad H. Hajkazemi, Ata Turk, Orran Krieger, and Peter Desnoyers. Caching in the Multiverse. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[18] Waleed Ali, Siti Mariyam Shamsuddin, and Abdul Samad Ismail. A Survey of Web Caching and Prefetching. *International Journal of Advances in Soft Computing and its Applications*, 3, 03 2011.

[19] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association.

[20] 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.

[21] Danilo Ardagna, Enrico Barbierato, Athanasia Evangelinou, Eugenio Gianniti, Marco Gribaudo, Túlio B. M. Pinto, Anna Guimarães, Ana Paula Couto da Silva, and Jussara M. Almeida. Performance Prediction of Cloud-Based Big Data Applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 192–199, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[23] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, Cascais, Portugal, October 2011. Association for Computing Machinery.

[24] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1205–1223. USENIX Association, November 2020.

[25] Jon Crowcroft and Philippe Oechslin. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Comput. Commun. Rev.*, 28(3):53–69, July 1998.

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

[27] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 7–17, Feb 2009.

[28] Iman Elghandour and Ashraf Aboulnaga. ReStore: Reusing Results of MapReduce Jobs. *Proc. VLDB Endow.*, 5(6):586–597, February 2012.

[29] Hajar Falahati, Mania Abdi, Amirali Baniasadi, and Shahin 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.

[30] Hajar Falahati, Shahin Hessabi, Mania Abdi, and Amirali Baniasadi. Power-efficient prefetching on GPGPUs. *The Journal of Supercomputing*, 71(8):2808–2829, Aug 2015.

[31] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.

[32] Seyedeh G. Ghaemi, Iman Ahmadpour, Mehdi Ardebili, and Hamed Farbeh. SMARTag: Error Correction in Cache Tag Array by Exploiting Address Locality. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1658–1663, 2018.

[33] Seyedeh G. Ghaemi, Iman Ahmadpour, Mehdi Ardebili, and Hamed Farbeh. Sleepy-LRU: extending the lifetime of non-volatile caches by reducing activity of age bits. *The Journal of Supercomputing*, 75(7):3945–3974, 2019.

[34] Seyedeh G. Ghaemi, Amir M. H. Monazzah, Hamed Farbeh, and Seyed G. Miremadi. LATED: Lifetime-Aware Tag for Enduring Design. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 97–107, 2015.

[35] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[36] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[37] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters, booktitle = Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[38] Mohammad H. Hajkazemi, Mania Abdi, and Peter Desnoyers. Minimizing Read Seeks for SMR Disk. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 146–155, 2018.

[39] Mohammad H. Hajkazemi, Mania Abdi, and Peter Desnoyers. uCache: a mutable cache for SMR translation layer. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.

[40] 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.

[41] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. *SIGCOMM Comput. Commun. Rev.*, 45(4):407–420, August 2015.

[42] 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.

[43] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 191–203, New York, NY, USA, 2018. ACM.

[44] Emine Ugur Kaynar, Mania Abdi, Mohammad H. Hajkazemi, Ata Turk, Raja R. Sambasivan, David Cohen, Larry Rudolph, Peter Desnoyers, and Orran Krieger. D3N: A multi-layer cache for the rest of us. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 327–338, 2019.

[45] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, July 2018. USENIX Association.

[46] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[47] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[48] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[49] Umesh Kumar and Jitendar Kumar. A Comprehensive Review of Straggler Handling Algorithms for MapReduce Framework. *International Journal of Grid and Distributed Computing*, 7(4):139–148, August 2014.

[50] 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.

[51] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[52] Apoorve Mohan, Shripad Nadgowda, Bhautik Pipaliya, Sona Varma, Sahil Suneja, Canturk Isci, Gene Cooperman, Peter Desnoyers, Orran Krieger, and Ata Turk. Towards Non-Intrusive Software Introspection and Beyond. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pages 173–184, 2020.

[53] Apoorve Mohan, Ata Turk, Ravi S. Gudimetla, Sahil Tikale, Jason Hennesey, Emine Ugur Kaynar, Gene Cooperman, Peter Desnoyers, and Orran Krieger. M2: Malleable Metal as a Service. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 61–71, 2018.

[54] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting Security Sensitive Tenants in a Bare-Metal Cloud. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 587–602, Renton, WA, July 2019. USENIX Association.

[55] Siegfried Nijssen and Joost N. Kok. The Gaston Tool for Frequent Subgraph Mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77 – 87, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

[56] 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.

[57] 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.

[58] 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.

[59] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 1–8, New York, NY, USA, 2018. Association for Computing Machinery.

[60] Elizabeth Shriver, Christopher Small, and Keith A Smith. Why does file system prefetching work? 1999.

[61] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[62] 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.

[63] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

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

[65] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[66] 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, March 2016. USENIX Association.

[67] 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.

[68] Luna Xu, Min Li, Li Zhang, Ali R. Butt, Yandong Wang, and Zane Zhenhua 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.

[69] Yinggen Xu, Liu Liu, and Zhijun Ding. DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 378–387, May 2020.

[70] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Trans. Comput. Syst.*, 29(2):5:1–5:51, May 2011.

[71] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled 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.

[72] 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, Berkeley, CA, USA, 2010. USENIX Association.