Ph.D. DISSERTATION

Efficient and Adaptive Resource Management for Dynamically Optimizing Distributed Data Processing Systems

효율적이고 유연한 자원 관리를 통한 분산 데이터 처리 시스템 성능의 동적 최적화

August 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Won Wook SONG

Efficient and Adaptive Resource Management for Dynamically Optimizing Distributed Data Processing Systems

효율적이고 유연한 자원 관리를 통한 분산 데이터 처리 시스템 성능의 동적 최적화

지도교수 전 병 곤

이 논문을 공학박사 학위논문으로 제출함

2023년 6월

서울대학교 대학원

컴퓨터공학부

송 원 욱

송원욱의 공학박사 학위논문을 인준함

2023년 6월

위 원 장	황 승 원	(인)
부위원장	전 병 곤	(인)
위 원	윤 성 로	(인)
위 원	강 유	(인)
위 원	전 명 재	(인)

Abstract

Today, large amount of data are being generated on numerous heterogeneous machines around the globe. Due to its the large sizes of the data in terms of number and volume, big data processing increasingly demands for higher throughput performances in order to extract business-critical information while meeting the service level objective (SLO) requirements. While the target data for distributed processing can be categorized into large-scale batch data or real-time data streams, both types of data possess different characteristics that can be unpredictable and quickly alter in its nature. In order to dynamically optimize to the different environments, there exists different cases where it requires efficient and adaptive management for different resources including *CPU*, *network*, and *memory*.

For example, real-time event streams of data can always change in its volume, as there can be incalculable random events that can cause the data traffic to increase, which requires dynamic adaptation in *CPU* power to increase the engine throughput. In another case, data that have to be processed could be scattered around the globe, and requires them to be gathered during the data analytics job through heterogeneous and unstable long-distance *networks*, for them to be summarized into particular statistics in order to timely provide the user with useful information. Moreover, in cases of dealing with iterative workloads that accumulate large amounts intermediate data, like machine learning or graph processing, one may optimize the workload through caching reusable data, where management of *memory* resources is crucial to timely provide the job with its cached data while preventing recomputation overheads whenever

required. If these conditions and environments for the data are not handled, it causes massive performance losses upon facing such situations. Moreover, these problems are commonly unpredictable and alters dynamically during runtime.

In order to deal with the unpredictable resource problems, this dissertation proposes *dynamic* resource management techniques that makes *efficient* and *adaptive* use of resources from cloud environments to overcome resource shortages and bottlenecks in terms of *CPU*, *network*, and *memory*, with mathematical modeling and analytical approaches, through systems called Sponge, SWAN, and Blaze. Sponge provides fast dynamic adaptation for stream workloads to overcome shortages in CPU resources by acquiring resources from serverless instances to provide the system with additional CPU at a sub-second latency under situations where the input load increases sporadically and instantaneously. SWAN dynamically measures and analyzes the different bandwidth capacities of heterogeneous network connections to find the optimal path and operator placement among the operators to mitigate the limited network resources and ensure that the data efficiently flows from one place to another in a globally scattered environment. Blaze provides automatic caching mechanisms based on live tracking of partition metrics and sophisticated predictions on cache usages and overheads to efficiently use limited memory resources for caching in a timely manner for iterative data processing workloads.

Our evaluations show that the dynamic resource management methods significantly improve system performance in terms of throughput, latency (i.e., for streaming workloads), and end-to-end completion time (i.e., for batch workloads) compared to existing systems, by up to 5.64× increase, 88% reduction, and 2.86× speed-up, respectively, by providing the distributed data processing systems with sufficient resources at all times and by efficiently using the limited resources with regard to the dynamically changing environments.

Keywords: Distributed Systems, Big Data, Machine Learning, Cloud Computing, Resource Management, Scheduling

Student Number: 2017-28182

For my family and friends, for all the love and support

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Resource Management in Distributed Data Processing

Today, large amount of data are produced from numerous heterogeneous machines distributed across the globe. Such data are often processed by data analytics workloads, especially as a distributed workload, in order to refine the vast amount of data to make business decisions and to provide various statistics. Due to the increasing number and volume of such data, the input data and the cluster environment where we run our distributed data processing systems are becoming more and more diversified with different characteristics. In order to deal with the sudden changes in the environments of distributed data processing systems, it is important to adaptively provide the system with sufficient computational resources for them at all times to provide satisfying system performances.

While there are different runtime components that contribute to the per-

formance of distributed data analytics, *CPU*, *network*, and *memory* are the computational resources that play a key role in determining the performances in distributed systems. First of all, *CPU* directly affects the throughput performance, as it is related to the number of computations that a machine can perform for a specific period of time. Next, *network* affects the performance while transferring data between operators on different machines, especially for shuffle operations, which is an operation that transfers data to the following aggregate task related to a particular key into a particular aggregated intermediate data (§ 2.2). Finally, *memory* affects the performance for aggregating intermediate data and operator states for aggregative operators in stream workloads, as well as for caching and providing reusable input and intermediate data to downstream operators in batch workloads. In modern data processing, we face many diverse situations where we require the system to dynamically manage the computational resources for different cases.

For example, streaming workloads, which deal with data that is generated in real-time, has to deal with data volumes that are often unpredictable and change rapidly in volume [108, 107]. To deal with these fluctuations, current systems aim to dynamically scale in and out their *CPU* resources, to redistribute and migrate computing tasks across a cluster of machines. In another case, wide-area stream analytics is commonly being used to extract operational or business insights from the data issued from multiple distant datacenters. However, timely processing of such data streams is challenging because wide-area *network* (WAN) bandwidth is scarce and varies widely across both different geo-locations (i.e., spatially) and points of time (i.e., temporally). Lastly, modern data processing workloads, such as machine learning and graph processing, involve iterative computations to create models that converge into higher accuracy. An effective caching mechanism is crucial to expedite iterative com-

putations since the intermediate data that need to be stored in memory grows larger over time, often exceeding the *memory* capacity.

In order to solve these problems, many prior works have focused on reducing the overhead of system reconfiguration and state migration on pre-allocated cluster resources to redistribute *CPU* resources, these approaches still face significant challenges in meeting latency SLOs at low operational costs, especially upon facing unpredictable bursty loads. On the other hand, stream analytics desirable under a WAN setup requires the consideration of path diversity and the associated bandwidth from data source to sink when performing operator task placement for the query execution plan. It also has to enable fast adaptation to dynamic resource conditions, e.g., changes in *network bandwidth*, to keep the query execution stable. Lastly, to provide caching mechanisms in *memory*, existing systems handle intermediate data through separate operational layers (e.g., caching, eviction, and recovery), with each layer working independently in a cost-agnostic manner. These layers typically rely on user annotations and past access patterns, failing to make globally optimal decisions for the workload.

In order to dynamically adapt our *CPU* resources, we propose Sponge, a new stream processing system that enables fast reactive scaling of long-running stream queries by leveraging serverless framework (SF) instances. Sponge absorbs sudden, unpredictable increases in input loads from existing VMs with low latency and cost by taking advantage of the fact that SF instances can be initiated quickly, in just a few hundred milliseconds. Sponge efficiently tracks a small number of metrics to quickly detect bursty loads and make fast scaling decisions based on these metrics. Moreover, by incorporating optimization logic at compile-time and triggering fast data redirection and partial-state merging mechanisms at runtime, Sponge avoids optimization and state migration overheads during runtime while efficiently offloading bursty loads from existing

VMs to new SF instances. Our evaluation on AWS EC2 and Lambda using the NEXMark benchmark shows that Sponge promptly reacts to bursty input loads, reducing 99^{th}-percentile tail latencies by 88% on average compared to other stream query scaling methods on VMs. Sponge also reduces cost by 83% compared to methods that over-provision VMs to handle unpredictable bursty loads.

Next, to overcome unstable long-distance *network* conditions, we present SWAN, a WAN stream analytics engine that incorporates two key techniques to meet the aforementioned requirements. First, SWAN provides a fast heuristic model that captures WAN characteristics at runtime and evenly distributes tasks to nodes while maximizing the network bandwidth for intermediate data. Second, SWAN exploits a stream relaying operator (or RO) to extend a query plan for better facilitating path diversity. This is driven by our observation that oftentimes, a longer path with more communication hops provides higher bandwidth to reach the data sink than a shorter path, allowing us to trade-off query latency for higher query throughput. SWAN stretches a given query plan by adding ROs at compile time to opportunistically place it over such a longer path. In practice, throughput gains do not necessarily lead to significant latency increases, due to higher network bandwidth providing more in-flight data transfers. Our prototype improves the latency and the throughput of stream analytics performances by 77.6% and 5.64×, respectively, compared to existing approaches, and performs query adaptations within seconds.

Finally, to overcome the limitations regarding *memory* shortage for caching in iterative jobs, Blaze introduces a unified caching mechanism that integrates the separate operational layers together. Blaze dynamically captures the workload structure and metrics using profiling and inductive methods, and automatically estimates the potential data caching efficiency associated with different

operational decisions based on the profiled information. By analyzing the workload, Blaze incorporates the potential data recovery costs across stages into a single cost optimization function, which informs the optimal state for each individual partition. This fine-grained approach reduces the significant disk I/O overheads caused by oversized partitions and the recomputation overheads for partitions with long lineages, while maximizing the efficient utilization of memory space. Our evaluations demonstrate that Blaze can accelerate end-to-end application completion time by up to 2.86× and reduce cache data stored on disk by 90% on average compared to Spark.

In the following subsections, we introduce the individual problems in more detail.

1.1.1 Dynamic CPU Resources

Stream queries continuously process real-time data to extract insights and make business-critical decisions, such as analyzing real-time logs to extract statistics, detect anomalies, and provide notifications [7, 68, 23, 111, 120]. Latency is an essential service level objective (SLO) in these streaming workloads, as faster up-to-date results mean more value. Stream systems are expected to run 24/7 while meeting their SLOs [121].

Meanwhile, stream systems regularly face significant challenges due to sudden, unpredictable bursts of input loads caused by random events, e.g., influencer tweets, breaking news, and natural disasters [108, 107]. These bursts can abruptly increase the input load by more than 10× in just a few seconds [62, 44, 131, 89, 30]. If stream processing systems do not quickly acquire additional computing resources that can handle the bursty loads and do not promptly redistribute the load to the newly allocated computing resources, events will soon pile up on the existing resources, leading to cascading impacts

on query latencies that can have fatal consequences such as reduced user satisfaction and revenues [111].

One approach to quickly acquiring additional computing resources is to over-provision resources. Existing work such as Rhino [45], Megaphone [59], and Chronostream [130] builds efficient stream load redistribution mechanisms by harnessing over-provisioned resources to minimize latency spikes on load bursts. For instance, Megaphone [59] smoothly migrates stream query loads to extra resources during stable load in preparation for load spikes. However, over-provisioning solutions can be *costly* and inefficient, as a significant amount of resources will remain idle for most of the time.

Cloud services can reduce operational costs by offering on-demand resource allocations. Existing scaling approaches for on-demand resources dynamically migrate stream operator instances, in units of parallel *tasks*, to the allocated on-demand virtual machines (VMs). They redistribute the tasks and their states, which are key-value pairs of aggregated intermediate results [25, 39, 112, 40, 47, 84]. However, migrating tasks and their states incurs extra overheads (e.g., (de)serialization), which increase proportionally to the state size (e.g., a large number of key-value pairs), and can violate low latency SLOs. Moreover, using VMs, which are popular on-demand cloud resources, can further exacerbate latency spikes due to the considerable launch delay of VM instances which can take dozens of seconds (i.e., 25-30 secs) with conventional cloud providers [49, 102, 71].

1.1.2 Dynamic Network Resources

With the surging demand for global services, service providers are increasingly demanding wide-area stream data analytics in order to extract information from the global data generated from multiple distant datacenters [76]. For example,

global services need a real-time log processing system for monitoring systems from thousands of distant servers to ensure their SLOs [64, 83]. Also, global services like Twitter [76] need to process distant data in real-time to keep track of global news and social media. Many of such applications often require processing the data with high throughput and low latency as extracting timely information means more value for service providers.

A wide-area analytics system is typically composed of multiple geo-distributed edge clusters and datacenters connected by wide-area networks (WAN) [57, 61, 105, 140]. In this setup, the variability and unpredictable nature of WAN bandwidths make it challenging to achieve both high throughput and low latency. WAN exhibits diverse levels of peer-to-peer (P2P) bandwidths depending on the geo-location, each of which can change in the order of minutes [140, 127]. Essentially, a streaming engine for WAN analytics should be adaptive to these spatial and temporal variability of network bandwidths.

Prior works that consider the spatial variability of networks under wide-area data analytics often focus on short-lived batch processing while reducing network data transfers to lower the network budget and assume that network bandwidth is relatively stable throughout the query execution [101, 123, 125]. On the other hand, existing WAN-aware stream processing systems that run long-running streaming queries try to perform centralized processing after adaptively collecting data to a single data center. Such an approach requires users to trade-off the query output accuracy for performance through pre-aggregation, degradation, and statistical approximation [58, 105, 140], especially when transmitting a large volume of raw data under limited network bandwidths. Despite their great effectiveness, these approaches are frequently application-dependent and may not apply generally. For queries that require high accuracies, such as fraud detection and billing queries, any loss in the query output accuracy may

result in undesirable reliability or additional costs. Moreover, determining the right accuracy-performance trade-off typically relies heavily on the expertise of the analyst and requires parameter tuning for each of the different workloads, which may be cumbersome.

1.1.3 Dynamic Memory Resources

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 [96], while logistic regression has proven effective in forecasting probabilities of certain events across numerous fields [17]. Many of such applications exhibit multiple iterations of repetitive operations that evolve the model into high accuracy [51, 88, 37, 60]. For these workloads, caching plays a key role in system efficiency by reusing intermediate data to avoid recomputations on their repeated usages.

However, these iterative computations result in a consistent expansion of intermediate data, putting a strain on memory resources [117]. Unfortunately, simply provisioning ample memory to accommodate all intermediate data is not a silver-bullet solution given that data size can increase more than 10× the input size over the iterations (§7.3.2). Moreover, while each of the iterations has a declarative job structure, such jobs are submitted until convergence iteration-by-iteration, which makes the overall lineage of the workload unpredictable before the actual execution. As the memory size is fixed and the workload lineage expands, it causes intermediate data to be evicted from memory and the state of intermediate data to dynamically change within the fixed memory space along the iterations, posing a challenge in accurately estimating the potential recovery costs of evicted intermediate data.

Existing systems provide caching mechanisms composed of three separate operational layers that individually behave upon predefined conditions: *caching*, *eviction*, and *recovery* layers [14, 115, 135, 136, 110]. For caching, existing systems traditionally delegate caching decisions to expert users with sophisticated knowledge of each of the workloads, like the aforesaid `PageRank` application. 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 [115, 135, 13, 14, 39]. With the *caching layer* that blindly adheres to user annotations without considering whether or not each individual partition provides more significant caching benefits than others, some annotated data may incur minimal benefits from caching or even have no future use at all [20]. As a result, this approach often leads to inefficient utilization of memory space and inevitable cache misses.

Furthermore, when memory falls short, existing systems typically *evict* cached data and *recover* it upon subsequent access. 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 usually relies on heuristic methods that leverage past usage patterns, e.g., LRU [18] and LFU [46]. The recovery of evicted data can be achieved either through recomputation from its ancestor operators or input data or data swapping on multi-tiered storages, but the *potential recovery costs* of each method can vary significantly[104, 142]. 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 handle victims for performance and cost efficiency within the memory capacity.

1.2 Proposed Solutions

We suggest the following methods to overcome the problems mentioned above. First, we suggest our mathematical model that orchestrates the process of dynamically supplying CPU resources to our existing VMs through serverless frameworks. Second, we suggest methods for profiling and probing for network paths with diverse locations and lengths to find efficient network paths in the global scale of network connections. Third, we suggest dynamic tracking and calculation of the potential costs to find the caching solution that minimizes the future expectations of recomputation and disk overheads.

1.2.1 Dynamic Supply of CPU Resources through Serverless Frameworks

In this dissertation, we design Sponge, a new stream processing system that requires low operational costs and keeps low latency upon sudden bursty loads [114]. Sponge is designed with the following three design principles:

Combining two heterogeneous cloud resources to have the best of both worlds: Sponge harnesses two heterogeneous cloud resources: VMs and serverless function (SF) instances. Serverless solutions provided by conventional cloud providers [62, 44, 131, 89, 30] only take hundreds of milliseconds (i.e., 300-750 ms) to launch and prepare and are designed to achieve high scalability, while the operational costs are much higher than those of VMs. Therefore, to achieve low latency and low operational costs, Sponge uses VMs for processing stable streaming loads for longer periods of time, while quickly invoking SF instances and using them for short periods of time to handle bursty loads. If the bursty input loads persist, we may consider launching new VMs to permanently offload the tasks with existing state migration techniques [112, 40, 47, 84, 67, 130, 106,

59, 45, 53]. In such cases, on-demand SF instances can be used to accomplish system SLOs by hiding the launch overhead during the preparation of the new VM instances.

Keeping tasks with high migration overheads on VMs, while quickly redirecting data to SFs: When VMs process streaming data with stable loads over long periods of time, the states of stream tasks are materialized, and the state size may increase on the existing VMs. To avoid the state migration overheads from VMs to SFs, Sponge incorporates the *redirect-and-merge* mechanism: Sponge immediately redirects the increased load to SFs, which are imminent to offload, so that each SF instance can build small partial states and periodically send them back to the VMs to merge with the original states. This approach allows Sponge to promptly exploit fast-launching SF instances and bypass the prohibition of direct network communication between SF instances. For quick data redirection, Sponge exploits SF properties to prevent cold start latencies and pre-initiates copies of VM tasks on SFs to keep its runtime, process, and pre-initiated tasks readily available on time.

***Fast reactive scaling*:** On top of the fast resource scaling mechanisms on SF instances, Sponge identifies bottleneck tasks *reactively* and makes precise decisions on which part of the query to offload and how much of the compute resources to request. At runtime, Sponge continuously monitors the CPU usage, the major resource constraint of task execution, to quickly react to the changing input loads. Our offloading policy determines the fraction of input loads to offload based on excess events accumulated in the input queue and accounts for the optimal time to recover from load increases to meet the SLOs for a given query.

Sponge is built atop Apache Nemo [135, 115] with about 10K lines of code. We evaluate Sponge on EC2 instances (5× r5.xlarge) and AWS Lambda in-

stances (up to 200 Lambda instances of 1,769MB memory with one full CPU core) with NEXMark [119], a popular benchmark for stream processing. The effectiveness of our optimizations varies according to the characteristics of queries (e.g., dataflow pattern, # of tasks, and state size). Our evaluations show that Sponge exhibits similar performance to costly over-provisioned approaches, and reduces input event 99^{th}-percentile tail latencies by 88% on average compared to scaling queries on VMs and by 70% compared to scaling on SFs without our techniques.

1.2.2 Profiling Heterogeneous Networks to Find Efficient Global Network Paths

To overcome the shortcomings of prior approaches, we investigate a solution that can effectively distribute the query workload over the nodes without loss of query accuracy. In particular, there are multiple ways to distribute the tasks of an analytics workload over geo-distributed computing resources. Our approach for WAN analytics seeks to avoid exercising a path from data source to sink that provides poor bandwidth that comes from very different P2P bandwidths. Moreover, the system aims to adapt the task placement quickly to keep task executions stable despite changing network bandwidths [9, 10, 95]. To achieve the goals, we profile networks to obtain a holistic view on the network path diversity, and keep monitoring the network usage so that more resources could be spent on the networks that need more attention and the system can rapidly adapt to the abrupt changes in resource conditions.

To overcome the problem, we propose SWAN, a new WAN stream analytics engine that achieves the goals by incorporating two key techniques [113]. First, instead of trying to make an ultimate task scheduling solution, SWAN aims to alter the focus on providing a fast solution to keep the latency low, by pro-

viding a speed-oriented solution based on a fast heuristic model. Next, SWAN improves the quality of the generated solution by providing more flexibility to task placements through leveraging longer network paths that exhibit higher bandwidths. We have implemented SWAN on Apache Nemo [135, 115] and evaluated it with the NEXMark benchmark suite [119], a popular benchmark suite including multiple queries focusing on different areas of stream analytics. Within seconds, SWAN reduces the average job latency of the queries by 77.6% and increases the throughput rate by 5.64× over using the state-of-the-art distributed query execution.

1.2.3 Dynamic Tracking of Partition Metrics to Find the Optimal Caching State

To prevent inefficient memory utilization, Blaze introduces a unified caching mechanism that integrates the separate operational layers together. In doing so, Blaze introduces techniques that capture workload lineage through profiling and induction methods and dynamically monitor and update 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. 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. 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.

Blaze's cost model solves for selecting optimal partition states that will result in the smallest potential recovery cost in the workload. The state of a partition is defined by its location—whether it is in memory, on disk, or not in any persistent storage. The cost model is implemented as an integer linear programming (ILP) model, a popular solution for finding optimal values for minimization problems [56].

We implement Blaze on Apache Spark [14], with 6K lines of code. Our evaluations on 11 r5.xlarge AWS EC2 instances [2] show the performance improvements on two graph processing workloads, PageRank and Connected Components, logistic regression (LR), representing supervised iterative ML algorithms, and singular value decomposition++ (SVD++), representing unsupervised iterative ML algorithms. In the evaluations, Blaze accelerates the end-to-end execution time by up to 2.86× and reduces the cache data stored on disk by 90% on average compared to Spark.

1.3 Contribution

In this dissertation, we make the following contributions:

• We provide insights in the resource management problem in distributed data processing

• We propose dynamic system optimization techniques to supply the system with sufficient resources at all times for the different problems of data processing systems.

• We design and implement complete systems that run with our suggested mathematical models and calculations, to show that our proposed techniques significantly improve the performance in terms of throughput, latency, and JCT.

1.4 Dissertation Structure

The rest of the dissertation is organized as follows. Chapter 2 describes batch and stream processing models, and describe the data processing runtime layer that utilizes computer resources to execute data analytics workloads. Chapter 3 proposes our method of dynamically supplying CPU resources for unpredictable fluctuation of input loads through serverless frameworks. Chapter 4 proposes our methods to profile and find efficient network paths with higher bandwidths among the highly unpredictable heterogeneous global network connections. Chapter 5 proposes our methods to dynamically track and calculate potential overheads within iterative data processing workloads to find the most effective caching methods. Chapter 6 illustrates how each of the systems are implemented, and Chapter 7 shows the evaluation results with which we show that the solutions are effective in solving each of the problems. Chapter 8 describes existing works that try to handle similar problems to point out the novelty and the differences, and Chapter 9 concludes the work.

Chapter 2

Background

2.1 Iterative Data Processing Models and Caching

2.1.1 Dataflow Execution Model

Modern data analytics systems adopt a dataflow-based execution model, which represents data processing jobs as directed acyclic graphs (DAGs) [135, 115, 139, 3], as illustrated in Fig. 2.1. For example, in Spark [139, 14], vertices of a computational DAG represent resilient distributed datasets (RDD) and edges represent the computations that occur between the datasets. The computations can be categorized into a *transformation* or an *action*, which 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 [139], whereas actions trigger computations and output results (e.g., collect, reduce).

Different computations have different overheads and resource usage pat-

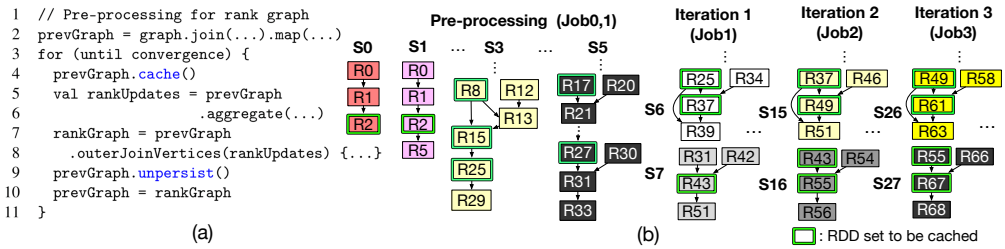

Figure 2.1: A simplified code snippet of a Spark PageRank [19] on GraphX [51] (a), and the RDD dependencies of the PageRank application (b). A Spark job is submitted for each iteration. Some stages, RDDs, and shuffle dependencies are omitted for simplicity. **Sx** denotes a stage number and **Rx** denotes the ID of an RDD.

terns. For example, simple operators like map and filter use less resources (e.g., CPU, memory, network) compared to resource-heavy join or groupByKey operators [117, 113]. Most dataflow execution models behave in this way, and a *job* acts as a unit of execution, triggered by an action, defined by the designated group of operators represented as a DAG. In iterative workloads, each iteration is triggered as identically-shaped jobs, which are chained according to their dependencies on the input read job and previous iterations. Each logical dataset (i.e., RDD) can be annotated to be cached or unpersisted through user APIs.

2.1.2 Parallel Execution and Partition Sizes

A job can be divided into 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) can be partitioned and processed by parallel computational *tasks* containing the computational logic defined by the operators within a stage, to 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 [14, 39, 135, 115]. 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 computations. Hence, bottleneck tasks are key to optimization as their dependent tasks require them to complete 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.

2.1.3 Caching in Existing Systems

Modern dataflow applications often consist of multiple jobs to execute complex workloads like iterative machine learning and graph processing applications that share and reuse data partitions within their application DAGs. In order to prevent redundant recomputations caused by missing partitions within the system storage, data processing systems provide user APIs for caching reusable data within memory or disks by annotating the datasets to *cache*, preparing them for potential computations (Fig. 2.1(a), Fig. 2.2❶). Once each dataset is no longer required in the workload, the user can also annotate them to be

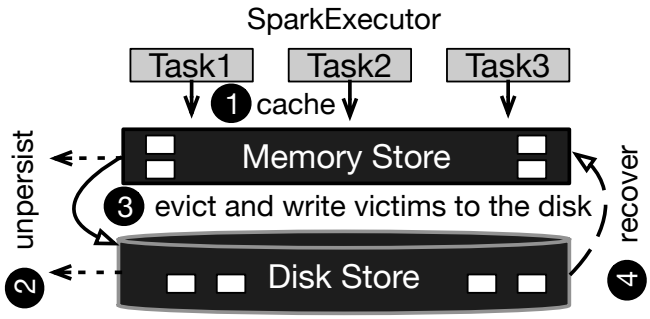

Figure 2.2: Caching and eviction on a Spark executor. Each task computes and caches RDD partitions into memory or disk within the executor that it is scheduled onto.

unpersisted and *discarded* from the system (Fig. 2.1(a), Fig. 2.2❷). 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 [18]. Data eviction occurs by unpersisting and discarding data (Fig. 2.2❷) or spilling data on disk (Fig. 2.2❸), according to the system settings (§5.1.2), especially upon memory-heavy computations like join and groupByKey [117, 113]. Upon cache misses during the execution, the system recovers the data by fetching them from disk (Fig. 2.2❹), or regenerates the data in memory through recomputations, instructed by recursive fault-tolerance mechanisms of the systems [139, 14]. Data recovery may also incur evictions to provide enough space in memory, and it 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, as `cache` and `unpersist` operations are performed by the user, evictions occur according to the eviction policy, and data is recovered by retrieving them from

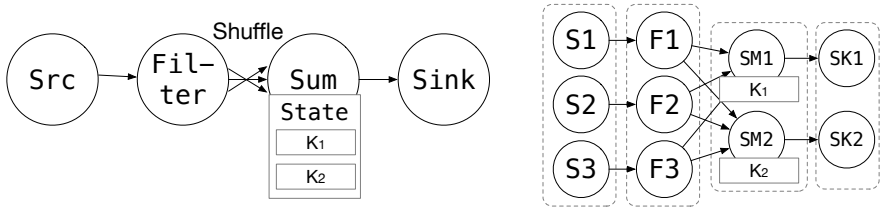

Figure 2.3: (a) Logical DAG of four operators including a stateful `Sum` operator with two key groups. (b) The corresponding physical DAG with parallel tasks.

disks or recomputing them upon cache misses [139, 14].

2.2 Stream Processing Models and Event Latency

2.2.1 Execution Model

A stream processing query processes an unbounded number of timestamped events to derive specific results (e.g., top K, statistics) on every temporal window. The execution of the query is generally expressed as a directed acyclic graph (DAG) of operators and data dependencies. As shown in Fig. 2.3, a vertex represents a stream operator that transforms input events and emits output events, and an edge represents how data flows between its adjacent operators. Popular stream engines like Flink [39], Spark Streaming [25], and Beam [31] aid users with high-level languages (e.g., declarative language) to facilitate query expressions. To provision compute resources over stream operators in response to the input data rate, the stream engine generates an optimized physical DAG (Fig. 2.3(b)) after translating a user query into a logical DAG (Fig. 2.3(a)). In a physical DAG, each logical operator is expanded into n parallel tasks, $p_0, ..., p_{n-1}$, where each task processes a disjoint data partition.

Stream operations that simply process an input record and emit its result

to downstream operations (e.g., map, filter) are connected with one-to-one dependencies, while operations that accumulate data from multiple source tasks (e.g., groupByKey, join) require shuffle dependencies ahead of performing the transform. Systems often group the operations connected with one-to-one dependencies as a stage, to pipeline the operations of a task together on a particular machine to reduce the data transfer. These stages are split into parallel tasks to distribute the job to a cluster of multiple machines. Unlike one-to-one dependencies, shuffle dependencies require data transfers over the network from multiple upstream tasks placed on different machines. In an environment with limited network resources, shuffle operations have to occur on the right network in order to prevent the query from suffering network bottlenecks.

2.2.2 Stream Operators and Resource Demands

A stream operator is either stateless or stateful. Stateless operators, such as `map` and `filter`, are typically used to compute individual events or drop unnecessary events or fields by applying predicates. Due to their simplicity, stateless operators can be pipelined together within a single node to leverage data locality and reduce network overheads. On the other hand, stateful operators, such as `groupByKey` and `join`, perform data grouping within a window boundary to organize unbounded streaming events into disjoint groups based on timestamps and aggregation keys, requiring computationally extensive key lookups . Thus, most streaming engines apply parallelism specifically to stateful operators such that a single stateful task p_i processes events that only belong to a non-overlapping key partition group K_i out of the entire key space $K = \cup_{i=0}^{n-1} K_i$.

Stateful operators are often the major source of system bottlenecks [113, 90]. In particular, since each parallel stateful task is assigned to a key partition group, it incurs shuffle communication for the events in its key group that are

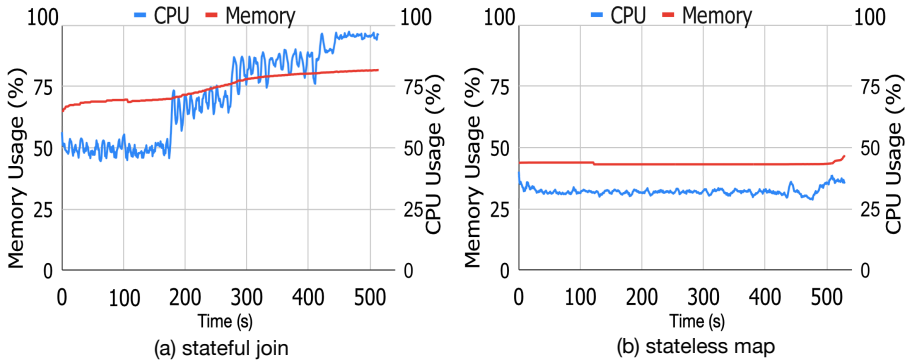

Figure 2.4: CPU and memory usage patterns for (a) stateful windowed `join` and (b) stateless `map` operators upon processing a fixed input rate of 80K events/s on identical 4 vCore nodes. The CPU and memory usage of the stateful operator increase until the window is full.

collected from the preceding (upstream) operators. Shuffle communication often requires the data to travel across different nodes, requiring data serialization and deserialization on top of the computation performed for the key lookups. As a result, as shown in Fig. 2.4, it is prevalent to provision more CPUs to execute stateful operators rather than stateless operators [67, 128].

2.2.3 Stream Operator Placement

In conventional stream processing systems [39, 54, 81, 103, 130], tasks are generally scheduled in a round-robin fashion for an even distribution of tasks across executors. In order to perform custom task placements on such systems, one must annotate node names on the individual tasks, with the features supported by resource managers. Since existing stream processing systems generally run on local clusters equipped with an excess amount of network resources, they focus on optimizing CPU and memory resources. In order to implement custom scheduling policies, existing systems require modifications on the scheduling

layer, as they are often designed with simple support for batch and stream modes for scheduling.

2.3 Resource Provisioning

Several real-world stream analytics systems report high temporal variability in the event count of data streams, even across one-minute time windows [84, 100, 80, 111]. This means that stream processing may need to frequently adjust resource provisioning and query execution plans in response to changes in input loads. Upon facing increased input loads, the system needs to allocate more resources to avoid operators being congested and maintain stable query latency.

Cloud offers primarily two options for on-demand resource allocation: virtual machines (VM) and serverless functions (SF). We compare three representative characteristics between these two options in more detail.

2.3.1 Resource Size

VMs are machine-isolated by bare-metal hypervisors, whereas SFs are process-isolated by OSes. Therefore, SFs are much more flexible in allocating resources. Cloud providers typically provide VMs in chunks of a pre-defined, fixed amount of resources (e.g., r5.xlarge with 4 vCores and 32GB memory). In contrast, SFs are allocated based on a specified memory size. For the memory size, cloud providers assign a certain number of CPU power (e.g., vCores) guided by their pricing model [27]. We observe that network bandwidth per SF instance is about 100Mbps and concurrently using multiple SFs can increase the bandwidth up to GBs of effective bandwidth, which VMs already support, providing enough capacities to handle most streaming workloads.

2.3.2 Start-up time

VM instances take a significant amount of time to launch and to prepare the runtime stack for query workload as they virtualize resources using bare-metal hypervisors. We observe that provisioning a new VM instance in major cloud service providers, like AWS, Azure, and GCP, mostly has a latency of over 25 seconds. On the contrary, SF instances provided by these cloud vendors take only 300-750 ms to launch and be ready to run because SF instances share runtimes and resources at the OS level.

2.3.3 Usage cost

SF instances are much more expensive to use than VM instances, e.g., 4× more expensive when running a 1GB SF instance with AWS Lambda (with $<$ 1 vCPU) compared to a t2.micro EC2 instance, which is equipped with 1 vCPU and 1GB RAM. However, temporarily using SF instances primarily for frequent short-lived bursty loads that constitute only a small fraction of time throughout the day [62] does not significantly increase the operational cost (§ 7.1.5).

Chapter 3

Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks

3.1 Challenges

Based on these observations, we propose to use a combination of VMs and SFs to have the best of both worlds. To achieve low latency and cost, we use cheap and stable VMs for handling continuous loads for long periods of time, and costly and reactive SFs for bursty loads during short periods of time. In this section, we describe several challenges in scaling streaming loads from VMs to on-demand SF instances.

C1. Migration with large operator states. For stream scaling in the cloud, existing approaches trigger resource adaptation primarily by re-scaling operators (i.e., increasing or decreasing parallelism) and migrating the bottlenecked tasks to the instances with available resources (i.e., load redistribu-

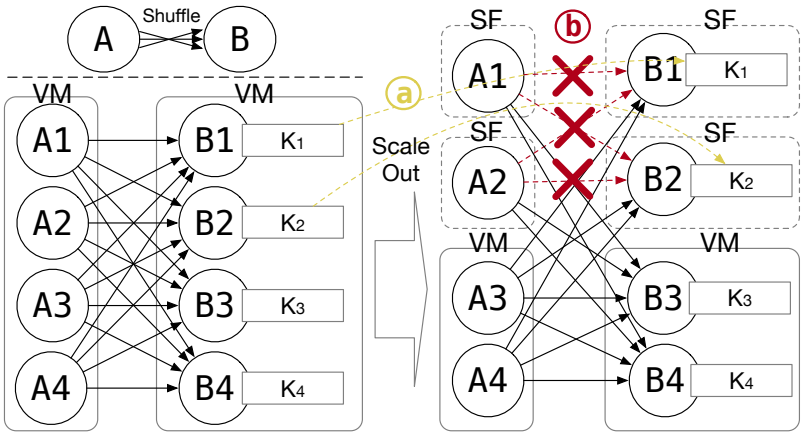

Figure 3.1: While scaling out on SF instances, the system must be aware that ⓐ task state migration overheads lead to latency spikes, and ⓑ direct data communication among adjacent tasks is prohibited between SF instances.

tion) [112, 40, 47, 84, 67, 25, 39]. Thus, even if we can set up SF instances quickly, the task state migration overhead is inevitable with existing systems, as shown by Fig. 3.1ⓐ, and paradoxically often inflicts damage to system performance.

Fig. 3.2 illustrates the various overheads that occur during a single workload scaling for the queries evaluated in § 7.1. As shown in this figure, the task migration and reconfiguration require a few extra seconds (3-4 seconds) to resume the work after the migration. Also, the state migration takes several seconds (e.g., from 4 to 17 seconds) depending on the state size because of the (de)serialization overheads of states. These task and state migration overheads lead to increased query latency due to the delay in receiving events from upstream tasks. The system that aims to meet low-latency SLOs must correctly and rapidly carry out task offloading to SFs. In particular, some use cases are expected to generate outputs even in order of seconds or less, without query

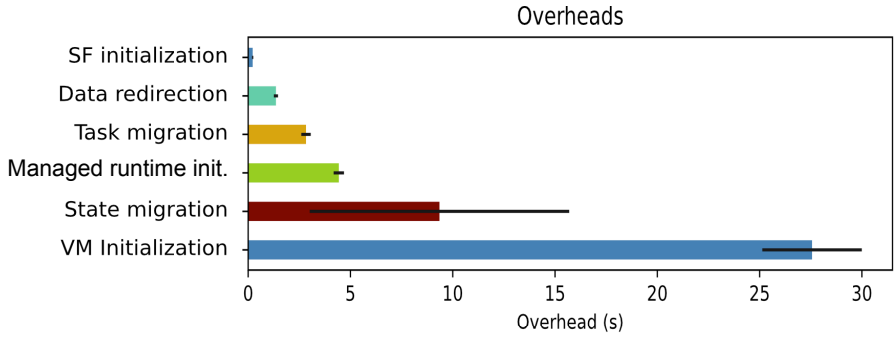

Figure 3.2: A comparison of the overheads of different steps of workload scaling on stream processing systems in the cloud. The VM, SF, and managed runtime initialization overheads are averaged across all instances, and the data redirection and task/state migration overheads are averaged across all single-scaling operations for the 5× input load experiment in §7.1. Error bars indicate the 95% confidence interval.

accuracy loss [111, 84].

C2. Indirect data communication between SF instances. As SFs are designed to be provisional and temporary, cloud vendors usually prohibit running a server process that can accept inbound network connections on an SF instance. Hence, direct data communication across SF instances is prohibited. This prevents neighboring stream operators (parent and child operators) from being offloaded to SFs simultaneously, as these operators require direct shuffle data transfers to group data by its key partitions, as shown in Fig. 3.1ⓑ.

Therefore, we can choose to migrate only certain tasks to SFs (e.g., either operator A or B in Fig. 3.1), but this eventually leads the bursty input load to end up on VMs on the adjacent operators and fails to alleviate latencies. Alternatively, we can offload all the tasks involved in the shuffle communication

on a large SF instance. However, this forces parallel tasks to be located on a single SF instance, which can lead to network pressure while leaving VMs idle. Consequently, it is essential to design the system to be able to offload adjacent operators together to SFs while bypassing the prohibited direct communication across SF instances.

C3. Quick decision making and scaling. With frequent unpredictable changes in input events, offloading decisions must be made quickly at runtime. Stream systems often detect symptoms of bottlenecks from system metrics and decide on whether and how much to scale. However, existing approaches can be too slow, as they require multiple iterations of optimization that scale bottleneck operators one after another [47]. Other work prevents such iterations by providing a global optimum after collecting all metrics from all executors to redistribute tasks [67]. While these approaches effectively find the target throughput and may be suitable for throughput-oriented workloads, they only work in intervals of multiple 10s of seconds and may not be suitable for latency-oriented workloads. For a stream system operating with diverse intervals and window sizes, it is important to have a uniformly fast and effective optimization level to prevent window outputs from being delivered too late.

3.2 Sponge Design

In this section, we describe the key pillars of our system design and explain the details by illustrating our graph rewriting algorithm, dynamic offloading policy, and the mechanisms that prevent cold start latencies and enable system correctness.

3.2.1 Design Overview

Latency spikes occur when the input rate r_i exceeds the maximum throughput m_i on a particular task p_i. When this happens, data starts to accumulate on the event queue, along with the operator state in memory, leading to high CPU usage and memory pressure. In a cloud environment, the maximum throughput m_i often depends on the CPU capacity allocated to the task, regardless of the operator type. This is because most cloud providers are equipped with GBs of network bandwidths, and memory pressure starts to increase when the CPU becomes saturated, and the input data builds up in the event queue with $r_i > m_i$. Therefore, our goal is to primarily focus on relieving CPU pressure. To achieve this, we design a system that accurately estimates the amount of additional resources needed and provides fast mechanisms for offloading CPU computation from VMs to SFs through two design principles: *redirect-and-merge* and *fast reactive scaling*.

Redirect-and-merge. Sponge is designed to rapidly forward increased input load to available resources in SF instances. To ensure speed, we bypass expensive query optimizations during runtime by performing DAG optimizations during compile time, i.e., when the application is launched (Fig. 3.3ⓐ). During compile time, there are no concerns yet about runtime synchronization and progress, so it only takes about 200ms upon workload initialization to perform the DAG optimizations. After the optimization, Sponge scheduler places tasks on appropriate executors (Fig. 3.3ⓑ). This allows Sponge to focus on which operators and how much of their data volume to offload to SFs based on monitoring CPU usage (Fig. 3.3ⓒ) without having to relaunch queries at runtime.

While stateful operators are our primary focus as initial bottlenecks, any

operator can become a subsequent bottleneck. Thus, we enable offloading for any operator, regardless of its type and statefulness. We design transient operators (TOs) so that operator logic can be prepared on SF instances to receive events immediately after detecting an increase in the input load and CPU usage on VMs (Fig. 3.3ⓓ). We also enable offloading to be activated at any time with high efficiency and responsiveness. To meet these requirements, we introduce a set of new proxy operators: router operators (ROs) and merge operators (MOs). ROs supervise the data communication to downstream VM and SF instances, in order to enable the system to rapidly and elastically *forward* data from any existing operators to the designated instances (Fig. 3.3ⓔ). To minimize state migration overhead, which is a major bottleneck in task migration [130, 59, 45, 53], the states, exclusively for the offloaded input load, are maintained separately on SFs. MOs enable the system to later *merge* the corresponding states of offloaded workload created on SFs with the states on the original VMs for any stateful operators (Fig. 3.3ⓕ). This way, the offloading overhead for both stateful and stateless operators is substantially reduced, as we only have to offload the computational logic, and not the states. The proxy operators are inactive during non-scaling periods to avoid extra costs and are only activated when needed.

Fast reactive scaling. With the principle above, we provide a fast reactive approach that prevents inaccurate predictions on resource provisioning by monitoring local performance metrics within the executors. Bottlenecks often occur individually on VMs, so it is sufficient to mitigate them *locally* within each VM. As briefly mentioned, relieving CPU pressure when the input rate r_i is greater than the operator throughput m_i is key to reducing CPU and memory strain in stream processing systems. Sponge has low monitoring overhead, with less

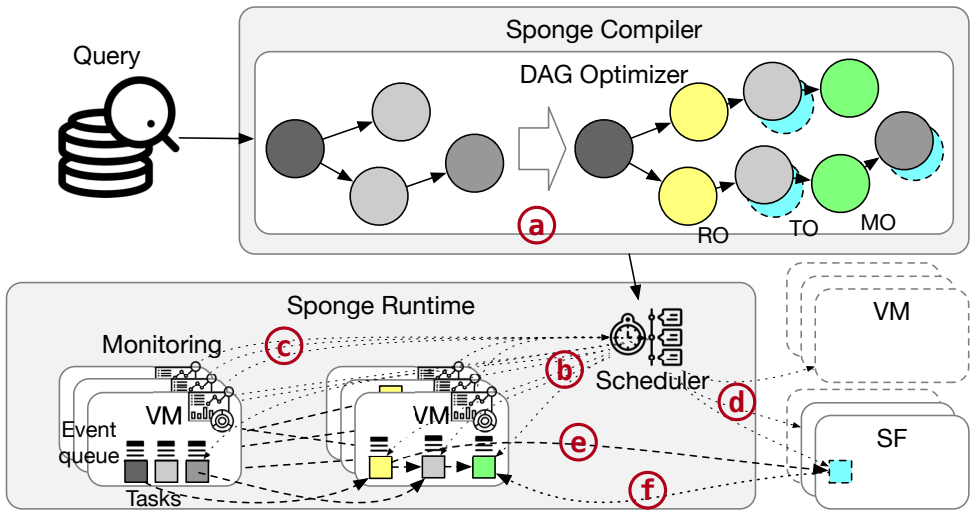

Figure 3.3: Sponge architecture.

than *10ms per observation*. Based on input rate and CPU usage observations, Sponge estimates the amount of CPU resources needed to increase operator throughput and meet our SLOs under increased input loads.

3.2.2 Compile-time Graph Rewriting Algorithm

At the start of the application, our compiler applies the graph rewriting algorithm (Algorithm 1) to the application DAG, which produces a new DAG based on a set of conditional rules, as shown in Fig. 3.4. In our algorithm, TOs, ROs, and MOs are inserted as follows. *TOs* are cloned stream operators with additional features to run on SF instances, such as maintaining partial states for stateful operators. Since all original operators are potential candidates for offloading, we first create TOs for all operators (line 3, Fig. 3.4ⓐ). This way, all operators causing CPU bottlenecks can scale on SFs with TOs. *ROs* enable data communications between VM and SF instances when the communication

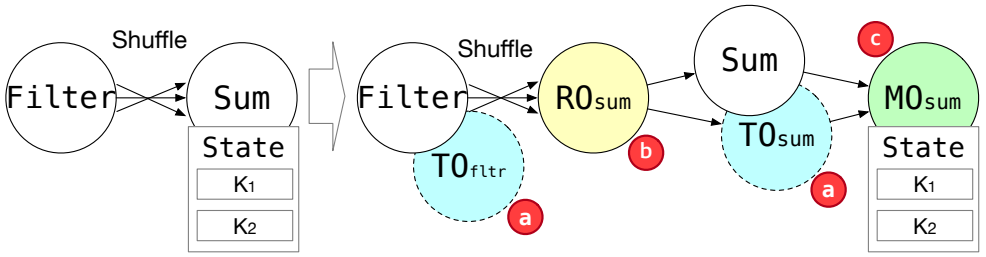

Figure 3.4: DAG transformation after graph rewriting.

pattern involves a shuffle or a broadcast (as one-to-one communications typically occur locally between pipelined operators) (§ 2.2). ROs run on existing VMs to redirect the input data to the downstream tasks running on either VMs or SFs without performing additional computations (line 5-12, Fig. 3.4ⓑ). If the communication pattern involves the same number of partitions and tasks between two operators, we pipeline the corresponding TOs with a one-to-one edge (line 13-15). ROs incur almost no costs as they simply redirect events to the tasks on target instances (e.g., conventional or TO tasks). Lastly, we insert an *MO* after each stateful operator for every edge, so that the partially aggregated states on the TOs can be merged back into the original states on VMs (line 16-23, Fig. 3.4ⓒ), where the details of the merging mechanisms are provided in § 3.2.5. Stateless operators do not need to merge states, so they simply pass on their output to the following operators (e.g., filter operator in Fig. 3.4). During non-scaling periods, ROs are not activated and TOs and MOs do not receive any data, adding no computational costs to the runtime. The operators are only activated upon offloading actions.

3.2.3 Dynamic Offloading Policy

In this section, we describe when Sponge triggers offloading, how many SF instances it uses, and how many events it offloads. Our goal is to constantly maintain low query latencies while keeping CPU utilization stable across all active cloud instances. To achieve this, Sponge quickly calculates the total number of CPU cores needed to meet this goal and the Sponge scheduler redistributes the workload accordingly among the tasks placed on VMs and SFs.

Overall workflow. The Sponge runtime, shown in Fig. 3.3, is a main system component that performs monitoring of the resources and operator states to take immediate scaling actions as needed. Each executor continuously monitors CPU resources and input rates, typically every second, and observes if the CPU load falls outside a stable range for consecutive periods. If so, the Sponge runtime initiates the scaling phase by first calculating the target system throughput, based on the over-subscription period of CPUs and the current input rates (that jointly decide the number of events in the queue), and the recovery deadline (the time remaining to clear the events and return the system to a stable state). Subsequently, the Sponge runtime adds new SF instances as needed to meet the recovery deadline by sending requests to the Sponge scheduler. The number of new SF instances is chosen to be minimum to neither over-subscribe nor under-subscribe the active cloud instances, minimizing operational costs. After a scaling action is taken, the Sponge runtime returns to the monitoring phase. It is possible that the Sponge runtime may go through multiple monitoring-scaling phases before the system becomes stable.

3.2.3.1 Detailed Offloading Process

CPU utilization goals. Along with system metrics, such as the input rate and operator latency, Sponge measures the CPU load of the executor in order to maintain adequate CPU loads on individual nodes. Through extensive experiments, we have observed that the input rate r_i exceeds operator throughput m_i and event queues start to build up (i.e., $r_i > m_i$) when the CPU is occupied at around 75-80% of its capacity. We have also seen symptoms of over-provisioned system resources when the CPU load falls under 50-60%. Due to such reasons, we aim to maintain the CPU utilization range between 60-80%.

Events in the queue. Assuming $r_i(t) > m_i(t)$ between times t_p and t_{p+1} ($t_p < t_{p+1}$), the number of excess events accumulated in the queue can be formulated as $\int_{t_p}^{t_{p+1}} (r_i(t) - m_i(t))\, dt$. Obviously, the accumulated events in the queue will be smaller if the duration $d = t_{p+1} - t_p$ is smaller. This is the main reason for using SF instances over VMs – to reduce the duration of $r_i(t) > m_i(t)$.

Recovery deadline. Recovering from this event backpressure is achieved by providing the system with additional resources to achieve higher throughput, m_{i_o}. If additional resources are available from time t_o, we should set a deadline t_{o+1} until which we aim to empty the queue to return to a stable state for our streaming system. We base the deadline on the window interval of the query (e.g., 10 seconds) so that we can deliver the query results within the query's next output boundary. Then, the number of additional events that can be processed from the queue can be expressed as $\int_{t_o}^{t_{o+1}} (m_{i_o} - r_i(t))\, dt$, where the increased throughput is larger than the input rate ($m_{i_o} > r_i(t)$). As a result, we should adjust our throughput m_{i_o} with sufficient additional resources to meet our recovery deadline t_{o+1} (e.g., $t_{o+1} - t_o \leq 10$) such that the following equation

holds:

$$\int_{t_p}^{t_{p+1}} (r_i(t) - m_i(t))\, dt \leq \int_{t_o}^{t_{o+1}} (m_{i_o} - r_i(t))\, dt \tag{3.1}$$

The approximation of the integrals is based on Simpson's rule provided by [12], which turns complex calculations into simple arithmetic that incurs trivial overheads.

Stable throughput per VM CPU core. To calculate the target number of SF instances required to achieve our target throughput, we maintain records of the CPU usage rate of the VM node during stable loads. Assuming that a task p_i runs on a single VM core with an average usage rate of $\overline{u_{cpu_i^{VM}}}(\%)$ and an average task input rate of $\overline{r_i}[events/sec]$ based on the records, we scale and approximate the input rate and throughput for 100% utilization of the VM core $rpc_i^{VM}[events/(sec \cdot core)]$ for task p_i, as follows:

$$rpc_i^{VM} = \frac{\overline{r_i}}{\frac{\overline{u_{cpu_i^{VM}}}}{100}} \qquad [events/(sec \cdot core)] \tag{3.2}$$

Required number of SF instances and data redistribution. Based on the approximation of how much input throughput a VM core can handle, we can calculate the number of required SF instances to achieve our target throughput m_{i_o} with a simple division. We offload tasks from VMs to SFs to keep the CPU utilization of VM clusters between $60 - 80\%$ in order to prevent resource over-provisioning. Hence, we target the VM CPU core usages at 70%, for our approximation to solidly fall into our target with a $\pm 10\%$ buffer even when our profiling measurements exhibit minor errors on time-varying variables like $r_i(t)$.

Assuming the CPU capacity of each SF instance core is different from that of a VM core, we can derive a relation between them with profiling: $capa_{core}^{SF} = \rho * capa_{core}^{VM}$. Correspondingly, $rpc_i^{SF} = \rho * rpc_i^{VM}$ because the throughput is proportional to the CPU capacity. Altogether, the number of total SF cores c

that we need to prepare to meet our latency goal for task p_i can be derived with Eq. 3.2 as follows:

$$c = \lceil \frac{m_{i_o}}{0.7 \cdot rpc_i^{SF}} \rceil = \lceil \frac{m_{i_o} \cdot \overline{u_{cpu_s}}}{0.7 \cdot 100 \cdot \rho \cdot \overline{r_i}} \rceil \tag{3.3}$$

where the number of required SF instance cores increases as ρ decreases. Finally, the number of SF instances can be calculated with $\frac{c}{k}$ where k is the number of cores per SF instance ($k = 1$ in our evaluation).

When offloading stateless or stateful tasks, Sponge evenly redirects data or redistributes keys to the $\frac{c}{k}$ SF instances, while processing remaining events on VMs to keep 70% CPU usage in average. If the event distribution is skewed across the key space, the solution can be extended to use key histograms for more accurate key partitioning, as in existing approaches [69, 35]. Both during scaling up and down, the target CPU usage is set at 70% within our target range.

3.2.4 Reducing Cold Start Latency

In order to timely gain access to SFs, Sponge provides two options: (1) warm-up SF workers in advance by sporadically processing short events [92, 49, 141] to minimize the managed runtime initialization overheads [82], and (2) use solutions like AWS SnapStart [74], which bring shorter initialization times of SFs by taking a snapshot of the initialized SF instance environment and caching it for low-latency access [6]. As SFs are charged based on the memory usage time and the number of requests [28, 29, 52], prices for pre-warming SFs are trivial (nearly zero). By default, Sponge prepares and keeps enough SFs warm to handle up to 5× the stable load during the workload. For bursty loads that exceed 5× the stable load, Sponge timely prepares new instances with SnapStart [74] on AWS. SFs on AWS SnapStart [74, 6] show a slightly worse start-up time compared to the instances that are kept warm in advance, but the overhead

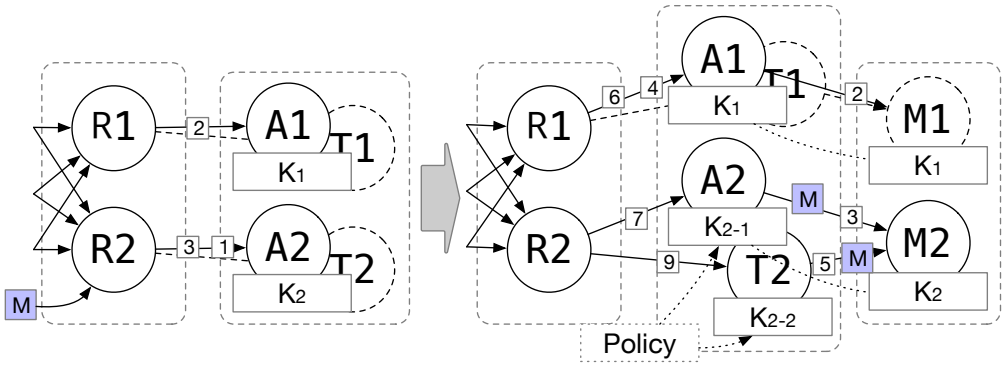

Figure 3.5: Once an operator (A2) tries to scale, an offload message (M) is generated at the RO (R2) to activate its TO (T2) and MO (M2). The offload message acts as a boundary among input events (1-9) for operator scaling and state merging.

is reduced by more than 80% compared to unoptimized JVM initializations, resulting in a sub-second total start-up time for SFs (§ 7.1.4). As a result, by enabling both methods for initializing the managed runtime, Sponge can timely gain access to SFs upon facing unpredictable bursty input loads.

3.2.5 Correctness

As stream systems are designed to be long-running, progress is tracked by the positions of the *watermarks* that flow along with stream events [135, 115, 39, 31]. Based on the intuition, Sponge maintains correctness by (1) introducing a watermark in the event stream as a control message upon (de)activating an operator and (2) ensuring that all events between two watermarks are processed in the original system (i.e., without offloading) or on the offloaded operators [84, 53].

Concretely, upon detecting a possible bottleneck in an operator task p_i on an executor, Sponge fires a watermark message `M` into the data channel (Fig. 3.5).

Upon receiving watermark messages, operators checkpoint their states to later recover from the checkpointed states, guaranteeing exactly-once processing. Sponge scales after temporarily pausing operators upon control messages and delivering messages to downstream tasks. Once all on-the-fly events in the data plane are consumed, downstream tasks send acks back to the upstream tasks to guarantee no event and state loss.

Thus, the events that arrive after `M` are immediately redirected to the tasks of the TOs on SFs, where partial states are aggregated if the operator is stateful. For stateful operators, TOs send the aggregated states to the following MOs placed on VMs, which know where to start merging the partial states with the original ones. Both incremental and appended aggregation can be mergeable with partial states, similar to how Flink [39] manages shared states, which causes moderate overhead on VMs. Even if events arrive out-of-order in the merge operators, they wait for the same watermark to arrive from the task in VM and its transient tasks so that the states can be synchronized. This ensures that all input data before and after `M` are processed according to the proposed optimizations.

Algorithm 1: DAG rewriting for operator insertion.

```
1  Function OperatorInsertion(dag)
2      for vertex, inedges in dag.topological_sort() do
3          t_op = TransientOp.for(vertex)
4          for inedge in inedges do
5              if inedge.comm != 1to1 then
6                  r_op = RouterOp.create()
7                  dag.remove_edge(inedge)
8                  e1 = {inedge.src→r_op, inedge.comm}
9                  e2 = {r_op→vertex, 1to1}
                   // connect transient operators
10                 e3 = {inedge.src.t_op→r_op, inedge.comm}
11                 e4 = {r_op→vertex.t_op, 1to1}
12                 dag.add_edges([e1, e2, e3, e4])
13             else
14                 e = {inedge.src.t_op→t_op, 1to1}
15                 dag.add_edges([e])
16             if inedge.src.is_stateful() then
17                 m_op = MergeOp.create()
18                 dag.remove_edge(inedge)
19                 e1 = {inedge.src→m_op, inedge.comm}
20                 e2 = {m_op→vertex, 1to1}
21                 e3 = {inedge.src.t_op→m_op,inedge.comm}
22                 e4 = {m_op→vertex, 1to1}
23                 dag.add_edges([e1, e2, e3, e4])
24     return dag
```

Chapter 4

SWAN: WAN-aware Stream Processing on Geographically-distributed Clusters

4.1 Challenges

Existing policies designed for local clusters result in significant performance loss and inefficient resource utilization under WAN, due to the fundamental differences in network environments [101, 123, 125]. While the superfluous network infrastructure on local clusters enables datacenters to reserve network resources for network traffics from particular machines, long-distance cables installed across continents must be shared by multiple different network traffics due to the limited infrastructure. Due to such nature of WAN networks, it exhibits unpredictable variability in both space and time. We characterize the network with two aspects, into spatial and temporal variations, to gain insights on how to optimize streaming engines for WAN environments.

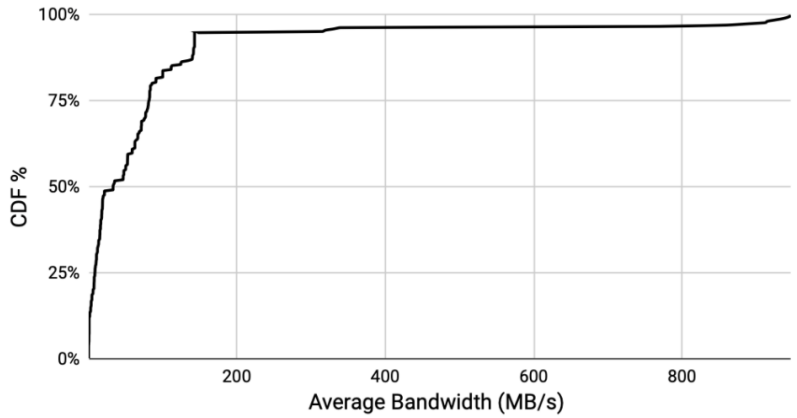

Figure 4.1: A CDF of networks showing the spatial variation of a geo-distributed cluster.

Spatial variability. Spatial variations are caused by the distances and the different network infra connecting the clusters. The infrastructures (e.g., cables, satellites) are usually managed by the internet service providers (ISPs) to connect different LAN networks together. As the infrastructures are each designed with different technologies and budgets, a large diversity exists among the different paths over the WAN. Fig. 4.1 shows the diversity of network bandwidths in a geo-distributed cluster of 16 nodes (i.e., ${}_{16}C_2 = 120$ connections), scattered around 8 different sites over 3 continents (details in §7.2.1). Here, we can witness varying average bandwidths as low as about 500KB/s up to 900MB/s, depending on the different locations and distance between the sites, while most network connections show average bandwidths below 100MB/s.

When suggesting a network path between two sites, the ISPs usually provide the path with the lowest latency, but this does not necessarily have the highest bandwidth. If the size of the data to be transferred is larger than the provided bandwidth, it results in network congestion, which causes the latency to rise, as the large traffic has to wait in the queue before being processed. Therefore,

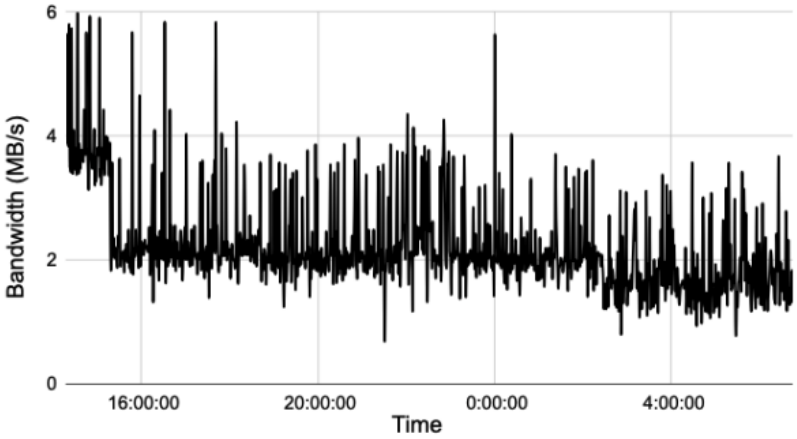

Figure 4.2: A graph showing the temporal variation of a network through time.

it is important to find a network connection that has sufficient bandwidth to accommodate the expected traffic. Subsequently, operator placement decisions need to appropriately made to effectively harvest the available bandwidth across the globe.

Temporal variability. Temporal variations are caused by the high concentration of network traffics through the limited bandwidths of WAN settings. Due to the large number of network users sharing the provided WAN bandwidths, network patterns are highly unpredictable and display diverse patterns over time. Also, the long distance within the WAN adds other physical factors that contribute to the instability of the network. While WAN networks over longer distances often show more instability compared to the ones that are relatively shorter, there also exists heterogeneous variability among the participating regions. WAN network bandwidth patterns can be transient or permanent, and steep or gradual, depending on the cause of the effect, and can occur in band-

width rises or drops, as shown in Fig. 4.2. Our observations on a geo-distributed cluster show that over half of WAN networks suffer from bandwidth drops of over 20% every 6 minutes on average.

While bandwidth rises do not directly affect the stream analytics performance, bandwidth drops result in increased latency and lower throughput if not handled appropriately. Permanent changes or significant bandwidth drops are particularly considered detrimental and require the system to adapt the task placement to mitigate potential network bottlenecks.

Prior approaches and limitations. Existing data processing systems designed for WAN environments focus on a single type of variation depending on the type of data that they target. Existing systems that target short-lived batch processing jobs [123, 101, 124] focus on the spatial variability to reduce the network data transfers for lower network budget and assume stable network throughout the job.

Regardless, the question of how to perfectly schedule tasks to the geo-distributed nodes while considering all network conditions and task dependencies altogether is known to be NP-hard [85, 91]. Existing works often depend on ILP solvers to schedule tasks in a way that minimizes the longest link transfer finish time of the reduce tasks [123, 101]. Specifically, they optimize the job by observing each stages one after another, to find the right proportion of tasks to place for each node of the geo-distributed cluster. Nevertheless, scheduling tasks using ILP solvers are still significantly slow compared to other conventional scheduling methods despite their simplicity of illustrating the problem [101]. The optimization overhead depends on the length of the query execution plan, but on average, it is 25× slower than conventional scheduling for NEXMark streaming benchmark queries (§ 7.2.3). Since stream processing has strict la-

tency requirements, it is difficult to adopt an ILP model to effectively handle the temporal variation.

On the other hand, existing systems that target WAN-aware stream processing [105, 140, 61] describe methods to adaptively collect data to a single data center. In order to do so, such systems propose effective ways to pre-aggregate, degrade, and statistically estimate the raw data and trade the output quality for better performance over limited bandwidths. However, while such optimizations are very effective in specific applications (e.g., video processing), accuracy-sensitive applications, such as fraud detection, billing queries, and global stock or transactional analysis, cannot adapt such methods, as lower accuracy can often lead to undesirable reliability and additional problems for such queries [57, 61, 105, 140]. In order to bypass such problems, it is required for the WAN-aware stream processing systems to be designed to run on tasks distributed across a geo-distributed cluster, in a way that can rapidly adapt to the altering conditions.

4.2 System Design

4.2.1 Insights

Good heuristics over an expensive solver. While ILPs provide efficient simple abstractions for materializing the optimization problem of distributing tasks to geo-distributed nodes of a cluster to solve the spatial variability, ILP solvers are too slow to be dynamically used for stream processing systems. ILP could be a good solution if WAN networks do not possess temporal variability and the optimization occurs just a single time. However the 25× overhead is not trivial with job latency if the optimization is to occur repeatedly. In order to mitigate the optimization overhead, we propose using a fast heuristic

model that effectively captures WAN characteristics. In building the heuristic model, we aim to put our focus on two primary aspects. First, we aim to find a model that accurately captures the network costs, based on the number of upstream and downstream tasks and the measured network bandwidths, and minimize the network cost throughout the stream analytics job. Second, we try to distribute an even number of tasks to each nodes if possible, in order to prevent computational bottlenecks that can occur when the distribution of tasks are too concentrated on a specific set of nodes.

Query rewriting to fully cover promising longer paths. Wide-area networks are usually managed by ISPs, who, by default, provide the network with the minimum latency upon each request. However, this does not relate to higher bandwidth. Fig. 4.4 shows an example of a network connection between Seoul and Paris that exhibits more network bandwidth if travelled through New York, instead of travelling directly to each other. When provided with a low-latency network with limited bandwidths, the data transfer can suffer from even more latency due to the congestion caused by excess network traffics. Moreover, limited bandwidths lead to full usage of the bandwidth, leaving less room to act as a buffer upon sudden small bandwidth drops with temporal variations. In order to prevent such cases and leverage the network connections with more bandwidths, we perform query rewriting to cover longer paths that are more promising for our workload. In order to capture such network connections, we enable our system to extend the query execution plan with relay operators that simply pass on the data from the uplinks to the downlinks.

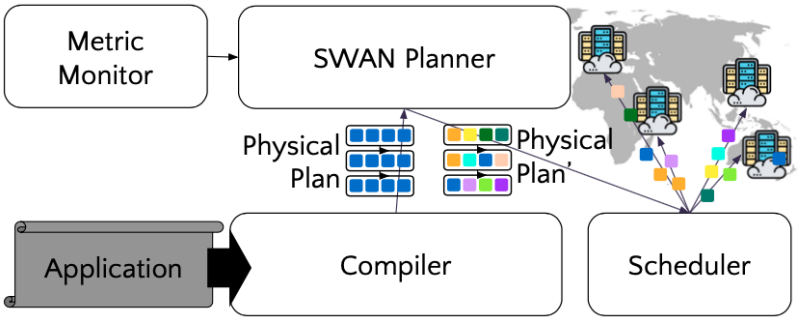

Figure 4.3: An overall architecture of the SWAN system.

4.2.2 Design Overview

We capture these insights in our system, SWAN, which uses a fast and effective heuristic model for placing tasks on the geo-distributed cluster, with extended optimization techniques to rewrite queries and expand the query execution plan to capture promising longer paths with more bandwidths. Fig. 4.3 shows the key system components involved in the scheduling and optimization of the query execution plan. Once a dataflow application is submitted to SWAN, the compiler performs basic optimizations, such as stage partitioning and determining the number of tasks for the workload, and builds the application into a physical execution plan, which is composed of a group of tasks. With the query execution plan, the SWAN planner collects network metrics and the total number of tasks and their dependencies, to calculate the predicted network cost, and determines where to place each of the tasks. The scheduler takes the physical plan with the placement information and distributes them to the geo-distributed cluster in the way specified by the plan.

When the latency increases in the workload, the metric monitor triggers dynamic optimization, which submits a modified physical plan with the new placement specification to the scheduler. The scheduler fires an optimization

mark, which is a special implementation of a watermark that triggers each tasks to checkpoint their data to be migrated according to the new placement specification. Each tasks are sequentially migrated to replay the data from the point of the optimization mark.

4.2.3 Operator Placement Algorithm

A stream processing application typically has source tasks fixed on specific sites. The tasks of children stages can predict the potential network cost that it incurs if placed on a specific site $s \in S$ among all sites S, based on the number of upstream tasks on the upstream site $u \in S$ and the given bandwidth between the sites $bandwidth_u^s$. We also find the appropriate number of task slots for each site to evenly distribute tasks among the multiple sites. Based on the calculation of these two values, we distribute tasks according to the ratio of the number of remaining slots divided by the expected network cost of a site. This way, we distribute more tasks to the sites where the network cost is smaller, while also distributing tasks to sites that have more task slots left. We describe the logic above with the following equations:

$$Distribution\_ratio\_factor_s = \frac{task\_slots_s - tasks\_count_s}{network\_cost_s}, \tag{4.1}$$

$$s.t.\ network\_cost_s = \sum_{u \in S} \frac{upstream\_tasks\_count_u}{bandwidth_u^s} \tag{4.2}$$

$$and\ task\_slots_s = \sum_{node \in s} \lceil \frac{\sum tasks\_count}{\sum node\_count} + \frac{1}{2} \rceil \tag{4.3}$$

To understand the scheduling algorithm, we use an example on a WAN setting illustrated on Fig. 4.4, where three nodes are allocated on each of the three sites, making it a cluster of $\sum node\_count = 9$ nodes. Let us assume a case of executing a three-stage application, where 8 tasks are generating source

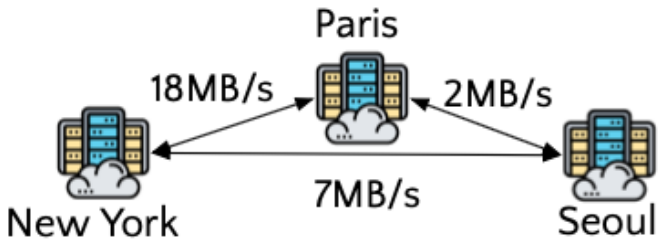

Figure 4.4: An example of a geo-distributed cluster setup.

data in stage 0, followed by 5 tasks in stage 1, and 3 tasks in stage 2. Since the application consists of a total of 16 tasks to be placed on a total of 9 nodes, we set the upper limit for the number of tasks on each node to $\lceil \frac{\sum tasks\_count}{\sum node\_count} + \frac{1}{2} \rceil = \lceil \frac{16}{9} + \frac{1}{2} \rceil = 3$, which results in a total of $3 \times 3 = 9$ slots for each site. Specifying task slots enables us to prevent tasks from being too crowded on a specific site.

Let us assume a case where 3 data source tasks are placed in Seoul and NY, while 2 data source tasks are placed in Paris. In order to schedule the following 5 tasks in stage 1, we first observe the remaining slots for each site, where a total of $9-3=6$ slots are left in Seoul and NY and $9-2=7$ slots in Paris. We next calculate the network cost for a potential individual task if it was to be placed on a specific site, described in Eq. 4.2. By this calculation, NY, Paris, and Seoul has the cost of $\frac{2}{18}+\frac{3}{7} \approx 0.5$, $\frac{3}{18}+\frac{3}{2} \approx 1.7$, and $\frac{3}{7}+\frac{2}{2} \approx 1.4$, respectively. With these numbers, we find the target ratio of task distribution as described in Eq. 4.1. This way, NY, Paris, and Seoul has the target distribution ratio of $\frac{6}{0.5} : \frac{7}{1.7} : \frac{6}{1.4} \approx 3 : 1 : 1$. Consequently, 3 tasks of Stage 1 are scheduled on NY, and Seoul and Paris are each scheduled with a single task.

The remaining 3 tasks of stage 2 are scheduled by repeating the steps above. The remaining slots are 3, 6, 5, respectively for NY, Paris, and Seoul. Network costs are $\frac{1}{18} + \frac{1}{7} \approx 0.2$, $\frac{3}{18} + \frac{1}{2} \approx 0.7$, and $\frac{3}{7} + \frac{1}{2} \approx 0.9$ for NY, Paris, and Seoul. According to the logic above, the target distribution ratio is $\frac{3}{0.2} : \frac{6}{0.7} : \frac{5}{0.9} \approx 6 :$

$3 : 2 \approx 2 : 1 : 0$, and hence 2 tasks are scheduled on NY and a single task is scheduled in Paris.

Our operator placement algorithm can be generally applied to any physical plan consisting of multiple tasks, to place them on an arbitrary number of nodes placed on different sites. In our example, we can observe that NY has the best network conditions among the three sites, and our scheduling algorithm places a total of 8 tasks on NY, while placing 4 tasks each in Paris and Seoul. This way, data can flow into the direction of the site with the most available bandwidths, while preventing a specific site from having too many tasks. While our example illustrates a small example, the slot allocation comes into great usage when the scale of the physical plan grows to hundreds of tasks.

4.2.4 Query Rewriting

Now that we have a query execution plan with annotations specifying the nodes to place each of the tasks on, we can do further optimizations on the scheduling. On the setting illustrated in Fig. 4.4, we can see that the network between Seoul and Paris exhibit a much narrower bandwidth compared to the bandwidth between two areas through New York. Although we try to use the high bandwidths as much as possible with our operator placement algorithm, a few tasks inevitably have to transfer data through the low bandwidth between Seoul and Paris. In such cases, we provide an extra option to insert a relay task between the tasks in order to be able to send the data through the network going through New York, instead of using the original option.

Nevertheless, if too many tasks transfer data over the high bandwidth, that bandwidth can also be congested due to the large number of traffic. Therefore, we choose to insert the relay task only if the average bandwidth among the tasks transferring data through the network becomes higher with the new option:

$$\frac{bandwidth_{relay\_network}}{\sum tasks\_count_{relay\_network}} > \frac{bandwidth_{original}}{\sum tasks\_count_{original}} \tag{4.4}$$

Since the major problems in WAN-based analytics occur with lower bandwidths that are intolerable with temporal fluctuations, inserting relay tasks do not significantly increase the latency, as with higher bandwidth, we can allow higher degree of concurrent stream data transfers.

Chapter 5

Blaze: Holistic Caching for Iterative Data Processing

5.1 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.

5.1.1 Caching and Eviction Mechanisms

Caching layer. As shown in Fig. 2.1(a), caching interfaces are provided through `cache()` and `unpersist()` APIs for users to analyze the workload and hint the datasets to persist after each iteration and later reuse them without recurring recomputation overheads. For example, in Fig. 2.1(b) `Iteration 2`, we can see that as a result of caching annotations in Fig. 2.1(a), `R49` and `R55` are cached, while `R37` and `R43` are unpersisted from disk, which happens similarly in different iterations. Nevertheless, not only do the current interfaces

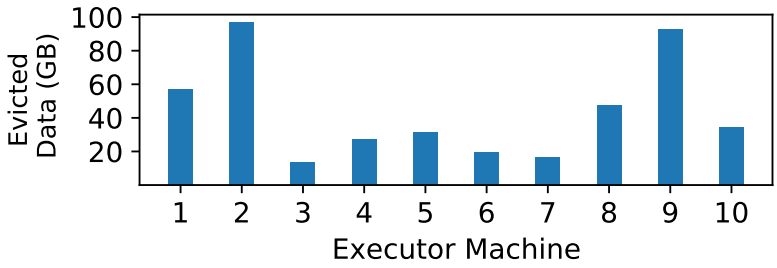

Figure 5.1: Caching at dataset granularity causes different sizes of evicted data among different executor machines on a `PageRank` application in our evaluation (§ 7.3).

require the users to have app-specific knowledge to cache the right datasets, but they also fall short in providing fine-grained caching at partition granularity, while each partition varies in terms of size and computational time. As shown in Fig. 5.1, this can become problematic as the current coarse-grained caching at dataset granularity causes executors to cache partitions with different sizes (i.e. disk I/O overheads) and computation overheads, to incur different amounts of evictions on different executors. Also, coarse-grained caching can lead to unnecessary caching for 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. 2.1(a), it leads to caching of all relevant partitions of the RDDs in Fig. 2.1(b), some of which are unnecessary as they may not have any future usages or incur trivial recovery costs [55] (§ 7.3.2).

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. A few different eviction policies include classic LRU (least recently used) [18] and GDWheel [77] policies, learning-based TinyLFU (lightweight least frequently used) [46] and LeCaR (learning cache replacement) [122] policies, as well as dependency-aware LRC (least reference count) [137] and MRD (most reference distance) [98] policies. As each name suggests, each policy determines which cached partitions to prioritize and evict from memory based on historical usage patterns and logical references. While these policies have been successful in handling caches in other contexts (e.g., CDN and web services) [77, 116, 46, 122, 34, 26, 36, 33, 32], 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. As such information evolves dynamically over the iterations, it requires the system to be adaptive to changing situations.

5.1.2 Recomputation and Disk I/O Costs

While managing cache storage according to the caching policies, current systems *fix* their way (i.e., `MEMORY_ONLY` or `MEMORY_AND_DISK`) for each workload in using the different storages in handling evictions, without the flexibility to choose which way to take for the different datasets or partitions [14, 139, 115, 135], due to their implementation designs and challenges regarding the dynamically changing potential recovery costs upon different situations (§ 5.2.3). The first way is to simply discard the data from memory upon eviction (i.e., `MEMORY_ONLY`), to rely on the fault tolerance mechanisms that recover data through recursive recomputations. Similar ways include (de)serializing data in memory or in off-heap space to save some memory, but the basic mechanism is

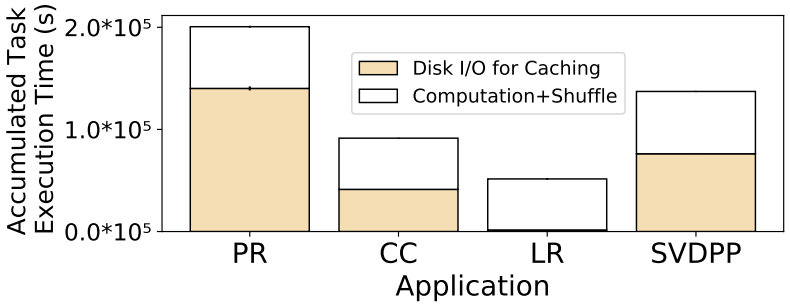

Figure 5.2: The accumulated execution time of tasks in four applications (§ 7.3), including the total time of disk I/O costs for recovering evicted data. Data (de)serialization is included in the disk I/O time.

similar in that they recover data through recomputations. The second way is to keep a two-tiered storage to evict data into cheaper supplementary storage, to recover data by fetching them from slower secondary storages upon cache misses (i.e., `MEMORY_AND_DISK`) [1]. 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 [104, 142].

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 bring the data back in memory for evicting and recovering data in `MEMORY_AND_DISK` mode. We can see in Fig. 5.2 that disk I/O overhead is the major source of bottleneck in two graph processing (i.e., PageRank and ConnectedComponents) and one machine learning (i.e., SingularValueDecomposition++) applications, where the detailed experimental setup is described in § 7.3. 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

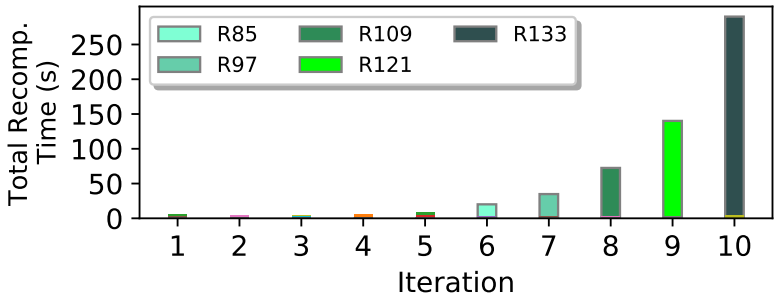

Figure 5.3: Breakdown of the total recomputation time for each iteration in PageRank (§ 7.3). The RDDs incurring the highest recomputation time within the iteration are labeled from iteration 6 to 10 (`RDD 85`, `97`, `109`, `121`, and `133`).

costs. Therefore, for applications with large partition sizes, it incurs 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. 5.3, 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, in Fig. 5.2, we can see 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.

5.2 Design Goals and Challenges

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

5.2.1 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 [14, 18], one needs to first determine whether or not it would be advantageous to cache the data in memory (Fig. 2.2❶). We should 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_i, it is obvious that it is advantageous to cache it if there is enough memory space to store p_i. However, in cases where memory space is constrained, it is advantageous to cache data in memory only if the p_i 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_i, we must 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 should be 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.

5.2.2 To Evict, or Not To Evict?

In cases where it requires some evictions of cached partitions, when the fixed memory space is saturated, to store a potentially expensive partition p_i, one

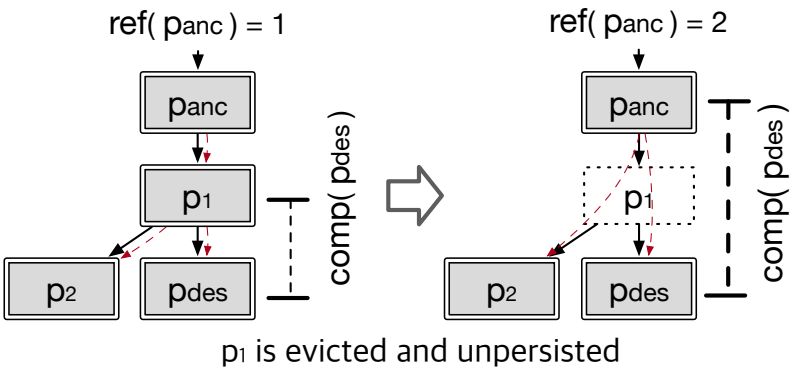

Figure 5.4: The dynamically changing `comp`(p_{des}) and `ref`(p_{anc}) upon evicting and unpersisting p_1 from the cache.

needs to carefully select the partitions to evict based on the potential costs, instead of on history-based caching policies [18, 137, 87, 46, 122]. 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 §5.1.2. 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. 2.2❷). For others, it may be more advantageous to spill and store them on disk, if they have smaller partition sizes while their computations have been more time-consuming (e.g., model calculation for LR) (Fig. 2.2❸). Both aspects have to be carefully considered in choosing and evicting partitions from memory, while also keeping the potential recovery costs in the calculations, also to perform data recovery in memory or from disks in a timely manner if it is more beneficial to do so (Fig. 2.2❹).

5.2.3 Dynamically Changing Data Dependency

The most important challenge to address is that 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. 5.4, if p_1 is unpersisted, the recomputation cost for p_{des} will increase from $p_1 \rightarrow p_{des}$ to $p_{anc} \rightarrow p_1 \rightarrow p_{des}$, if p_{anc} 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. Moreover, future dependencies can also dynamically change upon unpersisting partitions. When trying to compute for p_{des} and p_2, it initially does not incur any references to p_{anc}, but after unpersisting p_1, p_{anc} is referenced by both p_{des} and p_2 through p_1. Such evictions and unpersist actions 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.

5.3 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.

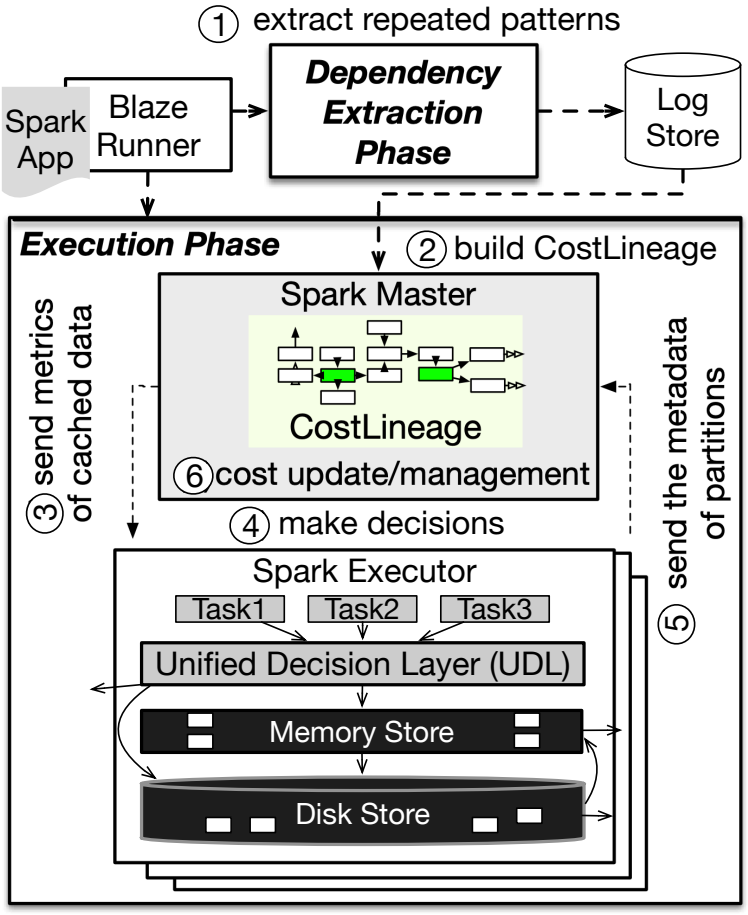

Figure 5.5: The overview of Blaze.

5.3.1 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 goes over two phases, as shown in Fig. 5.5. First, ① Blaze runs the workload on a small portion of the original input data (i.e., $<$ 1MB) to extract the code path and dependencies between datasets, 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 (§ 5.3.3). Next, ③ Blaze sends the metrics to the executor ④ to automatically make unified decisions for caching, eviction, and recovery (§ 5.3.5, § 5.3.6), based on the lineage and the calculated costs on the `CostLineage` (§ 5.3.4). 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.

5.3.2 Design Principles

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

$$\forall p_i \in P,\ m_i + d_i + u_i = 1\ (m_i,\ d_i,\ u_i \in \{0,\ 1\}) \tag{5.1}$$

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 ($d_i \rightarrow m_i$). 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, $cost_d(p_i, t)$, along with the potential recomputation cost of partition p_i at time t, $cost_r(p_i, t)$ (§ 5.3.4). If the $cost_r(p_i, t)$ is smaller than $cost_d(p_i, t)$, discarding and recomput-

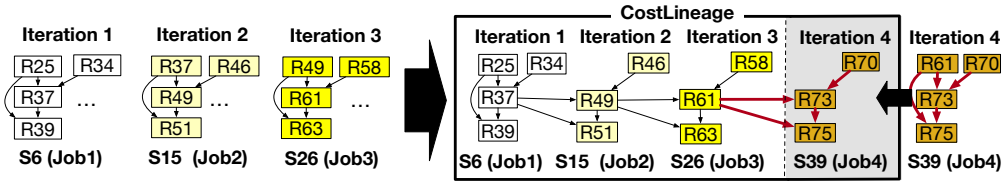

Figure 5.6: A `CostLineage` constructed from the extracted RDD lineages of the PageRank application in the dependency extraction phase. Duplicate RDDs are dynamically detected and merged upon new iterations and future iterations are induced.

ing 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 it was not cached in memory, would be the minimum between the two values:

$$cost(p_i,\ t) = \min(cost_d(p_i,\ t),\ cost_r(p_i,\ t)) \tag{5.2}$$

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, $\sum_{i \in P-M} cost(p_i, t)$, as low as possible, where $M = \{p_i \in P \mid m_i = 1\}$ while Eq. 5.1.

5.3.3 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 profiling phase. Since the input data is minuscule (i.e., $<$ 1MB), the profiling 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 (§5.3.5).

As shown in Fig. 5.6, 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. After the actual execution of the workload starts, the partition metrics for the initial iterations are simply recorded on the `CostLineage` since the initial iterations of the actual workload do not request for any evictions due to sufficient memory space. `CostLineage` updates the partition metrics upon receiving new partition metadata from the executors, which actually materialize the iterators of the partition data upon execution, and gain access to the partition sizes, locations, and computation time. 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 datasets that play the same role in different iterations by analyzing the sizes of the datasets from the initial iterations. Concretely, `CostLineage` takes the sum of partition sizes for each dataset and uses a simple automata algorithm based on the differences in the dataset sizes of adjacent operators to find the repeated patterns. By doing so, we can detect the iterative patterns

among datasets that are generated from the same code path in the loop, and perform inductions to predict the partition sizes for future iterations. For the missing values of partition metrics in the `CostLineage`, Blaze inductively fills in temporary approximated values of metrics for the undiscovered partitions by applying a lightweight linear regression model based on the existing metrics from previous iterations, until the end of the application. Likewise, future iterations that had not been captured during the profiling phase can be inducted similarly. Based on these partition metrics, we can estimate potential costs for recomputation and disk overheads of the different partitions whenever a caching decision needs to be made.

5.3.4 Potential Recovery Cost Estimation

We describe how we estimate $cost_d(p_i, t)$ and $cost_r(p_i, t)$ individually. The disk cost, $cost_d(p_i, t)$, can be calculated simply by dividing the size of the partition $size(p_i)$ with the profiled read/write throughput of the disk, $throughput_{disk}$, which can be profiled within the system during runtime or initially approximated through conventional softwares [4]. The recomputation cost for a partition p_i has to be defined recursively with respect to the ancestor upstream partitions $A_i = \{p_j \in P \mid p_j \rightarrow p_i\}$ that are not cached in memory $\{p_j \in A_i \mid m_j = 0\}$. We define the longest recomputation time from the upstream partitions as the recomputation cost, $cost_r(p_i, t)$ (§ 2.1.2), which dynamically changes according to the `CostLineage`. We can compute the potential recovery costs within milliseconds with the following logic:

$$cost_d(p_i,\ t) = \frac{size(p_i)}{throughput_{disk}} \tag{5.3}$$

$$cost_r(p_i,\ t) = \max_{p_j \in A_i}((1 - m_j) \cdot cost(p_j,\ t) + cost_{j \rightarrow i}) \tag{5.4}$$

where $cost_{j \to i}$ is the computation time for deriving p_i from p_j and $cost(p_j,\ t)$ is defined in Eq. 5.2.

5.3.5 Finding the Optimal Partition States

Based on the collected dependencies and the partition metrics on our `CostLineage` (§ 5.3.3), we can now formulate our solution as an integer linear programming (ILP) model, with respect to our cost estimation methods (§5.3.4). 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, $p_j \in J$, which is captured and induced within our `CostLineage`. We can set up a constraint for our memory space for all relevant partitions, $p_i \in P$, that are recorded within our `CostLineage`, as well as our objective function to minimize the potential costs for the partitions that are to be used in our upcoming jobs $p_j \in J$. 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 [16], to ensure ILP performance (i.e., < 5 seconds). This adaptively minimizes the potential costs, including the disk I/O and recomputation overheads, for the near future, regardless of the workload progress in the current job:

$$\text{Minimize} \quad \sum_{p_j \in J} (d_j \cdot cost_d(p_j,\ t) + u_j \cdot cost_r(p_j,\ t)) \tag{5.5}$$

$$\text{Subject to:} \quad \sum_{p_i \in P} size(p_i) \cdot m_i \leq capacity_{mem}\ ,$$

$$Eq.\ 5.1,\ Eq.\ 5.2,\ Eq.\ 5.3,\ Eq.\ 5.4 \tag{5.6}$$

In cases where disk space is also constrained, the ILP can be simply extended by adding another constraint, $\sum_{p_i \in P} size(p_i) \cdot d_i \leq capacity_{disk}$, where we set $capacity_{disk}$ as an abundant value in this dissertation.

5.3.6 Automatic Caching

As the `CostLineage` keeps track of the partition states and metrics, and hence the potential recovery costs of each of the partitions, it can easily find the minimum potential cost among the partitions that reside in memory. If a partition has future references in the `CostLineage` and is expected to be reused in the future, Blaze attempts to *automatically cache* the partition if its potential recovery cost is larger than any of the partitions that reside in memory. Similarly, if the partition does not have any future usages, Blaze automatically unpersists the data from the cache storage to quickly acquire free space, like [55]. Note that automatic caching and unpersists consider the full application DAG captured by the `CostLineage`, whereas our ILP considers the potential costs only for a couple of upcoming iterations, to optimize ILP performance 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. 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.

Chapter 6

Implementation

We implement each of the systems as described in this chapter.

6.1 Sponge Implementation

We have implemented Sponge with about 10K lines of Java with support for AWS Lambda, as follows:

Programming interface: To express a stream query as an application DAG, we use Apache Beam [31] application semantics, which is a widely used dataflow programming interface for various systems (e.g., Spark [25], Flink [39], Cloud-Dataflow [8]). In addition, as Beam provides APIs for developers to build associative and commutative operations (e.g., combiners), Sponge can extract this information to build the merge operators.

Compiler: Apache Nemo [135, 115] provides the intermediate representation and optimization pass abstractions, with which we can flexibly optimize application DAGs. We split our operator insertion into three separate optimization

passes for inserting ROs, TOs, and MOs on Nemo to reshape the application DAG, defined by Apache Beam semantics.

Runtime: We modify the Nemo runtime [135, 115] to support the migration of tasks and the redirection of the data from VMs to SFs. Sponge executes worker processes on VM and SF instances, which each manages a thread pool that contains a fixed number of threads and assigns tasks to the threads. VM workers set up Netty [94] network channels and communicate with other VMs and SF workers, while there are no network channels set up between SF instances due to the communication constraint. For launching new VM and SF instances, as well as for deploying the worker code on AWS Lambda, we use boto3, the AWS SDK API for controlling AWS instances [38].

6.2 SWAN Implementation

SWAN is implemented on top of Apache Nemo [135, 115] with around 2K lines of Java code. We use Apache Beam [31] application semantics to express a stream query as an application DAG, which is widely used as a dataflow programming interface for various systems (e.g., Spark [25], Flink [39], CloudDataflow [8]). Apache Nemo [135, 115] provides the intermediate representation and optimization pass abstractions, with which we can flexibly optimize application DAGs. We implement optimization passes to insert relay vertices where applicable. We measure network performances with the iPerf [75] library on linux machines, and use Google OR-tools [99] to implement ILP solvers for our baseline.

6.3 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. We

implement `BlazeBlockManagerEndPoint` in the Spark master that maintains `CostLineage`. We modify the default Spark memory manager, `UnifiedMemoryManager` [22] to cache data in the local executor memory. To communicate with the `BlazeBlockManagerEndPoint`, we also modify the Spark `BlockManager` and `MemoryStore`. The `BlockManager` and `MemoryStore` communicate with the `BlazeBlockManagerEndPoint` to acquire the costs of different partitions and make decisions. 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. The ILP solver is implemented with the Gurobi optimizer 10.0.1 [56].

Chapter 7

Evaluation

We evaluate the performances of the designs proposed for each of the solutions in this chapter.

7.1 Sponge Evaluation

In our evaluation, we observe Sponge performance compared to other scaling mechanisms (§ 7.1.2), distinguish the factors that contribute to the Sponge performance (§ 7.1.3), compare the cold start latency reduction mechanisms (§7.1.4), and observe the latency-cost trade-off between SFs and VMs (§7.1.5).

7.1.1 Methodology

Environment. We use AWS EC2 r5.xlarge instances ($32GB$ of memory and 4 vCores) as VM workers, and AWS Lambda instances as SF workers. As AWS Lambda offers one vCPU per $1,769MB$ and provides constant network bandwidth (i.e., $\sim$ 100Mbps) regardless of the instance size, we use single-core SF in-

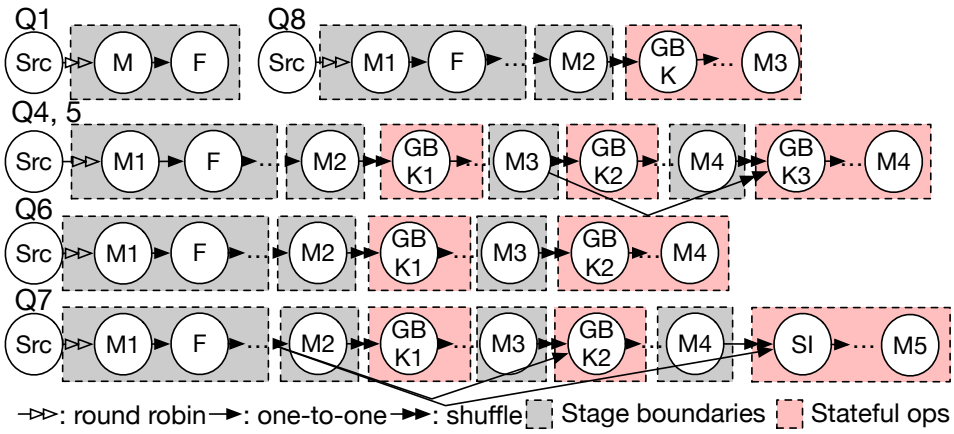

Figure 7.1: A simplified application DAG of stream queries used in our evaluation. `M` and `F` are map and filter operators, `GbK` is a stateful group-by-key operator for incremental aggregation on windows, and `SI` is a non-mergeable stateful operator for the join operation.

stances of $1,769MB$ to provision each instance with enough network bandwidth to achieve the throughput of the CPU core. VMs generally provide 10Gbps networks, which effortlessly cover the traffic generated by the CPU core throughput (i.e., $< 10\%$ bandwidth utilization when offloading 450K events/sec). We set up Amazon Virtual Private Cloud (VPC) for the data communication between the VM and SF instances for stable network connections.

Workloads. NEXMark[119] is a widely used streaming benchmark [67, 79] that analyzes auctions and bid data streams. NEXMark contains diverse stream queries with complex dataflow and stateful operations. Among the 8 (Q1-8) NEXMark queries, we choose 6 queries as shown in Table 7.1 because they represent distinctive data communication patterns and stateless and stateful operations. We omit Q2-3 because Q2 is a stateless query similar to Q1, and Q3 is a non-associative stateful query like Q7.

Query	Stateful	State Size	# of Tasks (per Op.)	Stable input rate
Q1	X	-	120	550 K/s
Q4	O	∼90 MB	60	190 K/s
Q5	O	∼2.4 GB	70	19 K/s
Q6	O	∼73 MB	70	230 K/s
Q7	O	∼1.5 GB	90	15 K/s
Q8	O	∼7 GB	60	60 K/s

Table 7.1: Characteristics of different NEXMark stream queries.

Fig. 7.1 illustrates the simplified original DAG of NEXMark queries, and Table 7.1 summaries the characteristics of the queries. The queries except for Q1 contain windowed operations. We configure the window size of queries as 60 seconds and the window interval as 1 second. While the system throughput declines with larger and more frequent windows, we evaluate under a frequent window interval to test Sponge under requirements for frequent, time-critical resource scaling. The throughput of the evaluated engine [135, 115] is similar to the performance of other stream processing engines [39, 25] when the same window size and interval are used. Nonetheless, in our evaluation, the bursty traffic is increased by up to 10× events/sec and represents a wide range of realistic input rates in the field (§7.1.2).

Baseline. We compare Sponge with the following baselines:

• **NoScaling** executes stream queries on static VMs without scaling out stream queries.

• **VMBase** dynamically creates new VMs and migrates tasks to the new VMs

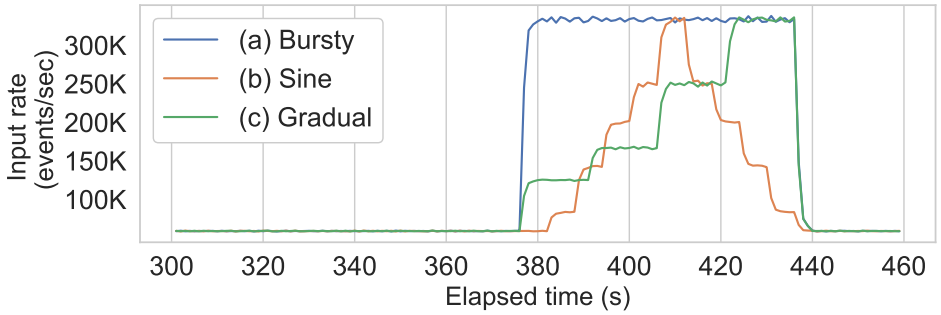

Figure 7.2: Examples of different bursty input patterns used in some experiments, where input rates increase at time $t = 380$. (a) shows a sudden increase from $60K$ to $300K$ ($5\times$) for 60 seconds, (b) shows a sine-curve increase and decrease, and (c) shows a gradual increase.

for scaling without dataflow reshaping.

• **SFBase** dynamically creates SF instances and migrates tasks to SFs for scaling without dataflow reshaping. For SFBase and Sponge, we prevent cold start latencies on SF workers as described in § 3.2.4.

• **VMInit** initializes new VMs in advance and migrates tasks to the new VMs for scaling without dataflow reshaping.

• **Over** over-provisions VMs and already has enough resources to cover input loads without dataflow reshaping.

Bursty traffic and resource allocation. We emulate bursty traffic by increasing the input rate over a short period of time, as shown in Fig. 7.2. In this traffic pattern, we first generate stable input streams where the input rate is stable and does not fluctuate. At a specific point (e.g., $t = 380s$ in our evaluation), we increase the input rate for a short period of time (e.g., 60 seconds) to emulate an increased load and then decrease the rate back to the stable input rate. In § 7.1.2, we observe the average performance of the different systems under

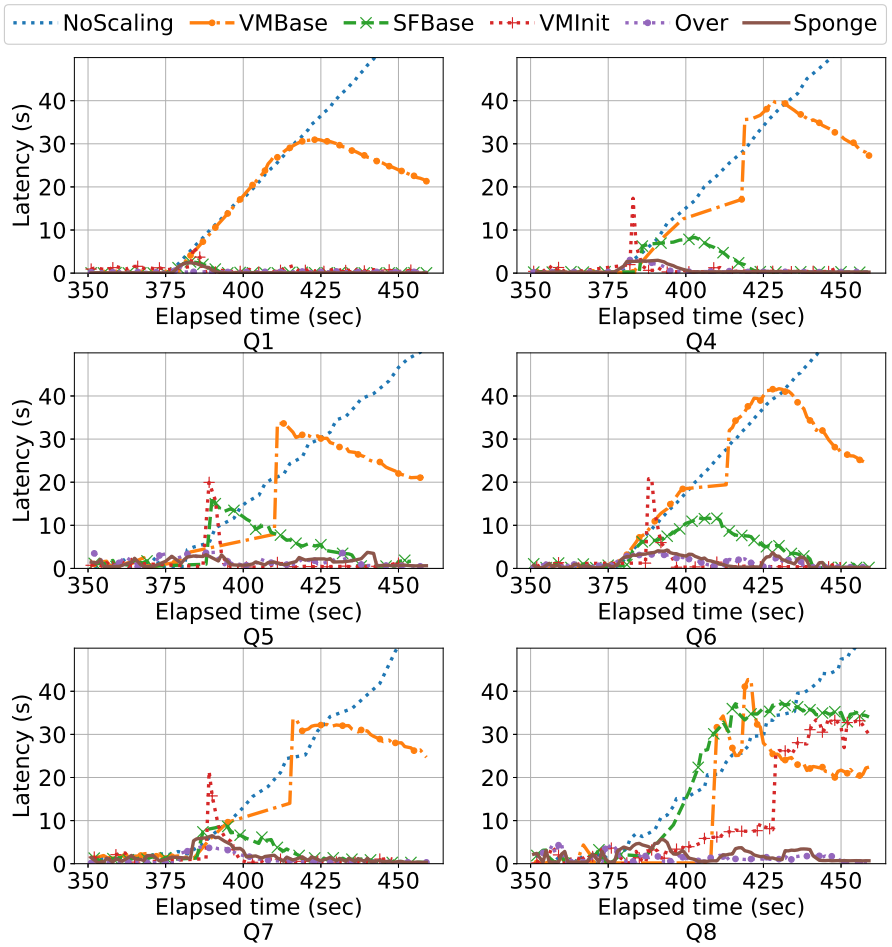

Figure 7.3: The tail latency graph, under a bursty load (Fig. 7.2(a)) at $t = 380s$ and scaling is triggered at $t = 381s$.

up to 10× burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$), and we provide detailed analysis on the effects of the burstiness rising from 3× to 6× in §7.1.3. By default, the burstiness is set to 5×, as it distinctly shows the limitations of existing approaches comparatively. For example, as the stable CPU load is kept at $60 - 80\%$, most baselines already experience high latencies from 2× burstiness, but the performance results are more clearly distinguishable between the baselines under the

5× burstiness.

During the stable load, we run 5 VM workers. We generate events (per second) for the stable load such that all 5 VM workers undergo CPU usage between 60% and 80%, preventing the VM cluster from being under-loaded or over-loaded. As queries have different computational complexity, the stable input rate is configured differently for each query as shown in the last column of Table 7.1. Once bursty loads occur, we dynamically allocate up to 200 single-core SF instances for *Sponge* and *SFBase*, and up to 50(10×) new extra VM instances for *VMBase* depending on the query load.

7.1.2 Performance Analysis

Fig. 7.3 and Fig. 7.4 illustrate the 99^{th}-percentile tail latency and CPU utilization, respectively, of the different systems across different queries for the Burst traffic pattern in Fig. 7.2. Overall, *Sponge* and *Over* exhibit lower latencies compared to others during bursty periods and successfully keep the CPU utilization stable. The latency of *NoScaling* continuously increases with full CPU utilization as the existing VMs are overloaded and never perform offloading. Henceforth, we discuss Sponge and other baselines that perform scaling. For SF-based strategies that are restricted by the prohibited direct communication between SF instances, we profile their operator costs and manually configure them to make the best scaling decisions.

Sponge. Sponge reduces the tail latency on average by 88% compared to *VMBase* and 70% compared to *SFBase* and performs comparably to *Over*. Sponge also keeps the CPU utilization relatively stable across time, as shown in Fig. 7.4. Subsequently, we illustrate below why other scaling strategies cannot deliver the same benefits as Sponge in further detail.

VMBase. The latency of *VMBase* in Fig. 7.3 increases by at least 30s due to

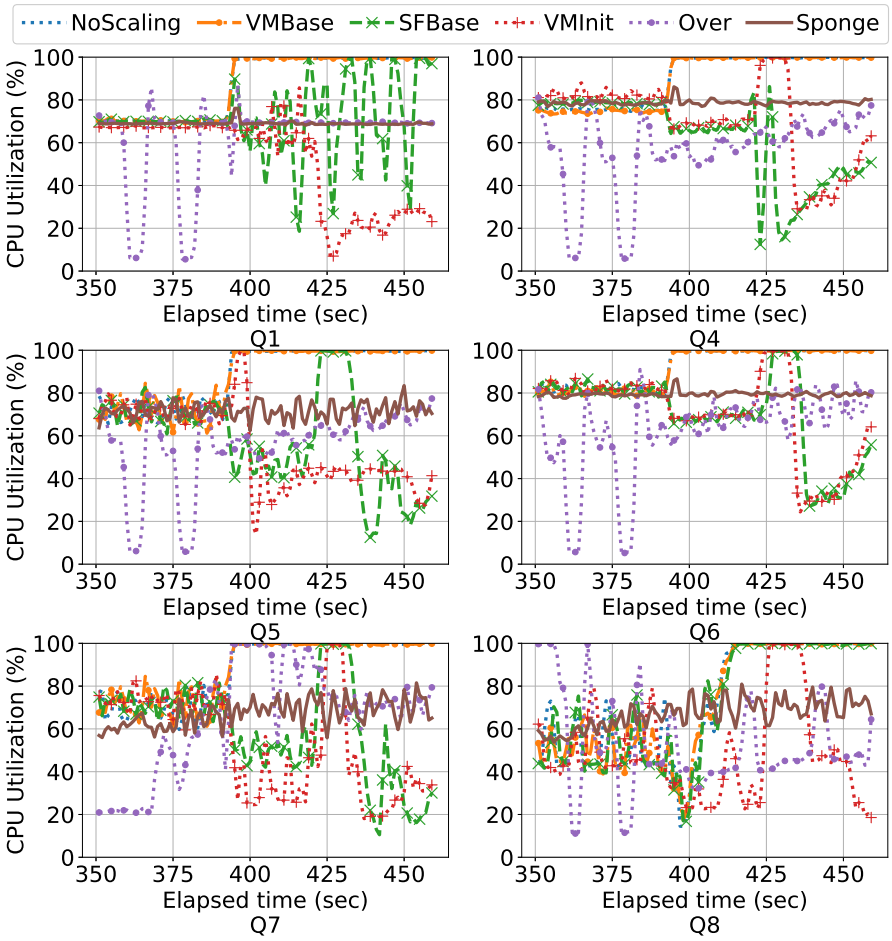

Figure 7.4: The CPU utilization graph, under a bursty load (Fig. 7.2(a)) at $t = 380s$ and scaling at $t = 381s$.

the slow start-up time of the VMs. Specifically, we observe that it takes around 25-30 seconds for the VMs to start, and around 4 extra seconds for managed runtime (i.e., JVM) worker processes to start on the newly started VMs. Moreover, as JVM processes are cold at the beginning and JIT compilation is not triggered, the processing throughput is low in the beginning, which causes extra latency of up to 44 seconds. After new VMs are instantiated, tasks are migrated

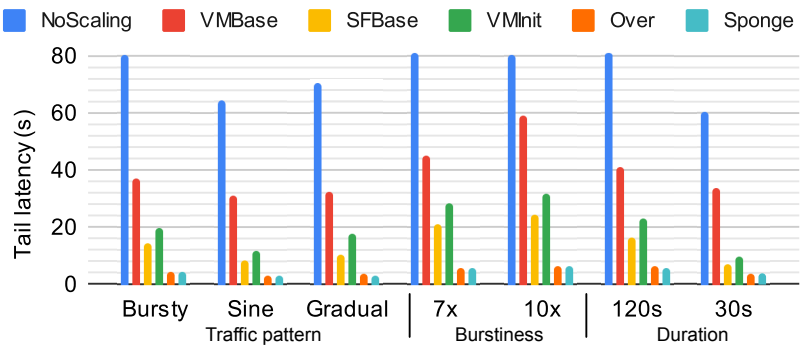

Figure 7.5: Summarized results of the experiments, with similar settings as in Fig. 7.3, displaying the average peak tail latency across the different NEXMark queries under diverse input patterns and burstiness.

to new VMs, and the latency of the VM decreases as the throughput eventually becomes larger than the input rate. Nevertheless, the CPU utilization of *VMBase* shown in Fig. 7.4 is continually kept high after the peak load, as it tries to climb down from the latency peak by heavily processing the data in the event queue.

SFBase. The slow start-up time of VMs can be mitigated by using SFs as shown in *SFBase*. Upon scaling out Q1 (a simple stateless query), *SFBase* significantly reduces the latency and CPU compared to *VMBase*, as the start-up time of an SF instance only takes a few hundred milliseconds in our evaluation. This result suggests that only by using SFs instead of VMs, we can significantly improve the latency for scaling out a simple stateless query, similar to MArk which handles bursty loads of stateless inference jobs [141].

However, for scaling out other complex queries with N-to-N shuffle data communications and stateful operations, the performance gain of *SFBase* compared to *VMBase* declines. It indicates that naïvely scaling queries on SFs without any operator insertion has limitations due to the challenges explained

in § 3.1. In Q4 and Q6, latency increases by up to 12 seconds because the operators with shuffle edges cannot be redistributed to SFs and VMs become the bottleneck. In Q5, Q7, and Q8, latency and CPU spikes are caused by task and state migration overheads.

VMInit. Like *SFBase*, *VMInit* reduces the slow start-up time of VMs by starting them in advance. For *VMInit*, queries with N-to-N shuffle data communications can be offloaded, but we can see that it still incurs task and state migration overheads resulting in short steep peaks of tail latencies and CPU usage, which is highlighted in Q8.

Over. The over-provisioned case is the most expensive solution, providing enough resources for the peak loads without considering an upper bound for runtime costs. In Fig. 7.3, we can see a slight increase in latency as the input load increases, but it soon stabilizes back. The CPU usage in Fig. 7.4 displays an under-utilization before the peak load, but shows an adequate utilization rate afterward, as it is allocated with an adequate amount of resources for the peak load.

Input patterns. In Fig. 7.5, we can see the average tail latency among the different queries along the different input patterns. We can see that *Sponge* and *Over* show good performance among all settings, and *NoScaling* continuous increases in most cases. The sine and gradual bursts show a relatively mild effect compared to others, as their bursts are more gentle. We can see that while 120s and 30s bursts show somewhat similar results, 7× and 10× bursts show higher tail latencies due to the increased load.

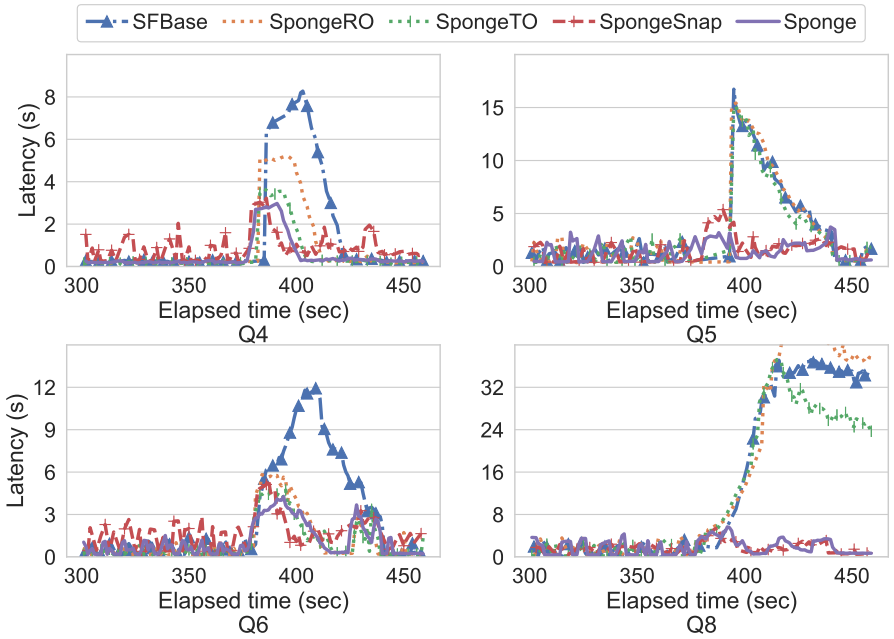

Figure 7.6: The latency graphs for SF, *SpongeRO*, *SpongeTO*, SpongeSnap, and Sponge to analyze and break down the performance improvements of Sponge.

7.1.3 Graph Rewriting Effect

To validate our design, we analyze the performance gain on Sponge with the following additional baselines:

• **SpongeRO** scales queries on SFs while allowing direct communication between SF instances with ROs only.

• **SpongeTO** scales queries on SFs by adding event redirection atop *SpongeRO* with ROs *and* TOs.

• **SpongeSnap** shows performance for Sponge, with ROs, TOs, and MOs, on SnapStart, without pre-warming instances.

Fig. 7.6 illustrates the tail latencies of *SFBase*, *SpongeRO*, *SpongeTO*, *SpongeSnap*, and *Sponge* in more detail. Q1 and Q7 are omitted in the figure, as Q1 is a simple stateless query, and Q7 is represented by Q5 and Q8.

Router operator effect. Comparing *SpongeRO* with *SFBase* shows the effect of router operators. In Q4 and Q6, SFs exhibit higher latencies as VMs are bottlenecked while processing events for `M` operators (Fig. 7.1) on VMs (only 3% of input events are filtered before `M2`). As naive SFs can only offload one of `M` and `GbK`, we choose to offload `GbK`, as the amount of computation on `M` is smaller than that of `GbK` due to the additional aggregation. However, the input rate of `M` on VMs becomes higher than the maximum throughput on the VMs with 5× bursts in Q4 and Q6, and events pile up in `M` operators, incurring latency increases in SFs. In Q5 and Q8, the latency of *SFBase* is similar to *SpongeRO* as VMs sufficiently handle the load on `M` operators. The main bottlenecks in Q5 and Q8 are `GbK` operators, which incur large aggregate computations. This result indicates that the RO is effective when the input rate and the overhead caused by the operators running on VMs are high.

We also evaluate when VMs become bottlenecks on `M` operators, by varying the burstiness ($\frac{\text{bursty input rate}}{\text{stable input rate}}$ from 3 to 6 in Q4 (Fig. 7.7(a)). In the figure, the bottom and top of the box are the 25^{th} and 75^{th} percentiles, the line indicates the median, error bars are the 95% confidence interval, and outliers are dotted as rhombi. VMs sufficiently handle 3× and 4× burstiness, and the latency of *SFBase* does not increase and is similar to *SpongeRO*. However, when the burstiness increases to 6×, VMs become the bottleneck in processing the input events of `M`. Unlike *SFBase*, *SpongeRO* adds an RO between `M` and `GbK`, and migrates both `M` and `GbK` to SFs while keeping the RO on VMs. As an RO does not (de)serialize events and does not perform computation, the amount of

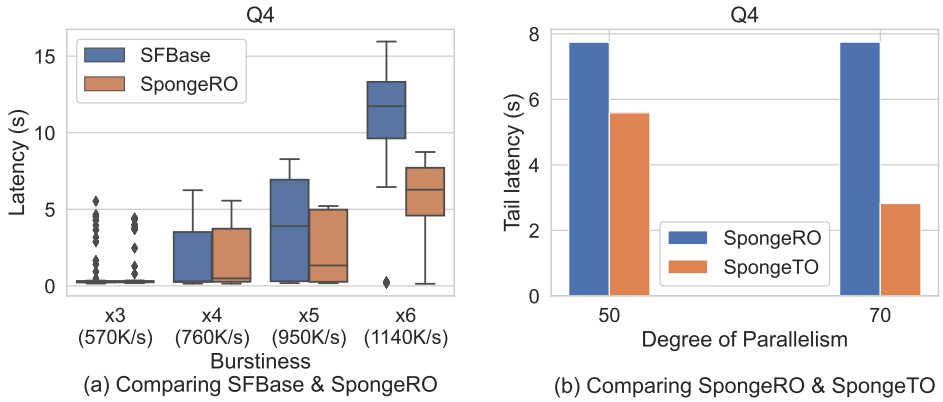

Figure 7.7: Comparison on Q4 for (a) *SFBase* and *SpongeRO* on diverse burstiness, and (b) *SpongeRO* and *SpongeTO* on different degrees of parallelism (# of parallel tasks).

computation of RO is always smaller than that of `M`, and reduces latencies by up to 70%.

Transient operator effect. Transient operators enable Sponge to redirect data without stopping the workload for rescheduling. The effectiveness of transient operators increases as the number of tasks to be migrated (or redirected) increases. Q4 requires a large number of tasks to be migrated or redirected. For Q4, we had to migrate or redirect 85% of total tasks to SF to mitigate the bottleneck in the VMs in *SpongeRO* and *SpongeTO*. *SpongeRO* takes around 2.8 seconds for migrating its tasks. In contrast, *SpongeTO* takes around 1.4 seconds for redirecting its tasks. Due to the fast redirection mechanism, *SpongeTO* additionally reduces the latency by up to 28% compared to *SpongeRO*.

When the degree of parallelism increases, the number of tasks to be migrated or redirected also increases. Fig. 7.7(b) illustrates the tail latency under different degrees of parallelism in Q4. With 50 parallel tasks for each operator,

the task migration/redirection overhead is small, but the latency increases after the migration and redirection in both *SpongeRO* and *SpongeTO*, as a smaller degree of parallelism makes the system more prone to unevenly skewed tasks. With 70 parallel tasks, the overall latency decreases but the task migration overhead increases in *SpongeRO*. As a result, the peak latency increases by up to 8 seconds. In contrast, due to the lightweight redirection optimization, the peak latency of *SpongeTO* is kept at around 3.5 seconds, which is 56% smaller than *SpongeRO*.

Merge operator effect. Even with ROs and TOs, *SpongeTO* still suffers from high latencies in Q5 and Q8 due to the state encoding/decoding overheads. The state migration overhead is trivial in Q4 and Q6 ($<$ 100MB), but the overhead increases with the state size. The time to encode/decode the states of Q5 and Q8 takes around 13s (for $\sim$2.4GB state) and 35s (for $\sim$ 7 GB state), respectively. As a result, the latency of *SpongeTO* increases by up to 15 and 38 seconds in Q5 and Q8. In contrast, Sponge significantly reduces the latencies to 4 seconds in Q5, and to 6 seconds in Q8, preventing state migration overheads with MOs.

7.1.4 Cold Start Latency Reduction Methods

In § 3.2.4, we describe two methods for reducing the cold start latency: by keeping warm SF instances and by using snapshots of SFs through tools like SnapStart [74]. In Fig. 7.6, we can see that the performance of *SpongeSnap*, which solely bases its initialization method on SnapStart [74], is slightly worse, but comparable with *Sponge*, which uses a hybrid of both methods in optimizing the managed runtime (e.g., JVM) initialization overhead. Since the overhead is reduced by more than 80% with SnapStart [74], and $>$ 90% with warm SF

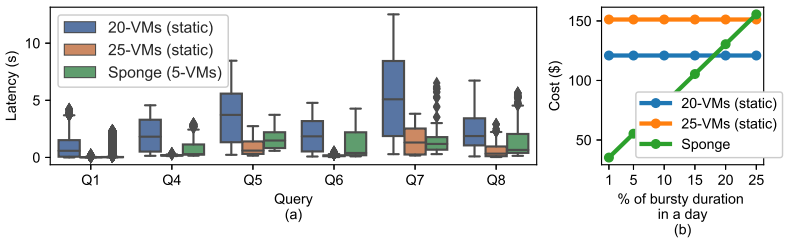

Figure 7.8: (a) The latency during a bursty period, and (b) a rough calculation of the cost according to the % of the bursty duration throughout the day.

instances compared to the original managed runtime initialization methods on SF instances, both methods succeed to timely supply SFs within a sub-second total start-up time.

7.1.5 Latency-Cost Trade-Off

The cost of using SF instances may be higher than over-provisioning VMs when the bursty input persists. In such cases, it makes more sense to launch new VMs while Sponge handles the bursty traffic and offload our tasks to the VM. To investigate the latency-cost trade-off and figure out when it is more beneficial to launch new VMs, we compare the following two VM over-provisioning approaches with Sponge in terms of latency and cost on the workload shown in Fig. 7.2(a). One is *20-VMs (static)*, where 20 VMs are statically allocated without dynamic scaling, and the other is *25-VMs (static)*, where 25 VMs are statically allocated. As the default number of VMs used in *Sponge* is 5, *20-VMs* and *25-VMs* allocate 4× and 5× more VMs compared to *Sponge*, respectively.

Fig. 7.8(a) illustrates the latency of *20-VMs*, *25-VMs*, and *Sponge* during the bursty period. The latency of *Sponge* is in between *20-VMs* and *25-VMs*. Compared to *25-VMs*, which has enough resources to handle 5× bursty loads, Sponge has inherent scaling overheads due to the redirection and migration

protocols. This is why the latency of *Sponge* is slightly higher than *25-VMs*.

In terms of cost, Fig. 7.8(b) shows a rough calculation of cost according to the bursty duration in a day. For instance, 1% of bursty duration represents that bursty loads happen for $24hr * 0.01$ during a day. Basically, the cost of *Sponge* is smaller than others when bursty loads occur in short durations. When the duration of the bursty load is less than 15%, *Sponge* has lower latency and cost compared to *20-VMs*. When the bursty load persists (at more than 25% in Fig. 7.8(b)), the cost of *Sponge* exceeds *25-VMs* due to the high cost of the SF instances. In this case, it is more beneficial to statically over-provision VM resources in terms of latency and cost. Nevertheless, as presented in existing works [84, 93], bursty loads are mostly short-lived, and persistent peaks are comparatively much rare, resulting in their duration generally falling much below 15% of the total time. Sponge provides mechanisms to initially provide prompt scaling with fast-starting SFs regardless of the peak duration and later expands the cluster to additional slow-starting VMs if the peaks persist, making the solution effective with any bursty traffic in terms of both cost and latency.

7.2 SWAN Evaluation

7.2.1 Methodology

Testbed setup. We deploy our system on 16 GCP Compute Engine e2-standard-4 nodes, each equipped with 4vCPUs and 16GB of memory. We launch 2 nodes on each of the 8 regions on three continents: Taiwan, Mumbai (Asia), Finland, Belgium, Netherlands (Europe), Iowa, South Carolina, and Oregon (America). All nodes run Ubuntu 18.04.

Workloads. We measure the performance of queries from the NEXMark Benchmark Suite [119], a popular benchmark containing a large variation of stream processing queries representing an online auction system. Among the different queries, we spotlight query 4, which calculates for the average price for each category, illustrating an example of using join and aggregation, which involves shuffle operations.

Prior approaches in comparison. In addition to our implementation of the scheduling policy for operator placement and query rewriting, we also implement prototypes of existing ILP-based solutions described in both Clarinet [123] and WASP [66]. Among the ILP solutions, I have included the ILP solutions that perform better among the two solutions for the results. In order to compare the effectiveness of the heuristic model, we also run the conventional computation-oriented scheduling algorithm for comparison [39, 135, 115].

Performance metrics. We measure the latency and the throughput of the stream analytics for different queries. For measuring latency, we fix the throughput at a fixed input rate and observe the 95th percentile latency changing over time. For measuring throughput, we maximize the input rate, and observe the system performance under the given conditions.

7.2.2 Throughput and Latency

Fig. 7.9 shows a 95th percentile latency graph comparing the heuristic model with ILP-based model and the conventional scheduling policy over time. In this workload, we set the input rate to 100K events per second, which is about 10MB/s in its actual size. On the graphs, we can see that ILP-based models exhibit the slowest starting time, displaying high latency at the beginning of

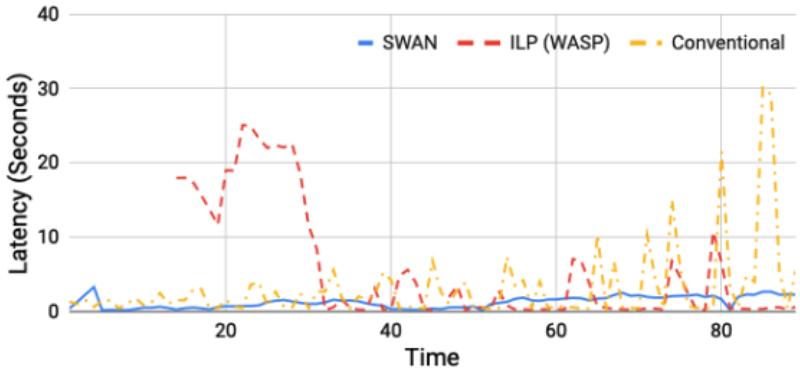

Figure 7.9: A graph of the 95th percentile latency of the workload after triggering optimization at time = 0 of execution of NEXMark benchmark query 4.

the workload. As the time goes on, it performs better than the conventional approach, which gradually degrades over time, as it is designed without the consideration of WAN networks. The heuristic model on SWAN exhibits some latency at the beginning of the job compared to the conventional approach, but the latency is negligible compared to ILP solvers. Among the different solutions, SWAN shows the most stable overall latency throughout the workload, displaying its effectiveness for optimizing operator placement on appropriate WAN.

7.2.3 Query Placement Speed

Fig. 7.10 displays a graph comparing the different approaches of the scheduling algorithms. As described earlier, we can witness a huge overhead, ranging from 20× up to 32× overhead in both ILP cases, compared to the simple heuristic algorithm described in § 4.2.3. We can witness ILPs suffering from higher overheads with larger query execution plans for queries 4, 13, and 14, with $\sigma = 1747ms$ and $\sigma = 1824ms$ each, while the heuristic model exhibits

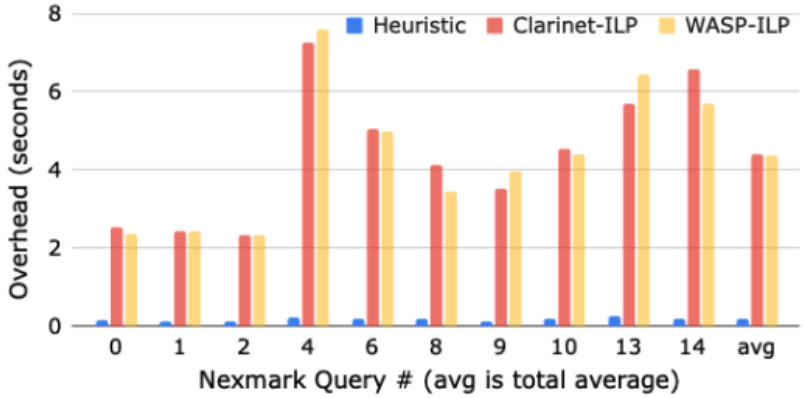

Figure 7.10: A graph comparing the scheduling overhead of the different approaches.

$\sigma = 46.2ms$ throughout the list of different queries.

7.2.4 Effect of Query Rewriting

In order to spotlight the effect of relay tasks upon query rewriting, we measure the data transfer rate of the workload over time for a workload with inserted relay tasks, and another without the optimization, as shown in Fig. 7.11. We launch the job with maximum input rate, and observe the data transfer rate for the workload. On the graph, we can see that the throughput with the insertion of the relay task shows much better performance compared to the original option. While the throughput rate gradually decreases with the network bottleneck for both options, the relay-inserted workload always displays superior performance for the data transfer compared to the original approach.

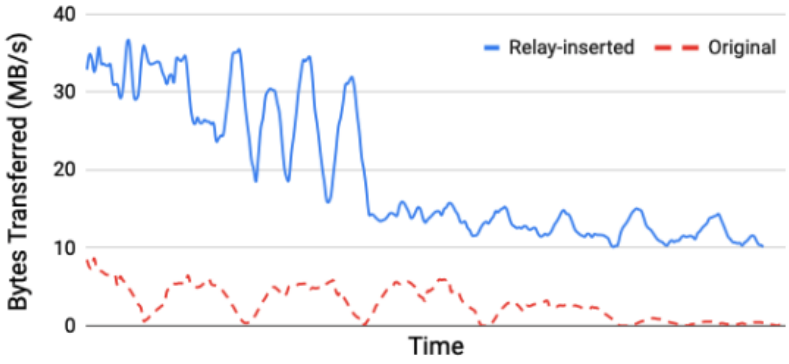

Figure 7.11: A graph of the throughput of the data transfer rate with and without the relay task insertion in NEXMark benchmark query 4.

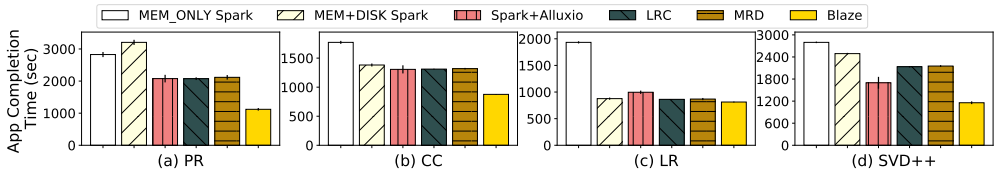

Figure 7.12: An end-to-end performance comparison on `MEM_ONLY Spark`, `MEM+DISK Spark`, `Spark+Alluxio`, `LRC`, `MRD`, and `Blaze` in various applications. We run each application three times and plot the average with an error bar at the top.

7.3 Blaze Evaluation

In our evaluation, we observe Blaze performance compared to other caching mechanisms (§7.3.2), distinguish the factors that contribute to the Blaze performance improvement (§7.3.3), and provide additional details on specific settings and Blaze components (§7.3.4, §7.3.5).

7.3.1 Methodology

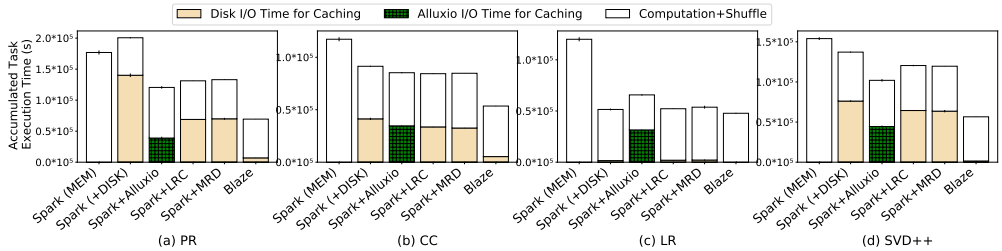

Figure 7.13: A breakdown of cost with the accumulated total task execution times. In `MEM+DISK Spark` (annotated as `Spark (+DISK)`), `LRC`, and `MRD`, the disk I/O time of cached data becomes the cost. In `Spark+Alluxio`, the Alluxio I/O time of cached data becomes the cost, as they are the potential recovery cost experienced from the applications that are run on Spark.

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 [22], 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 two 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 [51] and MLlib [88] libraries [19, 15, 17, 21].

• **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 [96]. We generate a synthetic dataset with 25 million vertices using SparkBench [78].

• **Connected Components (CC):** CC is another graph processing algorithm that finds all connected components in a given graph [43]. We use the same input dataset as in PR.

• **Logistic Regression (LR):** LR is a basic regression algorithm in machine learning [70]. For input data, we use the criteo dataset [118] day 0 data, which is 106 GB, among the 24 days of data. LR also represents supervised iterative ML algorithms like decision trees and support vector machines.

• **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 [72]. We generate a synthetic 31 GB input dataset with a rating data of 15 million users, each with 50 items. SVD++ also represents unsupervised iterative ML algorithms like K-means clustering.

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.

• `Spark + Alluxio:` Alluxio [1] 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) [137] and MRD

(Most Reference Distance) [98] as representative ones. The considered conventional caching algorithms include LRU, FIFO, LFUDA [24, 86], GDWheel [77], TinyLFU [46], and LeCaR [122], and data dependency-aware algorithms include LERC [138], LCRC [126], and LCS [50]. 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 dependencies across multiple jobs. LRC and MRD on `MEM+DISK Spark` are evaluated in § 7.3.2, and on `MEM_ONLY Spark` are evaluated in § 7.3.4.

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 $d_i \rightarrow u_i$ (§ 5.3.2). 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.

7.3.2 Performance Analysis

Fig. 7.12 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 $< 4\%$ of the total ACT. Overall, Blaze achieves 2.52×, 2.02×, 2.38×, and 2.42× speed up compared to `MEM_ONLY Spark`, and 2.86×, 1.57×, 1.08×, and 2.15× speed-up compared to `MEM+DISK Spark` in PR, CC, LR, 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. 7.13, while `MEM_ONLY Spark` does not exhibit any disk usages, Blaze reduces the disk I/O overhead by 95%, 87%, 99%, and 98% for the accumulated value for all tasks on `MEM+DISK Spark` in PR, CC, LR, and SVD++, respectively. This indicates that Blaze uses the fixed memory space efficiently as a caching store.

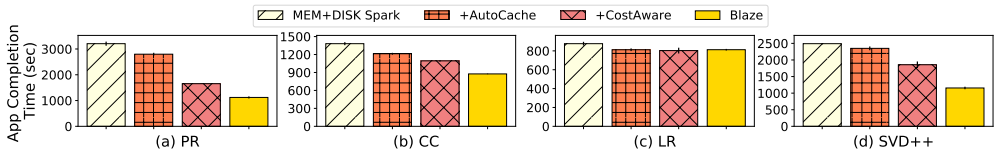

Figure 7.14: A performance breakdown for Blaze.

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. 7.12 (a)). This is because, in PR, aggregate disk I/O time takes 69% of the accumulated total task execution time of `MEM+DISK Spark`, which is the largest percentage among all applications (45%, 2.9%, and 55% in CC, LR, 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% and 97% in PR, CC, LR, 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. 7.12 (c)) which is relatively small, because the main bottleneck comes from the computation, and not the disk I/O overhead, due to fewer references 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 [1].

Interestingly, the amount of cached size of SVD++ is smaller than CC in `MEM+DISK Spark`, but its disk I/O time takes 55% 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× 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.

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.

7.3.3 Performance Breakdown

In this section, we provide a detailed breakdown of the performance gain achieved by Blaze through Fig. 7.14. 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×, and 1.06× in PR, CC, LR 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 §7.3.2. In PR, CC, 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 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×, and 1.27× in PR, CC, 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 in PR, CC and SVD++.

Blaze vs. +CostAware. Comparing Blaze against +CostAware shows the effectiveness of the ILP caching decision solution on Blaze. Compared to +Cost-

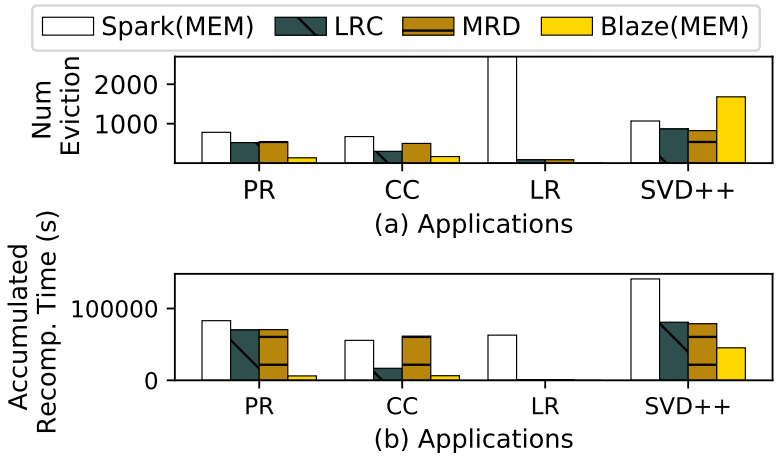

Figure 7.15: The number of evictions and total recomputation time of evicted RDDs while only using memory.

`Aware`, Blaze further accelerates the ACT by 1.47×, 1.25×, and 1.61× in PR, CC, and SVD++, respectively. The main difference between Blaze and `+Cost-Aware` 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 (§5.2.1). 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×, 1.02×, and 2.15× speed-up in ACT compared to `MEM+DISK Spark` in PR, CC, LR, and SVD++, respectively.

7.3.4 Number of Evictions and Recomputation Time

Fig. 7.15 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 (§ 7.3.2). 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.

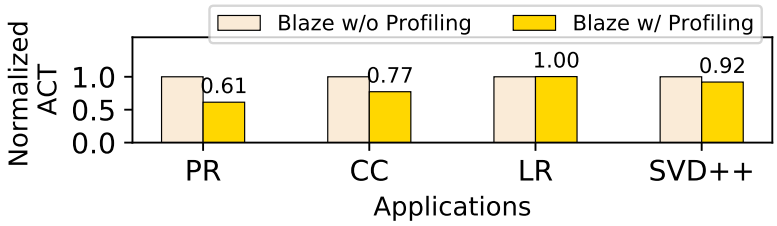

Figure 7.16: The normalized ACT of Blaze with and without dependency profiling, including the profiling overhead.

7.3.5 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. 7.16. 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, as shown in Fig. 7.16. The profiling bases its execution on a $<$ 1MB data from the original input load, and the overhead is upper-bounded by the 10 second timeout, which takes up $<$ 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.

Chapter 8

Related Works

Data communication across SF instances. Researchers have exploited fast-starting SF instances for various workloads such as interactive data analytics [102, 63], video analytics [11, 49], and daily applications [48]. These applications are also represented as DAGs, and shuffle operations are required between SF instances. Their solutions to enable data communication across SF instances enable using additional VM relay servers [49], using HDFS in VMs [63], building an ephemeral storage service [71], and using a NAT-traversal technique [48]. Sponge router operators enable data communication across SF instances preserving event-based stream processing with low latency, without requiring any of the additional VM resources or NAT-traversal technique.

Optimizing state migration. Rhino [45] and ChronoStream [130] replicate states of stream queries across extra (over-provisioned) machines to minimize state migration overheads. Replicating and holding states requires costly over-provisioning of long-running resources like VMs. Holding states on SFs will

cause additional state recovery and cost when SFs are reclaimed by cloud vendors. Megaphone [59] proposes fluid migration that smoothly migrates states from source to destination resources for a long period to reconfigure system configurations. However, when bursty loads happen, the reconfiguration must be executed in a short period of time. As a result, a large amount of state migration is inevitable to quickly migrate the load on VMs. In contrast, Sponge avoids state migration from VMs to SFs by just forwarding data to SFs and merging partial states in SFs into the existing VMs.

Scaling policy. Regarding scaling policies, SEEP [40], StreamCloud [54], and Dhalion [47] use metrics like CPU utilization for their decisions. Systems such as DS2 [67] aim to measure the processing and output rates of individual dataflow operators through spongetem instrumentation. Many of these scaling policies are designed to be agnostic to the underlying scaling mechanisms and resource acquisition schemes. In contrast, the Sponge scaling policy also explicitly considers the characteristics of SF instances and offloads a right amount of computations to keep the CPU utilization high.

Cost-Aware Caching. Cost-aware caching is a widely-used approach to optimize caching and eviction in various fields including web services [36, 33], in-memory key-value stores [77, 32], and CDNs [34, 26], 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 [98, 5] and evictions [132, 137, 98, 138, 126, 50, 133]. 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 § 7.3.

Computation Sharing across Applications. In datacenters, there are many works on sharing computations across different applications to improve the application performance [55, 65, 109, 42]. 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 mem-

ory, evict to CPU memory, or to unpersist and recompute [41, 129, 97, 134, 143]. 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.

Chapter 9

Conclusion and Future Directions

9.1 Conclusion

In this dissertation, I have proposed three systematic approaches that provide solutions for fast, dynamic runtime adaptation for system resources under unpredictable conditions, in terms of CPU, network, and storage in distributed data processing and machine learning.

Specifically, Sponge harnesses SF instances for offloading bursty loads from existing VMs in streaming workloads. Sponge minimizes task migration overheads and addresses data communication constraints on SF instances by inserting new stream operators in the application DAG: router, transient, and merge operators. Sponge also provides an offloading policy that determines when and how to offload the increased input loads. Our evaluation on AWS EC2 and Lambda shows that the Sponge operators are effective in significantly reducing tail latencies in stream processing upon unpredictable bursty loads, compared to existing scaling mechanisms on VMs and SFs.

Next, SWAN is a stream processing system tailored for distributed stream analytics on geo-distributed environments. We point out the problems of spatial and temporal variations that co-exist in WAN settings, and discuss a fast and effective heuristic-based model to solve the problem of scheduling tasks on the different nodes with heterogeneous network conditions in a geo-distributed cluster. In addition, we also discuss a way to optimize the solution further, to add relay tasks appropriately at points where the network can be further optimized by utilizing longer network paths that exhibit more bandwidths, to bypass original supplies of low-bandwidth networks. Our experimental evaluations on latency and throughput show a 77.6% reduction in the average job latency and a 5.64× increase in the throughput rate within seconds.

Finally, 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 90% on average with optimized automatic caching compared to conventional caching mechanisms.

9.2 Future Directions

9.2.1 Centralized vs. Decentralized Designs

SWAN deals with the problems in distributed data processing mainly within a centralized design, to get access to the information that encompass the resources throughout the distributed cluster, and derive an optimal decision based on the global information. Nevertheless, in order to further optimize the system, we may take a decentralized design to deliver faster results to executor machines that require shorter latencies for their SLOs [73]. In doing so, we may further optimize our systems, for example, to combine the global and local information together to derive an estimation of the global optimum value for resource management, while providing asynchronous methods for updating the global information. Moreover, we may consider optimizations on the scheduling of tasks in batch workloads and events in streaming workloads, to prevent the wait time for the acks and execution results on the central coordinator machine, and keep the workers as busy as possible at all times. While Sponge and Blaze take similar approaches to deliver fast optimization for resource management in the executor machines, we can take closer observations on system performances and optimize the systems, both in terms of the control plane and the data plane, to explore for further optimization opportunities.

9.2.2 Increasing The Model Complexity

While we consider the factors that we have observed as the main bottleneck within the context of each of the solutions, there may also be other factors that contribute to the system performances. In order to further optimize our solutions, we can put these unobserved factors into consideration within our performance analytics and proposed methods for mathematical modeling, to deliver even faster job execution, on top of the solutions proposed in this dis-

sertation. Also, we may simultaneously take all of these resources into consideration for optimization as well. By doing so, we may increase the complexity of the mathematical models, which also leaves room for further optimizations on the performances of the model calculation, e.g., ILP solver performances. It is also widely known that ILP solvers exhibit large overheads when it attempts to solve for more complex models, and in such cases we may convert the problem into LP solvers and approximate for a near-optimal solution. While we propose solutions that provide satisfying results for resolving the sources of main bottlenecks that had been observed in each of the works, we may also extend our solutions to find an optimal balance point between the cost-performance trade-off for each of the models, for example, by using techniques for finding Pareto optimum values.

9.2.3 Expanding to Other Resources and New Hardwares

With the advent of various deep learning and artificial intelligence workloads, GPU resources had become one of the main resources in modern data processing workloads for system optimization. While we deal with the traditional computational resources in this dissertation, like CPU, network, and memory, we may take the knowledge and information on other specific environments into consideration (e.g., deep learning on GPUs), and use similar mathematical modeling and analytical approaches to optimize GPU resources, as well as under environments equipped with other new hardwares like Non-Volatile Memory Express (NVMe) drivers and Remote Direct Memory Access (RDMA) protocols for the problems that occur in relevant domains. Furthermore, we may also make use of the new technologies and hardwares to further optimize our systems to bring faster results on top of the mechanisms proposed in this dissertation, which we leave as future work.

초록

오늘 날, 다양한 전자기기가 지속적으로 보편화 됨에 따라 전 세계 각지에서 더 많은 대량의 데이터가 생성되고 있다. 이러한 데이터의 종류나 양이 점점 더 증가함에 따라 빅데이터 처리는 꾸준한 서비스 제공 시간 등의 기준을 충족하면서 유의미한 정보를 추출하기 위해 시스템에 지속적으로 보다 높은 처리량 성능을 요구한다. 현재 빅데이터 처리는 크게 대규모 배치 데이터 또는 실시간 데이터 스트림을 처리하는 용도로 나뉠 수 있으며, 이들 분산 처리의 대상이 되는 데이터는 각각의 이유로 예측이 불가능하며, 실행 중 빠르게 그 특성이 변할 수 있는 여러가지 요소들을 내재하고 있다. 이처럼 다양한 환경에 동적으로 최적화하여 만족스러운 성능을 보장하기 위해서는 *CPU*, *네트워크*, *메모리* 등 리소스에 대한 효율적이고 유연한 관리가 필수적인 경우들이 많이 존재한다.

구체적인 예시로는 실시간 이벤트 데이터 스트림의 경우, 우리가 흔히 급작스럽고 예측하지 못하는 자연재해나 테러, 경제위기 등의 상황들을 직면했을 때 데이터 트래픽이 기하급수적으로 증가할 수 있으며, 이와 같은 경우에서는 데이터 처리량을 늘리기 위해 *CPU* 성능을 동적으로 늘릴 수 있어야 한다. 또 다른 예시로는 글로벌 서비스 등에서 세계 각지에서 흩어져서 생성된 데이터를 분석하여 사용자에게 유용한 실시간 정보를 적시에 제공하기 위해서는 다양한 대역폭의 *네트워크*를 통해 데이터를 수집하여 특정 통계값 등으로 요약해야 되는 경우가 있는데, 이 때 안정적이지 않은 장거리 네트워크를 이용해야 하는 경우들이 흔하게 존재한다. 또한, 머신러닝이나 그래프처리와 같이 대량의 중간 데이터가 축적되는 반복적인 워크로드를 처리하는 경우에는 재사용 가능한 데이터를 캐싱하여 반복적인 재계산 작업을 없앰으로써 워크로드를 최적화할 수 있는데, 이 때 캐시 스토리지에 저장된 입력 데이터를 필요할 때마다 작업에 제때 제공하기 위해서는 *메모리* 리소스 관리가 매우 중요하다. 이와 같은 환경 아래에서 위와 같은 문제는

일반적으로 미리 예측할 수가 매우 힘들고 런타임 중에 동적으로 변경되는데, 이 때 데이터 처리를 위한 자원을 제 때 제공하지 않으면 위와 같은 상황 아래에서 막대한 성능 손실을 초래할 수 있다.

본 논문에서는 이러한 예측 불가능한 자원 문제를 해결하기 위해 클라우드 환경 상의 자원을 수학적 모델링과 분석적 접근 등을 통해 효율적이고 동적으로 사용함으로써 시스템 상의 자원 부족과 병목 현상을 극복하는 동적 자원 관리 기법을 스펀지, 스완, 블레이즈라는 시스템을 통해 제안한다. 스펀지는 입력 부하가 산발적이고 순간적으로 증가하는 상황에서 서버리스 인스턴스로부터 리소스를 확보하여 1초 미만의 지연 시간으로 시스템에 추가 CPU를 제공함으로써 CPU 리소스 부족에 빠르게 동적으로 적응할 수 있도록 한다. 스완은 서로 다른 네트워크 연결의 다양한 대역폭 용량을 동적으로 측정하고 분석하여 제한된 네트워크 리소스를 완화하고 전 세계에 흩어져 있는 환경에서 데이터가 한 곳에서 다른 곳으로 효율적으로 이동할 수 있도록 최적의 경로와 연산자 배치 방법을 찾는다. 블레이즈는 파티션 메트릭의 실시간 추적과 캐시 사용량 및 오버헤드에 대한 정교한 예측을 기반으로 자동 캐싱 메커니즘을 제공하여 반복적인 데이터 처리 워크로드를 위해 제한된 메모리 리소스를 적시에 캐싱에 효율적으로 사용할 수 있도록 한다.

실험 수행 결과, 이 논문에서 제시하는 동적 리소스 관리 방식을 통해 분산된 데이터 처리 시스템에 항상 충분한 리소스를 제공하고 리소스 및 입력 부하가 동적으로 변화하는 환경에 맞춰 제한된 리소스를 효율적으로 사용함으로써 기존 시스템 대비 처리량, 지연 시간(스트리밍 워크로드), 엔드투엔드 완료 시간(배치 워크로드) 측면에서 각각 최대 5.64× 증가, 88% 감소, 2.86× 속도 향상 등의 결과로 시스템 성능을 크게 향상시키는 것을 관찰할 수 있다.

주요어: 분산 시스템, 빅데이터, 머신러닝, 클라우드 컴퓨팅, 리소스 관리, 스케줄링

학번: 2017-28182

Bibliography

[1] Alluxio - Data Orchestration for the Cloud. `https://www.alluxio.io`.

[2] Amazon AWS. `https://aws.amazon.com`.

[3] Google Cloud Dataflow. `https://cloud.google.com/dataflow`.

[4] iostat. `https://github.com/sysstat/sysstat`.

[5] M. Abdi, A. Mosayyebzadeh, M. H. Hajkazemi, A. Turk, O. Krieger, and P. Desnoyers. Caching in the multiverse. In *HotStorage*, pages 1–7, 2019.

[6] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 419–434, USA, 2020. USENIX Association.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at internet scale. *VLDB Journal*, 6(11):1033–1044, 2013.

[8] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *VLDB Journal*, 8(12):1792–1803, 2015.

[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, 2013.

[10] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[11] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Apache Commons Math. `https://commons.apache.org/proper/commons-math/`, 2023.

[13] Apache Software Foundation. Apache Nemo, 2023. `https://nemo.apache.org`.

[14] Apache Software Foundation. Apache Spark. `https://spark.apache.org`, 2023.

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

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

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

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

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

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

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

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

[23] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 480–491, 2004.

[24] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Perform. Eval. Rev.*, 27(4):3–11, mar 2000.

[25] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[26] N. Atre, J. Sherry, W. Wang, and D. S. Berger. Caching with delayed hits. In *ACM SIGCOMM*, pages 495–513, 2020.

[27] AWS. Configuring Lambda function options, 2023. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html.

[28] AWS Lambda. https://aws.amazon.com/lambda, 2023.

[29] Azure Function. https://docs.microsoft.com/en-us/azure/azure-functions/, 2023.

[30] Various Traffics in the Cloud. https://intercept.cloud/en/news/checklist-part-1-choose-your-strategy-before-you-migrate-to-azure/, 2023.

[31] Apache beam. https://beam.apache.org/.

[32] N. Beckmann, H. Chen, and A. Cidon. {LHD}: Improving cache hit rate by maximizing hit density. In *NSDI*, pages 389–403, 2018.

[33] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. Robinhood: Tail latency aware caching – dynamic reallocation from cache-rich to cache-poor. In *OSDI*, pages 195–212, 2018.

[34] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *NSDI*, pages 483–498, 2017.

[35] L. Bindschaedler, J. Malicevic, N. Schiper, A. Goel, and W. Zwaenepoel. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[36] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[37] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda. Systemml: Declarative machine learning on spark. *VLDB Endowment*, 9(13):1425–1436, 2016.

[38] AWS SDK for Python (Boto3). https://boto3.amazonaws.com/v1/documentation/api/latest/index.html, 2023.

[39] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[40] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 725–736, New York, NY, USA, 2013. Association for Computing Machinery.

[41] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[42] A. Chung, S. Krishnan, K. Karanasos, C. Curino, and G. R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *OSDI*, pages 1205–1223, 2020.

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

[44] Stream Processing with IoT Data: Challenges, Best Practices, and Techniques. https://www.confluent.io/blog/stream-processing-iot-data-best-practices-and-techniques/, 2023.

[45] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. Rhino: Efficient Management of Very Large Distributed State for Stream Processing Engines.

In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2471–2486, New York, NY, USA, 2020. Association for Computing Machinery.

[46] G. Einziger, R. Friedman, and B. Manes. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage*, 13(4), nov 2017.

[47] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[48] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[49] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association.

[50] Y. Geng, X. Shi, C. Pei, H. Jin, and W. Jiang. Lcs: an efficient data eviction strategy for spark. *International Journal of Parallel Programming*, 45(6):1285–1297, 2017.

[51] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

[52] Google Cloud Function. https://cloud.google.com/functions/docs/, 2023.

[53] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang. Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 539–556, Carlsbad, CA, July 2022. USENIX Association.

[54] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[55] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, pages 1–8, 2010.

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

[57] B. Heintz, A. Chandra, and R. K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 133–144, 2015.

[58] B. Heintz, A. Chandra, and R. K. Sitaraman. Trading timeliness and accuracy in geo-distributed streaming analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 361–373, 2016.

[59] M. Hoffmann, A. Lattuada, and F. McSherry. Megaphone: latency-conscious state migration for distributed streaming dataflows. *Proceedings of the VLDB Endowment*, 12(9):1002–1015, 2019.

[60] Y. Huang, X. Yan, G. Jiang, T. Jin, J. Cheng, A. Xu, Z. Liu, and S. Tu. Tangram: Bridging immutable and mutable abstractions for distributed data analytics. In *USENIX ATC*, pages 191–206, 2019.

[61] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[62] S. Islam, S. Venugopal, and A. Liu. Evaluating the Impact of Fine-Scale Burstiness on Cloud Elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 250–261, New York, NY, USA, 2015. Association for Computing Machinery.

[63] A. Jain, A. F. Baarzi, G. Kesidis, B. Urgaonkar, N. Alfares, and M. Kandemir. SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 236–250, New York, NY, USA, 2020. Association for Computing Machinery.

[64] J. Jiang, S. Sun, V. Sekar, and H. Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 393–406, 2017.

[65] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *ACM SIGMOD*, pages 191–203, 2018.

[66] A. Jonathan, A. Chandra, and J. Weissman. Wasp: wide-area adaptive stream processing. In *Proceedings of the 21st International Middleware Conference*, pages 221–235, 2020.

[67] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, Carlsbad, CA, Oct. 2018. USENIX Association.

[68] F. Kalim, L. Xu, S. Bathey, R. Meherwal, and I. Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 249–262, 2018.

[69] Q. Ke, M. Isard, and Y. Yu. Optimus: A Dynamic Rewriting Framework for Data-Parallel Execution Plans. In *Eurosys 2013*. ACM, April 2013.

[70] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein. *Logistic regression*. Springer, 2002.

[71] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, Oct. 2018. USENIX Association.

[72] Y. Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *ACM SIGKDD*, pages 426–434, 2008.

[73] F. Lai, J. You, X. Zhu, H. V. Madhyastha, and M. Chowdhury. Sol: Fast distributed computation over slow networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 273–288, Santa Clara, CA, Feb. 2020. USENIX Association.

[74] Lambda SnapStart. `https://docs.aws.amazon.com/lambda/latest/dg/snapstart.html`, 2023.

[75] Lawrence Berkeley National Laboratory. iPerf. `https://github.com/esnet/iperf`, 2023.

[76] K. Leetaru, S. Wang, G. Cao, A. Padmanabhan, and E. Shook. Mapping the global twitter heartbeat: The geography of twitter. *First Monday*, 2013.

[77] C. Li and A. L. Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *EuroSys*, pages 1–15, 2015.

[78] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *ACM ICCF*, pages 1–8, 2015.

[79] S. Li, P. Gerver, J. MacMillan, D. Debrunner, W. Marshall, and K.-L. Wu. Challenges and Experiences in Building an Efficient Apache Beam Runner for IBM Streams. *Proc. VLDB Endow.*, 11(12):1742–1754, aug 2018.

[80] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. In *2011 Proceedings IEEE INFOCOM*, pages 1098–1106, 2011.

[81] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 439–453, 2016.

[82] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 383–400, USA, 2016. USENIX Association.

[83] H. H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang. Efficiently delivering online services over integrated infrastructure. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 77–90, 2016.

[84] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proceedings of the VLDB Endowment*, pages 1303–1316, 2018.

[85] M. Mastrolilli and O. Svensson. (acyclic) job shops are hard to approximate. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 583–592. IEEE, 2008.

[86] D. Mátáni, K. Shah, and A. Mitra. An o(1) algorithm for implementing the lfu cache eviction scheme. *ArXiv*, abs/2110.11602, 2021.

[87] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, Mar. 2003. USENIX Association.

[88] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[89] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting Realistic Burstiness to a Traditional Client-Server Benchmark. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, page 149–158, New York, NY, USA, 2009. Association for Computing Machinery.

[90] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 167–181, New York, NY, USA, 2019. Association for Computing Machinery.

[91] M. MONALDO and S. OLA. Improved bounds for flow shop scheduling. ICALP, 2009.

[92] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings*

of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

[93] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, Dec. 2008. USENIX Association.

[94] Netty. `http://netty.io/`, 2023.

[95] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, 2015.

[96] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[97] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *ASPLOS*, page 891–905, 2020.

[98] T. B. G. Perez, X. Zhou, and D. Cheng. Reference-distance eviction and prefetching for cache management in spark. In *ICPP*, pages 1–10, 2018.

[99] L. Perron and V. Furnon. Or-tools, 2023.

[100] R. Potharaju, T. Kim, W. Wu, V. Acharya, S. Suh, A. Fogarty, A. Dave, S. Ramanujam, T. Talius, L. Novik, and R. Ramakrishnan. Helios:

Hyperscale Indexing for the Cloud & Edge. *Proc. VLDB Endow.*, 13(12):3231–3244, Aug 2020.

[101] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[102] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, Feb. 2019. USENIX Association.

[103] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14, 2013.

[104] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006.

[105] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 275–288, 2014.

[106] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe. Gloss: Seamless Live Reconfiguration and Reoptimization of Stream Programs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, AS-

PLOS '18, page 98–112, New York, NY, USA, 2018. Association for Computing Machinery.

[107] S. H. Rastegar, A. Abbasfar, and V. Shah-Mansouri. Rule Caching in SDN-Enabled Base Stations Supporting Massive IoT Devices With Bursty Traffic. *IEEE Internet of Things Journal*, 7(9):8917–8931, 2020.

[108] B. Robinson, R. Power, and M. Cameron. A Sensitive Twitter Earthquake Detector. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, page 999–1002, New York, NY, USA, 2013. Association for Computing Machinery.

[109] A. Roy, A. Jindal, H. Patel, A. Gosalia, S. Krishnan, and C. Curino. Sparkcruise: Handsfree computation reuse in spark. *Proc. VLDB Endow.*, 12(12):1850–1853, 2019.

[110] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *ACM SIGMOD*, page 1357–1369, 2015.

[111] A. Sandur, C. Park, S. Volos, G. Agha, and M. Jeon. Jarvis: Large-scale Server Monitoring with Adaptive Near-data Processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1408–1422. IEEE, 2022.

[112] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 25–36, 2003.

[113] W. W. Song, M. Jeon, and B.-G. Chun. SWAN: WAN-Aware Stream Processing on Geographically-Distributed Clusters. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 78–84, New York, NY, USA, 2022. Association for Computing Machinery.

[114] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun. Sponge: Fast reactive scaling for stream processing with serverless frameworks. In *USENIX ATC*, 2023.

[115] W. W. Song, Y. Yang, J. Eo, J. Seo, J. Y. Kim, S. Lee, G. Lee, T. Um, H. Cho, and B.-G. Chun. Apache nemo: A framework for optimizing distributed data processing. *ACM Trans. Comput. Syst.*, 38(3–4), oct 2021.

[116] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, E. Witchel, et al. Learning relaxed belady for content distribution network caching. In *NSDI*, pages 529–544, 2020.

[117] Taking Advantage of a Disaggregated Storage and Compute Architecture. `https://databricks.com/session/taking-advantage-of-a-disaggregated-storage-and-compute-architecture`.

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

[119] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark—a benchmark for queries over data streams draft. Technical report, Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[120] T. Um, G. Lee, and B.-G. Chun. Pluto: High-performance iot-aware stream processing. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 79–91, 2021.

[121] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 374–389, New York, NY, USA, 2017. Association for Computing Machinery.

[122] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[123] R. Viswanathan, G. Ananthanarayanan, and A. Akella. Clarinet:wan-aware optimization for analytics queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 435–450, 2016.

[124] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese. Wanalytics: Geo-distributed analytics for a data intensive world. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1087–1092, 2015.

[125] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 323–336, 2015.

[126] B. Wang, J. Tang, R. Zhang, W. Ding, and D. Qi. 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)*, pages 956–963, 2018.

[127] H. Wang, D. Niu, and B. Li. Dynamic and decentralized global analytics via machine learning. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 14–25, 2018.

[128] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang. Elasticutor: Rapid Elasticity for Realtime Stateful Stream Processing. In *the ACM International Conference on Management of Data conference (SIGMOD)*. ACM, 2019.

[129] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *ACM SIGPLAN PPoPP*, page 41–53, 2018.

[130] Y. Wu and K.-L. Tan. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st International Conference on Data Engineering*, pages 723–734, 2015.

[131] D. Xu, X. Liu, and A. V. Vasilakos. Traffic-aware resource provisioning for distributed clouds. *IEEE Cloud Computing*, 2(1):30–39, 2015.

[132] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *IEEE IPDPS*, pages 383–392, 2016.

[133] Y. Xu, L. Liu, and Z. Ding. Dag-aware joint task scheduling and cache management in spark clusters. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 378–387, 2020.

[134] D. Yang and D. Cheng. Efficient gpu memory management for nonlinear dnns. In *ACM HPDC*, page 185–196, 2020.

[135] Y. Yang, J. Eo, G.-W. Kim, J. Y. Kim, S. Lee, J. Seo, W. W. Song, and B.-G. Chun. Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 177–190, Renton, WA, July 2019. USENIX Association.

[136] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, page 1–14, 2008.

[137] Y. Yu, W. Wang, J. Zhang, and K. Ben Letaief. Lrc: Dependency-aware cache management for data analytics clusters. In *INFOCOM*, pages 1–9, 2017.

[138] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. Lerc: coordinated cache management for data-parallel systems. In *IEEE GLOBECOM*, pages 1–6, 2017.

[139] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[140] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252, 2018.

[141] C. Zhang, M. Yu, W. Wang, and F. Yan. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[142] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pages 377–392, 2017.

[143] B. Zheng, N. Vijaykumar, and G. Pekhimenko. Echo: Compiler-based gpu memory footprint reduction for lstm rnn training. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 1089–1102. IEEE, 2020.