Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics

Benjamin Heintz
Department of Computer Science
University of Minnesota
heintz@cs.umn.edu

Abhishek Chandra
Department of Computer Science
University of Minnesota
chandra@cs.umn.edu

Ramesh K. Sitaraman
Department of Computer Science
UMass Amherst & Akamai Tech.
ramesh@cs.umass.edu

Abstract

Many applications must ingest rapid data streams and produce analytics results in near-real-time. It is increasingly common for inputs to such applications to originate from geographically distributed sources. The typical infrastructure for processing such geo-distributed streams follows a hub-and-spoke model, where several edge servers perform partial computation before forwarding results over a wide-area network (WAN) to a central location for final processing. Due to limited WAN bandwidth, it is not always possible to produce exact results. In such cases, applications must either sacrifice timeliness by allowing delayed—i.e., *stale*—results, or sacrifice accuracy by allowing some *error* in final results.

In this paper, we focus on *windowed grouped aggregation*, an important and widely used primitive in streaming analytics, and we study the tradeoff between staleness and error. We present *optimal offline algorithms* for minimizing staleness under an error constraint and for minimizing error under a staleness constraint. Using these offline algorithms as references, we present *practical online algorithms* for effectively trading off timeliness and accuracy under bandwidth limitations. Using a workload derived from an analytics service offered by a large commercial CDN, we demonstrate the effectiveness of our techniques through both trace-driven simulation as well as experiments on an Apache Storm-based implementation deployed on PlanetLab. Our experiments show that our proposed algorithms reduce staleness by 81.8% to 96.6%, and error by 83.4% to 99.1% compared to a practical random sampling/batching-based aggregation algorithm across a diverse set of aggregation functions.

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
 ISBN 978-1-4503-4525-5/16/10...$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987580

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems—Distributed Applications

Keywords Geo-distributed systems, stream processing, aggregation, approximation

1. Introduction

Stream computing has emerged as a critically important topic in recent years. Whether comprising sensor data from smart homes, user interaction logs from streaming video clients, or server logs from a content delivery network, rapid streams of data represent rich sources of meaningful and actionable information. The key challenge lies in extracting this information quickly, while it is still relevant. Modern streaming analytics systems therefore face the daunting task of ingesting massive amounts of data and performing computation in near-real-time. Several scalable systems have been proposed to address this challenge [2, 3, 6, 10, 12, 26, 31, 36].

Adding to the challenge is the fact that many interesting data streams are geographically distributed. For example, smart home sensor data and CDN log data originate from sources that are physically fixed at diverse locations all around the globe; such data are truly "born distributed" [33]. The distributed infrastructure of a typical geo-distributed analytics service such as Google Analytics or Akamai Media Analytics follows a hub-and-spoke model. In this architecture, numerous distributed data sources send their streams of data to nearby "edge" servers, which perform partial computation before forwarding results to a central location responsible for performing any remaining computation, storing results, and serving responses to analytics users' queries. While this central location is often a well-provisioned data center, resources are typically more limited at the edge locations. Perhaps most critically, the wide-area network connection between edges and the center may have highly constrained bandwidth. Geo-distributed analytics systems have received growing research attention in recent years [24, 30, 32–34]

A crucial question for a geo-distributed analytics system is how best to utilize resources at both the edges and the center in order to deliver timely results. In particular, a geo-distributed analytics system must determine how much computation to perform at the edges, and how much to leave for the center (i.e., *where* to compute), as well as *when* to send partial results from edges to the center.

In this paper, we examine these questions in the context of *windowed grouped aggregation*, a key primitive for streaming analytics, where data streams are logically subdivided into non-overlapping time windows within which records are grouped by common attributes, and summaries are computed for each group. This pattern is ubiquitous in both streaming and batch settings. For example, the SQL `Group By` construct allows concise expression of grouped aggregation, while grouped aggregation represents the fundamental abstraction underlying MapReduce. Although it may seem restrictive, a surprisingly broad range of applications fit into the grouped aggregation pattern [10].

Our prior work [22] examined questions of where to compute and when to communicate in the context of exact computation. It presented the design of algorithms for performing windowed grouped aggregation in order to optimize two key metrics of any geo-distributed streaming analytics service: WAN traffic, and staleness (the delay in getting the result for a time window), which are respectively measures of cost [4, 21] and performance. There, we assumed that (a) applications require exact results, and (b) resources—WAN bandwidth in particular—were sufficient to deliver exact results. In general, however, these assumptions do not always hold. For one, it is not always feasible to compute exact results with bounded staleness [32]. Further, many real-world applications can tolerate some staleness or inaccuracy in their final results, albeit with diverse preferences. For instance, a network administrator may need to be alerted to potential network overloads *quickly* (within a few seconds or minutes), even if there is some degree of error in the results describing network load. On the other hand, a Web analyst might have only a small tolerance for error (say, $<1\%$) in the application statistics (e.g., number of page hits), but be willing to wait for some time to obtain these results with the desired accuracy.

In this paper, we study the *staleness-error tradeoff*, recognizing that applications have diverse requirements: some may tolerate higher staleness in order to achieve lower error, and vice versa. We devise both theoretically optimal as well as practical algorithms to solve two complementary problems: minimize staleness under an error constraint, and minimize error under a staleness constraint. Our algorithms enable geo-distributed streaming analytics systems to support a diverse range of application requirements, whether WAN capacity is plentiful or highly constrained.

Research Contributions

- We study the tradeoff between staleness and error (measures of timeliness and accuracy, respectively) in a geo-distributed stream analytics setting.
- We present optimal offline algorithms that allow us to optimize staleness (resp., error) under an error (resp., staleness) constraint.
- Using these offline algorithms as references, we present practical online algorithms to efficiently trade off staleness and error. These practical algorithms are based on the key insight of representing grouped aggregation at the edge as a *two-level cache*. This formulation generalizes our caching-based framework for exact windowed grouped aggregation [22] by introducing cache *partitioning* policies to identify which partial results must be sent and which ones can be discarded.
- We demonstrate the practicality and efficacy of these algorithms through both trace-driven simulations and implementation in Apache Storm [3], deployed on a PlanetLab [1] testbed. Using workloads derived from traces of a popular web analytics service offered by Akamai [28], a large commercial content delivery network, our experiments show that our algorithms reduce staleness by 81.8% to 96.6%, and error by 83.4% to 99.1% compared to a practical random sampling/batching-based aggregation algorithm.
- We demonstrate that our techniques apply across a diverse set of aggregates, from distributive and algebraic aggregates [20] such as `Sum` and `Max` to holistic aggregates such as unique count (via sketch data structures such as `HyperLogLog` [18]).

2. Problem Formulation

2.1 Problem Statement

System Model

We consider the typical hub-and-spoke architecture of an analytics system with a center and multiple edges. Data streams are first sent from each source to a nearby edge. The edges collect and (optionally, partially) aggregate the data. This aggregated data can then be sent from the edges to the center where any remaining aggregation takes place. The final aggregated results are available at the center. Users of the analytics service query the center to visualize their analytics results. To perform grouped aggregation, each edge runs a local aggregation algorithm: it acts independently to decide when and how much to aggregate the incoming data. (Coordination between edges is an interesting topic [13] for future work, but is outside scope of the present paper.)

Windowed Grouped Aggregation over Data Streams

A *data stream* comprises *records* of the form (k, v) where k is the *key* and v is the *value*. Data records of a stream *arrive*

at the edge over time. Each key k can be multi-dimensional, with each dimension corresponding to a data attribute. A *group* is a set of records sharing the same key.

Customarily, time is divided into non-overlapping intervals, or *windows*, of user-specified length W.[1] *Windowed grouped aggregation* over a time window $[T, T+W)$ is then defined as follows from an input/output perspective. The input is the set of data records that arrive within the time window. The output is determined by first placing the data records into groups, where each group is a set of records with the same key. For each group $\{(k, v_i) : 1 \leq i \leq n\}$ corresponding to the n records in the time window that have key k, an aggregate value $V_k = v_1 \oplus v_2 \oplus \cdots \oplus v_n$ is computed, where $\oplus$ is an application-defined associative binary operator; e.g., `Sum`, `Max`, or `HyperLogLog` merge.[2]

To compute windowed grouped aggregation, the distributed infrastructure can perform aggregation at both the edge and the center. The records that arrive at the edge can be partially aggregated there, and the edge can maintain a set of partial aggregates, one for each distinct key k. The edge may transmit, or *flush* these aggregates to the center; we refer to these flushed records as *updates*. The center can further apply the aggregation operator $\oplus$ on incoming updates as needed in order to generate the final aggregate result. We assume that the computational overhead of the aggregation operator is a small constant compared to the network overhead of transmitting an update.

Approximate Windowed Grouped Aggregation

An aggregation algorithm runs on the edge and takes as input the sequence of arrivals of data records in a given time window $[T, T+W)$. The algorithm produces as output a sequence of updates that are sent to the center.

For each distinct key k with $n_k > 0$ arrivals in the time window, suppose that the i^{th} data record $(k, v_{i,k})$ arrives at time $a_{i,k}$, where $T \leq a_{i,k} < T+W$ and $1 \leq i \leq n_k$. For each such key k, the output of the aggregation algorithm is a sequence of m_k updates, where $0 \leq m_k \leq n_k$. The j^{th} update $(k, \hat{v}_{j,k})$ departs[3] for the center at time $d_{j,k}$ where $1 \leq j \leq m_k$. This update aggregates all values for key k that have arrived but have not yet been included in an update.

The final, possibly approximate, aggregated value for key k is given by $\hat{V}_k = \hat{v}_{1,k} \oplus \hat{v}_{2,k} \oplus \cdots \oplus \hat{v}_{m_k,k}$. Approximation arises in two cases. The first is when, for key k with $n_k > 0$ arrivals, no update is flushed; i.e., $m_k = 0$. This leads to an *omission error* for key k. The second is when at least one record arrives for key k after the final update is flushed; i.e., when $d_{m_k,k} < a_{n_k,k}$. This leads to a *residual error* for key k.

[1] These are often called *tumbling* windows in analytics terminology.

[2] More formally, $\oplus$ is any associative binary operator such that there exists a semigroup $(S, \oplus)$.

[3] Upon departure from the edge, an update is handed off to the network for transmission.

Optimization Metrics

Staleness is defined as the smallest time interval s such that the results of grouped aggregation for the time window $[T, T+W)$ are available at the center at time $T+W+s$. In other words, staleness quantifies the time elapsed from when the window closes to when the last update for that window reaches the center and is included in the final aggregate.

Over a window, we define the *per-key error* e_k for a key k to be the difference between the final aggregated value $\hat{V}_k$ and its true aggregated value V_k: $e_k = \text{error}(\hat{V}_k, V_k)$, where error is application-defined.[4] *Error* is then defined as the *maximum* error over all keys: $Error = \max_k e_k$. Such error arises when an algorithm flushes updates for some keys prior to receiving all arrivals for those keys, or when it omits flushes for some keys altogether.[5]

As we show in the next section, staleness and error are fundamentally opposing requirements. The goal of an aggregation algorithm is therefore either to *minimize staleness given an error constraint*, or to *minimize error given a staleness constraint*.

2.2 The Staleness-Error Tradeoff

Mechanics of the Tradeoff

To understand the mechanics behind the staleness-error tradeoff, it is useful to consider what causes staleness and error in the first place. Recall from Section 2.1 that staleness for a given time window is defined as the delay between the end of that window and the time at which the last update for that window reaches the center.[6] Error is caused by any updates that are not delivered to the center, either because the edge sent only partial aggregates, or because the edge omitted some aggregates altogether.

Intuitively, the two main causes of high staleness are either using too much network bandwidth (causing delays due to network congestion), or delaying transmissions until the end of the window. Thus, we can reduce staleness by *avoiding some updates altogether* (to avoid network congestion), and *sending updates earlier during the window*. Unfortunately, both of these options for reducing staleness lead directly to sources of error. First, if no update is ever sent for a given key, then the center never gets any aggregate value for this key, leading to an omission error for the key. Second, if the last update for a key is scheduled prior to the final arrival for that key, then the center will see only a partial aggregate, leading to a residual error for that key. Thus, we see that there is a fundamental tradeoff between achieving low staleness and low error.

[4] Our work applies to both absolute and relative error definitions.

[5] Our work also applies to approximate value types, as long as the error function appropriately incorporates this value-level approximation.

[6] Wide-area *latency* contributes to this delay, but this component of delay is a function of the underlying network infrastructure and is not something we can directly control through our algorithms. We therefore focus our attention on network delays due to wide-area *bandwidth* constraints.

Challenges in Optimizing Along the Tradeoff Curve

To understand the challenges of optimizing for either of these metrics while bounding the other, consider two alternate approaches to grouped aggregation: *streaming*, which immediately sends updates to the center without any aggregation at the edge; and *batching*, which aggregates all data during a time window at the edge and only flushes updates to the center at the end of the window. When bandwidth is sufficient, streaming can deliver extremely low staleness and high accuracy, as arrivals are flushed to the center without additional delay, and updates reach the center quickly. When bandwidth is constrained, however, it can lead to high staleness and error. This is because it fails to take advantage of edge resources to reduce the volume of traffic flowing across the wide-area network, leading to high congestion and unbounded network delays and in turn unbounded staleness.

Batching, on the other hand, leverages computation at edge resources to reduce network bandwidth consumption. When the end of the window arrives, a batching algorithm has the final values for every key on hand, and it can prioritize important updates in order to reach an error constraint quickly, or to reduce error rapidly until reaching a staleness constraint. On the other hand, batching introduces delays by deferring all flushes until the end of the window, leading to high staleness for any given error.

One alternate approach is a *random sampling* algorithm that prioritizes important transmissions after the end of the window, but sends a subset of aggregate values selected randomly during the window. This approach improves over batching by sending updates earlier during the window, thus reducing staleness. It also improves over streaming under bandwidth constraints by reducing the network traffic. As we show in Sections 3 and 4, however, this approach can still yield high error and staleness due to lack of prioritization among different updates during the window.

To satisfy our optimization goals, we need more principled approaches.

3. Offline Algorithms

We now consider the two complementary optimization problems of minimizing staleness (resp., error) under an error (resp., staleness) constraint. Before we present practical online algorithms to solve these problems, we consider optimal *offline* algorithms. Although these offline algorithms cannot be applied directly in practice, they serve both as baselines for evaluating the effectiveness of our online algorithms, and also as design inspiration, helping us to identify heuristics that practical online algorithms might employ in order to emulate optimal algorithms.

In this section, we provide proofs only for the main theorems and omit proofs for other lemmas due to space constraints. For details, see our technical report [23].

3.1 Minimizing Staleness (Error-Bound)

The first optimization problem we consider is minimizing staleness under an error constraint (error $\leq E$), where E is an application-specified error tolerance value.

In this case, the goal is to flush only as many updates as is strictly required, and to flush each of these updates as early as possible such that the error constraint is satisfied. In short, an offline optimal algorithm achieves this by flushing an update for each key as soon as the aggregate value for that key falls within the error constraint.

Throughout this section, consider the time window of length W beginning at time T, let $\oplus$ denote our binary aggregation function, and let n denote the number of unique keys arriving during the current window. Define the *prefix aggregate* $V_k(t)$ for key k at time t to be the aggregate value of all arrivals for key k during the current window prior to time t. We define the prefix aggregate $V_k(t)$ to have a logical zero value prior to the first arrival for key k. Let $\text{error}(\hat{x},x)$ denote the error of the aggregate value $\hat{x}$ with respect to the true value x. Further, define *prefix error* $e_k(t)$ for key k at time t to be the error of the prefix aggregate for key k with respect to the true final aggregate: $e_k(t) = \text{error}(V_k(t), V_k(T+W))$, for $T \leq t < T+W$. We refer to the prefix error of key k at the beginning of the window—$e_k(T)$—as the *initial prefix error* of key k.

Definition 1 (Eager Prefix Error)**.** *Given an error constraint E, the Eager Prefix Error (EPE) algorithm flushes each key k at the first time t such that $e_k(t) \leq E$. If $e_k(T) \leq E$, then EPE avoids flushing key k altogether.*

Theorem 1. *The EPE algorithm is optimal.*

Proof. Such an algorithm satisfies the error constraint by construction, and because flushes are issued as early as possible for each key, and only if strictly necessary, there cannot exist another schedule that achieves lower staleness. □

3.2 Minimizing Error (Staleness-Bound)

Next, we consider the optimization problem of minimizing error under a staleness constraint (staleness $\leq S$), where S is an application-specified staleness tolerance (or deadline).

To minimize error under a staleness constraint, we abstract the wide-area network as a sequence of contiguous *slots*, each representing the ability to transmit a single update. The duration of a single slot in seconds is then $\frac{1}{b}$, where b represents the available network bandwidth in updates per second.[7] If S denotes the staleness constraint in seconds, then the final slot ends S seconds after the end of the window. Figure 1 illustrates this model for the time window $[T, T+W)$.

Note that there is no reason to flush a key more than once during any given window, as any updates prior to the last could simply be aggregated into the final update before

[7] In general, bandwidth need not be constant.

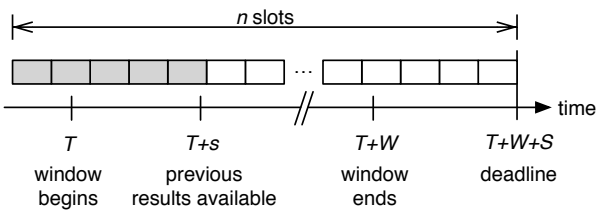


Figure 1. We view the network as a sequence of contiguous slots. Shaded slots are unavailable to the current window due to the previous window's staleness.

it is flushed. Given this fact, we can focus on scheduling flushes for the n unique keys that arrive during the window by assigning each into one of n slots.

Note that flushes from the previous window occupy the network for the first s seconds of the current window, where $s \leq S$ is the staleness of the previous window. These slots are therefore unavailable to the current window, and assigning a key to such a slot has the effect of sending no value for that key. In general, the first slot of the current window may begin prior to this time, or in fact prior to the beginning of the current window.

To understand how we assign keys to slots, we must first introduce the notion of *potential error*. The potential error[8] $E_k(t)$ for key k at time t is defined to be the error that the center would see for key k if key k were assigned to a slot beginning at time t. Recall that, for $t < T + s$, slots are unavailable, as the network is still in use transmitting updates from the previous window. Assigning a key to such a slot therefore has the effect of sending no value for that key. Similarly, if a key is assigned to a slot prior to that key's first arrival, there is no value to send, so making such an assignment is equivalent to sending no value for that key. We refer to either of these cases as *omitting* a key. Overall then, potential error is given by

$$E_k(t) = \begin{cases} e_k(T) & \text{if } t < T + s, \\ e_k(t) & \text{otherwise.} \end{cases}$$

We assume a *monotonicity property*: the potential error $E_k(t)$ of a key k at any time t is no larger than its potential error $E_k(t')$ at a prior time $t' < t$ within the window, i.e., $E_k(t) \leq E_k(t')$.[9]

Definition 2 (Smallest Potential Error First)**.** *The Smallest Potential Error First (SPEF) algorithm iterates over the n slots beginning with the earliest, assigning to each slot a key with the* smallest potential error *that has not yet been assigned to a slot.*

[8] Throughout this section, we discuss error with respect to the window $[T, T+W)$.

[9] Note that we can easily extend our algorithms to relax this monotonicity property. See our technical report [23] for details.

Lemma 1. *A schedule D which flushes keys i and j at times t_i and t_j such that $t_i \leq t_j$ and $E_i(t_i) \leq E_j(t_i)$ (i.e., in SPEF order) cannot have higher error than a schedule D' that swaps these keys.*

Lemma 2. *A schedule D that flushes only $m < n$ keys can be transformed into another schedule D' that flushes n keys without increasing staleness or error.*

Theorem 2. *The SPEF algorithm is optimal.*

Proof. The SPEF algorithm satisfies the staleness constraint due to our definition of slots. To see why it minimizes error, let D_{SPEF} denote a sequence of flushes in smaller-potential-error-first order. Let D_{opt} denote an optimal sequence of flushes sharing the longest common prefix with D_{SPEF}. Then there exists an m such that D_{OPT} and D_{SPEF} differ for the first time at index m. If D_{SPEF} and D_{opt} are not already identical, then $m < n$ where n is the number of unique keys during the window, and hence the length of D_{SPEF}.

Assume that D_{opt} also has length n. (If not, then transform according to Lemma 2.) There must then exist an index $m < p \leq n$ such that $E_m(t_m) \geq E_p(t_m)$. In other words, in slot m, D_{opt} emits a key that does not have the smallest potential error. By Lemma 1, we can transform D_{opt} into D'_{opt} by swapping keys m and p without increasing error or staleness. In particular, if $p = \text{argmin}_{m<i\leq n} E_i(t_m)$, then this swap yields D'_{opt} that is optimal and identical to D_{SPEF} up to index $m+1$. This exposes a contradiction: it could not have been true that D_{opt} had the longest prefix in common with D_{SPEF}. Therefore m cannot be less than n: there must exist some optimal departure sequence identical to D_{SPEF}. □

An Alternate Optimal Algorithm

Recall that assigning a key to a slot prior to the first arrival for that key, or before the network is available, is equivalent to sending no value for that key; i.e., *omitting* that key. By studying how the SPEF algorithm schedules such omissions, we can derive an alternative optimal algorithm that is more amenable to emulation in an online setting.

Definition 3 (SPEF with Early Omissions)**.** *Let m be the number of keys that the SPEF algorithm omits. Then the SPEF with Early Omissions (SPEF-EO) algorithm first omits the m keys with the smallest initial prefix errors, then assigns the remaining $n - m$ keys in SPEF order.*

Lemma 3. *A key omitted by the SPEF algorithm has, at the time it is omitted, the smallest initial prefix error among all not-yet-assigned keys.*

Theorem 3. *The SPEF-EO algorithm is optimal.*

Proof. By Lemma 3, the final key omitted by the SPEF algorithm contributes the greatest error of all prior omissions, and this error is no less than the m^{th}-smallest initial prefix error. Error therefore cannot be reduced by flushing (i.e., not

omitting) any of the m keys with the smallest initial prefix errors.

The remaining $n-m$ slots occur no earlier than the $n-m$ slots carrying non-zero updates in the original algorithm. By monotonicity, assigning the remaining $n-m$ keys to these slots therefore cannot increase error. □

3.3 High-Level Lessons

These offline optimal algorithms, although not applicable in practical online settings, leave us with several high-level lessons. First, they flush each key at most once, thereby avoiding wasting scarce network resources. Second, they make the best possible use of network resources. In the error-bound case, EPE achieves this by sending only keys that have already satisfied the error constraint. For the staleness-bound case, with SPEF and SPEF-EO, this means using each unit of network capacity (i.e., each slot) to send the most up-to-date value for the key with the minimum potential error, and for SPEF-EO, sending only those keys with the largest initial prefix errors.

4. Online Algorithms

We are now prepared to consider practical online algorithms for achieving near-optimal staleness-error tradeoffs. The offline optimal algorithms from Section 3 serve as useful baselines for comparison, and they also provide models to emulate. We first present our practical online algorithms and then present a prototype implementation on Apache Storm.

4.1 The Two-Level Cache Abstraction

Exact Computation

Our online algorithms for approximate windowed grouped aggregation generalize those for exact computation [22], where we view the edge as a *cache* of aggregates and emulate offline optimal algorithms through cache *sizing* and *eviction* policies. When a record arrives at the edge, it is inserted into the cache, and its value is merged via aggregation with any existing aggregate sharing the same key. The sizing policy, by dictating how many aggregates may reside in the cache, determines *when* aggregates are evicted. For exact computation, an ideal sizing policy allows aggregates to remain in the cache until they have reached their final value, while avoiding holding them so long as to lead to high staleness. The eviction policy determines *which* key to evict, and an ideal policy for exact computation selects keys with no future arrivals. Upon eviction, keys are enqueued to be transmitted over the WAN, which we assume services this queue in FIFO order.

Approximate Computation

For approximate windowed grouped aggregation, we continue to view the edge as a cache, but we now partition it into *primary* and *secondary* caches. The reason for this distinction is that, when approximation is allowed, it is no longer necessary to flush updates for all keys. An online algorithm must determine not just *when* to flush each key, but also *which* keys to flush. The distinction between the two caches serves to answer the latter question: *updates are flushed only from the primary cache*. It is the role of the cache *partitioning* policy to define the boundary between the primary and secondary cache, and the main difference between our error-bound and staleness-bound online algorithms lies in the partitioning policy. As we discuss throughout this section, our error-bound algorithm defines this boundary based on the *values* of items in the cache, while the staleness-bound algorithm uses a dynamic *sizing* policy to determine the size of the primary cache.

In addition, the primary cache logically serves as the outgoing network queue: updates are flushed from this cache when network capacity is available. Unlike FIFO queuing, this ensures that our online algorithms make the most effective possible use of network resources. In particular, it ensures that flushed updates always reflect the most up-to-date aggregate value for each key. Additionally, it allows us to use our eviction policies to choose the most valuable key to occupy the network at any time.

4.2 Error-Bound Algorithms

Cache Partitioning Policy

Our online error-bound algorithm uses a value-based cache partitioning policy, which defines the boundary between primary and secondary caches in terms of aggregate values. It emulates the offline optimal EPE algorithm (Section 3.1) that only flushes keys whose prefix error is within the error bound. Specifically, new arrivals are first added to the secondary cache, and they are *promoted* into the primary cache only when their aggregate grows to exceed the error constraint. More rigorously, let F_k denote the total aggregate value flushed for key k so far during the current window (logically zero if no update has been flushed), and let V_k denote the aggregate value currently maintained in the cache for key k. Then the *accumulated error* for key k is defined as $\text{error}(F_k, F_k \oplus V_k)$; i.e., the error between the value that the center currently knows and the value it would see if key k were flushed.[10] Key k is moved from the secondary cache to primary cache when its accumulated error exceeds E. Given this policy, and the fact that updates are only flushed from the primary cache, we are guaranteed to flush only keys that strictly must be flushed in order to satisfy the error constraint; i.e., this approach avoids false positives.

After the end of the window, our online algorithm flushes the entire contents of the primary cache, as all of its constituent keys exhibit sufficient accumulated error to violate the error constraint if not flushed.[11]

[10] Note we can compute this value online.

[11] The order of these flushes is unimportant, as all of them are required to bring error below E.

Cache Eviction Policy

Prior to the end of the window, some prediction is involved: When the network pulls an update from the primary cache, the eviction policy must identify a key that has reached an aggregate value within E of its final value. We explore several alternatives [23] and find that well known cache replacement policies—in particular least-recently-used (LRU)—work well. Intuitively, LRU assumes that the key with the least recent arrival is also the least likely to receive future arrivals, and hence closest to its final value. Therefore LRU evicts the key with the least recent arrival.

4.3 Staleness-Bound Algorithms

An online algorithm that aims to minimize error under a staleness constraint faces slightly different challenges. In particular, we can no longer define the boundary between primary and secondary caches by value, as we do not know a priori what this value should be. Instead our staleness-bound online algorithm uses a dynamic *sizing-based* cache partitioning policy to emulate the offline optimal SPEF-EO algorithm (Section 3.2). To understand this approach, recall from Definition 3 that the SPEF-EO algorithm flushes only the keys with the largest initial prefix errors. Given that the role of the primary cache is to contain the keys that must be flushed, we emulate this behavior by 1) dynamically ranking cached keys by their (estimated) initial prefix error, and then 2) defining the primary cache at time t to comprise the top $\sigma(t)$ keys in this order. In practice, this is challenging as we must predict both initial prefix errors as well as an appropriate sizing function $\sigma(t)$.

Initial Prefix Error Prediction

During the window, there are many ways to predict initial prefix errors. One straightforward approach is to use accumulated error (Section 4.2) as a proxy for initial prefix error. We call this the *Acc* policy for short. Intuitively, this policy assumes that the keys that have accumulated the most error so far also have the largest initial prefix errors, and therefore ranking keys by accumulated error is a good approximation for ranking them by initial prefix error. We find this policy performs better than more sophisticated policies such as those that try to predict initial prefix error explicitly [23].

Cache Size Prediction

Predicting the appropriate primary cache size function $\sigma(t)$ raises several additional challenges. To begin, consider how to define this function in an ideal world where we have a perfect eviction policy and a perfect prediction of initial prefix error. At time t, the primary cache size $\sigma(t)$ represents the number of keys that are *present* in the primary cache. These $\sigma(t)$ keys will need to be flushed prior to the staleness bound, as will any not-yet-arrived keys with sufficiently high initial prefix error. Let $f(t)$ denote the number of future arrivals of these large-prefix-error keys. Then, if the total number of network slots remaining until the staleness constraint is given by $B(t)$, an ideal primary cache size function $\sigma(t)$ satisfies $B(t) = \sigma(t) + f(t)$.

In practice, we have neither a perfect eviction policy, nor a perfect prediction of initial prefix error, and determining $f(t)$ is a nontrivial prediction on its own. We have explored a host of alternative approaches [23], and among these, we find that a policy based on the previous window's arrival sequence works well. This policy, referred to as *PrevWindow*, assumes that arrivals in the current window will resemble those in the previous window, and it therefore uses the previous window's arrival sequence to compute $f(t)$. Using an exponentially weighted moving average of the available WAN bandwidth to predict $B(t)$, this $f(t)$ is then used to compute the appropriate primary cache size function $\sigma(t)$.

Cache Eviction Policy

During the window, we use the sizing-based cache partitioning policy to delineate the boundary between primary and secondary caches. When network capacity is available, it remains the role of the cache eviction policy to determine which key should be evicted from the primary cache. As in Section 4.2, we again find that simple well known policies such as LRU perform well.

Upon reaching the end of the window, there is no need for prediction since all final aggregate values are known. At this point, keys are evicted from the union of the primary and secondary caches in descending order by their accumulated error until reaching the staleness bound.

4.4 Implementation

We implement our algorithms in Apache Storm [3], extending our prototype for exact computation [22], which uses a distinct Storm cluster at each edge, as well as one at the center, in order to distribute the aggregation workload. Figure 2 shows the overall architecture. Here we briefly discuss each component in the order of data flow from edge to center.

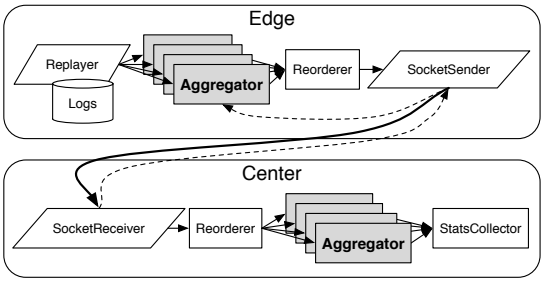


Figure 2. Aggregation is distributed over Apache Storm clusters at each edge as well as at the center.

Edge. Data enters our prototype at the edge through the `Replayer` spout, which replays timestamped logs from a file. Each line is parsed using a query-specific parsing function to produce a `(timestamp, key, value)` triple. Our

implementation leverages Twitter's Algebird[12] library to support a broad set of value types and associated aggregations. The `Replayer` emits records according to their timestamp; i.e., event time and processing time [7] are equivalent at the `Replayer`. The `Replayer` emits records downstream, and also periodically emits punctuation messages to indicate that no messages with earlier timestamps will be sent in the future.

The next step in the dataflow is the `Aggregator`, for which one or more tasks run at each cluster. The `Aggregator` defines window boundaries in terms of record timestamps, and maintains the two-level cache from Section 4, with each task aggregating a hash-partitioned subset of the key space. We implement the two-part cache using efficient priority queues, allowing us to generalize over a range of cache partitioning and eviction policies in order to implement our online algorithms.

The `Aggregator` tasks send their output to a single instance of the `Reorderer` bolt, which is responsible for delaying records as necessary in order to maintain punctuation semantics. Data then flows into the `SocketSender` bolt, which transmits (partial) aggregates to the center using TCP sockets.

Center. At the center, data flows in the reverse order. First, the `SocketReceiver` spout receives partial aggregates and punctuations and emits them downstream into a `Reorderer`, where the streams from multiple edges are synchronized. From there, records flow into the central `Aggregator`, each task of which is responsible for performing the final aggregation over a hash-partitioned subset of the key space. Upon completing aggregation for a window, these central `Aggregator` tasks emit summary metrics including error and staleness, and these metrics are summarized by the final `StatsCollector` bolt. Staleness is computed relative to the wall-clock (i.e., processing) time at which the window closes. Clocks are synchronized using NTP.[13]

Our online algorithms treat the two-part cache as the outgoing network queue, but Storm does not implement flow control at the bolt level, and the Storm programming model only allows bolts to *push* tuples downstream. To resolve this apparent mismatch, we track queuing delays within both the `SocketSender` and the `SocketReceiver`. These delays are communicated upstream, and the edge `Aggregators` use them as congestion signals, applying an additive-increase / multiplicative-decrease (AIMD) approach to dynamically adjust the rate at which they push records downstream.

5. Evaluation

In this section, we evaluate our online algorithms using both a trace-driven simulation as well as experiments using our Storm-based implementation on PlanetLab [1] and local testbeds.

5.1 Dataset and Queries

Our simulations and experiments are driven by an anonymized workload trace obtained from a real-world analytics service[14] by Akamai, a large commercial content delivery network. This analytics service is used by content providers to track important metrics about who has downloaded content, where these clients were located, what performance they experienced, how many downloads completed successfully, etc. The data source is a software called Download Manager, which is installed on mobile devices, laptops, and desktops of millions of users around the world, and used to download software updates, security patches, music, games, and other content. The Download Managers installed on users' devices around the world send information about the downloads to the widely-deployed Akamai edge servers using "beacons".[15] Each download results in one or more beacons being sent to an edge server, and these beacons contain information about the time the download was initiated, url, content size, number of bytes downloaded, user's ip, user's network, user's geography, server's network and server's geography. We use the anonymized beacon logs from Akamai's download analytics service for the month of December, 2010. *Note that, for confidentiality reasons, we normalize derived values from the data set such as time durations and error values.*

In our evaluation, we focus on windowed grouped aggregation for a query that groups its input records by content provider id, client country code, and url. Because this last dimension—url—can take on hundreds of thousands of distinct values, we call this a "large" query. Our previous work on exact windowed grouped aggregation [22] also considers "smaller" queries; we focus here on the largest queries as they are especially challenging under bandwidth constraints.

We consider three diverse aggregations: `Sum`, `Max`, and `HyperLogLog` [18]. Each of these aggregations uses the `large` query key. The `Sum` aggregation computes the total number of bytes successfully downloaded for each key, while the `Max` aggregation computes the largest successful download size for each key. The `HyperLogLog` aggregation is used to approximate the number of unique client IP addresses for each key. This demonstrates an important point: our techniques apply to a broad range of aggregations, from distributive and algebraic [20] aggregates such as `Sum`, `Max`, or `Average` to holistic aggregates such as unique count via `HyperLogLog`.[16]

[12] `https://github.com/twitter/algebird`

[13] Clock synchronization errors are small relative to the staleness ranges we explore in our experiments.

[14] `http://www.akamai.com/dl/feature_sheets/Akamai_Download_Analytics.pdf`

[15] A beacon is simply an http `GET` issued by the Download Manager for a small GIF containing the reported values in its url query string.

[16] We adopt the common approach [15] of approximating holistic aggregates (e.g., unique count, Top-K, heavy hitters) using sketch data structures (e.g., HyperLogLog, CountMinSketch [14])

5.2 Baseline Algorithms

We compare the following aggregation algorithms.

Streaming: A streaming algorithm performs no aggregation at the edge, and instead simply flushes each key immediately upon arrival; that is, it streams arrivals directly on to the center. An error-bound variant continues streaming arrivals until the error constraint E is satisfied, while a staleness-bound variant continues streaming until reaching the staleness limit S.

Batching: A batching algorithm aggregates arrivals until the end of the window without sending any updates. At the end of the window at time $T+W$, an error-bound variant flushes all keys for which omitting an update would exceed the error bound; i.e., all keys with initial prefix error greater than E. A staleness-bound variant begins flushing keys at the end of the window, and does so in descending order of initial prefix error, stopping upon reaching the staleness limit S.

Batching with Random Early Updates (Random): The Random algorithm effectively combines batching with streaming using random sampling. Concretely, the Random algorithm aggregates arrivals at the edge as batching does, but it sends updates for random keys during the window whenever network capacity is available. When the end of the window arrives, it flushes keys as batching does, in decreasing order by their impact on error. This algorithm is a useful baseline to show that the choice of which key to flush during the window is critical.

Optimal: To provide an optimal baseline for evaluating our online algorithms, we implement the EPE and SPEF-EO algorithms from Section 3 for the error-bound and staleness-bound optimizations, respectively.

Caching: These refer to our caching-based online algorithms presented in Section 4. For the error-bound case, we use the emulated EPE algorithm, consisting of the two-part caching approach with an LRU primary cache eviction policy, as discussed in Section 4.2. For the staleness-bound case, we employ the emulated SPEF-EO algorithm discussed in Section 4.3, consisting of LRU eviction, Accumulated Error-based estimation, and PrevWindow cache sizing policies.

5.3 Simulation Results

Before presenting experimental results on our PlanetLab deployment, we use simulations to compare our online algorithms to a number of baseline algorithms. Simulation allows us to more rapidly explore a large design space, to compare against an offline optimal algorithm, and to select a practical baseline approach to use in our experiments later.

We implement a simple discrete event simulator in Python. Our simulation is deliberately simplified, as the focus here is not on understanding performance in absolute terms (we rely on the experiments in Section 5.4 for that) but rather to compare the tradeoffs between different algorithms. Note that error for a schedule of flushes can be computed directly from its definition; it is only the staleness that we simulate. To compute staleness, we model the wide-area network as a queue with deterministic service times based on a constant network bandwidth. Arrival times to this queue correspond to the time at which updates are flushed. An aggregation algorithm is then simulated by scheduling the updates based on the algorithm's schedule. All simulation results present median values (staleness or error) over 25 consecutive time windows from the Akamai trace, spanning multiple days worth of workload.

5.3.1 Minimizing Staleness (Error-Bound)

We first compare our caching-based online algorithm against the baseline algorithms for the error-bound optimization algorithm which has the goal of minimizing staleness for a given error constraint. Figure 3 shows staleness (normalized relative to the window length) for a range of error bounds E (normalized relative to the largest possible error for our trace) for our online algorithm, as well as for our three baseline algorithms and an optimal offline algorithm. It is clear that streaming is infeasible, as staleness far exceeds the window length (shown as the dotted horizontal line). By performing aggregation at the edge, batching significantly improves upon streaming, but it still yields high staleness because it delays all flushes until the end of the window. Random improves only slightly over batching, showing the limitation of our random sampling approach. Our caching-based online algorithm, on the other hand, chooses which keys to flush during the window in a principled manner, and yields staleness much closer to an offline optimal algorithm as a result.

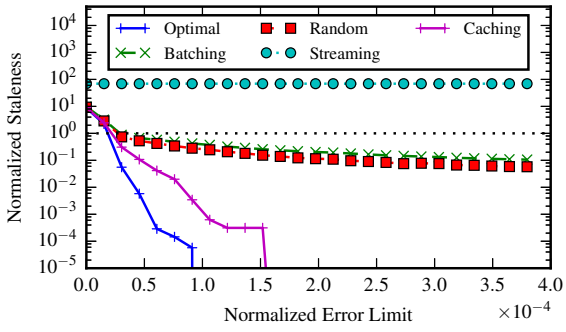


Figure 3. Normalized staleness for `Sum` aggregation with various error bounds. Note the logarithmic y-scale.

5.3.2 Minimizing Error (Staleness-Bound)

Figure 4 demonstrates the effectiveness of our approach compared to the other baselines for the case where we

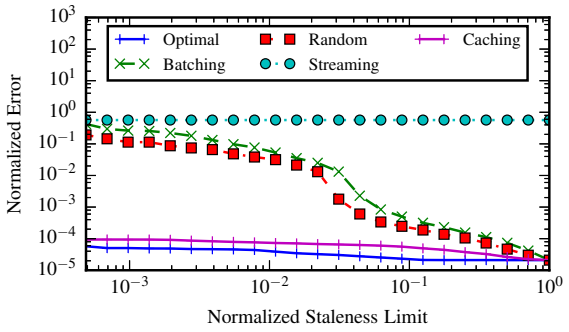


Figure 4. Normalized error for `Sum` aggregation with various staleness bounds. Note the logarithmic axis scales.

are minimizing the error given a staleness limit. The figure shows the normalized error values obtained for different staleness limits. We again see that streaming is impractical, as its fails to effectively use the limited WAN capacity to transmit the most important updates. Batching yields a significant improvement over streaming, as long as the tolerance for staleness is sufficiently high. Batching with random early updates represents a further improvement, though again, the benefit of these early updates is not dramatic. Our caching-based online algorithm, on the other hand, by making principled decisions about which keys to flush during the window, yields near-optimal error across a broad range of staleness limits.

Note that all algorithms except streaming converge to the same error as the staleness bound reaches one window length. The reason is that, as the staleness bound approaches the window length, there are fewer and fewer opportunities to flush updates during the window, and these algorithms converge toward pure batching.

5.4 Experimental Results

To demonstrate how our techniques perform in a real-world setting, we now present results from an experimental evaluation conducted on PlanetLab [1] using our Storm-based implementation. Throughout our experiments, we use two combinations of PlanetLab nodes at the University of Oregon as well as local nodes at the University of Minnesota. The Oregon site includes three nodes, each with 4 GB of physical memory and four CPU cores. The Minnesota site includes up to eight nodes, each with 12 GB of physical memory and eight CPU cores. WAN bandwidth from Minnesota to Oregon averages 12.5 Mbps based on `iperf3` measurements. For all of our experiments, we use the anonymized Akamai trace as input.

Staleness and error are measured at the center, and we treat the staleness constraint as a hard deadline: when the deadline is reached, we compute the error based on records that have arrived at the center up to that point. In general, there may be arrivals after the deadline due to varying WAN delays, but we disregard these late arrivals as they violate the staleness constraint.

Based on our results from Section 5.3, we use Random as a practical baseline algorithm for comparison, since it outperforms both batching and streaming.

5.4.1 Geo-Distributed Aggregation

In order to demonstrate our techniques in a real geo-distributed setting, we begin with experiments using Minnesota nodes as the edge and Oregon nodes as the center. Note that we use local Minnesota nodes rather than PlanetLab nodes as the edge because PlanetLab imposes restrictive daily limits on the amount of data sent from each node, rendering repeated experiments with PlanetLab edge nodes impractical.[17]

Figure 5(a) and Figure 5(b) show staleness and error for error-constrained and staleness-constrained algorithms, respectively. Both staleness and error are normalized relative to the Random baseline for ease of interpretation. We consider the median staleness or error during a relatively stable portion of the workload, and plot the mean over five runs. Error bars show 95% confidence intervals [17].

Figure 5 demonstrates that our Caching algorithms significantly outperform a practical random sampling/batching-based baseline in a real-world geo-distributed deployment. Staleness is reduced by 81.8% to 96.6% under an error constraint, while error is reduced by 83.4% to 99.1% under a staleness constraint. Note that the reduction is largest for `Max` aggregation, as many keys attain their final value well in advance of the end of the window, allowing greater opportunity for early evictions.

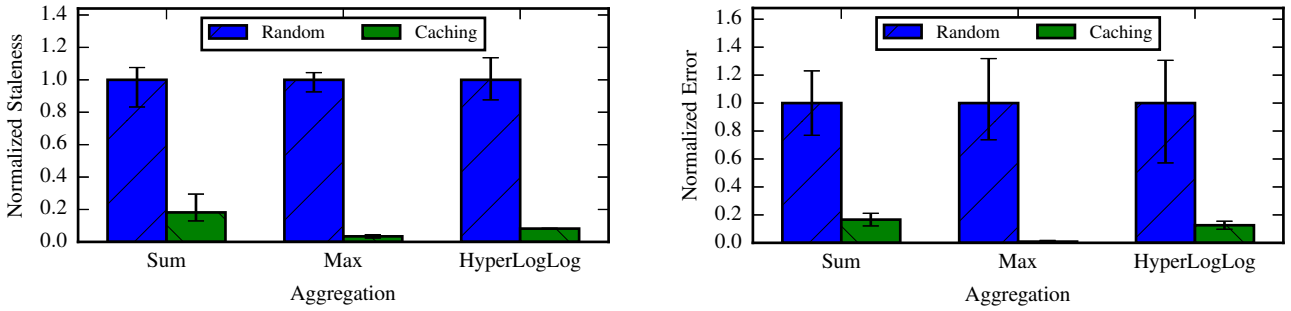
These results demonstrate that our techniques perform well in a real geo-distributed deployment over real data, and they do so for a diverse set of aggregations.

5.4.2 Dynamic Workload and WAN Bandwidth

To dig deeper, we assess how our algorithms perform under dynamic workload and WAN bandwidth using an alternative deployment with eight total Minnesota nodes: four as the edge and four as the center. We use `tc` to emulate constrained WAN bandwidth between the edge and center clusters. By changing the emulated WAN bandwidth at runtime, we can study how our algorithms adapt to dynamic WAN bandwidth. Further, our anonymized Akamai trace exhibits a significant shift in workload early in the trace, providing a natural opportunity to study performance under workload variations.

Figure 6 shows how our staleness-bound and error-bound algorithms behave under these workload and bandwidth changes for the `Sum` aggregation. Figure 6(a) plots staleness normalized relative to the window length, while Figure 6(b) plots error normalized relative to the largest possible error.

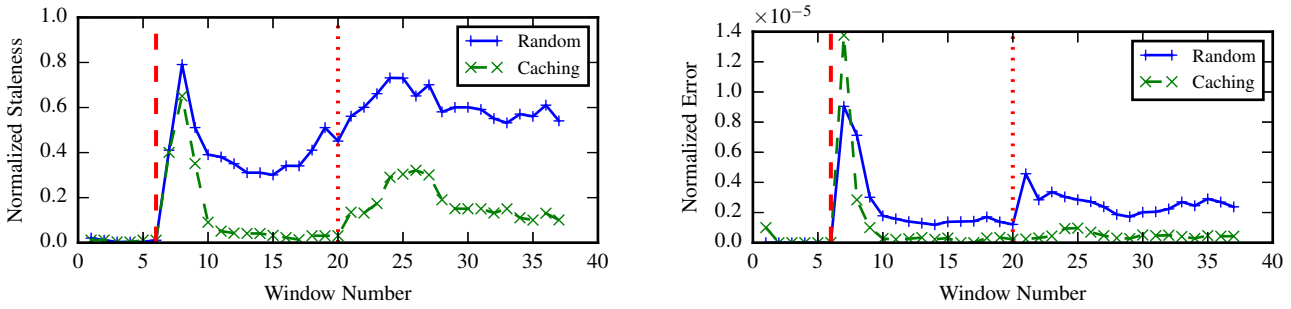
[17] For example, each one of the five repetitions of the `HyperLogLog` experiments was enough to exhaust the daily bandwidth limit.



(a) Staleness under an error constraint.

(b) Error under a staleness constraint.

Figure 5. Comparison of Random and Caching algorithms for various aggregation functions on a PlanetLab testbed.



(a) Staleness under an error constraint.

(b) Error under a staleness constraint.

Figure 6. Comparison of Random and Caching algorithms for `Sum` aggregation under dynamic workload and WAN bandwidth on a local testbed. Arrival rate increases during window 6 (dashed vertical line), and available bandwidth decreases during window 20 (dotted vertical line).

For both figures, we run the experiment five times, and plot the median staleness or error for each window.

We see that both Random and Caching algorithms require some time to adapt to the workload change that occurs during window 6. Once stabilized, Caching yields significantly lower staleness and error than Random. During window 20, we reduce WAN bandwidth by 62.5%. While staleness and error increase for both algorithms, our Caching algorithms continue to significantly outperform the Random baseline.

5.4.3 Various Error and Staleness Constraints

Using the eight-node Minnesota deployment, we further study the performance of our algorithms under various error and staleness constraints. Figure 7 shows staleness (normalized relative to the window length) and error (normalized relative to the largest possible error) for the `Sum` aggregation. As in Figure 5, we plot 95% confidence intervals over five runs.

As we saw from our simulation results in Section 5.3, we see that higher error (resp., staleness) constraints lead to lower staleness (resp., error), demonstrating the tradeoff between staleness and error in a real deployment. Our Caching algorithms significantly outperform Random across a range of error and staleness constraints,

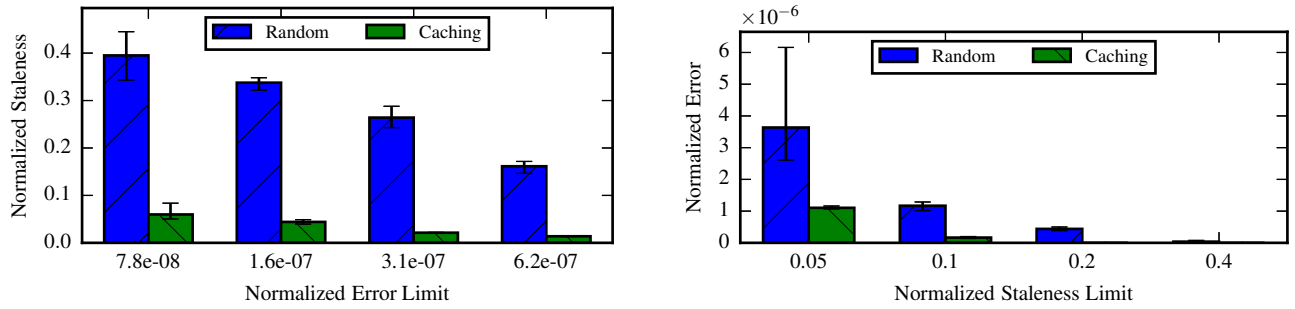
Overall, our experiments demonstrate that our Caching algorithms apply across a diverse set of aggregations, and that they outperform the Random baseline in real geo-distributed deployments, under dynamic workloads and bandwidth conditions, and across a range of error and staleness constraints.

6. Related Work

Numerous streaming systems [2, 6, 11, 12, 26, 31, 36] have been proposed in recent years. These systems provide many useful ideas for new analytics systems to build upon, but they do not fully explore the challenges described in this paper, in particular how to strike a near-optimal balance between timeliness and accuracy.

Google recently proposed the Dataflow model [7] as a unified abstraction for computing over both bounded and unbounded datasets. Our work fits within this model, but we focus in particular on aggregation over tumbling windows on unbounded data sets.

Wide-area computing has received increased research attention in recent years, due in part to the widening gap be-



(a) Staleness under various error constraints.

(b) Error under various staleness constraints.

Figure 7. Comparison of Random and Caching algorithms for `Sum` aggregation with various error and staleness constraints on a local testbed.

tween data processing and communication costs. Much of this attention has been paid to batch computing [30, 33, 34]. Relatively little work on streaming computation has focused on wide-area deployments, or associated questions such as where to place computation. Pietzuch et al. [29] optimize operator placement in geo-distributed settings to balance between system-level bandwidth usage and latency. Hwang et al. [25] rely on replication across the wide area in order to achieve fault tolerance and reduce straggler effects.

JetStream [32] considers wide-area streaming computation, and like our work, addresses the tension between timeliness and accuracy. Unlike our work, however, it focuses at a higher level on the appropriate abstractions for navigating this tradeoff. Meanwhile BlinkDB [5] provides mechanisms to trade accuracy and response time, though it does not focus on processing streaming data. Our focus, on the other hand, is on concrete algorithms to approach an optimal tradeoff between timeliness and accuracy in streaming settings.

Das et al. [16] consider tradeoffs between throughput and latency in Spark Streaming, but they focus on exact computation and consider only a uniform batching interval for the entire stream, while we consider approximate computation and address scheduling on a per-key basis.

Aggregation is a key operator in analytics, and grouped aggregation is supported by many data-parallel programming models [10, 20, 35]. Larson et al. [27] explore the benefits of performing partial aggregation prior to a join operation, much as we do prior to network transmission. While they also recognize similarities to caching, they consider only a simple fixed-size cache, whereas our approach uses a novel two-part cache to determine both whether and when to transfer partial aggregates. Amur et al. [8] study grouped aggregation, focusing on the design and implementation of efficient data structures for batch and streaming computation, though they do not consider staleness, a key performance metric in our work.

Our work bears some resemblance to anytime algorithms (e.g., [9, 37]), which improve their solutions over time and can be interrupted at any point to deliver the current best solution. While our staleness-bound algorithm is essentially an anytime algorithm, terminating when the staleness limit is reached, our error-bound algorithm terminates only when the error constraint is satisfied.

Our techniques complement other approaches for approximation in stream computing. For example, our work applies to approximate value types such as the Count-min Sketch [14] or HyperLogLog [18], while recent work in distributed function monitoring [13, 19] could serve as a starting point for coordination between multiple edges.

7. Conclusion

In this paper, we considered the problem of streaming analytics in a geo-distributed environment. Due to WAN bandwidth constraints, applications must often sacrifice either timeliness by allowing *stale* results, or accuracy by allowing some *error* in final results. In this paper, we focused on windowed grouped aggregation, an important and widely used primitive in streaming analytics, and studied the tradeoff between the key metrics of staleness and error. We presented optimal offline algorithms for minimizing staleness under an error constraint and for minimizing error under a staleness constraint. Using these offline algorithms as references, we presented practical online algorithms for effectively trading off timeliness and accuracy in the face of bandwidth limitations. Using a workload derived from a web analytics service offered by a large commercial CDN, we demonstrated the effectiveness of our techniques through both trace-driven simulation as well as experiments using an Apache Storm-based prototype implementation deployed on PlanetLab. Our results showed that our proposed algorithms outperform practical baseline algorithms for a range of error and staleness bounds, for a variety of aggregation functions, and under varied network bandwidth and workload conditions.

Acknowledgments

The authors would like to acknowledge NSF Grant CNS-1413998, and an IBM Faculty Award, which supported this research.

References

[1] PlanetLab. `http://planet-lab.org/`, 2015.

[2] Apache Flink: Scalable Batch and Stream Data Processing. `http://flink.apache.org/`, 2016.

[3] Apache Storm. `http://storm.apache.org/`, 2016.

[4] M. Adler, R. K. Sitaraman, and H. Venkataramani. Algorithms for optimizing the bandwidth cost of content delivery. *Computer Networks*, 55(18):4007–4020, Dec. 2011. ISSN 1389-1286.

[5] S. Agarwal et al. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proc. of EuroSys*, pages 29–42, 2013.

[6] T. Akidau et al. MillWheel: Fault-tolerant stream processing at internet scale. *Proc. of VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[7] T. Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. of the VLDB Endowment*, 8:1792–1803, 2015.

[8] H. Amur, W. Richter, D. G. Andersen, M. Kaminsky, K. Schwan, A. Balachandran, and E. Zawadzki. Memory-efficient groupby-aggregate using compressed buffer trees. In *Proc. of SoCC*, pages 18:1–18:16, 2013.

[9] M. Boddy. Anytime problem solving using dynamic programming. In *Proc. of AAAI*, pages 738–743, 1991.

[10] O. Boykin, S. Ritchie, I. O'Connel, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. In *Proc. of VLDB*, volume 7, pages 1441–1451, 2014.

[11] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of CIDR*, 2003.

[12] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proc. of SIGMOD*, pages 1087–1098, 2016.

[13] G. Cormode. Continuous distributed monitoring: A short survey. In *Proc. of AlMoDEP*, pages 1–10, 2011.

[14] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005. ISSN 0196-6774.

[15] G. Cormode, T. Johnson, F. Korn, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Holistic UDAFs at streaming speeds. In *Proc. of SIGMOD*, pages 35–46, 2004.

[16] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proc. of SoCC*, pages 16:1–16:13, 2014.

[17] B. Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185, 1987.

[18] P. Flajolet, É. Fusy, O. Gandouet, et al. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. In *Proc. of AOFA*, 2007.

[19] N. Giatrakos, A. Deligiannakis, and M. Garofalakis. Scalable approximate query tracking over highly distributed data streams. In *Proc. of SIGMOD*, pages 1497–1512, 2016.

[20] J. Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[21] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, Dec. 2008. ISSN 0146-4833.

[22] B. Heintz, A. Chandra, and R. K. Sitaraman. Optimizing grouped aggregation in geo-distributed streaming analytics. In *Proc. of HPDC*, pages 133–144, 2015.

[23] B. Heintz, A. Chandra, and R. K. Sitaraman. Trading timeliness and accuracy in geo-distributed streaming analytics. Technical Report 16-003, Department of Computer Science, University of Minnesota, 2016.

[24] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *Proc. of SoCC*, pages 111–124, 2015.

[25] J.-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and highly-available stream processing over wide area networks. In *Proc. of ICDE*, pages 804–813, 2008.

[26] S. Kulkarni et al. Twitter heron: Stream processing at scale. In *Proc. of SIGMOD*, pages 239–250, 2015.

[27] P.-A. Larson. Data reduction by partial preaggregation. In *Proc. of ICDE*, pages 706–715, 2002.

[28] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010.

[29] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. of ICDE*, 2006.

[30] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *Proc. of SIGCOMM*, pages 421–434, 2015.

[31] Z. Qian et al. TimeStream: reliable stream computation in the cloud. In *Proc. of EuroSys*, pages 1–14, 2013.

[32] 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 *Proc. of NSDI*, pages 275–288, 2014.

[33] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. WANalytics: Analytics for a geo-distributed data-intensive world. In *Proc. of CIDR*, January 2015.

[34] 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 *Proc. of NSDI*, pages 323–336, May 2015.

[35] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. of SOSP*, pages 247–260, 2009.

[36] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of SOSP*, pages 423–438, 2013.

[37] S. Zilberstein. Operational rationality through compilation of anytime algorithms. Technical report, 1993.