WASP: Wide-area Adaptive Stream Processing

Albert Jonathan
University of Minnesota - Twin Cities
albert@cs.umn.edu

Abhishek Chandra
University of Minnesota - Twin Cities
chandra@cs.umn.edu

Jon Weissman
University of Minnesota - Twin Cities
jon@cs.umn.edu

Abstract

Adaptability is critical for stream processing systems to ensure stable, low-latency, and high-throughput processing of long-running queries. Such adaptability is particularly challenging for wide-area stream processing due to the highly dynamic nature of the wide-area environment, which includes unpredictable workload patterns, variable network bandwidth, occurrence of stragglers, and failures. Unfortunately, existing adaptation techniques typically achieve these performance goals by compromising the quality/accuracy of the results, and they are often application-dependent. In this work, we rethink the adaptability property of wide-area stream processing systems and propose a resource-aware adaptation framework, called WASP. WASP adapts queries through a combination of multiple techniques: task re-assignment, operator scaling, and query re-planning, and applies them in a WAN-aware manner. It is able to automatically determine which adaptation action to take depending on the type of queries, dynamics, and optimization goals. We have implemented a WASP prototype on Apache Flink. Experimental evaluation with the YSB benchmark and a real Twitter trace shows that WASP can handle various dynamics without compromising the quality of the results.

1 Introduction

Wide-area data analytics has gained much attention in recent years due to the emerging applications that need to extract insights from large amounts of data generated in a geo-distributed fashion for operational decisions. For example, Internet service providers are monitoring system logs from thousands CDN servers to ensure low-latency service delivery [28, 39, 43]. Public services are also analyzing live video streams from cameras installed all over a city for traffic control and surveillance [4, 5, 40]. Since many of these applications rely on timely information, achieving low-latency analysis is crucial.

A wide-area streaming analytics system typically comprises multiple geo-distributed clusters, i.e., *edge clusters* and data centers, that are connected by a wide-area network (WAN) [20, 24, 49, 62]. For such systems, achieving low-latency and high-throughput processing is key to extract insights in a timely manner. However, ensuring a stable execution of long-running queries in a wide-area environment is very challenging due to the variability and unpredictable nature of both workload and WAN bandwidth. Recent work has shown that WAN bandwidth may change at an interval of minutes [59, 62], while Internet workload exhibits strong variability, both temporally and spatially [18, 37]. Furthermore, stragglers and failures are inevitable in large-scale distributed systems [9, 10, 45].

Existing work has addressed the importance of adaptability in distributed stream processing systems, but has mainly focused on the computational bottlenecks in a cluster environment and hence, is WAN-agnostic [13, 19, 38, 48, 60]. As argued by recent work, existing cluster-based policies that lack WAN awareness may result in significant performance loss and/or wasteful resource utilization due to the fundamental differences in the environments [47, 57, 58].

Early work in wide-area data analytics has focused on short-lived *batch* processing and assumed that network bandwidth is relatively stable throughout the runtime of queries [47, 57, 58], which is an invalid assumption for long-running streaming queries. Others have also addressed the importance of adaptability in the context of stream processing, but often require users to trade quality/accuracy for performance through aggregation, degradation, and statistical estimation [21, 35, 49, 62]. We argue that these approaches are application dependent, and may not apply generally. For example, reducing a frame rate may be applicable for some video analytics applications, but dropping data is not tolerable for queries that require high accuracy such as fraud detection, global stock or transactional analysis, and billing queries [3, 7]. Furthermore, determining the *right* accuracy-performance trade-off typically relies heavily on analysts' expertise and involve extensive parameter tuning, making it cumbersome in practice.

In this work, we rethink the adaptability property of wide-area stream processing systems. Our goal is to allow queries to maintain low-latency execution while preserving the quality of the results in the face of dynamics as far as possible. This is especially critical for queries that require high accuracy or those that cannot tolerate significant quality loss. Furthermore, it is critical to ensure the systems achieve such performance while utilizing the resources efficiently to reduce operational cost. Thus it is desirable for such systems to automatically handle inefficient resource utilization and misconfiguration due to workload/environment changes.

To address the above challenges, we propose a resource-aware adaptation framework called WASP (**W**ide-area **A**daptive **S**tream **P**rocessing). In contrast to most existing work that largely rely on one single adaptation technique (data degradation) to handle runtime dynamics, WASP employs multiple adaptation techniques by re-optimizing the execution of a query and adjusting its resource allocation,

Doi: 10.1145/3423211.3425668

using data degradation as a last resort. It adapts queries at runtime through a combination of multiple techniques: (1) task re-assignment, (2) operator scaling, and (3) query re-planning. Both task re-assignment and operator scaling adapt the physical execution of a query, while query re-planning adapts the logical plan. We show that a combination of these techniques are generally applicable for different types of queries. WASP can automatically determine *how* to adapt a query depending on the type of dynamics, whether resource or workload variation, occurrence of stragglers, failures, and resource misconfiguration. For example, WASP handles a compute-constrained task by *scaling up* its bottleneck operator *within* a site, but it handles network bottlenecks differently by *scaling out* the bottleneck operator *across* sites to distribute the workload over multiple network links. The key challenge here is to ensure an effective and efficient adaptation, i.e., resolve bottlenecks without hoarding resources.

There are several novel challenges in adapting a query in wide-area settings as opposed to intra-data-center settings. First, the system needs to account for the limited WAN bandwidth as it is typically the bottleneck in a wide-area environment. Secondly, wide-area network links are highly heterogeneous in terms of their bandwidth capacity and latency as they may vary by two orders of magnitude [23, 30]. Thirdly, wide-area dynamics may happen frequently and unpredictably. Hence, any adaptation should be done with low overhead. Fourthly, it is critical to ensure a stable query execution without over-allocating resources to avoid wasteful resource consumption. Lastly, determining the *right* adaptation is very challenging since it depends on a lot of factors and different adaptation actions result in different trade-offs.

To handle these challenges, we study the applicability, overhead, and benefits of different adaptation techniques, and propose a practical approach to determine *which* adaptation action to take based on the type of queries (stateless vs. stateful), bottlenecks (compute vs. network), and optimization objectives. To ensure low-overhead adaptation, WASP incorporates a localized checkpointing mechanism along with a network-aware state migration and state partitioning technique in adapting queries with stateful operators. We have implemented a WASP prototype on Apache Flink [13]. Experimental evaluation using the YSB benchmark [15] and a real Twitter trace demonstrates that WASP can maintain low-latency execution without compromising quality and with low overhead in the face of dynamics.

We summarize our contributions as follows:

• We propose a network-aware adaptation framework that is able to adapt query execution and resources in the face of dynamics in wide-area streaming analytics.

• We qualitatively compare the applicability, overhead, and benefits between different adaptation techniques, and propose an adaptation policy that can automatically determine *which* adaptation to use depending on the types of queries, dynamics, and goals.

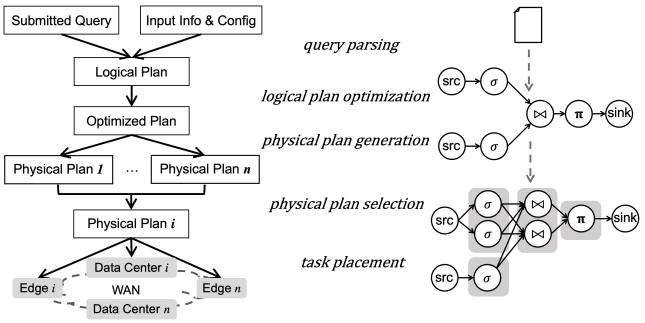


Figure 1. Wide-area query execution pipeline.

• We further highlight the importance of network awareness in adapting a query execution to ensure low overhead.

• We have implemented a WASP prototype on Apache Flink, and experimentally demonstrate that WASP can achieve low-latency execution while preserving the quality of the results.

2 Background

2.1 Wide-area Stream Processing Systems

We consider a wide-area stream processing system spanning multiple geo-distributed edge clusters and data centers [20, 24, 49]. They are connected by WAN with diverse inbound and outbound bandwidth and latency. A global *Job Manager* running in one of the sites (typically in a data center) provides an interface for query submission, and it optimizes and deploys queries across multiple sites. The inputs of a query can be generated or collected at any site, e.g., system and user log updates from multiple CDN servers.

Figure 1 shows the execution pipeline of a typical data analytics query. Each query is parsed into a logical plan represented using a *directed acyclic graph* (DAG), where the *vertices* correspond to stream operators and the *edges* refer to data flows between operators. Each query's logical plan is optimized (e.g., pushing filter operators upstream to reduce data rates) and translated into multiple physical plan candidates. A query's physical plan consists of one or more *execution stages* (jobs), each of which can run in parallel as *execution instances* (tasks). The number of instances of each stage is typically predetermined by the *parallelism* value in the configuration. The system will deploy the tasks across multiple sites with WAN awareness to minimize query execution latency or WAN bandwidth consumption [25, 26, 47, 57, 58].

2.2 Wide-area Resource Constraints

Scarce and heterogeneous resources. Extracting real-time insights from large continuous data streams in wide-area settings is challenging due to the highly heterogeneous and scarce WAN bandwidth [23, 25, 30, 59]. Although recent work has argued that higher WAN bandwidth between data centers can be achieved by leveraging higher VM instances [36], this incurs higher monetary cost. Furthermore,

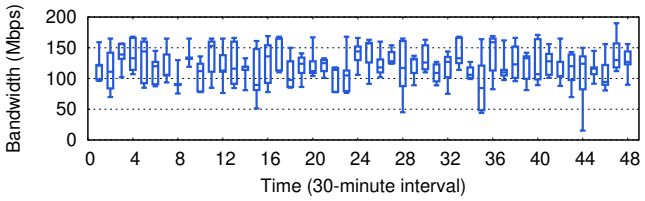


Figure 2. Bandwidth variability from Oregon→Ohio.

WAN bandwidth is highly dynamic in practice (will be discussed later). The emergence of *Edge Computing* comprising small edge clusters further introduces additional heterogeneity [24, 25]. They typically have limited computational resources and they are connected using the public Internet, whose bandwidth is even more constrained, with an average of <10Mbps, as reported by Akamai [1].

Resource and workload dynamics. Most of the work in adaptive stream processing systems has focused on a cluster environment where the main source of bottlenecks is due to the limited computational resources. They typically handle this problem by scaling out bottleneck operators *within a cluster* [14, 17, 19, 31, 56]. In wide-area settings, network bandwidth between sites imposes additional dynamic [49, 59, 62]. We conducted a one-day measurement of WAN bandwidth variation between 8 Amazon EC2 data centers (Oregon, Ohio, Ireland, Frankfurt, Seoul, Singapore, Mumbai, and Sao Paulo). We used `iperf` to measure the pair-wise bandwidth between sites every 5 minutes. Figure 2 shows the bandwidth variation between the Oregon and Ohio data centers. We can see that the bandwidth has a high variation (25% to 93% deviation from the mean). Others have also reported that the inter-data center network topology may change every 5-10 minutes [22, 27], supporting the dynamic nature of WAN bandwidth.

In addition to WAN dynamics, studies have also reported that many Internet applications have variable workload patterns, both temporally and spatially [18, 52]. For example, Twitter workload exhibits strong spatial and temporal variations, with day hours having 2× higher workload compared to night hours [37]. Thus, relying on a static deployment is a poor fit in such a highly dynamic environment. This may lead to performance degradation and wasteful resource utilization during high and low workload periods respectively.

3 WASP Overview

We first present an overview of WASP. We discuss WASP's adaptation techniques, design adaptation for wide-area settings, and how the system determines *which* technique to use in §4, §5 and §6 respectively.

3.1 System Architecture

Figure 3 shows the WASP system architecture. It consists of a *Job Manager* and multiple *Task Managers* geo-distributed

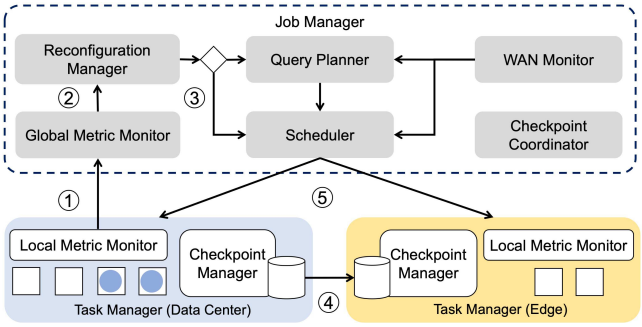


Figure 3. System overview of WASP.

across multiple edge clusters and data centers. The *Job Manager* consists of a *Query Planner* and a *Scheduler* that are responsible for planning query executions and deploying tasks respectively. We define the computational resources provided by each *Task Manager* using a *computing slot* abstraction, each of which can handle exactly one task. Each *Task Manager* continuously monitors and gathers its task's performance metrics (e.g., processing latency and input/output stream rates) through a *Local Metric Monitor* and reports them to the *Global Metric Monitor* ①. The *Global Metric Monitor* uses this information to diagnose any unhealthy execution or identify wasteful resource consumption ② and asks the *Reconfiguration Manager* to resolve it based on various factors (will be discussed in §6) ③.

3.2 Runtime Monitoring

Each operator keeps track of its runtime execution metrics such as processing rate (λ_P), output rate (λ_O), and the *selectivity* of the operator (σ) which is defined as the ratio between the output rate and the processing rate. These metrics are periodically reported to the *Global Metric Monitor* for diagnosis. The execution metric of an operator is computed based on the aggregate runtime information of all of its execution instances/tasks over the past time interval.

$$\lambda_P = \sum_{i=1}^{p} \lambda_P[i] \qquad \lambda_O = \sum_{i=1}^{p} \lambda_O[i] \qquad \sigma = \frac{\lambda_O}{\lambda_P}$$

WASP considers an execution to be *healthy*, i.e., unconstrained by the allocated resources, if no *backpressure* (will be discussed in §3.3) is observed and the following conditions hold:

1. The processing rate is equal to its input rate: $\lambda_P = \lambda_I$.
2. The input rate is approximately equal to the aggregated output rates of its upstream operators U: $\lambda_I \approx \sum_{u \in U} \lambda_O[u]$.

The first condition ensures that all tasks have sufficient computing resources to process their input streams, while the second condition ensures no network congestion in transmitting data streams from its upstream operators. However, these

conditions may not always hold due to the dynamic nature of the actual workload and WAN bandwidth.

If $\lambda_P < \lambda_I$, this indicates that the execution is constrained by the available computing resources. This may happen due to an increasing workload [37]. On the other hand, $\lambda_I < \sum_{u \in U} \lambda_O[u]$ may happen if the network bandwidth between an operator and its upstream operators is constrained or congested. This may happen due to an increasing workload and/or the reduction of network bandwidth availability caused by a change in the underlying network topology or bandwidth contention with other executions.

3.3 Estimating the Actual Workload

Modern distributed stream processing systems often rely on a *backpressure* to identify bottlenecks. When the resources are constrained, a bottleneck operator will trigger a control-rate message to its upstream operators to reduce the workload [13, 34]. In this case, the observed input and output rates of an operator do not reflect the actual workload, i.e., the actual stream rates from the *source* operators. Yet, to accurately determine the effective adaptation action that can resolve the bottleneck, the system should rely on the actual workload instead of the observed information [31]. Thus, we estimate the expected input and output rates of each operator based on the actual workload generated by the *source* operators ($\lambda_O[src]$), which is computed recursively as follows:

$$\hat{\lambda}_P = \hat{\lambda}_I = \begin{cases} \sum_{u \in U} \hat{\lambda}_O[u], & \text{if } U \neq \emptyset \\ \lambda_O[src], & \text{otherwise} \end{cases} \qquad \hat{\lambda}_O = \sigma \cdot \hat{\lambda}_I$$

4 Optimization-Based Adaptation

This section presents the 3 re-optimization adaptation techniques to handle wide-area dynamics: task re-assignment (§4.1), operator scaling (§4.2), and query re-planning (§4.3). We discuss how to apply these techniques in WAN-aware manner in §5 and present WASP's adaptation policies in §6.

4.1 Task Re-Assignment

Existing work has addressed the importance of WAN awareness in scheduling tasks in wide area settings to minimize query execution latency or WAN consumption [47, 57, 58]. However, they have mainly focused on optimizing the *initial* placement and do not consider re-evaluating it after the deployment. Yet, the initial placement may become obsolete when the environment/workload has changed significantly.

Resource-aware task placement. Most task placement algorithms rely only on the deployment of the predecessor (upstream) stages since they schedule *one-stage-at-a-time* in a topological order [25, 29, 47, 57]. However, re-assigning tasks of an already running stage based on solely the deployment of its upstream stages may result in a sub-optimal deployment because the deployment of its successor (downstream) stages rely heavily on its original placement. This may result in a cascading problem. Thus, our task re-assignment algorithm considers the deployments of *both* the upstream and downstream stages. Specifically, we re-compute the number of tasks to deploy in each site ($p[s]$) by solving the following Integer Linear Program (ILP):

Table 1. Descriptions of the used notations

Notation	Description
m	total number of sites
p	operator/stage parallelism
$p[s]$	number of tasks deployed at site s
$A[s]$	number of available slots at site s
ℓ_{s1}^{s2}	latency from site $s1$ to $s2$
B_{s1}^{s2}	available bandwidth from site $s1$ to $s2$
$\hat{\lambda}_I[s]$	expected input stream rate to site s
$\hat{\lambda}_O[s]$	expected output stream rate from site s
α	bandwidth utilization threshold

$$min \quad \sum_{s=1}^{m} p[s] \cdot (\ell_u^s + \ell_s^d), \qquad \forall u, \forall d \tag{1}$$

$$s.t. \quad \frac{p[s]}{p} \cdot \hat{\lambda}_I < \alpha B_u^s, \quad \forall s, \forall u, s \neq u \tag{2}$$

$$\frac{p[s]}{p} \cdot \hat{\lambda}_O < \alpha B_s^d, \quad \forall s, \forall d, s \neq d \tag{3}$$

$$0 \leq p[s] \leq A[s], \qquad \forall s \tag{4}$$

$$\sum_{s=1}^{m} p[s] = p, \qquad p \geq 1 \tag{5}$$

Table 1 summarizes the notations. Our goal is to minimize the network delay of transmitting data streams both from upstream (u) and to downstream (d) stages, which equivalently minimizes the delay incurred by a particular stage. Constraints 2 and 3 ensure sufficient inbound and outbound network bandwidth respectively. Constraint 4 ensures there are sufficient computing slots in each site, and Constraint 5 ensures that the system deploys all the tasks.

We include a maximum bandwidth utilization threshold, $\{\alpha \mid 0 < \alpha < 1\}$, in Constraints 2 and 3 for a few reasons. First, it provides a certain degree of stability in the solution by providing bandwidth *headroom* to handle slight workload and bandwidth variations. Secondly, the headroom makes the system robust to mis-estimation in measuring the actual inter-site bandwidth availability and data stream rates. Lastly, the reserved bandwidth can be used to process events that are queued during the transitioning process when an execution is adapted. Setting the α parameter too high (~ 1) leads to greater impact of misestimation and makes the system unstable, while setting it too low leads to a non-optimal

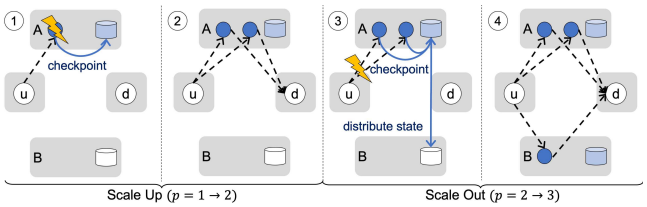


Figure 4. Scaling up/out operator within and across sites.

optimization. The automatic determination of the α parameter could probably benefit from the use of machine-learning techniques, an optimization that we leave for future work. Here, we set $\alpha = 0.8$.

If the system is able to find an alternative task placement, it may re-assign some of the existing tasks. Those that can be run at the original sites do not need to be migrated. For example, if the original placement is $S = \{s_1, s_2, s_3, s_4\}$ and the new placement is $S' = \{s_3, s_4, s_5, s_6\}$, only $(S - S') = \{s_1, s_2\}$ need to be migrated to $(S' - S) = \{s_5, s_6\}$. When re-assigning tasks, the system will (1) temporarily halt the execution, (2) instantiate new tasks at the new sites and terminate the old ones, and (3) resume the execution.

4.2 Operator Scaling

Although task re-assignment is generally applicable for any operator, the algorithm may not always be able to find a solution due to its constraint on the initial operator parallelism (Constraint 5). Yet, determining the *right* parallelism for every operator in advance may not be feasible given the highly dynamic environment. Thus, we consider (1) increasing the parallelism if an execution is constrained by the available resources, and (2) decreasing the parallelism to reduce any wasteful resource consumption.

Scale up/out. We define *scale up* and *scale out* in wide-area settings as instantiating new operator instances *within* a site and *across* sites respectively. In general, increasing parallelism can handle computational bottlenecks since it reduces the work performed by each individual task. However, scale up cannot resolve network bottlenecks while scale out can solve this by distributing the workload of any overloaded network link across multiple links.

Figure 4 shows how scale up and scale out can handle computational and network bottleneck respectively. When the system observes that a task's processing rate is less than its expected input rate ①, it may allocate additional slots and launch more instances. To prevent distributing large state over the WAN, the system will prioritize launching the new tasks within the same site ②. If the bandwidth from an upstream task u to Site-A is constrained ③, the system will instantiate a new task at Site-B and distribute the load across 2 links: $u \rightarrow A$ and $u \rightarrow B$ ④. This will reduce the load of the constrained network link but may require an inter-site state re-distribution in the case of stateful operator. We will discuss how to reduce such overhead later in §5. Although the example only shows inbound bandwidth contention, scale out can also handle outbound bandwidth contention.

When scaling up/out an operator, the system needs to determine the *scale up factor*, i.e., the increase in parallelism. We compute the scale factor based on the operator's execution model proposed in §3. Specifically, we compute the new parallelism of a bottleneck operator p' based on the ratio between the actual/expected input rate and the operator's processing rate. This is similar to the technique proposed by DS2 in handling computational bottleneck in a cluster-based stream processing system [31]:

$$p' = \left\lceil \frac{\hat{\lambda}_I}{\lambda_P} \cdot p \right\rceil$$

This equation gives the *minimum* parallelism value that can effectively resolve the bottleneck. Once the system has computed the new parallelism, it will determine the placement of the tasks by solving Equation 1. In the case of scale out, it is computed as the ratio between the stream rate that cannot be handled over the bandwidth availability. In general, increasing the parallelism can handle network bottleneck by distributing the data stream over multiple network links, i.e., $\frac{\lambda}{p'} < \frac{\lambda}{p}$ as $p' > p$, and hence, this will reduce the workload that needs to be transmitted to each individual network link.

Scale down. A system may over-allocate resources to a particular stage due to several reasons: misconfiguration, pessimistically reserving extra resources to handle peak workload, or as a result of scaling out/up an operator. This results in wasteful resource utilization. If the system identifies such a problem, it should scale down some of the underutilized tasks. We prioritize scaling down tasks that are not co-located with their upstream/downstream tasks to reduce the inter-site bandwidth consumption. However, the system needs to ensure the bandwidth to/from any of the sites is higher than the input/output rate after the scaling.

Determining the scale down factor has to be done carefully since it will increase the workload to the remaining tasks. In general, the scale down factor can be computed based on the ratio between the aggregated data stream rate and resource availability. However, aggressively scaling down an operator may result in a workload spike if the workload increases after the scale down. Yet, it is hard in practice to predict the future availability of the resources and the rate of the workload. Thus, we opt to gradually reduce the parallelism by 1 per iteration to prioritize performance stability over resource utilization. In every iteration, the system needs to ensure that any of the remaining tasks is not constrained, i.e., every task should have sufficient bandwidth and processing capacity to consume the additional workload (relayed data streams) from the terminated tasks. The system will observe its stability and may further scale down the operator in the subsequent iteration as needed.

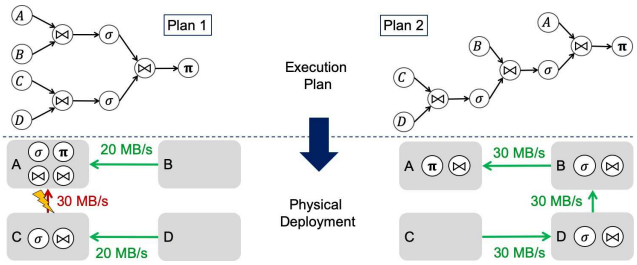


Figure 5. Different plans result in different executions.

4.3 Query Re-Planning

While task re-assignment and operator scaling focus on adapting the physical execution of a query, we further consider adapting its logical plan. Consider an example in Figure 5 which shows 2 different plans for the same query. It consumes input streams from 4 sources that are located at: A, B, C, and D, and joins them using a *full hash join*, which is commutative. A WAN-aware *Query Planner* may choose the first plan if the bandwidth is sufficient since it consumes less bandwidth (70MB/s for the first plan, and 90MB/s for the second plan). However, if the bandwidth between Site-C and Site-A is constrained, the *Query Planner* may opt for the second plan. This shows that the optimal plan depends on the runtime information *when* the query is deployed. Thus, the *Query Planner* may consider adapting the query plan when the environment has changed significantly [41].

To determine the optimal deployment of a query, the *Query Planner* and the *Scheduler* need to jointly optimize the query by evaluating different combinations of logical and physical plans. To avoid computing all possible combinations (that is NP-Hard), we rely on a heuristic cost-based estimation. It first applies any optimization that is independent of the environment similar to query optimization in the context of RDBMS [16, 50] (e.g., pushing filter operation upstream) and then evaluates multiple plans with different aggregation ordering. We only consider the ordering of aggregation operators since they are typically the ones that involve cross-site data transmission. The *Scheduler* will compute the optimal task placement for each plan and select the combined plan-placement pair with the lowest estimated delay.

Re-planning queries with stateful operators. The main challenge in re-planning a query is in preserving the processing state of a stateful operator. Changing the query plan of a stateless execution can be done by simply replacing the old execution with a new execution. However, in the case of stateful execution, the new execution must restore the state maintained by the previous execution. Although the *Query Planner* guarantees that alternative plans will output the same results, switching plans in the middle of an execution may not provide this guarantee because different plans may have different stateful operators with different state semantics. For example, the state of $\sigma(A \bowtie B)$ may not be compatible with $\sigma(B \bowtie C)$. Thus, the state of $\sigma(A \bowtie B)$ cannot be recovered by the operator instances of $\sigma(B \bowtie C)$.

To continue the progress from the old execution without losing any state, our *Query Planner* will only consider plans that comprise common sub-plans covering the stateful operators. For example, both Plan 1 and Plan 2 in Figure 5 exhibit a common sub-plan on $\sigma(C \bowtie D)$. Thus, the new instances of $\sigma(C \bowtie D)$ in the second plan can fully recover the states maintained by the previous plan. However, if $\sigma(A \bowtie B)$ is also stateful, changing from Plan 1 to Plan 2 may not be feasible unless the query can tolerate a certain degree of accuracy/quality loss. Another way to switch query plans for stateful execution is if the operator maintains a short and finite state where reconfiguration can be done at the end of the state interval. For example, in the case of a windowed-group aggregation with a tumbling window, this can be performed at the end of the window when the state is re-initialized. This is similar to the adaptation during the coordination interval in the *batch synchronous processing* (BSP) model [56, 61].

Re-evaluating both the logical and physical plans of a query typically results in a better adaptation than re-optimizing only its physical plan. However, the former is computationally expensive. Furthermore, query re-planning also has limited applicability for queries that comprise stateful operators. Thus, to preserve the accuracy of the results when re-planning a stateful execution, we only consider alternative plans that exhibit common sub-plans involving the stateful operators.

5 WAN-aware Design Adaptation

Local state management. Modern distributed stream processing systems support *stateful* computation, where each task tracks its processing progress as a *state* and periodically checkpoints it to a rendezvous storage system (e.g., HDFS [53]) [12, 14, 44]. This allows tasks to start/resume their executions from the last checkpointed state. Examples of state include intermediate aggregation results and partition offsets in Kafka [33]. Since tasks in wide-area streaming analytics are geo-distributed, their states are naturally generated in a geo-distributed fashion. To reduce the overhead of checkpointing large states over the WAN, WASP stores every state locally or to a nearby storage system. When a task is migrated to a different site, the *Checkpoint Coordinator* will first initiate a state migration and only after the state transfer completes, the task can resume its execution.

Network-aware state migration. Since WASP stores all computing states locally based on the deployment of the tasks, migrating a task to a different site requires migrating its state. Since the size of a state can be large in practice [11, 60], migrating a state over a low-bandwidth network

*Excluding the cross-site state migration overhead.
**Yes, if the state is not compatible or ignored by the new plan.

Table 2. Qualitative comparison between different adaptation techniques for streaming analytics queries.

Technique	Adaptation	Applicability	Granularity	Overhead*	Quality reduction
Task Re-Assignment	Task deployment	General	Stage	Low	No
Operator Scaling	Operator parallelism	General	Stage	Low	No
Query Re-Planning	Query execution plan	Query-specific	Query	High	No**
Data Degradation	Degradation policy	Query-specific	Policy-dependent	Low	Yes

link may incur a high state migration time, making it impractical for frequent dynamics. Thus, it is critical to reduce such overhead. As the major overhead of migrating a task is determined by the slowest state migration time, we determine the mapping from $(S - S')$ to $(S' - S)$ by solving a $minmax$ problem with the goal of minimizing the slowest state migration: $minmax(\frac{|state_{s_1}|}{B_{s_1}^{s_2}}), \forall s_1 \in (S - S'), \forall s_2 \in (S' - S)$. We show in Section 8.7 that a network-aware state migration can significantly reduce the overall adaptation overhead.

6 WASP's Adaptation Policy

In this section, we qualitatively compare different adaptation techniques (§6.1) and see how WASP determines *which* technique to use in the presence of dynamics (§6.2).

6.1 Adaptation Technique Comparison

Table 2 shows a qualitative comparison between different adaptation techniques. Both task re-assignment and operator scaling are generally applicable for any type of operators that can be parallelized whereas query re-planning has limited applicability for queries that comprise stateful operators since their states may not be compatible with the operators of a different plan. In contrast, data degradation is application- and algorithm-dependent and may not be applicable for queries that require high accuracy/quality.

The granularity of task re-assignment and operator scaling is on a stage level, but the latter is more flexible since it is not constrained by the initial operator parallelism. On the other hand, query re-planning typically results in a better adaptation since it re-optimizes the whole execution pipeline. However, it comes at the expense of high overhead since it needs to replace the entire execution. The granularity of data degradation depends heavily on the policies [49, 62]. For example, in video analytics, users can specify different frame rates (e.g., 30 and 60 FPS) and fidelity (e.g., 50% and 75%). Lastly, task re-assignment and operator scaling do not affect the output while degrading data may reduce the quality.

6.2 Determining Factors

Determining the *right* adaptation is complex and may not be feasible since it depends on a lot of factors. We propose a heuristic approach that considers (1) the type of bottlenecks, (2) the type of operators, (3) overhead, and (4) the type of dynamics. Figure 6 shows WASP's adaptability decision.

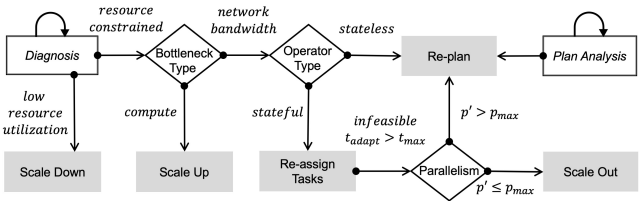


Figure 6. Determining *which* adaptability action.

Type of bottlenecks. To handle computational bottlenecks, WASP allocates additional resources and scales up the bottleneck operator. It will first try to scale the operator within the same site and only consider remote sites if the local resources are insufficient since the latter will incur additional network delay and WAN bandwidth consumption. On the other hand, if the execution is network-constrained, it further considers the type of the operator (stateless or stateful).

Type of operators. In the case of stateless execution, WASP will re-optimize the whole execution pipeline: both the logical and physical plans, since it does not require migrating any execution state. In the case of stateful execution, query re-planning may not always be feasible (as discussed in §4.3). Thus, WASP will first try to re-assign the existing tasks and only scale the operator if it cannot find an alternative placement with the given parallelism or the adaptation overhead is higher than a specific threshold ($t_{adapt} > t_{max}$). However, WASP may limit the number of additional tasks to scale per iteration to prevent resource hoarding or over-allocation, and may further choose to re-evaluate the query plan if the parallelism has exceeded the threshold ($p' > p_{max}$). In some cases, an operator may not be split without losing its semantic or requires modification to the query plan. For example, splitting a *counter* or *sink* operator requires an additional aggregator to combine the result. In this case, WASP will prevent scaling such an operator and simply re-plan the query.

Overhead. WASP estimates the adaptation overhead (t_{adapt}) based on the slowest state migration time ($t_{migrate}$) because the time required to migrate states over WAN typically dominates the overall adaptation overhead:

$$t_{adapt} = t_{migrate} = max(\frac{|state_{s_1}|}{B_{s_1}^{s_2}}), \forall s_1, \forall s_2$$

Here, s_1 and s_2 are the source and the destination sites respectively. If the overhead is too high, WASP will scale out the operator from p to p'. This can reduce the overhead

through state partitioning. Since most stream operators balance their workload among their tasks [31], the average state size per task after the scaling is $\frac{|state|}{p'} < \frac{|state|}{p}$, given that $p' > p$. Hence, $t_{migrate}$ can typically be reduced as the size of state partition is reduced. In a practical scenario, t_{max} can be set based on the frequency of the dynamics, so that $t_{max} <$ the frequency for the system to progress.

Type of dynamics. Our discussion so far has focused on addressing short-term dynamics. However, WASP can also be extended to handle long-term dynamics (e.g., daily workload shift [52]). This type of dynamics usually follows a specific pattern and can be predicted. Thus, WASP will handle this differently by periodically re-evaluating the query plan in the background. How to accurately model/profile the dynamics itself is out of the scope of this work.

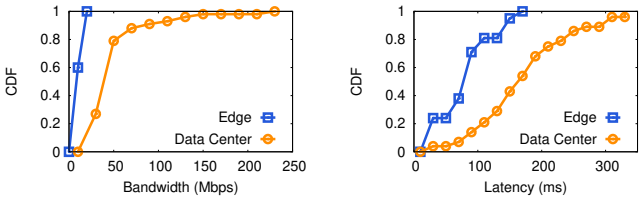
7 Discussion & Assumptions

Re-optimize or degrade? Data degradation has been widely used in wide-area streaming analytics and video analytics [24, 35, 49, 62, 63]. We believe that this approach is complementary to our work, and can be used in conjunction. For example, if maintaining accuracy is a priority, the system may prefer re-optimizing the execution to degradation. On the other hand, if the query can tolerate a certain degree of inaccuracy, the system may first degrade the accuracy and re-optimize the execution once it has reached a certain accuracy threshold. The main drawback of degradation with respect to operator scaling is that it cannot resolve misconfiguration and may result in significant quality loss when recovering from failures.

Transient workload spikes. We focus on workload dynamic that lasts longer than a few seconds. Re-optimizing a query execution to handle very short workload spikes may make the system unstable because the workload may have already changed when the query is adapted. Thus, WASP ignores transient workload fluctuations.

Homogeneous compute power across slots. WASP abstracts the computational resources in each location/site using *computing slots*. This is similar to the approach adopted by many distributed stream processing systems [13, 54, 61]. Since, the performance of most wide-area streaming analytics queries is predominantly determined by the inter-site data transmission, we hide the heterogeneity across *slots* and only consider the heterogeneity across sites based on the number of available slots per site.

Balanced event partitioning. A stream operator may partition its output to multiple downstream operators and tasks. For clarity reason, we assume the output stream is evenly distributed across tasks, which is common for stream operators [31]. However, our proposed techniques are not limited by this assumption. It is worth noting that we do not make any assumption on the input data distribution itself.



(a) Bandwidth distribution. (b) Latency distribution.

Figure 7. Inter-site network distribution. Edge connections only consider data centers within the same region.

8 Experimental Evaluation

Our experiments address the following questions:

- Is the re-optimization-based adaptation applicable for different types of queries (stateless and stateful) and dynamics (workload and network bandwidth) (§8.4)?
- How do task re-assignment, operator scaling, and query re-planning compare to each other (§8.5)?
- How does WASP perform in actual geo-distributed settings where workload variations, bandwidth changes, and failures may happen unpredictably and frequently (§8.6)?
- How does WASP mitigate the overhead of adapting stateful operators in wide-area settings (§8.7)?

8.1 System Implementation

We have implemented a WASP prototype on Apache Flink [13] by (1) implementing a network monitoring module (*WAN Monitor*) that periodically monitors the pair-wise available between sites in the background, (2) incorporating WAN awareness in planning queries and scheduling tasks, and (3) implementing an adaptability module that periodically gathers tasks' runtime information, diagnoses any unhealthy execution and wasteful resource utilization, and adapts them. We override the default task scheduler algorithm in Flink with our WAN-aware task placement algorithm that solves the ILP problem using the Gurobi optimization tool [2]. To reduce the overhead of evaluating query plans, the *Query Planner* only evaluates plans with different aggregation/join order as this type of operator typically involves data transafer over the WAN that may cause bottlenecks [57, 59].

8.2 Environment and System Setup

We evaluated WASP on a testbed derived from a real wide-area system deployment and created a driver program to introduce dynamics. We rely on a controlled environment for our evaluation to (1) clearly identify how each adaptation technique performs and (2) fairly contrast the techniques under a common condition. We later show how WASP performs under a live environment in (§8.6). Our testbed consisted of 16 nodes: 8 edge nodes (2−4 slots/node) and 8 data

Table 3. Location-based query details.

Application	State	Operators	Dataset
Advertising Campaign	<10 MB	filter, map window, join	YSB [15] synthetic data
Top-K Topics	∼100 MB	filter, map, union window, reduce	Twitter [6] trace (scaled)
Events of Interest	0 MB	filter, union project	Twitter trace (scaled)

center nodes (8 slots/node). A slot was configured with 1 CPU and 1GB of RAM. Figure 7 shows the network bandwidth and latency distributions between the nodes. The network between the data center nodes were configured based on a 1-day measurement of network bandwidth between 8 Amazon EC2 data centers: Oregon, Ohio, Ireland, Frankfurt, Seoul, Singapore, Mumbai, and Sao Paulo. On the other hand, the network between the edge nodes were configured based on the actual public Internet reported by Akamai [1]. We configure the system with $\alpha = 0.8$, $p_{max} = 3$, and a monitoring interval of 40 seconds to allow any adapted query to stabilize. These parameters were set based on the observation that WAN bandwidths are relatively stable within a period of 5 minutes [57].

8.3 Methodology

Query and Dataset. We evaluated WASP using the Yahoo! Streaming Benchmark (YSB) [15] and Twitter analytics queries. They cover (1) both stateful and stateless executions, (2) various combinations of commonly-used operators, and (3) real workloads with different characteristics (see Table 3 for details):

1. *Advertising Campaign* from the YSB monitors relevant advertisements related to specific campaigns every 10 seconds. To prevent bottlenecks in Redis and Kafka (e.g., partition mismatch), we replace all I/O operations with in-memory operations and cache intermediate results in memory.
2. *Top-K Popular Topic Detection* detects top events on a replayed geo-tagged Twitter trace. The query aggregates the top 10 most popular topics in each country over a period of 30 seconds. It consists of 2 different states: the source's offset and the intermediate aggregation results.
3. *Events of Interest* filters out tweets based on one or more attributes (e.g., language, topic, country of origin) and it does not maintain any internal state (stateless).

The YSB data were synthetically generated and distributed evenly across the 8 edge locations. On the other hand, the Twitter trace was distributed based on the actual geo-location information embedded in each tweet. Thus, the latter covers the spatial and temporal distributions of actual events. All operators were initially configured with $p = 1$ and we set a checkpointing interval of 30 seconds.

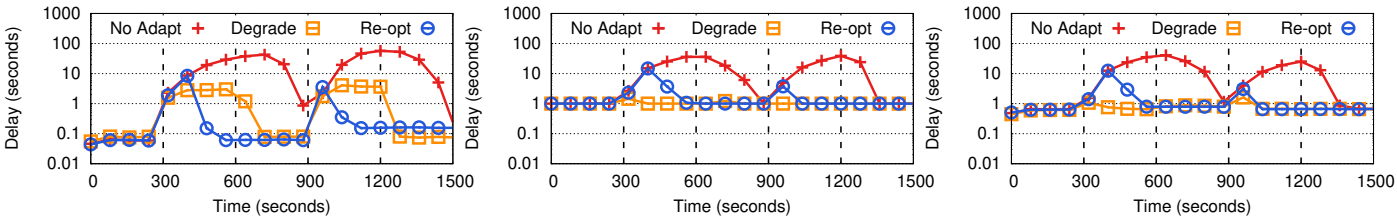
Metrics. We consider 2 main metrics in our experiments:

1. *Execution Delay.* The delay is measured as the average *event latency* which is the difference between the time an event is emitted at the sink and the time it was generated by the external source. In the case of windowed-group aggregation, the event generation time is set to the maximum event time of all events within a particular window (the latest event within a window) [32].
2. *Processing Ratio.* The processing ratio is computed as the ratio between the processing rate and the aggregated source rate over a time interval. A ratio of 1 indicates that the query is able to process all the events, while a ratio of < 1 indicates that the query is constrained by the allocated resources. This is more general than an accuracy metric [8, 55] since the latter is algorithm-specific [62].

8.4 Adapting to Wide-area Bottlenecks

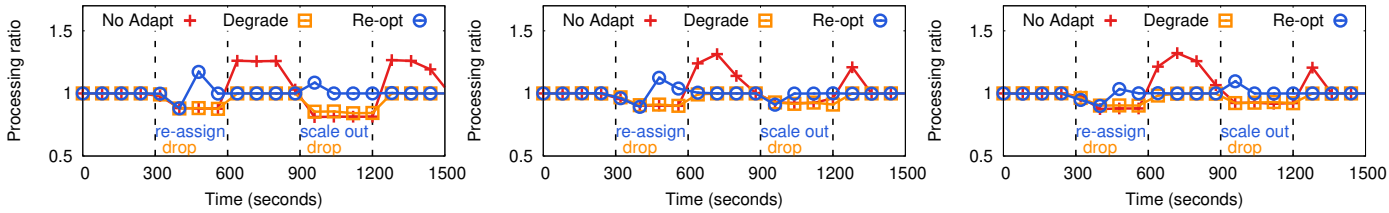
We initialized the input stream rate at each source to 10,000 events/second at $t = 0$ and introduced dynamics every 5 minutes. Specifically, we first increased the rate to 20,000 events/second at $t = 300$, and decreased it back to 10,000 events/second at $t = 600$. To see the effect of network bandwidth variation, we halved the bandwidth of every link at $t = 900$ and restore it at $t = 1200$. We compare our re-optimization-based approach, (1) `Re-opt`, against (2) `No Adapt` which did not adapt to dynamics, and (3) `Degrade` which dropped late events in case of insufficient resources. We set the SLO to 10 seconds for `Degrade`. Figure 8 and Figure 9 show the average delay and processing ratio of the all the 3 queries respectively.

$300 \leq t < 600$: We see from Figure 8 that the delay of `No Adapt` increased continuously by up to $2-3$ orders of magnitude as the workload increased because some network links could not sustain the workload. This shows in the reduction of the processing ratio from 1 to ∼0.86 (Figure 9). The processing ratio of `Degrade` also dropped to ∼0.86 but it was able to maintain the delay within the SLO by dropping any late events, which in practical scenario may affect the result's accuracy. In contrast, `Re-opt` was able to maintain low-latency processing without dropping any event (maintain the average processing ratio to ∼1) by *re-assigning* the bottleneck tasks to different locations at $t = 380$. The processing ratio of the YSB and Top-K momentarily dropped since the executions were suspended for approximately 2 and 10 seconds to migrate the states (Figure 9(a) and Figure 9(b)). Notice that the delay of `Degrade` increased to ∼8 seconds for the YSB case but it remained low for the Top-K case. This was because the key distribution of the former was much lower than the latter's, making it more sensitive to



(a) YSB Advertising Campaign. (b) Top-K Popular Topics. (c) Event of Interests.

Figure 8. Average execution delay under workload ($t = 300 \rightarrow 600$) and bandwidth ($t = 900 \rightarrow 1200$) dynamics.



(a) YSB Advertising Campaign. (b) Top-K Popular Topics. (c) Event of Interests.

Figure 9. Processing ratio under workload ($t = 300 \rightarrow 600$) and bandwidth ($t = 900 \rightarrow 1200$) variations.

late events when measuring the event time of the windowed-group aggregation.

$600 \leq t < 900$: When the workload reduced at $t = 600$, `Degrade` stopped dropping events and its processing ratio started to increase to 1. In the case of `No Adapt`, the processing ratio temporarily increased > 1 indicating that the system was consuming queued events that had been accumulating from the previous interval.

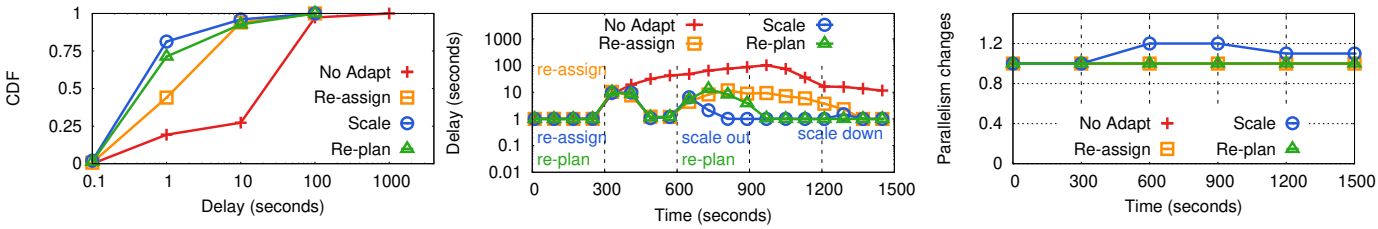
$900 \leq t \leq 1200$: To see the effect of network bandwidth variation, we halved the bandwidth capacity of all links at $t = 900$. We can see that the delay and processing ratio have similar trends to the effect of increasing workload. However, `Re-opt` took a different adaptation action by *scaling out* the bottleneck operator instead of *re-assigning* the tasks since the adaptation module could not find a single alternative link whose bandwidth was higher than the stream rate. We can also see that operator scaling resulted in a faster convergence, taking advantage of having more resources. Finally, the delay of the 3 queries dropped at $t = 1200$ when the bandwidth increased.

These results show that (1) the re-optimization-based adaptation can handle both workload and bandwidth variations, (2) this is generally applicable for both stateless and stateful executions, and (3) it can maintain low-latency execution without dropping any event. For the rests of the experiments, we used the stateful Top-K query as our workload since it is the best representation of an actual geo-distributed workload among the 3 queries, although they have a similar trend (Figure 8 and Figure 9).

8.5 Re-Assign vs. Scale vs. Re-Plan

Next, we compared task re-assignment, operator scaling, and query re-planning, in handling a combination of workload and bandwidth variations independently. We introduced dynamics every 5 minutes by varying the workload and bandwidth by a factor of $\{1, 2, 2, 1, 1\}$ and $\{1, 1, 0.5, 0.5, 1\}$ respectively. We compared (1) `No Adapt`: which did not adapt to dynamics, (2) `Re-assign`: which only handled dynamics by re-assigning tasks, (3) `Scale`: which would first try to re-assign the tasks but might scale some operators if it could not find a solution, and (4) `Re-plan`: which re-evaluated the execution plan based on the observed workload and resource availability. Both `Re-assign` and `Re-plan` never changed the parallelism.

First, we can see from Figure 10(a) that all of the techniques that adapt the query resulted in lower delay compared to `No Adapt`, highlighting the importance of adaptability in handling dynamics. Secondly, `Scale` resulted in the lowest overall delay compared to `Re-plan` and `Re-assign`, and `Re-plan` resulted in a lower delay for the majority of the events ($<$ 93rd percentile) with respect to `Re-assign`. Figure 10(b) breaks down the delay of each technique for each interval. At $t = 300$, both `Re-assign` and `Scale` *re-assigned* the tasks while `Re-plan` switched to another plan. All of them could handle the the workload increase during this interval. However, when the available bandwidth decreased at $t = 600$, `Re-assign` was unable to find an alternative task placement since it was constrained by the initial parallelism. In contrast, `Scale` was able to handle this problem by acquiring 20% additional slots (Figure 10(c)) and *scaling out* the bottleneck operators. `Re-plan` was also able to handle the problem by *re-planning* the query. However, `Scale` resolved the bottleneck faster than `Re-plan`, taking advantage of the additional resources. Lastly, `Scale` decreased the parallelism when the bandwidth availability



(a) Delay distribution. (b) Average delay over time. (c) Parallelism changes over time.

Figure 10. Breakdown comparison between different techniques in handling dynamics individually.

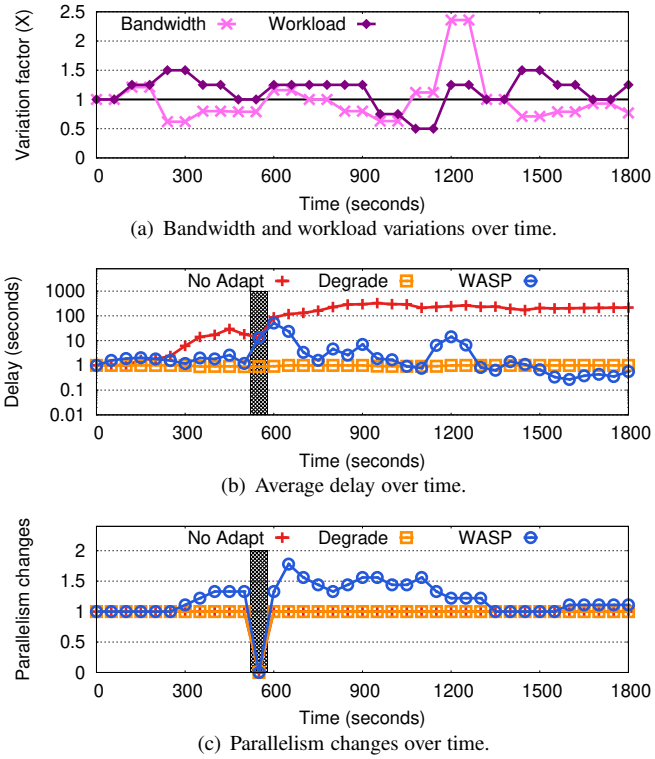
had increased at $t = 1200$ and some of the resources became underutilized.

Comparing `Re-assign` and `Scale`, we can see that dynamically adapting operator parallelism can better handle bottlenecks with the expense of consuming more resources. We can also see that re-optimizing both the logical and physical executions (`Re-plan`) results in a more optimal adaptation than simply re-assigning tasks (`Re-assign`) with the same parallelism, and hence the former is preferable whenever possible.

8.6 WASP in a Live Environment

In this experiment, we evaluated WASP using the Top-K popular topic detection query in a live, trace-driven environment where we introduced (1) network bandwidth dynamics based on a real pair-wise bandwidth variation trace between 8 Amazon EC2 data centers which ranged from 0.51 to 2.36, and (2) random workload patterns for each source with a variation factor ranging from 0.8 to 2.4, and (3) a failure at $t = 540$ by revoking all the computational resources and re-allocating them after 60 seconds. We added failure in this experiment to see how WASP can *scale* a query to quickly process accumulating events. Figure 11(a) shows the bandwidth and workload variations in our experiment. Here, we compared (1) `WASP` against (2) `No Adapt` and (3) `Degrade`. `WASP` could use any of the adaptation techniques: task re-assignment, operator scaling, and query re-planning, depending on the condition discussed in §6.2.

Figure 11(b) and Figure 11(c) show the delay and parallelism over time. We make a few observations here. First, `WASP`'s processing delay stayed close to 1 second (similar to the unconstrained case) for most of the time except for some intervals: At $300 \leq t < 540$, there was a variation in the delay when `WASP` *scaled out* 2 of the tasks to handle workload increases and bandwidth drops. At $t = 640$, `WASP` was able to quickly handle the accumulated events after recovering from failure by *scaling out* the bottleneck operators. It then gradually *scaled down* the operators after the execution stabilized. Finally, it further *scaled down* the majority of the additional tasks at $900 \leq t < 1320$ when the available bandwidth had increased.

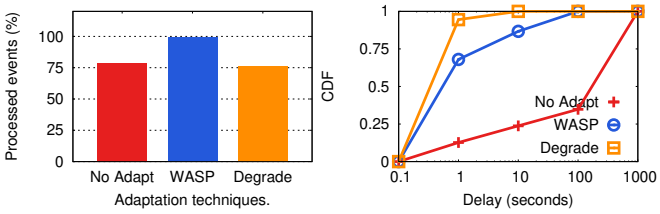


(a) Bandwidth and workload variations over time.
(b) Average delay over time.
(c) Parallelism changes over time.

Figure 11. WASP's adaptations to dynamics and failures.

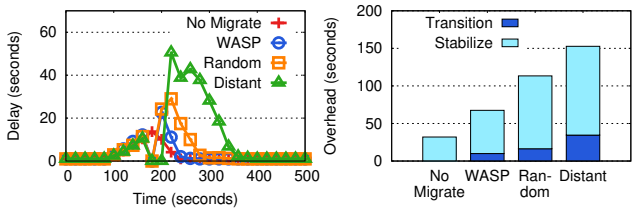
In contrast to `WASP`, the delay of `No Adapt` increased by more than 2 orders of magnitude, especially after the execution recovered from failure since it was unable to handle the queued events. Although `Degrade` could maintain the average delay within 1 second, it had to sacrifice up to ∼24% of the events. This resulted in an inaccurate result and hence, may not be tolerable for queries that require high accuracy. In contrast, `WASP` could process all of the events while maintaining the low-latency processing (Figure 12(a)). From Figure 12(b) we can see that `WASP` had a longer delay tail distribution compared to `Degrade`. We observed that the majority of the delay came from the monitoring process, the transitioning phase for migrating states, and the processing of queued events after `WASP` recovered from failure.

These results show that (1) WASP can handle real-world dynamics and failures without dropping any of the events,



(a) Average processed events. (b) Delay distribution.

Figure 12. Quality vs. delay trade-offs.



(a) Execution delay over time. (b) Adaptation Overhead.

Figure 13. Network-aware state migration.

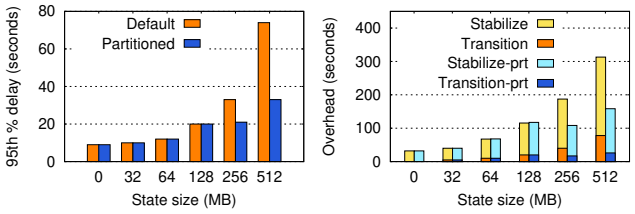
and (2) there is essentially a trade-off between the re-optimization and degradation-based adaptation techniques in maintaining the quality/accuracy of the results and maintaining the low-latency processing.

8.7 Mitigating Adaptation Overhead

In the last set of experiments, we see how WASP can reduce the overhead of adapting queries with large computation state. Specifically, we highlight the importance of network awareness and the benefit of state partitioning in reducing the overhead. We break down the overhead into 2 phases: (1) *transition time*: when the execution is suspended for state migration, and (2) *stabilizing time*: the time needed to consume all queued events that have been accumulating during the transition process. We highlight the importance of network-aware state migration in §8.7.1 and show the benefit of state partitioning to further reduce the overhead in §8.7.2. In both experiments, we controlled the size of the state that needed to be migrated.

8.7.1 Network-aware State Migration

To ensure low-overhead adaptation, WASP estimates the transition time of migrating a task to a different site based on the size of its state and the bandwidth availability between the initial site and the new site. The problem with the network-agnostic approach is that migrating a state over the WAN may take a long time if the bandwidth between the two sites is low. This is impractical for frequent dynamics that are common in wide-area settings.



(a) Execution delay distribution. (b) Adaptation overhead.

Figure 14. Mitigating overhead through operator scaling and state partitioning. "prt" stands for `Partitioned`.

In this experiment, we compared (1) `WASP` against (2) `No Migrate`[†] which did not migrate the state (equivalent to adapting stateless operators), (3) `Random`: which ignored the bandwidth availability, and (4) `Distant`: which migrated a state to a site. In any case, the system ensured that the destination site had sufficient bandwidth to process the actual data stream and hence, the execution would eventually stabilize. We fixed the state size to 60MB.

Figure 13(a) compares the effect of different state migration techniques to the overall query execution delay. In any of the cases, the system started adapting the query at $t = 180$. Here, we can see that `No Migrate` could quickly reduce the delay without migrating the state. However, this resulted in an incorrect result or a loss in accuracy. Comparing the other 3 techniques that maintained the state, we can see that `WASP` resulted in the lowest delay during the adaptation phase.

Figure 13(b) shows the breakdown of the overhead. First, `No Migrate` incurred $\sim$0 transition time since it only redirected the data streams. There was a stabilizing time and a slight increase in delay despite not migrating the state (similar to adapting stateless operator) due to the queued events during diagnosis period. Reducing this period may reduce the transition time but makes the system more susceptible to spikes and miss-estimation. Secondly, `WASP` resulted in 41$-$56% lower overhead and 7$-$20 seconds lower 99th percentile delay compared to `Random` and `Distant` respectively. The reason is because the two WAN-agnostic approaches migrated state over a low-bandwidth link, leading to a higher transitioning time and stabilizing time. These results show the importance of network awareness in reducing the adaptation overhead in wide-area settings.

8.7.2 State Partitioning

In addition to network awareness, we observe how partitioning large states across multiple links can further reduce the adaptation overhead. In this experiment, we compared `Default`: which never partitioned the state, and `Partitioned`: which would force the adaptation module to find an alternative placement (may involve operator scaling) and partition the state whenever the estimated transition time

[†]Ignoring the state will result in a loss of accuracy in the result.

exceeded a specific threshold. We varied the state size to {0, 32, 64, 128, 256, 512} MB and set the maximum threshold to 30 seconds.

Figure 14(a) shows the 95th percentile delay over different state size. We can see that the delay of `Default` increased as the state size increased. In contrast, `Partitioned` could reduce the delay in the case of large state (256MB and 512MB). This is due to the reduction in the adaptation overhead. Figure 14(b) shows the breakdown of the overhead. In general, the overhead of adapting a query increased as the state size increased. However, `Partitioned` was able to reduce this overhead by scaling out the bottleneck operator and partitioning its state across multiple network links. This reduced the overhead by more than 120 seconds (Figure 14(b)) which subsequently reduced the average delay by 42 seconds (Figure 14(a)). These results highlight another benefit of operator scaling in reducing the adaptation overhead.

9 Related Work

Wide-Area Data Analytics Systems. Wide-area data analytics systems can be classified into two different groups based on their processing model: (1) batch analytics, and (2) streaming analytics. Early work in wide-area data analytics has focused on incorporating WAN awareness in scheduling jobs and placing tasks across multiple sites with the goal of minimizing query execution latency [25, 47, 57] or saving WAN bandwidth consumption [58]. However, they ignore the dynamic nature of wide-area environment. Tetrium [25] considers dynamic resource availability but does not consider re-optimizing jobs that have already been deployed. Turbo [59] has looked at dynamic query re-planning for batch analytics but its techniques cannot be applied directly for long-running streaming analytics queries.

Existing work has also looked at optimizing streaming analytics queries in wide-area settings [21, 29, 46, 49, 62]. Pietzuch et al. [46] address the problem of network-aware operator placement, and Sana [29] considers sharing common execution between queries. However, they do not address workload/resource dynamics. Others have considered the dynamic nature of a wide-area environment, but focused on trading latency, WAN traffic, and accuracy. Heintz et al. [21] propose a technique to trade accuracy and delay in the context of windowed grouped aggregation. Kumar et al. [35] proposes a TTL-based approach that trades delay and WAN traffic for windowed grouped aggregation. JetStream [49] allows users to specify different degradation policies with a data-cube model. AWStream [62] relies on a profiling technique to determine which degradation policy to take to ensure a certain degree of accuracy. We argue that these degradation approaches are application-specific and they are complementary to our techniques.

Adaptability in stream processing systems. There have been a large body of work that address the importance of adaptability in cluster-based stream processing systems [11, 14, 17, 19, 31, 38, 42, 48, 51, 56, 60]. However, they have focused on addressing computational bottlenecks by scaling out tasks across multiple computing nodes within a cluster. These techniques cannot be directly applied to handle dynamics in a wide-area environment due to the highly heterogeneous and limited network bandwidth.

Others have also looked at the importance of minimizing the adaptability overhead. Drizzle [56] reduces the synchronization overhead for *Bulk Synchronous Processing* model. Chi [42] relies on control mechanism to reduce the overhead of global synchronization. ChronoStream [60] partitions and distributes large states across multiple nodes to allow fast recovery. DS2 [31] predicts the scaling factor based on the expected processing rate of each operator for dataflow model. Although these techniques are related to our work, they do not account for network constraints. In wide-area environment, the overhead of migrating large states over WAN is significantly higher than the partitioning overhead, and hence our techniques focus on minimizing this overhead.

10 Conclusion

In this work, we rethink the adaptability property of wide-area stream processing systems and propose a WAN-aware adaptation framework, WASP, that allows queries to handle dynamics without compromising quality. WASP adapts queries through a combination of multiple techniques: task re-assignment, operator scaling, and query re-planning. WASP can automatically determine which adaptation to take based on the type of queries, dynamics, and goals. WASP further incorporates network awareness to mitigate the overhead of adapting queries in wide-area settings. Experimental evaluation shows that WASP is able to handle wide-area dynamics with low overhead while maintaining the quality of the results.

Acknowledgments

The authors would like to thank the anonymous Middleware reviewers for their valuable comments and feedback. The work is supported by grant NSF CNS-1619254 and CNS-1717834.

References

[1] Akamai's state of the Internet report. https://bit.ly/2N6fL8r.
[2] Gurobi optimization. http://www.gurobi.com/. Accessed: 2018-11-27.
[3] Keystone real-time stream processing platform. https://bit.ly/2pxsMyI. Accessed: 2019-11-02.
[4] One nation under CCTV: The future of automated surveillance. https://bit.ly/2N7vfZE. Accessed: 2019-11-02.
[5] One surveillance camera for every 11 people in Britain. https://bit.ly/2Jlut3v. Accessed: 2019-11-02.
[6] Twitter streaming API. https://developer.twitter.com/. Accessed: 2019-06-04.

[7] Video access log processing with Apache Flink. https://bit.ly/2oIe8o6. Accessed: 2019-11-02.

[8] H. Abdi. The Kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[9] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[10] G. Ananthanarayanan, S. Kandula, A. G. 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)*, volume 10, page 24, 2010.

[11] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 577–588. ACM, 2013.

[12] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[14] 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*, pages 725–736. ACM, 2013.

[15] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[16] R. Elmasri. *Fundamentals of database systems*. Pearson Education India, 2008.

[17] 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.

[18] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 171–180. IEEE, 2007.

[19] 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.

[20] 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. ACM, 2015.

[21] 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. ACM, 2016.

[22] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.

[23] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*, pages 629–647, 2017.

[24] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2018.

[25] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, page 12. ACM, 2018.

[26] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 111–124. ACM, 2015.

[27] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[28] J. Jiang, S. Sun, V. Sekar, and H. Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *NSDI*, volume 1, page 3, 2017.

[29] A. Jonathan, A. Chandra, and J. Weissman. Multi-query optimization in wide-area streaming analytics. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–425. ACM, 2018.

[30] A. Jonathan, A. Chandra, and J. Weissman. Rethinking adaptability in wide-area stream processing systems. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, 2018.

[31] 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. USENIX Association, 2018.

[32] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518. IEEE, 2018.

[33] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[34] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[35] D. Kumar, J. Li, A. Chandra, and R. Sitaraman. A TTL-based approach for data aggregation in geo-distributed streaming analytics. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):29, 2019.

[36] F. Lai, M. Chowdhury, and H. Madhyastha. To relay or not to relay for inter-cloud transfers? In *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[37] K. Leetaru, S. Wang, G. Cao, A. Padmanabhan, and E. Shook. Mapping the global Twitter heartbeat: The geography of Twitter. *First Monday*, 18(5), 2013.

[38] W. Lin, H. Fan, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou. StreamScope: Continuous reliable distributed processing of big data streams. In *NSDI*, volume 16, pages 439–453, 2016.

[39] H. H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang. Efficiently delivering online services over integrated infrastructure. In *NSDI*, volume 1, page 1, 2016.

[40] F. Loewenherz, V. Bahl, and Y. Wang. Video analytics towards vision zero. *Institute of Transportation Engineers. ITE Journal*, 87(3):25, 2017.

[41] K. Mahajan, M. Chowdhury, A. Akella, and S. Chawla. Dynamic query re-planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 253–267. USENIX Association, 2018.

[42] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, et al. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, 2018.

[43] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang. Practical, real-time centralized control for CDN-based live video delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 311–324. ACM, 2015.

[44] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[45] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI. Making sense of performance in data analytics frameworks. In *NSDI*, volume 15, pages 293–307, 2015.

[46] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 49–49. IEEE, 2006.

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

[48] 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. ACM, 2013.

[49] 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 *NSDI*, volume 14, pages 275–288, 2014.

[50] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, 2000.

[51] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu. Elastic scaling of data parallel operators in stream processing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[52] Z. Shen, Q. Jia, G.-E. Sela, B. Rainero, W. Song, R. van Renesse, and H. Weatherspoon. Follow the sun through the clouds: Application migration for geographically shifting workloads. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 141–154. ACM, 2016.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. Ieee, 2010.

[54] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156. ACM, 2014.

[55] C. J. Van Rijsbergen. *Information Retrieval (2nd ed.)*. Butterworth-Heinemann, 1979.

[56] 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*, pages 374–389. ACM, 2017.

[57] 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)*, volume 16, pages 435–450, 2016.

[58] 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 *NSDI*, volume 7, pages 7–8, 2015.

[59] 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. ACM, 2018.

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

[61] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[62] 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. ACM, 2018.

[63] 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*, volume 9, page 1, 2017.