Blaze: Holistic Caching for Iterative Data Processing

Won Wook SONG
FriendliAI

Jeongyoon Eo
Seoul National University

Taegeon Um
Samsung Research

Myeongjae Jeon*
UNIST

Byung-Gon Chun*
Seoul National University and
FriendliAI

Abstract

Modern data processing workloads, such as machine learning and graph processing, involve iterative computations to converge generated models into higher accuracy. An effective caching mechanism is vital to expedite iterative computations since the intermediate data that needs to be stored in memory grows larger over iterations, often exceeding the memory capacity. However, existing systems handle intermediate data through separate operational layers (e.g., caching, eviction, and recovery), with each layer working independently in a greedy or cost-agnostic manner. These layers typically rely on user annotations and past access patterns, failing to make globally optimal decisions for the workload.

To overcome these limitations, Blaze introduces a unified caching mechanism that integrates the separate operational layers. Blaze dynamically captures the workload structure and metrics using profiling and inductive regression, and automatically estimates the potential data caching efficiency associated with different operational decisions based on the profiled information. To achieve this goal, Blaze incorporates potential data recovery costs across stages into a single cost optimization function, which informs the optimal partition state and location. This approach reduces the significant disk I/O overheads caused by oversized partitions and the recomputation overheads for partitions with long lineages, while efficiently utilizing the constrained memory space. Our evaluations demonstrate that Blaze can accelerate end-to-end application completion time by $2.02 - 2.52\times$ compared to recomputation-based `MEM_ONLY` Spark, and by $1.08 - 2.86\times$ compared to checkpoint-based `MEM+DISK` Spark, while reducing the cache data stored on disk by 95% on average.

*Corresponding authors

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629558

CCS Concepts: • **Computer systems organization → Distributed architectures**; • **Software and its engineering → Distributed systems organizing principles**; • **Information systems → Data management systems**.

Keywords: Data Processing, Distributed Systems, Caching

1 Introduction

Data analytics applications today are increasingly focused on formulating models to emulate real-world phenomena through methods like machine learning and graph processing. For example, the PageRank algorithm captures the importance of web pages from the vast amount of internet data [52], while logistic regression has proven effective in forecasting probabilities of certain events across numerous fields [12]. Many of such applications exhibit multiple iterations of repetitive operations that evolve the model into high accuracy [24, 35, 39, 50]. To express and execute these operations at hand, programmers typically rely on data analytics systems built upon dataflow-based execution models [5, 36, 46, 74]. Here, caching plays a key role in system efficiency, as reusing intermediate data eliminates the need for recomputations when data is repeatedly accessed.

However, iterative data processing results in a consistent expansion of intermediate data, putting a strain on memory resources [26]. Unfortunately, simply provisioning ample memory to accommodate all intermediate data is not a silver-bullet solution given that data size can increase by more than 10× the input size over the iterations (§7.2). Moreover, while each of the iterations has a declarative job structure, such jobs are dynamically submitted until convergence iteration-by-iteration, which makes the overall lineage of the workload unpredictable before the actual execution.

Furthermore, upon facing memory shortages, a substantial volume of intermediate data is dynamically evicted to be later retrieved for reuse during the successive iterations. Notably, the state (or location) of this intermediate data – whether it is in memory, on disk, or not in any persistent storage – also changes dynamically throughout the iterations, making static analyses for optimal caching infeasible. As the data distribution among tasks changes according to the input data for each workload due to different data partitioning and scheduler behaviors in different environments (i.e., even for regularly-triggered repetitive workloads), it is unrealistic for any system to have an oracle view of the data distribution

throughout the execution in advance. The key challenge to address in this paper is thus to accurately estimate and optimize the potential recovery costs of evicted intermediate data during runtime.

Existing systems provide caching mechanisms composed of three separate operational layers, *caching*, *eviction*, and *recovery*, each of which behaves independently based on predefined conditions [7, 57, 60, 68, 71]. For caching, existing systems traditionally delegate caching decisions to expert users with sophisticated knowledge of each of the workloads, e.g., `PageRank` application [52]. These systems offer proper APIs to users but allow intermediate data to be managed at a coarse-grained dataset granularity, despite individual data partitions being the actual computation units for each parallel task [5–7, 60]. The *caching layer* blindly adheres to user annotations without considering whether or not each individual partition provides more significant caching benefits than others, and is prone to caching annotated data that incur minimal benefits or even have no future usage at all [15]. As a result, this approach often leads to inefficient utilization of memory space and inevitable cache misses.

Furthermore, the *eviction* and *recovery layers* in these systems are far from performance-optimized because they tend to be cost-agnostic and lack proper harmonization. For example, the decision regarding which intermediate data to *evict* upon memory shortages usually relies on heuristic methods that leverage past usage patterns, e.g., LRU [13] and LFU [32]. Such policies fall short in providing accurate results based on actual partition metrics due to the aforementioned challenges for accurately estimating the dynamically changing task data distributions. The *recovery* of evicted data can be achieved through either recomputation from its ancestor operators or storing data on multi-tiered storages (e.g., memory and disks) to read it back upon subsequent accesses, but the *potential recovery costs* of each method can vary significantly [55, 75]. Specifically, victim data may incur substantial disk I/O overheads if the data is oversized or conversely may result in high regeneration costs if a long and expensive sequence of recomputations is necessary. Unfortunately, current data analytics systems do not determine how to appropriately handle victims for performance and cost efficiency within the memory capacity.

To prevent inefficient memory utilization, Blaze introduces a caching mechanism that unifies the separate operational layers. In doing so, Blaze uses techniques that capture workload lineage through light-weight profiling of the actual workload on the target environment and applying inductive regression on dynamically monitored and updated metrics for individual partitions, which continuously influence potential recovery costs throughout the workload. By capturing the lineage, Blaze enables an automatic caching mechanism that identifies caching candidates at partition granularity based on their anticipated future references throughout the workload. Specifically, the metrics profiled for each partition include the time required to recompute the partition from its computational input, the actual size of the partition, and the current location or state of the partition. The unobserved metrics during profiling are derived by inductive regression based on the observed metrics.

Using these lineage and partition metrics, Blaze provides a potential recovery cost estimation model. This model empowers the system to estimate potential recovery costs, including the overheads of recomputation from the list of available cached data based on the lineage, as well as the disk I/O overheads that would be incurred if the partition were to be written to and read from the disk, in an on-line manner. Blaze's cost model solves for selecting optimal partition states that will result in the smallest potential recovery cost in the workload. The cost model is implemented as an integer linear programming (ILP) model, a popular solution for finding optimal values for minimization problems [37].

We implement Blaze on Apache Spark [7], with 6K lines of code. Our evaluations on 11 r5.xlarge AWS EC2 instances [3] show the performance improvements on two graph processing workloads, PageRank and Connected Components, as well as on four ML algorithms, including logistic regression (LR), K-means clustering (KMeans), gradient boosted tree regression (GBT), and singular value decomposition++ (SVD++). In the evaluations, Blaze accelerates the end-to-end execution time by $2.02 - 2.52\times$ compared to recomputation-based `MEM_ONLY` Spark, and by $1.08 - 2.86\times$ compared to checkpoint-based `MEM+DISK` Spark. Moreover, since Blaze efficiently caches intermediate data, it achieves an average reduction of 95% in cached data stored on disk compared to `MEM+DISK` Spark.

2 Background

2.1 Dataflow Execution Model

Modern data analytics systems adopt a dataflow-based execution model, which represents data processing jobs as directed acyclic graphs (DAGs) [46, 60, 68, 74], as illustrated in Fig. 1. By representing jobs as DAGs, data processing systems have been able to express and perform iterative computations cohesively [31, 46, 68, 74]. Concretely, in Spark [7, 74], vertices of a computational DAG represent resilient distributed datasets (RDD) and edges represent the computations that occur between the datasets. Thus, iterative computations can be represented as a form of having vertices with several outgoing edges that span different iterations. The computations can be categorized into *transformations* or *actions*, which each computes a dataset to derive another dataset or a workload result (e.g., statistics or a converged model), respectively. Each transformation lazily performs parallel computations (e.g., map, filter, join, groupByKey) to build intermediate data as new RDDs [74], whereas actions trigger computations and output results (e.g., collect, reduce).

Different computations have different overheads and resource usage patterns. For example, simple operators like

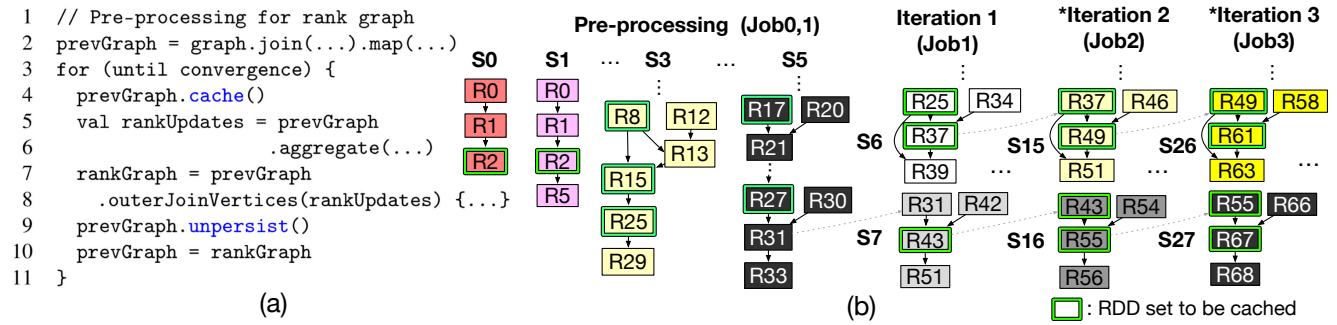

Figure 1. A simplified code snippet of a Spark PageRank [14] on GraphX [35] (a), and the RDD dependencies of the PageRank application (b). A Spark job is submitted for each iteration. Some stages, RDDs, and shuffle dependencies are omitted for simplicity. **Sx** denotes a stage number and **Rx** denotes the ID of an RDD.

map and filter use less resources (e.g., CPU, memory, network) compared to resource-heavy join or groupByKey operators [26, 58, 59]. Conventionally, a *job* acts as a unit of execution, which is triggered by an action and defined by the designated group of operators represented as a DAG [5, 7, 58–60, 68, 69, 71]. In iterative workloads, iterations are triggered as identically-shaped jobs, which are chained according to their dependencies on the input read job and previous iterations. While these jobs share their shape and computational logic across iterations, the underlying data partitions are different, leading to different memory consumption and data distributions across tasks (§2.2). Each logical dataset (i.e., RDD) can be annotated to be cached or unpersisted through user APIs provided by dataflow application semantics [4, 7, 46], incurring dataset state transitions throughout the execution.

2.2 Parallel Execution and Partition Sizes

A job consists of multiple *stages*, each of which is a pipeline of operators that can be performed on individual elements. Within a job, each logical dataset (i.e., RDD) is a set of multiple *partitions* that are processed by parallel computational *tasks*. Tasks describe the computational logic defined by the operators within a stage, and simultaneously produce computational results for the different partitions across a cluster of machines. Data processing engines schedule jobs in units of tasks on different machines, while also providing optimizations to leverage data locality by scheduling dependent tasks on equivalent machines [5–7, 60, 68]. Logically, stages have their boundaries on shuffle operators, which search and fetch elements of specific keys from each of the parent partitions (e.g., groupByKey), involving data transfer and aggregation over elements from multiple upstream tasks.

The actual computations defined by the operators are executed while materializing the intermediate data into objects in memory, and data (de)serialization is required if it requires disk access or data transfer across a network. Along the computations, partition sizes vary depending on the element keys that each partition is designated with, as one key may be overloaded compared to another. Consequently, task execution times also vary although parallel tasks perform identical

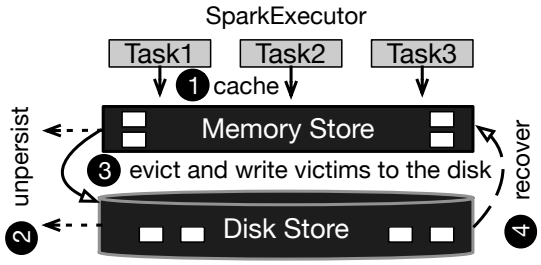

Figure 2. Caching and eviction on a Spark executor. Each task computes and caches RDD partitions into memory or disk within the executor that it is scheduled onto.

computations. Hence, bottleneck tasks are key to optimization as their dependent tasks require them to be completed before performing the following shuffle operations and computations. In iterative data processing, partitions of different jobs and stages have complex and repetitive data dependencies among each other, and particular intermediate data are reused over the iterations.

2.3 Caching Iterative Workloads in Existing Systems

Existing data processing systems provide user APIs for caching reusable data within memory or disks by annotating the datasets to *cache*, preparing them for potential invocations (Fig. 1(a)*L*4, Fig. 2❶). Once each dataset is no longer required in the workload, the user can also annotate them to be *unpersisted* and discarded from the system (Fig. 1(a)*L*9, Fig. 2❷). The system compliantly follows the annotations and performs caching and discarding in units of *datasets*. Upon caching, the system first checks whether there is enough space in memory, and evicts data according to the eviction policy if it requires additional space [13]. Data eviction occurs by unpersisting and discarding data (Fig. 2❷) or spilling data on disk (Fig. 2❸), according to the system settings (i.e., `MEM_ONLY`, `MEM+DISK`, §3.2), especially upon memory-heavy computations like join and groupByKey [26, 58]. Upon cache misses during the execution, the system recovers the data by fetching them from disk (Fig. 2❹), or regenerates the data in memory through recomputations, instructed by recursive fault-tolerance mechanisms of the systems [7, 74]. Data

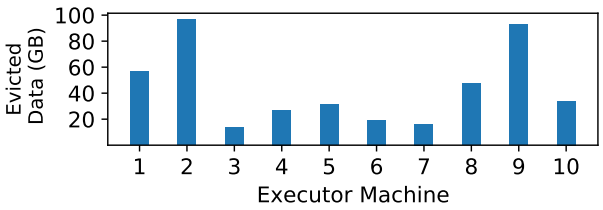

Figure 3. Caching at dataset granularity causes different sizes of evicted data among different executor machines on a PageRank application in our evaluation (§7).

recovery may also incur evictions to provide enough space in memory, and the recovered data can be cached again in memory according to the eviction policy. In short, runtime caching follows separate rules in a conditional manner on three different operational layers: `cache` and `unpersist` operations are performed by the user, evictions occur according to the eviction policy, and data is recovered by retrieving them from disks or recomputing them upon cache misses [7, 74].

3 Observation and Motivation

In this section, we observe the three separate operational layers of current caching mechanisms, composed of *caching*, *eviction*, and *recovery* layers, in more detail, and point out their limitations to motivate our work.

3.1 Caching and Eviction Mechanisms

Caching layer. As shown in Fig. 1(a)$L4$ and $L9$, caching interfaces are provided through `cache()` and `unpersist()` APIs for users to hint the datasets to persist after each iteration. For example, in Fig. 1(b) `Iteration 2`, we can see that as a result of caching annotations in Fig. 1(a), `R49` and `R55` are cached, while `R37` and `R43`, which were cached from the previous iteration, are unpersisted from disk. These caching and unpersisting happen repeatedly in the following iterations. Also, in addition to requiring manual efforts for deciding right datasets to cache, the current approaches fall short in providing fine-grained caching at partition granularity, although each partition varies in terms of size and computational time. This can become problematic as the current coarse-grained caching at dataset granularity causes executors to blindly cache partitions with different sizes (i.e. disk I/O overheads) and computation overheads.

To illustrate, we show the effects of the diverse caching overheads of the partitions in Fig. 3, which reveals two critical problems. First, we can directly see that coarse-grained caching leads to inconsistent amounts of evictions on different executors, despite the efforts of system schedulers to evenly distribute tasks among them [9]. This is mainly caused by unnecessary caching of certain partitions with smaller potential overheads, making inefficient usage of memory space and making the system prone to future evictions. For example, although the user only annotates caching for `rankGraph` in Fig. 1(a), it leads to caching of all consisting partitions of the RDDs in Fig. 1(b), while some partitions are unnecessary as they may not have any future usages or incur trivial recovery costs [36] (§7.2). Furthermore, we can also infer from these factors that caching overheads are often far from uniform across partitions, and cannot be easily predicted before the execution.

Eviction layer. Existing policy-based mechanisms for evicting and managing cache data show several limitations. Basically, an eviction policy keeps a list of partitions and determines the priority of partitions in which to evict from the cache storage. Many existing works focus on optimizing the eviction layer to improve the cache efficiency. A few different eviction policies include classic LRU (least recently used) [13] and GDWheel [44] policies, learning-based TinyLFU (lightweight least frequently used) [32] and LeCaR (learning cache replacement) [62] policies, as well as dependency-aware LRC (least reference count) [72] and MRD (most reference distance) [54] policies. As each name suggests, each policy determines the eviction priorities based on historical usage patterns and logical references, instead of on the specific metrics of the individual data partitions.

While these policies have been successful in handling caches in other contexts (e.g., CDN and web services) [19–23, 32, 44, 61, 62], eviction for data processing workloads requires consideration of many other factors. For example, for intermediate data partitions, evictions have to consider the future access patterns, different sizes, and the corresponding recomputation and disk I/O costs of the different partitions, in order to accurately capture the potential recovery costs. Nevertheless, the information regarding partition sizes, locations, and computation times only become available upon partitions being materialized in memory during the execution. Also, due to the challenges of varying data distribution and the memory usage of different executors according to the input data (Fig. 3), such factors cannot be easily predicted. As such information evolves dynamically over the iterations, it requires the system to be adaptive to the dynamically changing conditions.

3.2 Recomputation and Disk I/O Costs

While managing cache storage according to the caching policies, current systems *fix* their way (i.e., `MEM_ONLY` or `MEM+DISK`) for each workload in using the different storages for handling evictions. In other words, there is no flexibility to choose which way to take for the different datasets or partitions [7, 60, 68, 74]. This is primarily because existing approaches have not been able to utilize the dynamically changing potential recovery costs for their optimizations, due to their implementation designs and challenges for predicting and tracking the varying and dynamically changing partition data distribution among the tasks (§4.3). In short, `MEM_ONLY` mode discards data from memory upon evictions and depends on recursive recomputations, while `MEM+DISK`

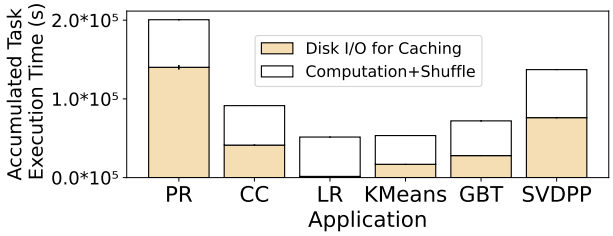

Figure 4. The accumulated execution time of tasks in six applications presented in our evaluation (§7), including the total time of disk I/O costs for recovering evicted data. Data (de)serialization is included in the disk I/O time.

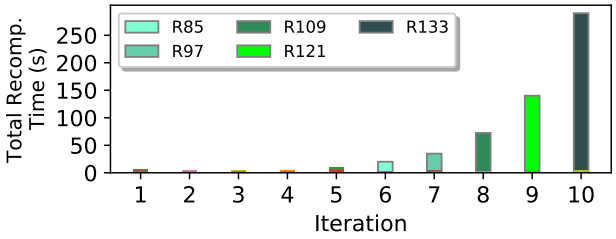

Figure 5. Breakdown of the total recomputation time for each iteration in PageRank (§7). The RDDs incurring the highest recomputation time within the iteration are labeled from iteration 6 to 10 (RDD 85, 97, 109, 121, and 133).

mode keeps a multi-tiered storage for eviction to recover data by fetching the checkpointed data from the slower secondary storages [2]. There also are ways to (de)serialize data in memory or in off-heap space to save some memory, but the basic mechanism is similar to `MEM_ONLY` mode in that they recover data through recomputations. While these two ways recover data by incurring different potential recovery costs, the costs are not comparably uniform or deterministic, making it difficult to calculate which method is better than the other [55, 75].

Disk I/O costs. Disk I/O costs are incurred upon writing and reading cached data to and from underlying storages (e.g., SSDs, HDDs) to spill and refetch the data for evicting and recovering data in `MEMORY_AND_DISK` mode. We can see in Fig. 4 that disk I/O overhead is the major source of bottleneck in two graph processing (i.e., PageRank and ConnectedComponents) and several machine learning (i.e., SingularValueDecomposition++, GradientBoostedTrees) applications, where the detailed experimental setup is described in §7. The disk I/O costs also exhibit the overhead for (de)serializing data in memory to access them on disks. Obviously, if the partitions are larger in size, it would incur more disk I/O costs. Therefore, applications with large partition sizes incur more disk I/O costs than other applications, especially like PageRank where disk costs compose more than 70% of the end-to-end execution time.

Recomputation costs. Recomputation costs are the computational time incurred when a partition requires upstream ancestor operators to recursively produce their intermediate data in order to derive the desired result. As shown in Fig. 5, computations with longer lineages in later iterations tend to incur more recomputation costs. While it can be easily sought that it would be more advantageous to use disks as secondary storages to store cache data, we can see in Fig. 4 that LogisticRegression is the only workload that produces small disk I/O overheads due to the relatively smaller size of the cached data (i.e., ML model) and fewer datasets that are annotated to cached. In other cases, recomputation costs may be smaller than disk costs, and they should be considered as an option for some partitions to take for recovering the intermediate data, instead of simply spilling them on disks.

4 Considerations for Cost-aware Caching

In this section, we describe our design goals and the challenges to achieving the ideal case for caching in comparison to the existing mechanisms.

4.1 To Cache, or Not To Cache?

First of all, instead of blindly caching all data that is annotated to be cached as in existing systems [7, 13], we aim to first determine whether or not it would be advantageous to cache the data in memory (Fig. 2❶). We must first keep in mind that only the partitions with *future usages* should be considered, as the others will occupy memory space without any benefit. For a reused partition p_8, it is obvious that it is advantageous to cache it if there is enough memory space to store p_8. However, in cases where memory space is constrained, it is advantageous to cache data in memory only if the p_8 is to incur more potential recovery costs than other cached partitions that are already in memory. If the potential recovery costs are not considered, it results in evictions of some partitions that will eventually incur more costs in the future, which is undesirable. Therefore, upon trying to cache a partition p_8, we compare its potential recovery costs against other available cached partitions, and also consider the options to directly discard them or write them on disk if the benefits of storing in memory are limited. This consideration is taken both when a partition first attempts to be cached, as well as when the partition is recovered through a cache miss and becomes a candidate for caching again.

4.2 To Evict, or Not To Evict?

In cases where it requires some evictions of cached partitions to store an expensive partition p_8, we aim to carefully select the partitions to evict based on the potential recovery costs, instead of on history-based caching policies [13, 32, 49, 62, 72]. Since each partition incurs a different recomputation and disk cost, it is also important to choose in which state to evict and keep each of the partitions, as discussed in §3.2. For some partitions, it may be advantageous to simply discard the data and recompute them, if they are too oversized compared to their recomputation overheads (Fig. 2❷). For others, it may be more advantageous to spill and store them on disk,

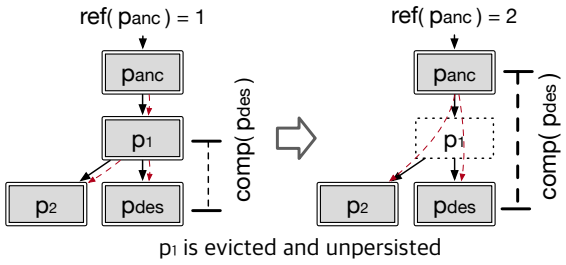

Figure 6. The dynamically changing comp(p_{34B}) and ref($p_{0=2}$) upon evicting and unpersisting p_1 from the cache.

if they have smaller partition sizes while their computations have been more time-consuming (e.g., model calculation for LR) (Fig. 2❸). Both aspects are carefully considered in our solution for choosing and evicting partitions from memory, along with the considerations for the potential recovery costs in the calculations, to perform data recovery in a timely manner (Fig. 2❹).

4.3 Dynamically Changing Data Dependency

While it is possible to find an optimal caching solution with prior knowledge of the partition sizes and the recomputation times, an accurate off-line analysis and prediction is extremely difficult to achieve. This is because even if the workload is repeatedly run on a daily or a weekly basis, the data distribution and the partition sizes vary with different input data. Moreover, even with an accurate estimation of the data distribution, the potential recovery costs dynamically change during runtime. At one point in time, a partition can be in memory, while at another point it can be evicted and discarded or written on disk. For example, in Fig. 6, if p_1 is unpersisted, the recomputation cost for p_{34B} will increase from $p_1 \rightarrow p_{34B}$ to $p_{0=2} \rightarrow p_1 \rightarrow p_{34B}$, if $p_{0=2}$ resides in the cache. This recomputation cost can be extended even further from the source input data if the required partitions are not in the cache. Also, future dependencies can also dynamically change upon unpersisting partitions. When trying to compute for p_{34B} and p_2, it initially does not incur any references to $p_{0=2}$, but after unpersisting p_1, $p_{0=2}$ is referenced by both p_{34B} and p_2 through p_1. Such evictions and unpersist decisions often cause dynamic chained reactions in the potential recovery costs over the progress of the workload, as the cached partitions also change dynamically. Therefore, it is exceptionally difficult to derive and consider all of the different cases to calculate the potential costs.

5 Blaze Design

In this section, we describe an overview of how our system operates and our design principles in formulating a universal cost model for caching. Next, we describe how we estimate and keep track of the partition metrics throughout the workload. Finally, we formulate our cost model for optimizing the memory space and describe our solution for finding the optimal caching state for each of the partitions. Our idea is applicable to all caching-enabled distributed data processing systems based on dataflow graphs with parallel tasks and partitions [5, 40, 46, 60, 68, 74].

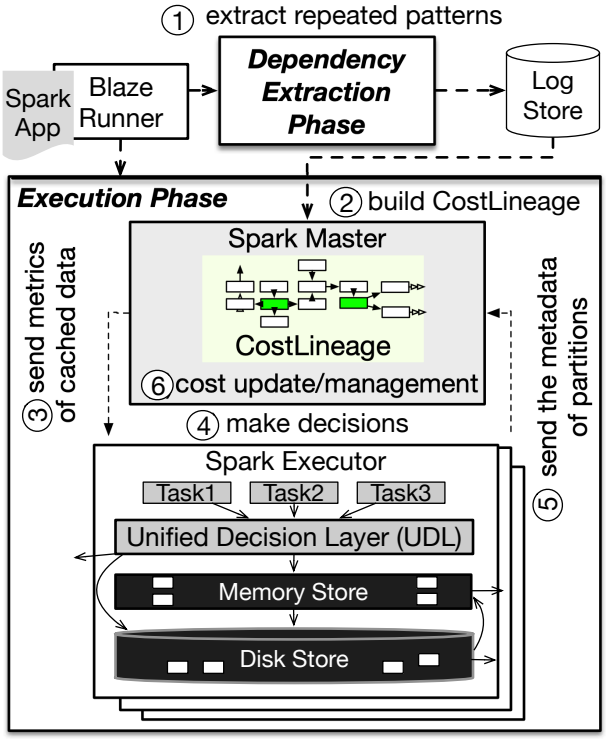

Figure 7. The overview of Blaze.

5.1 Blaze Overview

Blaze performs cost optimization based on the different partition states and metrics, and on the potential recovery costs derived from them. In order to track and accurately estimate the potential recovery costs, Blaze performs the following actions, as shown in Fig. 7. First, ① Blaze runs the workload on a small portion of the original input data (i.e., $\ddot{Y}$ 1MB) to extract and capture the code path and dependencies between datasets (i.e., DAG structure), and ② builds a CostLineage that keeps track of the workload lineage and performs inductions on future *partition metrics* based on the extracted partition dependencies and existing metrics (§5.3). Next, ③ Blaze sends the metrics to the executor and performs estimated calculations for the potential recovery costs for each partition on the CostLineage (§5.4). Based on the calculated costs, ④ Blaze automatically makes unified decisions for caching, eviction, and recovery based on our ILP-based solution (§5.5, §5.6). Once a task finishes its execution for a particular partition, ⑤ the executor sends the metadata of the new partitions back to the master to ⑥ dynamically update and manage the partition metrics back on the CostLineage with timely information on the run.

5.2 Design Principles

Our key idea for addressing the limitations of existing approaches is to devise a unified cost-aware caching mechanism that automatically decides the desired state of each partition. In Blaze, this state indicates whether to keep the cache data of a particular partition $p_i \in P$, delineated within each dataset abstraction (e.g., RDD), in memory (m_i), on disk (d_i), or to simply discard and unpersist (u_i) the partition, based on our cost estimation for the potential overheads. The state of each partition p_i can thus be defined as follows:

$$\forall p_i \in P,\ m_i + d_i + u_i = 1\ (m_i,\ d_i,\ u_i \in \{0,\ 1\}) \qquad (1)$$

As indicated, only one of these variables becomes 1 at any point in time, and this acts as a constraint to the partition state throughout the optimization. State transitions among the cached partitions can occur as evictions ($m_i \rightarrow u_i$, $m_i \rightarrow d_i$), recomputations ($u_i \rightarrow m_i$, $u_i \rightarrow d_i$), and recovery from disks ($d_i \rightarrow m_i$). Disks may also unpersist data ($d_i \rightarrow u_i$), in cases where the disk size is also constrained.

A potential recovery cost of a partition is the overhead that may occur in the future if the partition does not reside in memory at the execution time. Concretely, we estimate the potential disk access cost of p_i at time t, $cost_3(p_i, t)$, along with the potential recomputation cost of partition p_i at time t, $cost_A(p_i, t)$ (§5.4). If the $cost_A(p_i, t)$ is smaller than $cost_3(p_i, t)$, discarding and recomputing p_i would be more beneficial than writing it on disk, as it reduces the high (de)serialization and disk read/write time while incurring a small recomputation time. Therefore, assuming that we have abundant disk space for caching, the ideal potential recovery cost $cost(p_i, t)$ of the partition, if not cached in memory, would be the minimum between the two values:

$$cost(p_i,\ t) = \min(cost_3(p_i,\ t),\ cost_A(p_i,\ t)) \qquad (2)$$

If the cached partition p_i resides in memory (i.e., $m_i = 1,\ d_i = 0,\ u_i = 0$), the potential recovery cost is disregarded, while our aim is to keep the sum of all potential recovery costs, $\sum_{p_i \in P - M} cost(p_i, t)$, as low as possible for all partitions that do not reside in memory ($p_i \in P - M$), where $M = \{p_i \in P \mid m_i = 1,\ d_i = 0,\ u_i = 0\}$ ($\because Eq.\ 1$).

5.3 The CostLineage for Tracking Partition Metrics

In order to dynamically keep track of the partition metrics, Blaze first builds a CostLineage based on the workload DAG produced by the initial dependency extraction phase before the actual execution. Since the input data is minuscule (i.e., $\dot{Y}$ 1MB), the dependency extraction phase usually succeeds in capturing the dependency among the multiple iterations of the workload DAG until its convergence within its timeout (i.e., 10 sec). Even if it fails to do so, Blaze is able to perform inductions on future iterations based on the already captured iterations, which we describe later in this subsection. Also, Blaze can derive the number of potential references for each of the partitions until the end of the application based on the dependencies, which are used for automatic caching (§5.5).

As shown in Fig. 8, the CostLineage dynamically detects and merges duplicate dataset abstractions from different jobs together based on their IDs to manage them as a single abstraction. For example, R37 from iterations 1 and 2 are merged together in the CostLineage. On the captured data dependencies, CostLineage annotates the list of profiled partition sizes and their states on the vertices (i.e., datasets) and the computation times on the edges between each of the dependent partitions along the execution. Concretely, as the initial iterations of the actual workload do not request for evictions due to the sufficient memory space in the early stages of execution, the partition metrics for the initial iterations can be simply recorded on the CostLineage without the requirement for cache optimizations. Along the execution, executors materialize iterators of the partition data and gain access to the actual partition sizes, locations, and computation times. These metadata are continuously updated on the CostLineage to reflect up-to-date information for the partition metrics. Also, if the partition data is read or written to disk, Blaze also measures the time it takes for the disk operations and derives the disk throughput to keep the hardware performance metrics with timely information.

Once the metrics of the initial iterations are recorded, CostLineage detects the congruent datasets that play the same role in different iterations by analyzing the DAG structure and the sizes of the datasets from the initial iterations. Concretely, CostLineage takes the sum of partition sizes for each dataset and uses a simple pattern searching algorithm based on the differences in the dataset sizes of adjacent operators to find the repeated patterns [29]. By doing so, we can detect the iterative patterns among datasets that are generated from the same code path in the loop, and perform inductive regression to predict the trend of partition sizes for future iterations. Concretely, for the missing values of partition metrics in the CostLineage, Blaze inductively fills in temporarily approximated values of metrics for the undiscovered partitions by applying a lightweight linear regression model based on the existing metrics from previous iterations throughout the application [51]. Likewise, future iterations that hadn't yet been captured during the profiling phase can be inducted similarly. Based on these partition metrics, we can estimate the potential costs for recomputation and disk overheads of the different partitions whenever a caching decision needs to be made.

5.4 Potential Recovery Cost Estimation

We describe how we estimate $cost_3(p_i, t)$ and $cost_A(p_i, t)$ individually. Simply put, the cost is calculated in units of seconds to predict the time it potentially takes to recover data for future invocations. The disk cost, $cost_3(p_i, t)$, can be calculated simply by dividing the size of the partition $size(p_i)$ by the profiled read/write throughput of the disk,

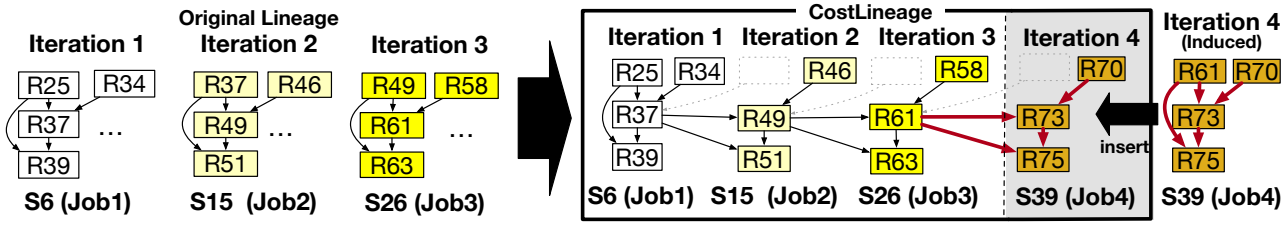

Figure 8. A `CostLineage` constructed from the extracted RDD lineages of the PageRank application in the dependency extraction phase. Duplicate RDDs are dynamically detected and merged upon new iterations and future iterations are induced.

$throughput_{SSB:}$, which can be profiled within the system during runtime or initially approximated through conventional softwares [34]:

$$cost_3(p_8,\ t) = \frac{size(p_8)}{throughput_{SSB:}} \quad (3)$$

The recomputation cost for a partition p_8 has to be defined recursively with respect to the ancestor upstream partitions that are not cached in memory $A_8 = \{p_: \in P - M \mid p_: \rightarrow p_8\}$. We define the longest recomputation time from the upstream partitions as the recomputation cost, $cost_A(p_8, t)$ (§3.2), which dynamically changes according to the `CostLineage`:

$$cost_A(p_8,\ t) = \max_{?_k \in\ _i} ((1 - m_:) \cdot cost(p_:,\ t) + cost_{:\rightarrow 8}) \quad (4)$$

where $cost_{:\rightarrow 8}$ is the computation time for generating p_8 from $p_:$ and $cost(p_:,\ t)$ is defined in Eq. 2, only effective when $p_:$ is not in memory (i.e., $m_: = 0,\ while\ Eq.\ 1$).

As the logics in Eq. 3 and Eq. 4 are respectively made up of simple arithmetic and requires a shallow stack of recursive computations on the recorded metrics, we can compute the potential recovery costs for both cases within milliseconds.

5.5 Finding the Optimal Partition States

Based on the collected dependencies and the partition metrics on our `CostLineage` (§5.3), we can now formulate our solution to minimize the sum of potential recovery cost (i.e., time) as an integer linear programming (ILP) model, with respect to our cost estimation methods (§5.4). Recognizing that a job corresponds to an iteration in iterative workloads, assume that we wish to optimize our cache storage for a set of future partitions within a set of jobs, $p_9 \in J$. We can set up a constraint for our memory space for all relevant partitions, $p_8 \in P$, that are recorded on our `CostLineage`, for our ILP objective function to minimize the potential costs for the partitions that are to be used in our upcoming jobs $p_9 \in J$. In our solution, we set the boundary for the set of jobs J to be the current job and its successive job, inferred by the `CostLineage` and the system metrics on the workload progress [9] to keep the ILP overhead under a performance boundary (i.e., $\ddot{Y}$ 5 seconds). While the ILP cost may seem non-negligible, Blaze hides away the overhead by carefully determining when to trigger optimizations to produce punctual results for seamless workload execution, which is further described in §5.6. As a result, potential costs related to disk I/O and recomputation are adaptively minimized for the near future, regardless of the workload progress in the current job:

$$\text{Minimize} \quad \tilde{O}_{?_i \in} (d_9 \cdot cost_3(p_9,\ t) + u_9 \cdot cost_A(p_9,\ t)) \quad (5)$$

$$\text{Subject to:} \quad \tilde{O}_{?_i \in \%} size(p_8) \cdot m_8 \leq capacity_{<4<},$$
$$Eq.\ 1,\ Eq.\ 2,\ Eq.\ 3,\ Eq.\ 4 \quad (6)$$

In cases where disk space is also constrained, the ILP can be simply extended by adding another constraint to Eq. 6, $\tilde{O}_{?_i \in \%} size(p_8) \cdot d_8 \leq capacity_{SSB:}$, where we set $capacity_{SSB:}$ as an abundant value in this paper.

5.6 Automatic Caching

Instead of relying on user annotations, Blaze attempts to *automatically cache* partitions after each stage execution, if the partition has future references in the `CostLineage` and is expected to be reused in the future. Upon caching, Blaze determines whether caching a partition would reduce more potential recovery cost than the already cached partitions that reside in memory, by comparing the size of the partitions and their expected potential costs. As the `CostLineage` dynamically keeps track of the partition states and metrics, it can easily find the potential recovery costs of the cached partitions. Similarly, if the partition does not have any future usages, Blaze automatically unpersists the data from the cache storage to quickly acquire free space after each stage execution, similar to Nectar [36].

Note that automatic caching and unpersists consider the full application DAG captured by the `CostLineage`, whereas the ILP considers the potential costs only for a couple of upcoming iterations (jobs) to optimize ILP performance with an upper bound and trigger state transitions only for the near future. The ILP solver is triggered whenever a new job is submitted and auto-caching is triggered whenever a stage is completed, and partitions are subsequently migrated or unpersisted. By doing so, ILP can solve for caching decisions for the upcoming job before it actually arrives, and the ILP overheads can be hidden. Through these steps, Blaze automatically decides on the caching, eviction, and recovery of the partitions on a unified layer in each executor, according to the derived optimal state of the partitions.

6 Blaze Implementation

Blaze is implemented on top of Spark 3.3.2 with around 6K lines of Scala 2.12 code, and the Blaze runner is implemented with 500 lines of bash script. In order to track the computation time and sizes of partitions, each RDD implementation is modified with code for profiling. The `BlazeRPCEndpoint` receives and updates these metrics on our `CostLineage` and `CostAnalyzer`, maintained in the `BlazeBlockManagerEndPoint` in the Spark master, and the ILP implementation uses this data to solve for the ILP, which is asynchronously triggered by Bash scripts. With the ILP results, the `(Unified)MemoryManager` [17], `MemoryBlockManager`, and `DiskBlockManager` interact with the `MemoryStore` [15] and `DiskStore` components to apply the results on the partitions cached in the local executor memory. Each task caches its partitions into the executor where it is scheduled, without storing and sending the partitions to other executors, as most tasks access the cached data on local executors with the locality-aware task scheduling optimizations implemented on Spark [9]. The ILP solver is implemented with the Gurobi optimizer 10.0.1 [37]. Our implementations can be similarly reproduced in other systems by modifying the components of the data plane and implementing an online metric tracker for the individual data partitions, accompanied by an ILP solver.

7 Evaluation

In our evaluation, we observe the system performance of Blaze compared to other caching mechanisms (§7.2), distinguish the factors that contribute to the Blaze performance improvement (§7.3), and provide additional details on specific settings and Blaze components (§7.4, §7.5).

7.1 Methodology

Environment. All evaluations are executed on 11 r5a.2xlarge (8 vCPU, 64GB memory, and 10Gbps network) AWS EC2 instances, where one is reserved for the master and the other ten are for the executors. A 100GB SSD (gp2) is used as a disk caching store in each instance. Each instance runs 2 executors, each with 25GB executor memory, which totals up to 20 executors with a total of 500GB executor memory in our evaluation. As the size of the memory that the system uses to store caches in each executor cannot be defined as a fixed value [17], we empirically consent on the upper bound of aggregate memory store capacity as 170 GB by observing that `Spark` uses up to 170 GB (i.e., 34%) of the total executor memory for caching for all applications in our evaluation. Although we set the total memory sizes in our environment to be reasonable towards the input dataset sizes in our evaluation workloads, it would require larger proportions of memory capacities on the cluster to execute workloads on larger datasets.

Workloads. We evaluate two graph processing and four machine learning applications, which are widely-used representative iterative applications that benefit from caching of RDDs in the execution. In all of the applications, the peak amount of cached data exceeds the cache memory size of Spark. For each application, we report an average of three results of the execution. For all systems aside from Blaze that require manual caching and unpersist decisions, we follow the caching decisions implemented on Spark GraphX [35] and MLlib [50] libraries [8, 10–12, 14, 16].

- **PageRank (PR):** PR is a graph processing algorithm that calculates the importance of web pages. Web pages are represented as vertices, and connectivities between them are represented as edges [52]. We generate a synthetic dataset with 25 million vertices using SparkBench [45].
- **Connected Components (CC):** CC is another graph processing algorithm that finds all connected components in a given graph [28]. We use the same input dataset as in PR.
- **Logistic Regression (LR):** LR is a basic regression algorithm in machine learning [42]. For input data, we use the criteo dataset [30] day 0 data, which is 106 GB, among the 24 days of data.
- **Gradient Boosted Trees (GBT):** GBT is an ensemble learning method that combines multiple decision trees to create a strong predictive model for classification and regression [70]. In the GBT workload, we utilize 50GB of data generated by HiBench [38] in the LibSVM format.
- **Singular Value Decomposition++ (SVD++):** SVD++ is an extension of SVD, which is a machine learning algorithm that uses matrix factorization for recommendations, such as for recommending new movies based on user preferences [43]. We generate a synthetic 31 GB input dataset with a rating data of 15 million users, each with 50 items.
- **K-means Clustering (KMeans):** K-Means is an unsupervised learning algorithm used for clustering data into groups [47]. For the K-Means workload, we employ 90GB of data generated by HiBench [38] based on uniform distribution.

Systems. We compare Blaze against the performance of the workloads on the following systems.

- **`MEM_ONLY` Spark:** Spark abides by the `cache` and `unpersist` annotations provided by users with a least-recently-used (LRU)-based eviction policy by default. Spark runs on the `MEM_ONLY` mode by default, which unpersists cached data upon evictions and performs recomputations to recover data on cache misses. We use Spark 3.3.2.
- **`MEM+DISK` Spark:** Spark provides the `MEM_AND_DISK` mode, which enables the system to use two-tiered storage for storing evicted cache data onto secondary storage like disks, to later recover data by reloading them from disks, instead of by recomputing them, on cache misses. This mode also follows caching annotations provided by users with the LRU-based eviction policy.

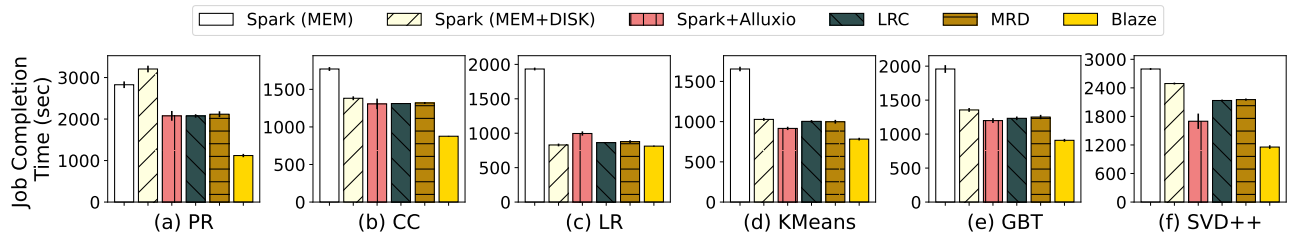

Figure 9. An end-to-end performance comparison on `MEM_ONLY Spark`, `MEM+DISK Spark`, `Spark+Alluxio`, `LRC`, `MRD`, and `Blaze` in various applications. We run each application three times and plot the average with an error bar at the top.

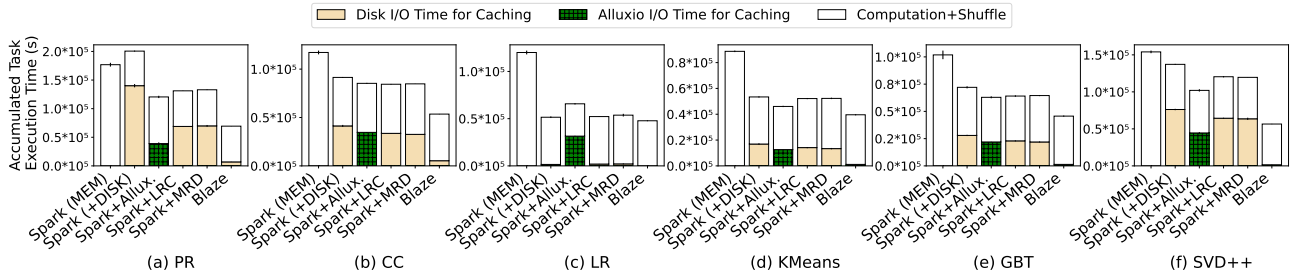

Figure 10. A breakdown of cost with the accumulated total task execution times. In `MEM+DISK Spark` (annotated as `Spark (+DISK)`), `LRC`, and `MRD`, the disk I/O time of cached data becomes the cost. In `Spark+Alluxio`, the Alluxio I/O time of cached data becomes the cost, as they are the potential recovery cost experienced from the applications that are run on Spark.

• **`Spark + Alluxio`:** Alluxio [2] is a widely used tiered distributed storage for data analytics systems. As an external caching store, Alluxio optimizes the placement of cached data between its fast (i.e., memory) and slow tiers (i.e., disks) while transparently exposing them to the client side. Spark+Alluxio also represents other variants of the `MEM_AND_DISK` Spark (e.g., `MEMORY_AND_DISK_SER` and `OFF_HEAP`), as it provides serialization to reduce the size of cached data in memory, with additional disk support. We integrate Alluxio v2.9.1 on Spark v3.3.2, where all cached data are written to and read from Alluxio. We configure the Alluxio memory tier for it to use the same amount of memory that Spark uses for its memory store, and co-locate Alluxio and Spark on the same cluster for its best performance.

• **`LRC` and `MRD`:** Among numerous works that optimize eviction policies through conventional algorithms and those that exploit the data dependency information on dataflow lineages, we choose LRC (Least Reference Count) [72] and MRD (Most Reference Distance) [54] as representative ones. The considered conventional caching algorithms include LRU, FIFO, LFUDA [18, 48], GDWheel [44], TinyLFU [32], and LeCaR [62], and data dependency-aware algorithms include LERC [73], LCRC [63], and LCS [33]. The conventional algorithms exhibit limitations in capturing future information and show marginal improvements, if any, to the default LRU algorithm, which exhibits limited performance compared to the dependency-aware algorithms. Among the dependency-aware algorithms, we selectively compare the ones with the best performances in our evaluations: LRC and MRD. LRC evicts data with the smallest reference count, which is the number of future references in RDD lineages. MRD evicts data with the largest reference distance, which is the number of stages left until being referenced, and prefetches data with the smallest reference distance whenever free space becomes available in the executor memory. Unlike Blaze, which captures application-wide dependencies during the dependency extraction phase, they only use the dependency information provided by the currently submitted job, without utilizing the complete knowledge of the partition metrics and their dependencies across multiple jobs. LRC and MRD on `MEM+DISK Spark` are evaluated in §7.2, and on `MEM_ONLY Spark` are evaluated in §7.4.

Terms. In order to reduce the confusion regarding jobs and completion times, we use the term *application completion time (ACT)* to describe the end-to-end completion times, instead of job completion time (JCT), in our evaluations. Also to distinguish the evictions to disks and by unpersisting, we use the terms *eviction* (*to disk*) to represent the state $m_i \rightarrow d_i$ and *unpersist* to represent the states $m_i \rightarrow u_i$ and $d_i \rightarrow u_i$ (§5.2). The accumulated task execution times are the sum of the execution times among all of the tasks from all the jobs in the particular applications.

7.2 Performance Analysis

Fig. 9 shows the end-to-end ACT for all workloads in various systems. Note that for all results of Blaze, the time taken for the dependency extraction phase and performing inductive methods are included in the measurements, which takes up $\dot{Y}$ 4% of the total ACT. Overall, Blaze achieves 2.52×, 2.02×, 2.38×, 2.11×, 2.15×, and 2.42× speed up compared to `MEM_ONLY Spark`, and 2.86×, 1.57×, 1.08×, 1.31×, 1.49×, and 2.15× speed-up compared to `MEM+DISK Spark` in PR, CC, LR,

KMeans, GBT, and SVD++, respectively. The key reason for the performance improvement comes from auto-caching and the unified decision layer of Blaze that significantly reduces the recomputation time and the aggregate disk I/O time. As shown in Fig. 10, while MEM_ONLY Spark does not exhibit any disk usages, Blaze reduces the disk I/O overhead by 95%, 87%, 99%, 97%, 97% and 98% for the accumulated value for all tasks on MEM+DISK Spark in PR, CC, LR, KMeans, GBT, and SVD++, respectively. This indicates that Blaze uses the fixed memory space efficiently as a caching store.

The potential benefit of the unified cost-aware caching and eviction decisions of Blaze is distinct in cases where the disk I/O overhead dominates the ACT. For example, Blaze achieves the highest speed-up of the end-to-end execution time compared to MEM+DISK Spark in PR (Fig. 9 (a)). This is because, in PR, the aggregated disk I/O time takes 70% of the accumulated total task execution time of MEM+DISK Spark, which is the largest percentage among all applications (45%, 3%, 32%, 39%, and 56% in CC, LR, KMeans, GBT, and SVD++, respectively).

The main reason for PR having the largest disk I/O overhead in MEM+DISK Spark is that its working set size is much larger than other applications. As the working set size increases, more amounts of data are written to disk, and this results in higher disk I/O overheads. For PR, the average total size of data on disk reaches 306 GB (peak 427 GB) in MEM+DISK Spark, whereas that of CC, LR, and SVD++ reaches 220 GB (peak 335 GB), 41 GB (peak 122 GB), and 45 GB (peak 98 GB), respectively. While the executor disk capacity is abundant (i.e., 1000GB) to host all spilled data in our evaluations, we can see that the performance of MEM+DISK Spark is worse compared to MEM_ONLY Spark due to the large disk I/O overheads. On the other hand, Blaze significantly reduces the amount of data on disk compared to MEM+DISK Spark; by 83%, 81%, 100%, 96%, 96%, and 97% in PR, CC, LR, KMeans, GBT, and SVD++, respectively. This is mainly due to the timely removal of data with smaller potential recovery costs on Blaze, which eliminates unnecessary disk I/O overheads caused by evictions to disk for the data that incur small recomputation overheads. Blaze writes data to disk only when its recomputation overhead is larger than its disk I/O overhead, which reduces the aggregate disk write time for the data with future usages.

In LR, the speed-up of Blaze is 1.08× compared to MEM+DISK Spark (Fig. 9 (c)) which is relatively small, because the main bottleneck comes from the computation, and not the disk I/O overhead, as LR exhibits fewer references to the cached data and smaller ML model sizes. Unlike other applications, LR only caches a total of three RDDs for each iteration, where only one of them is actually referenced to be reused later on. As Blaze automatically captures this fact through CostLineage, it prevents unnecessary disk I/O overhead and incurs no evictions at all. Other solutions blindly adhere to the caching annotation, and incur disk I/O overheads. While disk I/O overheads are relatively small in LR, this eventually incurs evictions from the unnecessary caching and inefficient memory usage in MEM_ONLY and MEM+DISK Spark. This causes large recomputation overheads in MEM_ONLY Spark and contributes to the 2.38× speedup in Blaze. LRC and MRD policies successfully capture future references within the job, and perform relatively well (1.06× speedup in Blaze for both cases) by avoiding misguided eviction decisions, but still incur disk I/O overhead for evicting the unnecessary data on disks. This also contributes to the reason for Spark+Alluxio performing worse compared to MEM+DISK Spark in LR, because Spark+Alluxio requires additional data (de)serialization overheads in memory, to read and write data through Alluxio [2].

Interestingly, the amount of cached size of SVD++ is smaller than CC in MEM+DISK Spark, but its disk I/O time takes 56% of the accumulated total task execution time, which is larger than that of CC. We observe that the average time for serializing a partition in SVD++ is $2.5 - 6.4\times$ larger than that of others, as the serialization overhead differs across different data types. Still, due to the smaller model sizes, MEM+DISK Spark shows better performance compared to MEM_ONLY Spark. In short, the main bottleneck of SVD++ comes from the data serialization time, which contributes to disk I/O overheads, and this is the main reason for SVD++ displaying large disk I/O overheads even when the amount of cached data is small.

In the case of KMeans and GBT, these workloads present larger model sizes compared to LR, yet they demonstrate smaller disk I/O overheads compared to SVD++. The differences are caused by the fact that while the KMeans model involves more centroids and cluster assignments along the iterations, GBT requires larger models due to its complex tree structures. Due to these factors, KMeans, GBT, and SVD++ show similar trends among the different baselines, while they exhibit differences in their intensity due to their differences in their recomputation and disk I/O overheads caused by the model sizes and the computational logic.

Compared to the MEM+DISK Spark that adapts *LRC* and *MRD* policies, Blaze achieves up to 1.8× speed-up, mainly because such eviction policies only optimize the eviction layer among the three layers (i.e., caching, eviction, and recovery layers), while Blaze incorporates the separate operational layers together in its solution. Moreover, as existing dependency-aware policies only consider the data dependency of the current job, they are prone to underestimating the future numbers of data references that are to be reused across future jobs. Also, they often face situations where multiple partitions have the same reference counts or reference distances, in which case they arbitrarily break the tie between the potential victims, without considering the fact that the different partitions are likely to incur largely different disk I/O overheads.

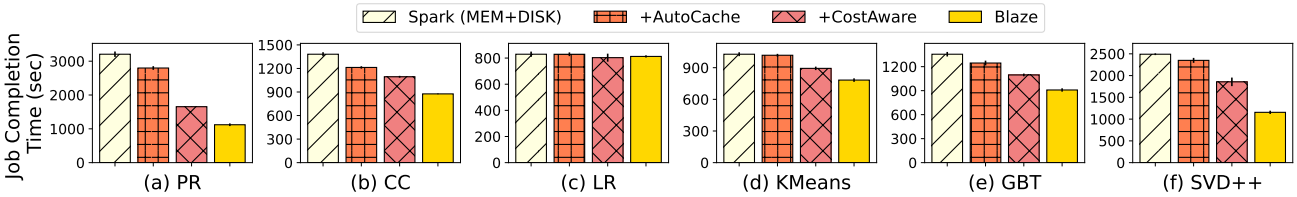

Figure 11. A performance breakdown for Blaze.

7.3 Performance Breakdown

In this section, we provide a detailed breakdown of the performance gain achieved by Blaze through Fig. 11. For the breakdown, we implement the following two cases on top of MEM+DISK Spark with individual components of Blaze:

- **+AutoCache** automatically caches and unpersists individual partitions based on future usages after each stage completion like Blaze, on top of MEM+DISK Spark, instead of adhering to user annotations in the caching layer. This option does not consider the potential recovery costs.
- **+CostAware** performs the cost-aware eviction like Blaze along with the auto-caching enabled, which additionally selects the victim partitions from the memory based on the sorted potential recovery costs for disk I/O overheads in the eviction layer. It includes the Blaze cost model for evictions to disks but excludes the option to recompute data for recovery, as well as the ILP solution.

Note that **Blaze** incorporates the AutoCache, CostAware mechanisms, and also the ILP caching decision solution that solves for the minimum potential recomputation and disk I/O costs, on top of MEM+DISK Spark.

+AutoCache vs. MEM+DISK Spark. Comparing +AutoCache against MEM+DISK Spark shows the effectiveness of the automatic caching and unpersisting mechanism of Blaze in the caching layer. +AutoCache accelerates the ACT by 1.15×, 1.14×, 1.08×, 1.01×, 1.08×, and 1.06× in PR, CC, LR, KMeans, GBT, and SVD++, respectively. +AutoCache in LR already consumes all of the 1.08× speed-up, as it successfully prevents the disk I/O overheads incurred by the evictions and recovery of the models in MEM+DISK Spark. With +AutoCache, the automatically-selected working set of potentially-referenced cached data in LR fits in memory during the iterations, as discussed in §7.2. In KMeans, auto-caching has limited improvement, due to the uniform distribution and thus smaller skews among partitions. In PR, CC, GBT, and SVD++, auto-caching improves system performance due to the following two reasons. First, it selects a smaller number of partitions to cache compared to the annotation-based and coarse-grained caching techniques on Spark. Concretely, auto-caching selectively caches partitions from 26 and 33 RDDs in PR and CC, whereas Spark caches 28 and 36 RDDs as a whole. The cached number of RDDs is identical in both cases for GBT and SVD++, but fine-grained caching reduces the amount of cached data. Second, as auto-unpersisting timely removes RDDs without future references at the end of each stage execution, it allows the system to quickly acquire free space, increasing the effective memory store space before having to find the user annotation to unpersist data. Consequently, this reduces the inefficient usage of memory space and the unnecessary disk write overheads caused by evictions of unused data.

+CostAware vs. +AutoCache. Comparing +CostAware against +AutoCache shows the effectiveness of the potential recovery cost model for disk I/O overheads, specifically within the eviction layer. Compared to +AutoCache, the +CostAware accelerates the ACT by 1.69×, 1.11×, 1.14×, 1.14×, and 1.27× in PR, CC, KMeans, GBT, and SVD++. The key reason for the performance improvement of applying the cost model comes from the reduced disk I/O overheads of evicted data, by selecting victim partitions with the smallest disk access costs. While LR does not benefit from the cost model in this experiment, the potential benefit is noticeable in cases where the size of the working set exceeds the available memory store capacity, as shown with PR, CC, KMeans, GBT, and SVD++.

Blaze vs. +CostAware. Comparing Blaze against +CostAware shows the effectiveness of the ILP caching decision solution on Blaze. Compared to +CostAware, Blaze further accelerates the ACT by 1.47×, 1.25×, 1.14×, 1.21×, and 1.61× in PR, CC, KMeans, GBT, and SVD++, respectively. The main difference between Blaze and +CostAware is two-fold. First, +CostAware always caches data in memory or writes data on disk regardless of the costs of the data to be cached, as it does not compare the costs before caching (§4.1). In contrast, Blaze unifies the caching decision for all partitions, and caches data in memory only when the cost of the data to cache is larger than the costs of the potential victims already in memory. This way, Blaze prevents the case of caching data with low cost at the expense of evicting data with high cost. Second, Blaze writes data on disk only when its potential recomputation cost is larger than the potential disk access cost, which reduces the potential disk I/O overhead. Also, the memory space is used more efficiently and effectively, as it successfully solves for the partition states in which it incurs the minimum potential recovery costs for near-future executions.

With all of the optimization combined, Blaze exhibits 2.86×, 1.57×, 1.08×, 1.31×, 1.49×, and 2.15× speed-up in ACT compared to MEM+DISK Spark in PR, CC, LR, KMeans, GBT, and SVD++, respectively.

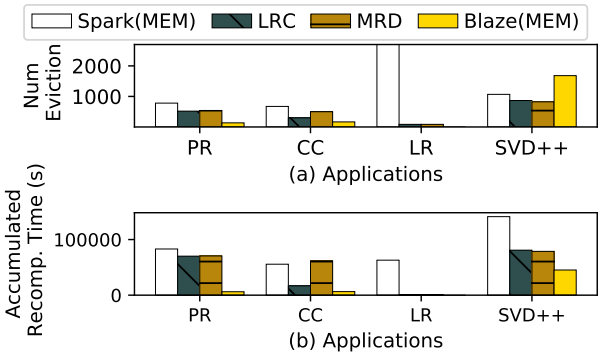

Figure 12. The number of evictions and total recomputation time of evicted RDDs while only using memory.

7.4 Number of Evictions and Recomputation Time

Fig. 12 illustrates the number of evictions and recomputation time of evicted partitions on Blaze without disk support and `MEM_ONLY Spark`, along with its variants that use LRC and MRD policies as the eviction policy. Blaze still shows performance improvements with its auto-caching and cost-aware eviction mechanisms, while demonstrating limited application, as it excludes the potential disk I/O costs from consideration within its solution. Especially for LR, Blaze does not incur any eviction, as the cached partitions fit in memory by automatically caching only the partitions with future references (§7.2). LRC and MRD are also successful in capturing the cached data with future references within the current job in LR, but evictions still occur as it also caches and evicts the unreferenced data to abide by the user annotations. In contrast, `MEM_ONLY Spark` incurs a large number of evictions, as blindly caching three RDDs exceeds the memory capacity, and the LRU policy results in frequent recomputations. For SVD++, while Blaze incurs more evictions than other systems in terms of number, the total recomputation time is only 32% compared to `MEM_ONLY Spark`, showing that Blaze successfully captures the potential recovery costs within its mechanism. For PR and CC, even though Blaze does not use disks and incurs some additional overheads, it successfully captures the potential recomputation overheads and manages to efficiently use the memory capacity to incur minimum potential overheads in the workload.

7.5 Profiling Overhead vs. Benefits

In order to analyze the performance benefit against the overhead for profiling, we show the comparison with and without the initial profiling phase for dependency extraction in Fig. 13. Without the dependency extraction across jobs, the profiling overhead can be avoided, but the cost of RDDs referenced in the future jobs can be underestimated and evicted, as Blaze can miss the potential usages of the partitions into reflection. Consequently, enabling the profiling phase accelerates the completion time by up to 1.64× compared to the approach that builds the application lineage on the run,

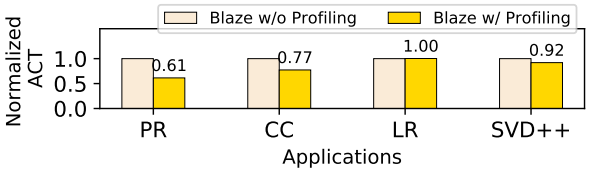

Figure 13. The normalized ACT of Blaze with and without dependency profiling, including the profiling overhead.

as shown in Fig. 13. The profiling bases its execution on a $\dot{Y}$ 1MB data from the original input load, and the overhead is upper-bounded by the 10 second timeout, which takes up $\dot{Y}$ 2% of the total execution time in our evaluations. Profiling plays a key role in providing automatic caching and estimating the potential costs for longer downstream lineages within the workload. Especially, profiling is more beneficial for applications where many partitions are referenced across multiple jobs (PR and CC). The benefit of profiling in LR is limited because it only has a single RDD in each iteration that is referenced within the jobs.

8 Related Work

Cost-Aware Caching. Cost-aware caching is a widely-used approach to optimize caching and eviction in various fields including web services [21, 23], in-memory key-value stores [20, 44], and CDNs [19, 22], where defining the cost metric that can properly capture the needs of various workloads plays a key role in achieving performance gain. However, little has been known about how to adopt cost-aware caching for iterative data analytics. Blaze defines the cost metric by identifying the key factors specific to the context of iterative workloads, where decisions based on the metrics successfully bring end-to-end performance improvements.

Exploiting Data Dependencies for Data Analytics. There are various approaches that exploit the data dependencies of data analytics applications to optimize prefetchings [1, 54] and evictions [33, 54, 63, 65, 66, 72, 73]. However, existing works limit optimization opportunities, as they primarily focus on optimizing the eviction layer among the three layers that consist of the caching mechanism: caching, eviction, and recovery layers. Blaze unifies the decisions for caching, eviction, and recovery of data in a single decision layer and provides automatic caching decisions based on the tracked information. Also, compared to the cost metric of Blaze that captures the actual performance penalties with future reference, computation, data size, and disk access time, utilizing only the data dependencies inside a job as a metric for eviction decision has many limitations in achieving end-to-end performance improvements, as shown in §7.

Computation Sharing across Applications. In datacenters, there are many works on sharing computations across different applications to improve the application performance [27, 36, 41, 56]. Their primary goal is to decide

on the data to keep and share in the cluster caching store, which is large enough to keep them all. Unlike such works, the primary goal of Blaze is to minimize the potential overheads caused by evictions and cache misses in an application across the memory or two-tiered storages with disks, in cases where the memory store is limited to fit all of the data to cache. Therefore, Blaze optimizes not only to decide on the data to cache, but also to select where to keep the data: in memory, on disk, or simply to unpersist them.

GPU Memory Management. Recent deep learning (DL) works research on optimizing tensor placements by deciding on the data to keep in GPU memory, evict to CPU memory, or to unpersist and recompute [25, 53, 64, 67, 76]. Although their approach is similar to Blaze, the cost metric and decision algorithm of Blaze is tailored for handling challenges that are more general than for DL-specific workloads. Blaze is tailored for any general distributed iterative data analytics workloads by efficiently managing and updating the estimated costs for a large number of parallel partitions.

9 Conclusion

Blaze provides an automatic caching mechanism, that unifies the separate operational layers of existing caching methods together (i.e., caching, eviction, and recovery layers), to adaptively provide optimal caching decisions at any time within iterative data processing workloads. Blaze bases its caching decisions on the dynamically-updated `CostLineage`, which is initially built by extracting data dependencies through profiling, and inductively updated and predicted on-the-run based on the partition metrics measured along the actual execution over the iterations. By automatically choosing the partitions with future references to cache, and calculating the potential costs with the partition metrics collected on `CostLineage`, Blaze successfully captures and derives the optimum states for the cached data in the way that it minimizes the sum of potential costs within the workload with an ILP-based solution. Our evaluations show that Blaze speeds up end-to-end performance by up to 2.86× and optimizes the cache data by 95% on average with optimized automatic caching compared to conventional caching mechanisms.

Acknowledgments

We thank our shepherd David Cock and the anonymous reviewers for their feedback. This work was supported by the Research Fund of Samsung Electronics DS Division, the 2023 Research Fund (1.230019) of UNIST, and the IITP grant funded by the Korea government (MSIT) (No.2020-0-01336, Artificial Intelligence graduate school support (UNIST); No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Mania Abdi, Amin Mosayyebzadeh, Mohammad Hossein Hajkazemi, Ata Turk, Orran Krieger, and Peter Desnoyers. 2019. Caching in the Multiverse. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/abdi

[2] Alluxio, Inc. 2023. Alluxio - Data Orchestration for the Cloud. https://www.alluxio.io

[3] Amazon Web Services, Inc. 2023. Amazon AWS. https://aws.amazon.com

[4] Apache Software Foundation. 2023. Apache Beam. https://beam.apache.org.

[5] Apache Software Foundation. 2023. Apache Flink. https://flink.apache.org/.

[6] Apache Software Foundation. 2023. Apache Nemo. https://nemo.apache.org.

[7] Apache Software Foundation. 2023. Apache Spark. https://spark.apache.org.

[8] Apache Software Foundation. 2023. Spark Connected Components. https://github.com/apache/spark/blob/v3.3.2/graphx/src/main/scala/org/apache/spark/graphx/lib/ConnectedComponents.scala.

[9] Apache Software Foundation. 2023. Spark DAG Scheduler. https://github.com/apache/spark/blob/branch-2.4/core/src/main/scala/org/apache/spark/scheduler/DAGScheduler.scala.

[10] Apache Software Foundation. 2023. Spark Gradient Boosted Trees. https://github.com/apache/spark/blob/v3.3.2/mllib/src/main/scala/org/apache/spark/ml/tree/impl/GradientBoostedTrees.scala.

[11] Apache Software Foundation. 2023. Spark K-means Clustering. https://github.com/apache/spark/blob/v3.3.2/mllib/src/main/scala/org/apache/spark/ml/clustering/KMeans.scala.

[12] Apache Software Foundation. 2023. Spark Logistic Regression. https://github.com/apache/spark/blob/v3.3.2/mllib/src/main/scala/org/apache/spark/ml/classification/LogisticRegression.scala.

[13] Apache Software Foundation. 2023. Spark LRU Eviction. https://spark.apache.org/docs/latest/rdd-programming-guide.html#removing-data.

[14] Apache Software Foundation. 2023. Spark PageRank. https://github.com/apache/spark/blob/v3.3.2/graphx/src/main/scala/org/apache/spark/graphx/lib/PageRank.scala.

[15] Apache Software Foundation. 2023. Spark RDD Block Eviction. https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/storage/memory/MemoryStore.scala#L434

[16] Apache Software Foundation. 2023. Spark SVD++. https://github.com/apache/spark/blob/v3.3.2/graphx/src/main/scala/org/apache/spark/graphx/lib/SVDPlusPlus.scala.

[17] Apache Software Foundation. 2023. Spark Unified Memory Manager. https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala

[18] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating Content Management Techniques for Web Proxy Caches. *SIGMETRICS Perform. Eval. Rev.* 27, 4 (mar 2000), 3–11. https://doi.org/10.1145/346000.346003

[19] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. 2020. Caching with Delayed Hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 495–513. https://doi.org/10.1145/3387514.3405883

[20] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann

[21] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 195–212. https://www.usenix.org/conference/osdi18/presentation/berger

[22] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 483–498. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/berger

[23] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 499–511. https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein

[24] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

[25] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. (2016). arXiv:1604.06174 [cs.LG]

[26] Brian Cho and Ergin Seyfe. 2019. Taking advantage of a disaggregated storage and compute architecture.

[27] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1205–1223. https://www.usenix.org/conference/osdi20/presentation/chung

[28] Fan Chung and Linyuan Lu. 2002. Connected components in random graphs with given expected degree sequences. *Annals of combinatorics* 6, 2 (2002), 125–145.

[29] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[30] CriteoLabs. 2023. Terabyte Click Logs. https://labs.criteo.com/2013/12/download-terabyte-click-logs-2

[31] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters

[32] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage* 13, 4, Article 35 (nov 2017), 31 pages. https://doi.org/10.1145/3149371

[33] Yuanzhen Geng, Xuanhua Shi, Cheng Pei, Hai Jin, and Wenbin Jiang. 2017. LCS: an efficient data eviction strategy for spark. *International Journal of Parallel Programming* 45, 6 (2017), 1285–1297.

[34] Sebastien Godard. 2023. iostat. https://github.com/sysstat/sysstat

[35] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[36] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/nectar-automatic-management-data-and-computation-datacenters

[37] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[38] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

[39] Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu. 2019. Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 191–206. https://www.usenix.org/conference/atc19/presentation/huang

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys*. 59–72.

[41] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 191–203. https://doi.org/10.1145/3183713.3190656

[42] David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. 2002. *Logistic Regression: A Self-Learning Text*. Springer.

[43] Yehuda Koren. 2008. Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Las Vegas, Nevada, USA) *(KDD '08)*. Association for Computing Machinery, New York, NY, USA, 426–434. https://doi.org/10.1145/1401890.1401944

[44] Conglong Li and Alan L. Cox. 2015. GD-Wheel: A Cost-Aware Replacement Policy for Key-Value Stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/2741948.2741956

[45] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '15)*. Association for Computing Machinery, New York, NY, USA, Article 53, 8 pages. https://doi.org/10.1145/2742854.2747283

[46] Google LLC. 2023. Google Cloud Dataflow. https://cloud.google.com/dataflow

[47] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[48] Dhruv Matani, Ketan Shah, and Anirban Mitra. 2021. An O(1) algorithm for implementing the LFU cache eviction scheme. (2021). arXiv:2110.11602 [cs.DS]

[49] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache

[50] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[51] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. 2021. *Introduction to linear regression analysis*. John Wiley & Sons.

[52] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[53] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. https://doi.org/10.1145/3373376.3378505

[54] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. 2018. Reference-Distance Eviction and Prefetching for Cache Management in Spark. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP '18)*. Association for Computing Machinery, New York, NY, USA, Article 88, 10 pages. https://doi.org/10.1145/3225058.3225087

[55] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 423–432. https://doi.org/10.1109/MICRO.2006.49

[56] Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, and Carlo Curino. 2019. SparkCruise: Handsfree Computation Reuse in Spark. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1850–1853. https://doi.org/10.14778/3352063.3352082

[57] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. 2015. 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* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1357–1369. https://doi.org/10.1145/2723372.2742790

[58] Won Wook Song, Myeongjae Jeon, and Byung-Gon Chun. 2022. SWAN: WAN-Aware Stream Processing on Geographically-Distributed Clusters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems* (Virtual Event, Singapore) *(APSys '22)*. Association for Computing Machinery, New York, NY, USA, 78–84. https://doi.org/10.1145/3546591.3547524

[59] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 301–314. https://www.usenix.org/conference/atc23/presentation/song

[60] Won Wook Song, Youngseok Yang, Jeongyoon Eo, Jangho Seo, Joo Yeon Kim, Sanha Lee, Gyewon Lee, Taegeon Um, Haeyoon Cho, and Byung-Gon Chun. 2021. Apache Nemo: A Framework for Optimizing Distributed Data Processing. *ACM Trans. Comput. Syst.* 38, 3–4, Article 5 (oct 2021), 31 pages. https://doi.org/10.1145/3468144

[61] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/nsdi20/presentation/song

[62] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[63] Bo Wang, Jie Tang, Rui Zhang, Wei Ding, and Deyu Qi. 2018. LCRC: A Dependency-Aware Cache Management Policy for Spark. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*. 956–963. https://doi.org/10.1109/BDCloud.2018.00140

[64] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3178487.3178491

[65] Luna Xu, Min Li, Li Zhang, Ali R. Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. MEMTUNE: Dynamic Memory Management for In-Memory Data Analytic Platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 383–392. https://doi.org/10.1109/IPDPS.2016.105

[66] Yinggen Xu, Liu Liu, and Zhijun Ding. 2020. DAG-Aware Joint Task Scheduling and Cache Management in Spark Clusters. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 378–387. https://doi.org/10.1109/IPDPS47924.2020.00047

[67] Donglin Yang and Dazhao Cheng. 2020. Efficient GPU Memory Management for Nonlinear DNNs. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 185–196. https://doi.org/10.1145/3369583.3392684

[68] Youngseok Yang, Jeongyoon Eo, Geon-Woo Kim, Joo Yeon Kim, Sanha Lee, Jangho Seo, Won Wook Song, and Byung-Gon Chun. 2019. Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 177–190. https://www.usenix.org/conference/atc19/presentation/yang-youngseok

[69] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 575–588. https://doi.org/10.1145/3064176.3064181

[70] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. 2009. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 2061–2064.

[71] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 1–14.

[72] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. 2017. LRC: Dependency-aware cache management for data analytics clusters. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2017.8057007

[73] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled B. Letaief. 2017. LERC: Coordinated Cache Management for Data-Parallel Systems. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. 1–6. https://doi.org/10.1109/GLOCOM.2017.8254999

[74] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[75] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. 2017. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 377–392. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang

[76] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-Based GPU Memory Footprint Reduction for LSTM RNN Training. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 1089–1102. https://doi.org/10.1109/ISCA45697.2020.00092