Multi-Query Optimization in Wide-Area Streaming Analytics

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

Wide-area data analytics has gained much attention in recent years due to the increasing need for analyzing data that are geographically distributed. Many of such queries often require real-time analysis on data streams that are continuously being generated across multiple locations. Yet, analyzing these geo-distributed data streams in a timely manner is very challenging due to the highly heterogeneous and limited bandwidth availability of the wide-area network (WAN). This paper examines the opportunity of applying multi-query optimization in the context of wide-area streaming analytics, with the goal of utilizing WAN bandwidth efficiently while achieving high throughput and low latency execution. Our approach is based on the insight that many streaming analytics queries often exhibit common executions, whether in consuming a common set of input data or performing the same data processing. In this work, we study different types of sharing opportunities and propose a practical online algorithm that allows streaming analytics queries to share their common executions incrementally. We further address the importance of WAN awareness in applying multi-query optimization. Without WAN awareness, sharing executions in a wide-area environment may lead to performance degradation. We have implemented our WAN-aware multi-query optimization in a prototype implementation based on Apache Flink. Experimental evaluation using Twitter traces on a real wide-area system deployment across geo-distributed EC2 data centers shows that our technique is able to achieve 21% higher throughput while saving WAN bandwidth consumption by 33% compared to a WAN-aware, sharing-agnostic system.

CCS CONCEPTS

• **Networks** → **Wide area networks**; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

Geo-distributed systems, stream processing systems, multi-query optimization, execution sharing

ACM Reference Format:
Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Multi-Query Optimization in Wide-Area Streaming Analytics. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 14 pages.
https://doi.org/10.1145/3267809.3267842

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267842

1 INTRODUCTION

Recent years have seen a growing interest in geo-distributed data analytics, where analysts need to extract meaningful information from large amounts of data that are distributed across multiple locations. Examples of these data include not only traditional log updates from content distribution networks (CDN) but also user-generated microblogs, sensor data from distributed IoT devices, and video streams from distributed surveillance and traffic control cameras. Such data are naturally produced in a geo-distributed manner *near the edge*. The main challenge in analyzing these data is in extracting information in a timely manner [25, 53].

The interest in real-time analysis over continuous data streams has resulted in the recent development of various scalable stream processing systems [5, 13, 40, 59, 66]. However, these systems have been designed primarily for a *centralized*, *tightly-connected* cluster environment where compute nodes are inter-connected with high-speed network. Using these centralized systems for analyzing geo-distributed data streams is impractical since it requires transmitting large amounts of data continuously over the wide-area network (WAN) that has limited bandwidth, slow, and expensive. This *centralization* approach typically leads to wasteful WAN bandwidth utilization and is often unable to satisfy the timeliness requirements of most data analytics applications, as has been shown by recent work in geo-distributed data analytics [29, 53, 61, 62].

Most of the work in geo-distributed data analytics has instead focused on batch-oriented processing, where finite input data sets are available prior to a query execution [29–31, 53, 61, 62]. In this case, the main challenge is to schedule each query that minimizes either the overall execution time or WAN bandwidth consumption. Others have also looked at the problem of geo-distributed data analytics in the context of stream-oriented processing where long-running queries are deployed to extract information from continuous data streams [26, 32, 52, 54]. However, most of them focused on optimizing an individual query execution. In contrast, we consider optimizing multiple queries by applying multi-query optimization in a WAN-aware manner.

In practice, the multitenancy nature of a Cloud environment leads to multiple queries running concurrently and competing for limited, shared resources. Recent work has indicated that it is common in a production environment for multiple queries to exhibit common executions, whether in reading the same set of inputs or performing the same data processing, especially for queries from the same application domain or those that rely on popular data [16, 20, 28, 42, 43, 49, 56, 63]. Furthermore, as more and more data are increasingly geo-dependent and made available to the public, it is increasingly likely that more geo-distributed data analytics queries will share common executions. As a concrete example, Twitter data streams are commonly analyzed for different purposes including sentiment analysis [3], finding relevant audiences for an advertisement/campaign [2], and detecting trending topics in a certain area

or globally [37, 41]. Another example includes CDN logs that are continuously monitored for different goals such as high quality service delivery, network monitoring, and user behavior analysis.

Based on this insight, we examine the opportunity of applying multi-query optimization in the context of wide-area streaming analytics. Our goal is to efficiently and effectively utilize limited WAN bandwidth while providing low latency and high throughput execution of multiple concurrent queries. We first study different types of cross-query sharing opportunities: (1) *input-sharing*: where multiple queries share a common subset of input data, (2) *operator-sharing*: where multiple queries perform the same data processing on the same inputs, and (3) *output-sharing*: where multiple queries additionally share partial output (or intermediate) results. Furthermore, we demonstrate the importance of *WAN awareness* in applying multi-query optimization in a wide-area environment: both for query planning and for operator scheduling.

There are a few challenges in applying multi-query optimization (MQO) in the context of wide-area streaming analytics. First, multiple queries may be submitted to the system independently at different times by different users and hence, it may not be possible to optimize these queries together prior to their deployment using the MQO techniques proposed for batch-oriented workloads [49, 50, 63]. Second, most streaming analytics queries are long-running and latency sensitive [15, 40, 60]. Thus, it is very inefficient and impractical to interrupt existing query executions whenever a new query arrives to optimize them together. Instead, our techniques optimize multiple query executions in an *online* manner by allowing queries to share their common executions *incrementally* without disrupting existing executions. The wide-area settings further impose unique challenges in applying multi-query optimization due to the highly heterogeneous and limited bandwidth availability of the wide-area network. We show that applying MQO designed for a local environment in a wide-area environment without network awareness is sub-optimal and may lead to performance degradation due to the assumptions of homogeneous and high-bandwidth network that are invalid in real wide-area deployment.

We have implemented our WAN-aware multi-query optimization into a system prototype called `Sana`: an Apache Flink [13]-based stream processing system that we have adapted for wide-area deployments. We quantitatively evaluate `Sana` using 14 geo-distributed EC2 [1] data centers. Our experimental evaluation using multiple streaming analytics queries [1, 17] on a Twitter trace shows that `Sana` is able to achieve 21% higher throughput while saving WAN bandwidth consumption by 33% compared to the state-of-the-art WAN-aware, sharing-agnostic system.

We summarize our contributions as follow:

- We propose a *multi-query optimization* approach that allows multiple streaming analytics queries to *incrementally* share their common executions in an *online* manner in wide-area settings (§4).
- We address the importance of *WAN awareness* in applying multi-query optimization in a wide-area environment, both in planning and scheduling multiple query executions (§5).
- We have implemented our WAN-aware multi-query optimization techniques in a system prototype based on Apache Flink (§6).
- We experimentally demonstrate the effectiveness of our WAN-aware multi-query optimization through a real system deployment across geo-distributed EC2 data centers using Twitter trace-driven queries (§7).

[1] https://aws.amazon.com/ec2/

2 BACKGROUND AND MOTIVATION

In this section, we discuss the background of wide-area streaming analytics and illustrate through an example the benefits of applying multi-query optimization to this context.

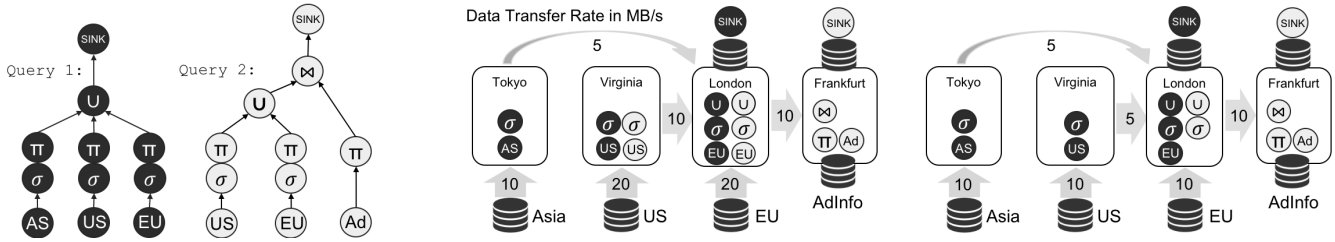
2.1 Wide-Area Streaming Analytics

Stream Processing Model. Stream processing systems can be generally classified into two different classes based on their computational model: (1) the *dataflow* model [6, 13, 44, 47], and (2) the *bulk-synchronous parallel* (BSP) model [14, 33, 66]. Here, we focus on the dataflow model where data streams flow continuously from one or more data sources into the system and are transformed by a set of *stream operators*. We consider this model over the BSP model for two reasons. First, it allows data streams to be processed with lower latency and higher throughput [17, 38]. Second, the BSP model incurs higher communication overhead due to the frequent synchronization at every micro-batch boundary [60], which will be inefficient in a wide-area environment. However, our proposed techniques are not limited to the dataflow processing model, and can be adapted to the BSP model.

Stream Query Model. A streaming analytics query is typically written using a high-level, SQL-like language [8, 58]. The query is (1) translated and optimized by a *query optimizer* into its corresponding execution plan, represented using a directed acyclic graph (DAG), and (2) deployed by a *job scheduler*. A query execution graph, denoted as $g = (V, E)$, consists of vertices V and edges E. Each vertex $v \in V$ corresponds to a stream operator f_v that consumes input streams I from its predecessor (upstream) vertices and produces output streams O to its successor (downstream) vertices ($O = f_v(I)$). Each edge $e \in E$ represents a data flow between two vertices. Example of stream operators include *source*, *map*, *reduce*, *join*, *filter*, *sink*, etc. The *source* and the *sink* operators are specialized operators that receive input streams from external sources and output the results to final destinations respectively.

Geo-Distributed Stream Processing. We consider a stream processing system consisting of multiple *compute nodes* that are geographically distributed across multiple sites, and a *master node* located in one of the sites. A streaming analytics query is submitted to the master node running a *query optimizer* and a *job scheduler*. The query optimizer will optimize the execution plan of the query (e.g., parallelize and chain multiple operators) and the scheduler will deploy each parallel execution instance (task) on a compute node.

The inputs of wide-area streaming analytics queries are produced by multiple sources that are geo-distributed, and they are continuously ingested into nearby edge clusters or data centers. Examples of such data streams include sensor readings, microblogs from social network applications, and log updates from distributed CDN servers. Each query continuously reads these geo-distributed input streams, processes them, and outputs its results to one or more final locations, e.g., stored in databases, displayed on a monitoring dashboard, or streamed back as new inputs for iterative analysis.



(a) Logical plan of `Query 1` and `Query 2` (b) Independent deployment of `Query 1` and `Query 2` (c) Shared deployment of `Query 1` and `Query 2`

Figure 1: Example: Execution sharing between two streaming analytics queries in a wide-area environment.

To minimize data transfer overhead between operators, the scheduler will try to deploy connecting operators on the same site. However, common operators such as *union*, *shuffle*, and *join* may require data to be transmitted across sites since their inputs may be generated at different locations. Thus, the query optimizer and the job scheduler should be aware of the underlying WAN topology to generate an optimized execution plan and a deployment decision that can effectively utilize WAN bandwidth respectively [30, 53, 61].

2.2 Benefits of Multi-Query Optimization in A Wide-Area Environment

Multi-Query Optimization in Data Analytics World. Multi-query optimization (MQO) is a well-studied topic in the database community to improve the performance of multiple query executions, especially in relational databases [9, 16, 21, 22, 24, 55, 67]. Since many data analytics queries often rely on common popular data sets and may perform common executions, recent work has argued that it is imperative to apply MQO in the context of data analytics to improve the performance of multiple data analytics queries [16, 49, 50, 57, 63]. Here, the query optimizer needs to identify the commonality between queries and potentially combine their executions to mitigate redundant executions. The combined execution must produce the same outputs as those produced by executing the queries independently.

In this paper, we argue that applying multi-query optimization in a wide-area environment can reduce WAN bandwidth consumption by eliminating the redundancy in transmitting duplicate data over the WAN. In the face of bandwidth constraints, this can improve the overall performance of concurrent query executions. Although there have been attempts that look at the opportunity of optimizing multiple queries in the context data analytics, their focus have been largely on batch-oriented workloads [49, 50, 63]. These approaches are not applicable for stream-oriented workloads because most streaming analytics queries are *deployed once and run indefinitely* [13, 44]. Thus, applying MQO in streaming analytics should be done in an *online* manner as new queries arrive by sharing any common execution *incrementally*.

Previous attempts have also looked at the opportunity of applying multi-query optimization for stream-oriented workload over continuous data streams, but focused on memory limitations because they were designed for a single-server deployment [20, 28, 45]. On the other hand, we consider a wide-area environment where *the limited WAN bandwidth is typically the main constraint.*

Illustrative Example. To make the problem concrete, consider the following illustration. Suppose there are 2 analytics queries that are submitted to the system:

`Query 1`: A marketing group is monitoring the trending topics in Twitter across the US, Europe, and Asia to support their operational decisions:

```
SELECT Time, Topic, COUNT(*)
FROM   Host.US, Host.EU, Host.Asia
GROUP BY WINDOW(Time.Minutes(1)), Topic
HAVING COUNT(*) > 100
```

`Query 2`: Another group of analysts is monitoring the impressions from Twitter users in the US and Europe that are related to a specific type of campaign:

```
SELECT Time, AdInfo.Campaign
FROM (SELECT Time, Topic
      FROM Host.US, Host.EU
      GROUP BY WINDOW(Time.Seconds(30)), Topic
      HAVING COUNT(*) > 100) AS Tweet, AdInfo
WHERE AdInfo.Topic = Tweet.Topic
```

Figure 1(a) shows the logical execution plans of both queries. In this example, both queries subscribe to common input sources (`US` and `EU`), deserialize, filter, reduce the data (σ and π) to remove irrelevant information (e.g., discard user profile), aggregate the results ($\cup$), and send only the relevant information to their corresponding final locations. In the case of `Query 2`, the intermediate results are further joined ($\bowtie$) with static data that are stored in `AdInfo`.

Figure 1(b) shows the independent deployment of the two queries. For clarity reasons, suppose the input stream rate from each source is $10MB/s$ and the *selectivity* of each selection and projection operator is 0.5. We also consider the data transfer overhead within a site to be negligible since intra-data center bandwidth is typically 1-2 orders of magnitude higher than inter-data center bandwidth [61]. In this case, deploying the two queries independently will consume WAN bandwidth with a rate of $75MB/s$ ($40MB/s$ for `Query 1` and $35MB/s$ for `Query 2`).

However, we can see that both queries partially share common input streams (`US` and `EU`) and perform similar data processing (e.g., filtering user info). If the query optimizer is able to identify these commonalities, it may combine their common executions, which will significantly reduce the WAN bandwidth consumption rate to $50MB/s = 40MB/s + 10MB/s$ (Figure 1(c)), which saves ~33% of the original bandwidth consumption. This illustration shows that optimizing multiple query executions in wide-area streaming analytics can significantly save WAN bandwidth consumption.

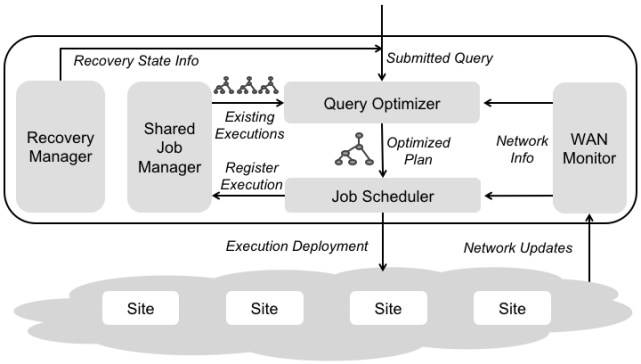


Figure 2: Sana system architecture.

In addition to saving WAN bandwidth consumption, sharing common executions between multiple queries can also improve the overall performance in the face of bandwidth constraints. In the previous example, if the available bandwidth from the Virginia data center to the London data center is less than $10MB/s$, deploying the two queries independently will result in bandwidth contention. One possible solution is to reduce the data transmission rate over the bottleneck link through approximation, aggregation, or data reduction, which trades the output's quality for higher overall performance [18, 20, 28, 54]. Alternatively, the query optimizer may choose a less optimal query plan that avoids the congested link [61]. However, we argue that making this trade-off is unnecessary if the system is able to detect that the problem arises due to redundant data transmission. Furthermore, these techniques still result in a wasteful bandwidth consumption that could be reduced.

3 SANA: SYSTEM ARCHITECTURE

We propose a geo-distributed stream proccessing system called `Sana` which implements multi-query optimization in a WAN-aware manner. Figure 2 shows the system architecture of `Sana`. When a new (possibly a recovery) query is submitted to the system, the *Query Optimizer* will optimize its execution plan while considering the inter-site bandwidth information that is monitored by the *WAN Monitor*. This network information is particularly important in wide-area settings since the optimal execution plan of a wide-area data analytics query highly depends on the WAN bandwidth availability between sites [61].

When applying multi-query optimization, the *Query Optimizer* will also consider the deployment of the existing queries that is provided by the *Shared Job Manager* to identify any commonalities between the newly submitted queries and the existing ones (§4). After the optimized query execution plan has been generated, the *Job Scheduler* will schedule and deploy each operator instance on a compute node in a WAN-aware manner to minimize the overall query execution latency and/or WAN bandwidth consumption (§5). Once a query has been deployed, it may periodically checkpoint its execution state and send the state metadata to the *Recovery Manager*. This mechanism allows the system to replay a query from its last checkpointed execution state in the case of failures. The implementation details will be discussed in §6.

4 MULTI-QUERY OPTIMIZATION

In this section, we look at how the query optimizer optimizes multiple query executions by sharing any commonality between them. We first study different types of sharing opportunities that can be exploited between two queries (§4.1), and show how to apply them across multiple queries (§4.2). We will discuss the WAN awareness in optimizing multiple query executions in §5.

4.1 Sharing Opportunities

4.1.1 Input-Operator Sharing. A natural way to determine whether two queries share common executions is to compare their vertices. Two vertices v_1 and v_2 are considered equivalent *iff* they share the same input streams $I_{v_1} = I_{v_2}$, perform the same transformation function $f_{v_1} = f_{v_2}$, and thus produce the same output streams $O_{v_1} = O_{v_2}$. We refer to this type of sharing as `IN-OP`. In this case, deploying the two vertices independently will result in a full redundancy in both transmitting and processing duplicate data. This redundancy can be eliminated by deploying only one of the vertices. In this case, the query optimizer can merge the two vertices together, i.e., let the job scheduler know that v_2 does not need to be scheduled if v_1 has already been deployed.

In practice, two vertices may share common inputs and operators, but output the results to a different set of downstream vertices (possibly with some overlap). We denote the set of v's downstream vertices as $D_v = \{d_v^0, \ldots, d_v^{n-1}\}$. These conditions are especially common in the early stages of executions where multiple queries may read the same input streams from the same data sources although their downstream vertices tend to be more specific to each individual query. In this case, the output streams to any of the downstream vertices that are not shared by the two vertices need to be replicated, while the common outputs can be transmitted only once (Figure 3).

4.1.2 Input-Only Sharing. Since in practice multiple vertices with different operators/transformation functions may rely on a common set of input streams, we relax the sharing requirement of the `IN-OP` type of sharing by removing the operator-equality condition, i.e., $f_{v_1} \neq f_{v_2}$, therefore $O_{v_1} \neq O_{v_2}$. This allows two vertices to share their common input streams even though they have different operators. We refer to this input-only sharing as `IN`. In this case, independently deploying the two vertices will result in redundancy in transmitting duplicate input data. Unlike `IN-OP`, this type of sharing requires both vertices to be deployed since they rely on different transformation functions. However, applying this type of sharing will eliminate the redundancy in transmitting duplicate input streams from their common upstream vertices, which can be highly beneficial in the case where the inputs are transmitted over slow and limited bandwidth links, as in a wide-area environment.

In wide area settings, the `IN` type of sharing can be exploited by deploying the two vertices on the same site (or the same node). However, the physical deployment of a stream operator is typically determined by the job scheduler after the query execution plan has been generated by the query optimizer. Thus, the query optimizer needs to provide a *hint* to the job scheduler in exploiting this type of sharing. The co-location deployment of two vertices does not necessarily eliminate the redundancy in transmitting duplicate data because they are still considered as two independent stream edges

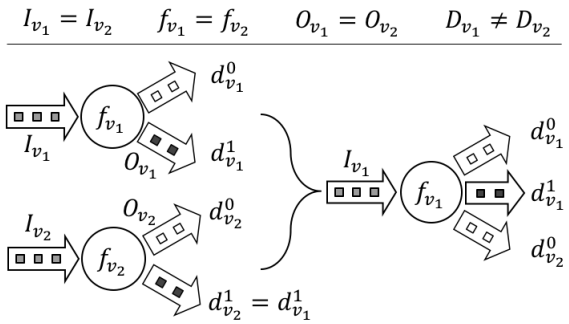


Figure 3: `IN-OP`: Input-Operator Sharing. Here, v_1 and v_2 share common input streams and operators, but only partially share the output streams.

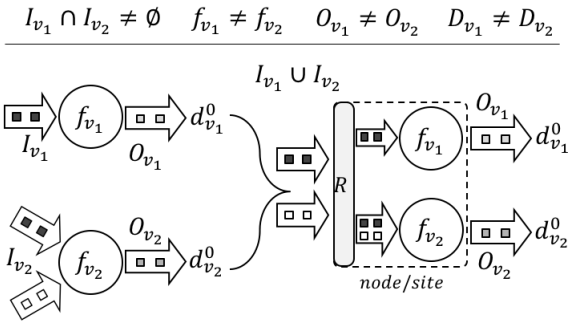


Figure 4: `IN`: Input Sharing. Here, v_1 and v_2 only partially share common input streams.

to their respective downstream vertices. To exploit this type of sharing, we introduce a lightweight *router* operator R which (1) keeps track of the input edges of each input stream originated from remote vertices, and (2) forwards each record to every downstream vertex without performing any data transformation. Note that the *router* operator does not buffer nor batch the records, instead it only routes the records to multiple operators, similar to the task of router in networks. Thus, the overhead of the router operator is negligible as shown in §7.

Partial Input Sharing. In the case of `IN-OP`, two vertices that share common operators must rely on the same *exact* input streams since in general applying the same transformation to different input sets does not guarantee the same resulting outputs. This strict input-stream-equality can further be relaxed in the case of `IN` since the two vertices do not rely on the same transformation results. Thus, the `IN` type of sharing allows two vertices with different operators to *partially* share their input streams (Figure 4).

4.2 Sharing Across Multiple Queries

Having discussed different sharing opportunities that can exist between two queries, we will now look at how the query optimizer exploits these opportunities across *multiple* queries. Since most streaming analytics queries are long-running, it is possible that a newly submitted query exhibits common executions with multiple existing queries that may have already been deployed. Thus, the query optimizer needs to determine with *which* of the queries it should share the new query.

One possible approach to determine *which* query to share is by finding a query that exhibits the highest *similarity score* using a subgraph-matching algorithm [19, 39]. However, we argue this approach is sub-optimal since it limits the sharing opportunities to only 1 query. Instead of finding the similarity in a *query-centric* manner, we adopt a *vertex-centric* philosophy where a query may share its vertices with *multiple* queries. This will result in a higher overall degree of sharing. We compare a new query with each of the existing queries topologically from the *source* vertices. Traversing the vertices in topological order gives the benefit of early termination in traversing a graph. If two vertices are not equivalent ($v_1 \neq v_2$), by definition, none of their downstream vertices are equivalent, and hence they do not need to be compared.

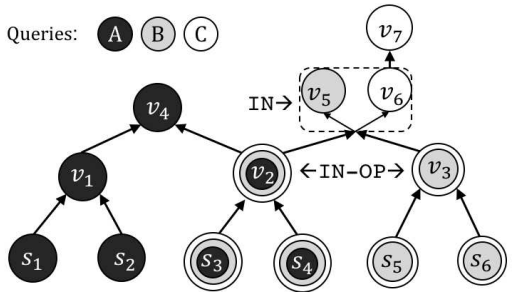


Figure 5: Exploiting common executions across multiple queries: Here, `C` shares its execution with both `A` and `B`.

Although finding common vertices among multiple queries can be computationally expensive, this step is only perfomed during the query planning stage. Since most streaming analytics queries are long-running, this overhead is justified for higher overall execution performance and better resource utilization. To reduce the analysis cost, the query optimizer may limit the number of queries to be analyzed or adopt a group-based analysis, as proposed by existing work in Internet Databases [16], which reduces the number of vertices that need to be analyzed.

Figure 5 shows an example where a query (`C`) shares its execution with multiple existing queries (`A` and `B`). When `C` arrives, the query optimizer finds that `C` shares (1) common input-operators with `A` and `B` at vertex v_2, as well as (2) input streams with `B` ($I_{v_5} \cap I_{v_6} \neq \emptyset$). In this case, the query optimizer may exploit these sharing opportunities by merging the common executions of these queries. Thus, the job scheduler only needs to deploy two additional vertices for query `C`: v_6 that exploits `IN` type of sharing with v_5, and v_7 that does not exhibit any sharing opportunity with the rest of the vertices, while s_3, s_4, s_5, s_6, v_2, and v_3 are shared with `IN-OP` type of sharing.

5 WAN-AWARE OPTIMIZATION

Our discussion so far has focused on the sharing opportunities between multiple queries without considering the wide-area constraints. In this section, we focus on addressing the challenges of applying these sharing opportunities in a wide-area environment. Specifically, we propose a WAN-aware optimization to the

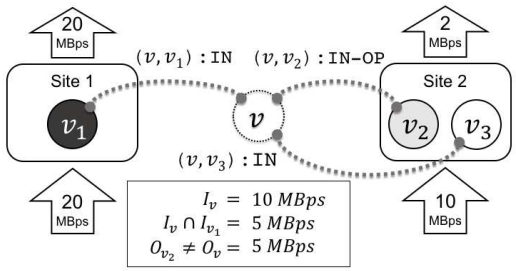


Figure 6: Multiple sharing opportunities: Here, v exhibits `IN-OP` with v_2, and `IN` with both v_1 and v_3.

query optimizer in generating and optimizing query execution plans while considering the sharing opportunities with existing query executions (§5.1) and WAN-aware operator placement to the job scheduler in deploying stream operators (§5.2).

5.1 WAN-Aware Query Planning

In the context of wide-area data analytics, the query optimizer needs to be aware of the inter-site bandwidth availability to generate an optimized query execution plan for each individual query [61]. Similarly, the query optimizer must also optimize multiple query executions in a WAN-aware manner. The WAN awareness in this context is used to determine whether a query should share its execution with other queries (when possible) based on the current WAN bandwidth availability between sites. Without WAN awareness, sharing executions across multiple queries may result in WAN bandwidth contention that will degrade the performance of either or both the new and the existing queries.

Since our query optimizer analyzes the commonality between queries in a vertex-centric manner, a vertex may exhibit more than one sharing opportunities with multiple vertices from different queries. Figure 6 shows a situation where vertex v can share both its inputs and operator with v_2, or partially share its inputs with either v_1 or v_3. In this case, the query optimizer needs to determine *which* of these sharing opportunities should be exploited, or decide not to share the execution at all.

One possible approach is to choose a vertex that maximizes the degree of sharing since intuitively it will maximize the duplicate elimination. However, this naive approach may result in a performance degradation. Consider the scenario shown in Figure 6. If the query optimizer always tries to maximize the sharing regardless of the network conditions, it will exploit the `IN-OP` type of sharing with v_2 since the input streams of vertex v are fully covered by v_2. However, we can see that Site-2 does not have sufficient bandwidth capacity for transmitting its output streams. Thus, exploiting `IN-OP` with v_2 may result in bandwidth contention between v, v_2, and v_3. On the other hand, if the query optimizer is aware of the bandwidth constraints, it may exploit the `IN` type of sharing with v_1 by partially sharing their input streams at Site-1. This decision is preferable because it does not cause any bandwidth contention that may degrade the overall performance. Thus, there is a trade-off between minimizing bandwidth consumption (maximizing sharing) and maximizing the performance of concurrent executions.

Algorithm 1 WAN-aware execution sharing

```
1:  procedure FIND-COMMON-VERTICES(v, V)
2:      for v_i ∈ V topologically do
3:          share ← getShareType(v, v_i)
4:          (BW_in, BW_out) ← getBandwidth(v_i)
5:          if share == IN-OP then
6:              ΔO ← (|D_v ∪ D_{v_i}| / |D_{v_i}|) × O_v
7:              if BW_out > ΔO then
8:                  add v_i to the set of IN-OP vertices
9:              end if
10:         else if share == IN then
11:             ΔI ← I_v − I_{v_i}
12:             if BW_out > O_v and BW_in > ΔI then
13:                 add v_i to the set of IN vertices
14:             end if
15:         end if
16:     end for
17: end procedure
```

Algorithm 1 shows how the query optimizer considers WAN bandwidth availability to determine *which* sharing opportunities (if any) to be exploited. In the case of `IN-OP`, the query optimizer needs to ensure that the site where the shared vertex v_i has been deployed, has sufficient egress bandwidth capacity to transmit additional output streams (Line 7). This can be estimated proportionally to the increase in the number of output stream consumers since both vertices rely on the exact same output data streams ($O_v = O_{v_i}$), and only their downstream vertices are different. In the case of `IN` where vertices only share partial input streams, the query optimizer needs to further ensure there is sufficient bandwidth in both the ingress and egress links to transmit additional input and output streams respectively. If the query optimizer predicts that exploiting the opportunity can potentially result in bandwidth contention, it will not exploit the opportunity, which trades off bandwidth utilization for higher overall performance.

Note from Lines 8 and 13 that the query optimizer outputs a set of vertices that can be shared by each vertex (if any) instead of only a single vertex as long as they ensure sufficient bandwidth for deployment. In this case, the job scheduler needs to choose *which* vertex to be shared. We adopt this design to give the job scheduler a flexibility to apply different optimization in scheduling different queries. For example, some queries may tolerate higher delay for lower bandwidth consumption while others may require real-time results even though they consume more bandwidth.

5.2 WAN-Aware Operator Scheduling

While the previous section focuses on bringing WAN awareness to the query optimizer in planning a query execution while considering any commonality with existing queries, this section focuses on incorporating WAN awareness to the job scheduler in deploying the execution. Once the query optimizer has identified a set of vertices that can be shared for each vertex in the query execution plan, the job scheduler is responsible for the actual deployment of the vertices themselves. Algorithm 2 shows how the job scheduler schedules each operator while considering the sharing opportunities that have been identified by the query optimizer. The scheduler will place and deploy each operator in the physical execution graph

Algorithm 2 WAN-aware operator placement

```
1:  procedure SCHEDULE(v)
2:      if find v_i ∈ set IN−OP then
3:          add edges from v_i to ΔD ← D_v \ D_{v_i}
4:      else if find v_i ∈ set IN then
5:          deploy v at the same site as v_i
6:      else if I_v are local input streams then
7:          site-locality deployment
8:      else                          ▷ neither share-able nor a local operator
9:          WAN-aware deployment
10:     end if
11: end procedure
```

topologically based on the deployment of its upstream vertices. Although this approach may not result in the most optimal end-to-end deployment of the entire graph, this has been shown to work reasonably well in practice with significantly lower complexity [30].

In exploiting the sharing opportunities, the job scheduler prioritizes exploiting IN−OP over IN because the gain of IN−OP ≥ IN in terms of minimizing WAN bandwidth consumption since the former type of sharing covers the benefits of the latter. Note that exploiting any of these opportunities guarantees sufficient bandwidth deployment since the query optimizer has already omitted those that may result in bandwidth contention. If a vertex exploits the IN−OP type of sharing with any of the existing vertices, the job scheduler does not need to deploy the vertex. However, the job scheduler may need to update the existing execution by creating additional edges from the shared vertex to any of the additional downstream vertices that are not shared by the two vertices (Line 3). On the other hand, vertices that exhibit IN type of sharing will be deployed on the same site as their corresponding shared vertices to mitigate redundant data transmission over the WAN (Line 5).

If a vertex can be shared with multiple vertices of the same sharing type (e.g., v exhibits IN with both v_1 and v_3 in Figure 6), the job scheduler needs to determine *which* of the vertices should be shared (Lines 2 and 4). Since our goal is to minimize WAN bandwidth consumption, our job scheduler will choose a vertex that maximize the sharing. Although maximizing sharing may not necessarily minimize the execution latency, in practice this will result in an improved execution performance [62]. If the goal is to minimize delays, the scheduler may choose the vertex that minimizes latency.

Vertices that do not exhibit any sharing opportunities will be deployed based on the locations of their input streams. Those that rely only on local input streams will be deployed on the same site as their upstream vertices to minimize the communication overhead, especially the high latency of the wide area network. On the other hand, vertices that rely on one or more input streams originated from remote sites will be deployed using a WAN-aware operator deployment. We adapt the cost model from Hourglass [52] which optimizes stream operator placement that balances WAN bandwidth consumption and latency, by minimizing $\sum_{\ell \in L} \frac{DR_\ell (Lat_\ell)^2}{BW_\ell}$ where ℓ is a link between two sites, DR_ℓ is the data rate transmitted over the link, Lat_ℓ is the latency overhead, and BW_ℓ is the available bandwidth of the link. Any updates on the workload of a link will be reflected in the bandwidth availability that is continuously being monitored by the WAN Monitor.

6 IMPLEMENTATION

We have implemented Sana in a system prototype based on Apache Flink [13] - a stream processing system with the dataflow computational model. We have modified and adapted the original Flink system to a wide-area environment by implementing network monitoring and multi-query optimization modules, as well as incorporating WAN awareness to both the query optimizer and job scheduler.

WAN Bandwidth Monitoring. The bandwidth availability between sites is continuously monitored by the WAN Monitor. Congested links are detected by the ratio of the current bandwidth utilization over the maximum available bandwidth [54]. A ratio of <1 indicates that the network link has spare bandwidth capacity while a ratio >1 indicates that the bandwidth is contended. This available bandwidth information is shared with both the query optimizer and the job scheduler to implement the WAN-aware query planning (§5.1) and operator scheduling (§5.2) policies respectively.

Multi-Query Optimization. We have implemented our WAN-aware multi-query optimization module in Flink to find any common executions between a newly submitted and existing queries in a WAN-aware manner. To exploit the IN type of sharing, the query optimizer will modify the original query's execution plan by adding a *router* operator for every vertex that rely on remote input streams. The *router* operators are added proactively to prevent suspending the execution of an existing vertex. Although the use of *router* operators would still incur duplicate data streams from the *router* to the downstream operators, this data forwarding happens within a local environment (within a site or even a node) and hence, its overhead is negligible compared to the overhead from transmitting duplicate data across sites. We show in §7 that the overhead of the *router* operator is negligible even when it is not shared.

WAN-aware Scheduling. The default Flink scheduler has already implemented node-locality scheduling, which tries to schedule a vertex on the same node with any of its upstream vertices. However, if an operator relies on input streams from different nodes, the original scheduler will choose one of the nodes without considering the network condition (bandwidth availability and latency) between them. This simple policy works well in a centralized cluster environment, for which Flink has been designed. However, this scheduling policy may result in a non-optimal operator placement in wide-area settings. We have modified the default Flink's scheduler by incorporating the WAN awareness discussed in §5.2.

Fault Tolerance. A query whose vertices are shared with other queries may be terminated either intentionally (e.g., the analysis is complete) or unintentionally (e.g., failure in one of the vertices in the query plan). To handle these issues, the Shared Job Manager keeps track of every vertex that is shared with other queries. Whenever a query that shares a vertex is terminated, it removes the reference to the shared vertex. A vertex execution will only be terminated if *all* queries that share the execution have been terminated. This simple approach prevents cascading failures unless they happen directly on the stream operator logic.

Recovering from failures that involves shared vertices is challenging since a stream processing system needs to ensure the exactly-once semantic processing guarantee. Sana uses a *checkpoint-and-replay* fault recovery mechanism, where each query periodically checkpoints its processing state and thus the system can

Table 1: Query Set

Category	Query Examples	Num. Operators
Tweet Statistics	[rate, count] of [tweet, hashtag] on [location, language, topic]	10-18
Users Analysis	[rate, count] of [tweet, hashtag, retweet] on [gender, age-group] per [location, language]	12-18
Top-k Analysis	Top-k [popular, trending] [hashtag, topic, retweet] per [language, location]	10-15
Sentiment Analysis	[aggregate, categorize] sentiment of each [hashtag, country, topic]	12-18
System Load	[rate, count] of [bandwidth usage, request] per [node, region]	6-10

restore its execution from the last checkpointed state upon recovering from failures [12]. We maintain an independent state for each vertex that is shared by multiple queries. Thus, if a sharing query fails, other queries can continue their executions and update their states independently. When a failed query is restarted, it may not be able to immediately share the vertex it was sharing earlier since the shared vertex may have a different state. In this case, the query needs to catch up its processing in order to re-share the vertex.

Query Reconfiguration. Since many streaming analytics queries are long running, a query needs to gracefully adapt to runtime dynamics, such as changes in workload or network topology [27, 35]. In this case, the query optimizer and the scheduler may change the execution plan and/or the deployment of the query respectively whenever the environment changes significantly. We are currently investigating different adaptability policies that can be used to dynamically adapt a query execution efficiently to handle runtime dynamics in a wide-area environment without sacrificing performance nor the resulting output quality [36].

7 EVALUATION

Experimental Setup. We experimentally evaluate the effectiveness of `Sana` using a wide-area system deployment across 14 geo-distributed EC2 data centers. The compute nodes were deployed on 8 of the sites (Virginia, California, Canada, London, Frankfurt, Sydney, Tokyo, and Singapore) and the input streams are generated by external sources that were located on the other 6 sites (Ohio, Oregon, Ireland, Seoul, Mumbai, and Sao Paulo). To prevent an inaccurate evaluation caused by the data exchange overhead between the external sources and the system, we follow the design proposed by recent work which uses distributed in-memory data generators instead of message brokers as the external sources [38].

We also measured the bandwidth availability and the latency between the sites prior to running the experiments as initial network information to the Network Monitor. Our measurements show that WAN bandwidth between EC2 data centers ranged from 20Mbps to 280Mbps, confirming a similar trend from prior work [30, 61].

Dataset and Queries. All experiments are based on real Twitter data that was collected from Twitter Streaming APIs [2] in December 2015. It consists of approximately 4 million tweets per day. Since the trace only represents a sample of real Twitter workload, we scaled the playback rate to 6000~8000 tweets per second to reflect the actual tweet rate [4]. The tweets were distributed across the input sources based on their geographic information.

We implemented 12 analytical queries based on actual streaming analytic queries on Twitter streams [1, 3]. Table 1 shows the summary of the queries. Each query consisted of various combination of operators including *map*, *reduce*, *filter*, *join*, *union*, and *window*. Each query subscribed to 4-6 input sources and outputs its final result locally at the *sink* operator. Some of the queries also rely on static data sources. For example, in the case of trend analysis, the query discards all the irrelevant words by consulting an external database. Another example includes a sentiment dictionary used in sentiment analysis. In all of the experiments, each query is submitted independently with a time gap of 10 seconds to mimic the independent deployment of most streaming analytics queries in a practical scenario. Hence, batching multiple queries together prior to their deployment is impractical.

Evaluation metrics. We use the following metrics to evaluate and compare the performance of the systems:

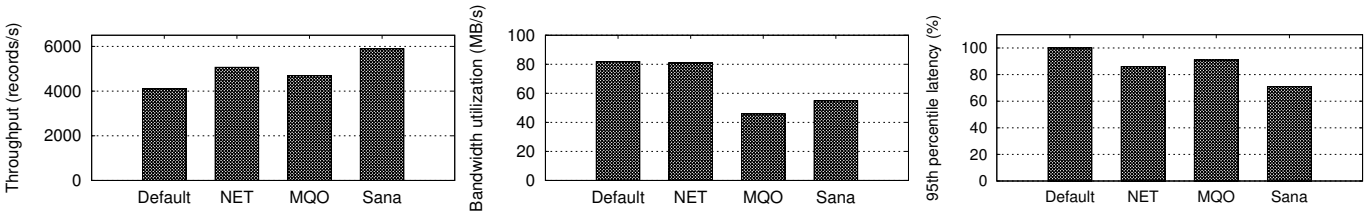
- *Throughput:* The average rate of *distinct* records/second processed by the system for each query. In the face of bandwidth constraints, the system may trigger a backpressure to reduce the rate of an input stream.
- *WAN Bandwidth Utilization:* The average rate of records (including duplicates) transmitted over the WAN. This is particularly critical in a wide-area environment that typically has limited bandwidth.
- *Latency:* The latency is measured as an *event time* latency, which is the difference between the time when a record is generated at the external data source and when its processed output is written to the final location by the *sink* operator.

7.1 Baseline System Comparison

We evaluated the benefit of WAN-aware multi-query optimization by comparing the following systems:

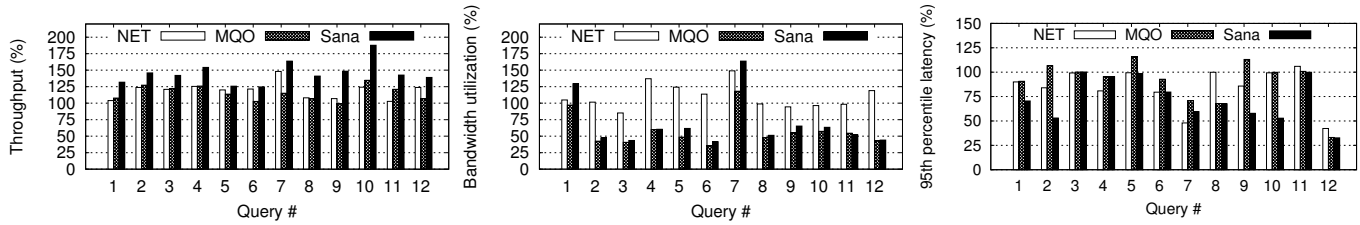
- `Default`: The default Flink system that does not allow sharing executions nor does it implement WAN-aware scheduling, but implements node-locality scheduling.
- `NET`: A modified Flink system that adopts the WAN-aware task scheduling algorithm as proposed in Clarinet [61] which minimizes the execution time by distributing tasks across sites that provide sufficient bandwidth. However, it does not allow queries to share common executions. The batch-schedule optimization in Clarinet is not applicable to this context due to the independent deployment of the queries.
- `MQO`: A modified Flink system that allows queries to share common executions. But, it does not implement WAN-aware scheduling (default Flink scheduler).
- `Sana`: Our modified Flink system that incorporates the WAN awareness in both optimizing multiple query executions and scheduling stream operators.

[2] https://developer.twitter.com/en/docs



(a) Overall throughput (b) Average bandwidth utilization (c) Average 95th latency relative to `Default`

Figure 7: Overall system performance comparison



(a) Per-query throughput (b) Per-query bandwidth utilization (c) Per-query latency

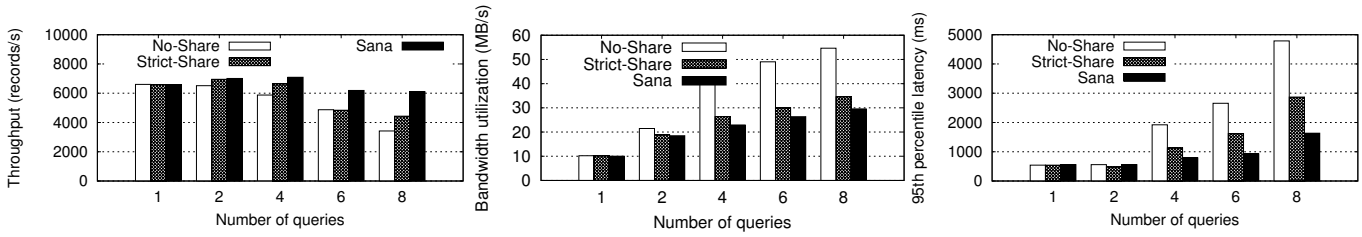
Figure 8: Per-query performance gain and bandwidth utilization relative to `Default`.

Figure 7 compares the overall performance of different systems for running the 12 queries concurrently. We can see from Figure 7(a) that `Sana` resulted in 44%, 16%, and 26% higher throughput compared to `Default`, `NET`, and `MQO` respectively. Figure 7(b) also shows that `Sana` was able to achieve these performance gains while consuming significantly lower bandwidth compared to both `Default` and `NET` (~33% less bandwidth utilization). This indicates that `Sana` can efficiently utilize WAN bandwidth by preventing transmitting duplicate records over the constrained network links. Although `MQO` consumed less bandwidth with respect to `Sana`, the bandwidth utilized by `Sana` is effectively used to transmit a higher number of records per second. Furthermore, `MQO` resulted in a higher overall latency compared to `Sana` and `NET`, as shown in Figure 7(c). This highlights the importance of WAN-aware operator scheduling to *effectively* utilize limited network bandwidth in a wide-area environment. The latency and throughput gains achieved by `MQO` with respect to `Default` is because `MQO` utilized the available bandwidth more efficiently by preventing transmitting redundant data over the WAN.

We further break down the overall performance and WAN bandwidth consumption rate of the queries to observe the gain for each individual query relative to `Default` (see Figure 8). We make a few observations. First, we can see from Figure 8(a) that `NET` was able to improve the overall throughput of each query by up to 48% and resulted in 40% lower latency compared to `Default` (see Figure 8(c)). However, we can also see from Figure 8(b) that the WAN-aware scheduling in `NET` that tries to minimize query execution latency does not reduce the overall WAN bandwidth consumption even though it resulted in higher throughput. This indicates that `NET` was able to process a higher rate of data streams by avoiding overloaded network links.

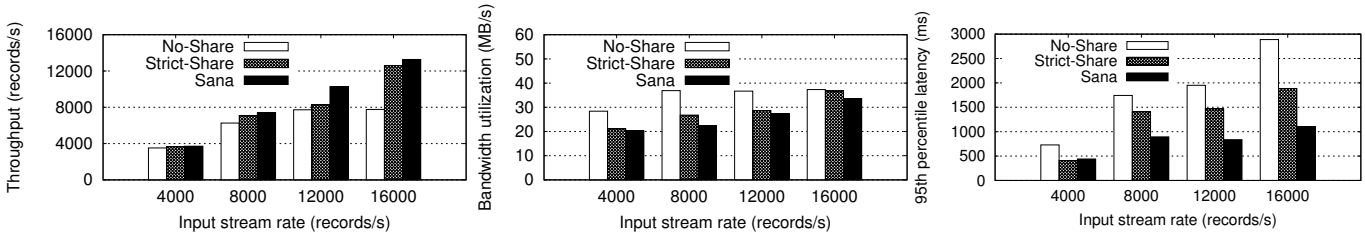
Secondly, we can see from Figure 8(b) that `MQO` is able to significantly reduce the bandwidth utilization by up to 60% by sharing common executions between queries. The only cases where the `MQO` could not reduce the bandwidth utilization were for query 1 and 7 which do not exhibit any commonality with the other queries. However, we can see from Figure 8(a) that query 7 was able to process more data streams with a similar increase. This indicates that the bandwidth was *efficiently* used for transmitting a higher rate of data streams. We can also see from Figure 8(b) and Figure 8(c) that although `NET` consumed higher network bandwidth compared to the `MQO` it was able to outperform `MQO` for most queries in terms of minimizing execution latency. This shows that minimizing WAN bandwidth consumption in a wide-area environment does not necessarily minimize the query execution latency.

Thirdly, we can see that `Sana` improves the overall performance of each query execution while significantly consuming less network bandwidth. Specifically, it resulted in up to of 87% higher throughput and 68% lower latency compared to `Default`. Similar to the `MQO` case, both query 1 and 7 consumed higher bandwidth, but the extra bandwidth is used for transmitting more data. Furthermore, `Sana` also achieves 21% higher throughput compared to `NET` by eliminating redundant data transmission, as reflected by the reduction in bandwidth utilization for most queries. Lastly, even though `Sana` consumed more bandwidth compared to `MQO`, it resulted in a higher throughput for transmitting more data. These experiments show `Sana` can utilize WAN bandwidth effectively and efficiently.



(a) Average throughput (b) Average bandwidth utilization (c) 95th percentile latency

Figure 9: Impact of higher degree of sharing over different number of concurrent queries



(a) Average throughput (b) Average bandwidth utilization (c) 95th percentile latency

Figure 10: Impact of higher degree of sharing over different input stream rate

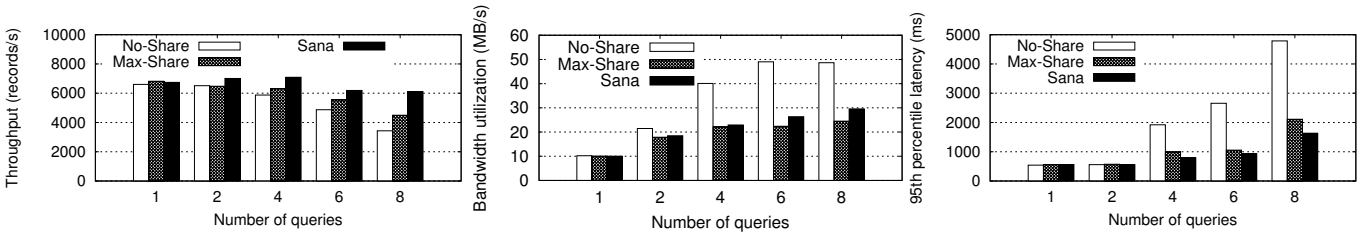
7.2 Impact of Degree of Sharing

In the next set of experiments, we explore the impact of degree of sharing in applying multi-query optimization. Specifically, the benefit of allowing queries to partially share common input streams even though their operators are different. All systems in the following experiments apply WAN-aware operator scheduling. Thus, the differences in the results are strictly based on the different execution plans generated by the query optimizer. We compare `Sana` (which allows `IN` and `IN-OP`) against (1) `No-Share`, which does not consider execution sharing, and (2) `Strict-Share` whose query optimizer only allows queries to share vertices if they share the same inputs and operators (`IN-OP` only). In contrast to `Strict-Share`, `Sana` allows queries to share partial input streams.

Varying number of concurrent queries. Figure 9 compares the three query optimizers over varying number of concurrent queries. In the case of a single query execution, all the query optimizers generated the same execution plan. However, as the number of queries increased `Sana` was able to exploit a higher degree of sharing by allowing queries to partially share their executions. This resulted in a lower bandwidth consumption and approximately 78% and 37% higher throughput execution compared to `No-Share` and `Strict-Share` respectively (see Figures 9(b) and 9(a)). We can also see from Figure 9(c) that allowing partial input sharing can also reduce the overall execution latency due to the higher bandwidth availability, which provides a higher flexibility to the job scheduler to deploy the queries more optimally.

Figure 9(b) shows that although `No-Share` consumed 41% and 65% higher bandwidth compared to `Strict-Share` and `Sana` respectively, it resulted in an overall lower throughput. This indicates there was a large amount of redundant data being transmitted over the WAN. The `Strict-Share` also consumed slightly more bandwidth compared to `Sana` but resulted in a lower throughput, which highlights the importance of (partially) sharing common input streams even for different operators. From Figure 9(c), we can also see that the overhead of the *router* operators that were added to route input streams from remote sites is negligible (~5%) even when they are not utilized, as shown in the case with 1 query execution. Thus, the *router* operator can reduce the redundancy in transmitting duplicate data over the WAN.

Varying input stream rates. In the following experiments, we evaluate the impact of the degree of sharing over different rates of input streams with 4 concurrent queries. Figure 10(a) shows that as the input data rate increases, `Sana` resulted in a higher throughput while consuming lower bandwidth compared to both `No-Share` and `Strict-Share` (Figure 10(b)). Furthermore, `Sana` was able to significantly reduce the overall execution latency compared to `No-Share` and `Strict-Share`, similar to the effect of increasing the number of queries (Figure 10(c)). This shows that (1) applying WAN-aware multi-query optimization allows the system to scale as workload increases, and (2) allowing queries to share common input streams even if they have different operators will further improve the performance and reduce the overall bandwidth consumption.



(a) Average throughput (b) Average bandwidth utilization (c) 95th percentile latency

Figure 11: WAN-aware query planning: trading-off bandwidth utilization for performance

7.3 WAN-Aware Execution Sharing: Bandwidth Utilization vs. Performance Trade-Off

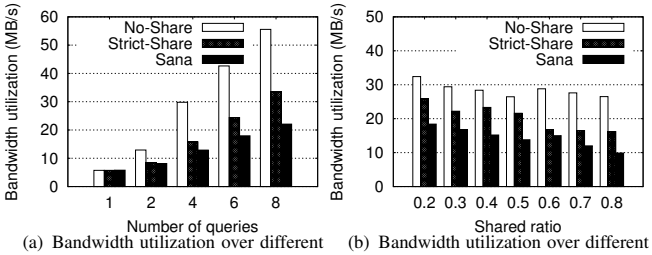
In the following experiments, we show the importance of network awareness in applying multi-query optimization in a wide-area environment to maintain high performance executions while reducing WAN bandwidth consumption (§5.1). We compared `Sana` against (1) `No-Share` which did not exploit any execution sharing and (2) `Max-Share` which allowed queries to share common executions but did not consider the WAN bandwidth availability in sharing executions. In contrast to `Sana`, the latter will always try to exploit any sharing opportunity that maximize the sharing regardless of the WAN bandwidth availability, which is essentially the traditional multi-query optimization for a local environment. The main problem with maximizing sharing without network awareness in a wide-area environment is that it may result in WAN bandwidth contention between queries, which can degrade the performance of either or both the sharing and the shared executions.

Figure 11(a) and Figure 11(c) show that `Sana` resulted in 35% higher throughput and 23% lower latency compared to `Max-Share`, but consuming approximately 20% higher bandwidth. The performance gain achieved by `Sana` compared to `Max-Share` is because `Sana`'s query optimizer prevented exploiting sharing opportunities that led to bandwidth contention which would degrade the overall performance. We can also see that as the number of queries increases, the performance gap between `Sana` and `Max-Share` also increases. This indicates that the WAN awareness in `Sana` resulted in less number of contended links. Thus, there is a trade-off between minimizing WAN bandwidth utilization and maximizing the overall performance of multiple query executions.

7.4 Potential Bandwidth Saving

In the following experiments, we observe the potential bandwidth saving from applying multi-query optimization in the case where network bandwidth is not constrained. We deployed `Sana` on a localized CloudLab[3] environment where the available bandwidth between nodes are higher than the rate of the data streams. In such a condition where bandwidth is sufficient, reducing the data transfer over the network is still desirable in a wide-area environment since WAN bandwidth is expensive in terms of monetary cost [64].

[3]https://www.cloudlab.us/



(a) Bandwidth utilization over different number of queries (b) Bandwidth utilization over different shared-ratio

Figure 12: Network bandwidth saving

Figure 12(a) and Figure 12(b) show the average bandwidth consumption rate over different number of concurrent queries and sharing ratio respectively. The sharing ratio is defined as the percentage of vertices that are shared between queries. The average sharing ratio between the queries in Figure 12(a) was approximately 0.2 whereas the number of concurrent queries in Figure 12(b) was set to 4. We can see from both figures that `Sana` greatly mitigates the bandwidth consumption as the number of queries and the sharing ratio increase. Specifically, it resulted in up to 60% reduction in bandwidth consumption rate compared to the sharing-agnostic approach. Thus applying multi-query optimization even in an unconstrained wide-area environment can still reduce the bandwidth utilization and save monetary cost.

8 RELATED WORK

Geo-Distributed Data Analytics. Table 2 shows where `Sana` stands in the world of geo-distributed data analytic systems. Iridium [53] proposes a WAN-aware optimization that minimizes query execution latency for batch-oriented workloads by proactively migrating input data prior to the arrivals of queries based on history. Geode [62] also relies on recurring queries but focuses on minimizing WAN bandwidth consumption by sending only the *diff* of input data over the wide-area network for subsequent queries. In contrast to both approaches, `Sana` focuses on stream-oriented workloads where most queries rely on continuous data streams that are continuously being generated in real time. Furthermore, `Sana` does not make any assumptions on the query arrivals. None of these techniques support multi-query optimization.

Table 2: Geo-distributed Data Analytics Systems

Systems	Workload Type	WAN-Aware Optimization	Multi-Query Optimization
Iridium [53]	Recurring	Data and task placements prior to query arrivals	N/A
Geode [62]	Recurring	*diff* or incremental data transfer over the WAN	N/A
Clarinet [61]	Batch	WAN-aware query plan selection	Multiple job scheduling
Tetrium [30]	Batch	Heterogeneous network and compute resource scheduling	Multiple job scheduling
JetStream [54]	Stream	Data aggregation/degradation using data cube abstraction	N/A
Sana	Stream	WAN-aware operator sharing and scheduling	Execution sharing (data transfer and processing)

Both Clarinet [61] and Tetrium [30] look at optimizing batch-oriented queries in a wide-area environment. Specifically, Clarinet incorporates WAN awareness into the query optimizer to choose a query execution plan based on inter-site bandwidth availability, whereas Tetrium additionally considers the heterogeneity of computational resources across sites in scheduling jobs. In addition to incorporating WAN-aware optimization for single-query deployment, both of them consider optimizing multiple query executions by batch-scheduling multiple queries (jobs) rather than scheduling each query independently. This approach, however, is not practical for stream-oriented workloads. Furthermore, they do not allow queries to share common executions and hence their techniques would still result in a redundant data transmission and processing. In contrast, Sana can eliminate redundant executions and optimize multiple query executions in an incremental manner, which is critical for long-running, continuous queries.

Recent attempts also consider optimizing stream-oritented workloads in a wide-area environment. Photon [7] and Ubiq [10] address the fault tolerant aspect of geo-distributed data analytics over continuous data streams in production clusters. JetStream [54] handles WAN bandwidth limitation by making a trade-off between quality and performance, which may not be applicable for queries that rely on exact computations. Heintz et al. [26] propose an online algorithm that trades off timeliness and acccuracy in the context of windowed grouped aggregation. Pietzuch et al. [52] examine the problem of operator placement on the open Internet environment. Although they related to our work, they mainly focus on optimizing each individual query independently.

Others have also looked at optimizing different types of workload in a wide-area environment. Gaia [29] proposes a system that optimizes machine learning workloads in a wide-area environment by identifying and eliminating any insignificant updates over the WAN. Monarch [34] focuses on geo-distributed graph analytics workloads by optimizing existing processing model of a graph processing systems in a wide-area environment.

Multi-Query Optimization. The problem of multi-query optimization has been extensively studied in classic RDBMS [21, 55] and have been adopted for OLAP workloads [9, 11, 22, 24, 67] and later, data analytics [43, 49, 63]. Sana adopts the *data-centric* philosophy with pipelining technique [24] to share common executions between streaming queries. Although most of them are related to our work, they focus on a local environment whereas Sana focuses on a wide-area environment with different bottleneck. We show that applying traditional multi-query optimization designed for a local environment in wide-area settings without WAN awareness may lead to performance degradation.

Other research has also examined the problem of multi-query optimization over continuous data streams in streaming databases [20, 28, 56]. Seshadri et al. [56] propose an algorithm to find an optimal execution plan with reduced search space. Rule-based [28] and sketch-based [20] optimization have also been proposed for multiple queries over data streams, and NiagaraCQ [16] addresses the scalability issue in applying multi-query optimization for Internet Databases. Although they are related to our work, they are typically constrained by memory resources since they focus on a single-server deployment.

Incremental Processing and Caching Systems. It is worth mentioning that our work shares similarity with other work in incremental processing [46, 51] and caching systems [14, 23, 48, 65] since they also address the problem of redundant computation. However, they are orthogonal to our work. The incremental processing technique can be applied by each individual query by updating its state incrementally instead of computing from the beginning [37]. However, this is application-specific. Caching intermediate *hot* data also prevents performing redundant data processing, but it may not be applicable for queries that rely on real-time data streams. Thus, these techniques can be used in conjunction with our techniques.

9 CONCLUSION

This paper introduces Sana a system that optimizes multiple query executions in the context of wide-area streaming analytics. We observed that many streaming analytics queries often rely on common input streams from popular data sources and may exhibit common data processing. Thus, there is an opportunity of applying multi-query optimization in this context to mitigate the redundancy in transmitting duplicate data over the wide-area network (WAN) that has limited bandwidth. In this paper, we study different types of sharing opportunities and propose a multi-query optimization that allows multiple queries to *incrementally* share common executions in an *online* manner. We also address the importance of *WAN awareness* in applying multi-query optimization in a wide-area environment. We show that a WAN-agnostic multi-query optimization may lead to performance degradation. The evaluation using a wide-area system deployment across multiple geo-distributed EC2 data centers shows that Sana resulted in 21% higher throughput while saving WAN bandwidth utilization by 33% compared to a WAN-aware, sharing-agnostic system.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2017. Twitter analytics. (2017). https://analytics.twitter.com/about

[2] 2017. Twitter analytics for business. (2017). https://business.twitter.com/en/analytics.html

[3] 2017. Twitter sentiment analysis in Azure. (2017). https://docs.microsoft.com/en-us/azure/stream-analytics

[4] 2017. Twitter usage statistics. (2017). http://www.internetlivestats.com/twitter-statistics/

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[7] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*. ACM, 577–588.

[8] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.

[9] Subi Arumugam, Alin Dobra, Christopher M Jermaine, Niketan Pansare, and Luis Perez. 2010. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 519–530.

[10] Venkatesh Basker, Manish Bhatia, Vinny Ganeshan, Ashish Gupta, Shan He, Scott Holzer, Haifeng Jiang, Monica Chawathe Lenart, Navin Melville, Tianhao Qiu, et al. [n. d.]. Ubiq: A Scalable and Fault-tolerant Log Processing Infrastructure. ([n. d.]).

[11] George Candea, Neoklis Polyzotis, and Radek Vingralek. 2009. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment* 2, 1 (2009), 277–288.

[12] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. 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] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, Vol. 45. ACM, 363–375.

[15] Sirish Chandrasekaran and Michael J Franklin. 2002. Streaming queries over streaming data. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 203–214.

[16] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, Vol. 29. ACM, 379–390.

[17] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1789–1792.

[18] Michael Chow, Kaushik Veeraraghavan, Michael J Cafarella, and Jason Flinn. 2016. DQBarge: Improving Data-Quality Tradeoffs in Large-Scale Internet Services.. In *OSDI*. 771–786.

[19] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[20] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. 2004. Sketch-based multi-query processing over data streams. In *EDBT*, Vol. 4. Springer, 551–568.

[21] Sheldon Finkelstein. 1982. Common expression analysis in database applications. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. ACM, 235–245.

[22] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment* 5, 6 (2012), 526–537.

[23] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters.. In *OSDI*, Vol. 10. 1–8.

[24] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 383–394.

[25] Benjamin Heintz, Abhishek Chandra, and Ramesh K Sitaraman. 2015. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 133–144.

[26] Benjamin Heintz, Abhishek Chandra, and Ramesh K Sitaraman. 2016. Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics.. In *SoCC*. 361–373.

[27] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 15–26.

[28] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. 2009. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, 120–131.

[29] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. 2017. Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds.. In *NSDI*. 629–647.

[30] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 12.

[31] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. 2015. Scheduling jobs across geo-distributed datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 111–124.

[32] Jeong-Hyon Hwang, Ugur Cetintemel, and Stan Zdonik. 2007. Fast and reliable stream processing over wide area networks. In *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*. IEEE, 604–613.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[34] Anand Padmanabha Iyer, Aurojit Panda, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. 2018. Monarch: Gaining Command on Geo-Distributed Graph Analytics. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association.

[35] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. 2013. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 3–14.

[36] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Rethinking Adaptability in Wide-Area Stream Processing Systems. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association.

[37] Christopher Jonathan, Amr Magdy, Mohamed F Mokbel, and Albert Jonathan. 2016. GARNET: A holistic system approach for trending queries in microblogs. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 1251–1262.

[38] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. [n. d.]. Benchmarking Distributed Stream Data Processing Systems. ([n. d.]).

[39] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. 2011. Algorithms for graph similarity and subgraph matching. *Dept. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep* (2011).

[40] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.

[41] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. ACM, 591–600.

[42] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable multi-query optimization for SPARQL. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 666–677.

[43] Chuan Lei, Zhongfang Zhuang, Elke A Rundensteiner, and Mohamed Eltabakh. 2015. Shared execution of recurring workloads in MapReduce. *Proceedings of the VLDB Endowment* 8, 7 (2015), 714–725.

[44] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. 2016. StreamScope: Continuous Reliable Distributed Processing of Big Data Streams.. In *NSDI*, Vol. 16. 439–453.

[45] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM,

49–60.

[46] Mohamed F Mokbel, Xiaopeing Xiong, and Walid G Aref. 2004. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 623–634.

[47] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[48] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[49] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across multiple queries in MapReduce. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 494–505.

[50] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. 2008. Automatic Optimization of Parallel Dataflow Programs.. In *USENIX Annual Technical Conference*. 267–273.

[51] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications.. In *OSDI*, Vol. 10. 1–15.

[52] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-aware operator placement for stream-processing systems. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 49–49.

[53] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review* 45, 4 (2015), 421–434.

[54] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. 2014. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area.. In *NSDI*, Vol. 14. 275–288.

[55] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.

[56] Sangeetha Seshadri, Vibhore Kumar, and Brian F Cooper. 2006. Optimizing multiple queries in distributed data stream systems. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*. IEEE, 25–25.

[57] Sangeetha Seshadri, Vibhore Kumar, Brian F Cooper, and Ling Liu. 2007. Optimizing multiple distributed stream queries using hierarchical network partitions. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 1–10.

[58] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE, 996–1005.

[59] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[60] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 374–389.

[61] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries.. In *OSDI*. 435–450.

[62] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints.. In *NSDI*. 323–336.

[63] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment* 7, 3 (2013), 145–156.

[64] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. 2013. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 292–308.

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[66] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 423–438.

[67] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 533–544.